Prepare & Clean Data

Tuesday, December 24, 2019 12:22 PM

Note Written by: Zulfadli Zainal

Instead of prepare data manually, we should write functions.

Reasons:

1. Allow us to reproduce when get new dataset
2. We will gradually build a transformation library that we can reuse
3. We can use it in live system -> transform the data before feeding to algorithm

For this exercise, we try to separate predictors (Features) with label (Values):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

Data Cleaning

Most ML algorithm cannot work with missing features.

Eg: In the "total_bedrooms" attribute has some missing values. So, let's fix this.

How? Several Options ---

1. Get rid of corresponding districts (remove row)
2. Get rid of the whole attributes (remove column)
3. Set the values to some values (eg: 0, mean, median)

All option can be accomplished by:

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median)     # option 3
```

housing.dropna(subset=["total_bedrooms"]) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median) # option 3

What is the meaning of Option 3? (Read! Important!)

If you choose option 3, you should compute the median value on the training set, and use it to fill the missing values in the training set, but also don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.

Using Scikit Learn (Imputer) for Data Cleaning

Scikit-Learn provides a handy class to take care of missing values: Imputer

```
from sklearn.impute import SimpleImputer
```

from sklearn.impute import SimpleImputer

Start using it:

```
#################Cleaning Data - Using Scikit Learn Imputer####################

# Define Imputer strategy
imputer = SimpleImputer(strategy="median")

# Since Imputer can only process numerical data, we need to seperate text attributes
housing_num = housing.drop("ocean_proximity", axis=1)

# Fit imputer instance to training data
option4 = SimpleImputer.fit(imputer, housing_num)

# Now can use the trained Imputer to Transform training set, by replacing missing valued
option4 = SimpleImputer.transform(imputer, housing_num)
# Output is just numpy array

# Change back to pandas dataframe
housing_tr = pd.DataFrame(option4, columns=housing_num.columns)
```

Since the median can only be computed on numerical attributes, we need to create a copy of the data without the text attribute "ocean_proximity".

 # Define Imputer strategy
imputer = SimpleImputer(strategy="median")

 # Since Imputer can only process numerical data, we need to seperate text attributes
housing_num = housing.drop("ocean_proximity", axis=1)

Now you can fit the imputer instance to the training data using the fit() method:

 # Fit imputer instance to training data
option4 = SimpleImputer.fit(imputer, housing_num)

Now you can use this “trained” imputer to transform the training set by replacing missing values by the learned medians:

 # Now can use the trained Imputer to Transform training set, by replacing missing valued by learned medians
option4 = SimpleImputer.transform(imputer, housing_num)
 # Output is just numpy array

The result is a plain Numpy array containing the transformed features. If you want to put it back into a Pandas DataFrame, it’s simple:

 # Change back to pandas dataframe
housing_tr = pd.DataFrame(option4, columns=housing_num.columns)

Handling Text and Categorical Attributes

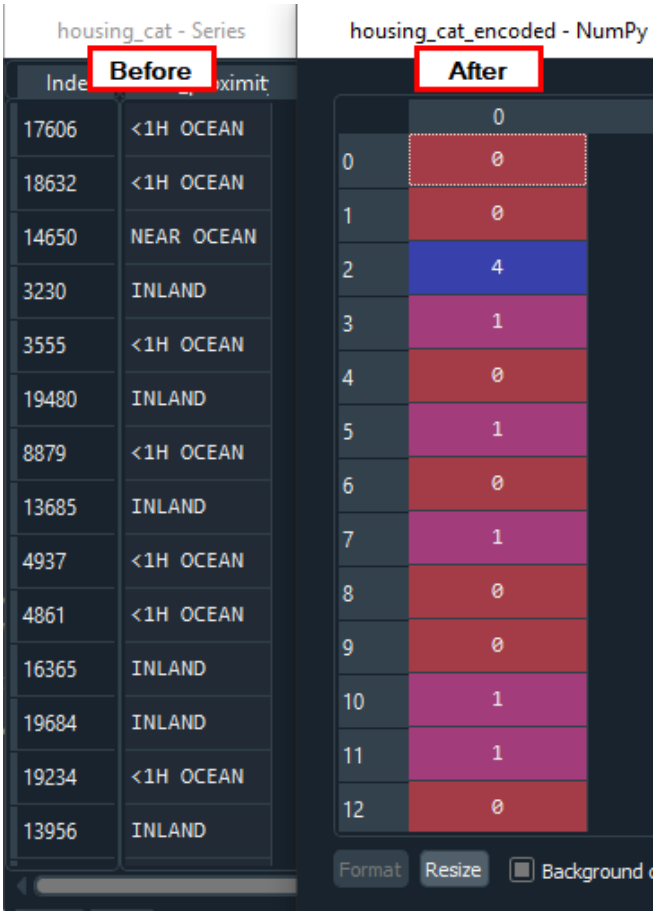
Earlier we left out the categorical attribute "ocean_proximity" because it is a text attribute so we cannot compute its median.

Text Category -> Integer Category (Label Encoder)

Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these text labels to numbers.

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([1, 1, 4, ..., 1, 0, 3])
```

```
from sklearn.preprocessing import LabelEncoder

#Define encoder function
encoder = LabelEncoder()

#Transform column to numerical category
housing_cat = housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)
```



Using this encoder function, the function able to learn the column containing text and change it to category (in numerical).

Eg: "<1H OCEAN" is mapped to 0, "INLAND" is mapped to 1, etc.

