# Handling Categorical Values

**A categorical variable takes only a limited number of values.**

1. Consider a survey that asks how often you eat breakfast and provides four options: "Never", "Rarely", "Most days", or "Every day". In this case, the data is categorical, because responses fall into a fixed set of categories.

2. If people responded to a survey about which what brand of car they owned, the responses would fall into categories like "Honda", "Toyota", and "Ford". In this case, the data is also categorical.

You will get an error if you try to plug these variables into most machine learning models in Python without preprocessing them first. In this tutorial, we'll compare three approaches that you can use to prepare your categorical data

**We will see three approaches to handle categorical data:**

1. Drop Categorical variables
2. Ordinal Encoding
3. One-hot encoding

## 1. Drop Categorical Variables

The easiest approach to dealing with categorical variables is to simply remove them from the dataset. This approach will only work well if the columns did not contain useful information.

## 2. Ordinal Encoding

Ordinal encoding assigns each unique value to a different integer.

This approach assumes an ordering of the categories: "Never" (0) < "Rarely" (1) < "Most days" (2) < "Every day" (3).

This assumption makes sense in this example, because there is an indisputable ranking to the categories. Not all categorical variables have a clear ordering in the values, but we refer to those that do as ordinal variables.

For tree-based models (like decision trees and random forests), you can expect ordinal encoding to work well with ordinal variables.

## 3. One-Hot Encoding

One-hot encoding creates new columns indicating the presence (or absence) of each possible value in the original data.

In the original dataset, "Color" is a categorical variable with three categories: "Red", "Yellow", and "Green". The corresponding one-hot encoding contains one column for each possible value, and one row for each row in the original dataset. Wherever the original value was "Red", we put a 1 in the "Red" column; if the original value was "Yellow", we put a 1 in the "Yellow" column, and so on.

In contrast to ordinal encoding, one-hot encoding does not assume an ordering of the categories. Thus, you can expect this approach to work particularly well if there is no clear ordering in the categorical data (e.g., "Red" is neither more nor less than "Yellow"). We refer to categorical variables without an intrinsic ranking as nominal variables.

One-hot encoding generally does not perform well if the categorical variable takes on a large number of values (i.e., you generally won't use it for variables taking more than 15 different values).

In [8]:
```
# Loading Melbourne Housing dataset

import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
data = pd.read_csv('datasets/melb_data/melb_data.csv')

# Separate target from predictors
y = data.Price
X = data.drop(['Price'], axis=1)

# Divide data into training and validation subsets
X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size=0.8
                                                                random_state=0)

# Drop columns with missing values (simplest approach)
cols_with_missing = [col for col in X_train_full.columns if X_train_full[col].isnull
X_train_full.drop(cols_with_missing, axis=1, inplace=True)
X_valid_full.drop(cols_with_missing, axis=1, inplace=True)

# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitra
low_cardinality_cols = [cname for cname in X_train_full.columns if X_train_full[cnam
                        X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dty

# Keep selected columns only
my_cols = low_cardinality_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

In [9]:
```
X_train.head()
```

Out[9]:

| | Type | Method | Regionname | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | u | S | Southern Metropolitan | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 |
| **6524** | h | SA | Western Metropolitan | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 |
| **8413** | h | S | Western Metropolitan | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 |
| **2919** | u | SP | Northern Metropolitan | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 |

| | Type | Method | Regionname | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize |
|---|---|---|---|---|---|---|---|---|---|
| **6043** | h | S | Western Metropolitan | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 |

In [38]:
```
X_train['Type'].unique()
```

Out[38]:
```
array(['u', 'h', 't'], dtype=object)
```

### getting a list of all the categorical columns ---> Method 1

In [10]:
```
# getting a list of all the categorical columns ---> Method 1

s = X_train.dtypes
type(s)
```

Out[10]:
```
pandas.core.series.Series
```

In [11]:
```
X_train.dtypes
```

Out[11]:
```
Type              object
Method            object
Regionname        object
Rooms              int64
Distance         float64
Postcode         float64
Bedroom2         float64
Bathroom         float64
Landsize         float64
Lattitude        float64
Longtitude       float64
Propertycount    float64
dtype: object
```

In [20]:
```
s = X_train.dtypes == 'object'
print(s)
print(type(s))
```

```
Type              True
Method            True
Regionname        True
Rooms            False
Distance         False
Postcode         False
Bedroom2         False
Bathroom         False
Landsize         False
Lattitude        False
Longtitude       False
Propertycount    False
dtype: bool
<class 'pandas.core.series.Series'>
```

In [14]:
```
s[s]
```

Out[14]:
```
Type          True
Method        True
```

