Manish Pathak
May 22nd, 2018

DATA MANIPULATION +1

Handling Categorical Data in Python

Learn the common tricks to handle categorical data and preprocess it to build machine learning models!

If you are familiar with machine learning, you will probably have encountered categorical features in many datasets. These generally include different categories or levels associated with the observation, which are non-numerical and thus need to be converted so the computer can process them.

In this tutorial, you'll learn the common tricks to handle this type of data and preprocess it to build machine learning models with them. More specifically, you will learn:

- The difference between categorical and continuous data in your dataset and identifying the type of data.
- to do basic exploration of such data to extract information from it.
- You will learn more about various encoding techniques in machine learning for categorical data in Python.
- Lastly, you'll explore how you can deal with categorical features in big data with Spark:

Want to leave a comment?

Identifying Categorical Data: Nominal, Ordinal and Continuous

Categorical features can only take on a limited, and usually fixed, number of possible values. For example, if a dataset is about information related to users, then you will typically find features like country, gender, age group, etc. Alternatively, if the data you're working with is related to products, you will find features like product type, manufacturer, seller and so on.

These are all categorical features in your dataset. These features are typically stored as text values which represent various traits of the observations. For example, gender is described as Male (M) or Female (F), product type could be described as electronics, apparels, food etc.

Note that these type of features where the categories are only labeled without any order of precedence are called nominal features.

Features which have some order associated with them are called ordinal features. For example, a feature like economic status, with three categories: low, medium and high, which have an order associated with them.

There are also continuous features. These are numeric variables that have an infinite number of values between any two values. A continuous variable can be numeric or a date/time.

Regardless of what the value is used for, the challenge is determining how to use this data in the analysis because of the following constraints:

- Categorical features may have a very large number of levels, known as high cardinality, (for example, cities or URLs), where most of the levels appear in a relatively small number of instances.
- Many machine learning models, such as regression or SVM, are algebraic. This means that

- For the machine, categorical data doesn't contain the same context or information that humans can easily associate and understand. For example, when looking at a feature called `City` with three cities `New York`, `New Jersey` and `New Delhi`, humans can infer that New York is closely related to New Jersey as they are from same country, while New York and New Delhi are much different. But for the model, New York, New Jersey and New Delhi, are just three different levels (possible values) of the same feature `City`. If you don't specify the additional contextual information, it will be impossible for the model to differentiate between highly different levels.

You therefore are faced with the challenge of figuring out how to turn these text values into numerical values for further processing and unmask lots of interesting information which these features might hide. Typically, any standard work-flow in feature engineering involves some form of transformation of these categorical values into numeric labels and then applying some encoding scheme on these values.

General Exploration steps for Categorical Data

In this section, you'll focus on dealing with categorical features in the `pnwflights14` dataset, but you can apply the same procedure to all kinds of datasets. `pnwflights14` is a modified version of Hadley Wickham's `nycflights13` dataset and contains information about all flights that departed from the two major airports of the Pacific Northwest (PNW), SEA in Seattle and PDX in Portland, in 2014: 162,049 flights in total.

To help understand what causes delays, it also includes a number of other useful datasets:

- `weather`: the hourly meterological data for each airport
- `planes`: constructor information about each plane
- `airports`: airport names and locations

throughout the tutorial, namely `pandas` , `numpy` and `copy` .

Also make sure that you set Matplotlib to plot inline, which means that the outputted plot will appear immediately under each code cell.

```
import pandas as pd
import numpy as np
import copy
%matplotlib inline
```

Next you will read the `flights` dataset in a pandas DataFrame with `read_csv()` and check the contents with the `.head()` method.

```
df_flights = pd.read_csv('https://raw.githubusercontent.com/ismayc/pnwflights14/master/
```

```
df_flights.head()
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
0	2014	1	1	1.0	96.0	235.0	70.0	AS	N508AS	145	PDX	ANC	194.0	1542	0.0	1.0
1	2014	1	1	4.0	-6.0	738.0	-23.0	US	N195UW	1830	SEA	CLT	252.0	2279	0.0	4.0
2	2014	1	1	8.0	13.0	548.0	-4.0	UA	N37422	1609	PDX	IAH	201.0	1825	0.0	8.0
3	2014	1	1	28.0	-2.0	800.0	-23.0	US	N547UW	466	PDX	CLT	251.0	2282	0.0	28.0
4	2014	1	1	34.0	44.0	325.0	43.0	AS	N762AS	121	SEA	ANC	201.0	1448	0.0	34.0

As you will probably notice, the DataFrame above contains all kinds of information about flights like year, departure delay, arrival time, carrier, destination, etc.

Note if you are reading the `RDS` file formats you can do so by installing `rpy2` library. Checkout this link to install the library on your system. The simplest way to install the library is using `pip install rpy2` command on command line terminal.

