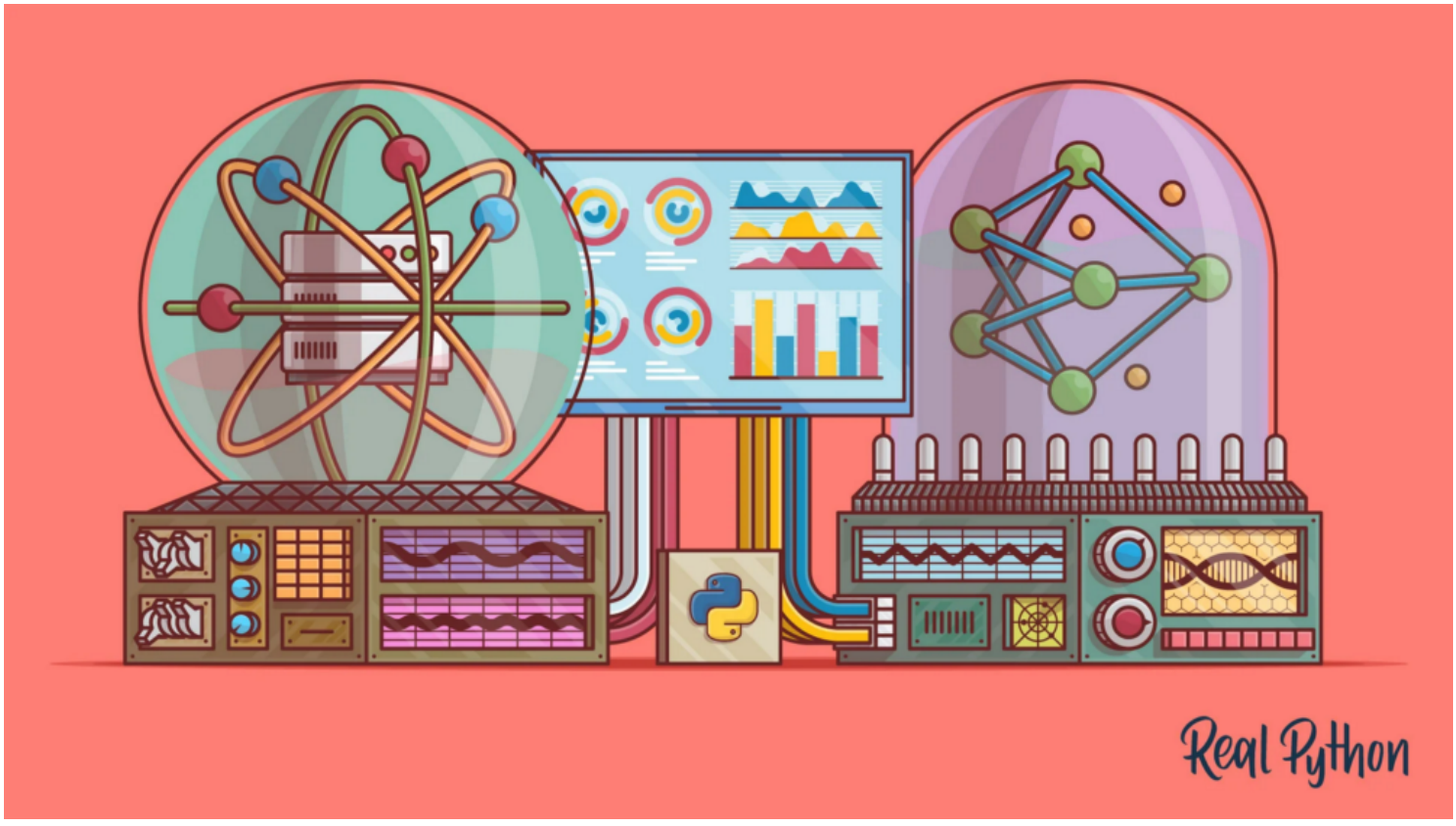
[Chapter-2] Data Preparation for End-to-End Machine Learning Project — part-3

Vishvdeep Dasadiya Follow
Dec 23, 2020 · 7 min read

Prepare the Data for Machine Learning Model…



Real python image all credit goes of the image to real python

We have done things so far and now we have better understanding of datasets and know where need to be clean and prepare your dataset.

It's time to prepare the data for the **Machine Learning Algorithm,** instead of doing it manually we should write a function, moreover we can write small framework for us to do things automatically 🧒🔧. But as of now we are just going to focus on summary of this chapter. Future posts will be on `How to Build small framework for data cleansing in python.`

- This will allow you to reproduce these transformations easily on any dataset.
- You will gradually build a small framework 🧒🏭 of data wrangling that you can use in future projects.
- You can use this library in ML pipeline for new data and new model version before feeding into your **Machine Learning algorithm.**
- This will make it possible for you to easily try various transformations and see which combination of transformation works best🤖.

. . .

Data Cleaning 😷

Most of **Machine Learning algorithm** cannot work with missing features, so let's create a few function to take care of them. In previous blog we have seen that **total_bedrooms** attribute has some missing values, so let's fix it 🤦. In order to fill these missing values we have three options:

- Get rid of the missing values.
- Get rid of the whole features.
- Set the values to some values (zero, the mean, the median, etc.)

First things 🤷, getting rid of the missing values might lost our one of the most critical scenario, so in the worst case we should do this and also goes for the same with getting rid of the while features. We do unless and until, we do not have any options left.

If you choose 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 do not forgot to save the median

somewhere 👩‍🎨, it will be handy later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live (into production) 👷 to replace missing values in new data.

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

Another things, in Scikit-Learn provides a handy class to take care of missing values: SimpleImputer 🤹. Here it how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

Since the median can only be computed on numerical attributes, we need to create a copy of the data. And then use **fit() method** on dataset 😤.

The imputer has simply computed the median of each attribute and stored the result in its **statistics_instance variable.** Although we know only **total_bedrooms** attribute had missing values, still it would be safer to implement on all numerical attribute before system goes live to 😎production.

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. ,
3.5409])

>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. ,
3.5409])

X = imputer.transform(housing_num)

housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

By using **transform() method** we implement transformation on dataset using imputer object 🤜. The result is a plain NumPy array containing the transformed features.

. . .

Handling Text and Categorical Attributes🧑

Earlier we left out the categorical attribute **ocean_proximity** because it is a text attribute so we cannot 😳 compute it's median:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
ocean_proximity
17606 <1H OCEAN
18632 <1H OCEAN
14650 NEAR OCEAN
3230 INLAND
3555 <1H OCEAN
19480 INLAND
8879 <1H OCEAN
13685 INLAND
4937 <1H OCEAN
4861 <1H OCEAN
```

Almost all **Machine Learning algorithm** prefer to work with numerical values anyway, so let's convert these categorical from text to numbers. For this one Scikit-Learn provides 😸 **Ordinal Encoder.**

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()

>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)

>>> housing_cat_encoded[:10]
array([[0.],
[0.],
[4.],
[1.],
[0.],
[1.],
```

```
[0.],
[1.],
[0.],
[0.]])

>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
dtype=object)]
```

Although it transforms the categorical values into numerical, still there is one issue with this representation is that ML algorithm will assume that two nearby values more similar than two distant values. This will be okay in some cases for ordered categories such as 'bad', 'average', 'good', and, 'excellent'. But it is not in the case of **ocean_proximity** column. Now what we can do to resolve it??? 🥸

Answer to this question is that create one binary attribute per category: one attribute is equal to one (1) when category is there, otherwise attribute equal to 0 when category is not there on particular column this is called **One-Hot-Encoding** 🤑**.** Because only one attribute will be equal to 1 (HOT), while the others will be 0 (COLD). New attribute sometimes called *dummy* attribute🤩.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
with 16512 stored elements in Compressed Sparse Row format>
```

This will be useful when thousands of categories are there, after **one-hot-encoding** we get thousands of column, and the matrix is full of zeros and ones. So, instead of sparse matrix use 2D array.