```
Regionname    True
dtype: bool
```

In [15]:
```
s[s].index
```

Out[15]:
```
Index(['Type', 'Method', 'Regionname'], dtype='object')
```

In [16]:
```
type(s[s].index)
```

Out[16]:
```
pandas.core.indexes.base.Index
```

In [17]:
```
categorical_cols = list(s[s].index)
categorical_cols
```

Out[17]:
```
['Type', 'Method', 'Regionname']
```

### getting categorical columns list ---> Method 2

In [30]:
```
# getting categorical columns list ---> Method 2

cat_cols = X_train.columns[X_train.dtypes == 'object']
```

In [31]:
```
cat_cols
print(cat_cols)
type(cat_cols)
```

Out[31]:
```
Index(['Type', 'Method', 'Regionname'], dtype='object')
pandas.core.indexes.base.Index
```

In [32]:
```
cat_cols = list(cat_cols)
```

In [33]:
```
cat_cols
```

Out[33]:
```
['Type', 'Method', 'Regionname']
```

# Appraoch-1: Drop Categorical Variables

In [34]:
```
# way 1

dropped_X_train1 = X_train.drop(cat_cols, axis=1)
dropped_X_train1.head()
```

Out[34]:

| | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize | Lattitude | Longtitude | Properi |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 | -37.85984 | 144.9867 | |
| **6524** | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 | -37.85800 | 144.9005 | |
| **8413** | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 | -37.79880 | 144.8220 | |
| **2919** | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 | -37.70830 | 144.9158 | |
| **6043** | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 | -37.76230 | 144.8272 | |

In [35]:
```
# way 2: We drop the object columns with the select_dtypes() method.

dropped_X_train2 = X_train.select_dtypes(exclude=['object'])
dropped_X_train2.head()
```

Out[35]:

| | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize | Lattitude | Longtitude | Properl |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 | -37.85984 | 144.9867 | |
| **6524** | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 | -37.85800 | 144.9005 | |
| **8413** | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 | -37.79880 | 144.8220 | |
| **2919** | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 | -37.70830 | 144.9158 | |
| **6043** | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 | -37.76230 | 144.8272 | |

# Approach-2: Ordinal Encoding

Scikit-learn has a OrdinalEncoder class that can be used to get ordinal encodings. We loop over the categorical variables and apply the ordinal encoder separately to each column.

In [36]:
```
# making a copy to avoid changing original data

X_train_copy = X_train.copy()
```

In [37]:
```
# code

from sklearn.preprocessing import OrdinalEncoder

# creating instance
encoder = OrdinalEncoder()


# applying ordinal encoder to each column with categorical datatype
X_train_copy[cat_cols] = encoder.fit_transform(X_train_copy[cat_cols])

X_train_copy.head()
```

Out[37]:

| | Type | Method | Regionname | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | 2.0 | 1.0 | 5.0 | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 |
| **6524** | 0.0 | 2.0 | 6.0 | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 |
| **8413** | 0.0 | 1.0 | 6.0 | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 |
| **2919** | 2.0 | 3.0 | 2.0 | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 |
| **6043** | 0.0 | 1.0 | 6.0 | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 |

In [39]:
```
X_train_copy['Type'].unique()
```

Out[39]:
```
array([2., 0., 1.])
```

We can see above that our categories have been converted to numbers.

In the code cell above, for each column, we randomly assign each unique value to a different integer.

This is a common approach that is simpler than providing custom labels; however, we can expect an additional boost in performance if we provide better-informed labels for all ordinal variables.

## Approach-3: One-Hot Encoding

We use the OneHotEncoder class from scikit-learn to get one-hot encodings. There are a number of parameters that can be used to customize its behavior.

1. We set handle_unknown='ignore' to avoid errors when the validation data contains classes that aren't represented in the training data, and
2. setting sparse=False ensures that the encoded columns are returned as a numpy array (instead of a sparse matrix).

To use the encoder, we supply only the categorical columns that we want to be one-hot encoded.

For instance, to encode the training data, we supply X_train[object_cols].

(object_cols in the code cell below is a list of the column names with categorical data, and so X_train[object_cols] contains all of the categorical data in the training set.)

In [40]:
```
# code
from sklearn.preprocessing import OneHotEncoder

# creating copy to avoid changing original dataframe
X_train_copy2 = X_train.copy()

# creating instance
OHencoder = OneHotEncoder(handle_unknown='ignore', sparse=False)

# applying one hot encoder to each column with categorical data
encoded_X_train = pd.DataFrame(
    OHencoder.fit_transform(X_train_copy2[cat_cols])
)

encoded_X_train.head()
```

Out[40]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **1** | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **2** | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **3** | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **4** | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |

In [41]:
```
```