How to check the category?

print(encoder.classes_)

```
In [63]: print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
       0         1        2         3          4
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously this is not the case (for example, categories 0 and 4 are more similar than categories 0 and 1).

To solve it: use One Hot Encoder Method

Integer Category -> Binary Vectors (One Hot Encoder)

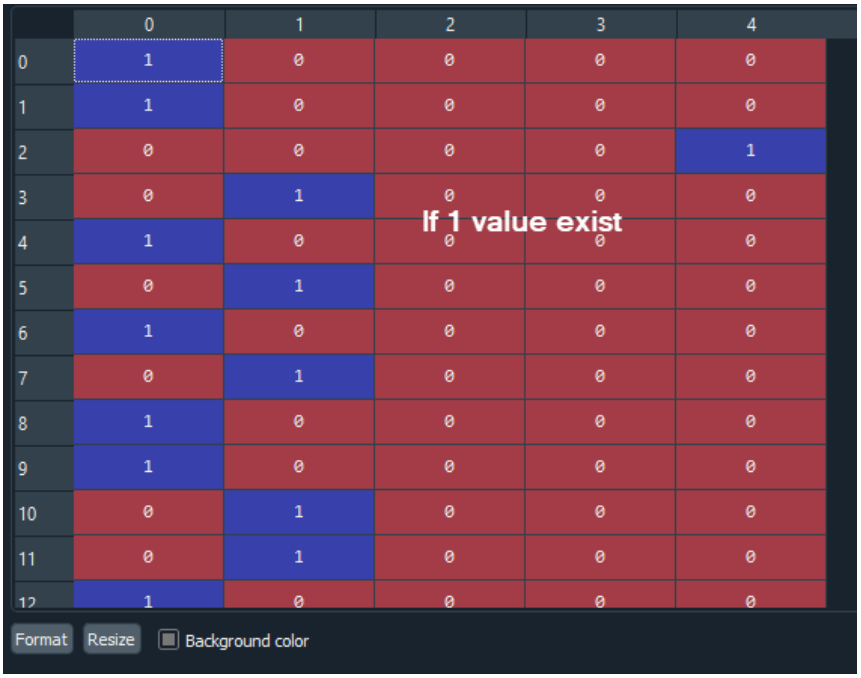
One Hot Encoder - Create one binary attribute (col) per category: one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on.

It is like create a binary switch - IF value exist 1, if not 0

Scikit-Learn provides a OneHotEncoder encoder to convert **integer** categorical values into one-hot vectors.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16513 stored elements in Compressed Sparse Row format>


 >>> housing_cat_1hot.toarray()
```

```
from sklearn.preprocessing import OneHotEncoder

# Define encoder function
encoder = OneHotEncoder()

# Transform integer category to 1hot
housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
#Output in Sparse array

# Convert to Numpy Array
housing_cat_1hot = housing_cat_1hot.toarray()
```



	0	1	2	3	4
0	1	0	0	0	0
1	1	0	0	0	0
2	0	0	0	0	1
3	0	1	0	0	0
4	1	0	0	0	0
5	0	1	0	0	0
6	1	0	0	0	0
7	0	1	0	0	0
8	1	0	0	0	0
9	1	0	0	0	0
10	0	1	0	0	0
11	0	1	0	0	0
12	1	0	0	0	0

THE GREATEST SOLUTION!

Text Category -> Integer Category -> Binary Vectors -> Numpy Array (Label Binarizer)

Perform all method in one shot.

```
from sklearn.preprocessing import LabelBinarizer
```

```
###############Text Categorical Method (Label Binalizer Method)#################

# Define encoder function
encoder = LabelBinarizer()

# Transform text category -> integer category -> one hot vector -> numpy array
housing_cat = housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)
```

from sklearn.preprocessing import LabelBinarizer

 # Define encoder function
encoder = LabelBinarizer()

 # Transform text category -> integer category -> one hot vector -> numpy array
housing_cat = housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)

Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes.

Sample:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]


          return np.c_[X, rooms_per_household, population_per_household,
                       bedrooms_per_room]
      else:
          return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Feature Scaling - Important!!

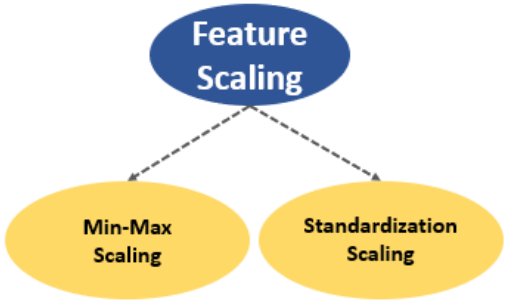
One of the most important transformations you need to apply to your data is feature scaling.

With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales.

For Example:

The total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15.

How to do feature scaling?



Min Max Scaling (Normalization)

1. Most people call this 'Normalization'
2. Rescale ranging from 0 to 1
3. Formula = [x - Min / (Max - Min)]
4. Scikit Learn provides a transformer - MinMaxScaler (It have hyperparameter to change the range of scale if we don't want 0 to 1 for some reason)

Standardization Scaling

1. First it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance.
2. Does not bound to any range - Problem for some algorithm (Eg: Neural Network always expected value from 0 to 1)
3. Less affected by outliers!

Important example: For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.

Transformation Pipelines

Scikit-Learn provides a transformer called StandardScaler for standardization.

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations.

Sample small pipeline:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

More details on how to build a pipeline class in book: Currently I am not focusing on pipeline transformation.