```
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
[1., 0., 0., 0., 0.],
[0., 0., 0., 0., 1.],
...,
[0., 1., 0., 0., 0.],
[1., 0., 0., 0., 0.],
[0., 0., 0., 1., 0.]])
```

If a categorical attribute has a large number of possible categories, then one-hot-encoding will in a large number of input features. This result may slow down training and degrade performance 😓. If this happens, you should replace categorical input with numerical, for instance, if feature is filled with country name or city name than replace it with country code 😳. Like this way you can avoid slow down training and tune it better 😉. Sounds very scary 😬 sometimes right do not worry we will discuss one full case study in future blogs😍.

. . .

Feature Scaling 👩‍🎨

One of the most important transformation you need to apply to your dataset is ***Feature Scaling*** 😺**.** Machine Learning algorithm do not perform well on different scales value.

There two most common ways to get all attributes to have the same scale :

- Min-Max scaling
- Standardization

Min-max scaling is kind of simple : shifted and re-scaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus min. Scikit-Learn provides a transformer called 🤘***MinMaxScaler*****.** It has a feature_range hyper-parameter that less you change the range if you don't want 0–1 for some reason.

Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean) and then it divides by the standard deviation so that the resulting distribution has unit variance🤔. Unlike min-max scaling, standardization does not bound values to a specific range, which be a problem for some algorithms. For example, neural networks often expect an input value ranging from 0 to 1.

However, standardization does affect less by outliers. Suppose a district had a median income equal to 100 by mistakenly. 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.

> 👍 ***Important Note: As with all the transformation, it is important to fit the scalars to the training data only, do not do it on full dataset.***

. . .

Transformation Pipelines 🧸

As you can see, there are probably transformation steps that need to be perform in particular line and should be executed in right order 🧒🏭. I am sure that you have heard about pipelines and if not, do not worry we will cover the concept here and 😇 in `small frame work for data wrangling.`

Here is a small pipeline for the numerical attributes:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
('imputer', SimpleImputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler()),
])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps. All but a last estimator should be transformers.

Name can be anything you like (should be relevant to transformation method). When you call **fit_transform()** sequentially on all transformers, passing the output of one method as parameter to other method.

The pipeline expose the same methods as the final estimator 👨‍🔬. So far so good, we have covered categorical and numerical data separately, but what if we want to do it in single pipeline and applies transformation accordingly, then it would be much much better. Let's see one example of it🤑.

Scikit-Learn introduced the **ColumnTransformer** class,

```
from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
full_pipeline = ColumnTransformer([
("num", num_pipeline, num_attribs),
("cat", OneHotEncoder(), cat_attribs),
])
housing_prepared = full_pipeline.fit_transform(housing)
```

Here is how this works:

- First we import the *ColumnTransformer* class, next we get the list of numerical columns names and the list of categorical names, and we construct a *ColumnTransformer*.
- Constructor requires a list of tuples, where each tuple contains a name. A transformer and list of column names applied accordingly.

> 👍 ***Important Note: OneHotEncoder*** *returns a sparse matrix, while the nun_pipeline returns a dense matrix. When there is such a mix of sparse and dense matrices, then we should implement a* ColumnTransformer *estimates the density of the final matrix.*

So that's it we have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column.

. . .

Thank you if you have made it so far, I am preparing data cleansing and data transformation Jupyter Notebooks, that i have learned and used on projects. Here 👆.

Next Article will be on **Train you Machine Learning Model** (Linear Regression)

Twitter Link: https://twitter.com/vishvdeep18 👚

Link Link: https://www.linkedin.com/in/vishvdeep-dasadiya 🧒💻

Sign up for Analytics Vidhya News Bytes

By Analytics Vidhya

Latest news from Analytics Vidhya on our Hackathons and some of our best articles! Take a look

Get this newsletter

Emails will be sent to nchizampeni@gmail.com.
Not you?

Data Science | Machine Learning | Python | Artificial Intelligence | Data Cleaning