```
from rpy2.robjects import pandas2ri
pandas2ri.activate()
readRDS = robjects.r['readRDS']
RDSlocation = 'Downloads/datasets/nyc_flights/flights.RDS' #location of the file
df_rds = readRDS(RDSlocation)
df_rds = pandas2ri.ri2py(df_rds)

df_rds.head(2)
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
0	2014	1	1	1	96.0	235	70.0	AS	N508AS	145	PDX	ANC	194.0	1542.0	0.0	1.0
1	2014	1	1	4	-6.0	738	-23.0	US	N195UW	1830	SEA	CLT	252.0	2279.0	0.0	4.0

The same `rpy2` library can also be used to read `rda` file formats. The code below reads and loads `flights.rda` into a `pandas` DataFrame:

```
from rpy2.robjects import r
import rpy2.robjects.pandas2ri as pandas2ri
file="~/Downloads/datasets/nyc_flights/flights.rda" #location of the file
rf=r['load'](file)
df_rda=pandas2ri.ri2py_dataframe(r[rf[0]])

df_rda.head(2)
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
0	2014	1	1	1	96.0	235	70.0	AS	N508AS	145	PDX	ANC	194.0	1542.0	0.0	1.0
1	2014	1	1	4	-6.0	738	-23.0	US	N195UW	1830	SEA	CLT	252.0	2279.0	0.0	4.0

The next step is to gather some information about different column in your DataFrame. You can do so by using `.info()`, which basically gives you information about the number of

```
year          162049 non-null int64
month         162049 non-null int64
day           162049 non-null int64
dep_time      161192 non-null float64
dep_delay     161192 non-null float64
arr_time      161061 non-null float64
arr_delay     160748 non-null float64
carrier       162049 non-null object
tailnum       161801 non-null object
flight        162049 non-null int64
origin        162049 non-null object
dest          162049 non-null object
air_time      160748 non-null float64
distance      162049 non-null int64
hour          161192 non-null float64
minute        161192 non-null float64
dtypes: float64(7), int64(5), object(4)
memory usage: 19.8+ MB
None
```

As you can see, columns like `year` , `month` and `day` are read as integers, and `dep_time` , `dep_delay` etc. are read as floats.

The columns with object `dtype` are the possible categorical features in your dataset.

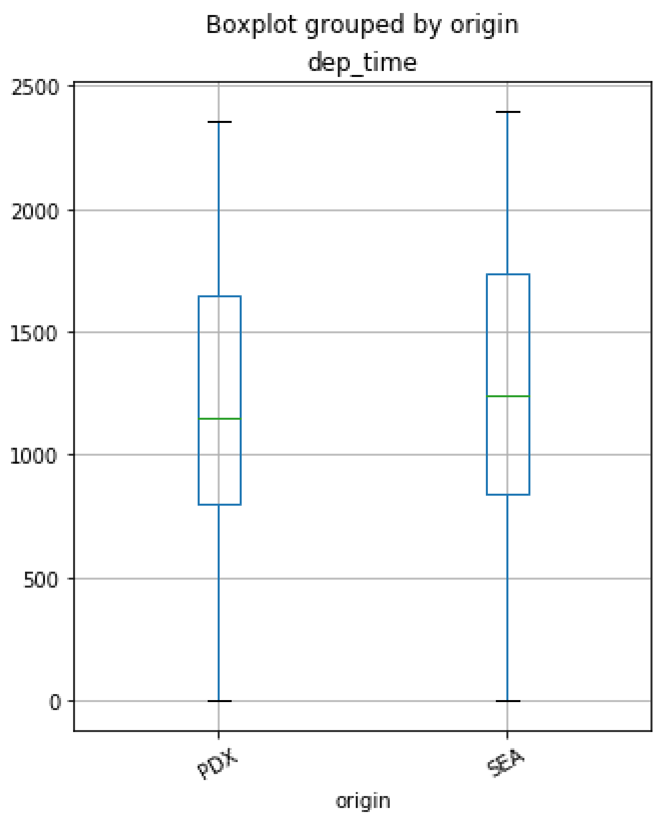
The reason why you would say that these categorical features are 'possible' is because you shouldn't not completely rely on `.info()` to get the real data type of the values of a feature, as some missing values that are represented as strings in a continuous feature can coerce it to read them as `object` dtypes.

quartiles are shown as horizontal lines at either side of the rectangle.

You can plot a boxplot by invoking `.boxplot()` on your DataFrame. Here, you will plot a boxplot of the `dep_time` column with respect to the two `origin` of the flights from `PDX` and `SEA`.

