Introduction to Data Pre-Processing

Let's first review the CRISP-DM framework:

- Business Understanding
- Data Understanding
- Data Preparation
- Modeling
- Evaluation
- Deployment

After data preparation is done, we will have a clean dataset. But before we apply machine learning algorithms on the dataset, we will still need to do some data pre-processing.

In this notebook, we introduce three types of data pre-processing:

- Encoding categorical variables
- Splitting dataset to training and testing dataset.
- Scaling and standardizing data

Table of Contents

Before proceeding with the *data exploration* section of this Notebook, we first have our standard notebook setup code.

In [1]:
```
# Set up Notebook

%matplotlib inline

# Standard imports
import pandas as pd
import numpy as np
import seaborn as sns

# We do this to ignore several specific warnings
import warnings
warnings.filterwarnings("ignore")
```

Back to TOC

Introduction to Scikit-Learn

The scikit-learn module, `sklearn` (https://scikit-learn.org/stable/), is a powerful, yet simple to use machine learning library written in the Python programming language. It features various classification, regression and clustering algorithms as well as data-preprocessing algorithms. We will use data pre-processing features in scikit-learn in this notebook.

Back to TOC

Categorical Variable Encoding

Almost all practical datasets will contain categorical variables. These variables are normally stored as text values. Some examples include Gender("Male" or "Female"), Size (“Small”, “Medium” or “Large”), or geographic designations (State or Country). Some machine learning algorithms can support categorical values without further manipulation but there are many more algorithms that do not. Therefore, we will need to turn these text categorical attributes into numerical values for further processing.

There are many ways to approach this problem. In this notebook, we will use pandas and scikit-learn modules to transform the categorical data into suitable numeric values.

Categorical features can take several forms. For example, a categorical feature can be categorized into nominal and ordinal features (note that while other classes are also possible, they are beyond the scope of this course).

Nominal feature: A nominal feature is either in a category or it isn't, and there are no relationships between the different categories. For example, the gender category is nominal since there is no numerical relation or ordering among the possible values, male and female.
Ordinal feature: An ordinal feature is a categorical feature where the possible values have an intrinsic relationship. For example, if we encode the results of a race as first, second, and third, these values have a relationship, in that first comes before second and second comes before third.

The process to convert categorical features to numerical values is generally known as encoding, and the scikit-learn library provides several different encodings in the preprocessing module.

To begin with, we first create a fictitious dataset shirt_order which contains the categorical features of Gender, Size and Color.

In [2]:
```
shirt_order = pd.DataFrame({'Name':['Alex', 'Ben', 'Cam', 'Dave', 'Eli', 'Frank',
                            'Gender':['F', 'M', 'M', 'M', 'F',  'M', 'F', 'M', 'F'
                            'Size':['Small', 'Large', 'Medium', 'Small', 'Medium',
                            'Color':['Blue', 'Yellow', 'Red', 'Red', 'Yellow', 'Re
                            })
shirt_order
```

Out[2]:

	Name	Gender	Size	Color
0	Alex	F	Small	Blue
1	Ben	M	Large	Yellow
2	Cam	M	Medium	Red
3	Dave	M	Small	Red
4	Eli	F	Medium	Yellow
5	Frank	M	Large	Red
6	Grace	F	Large	Blue
7	Henry	M	Large	Yellow
8	Iris	F	Small	Yellow
9	Jack	M	Small	Blue

Label Encoding

The simplest approach is to encode categorical values with a technique called Label Encoding", which allows you to convert each value in a column to a number. Scikit-Learn has LabelEncoder which supports Label Encoding. In the following Code cell, we create a new column Gender_cat to hold encoded Gender. Gender 'F' is encoded as 0 and 'M' as 1.