```
# One-hot encoding removed index; put it back

encoded_X_train.index = X_train.index
```

In [42]:
```
encoded_X_train.head()
```

Out[42]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **12167** | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **6524** | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **8413** | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **2919** | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **6043** | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |

In [43]:
```
# Remove categorical columns (will replace with one-hot encoding)

rm_cat_col_X_train_copy2 = X_train_copy2.drop(cat_cols, axis=1)
rm_cat_col_X_train_copy2.head()
```

Out[43]:

| | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize | Lattitude | Longtitude | Properi |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 | -37.85984 | 144.9867 | |
| **6524** | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 | -37.85800 | 144.9005 | |
| **8413** | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 | -37.79880 | 144.8220 | |
| **2919** | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 | -37.70830 | 144.9158 | |
| **6043** | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 | -37.76230 | 144.8272 | |

In [44]:
```
# Add one-hot encoded columns to numerical features

OH_X_train = pd.concat([rm_cat_col_X_train_copy2, encoded_X_train], axis=1)
OH_X_train.head()
```

Out[44]:

| | Rooms | Distance | Postcode | Bedroom2 | Bathroom | Landsize | Lattitude | Longtitude | Properi |
|---|---|---|---|---|---|---|---|---|---|
| **12167** | 1 | 5.0 | 3182.0 | 1.0 | 1.0 | 0.0 | -37.85984 | 144.9867 | |
| **6524** | 2 | 8.0 | 3016.0 | 2.0 | 2.0 | 193.0 | -37.85800 | 144.9005 | |
| **8413** | 3 | 12.6 | 3020.0 | 3.0 | 1.0 | 555.0 | -37.79880 | 144.8220 | |
| **2919** | 3 | 13.0 | 3046.0 | 3.0 | 1.0 | 265.0 | -37.70830 | 144.9158 | |
| **6043** | 3 | 13.3 | 3020.0 | 3.0 | 1.0 | 673.0 | -37.76230 | 144.8272 | |

5 rows × 25 columns

# Which approach is best?

In this dataset, dropping the categorical columns performed worst, since it had the highest MAE

score.

As for the other two approaches, since the returned MAE scores are so close in value, there doesn't appear to be any meaningful benefit to one over the other.

In general, one-hot encoding (Approach 3) will typically perform best, and dropping the categorical columns (Approach 1) typically performs worst, but it varies on a case-by-case basis.

In [ ]:

The world is filled with categorical data. You will be a much more effective data scientist if you know how to use this common data type!

## Problem with Ordinal Encoder

Unique values in 'Condition2' column in training data: ['Norm' 'PosA' 'Feedr' 'PosN' 'Artery' 'RRAe']

Unique values in 'Condition2' column in validation data: ['Norm' 'RRAn' 'RRNn' 'Artery' 'Feedr' 'PosN']

In [ ]:

If you now write code to:

fit an ordinal encoder to the training data, and then use it to transform both the training and validation data, you'll get an error.

Can you see why this is the case? (You'll need to use the above output to answer this question.)

In [ ]:

Fitting an ordinal encoder to a column in the training data creates a corresponding integer-valued label for each unique value that appears in the training data.

In the case that the validation data contains values that don't also appear in the training data, the encoder will throw an error, because these values won't have an integer assigned to them.

Notice that the 'Condition2' column in the validation data contains the values 'RRAn' and 'RRNn', but these don't appear in the training data -- thus, if we try to use an ordinal encoder with scikit-learn, the code will throw an error.

In [ ]:

This is a common problem that you'll encounter with real-world data, and there are many approaches to fixing this issue. For instance, you can write a custom ordinal encoder to deal with new categories. The simplest approach, however, is to drop the problematic categorical columns.

Run the code cell below to save the problematic columns to a Python list bad_label_cols. Likewise, columns that can be safely ordinal encoded are stored in good_label_cols.

In [ ]:
```
# code
# Categorical columns in the training data
object_cols = [col for col in X_train.columns if X_train[col].dtype == "object"]

# Columns that can be safely ordinal encoded
good_label_cols = [col for col in object_cols if
                   set(X_valid[col]).issubset(set(X_train[col]))]