```
df_flights.boxplot('dep_time','origin',rot = 30,figsize=(5,6))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f32ee10f550>
```



As you will only be dealing with categorical features in this tutorial, it's better to filter them out. You can create a separate DataFrame consisting of only these features by running the

```
cat_df_flights.head()
```

	carrier	tailnum	origin	dest
0	AS	N508AS	PDX	ANC
1	US	N195UW	SEA	CLT
2	UA	N37422	PDX	IAH
3	US	N547UW	PDX	CLT
4	AS	N762AS	SEA	ANC

One of the most common data pre-processing steps is to check for null values in the dataset. You can get the total number of missing values in the DataFrame by the following one liner code:

```
print(cat_df_flights.isnull().values.sum())
```

```
248
```

Let's also check the column-wise distribution of null values:

```
print(cat_df_flights.isnull().sum())
```

```
carrier      0
tailnum    248
origin       0
dest         0
dtype: int64
```

Want to leave a comment?

frequency distribution of each category in the feature, and then selecting the top category, which is the mode, with the `.index` attribute.

```
cat_df_flights = cat_df_flights.fillna(cat_df_flights['tailnum'].value_counts().index[(
```

Tip: read more about method chaining with pandas here.

Let's check the number of null values after imputation should result in a zero count.

```
print(cat_df_flights.isnull().values.sum())
```

```
0
```

Another Exploratory Data Analysis (EDA) step that you might want to do on categorical features is the frequency distribution of categories within the feature, which can be done with the `.value_counts()` method as described earlier.

```
print(cat_df_flights['carrier'].value_counts())
```

```
AS    62460
WN    23355
OO    18710
DL    16716
UA    16671
AA     7586
US     5946
B6     3540
VX     3272
```

To know the count of distinct categories within the feature you can chain the previous code with the `.count()` method:

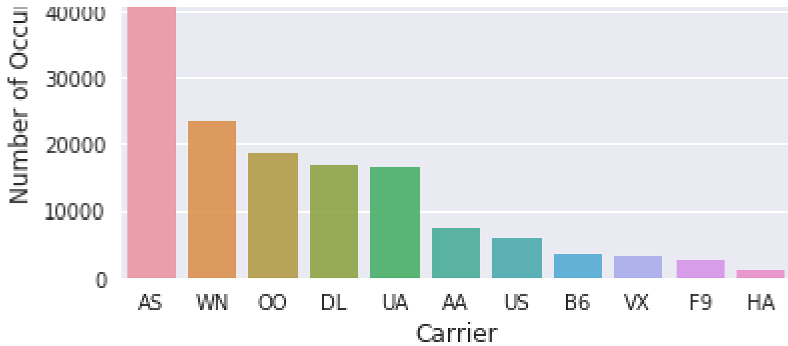
```
print(cat_df_flights['carrier'].value_counts().count())
```

```
11
```

Visual exploration is the most effective way to extract information between variables.

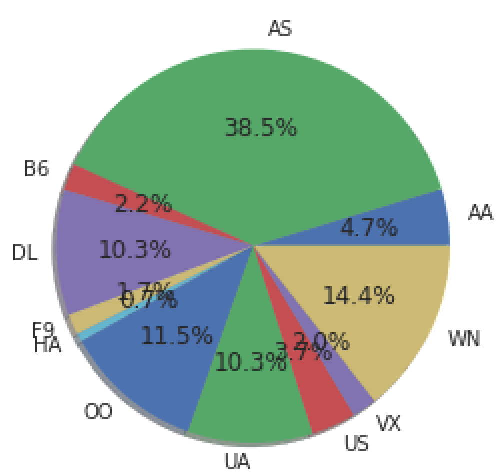
Below is a basic template to plot a barplot of the frequency distribution of a categorical feature using the `seaborn` package, which shows the frequency distribution of the `carrier` column. You can play with different arguments to change the look of the plot. If you want to learn more about seaborn, you can take a look at this tutorial.

```
%matplotlib inline
import seaborn as sns
import matplotlib.pyplot as plt
carrier_count = cat_df_flights['carrier'].value_counts()
sns.set(style="darkgrid")
sns.barplot(carrier_count.index, carrier_count.values, alpha=0.9)
plt.title('Frequency Distribution of Carriers')
plt.ylabel('Number of Occurrences', fontsize=12)
plt.xlabel('Carrier', fontsize=12)
plt.show()
```



Similarly, you could plot a pie chart with the `matplotlib` library to get the same information. The `labels` list below holds the category names from the `carrier` column:

```
labels = cat_df_flights['carrier'].astype('category').cat.categories.tolist()
counts = cat_df_flights['carrier'].value_counts()
sizes = [counts[var_cat] for var_cat in labels]
fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%', shadow=True) #autopct is show the % or
ax1.axis('equal')
plt.show()
```



Encoding Categorical Data

The techniques that you'll cover are the following:

- Replacing values
- Encoding labels
- One-Hot encoding
- Binary encoding
- Backward difference encoding
- Miscellaneous features

Replace Values

Let's start with the most basic method, which is just replacing the categories with the desired numbers. This can be achieved with the help of the `replace()` function in `pandas`. The idea is that you have the liberty to choose whatever numbers you want to assign to the categories according to the business use case.

You will now create a dictionary which contains mapping numbers for each category in the `carrier` column:

```
replace_map = {'carrier': {'AA': 1, 'AS': 2, 'B6': 3, 'DL': 4,
                                  'F9': 5, 'HA': 6, 'OO': 7 , 'UA': 8 , 'US': 9,'VX': 1
```

Note that defining a mapping via a hard coded dictionary is easy when the number of categories is low, like in this case which is 11. You can achieve the same mapping with the help of dictionary comprehensions as shown below. This will be useful when the categories count is high and you don't want to type out each mapping. You will store the category

```
{'carrier': {'AA': 1, 'OO': 7, 'DL': 4, 'F9': 5, 'B6': 3, 'US': 9, 'AS': 2, 'WN': 11,
```

Throughout this tutorial, you will be making a copy of the dataset via the `.copy()` method to practice each encoding technique to ensure that the original DataFrame stays intact and whatever changes you are doing happen only in the copied one.

```
cat_df_flights_replace = cat_df_flights.copy()
```

Use the `replace()` function on the DataFrame by passing the mapping dictionary as argument:

```
cat_df_flights_replace.replace(replace_map_comp, inplace=True)

print(cat_df_flights_replace.head())
```

	carrier	tailnum	origin	dest
0	2	N508AS	PDX	ANC
1	9	N195UW	SEA	CLT
2	8	N37422	PDX	IAH
3	9	N547UW	PDX	CLT
4	2	N762AS	SEA	ANC

As you can observe, you have encoded the categories with the mapped numbers in your DataFrame.

You can also check the dtype of the newly encoded column, which is now converted to

Tip: in Python, it's a good practice to typecast categorical features to a `category` `dtype` because they make the operations on such columns much faster than the `object` dtype. You can do the typecasting by using `.astype()` method on your columns like shown below:

```
cat_df_flights_lc = cat_df_flights.copy()
cat_df_flights_lc['carrier'] = cat_df_flights_lc['carrier'].astype('category')
cat_df_flights_lc['origin'] = cat_df_flights_lc['origin'].astype('category')

print(cat_df_flights_lc.dtypes)
```

```
carrier    category
tailnum      object
origin     category
dest         object
dtype: object
```

You can validate the faster operation of the `category` dtype by timing the execution time of the same operation done on a DataFrame with columns as `category` dtype and `object` dtype by using the `time` library.

Let's say you want to calculate the number of flights for each `carrier` from each `origin` places, you can use the `.groupby()` and `.count()` methods on your DataFrame to do so.

```
import time
%timeit cat_df_flights.groupby(['origin','carrier']).count() #DataFrame with object dty
```

```
10 loops, best of 3: 28.6 ms per loop
```

Note that the DataFrame with `category dtype` is much faster.

Label Encoding

Another 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. Numerical labels are always between 0 and n_categories-1.

You can do label encoding via attributes `.cat.codes` on your DataFrame's column.

```
cat_df_flights_lc['carrier'] = cat_df_flights_lc['carrier'].cat.codes
```

```
cat_df_flights_lc.head() #alphabetically labeled from 0 to 10
```

	carrier	tailnum	origin	dest
0	1	N508AS	PDX	ANC
1	8	N195UW	SEA	CLT
2	7	N37422	PDX	IAH
3	8	N547UW	PDX	CLT
4	1	N762AS	SEA	ANC

Sometimes, you might just want to encode a bunch of categories within a feature to some numeric value and encode all the other categories to some other numeric value.

You could do this by using `numpy` 's `where()` function like shown below. You will encode all the `US` carrier flights to value `1` and other carriers to value `0`. This will create a new column in your DataFrame with the encodings. Later, if you want to drop the original column, you can do so by using the `drop()` function in `pandas`.

0	AS	N508AS	PDX	ANC	0
1	US	N195UW	SEA	CLT	1
2	UA	N37422	PDX	IAH	0
3	US	N547UW	PDX	CLT	1
4	AS	N762AS	SEA	ANC	0

You can achieve the same label encoding using `scikit-learn` 's `LabelEncoder` :

```
cat_df_flights_sklearn = cat_df_flights.copy()

from sklearn.preprocessing import LabelEncoder

lb_make = LabelEncoder()
cat_df_flights_sklearn['carrier_code'] = lb_make.fit_transform(cat_df_flights['carrier'

cat_df_flights_sklearn.head() #Results in appending a new column to df
```

	carrier	tailnum	origin	dest	carrier_code
0	AS	N508AS	PDX	ANC	1
1	US	N195UW	SEA	CLT	8
2	UA	N37422	PDX	IAH	7
3	US	N547UW	PDX	CLT	8
4	AS	N762AS	SEA	ANC	1

Label encoding is pretty much intuitive and straight-forward and may give you a good performance from your learning algorithm, but it has as disadvantage that the numerical values can be misinterpreted by the algorithm. Should the carrier `US` (encoded to 8) be given 8x more weight than the carrier `AS` (encoded to 1) ?

Want to leave a comment?

There are many libraries out there that support one-hot encoding but the simplest one is using `pandas`' `.get_dummies()` method.

This function is named this way because it creates dummy/indicator variables (1 or 0). There are mainly three arguments important here, the first one is the DataFrame you want to encode on, second being the `columns` argument which lets you specify the columns you want to do encoding on, and third, the `prefix` argument which lets you specify the prefix for the new columns that will be created after encoding.

```
cat_df_flights_onehot = cat_df_flights.copy()
cat_df_flights_onehot = pd.get_dummies(cat_df_flights_onehot, columns=['carrier'], pre

print(cat_df_flights_onehot.head())
```

	tailnum	origin	dest	carrier_AA	carrier_AS	carrier_B6	carrier_DL	carrier_F9	carrier_HA	carrier_OO	carrier_UA	carrier_US	carrier_VX	carrier_WN
0	N508AS	PDX	ANC	0	1	0	0	0	0	0	0	0	0	0
1	N195UW	SEA	CLT	0	0	0	0	0	0	0	0	1	0	0
2	N37422	PDX	IAH	0	0	0	0	0	0	0	1	0	0	0
3	N547UW	PDX	CLT	0	0	0	0	0	0	0	0	1	0	0
4	N762AS	SEA	ANC	0	1	0	0	0	0	0	0	0	0	0

As you can see, the column `carrier_AS` gets value `1` at the 0th and 4th observation points as those points had the `AS` category labeled in the original DataFrame. Likewise for other columns also.

`scikit-learn` also supports one hot encoding via `LabelBinarizer` and `OneHotEncoder` in its `preprocessing` module (check out the details here). Just for the sake of practicing you will do the same encoding via `LabelBinarizer`:

```
cat_df_flights_onehot_sklearn = cat_df_flights.copy()
```

Want to leave a comment?

```
print(lb_results_df.head())
```

	AA	AS	B6	DL	F9	HA	OO	UA	US	VX	WN
0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	1	0	0	0	0	0	0	0	0	0

Note that this `lb_results_df` resulted in a new DataFrame with only the one hot encodings for the feature `carrier`. This needs to be concatenated back with the original DataFrame, which can be done via `pandas`' `.concat()` method. The `axis` argument is set to `1` as you want to merge on columns.

```
result_df = pd.concat([cat_df_flights_onehot_sklearn, lb_results_df], axis=1)

print(result_df.head())
```

	carrier	tailnum	origin	dest	AA	AS	B6	DL	F9	HA	OO	UA	US	VX	WN
0	AS	N508AS	PDX	ANC	0	1	0	0	0	0	0	0	0	0	0
1	US	N195UW	SEA	CLT	0	0	0	0	0	0	0	0	1	0	0
2	UA	N37422	PDX	IAH	0	0	0	0	0	0	0	1	0	0	0
3	US	N547UW	PDX	CLT	0	0	0	0	0	0	0	0	1	0	0
4	AS	N762AS	SEA	ANC	0	1	0	0	0	0	0	0	0	0	0

While one-hot encoding solves the problem of unequal weights given to categories within a feature, it is not very useful when there are many categories, as that will result in formation of as many new columns, which can result in the curse of dimensionality. The concept of the from that binary string are split into separate columns. This encodes the data in fewer dimensions than one-hot.

You can do binary encoding via a number of ways but the simplest one is using the `category_encoders` library. You can install category_encoders via `pip install category_encoders` on cmd or just download and extract the .tar.gz file from the site.

You have to first import the `category_encoders` library after installing it. Invoke the `BinaryEncoder` function by specifying the columns you want to encode and then call the `.fit_transform()` method on it with the DataFrame as the argument.

```
cat_df_flights_ce = cat_df_flights.copy()

import category_encoders as ce

encoder = ce.BinaryEncoder(cols=['carrier'])
df_binary = encoder.fit_transform(cat_df_flights_ce)

df_binary.head()
```

	carrier_0	carrier_1	carrier_2	carrier_3	tailnum	origin	dest
0	0	0	0	0	N508AS	PDX	ANC
1	0	0	0	1	N195UW	SEA	CLT
2	0	0	1	0	N37422	PDX	IAH
3	0	0	0	1	N547UW	PDX	CLT
4	0	0	0	0	N762AS	SEA	ANC

Notice that four new columns are created in place of the `carrier` column with binary encoding for each category in the feature.

Want to leave a comment?

5. Backward Difference Encoding

This technique falls under the contrast coding system for categorical features. A feature of K categories, or levels, usually enters a regression as a sequence of K-1 dummy variables. In backward difference coding, the mean of the dependent variable for a level is compared with the mean of the dependent variable for the prior level. This type of coding may be useful for a nominal or an ordinal variable.

If you want to learn other contrast coding methods you can check out this resource.

The code structure is pretty much the same as any method in the `category_encoders` library, just this time you will call `BackwardDifferenceEncoder` from it:

```
encoder = ce.BackwardDifferenceEncoder(cols=['carrier'])
df_bd = encoder.fit_transform(cat_df_flights_ce)

df_bd.head()
```

	col_carrier_0	col_carrier_1	col_carrier_2	col_carrier_3	col_carrier_4	col_carrier_5	col_carrier_6	col_carrier_7	col_carrier_8	col_carrier_9	col_carrier_10	col_tailnum	col_origin	col_dest
0	1.0	-0.909091	-0.818182	-0.727273	-0.636364	-0.545455	-0.454545	-0.363636	-0.272727	-0.181818	-0.090909	N508AS	PDX	ANC
1	1.0	0.090909	-0.818182	-0.727273	-0.636364	-0.545455	-0.454545	-0.363636	-0.272727	-0.181818	-0.090909	N195UW	SEA	CLT
2	1.0	0.090909	0.181818	-0.727273	-0.636364	-0.545455	-0.454545	-0.363636	-0.272727	-0.181818	-0.090909	N37422	PDX	IAH
3	1.0	0.090909	-0.818182	-0.727273	-0.636364	-0.545455	-0.454545	-0.363636	-0.272727	-0.181818	-0.090909	N547UW	PDX	CLT
4	1.0	-0.909091	-0.818182	-0.727273	-0.636364	-0.545455	-0.454545	-0.363636	-0.272727	-0.181818	-0.090909	N762AS	SEA	ANC

The interesting thing here is that you can see that the results are not the standard 1's and 0's you saw in the dummy encoding examples but rather regressed continuous values.

Miscellaneous Features

Sometimes you may encounter categorical feature columns which specify the ranges of values for observation points, for example, the `age` column might be described in the form of categories like 0-20, 20-40 and so on.

Want to leave a comment?

You will first create a dummy DataFrame which has just one feature `age` with ranges specified using the pandas `DataFrame` function. Then you will split the column on the delimeter `-` into two columns `start` and `end` using `split()` with a `lambda()` function. If you want to learn more about lambda functions, check out this tutorial.

```
dummy_df_age = pd.DataFrame({'age': ['0-20', '20-40', '40-60','60-80']})
dummy_df_age['start'], dummy_df_age['end'] = zip(*dummy_df_age['age'].map(lambda x: x.s

dummy_df_age.head()
```

	age	start	end
0	0-20	0	20
1	20-40	20	40
2	40-60	40	60
3	60-80	60	80

To replace the range with its mean, you will write a `split_mean()` function which basically takes one range at a time, splits it, then calculates the mean and returns it. To apply a certain function to all the entities of a column you will use the `.apply()` method:

```
dummy_df_age = pd.DataFrame({'age': ['0-20', '20-40', '40-60','60-80']})

def split_mean(x):
    split_list = x.split('-')
    mean = (float(split_list[0])+float(split_list[1]))/2
    return mean

dummy_df_age['age_mean'] = dummy_df_age['age'].apply(lambda x: split_mean(x))
```

Want to leave a comment?

3	60-80	70.0

Dealing with Categorical Features in Big Data with Spark

Now you will learn how to read a dataset in Spark and encode categorical variables in Apache Spark's Python API, Pyspark. But before that it's good to brush up on some basic knowledge about Spark.

Spark is a platform for cluster computing. It lets you spread data and computations over clusters with multiple nodes. Splitting up your data makes it easier to work with very large datasets because each node only works with a small amount of data.

As each node works on its own subset of the total data, it also carries out a part of the total calculations required, so that both data processing and computations are performed in parallel over the nodes in the cluster.

Deciding whether or not Spark is the best solution for your problem takes some experience, but you can consider questions like:

- Is my data too big to work with on a single machine?
- Can my calculations be easily parallelized?

The first step in using Spark is connecting to a cluster. In practice, the cluster will be hosted on a remote machine that's connected to all other nodes. There will be one computer, called the master that manages splitting up the data and the computations. The master is connected to the rest of the computers in the cluster, which are called slaves. The master sends the slaves data and calculations to run, and they send their results back to the master.

when you want to execute operations in a cluster. SparkContext tells Spark how and where to access a cluster. You'll start by importing SparkContext.

```
from pyspark import SparkContext
sc = SparkContext()
```

Note that if you are working on Spark's interactive shell then you don't have to import SparkContext as it will already be in your environment as `sc`.

To start working with Spark DataFrames, you first have to create a `SparkSession` object from your `SparkContext`. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

Note that if you are working in Spark's interactive shell you'll have a `SparkSession` called `spark` available in your workspace!

```
from pyspark.sql import SparkSession as spark
```

Once you've created a SparkSession, you can start poking around to see what data is in your cluster.

Your `SparkSession` has an attribute called `catalog` which lists all the data inside the cluster. This attribute has a few methods for extracting different pieces of information.

One of the most useful is the `.listTables()` method, which returns the names of all the tables in your cluster as a list.

```
print(spark.catalog.listTables())
```

To read a `.csv` file and create a Spark DataFrame you can use the `.read` attribute of your SparkSession object. Here, apart from reading the csv file, you have to additionally specify the `headers` option to be `True` , since you have column names in the dataset. Also, the `inferSchema` argument is set to `True` , which basically peeks at the first row of the data to determine the fields' names and types.

```
spark_flights = spark.read.format("csv").option('header',True).load('Downloads/datasets
```

To check the contents of your DataFrame you can run the `.show()` method on the DataFrame.

```
spark_flights.show(3)
```

```
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+---
|year|month|day|dep_time|dep_delay|arr_time|arr_delay|carrier|tailnum|flight|origin|des
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+---
|2014|    1|  1|       1|       96|     235|       70|     AS| N508AS|   145|   PDX| AN
|2014|    1|  1|       4|       -6|     738|      -23|     US| N195UW|  1830|   SEA| CL
|2014|    1|  1|       8|       13|     548|       -4|     UA| N37422|  1609|   PDX| IA
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+---
only showing top 3 rows
```

If you wish to convert a pandas DataFrame to a Spark DataFrame, use the `.createDataFrame()` method on your SparkSession object with the DataFrame's name as argument.

To have a look at the schema of the DataFrame you can invoke `.printSchema()` as follows:

Want to leave a comment?

```
 |-- day: integer (nullable = true)
 |-- dep_time: string (nullable = true)
 |-- dep_delay: string (nullable = true)
 |-- arr_time: string (nullable = true)
 |-- arr_delay: string (nullable = true)
 |-- carrier: string (nullable = true)
 |-- tailnum: string (nullable = true)
 |-- flight: integer (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
 |-- air_time: string (nullable = true)
 |-- distance: integer (nullable = true)
 |-- hour: string (nullable = true)
 |-- minute: string (nullable = true)
```

Note that Spark doesn't always guess the data type of the columns right and you can see that some of the columns (`arr_delay` , `air_time` , etc.) which seem to have numeric values are read as strings rather than integers or floats, due to the presence of missing values.

At this point, if you check the data in your cluster using the `.catalog` attribute and the `.listTables()` method like you did before, you will find it's still empty. This is because you DataFrame is currently stored locally, not in the `SparkSession` catalog.

To access the data in this way, you have to save it as a temporary table. You can do so by using the `.createOrReplaceTempView()` method. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific `SparkSession` used to create the Spark DataFrame.

```
spark_flights.createOrReplaceTempView("flights_temp")
```

Want to leave a comment?

Now you have registered the `flight_temp` table as a temporary table in your catalog.

Now that you have gotten your hands dirty with a little bit of PySpark code, it's time to see how to encode categorical features. To keep things neat, you will create a new DataFrame which consists of only the `carrier` column by using the `.select()` method.

```
carrier_df = spark_flights.select("carrier")
carrier_df.show(5)
```

```
+-------+
|carrier|
+-------+
|     AS|
|     US|
|     UA|
|     US|
|     AS|
+-------+
only showing top 5 rows
```

The two most common ways to encode categorical features in Spark are using `StringIndexer` and `OneHotEncoder`.

- `StringIndexer` encodes a string column of labels to a column of label indices. The indices are in [0, numLabels] ordered by label frequencies, so the most frequent label gets index 0. This is similar to label encoding in pandas.

You will start by importing the `StringIndexer` class from the `pyspark.ml.feature`

```
carr_indexer = StringIndexer(inputCol="carrier",outputCol="carrier_index")
carr_indexed = carr_indexer.fit(carrier_df).transform(carrier_df)

carr_indexed.show(7)
```

```
+-------+-------------+
|carrier|carrier_index|
+-------+-------------+
|     AS|          0.0|
|     US|          6.0|
|     UA|          4.0|
|     US|          6.0|
|     AS|          0.0|
|     DL|          3.0|
|     UA|          4.0|
+-------+-------------+
only showing top 7 rows
```

Since `AS` was the most frequent category in the `carrier` column, it got the index `0.0`.

- OneHotEncoder: as you already read before, one-hot encoding maps a categorical feature, represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values.

For example, with 5 categories, an input value of 2.0 would map to an output vector of [0.0, 0.0, 1.0, 0.0]. The last category is not included by default (configurable via OneHotEncoder `.dropLast` because it makes the vector entries sum up to one, and hence linearly dependent. That means that an input value of 4.0 would map to [0.0, 0.0, 0.0, 0.0].

Note that this is different from scikit-learn's OneHotEncoder, which keeps all categories. The

```
carrier_df_onehot = spark_flights.select("carrier")

from pyspark.ml.feature import OneHotEncoder, StringIndexer

stringIndexer = StringIndexer(inputCol="carrier", outputCol="carrier_index")
model = stringIndexer.fit(carrier_df_onehot)
indexed = model.transform(carrier_df_onehot)
encoder = OneHotEncoder(dropLast=False, inputCol="carrier_index", outputCol="carrier_ve
encoded = encoder.transform(indexed)

encoded.show(7)
```

```
+-------+-------------+--------------+
|carrier|carrier_index|   carrier_vec|
+-------+-------------+--------------+
|     AS|          0.0|(11,[0],[1.0])|
|     US|          6.0|(11,[6],[1.0])|
|     UA|          4.0|(11,[4],[1.0])|
|     US|          6.0|(11,[6],[1.0])|
|     AS|          0.0|(11,[0],[1.0])|
|     DL|          3.0|(11,[3],[1.0])|
|     UA|          4.0|(11,[4],[1.0])|
+-------+-------------+--------------+
only showing top 7 rows
```

Note that `OneHotEncoder` has created a vector for each category which can then be processed further by your machine learning pipeline.

There are some more methods available in Spark like `VectorIndexer` , but you have already

Want to leave a comment?

with basic EDA in pandas and then you practiced the different encoding methods available. You also learned a bit about Spark's architecture and moved to encoding categorical data in PySpark. That's a lot, but there is always so much more to learn. You will also have to make sure that you data is properly cleaned. To learn how to do so, take a look at our Cleaning Data in Python course. Happy exploring!!

COMMENTS

Xuezhou Liu
22/05/2018 07:45 PM
leanred something new

1 REPLY

Dung Doan
22/05/2018 08:10 PM
Good!

1 REPLY

Michal Lasak
24/05/2018 11:20 AM
I believe there's an error in the one-hot encoding/ LabelBinarizer part: it should be *lb_results_df = pd.DataFrame(lb_results, columns=lb.classes_)* , as *lb_style* is not defined anywhere before.

Apart from that, great tutorial!

Want to leave a comment?

Joe Bloggs
28/05/2018 05:55 AM

Some discussion of categorical embedding using neural networks would help round out the picture. This was applied successfully in a third place solution to the Rossmann stores competition. Seems like a great alternative to one-hot with the added benefit of dimensionality reduction.

▲ 1 REPLY

Sheshenderr Goswami
21/06/2018 03:11 AM

Hi

I'm getting a value error: fill value must be in categories. When i use binary encoding. The column to be coded has no missing values.

▲ 1 REPLY

Heba Mazhar
09/08/2018 10:11 AM

Could someone help me with downloading categorical_encoders??
When I use pip install categorical_encoders -- I get this error message "error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools"

And when I download and unzip the file, and run my code it says failed to import library "categorical_encoders"
Please help !!!

▲ 1 REPLY

Manish Pathak
11/08/2018 11:30 PM

Hi Heba,

Want to leave a comment?

inferential statistics can then be performed?

Can those features now be treated as continuous datatypes? Or, do they still get treated differently, with different statistics applied to them?

Do their dtypes need to be changed (or are automatically changed) to int/float dtypes?

▲ 1 REPLY

Manish Pathak
23/09/2018 12:21 PM

Hi,

So the answers to your questions are as follows:

1. inferential statistics can then be performed? -- It totally depends on the context. For example, if you have done label encoding for an ordinal categorical variable, suppose levels like bad(0),average(1),good(2), then it would make sense to infer something from mean values (like 1.5 etc.) which is a property for continuous variable. But if your categorical variable is nominal then properties which are valid for categorical data (like mode etc.) would be applicable.

2. Can those features now be treated as continuous datatypes? Or, do they still get treated differently, with different statistics applied to them? -- I believe it should be clear from the answer to the first question. Moreover, if you have done one-hot encoding or backward difference encoding they can be treated like continuous variables.

3. Do their dtypes need to be changed (or are automatically changed) to int/float dtypes? -- I have already shown the answer for this question in the tutorial.

▲ 1 REPLY

RAHUL KUMAR
29/09/2018 03:25 AM

Good !!! I have just one query ....Which IDE u have used ??

▲ 1 REPLY

Want to leave a comment?

Want to leave a comment?