In [3]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
shirt_order['Gender_cat'] = le.fit_transform(shirt_order.Gender)
shirt_order
```

Out[3]:

	Name	Gender	Size	Color	Gender_cat
0	Alex	F	Small	Blue	0
1	Ben	M	Large	Yellow	1
2	Cam	M	Medium	Red	1
3	Dave	M	Small	Red	1
4	Eli	F	Medium	Yellow	0
5	Frank	M	Large	Red	1
6	Grace	F	Large	Blue	0
7	Henry	M	Large	Yellow	1
8	Iris	F	Small	Yellow	0
9	Jack	M	Small	Blue	1

Ordinal Encoding

LabelEncoder finds the unique values present in a column and map the values in range [0, n-1], n bening the number of unique values in the column. The values are mapped in alphabetical order. Thus, in the previous case, 'F' is mapped to 0 and 'M' is mapped to 1.

If we use same approach to encode the Size column, the mapping will be:

Large: 0
Medium: 1
Small: 2

This mapping is not ideal since Size is an ordinal categorical feature. The three categories, Small, Medium and Large, have an order associated with them. We would like to have this mapping instead:

Small: 0
Medium: 1
Large: 2

There are multiple ways to achieve this. One of the simplest ways is to use a pandas Series map() function as shown below. First, we will need to find all unique values in the column, then define mapping dictionary, then create new column with mapped numeric values.

In [4]:
```
#First find all unique values in Size
```

Out[4]:
```
array(['Small', 'Large', 'Medium'], dtype=object)
```

In [5]:
```
#Define mapping dictionary
mapping_dict = {'Small':0, 'Medium':1, 'Large':2}
#Encode Size column
shirt_order['Size_cat'] = shirt_order.Size.map(mapping_dict)
```

Out[5]:

	Name	Gender	Size	Color	Gender_cat	Size_cat
0	Alex	F	Small	Blue	0	0
1	Ben	M	Large	Yellow	1	2
2	Cam	M	Medium	Red	1	1
3	Dave	M	Small	Red	1	0
4	Eli	F	Medium	Yellow	0	1
5	Frank	M	Large	Red	1	2
6	Grace	F	Large	Blue	0	2
7	Henry	M	Large	Yellow	1	2
8	Iris	F	Small	Yellow	0	0
9	Jack	M	Small	Blue	1	0

One Hot Encoding

Label Encoding is straightforward but it has a disadvantage in that the numeric values can be “misinterpreted” by the algorithms. For example, the value of 0 is obviously less than the value of 1, but does that really correspond to the data set in real life? Consider Color column in our shirt_order dataset. If we use label encoding, Blue is mapped to 0 and Yellow is mapped to 2, but Blue is not supposed to be "smaller" than Yellow. Ordinal encoding doesn't help in this case for the same reason.

A common alternative approach is called One Hot Encoding. The basic strategy is to convert each category value into a new column and assigns a 1 or 0 (True/False) value to the column. This has the benefit of not weighting a value improperly but does have the downside of adding more columns to the data set.

Again, there are multiple ways to do One Hot Encoding. We will introduce how to do it with pandas using the `get_dummies` function. This function is named this way because it creates dummy variables with values 0 or 1. We encode Color in the following Code cell. Three extra columns are created, one for each unique values in Color: Color_Blue, Color_Red and Color_Yellow. Depending on the value of Color, only one out of the three dummy columns has value 1.

Since `get_dummies` will replace original categorical column with dummy columns, we first duplicate Color to keep original values. We pass `prefix=["Color"]` to `get_dummies` to define dummy column names to prefix with 'Color'. Without this argument, the dummy columns will be named with values in the categorical feature. In this case, the names would be 'Blue', 'Red' and 'Yellow'.

In [6]:
```
#duplicate Color column to keep original values
shirt_order['Color_cat'] = shirt_order.Color
#convert Color_cat to dummy variables.
shirt_order_onehot = pd.get_dummies(shirt_order, columns=["Color_cat"], prefix=["C
```

Out[6]:

	Name	Gender	Size	Color	Gender_cat	Size_cat	Color_Blue	Color_Red	Color_Yellow
0	Alex	F	Small	Blue	0	0	1	0	0
1	Ben	M	Large	Yellow	1	2	0	0	1
2	Cam	M	Medium	Red	1	1	0	1	0
3	Dave	M	Small	Red	1	0	0	1	0
4	Eli	F	Medium	Yellow	0	1	0	0	1
5	Frank	M	Large	Red	1	2	0	1	0
6	Grace	F	Large	Blue	0	2	1	0	0
7	Henry	M	Large	Yellow	1	2	0	0	1
8	Iris	F	Small	Yellow	0	0	0	0	1
9	Jack	M	Small	Blue	1	0	1	0	0

Back to TOC

Dataset Splitting

Before we can apply a supervised machine learning algorithm to the data of interest, we must divide the data into training and testing data sets. The *training* data are used to generate the supervised model, while the *testing* data are used to quantify the quality of the generated model. In the scikit-learn library, we can do this by using the `train_test_split` method in the `model_selection` module.

The only tuning parameter at this point is the `test_size` parameter, which we have set to 0.4 via the `test_size` argument. This means that 40% of our data will be reserved for testing and 60% will be used to generate the model. By changing this value, we can explore how different algorithms perform with more or less training data. One last parameter this method takes is the `random_state` parameter, which initializes the random sequence used to determine the split into the testing and training data. By using the same value, we ensure reproducibility. Varying this parameter will generate different testing and training data, even with the same value for the `test_size` parameter.

We first load the Iris dataset as a supervised learning dataset, which has 150 rows and 5 columns. We will encode the Species column and use it as a label. The rest of the columns are data (we will discuss data and label in more detail in the next lesson). Then we split data and label to training and testing, with test_size=0.4. Training data and label have 90 rows and testing data and label will have 60 rows. `data` is split into `d_train` and `d_test`, `label` is split into `l_train` and `l_test`.

In [7]:
```
# Load the Iris Data
iris = pd.read_csv("iris.csv")
```
Out[7]:
```
(150, 5)
```

In [8]:
```
```
Out[8]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [9]:
```
#create new column to hold encoded species
iris['species_cat'] = LabelEncoder().fit_transform(iris.species)
```
Out[9]:

	sepal_length	sepal_width	petal_length	petal_width	species	species_cat
119	6.0	2.2	5.0	1.5	virginica	2
10	5.4	3.7	1.5	0.2	setosa	0
28	5.2	3.4	1.4	0.2	setosa	0
123	6.3	2.7	4.9	1.8	virginica	2
99	5.7	2.8	4.1	1.3	versicolor	1

In [10]:
```
#Define data and label
data = iris[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']]
label = iris['species_cat']
```

Out[10]:

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

In [11]:
```
from sklearn.model_selection import train_test_split

# Split data into training and testing
# Note that we have both 'data' and 'label'
d_train, d_test, l_train, l_test = train_test_split(data, label, test_size=0.4, ra
```

Out[11]:
```
((90, 4), (60, 4))
```

Student Exercise

In the preceding cells, we used the scikit-learn library to split the dataset. Make the following code changes in those Code cells and execute the notebook again to answer the associated question.

1. Change the first `test-size` split from 0.4 to 0.25. What is the size of training data now?

Back to TOC

Data Scaling

Many machine learning estimators in the scikit-learn library are sensitive to variations in the spread of features within a data set. For example, if all features but one span similar ranges (e.g., zero through one) and one feature spans a much larger range (e.g., zero through one hundred), an algorithm might focus on the one feature with a larger spread, even if this produces a sub-optimal result. To prevent this, we generally scale the features to improve the performance of a given scikit-learn estimator.

Data scaling in scikit-learn can take several forms, we will introduce two of them:

- **Standardizing** (http://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling): the data are scaled to have zero mean and unit (i.e., one) variance.
- **Normalizing** (http://scikit-learn.org/stable/modules/preprocessing.html#scaling-features-to-a-range): the data are scaled to span a defined range, such as $[0, 1]$.

One important caveat to scaling is that any scaling technique should be *trained* via the `fit` method on the training data used for the machine learning algorithm. Once trained, the scaling technique can be applied equally to the training and testing data. In this manner, the testing data will always match the space spanned by the training data, which is what is used to generate the predictive model.

We demonstrate this approach in the following Code cell, where we compute a standardization from our training data. This transformation is applied to both the training and testing data. We will first demonstrate standardizing with `sklearn StandardScaler`, then normalizing with `sklearn MinMaxScaler`.

In [12]:
```
from sklearn.preprocessing import StandardScaler

# Create and fit scaler
ss = StandardScaler()

ss.fit(d_train)

d_train_ss = ss.transform(d_train)
d_test_ss = ss.transform(d_test)
```

Out[12]:
```
(array([[ 1.14223232, -0.02334012,  0.71995791,  0.65054316],
        [-1.40234464,  0.44346228, -1.23485239, -1.34544153],
        [-0.00282731, -0.95694493,  0.14501371, -0.01478507],
        [-0.89342924,  0.67686349, -1.17735797, -0.94624459],
        [ 0.76054578, -0.49014253,  1.06492443,  1.18280574]]),
 array([[ 1.39669002,  0.21006108,  0.94993559,  1.18280574],
        [ 1.77837656,  0.44346228,  1.29490212,  0.7836088 ],
        [-0.00282731, -0.72354373,  0.20250813, -0.28091636],
        [-1.52957348,  0.21006108, -1.29234681, -1.34544153],
        [-0.13005616, -1.19034613,  0.71995791,  1.0497401 ]]))
```

In [13]:
```
from sklearn.preprocessing import MinMaxScaler

# Create and fit scaler
mms = MinMaxScaler()

mms.fit(d_train)

d_train_mms = mms.transform(d_train)
d_test_mms = mms.transform(d_test)
```

Out[13]:
```
(array([[0.70588235, 0.41666667, 0.6779661 , 0.66666667],
        [0.11764706, 0.5       , 0.10169492, 0.04166667],
        [0.44117647, 0.25      , 0.50847458, 0.45833333],
        [0.23529412, 0.54166667, 0.11864407, 0.16666667],
        [0.61764706, 0.33333333, 0.77966102, 0.83333333]]),
 array([[0.76470588, 0.45833333, 0.74576271, 0.83333333],
        [0.85294118, 0.5       , 0.84745763, 0.70833333],
        [0.44117647, 0.29166667, 0.52542373, 0.375     ],
        [0.08823529, 0.45833333, 0.08474576, 0.04166667],
        [0.41176471, 0.20833333, 0.6779661 , 0.79166667]]))
```

Standardizing or Normalizing?

Use normalizing as the default if you are transforming a feature. It is non-distorting. If there are outliers in the dataset, however, normalizing may be problematic. You might be better off removing the outliers before applying normalizing. There are other scaling methods that deal with outliers better (RobustScaler) but they are out of the scope of this course.

If a feature is relatively normally distributed, you may consider using standardizing. Outliers will have less impact when using standardizing. But if the feature is not normally distributed, standardizing is less effective than normalizing.

Not all machine learning algorithms require data scaling. For example, scaling is not necessary for decision tree or random forest. We will discuss data scaling in more details when we introduce machine learning algorithms in future lessons.

Ancillary Information

The following links are to additional documentation you might find helpful in learning this material. Reading these web-accessible documents is completely optional.

1. The scikit-learn tutorial (http://scikit-learn.org/stable/tutorial/basic/tutorial.html) on the sciki-learn website.