# Problematic columns that will be dropped from the dataset
bad_label_cols = list(set(object_cols)-set(good_label_cols))

print('Categorical columns that will be ordinal encoded:', good_label_cols)
print('\nCategorical columns that will be dropped from the dataset:', bad_label_cols
```

code should be written to drop the categorical columns in bad_label_cols from the dataset.

You should ordinal encode the categorical columns in good_label_cols.

In [ ]:
```
# Drop categorical columns that will not be encoded

label_X_train = X_train.drop(bad_label_cols, axis=1)
label_X_valid = X_valid.drop(bad_label_cols, axis=1)
```

In [ ]:
```
# full code should be

from sklearn.preprocessing import OrdinalEncoder

# Drop categorical columns that will not be encoded
label_X_train = X_train.drop(bad_label_cols, axis=1)
label_X_valid = X_valid.drop(bad_label_cols, axis=1)

# Apply ordinal encoder
encoder = OrdinalEncoder() # Your code here

label_X_train[good_label_cols] = encoder.fit_transform(label_X_train[good_label_cols
label_X_valid[good_label_cols] = encoder.transform(label_X_valid[good_label_cols])

# Check your answer
step_2.b.check()
```

# Cardinality

[('Street', 2), ('Utilities', 2), ('CentralAir', 2), ('LandSlope', 3), ('PavedDrive', 3), ('LotShape', 4), ('LandContour', 4), ('ExterQual', 4), ('KitchenQual', 4), ('MSZoning', 5), ('LotConfig', 5), ('BldgType', 5), ('ExterCond', 5), ('HeatingQC', 5), ('Condition2', 6), ('RoofStyle', 6), ('Foundation', 6), ('Heating', 6), ('Functional', 6), ('SaleCondition', 6), ('RoofMatl', 7), ('HouseStyle', 8), ('Condition1', 9), ('SaleType', 9), ('Exterior1st', 15), ('Exterior2nd', 16), ('Neighborhood', 25)]

The output above shows, for each column with categorical data, the number of unique values in the column. For instance, the 'Street' column in the training data has two unique values: 'Grvl' and 'Pave', corresponding to a gravel road and a paved road, respectively.

We refer to the number of unique entries of a categorical variable as the cardinality of that categorical variable. For instance, the 'Street' variable has cardinality 2.

**Also, we know that columns needed to one-hot encode any variable = cardinality of that categorical variable.**

For large datasets with many rows, one-hot encoding can greatly expand the size of the dataset.

For this reason, we typically will only one-hot encode columns with relatively low cardinality. Then, high cardinality columns can either be dropped from the dataset, or we can use ordinal encoding.

Instead of encoding all of the categorical variables in the dataset, you'll only create a one-hot encoding for columns with cardinality less than 10.

Run the code cell below without changes to set low_cardinality_cols to a Python list containing the columns that will be one-hot encoded. Likewise, high_cardinality_cols contains a list of categorical columns that will be dropped from the dataset.

In [ ]:
```
# code

# Columns that will be one-hot encoded
low_cardinality_cols = [col for col in object_cols if X_train[col].nunique() < 10]

# Columns that will be dropped from the dataset
high_cardinality_cols = list(set(object_cols)-set(low_cardinality_cols))

print('Categorical columns that will be one-hot encoded:', low_cardinality_cols)
print('\nCategorical columns that will be dropped from the dataset:', high_cardinali
```

## One-Hot Encoding Review in one block of code

Use the next code cell to one-hot encode the data in X_train and X_valid.

Set the preprocessed DataFrames to OH_X_train and OH_X_valid, respectively.

1. The full list of categorical columns in the dataset can be found in the Python list object_cols.
2. You should only one-hot encode the categorical columns in low_cardinality_cols. All other categorical columns should be dropped from the dataset.

In [ ]:
```
# my_code

from sklearn.preprocessing import OneHotEncoder

# Use as many lines of code as you need!

OHencoder = OneHotEncoder(sparse=False, handle_unknown='ignore')

OH_X = pd.DataFrame(
    OHencoder.fit_transform(X_train[low_cardinality_cols])
)

OH_valid = pd.DataFrame(
    OHencoder.transform(X_valid[low_cardinality_cols])
)

# setting the index
OH_X.index = X_train.index
OH_valid.index = X_valid.index

# drop the hot encoded columns
numeric_X_train = X_train.drop(object_cols, axis=1)
numeric_X_valid = X_valid.drop(object_cols, axis=1)
```

```
# joining the OH df with original df
OH_X_train = pd.concat([numeric_X_train, OH_X], axis=1)
OH_X_valid = pd.concat([numeric_X_valid, OH_valid], axis=1)
```

In [ ]:

## Author: Piyush Kumar

Github

In [ ]: