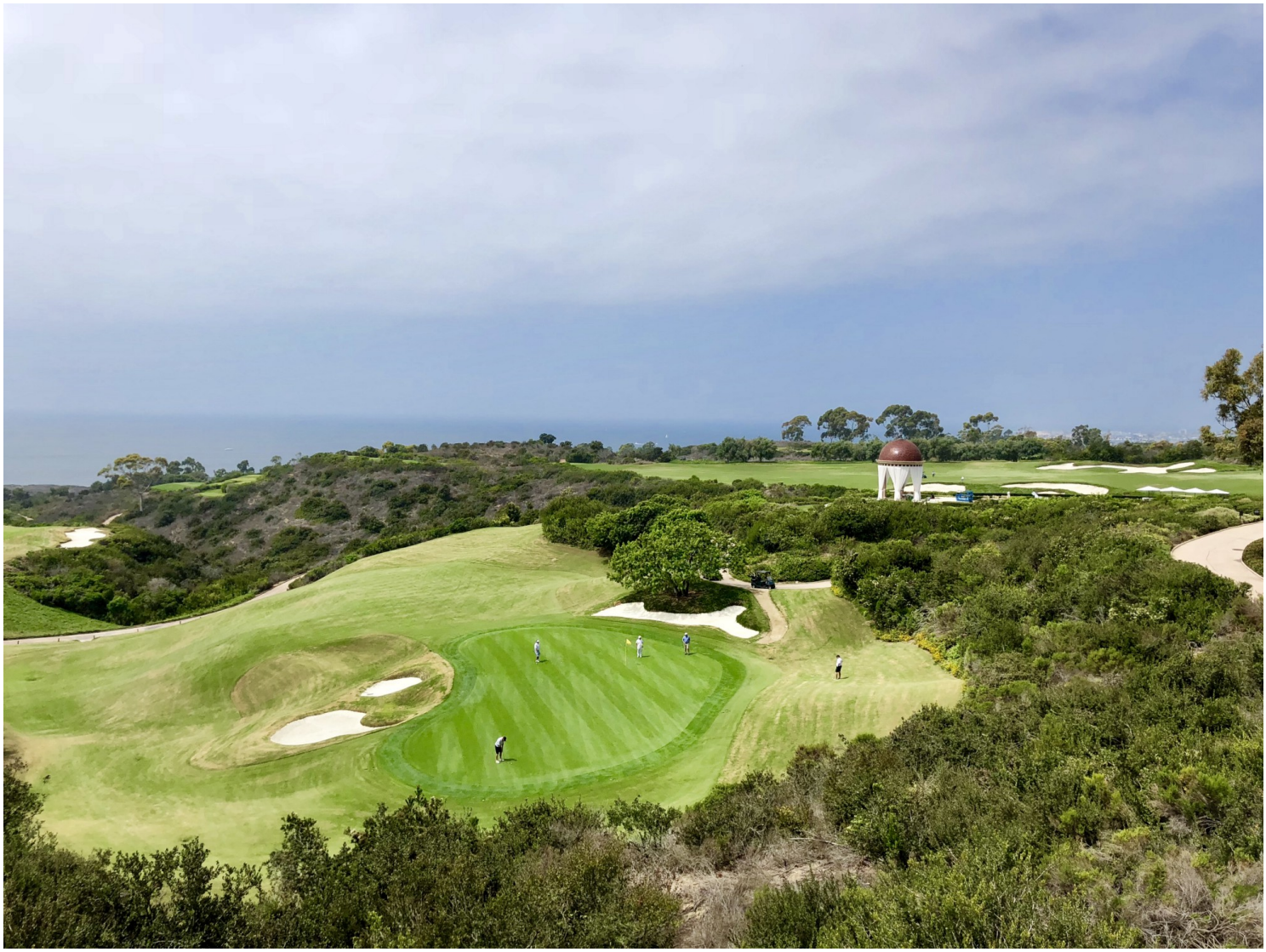
Encoding Categorical Features

Yang Liu Follow

Sep 13, 2018 · 6 min read



Introduction

In machine learning projects, one important part is feature engineering. It is very common to see categorical features in a dataset. However, our machine learning

algorithm can only read numerical values. It is essential to encoding categorical features into numerical values.

Here we will cover three different ways of encoding categorical features:

1. LabelEncoder and OneHotEncoder

2. DictVectorizer

3. Pandas get_dummies

For your convenience, the complete code can be found in my github.

Data Set

The data set we use here is from UCI Machine Learning Repository. It is used to predict whether a patient has kidney disease using various blood indicators as features. We use pandas to read the data in.

```
# load data
df = pd.read_csv('datasets/chronic_kidney_disease.csv', header=None,
 names=['age', 'bp', 'sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba',
'bgr', 'bu', 'sc', 'sod', 'pot',
 'hemo', 'pcv', 'wc', 'rc', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane',
'class'])

# head of df
df.head(10)
```

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	bu	sc	sod	pot	hemo	pcv	wc	rc	htn	dm	cad	appet	pe	ane	class
0	48	80	1.020	1	0	?	normal	notpresent	notpresent	121	36	1.2	?	?	15.4	44	7800	5.2	yes	yes	no	good	no	no	ckd
1	7	50	1.020	4	0	?	normal	notpresent	notpresent	?	18	0.8	?	?	11.3	38	6000	?	no	no	no	good	no	no	ckd
2	62	80	1.010	2	3	normal	normal	notpresent	notpresent	423	53	1.8	?	?	9.6	31	7500	?	no	yes	no	poor	no	yes	ckd
3	48	70	1.005	4	0	normal	abnormal	present	notpresent	117	56	3.8	111	2.5	11.2	32	6700	3.9	yes	no	no	poor	yes	yes	ckd
4	51	80	1.010	2	0	normal	normal	notpresent	notpresent	106	26	1.4	?	?	11.6	35	7300	4.6	no	no	no	good	no	no	ckd
5	60	90	1.015	3	0	?	?	notpresent	notpresent	74	25	1.1	142	3.2	12.2	39	7800	4.4	yes	yes	no	good	yes	no	ckd
6	68	70	1.010	0	0	?	normal	notpresent	notpresent	100	54	24.0	104	4.0	12.4	36	?	?	no	no	no	good	no	no	ckd
7	24	?	1.015	2	4	normal	abnormal	notpresent	notpresent	410	31	1.1	?	?	12.4	44	6900	5	no	yes	no	good	yes	no	ckd
8	52	100	1.015	3	0	normal	abnormal	present	notpresent	138	60	1.9	?	?	10.8	33	9600	4.0	yes	yes	no	good	no	yes	ckd
9	53	90	1.020	2	0	abnormal	abnormal	present	notpresent	70	107	7.2	114	3.7	9.5	29	12100	3.7	yes	yes	no	poor	no	yes	ckd

We can see from the table, we have several categorical features, such as 'rbc'(red blood cells), 'pc'(pus cell), 'pcc'(pus cell clumps) and so on. It is obvious the dataset contains missing values, but since this is out scope of the topic in this blog, we'll not cover how to do it here. But filling missing values is essential and should be done before encoding categorical features. You can refer to my github to see how to perform filling missing values. The completed dataset that ready for categorical encoding is shown as follows.

age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	bu	sc	sod	pot	hemo	pcv	wc	rc	htn	dm	cad	appet	pe	ane
48.0	80.0	1.020	1.0	0.0	normal	normal	notpresent	notpresent	121.0	36.0	1.2	138.0	4.4	15.4	44.0	7800.0	5.2	yes	yes	no	good	no	no
7.0	50.0	1.020	4.0	0.0	normal	normal	notpresent	notpresent	121.0	18.0	0.8	138.0	4.4	11.3	38.0	6000.0	4.8	no	no	no	good	no	no
62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	53.0	1.8	138.0	4.4	9.6	31.0	7500.0	4.8	no	yes	no	poor	no	yes
48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	56.0	3.8	111.0	2.5	11.2	32.0	6700.0	3.9	yes	no	no	poor	yes	yes
51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	26.0	1.4	138.0	4.4	11.6	35.0	7300.0	4.6	no	no	no	good	no	no
60.0	90.0	1.015	3.0	0.0	normal	normal	notpresent	notpresent	74.0	25.0	1.1	142.0	3.2	12.2	39.0	7800.0	4.4	yes	yes	no	good	yes	no
68.0	70.0	1.010	0.0	0.0	normal	normal	notpresent	notpresent	100.0	54.0	24.0	104.0	4.0	12.4	36.0	8000.0	4.8	no	no	no	good	no	no
24.0	80.0	1.015	2.0	4.0	normal	abnormal	notpresent	notpresent	410.0	31.0	1.1	138.0	4.4	12.4	44.0	6900.0	5.0	no	yes	no	good	yes	no
52.0	100.0	1.015	3.0	0.0	normal	abnormal	present	notpresent	138.0	60.0	1.9	138.0	4.4	10.8	33.0	9600.0	4.0	yes	yes	no	good	no	yes
53.0	90.0	1.020	2.0	0.0	abnormal	abnormal	present	notpresent	70.0	107.0	7.2	114.0	3.7	9.5	29.0	12100.0	3.7	yes	yes	no	poor	no	yes

LabelEncoder & OneHotEncoder

The labelEncoder and OneHotEncoder only works on categorical features. We need first to extract the categorial featuers using boolean mask.

```
# Categorical boolean mask
categorical_feature_mask = X.dtypes==object

# filter categorical columns using mask and turn it into a list
categorical_cols = X.columns[categorical_feature_mask].tolist()
```

LabelEncoder converts each class under specified feature to a numerical value. Let's go through the steps to see how to do it.

Instantiate a LabelEncoder object:

```
# import labelencoder
from sklearn.preprocessing import LabelEncoder
```

```
# instantiate labelencoder object
le = LabelEncoder()
```

Apply LabelEncoder on each of the categorical columns:

```
# apply le on categorical feature columns
X[categorical_cols] = X[categorical_cols].apply(lambda col:
le.fit_transform(col))

X[categorical_cols].head(10)
```

Note that the output of LabelEncoder is still a dataframe. The results is shown as following:

	rbc	pc	pcc	ba	htn	dm	cad	appet	pe	ane
0	1	1	0	0	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	1	0	0	0	1	0	1	0	1
3	1	0	1	0	1	0	0	1	1	1
4	1	1	0	0	0	0	0	0	0	0
5	1	1	0	0	1	1	0	0	1	0
6	1	1	0	0	0	0	0	0	0	0
7	1	0	0	0	0	1	0	0	1	0
8	1	0	1	0	1	1	0	0	0	1
9	0	0	1	0	1	1	0	1	0	1

As we can see, all the categorical feature columns are binary class. But if the categorical feature is multi class, LabelEncoder will return different values for different classes. See the following example, the 'Neighborhood' feature has as many as 24 classes.

	MSZoning	Neighborhood	BldgType	HouseStyle	PavedDrive
0	3	5	0	5	2
1	3	24	0	2	2
2	3	5	0	5	2
3	3	6	0	5	2
4	3	15	0	5	2

In this case, using LabelEncoder only is not a good choice, since it brings in a natural ordering for different classes. For example, under 'Neighborhood' feature, class_a has value 5 but class_b has value 24, is class_b 'greater' than class_a? The answer is obviously no. Thus allowing model learning this result will lead to poor performance. Therefore, for dataframe containing multi class features, a further step of OneHotEncoder is needed. Let's see the steps to do it.

Instantiate OneHotEncoder object:

```
# import OneHotEncoder
from sklearn.preprocessing import OneHotEncoder

# instantiate OneHotEncoder
ohe = OneHotEncoder(categorical_features = categorical_feature_mask,
sparse=False )
# categorical_features = boolean mask for categorical columns
# sparse = False output an array not sparse matrix
```

We need to specify categorical feature using its mask inside OneHotEncoder. The `sparse=False` argument outputs a non-sparse matrix.

Apply OneHotEncoder on DataFrame:

```
# apply OneHotEncoder on categorical feature columns
X_ohe = ohe.fit_transform(X) # It returns an numpy array
```

Note that the output is a numpy array, not a dataframe. For each class under a categorical feature, a new column is created for it. For example, there are 20 columns created for the ten binary class categorical features.

```
array([[0.0e+00, 1.0e+00, 0.0e+00, ..., 4.4e+01, 7.8e+03, 5.2e+00],
       [0.0e+00, 1.0e+00, 0.0e+00, ..., 3.8e+01, 6.0e+03, 4.8e+00],
       [0.0e+00, 1.0e+00, 0.0e+00, ..., 3.1e+01, 7.5e+03, 4.8e+00],
       ...,
       [0.0e+00, 1.0e+00, 0.0e+00, ..., 4.9e+01, 6.6e+03, 5.4e+00],
       [0.0e+00, 1.0e+00, 0.0e+00, ..., 5.1e+01, 7.2e+03, 5.9e+00],
       [0.0e+00, 1.0e+00, 0.0e+00, ..., 5.3e+01, 6.8e+03, 6.1e+00]])
```

DictVectorizer

As we can see, the LabelEncoder and OneHotEncoder usually need to be used together as two steps procedure. An more convenient way is using DictVectorizer which can achieve these two steps all at once.

First, we need to convert the dataframe into a dictionary. This can be achieved by Pandas `to_dict` method.

```
# turn X into dict
X_dict = X.to_dict(orient='records') # turn each row as key-value
pairs

# show X_dict
X_dict
```

The `orient='records'` is required to turn the data frame into a `{column:value}` format. The result is a list of dictionaries, among which each dictionary represent one sample. Note that, in this case we don't need to extract the categorical features, we can convert the whole dataframe into a dict. This is one advantage compared to LabelEncoder and OneHotEncoder.

```
[{'age': 48.0,
  'bp': 80.0,
  'sg': 1.02,
  'al': 1.0,
  'su': 0.0,
  'rbc': nan,
  'pc': 'normal',
  'pcc': 'notpresent',
  'ba': 'notpresent',
  'bgr': 121.0,
  'bu': 36.0,
  'sc': 1.2,
  'sod': nan,
  'pot': nan,
  'hemo': 15.4,
  'pcv': 44.0,
  'wc': 7800.0,
  'rc': 5.2,
  'htn': 'yes',
  'dm': 'yes',
  'cad': 'no',
  'appet': 'good',
```

```
 'pe': 'no',
 'ane': 'no'},
{'age': 7.0,
 'bp': 50.0,
 'sg': 1.02,
 'al': 4.0,
 'su': 0.0,
 'rbc': nan,
```

Now we instantiate a DictVectorizer:

```
# DictVectorizer
from sklearn.feature_extraction import DictVectorizer

# instantiate a Dictvectorizer object for X
dv_X = DictVectorizer(sparse=False)
# sparse = False makes the output is not a sparse matrix
```

The `sparse=False` makes the output to be a non-sparse matrix.

DictVectorizer fit and transform on the converted dict:

```
# apply dv_X on X_dict
X_encoded = dv_X.fit_transform(X_dict)

# show X_encoded
X_encoded
```

The result is a numpy array:

```
array([[4.80e+01, 1.00e+00, 0.00e+00, ..., 1.38e+02, 0.00e+00, 7.80e+03],
       [7.00e+00, 4.00e+00, 0.00e+00, ..., 1.38e+02, 0.00e+00, 6.00e+03],
       [6.20e+01, 2.00e+00, 1.00e+00, ..., 1.38e+02, 3.00e+00, 7.50e+03],
       ...,
       [1.20e+01, 0.00e+00, 0.00e+00, ..., 1.37e+02, 0.00e+00, 6.60e+03],
       [1.70e+01, 0.00e+00, 0.00e+00, ..., 1.35e+02, 0.00e+00, 7.20e+03],
       [5.80e+01, 0.00e+00, 0.00e+00, ..., 1.41e+02, 0.00e+00, 6.80e+03]])
```

Each row represents a sample and each column represents a feature. If we want to know what feature for each column, we can check the vocabulary of this DictVectorizer:

```
# vocabulary
vocab = dv_X.vocabulary_

# show vocab
vocab
```

```
{'age': 0,
 'bp': 6,
 'sg': 20,
 'al': 1,
 'su': 22,
 'bgr': 5,
 'bu': 7,
 'sc': 19,
 'sod': 21,
 'pot': 16,
 'hemo': 10,
 'pcv': 14,
 'wc': 23,
 'rc': 18,
 'rbc': 17,
 'pc': 12,
 'pcc': 13,
 'ba': 4,
 'htn': 11,
 'dm': 9,
 'cad': 8,
 'appet': 3,
 'pe': 15,
 'ane': 2}
```

Get Dummies

Pandas get_dummies method is a very straight forward one step procedure to get the dummy variables for categorical features. The advantage is you can directly apply it on the dataframe and the algorithm inside will recognize the categorical features and perform get dummies operation on it. Here is how to do it:

```
# Get dummies
X = pd.get_dummies(X, prefix_sep='_', drop_first=True)

# X head
X.head()
```

The `prefix_sep='_'` makes each class has a unique name separated by the delimiter. The `drop_first=True` drops one column from the resulted dummy features. The purpose is to avoid multicollinearity. Here is the results:

sod	pot	hemo	pcv	wc	rc	rbc_normal	pc_normal	pcc_present	ba_present	htn_yes	dm_yes	cad_yes	appet_poor	pe_yes	ane_yes
138.0	4.4	15.4	44.0	7800.0	5.2	1	1	0	0	1	1	0	0	0	0
138.0	4.4	11.3	38.0	6000.0	4.8	1	1	0	0	0	0	0	0	0	0
138.0	4.4	9.6	31.0	7500.0	4.8	1	1	0	0	0	1	0	1	0	1
111.0	2.5	11.2	32.0	6700.0	3.9	1	0	1	0	1	0	0	1	1	1
138.0	4.4	11.6	35.0	7300.0	4.6	1	1	0	0	0	0	0	0	0	0

Conclusion

LabelEncoder and OneHotEncoder is usually need to be used together as a two steps method to encode categorical features. LabelEncoder outputs a dataframe type while OneHotEncoder outputs a numpy array. OneHotEncoder has the option to output a sparse matrix. DictVectorizer is a one step method to encode and support sparse matrix output. Pandas get dummies method is so far the most straight forward and easiest way to encode categorical features. The output will remain dataframe type.

As my point of view, the first choice method will be pandas get dummies. But if the number of categorical features are huge, DictVectorizer will be a good choice as it supports sparse matrix output.

Categorical Data Get Dummies One Hot Encoder Label Encoder Dictvectorizer