Data Science

Linear Regression

CS202 11.30.22

Categorical Data

Practice handling categorical data

We've seen some examples of categorical data. We haven't discussed too much on how to handle them using python.

We have some manual ways of taking care of categorical data individually.

One way is we can map each piece of categorical data to a numerical value using the .map( ) method.

For example: If we want to use the .map( ) method to map Sunday to 0, Monday to 1, Tuesday to 2, Wednesday to 3, Thursday to 4, Friday to 5, Saturday to 6 for a column that has days of the week as a feature.

Categorical Data

.map( ) method

We could first create a dictionary of the domain (keys) of the map to the range (values) of the map. In our example since we want to map days to numerical values we would make the names of the days into keys and the numerical values we want assigned to them as values.

```
dict = {“Sunday”:0, “Monday”:1, “Tuesday”:2, “Wednesday”:3,
“Thursday”:4, “Friday”:5, “Saturday”:6}
```

Then we use the .map( ) method by selecting the column with the names of the days, apply the .map( ) to our dictionary called dict.

```
df[“days”] = df[“days”].map(dict)
```

Categorical Data

Date and Time

If you are given a column with dates such as: 10/21/2022 depending on what you want to use for your linear regression you can isolate months, days or years by using `.DatetimeIndex()` method.

Given a data frame, call it `df` and column with dates, called “`dates`”;

```
df .DatetimeIndex( df[“dates”]).month #returns month e.g. 10
df .DatetimeIndex( df[“dates”]).day #returns day e.g. 21
df .DatetimeIndex( df[“dates”]).year #returns year e.g. 2022
```

Cateogrical Data

Binary Data

Binary data is a data consisting of 0 or 1.

For example: suppose you have a data column which describes “gender” of a person, say Male of Female. Then you can assign 0 to Male and 1 to Female using the .apply( ) method.

We can begin by writing a function distinguishing the two genders:

```
def zero_one(x):
    if x == “Male”:
        return 0
    else:
        return 1
```

Then we can use the .apply( ) to your data frame column:

```
df[“gender”].apply(zero_one)
```

Categorical Data

One-Hot encoding

We can also use a method called one-hot encoding.

The basic strategy is to convert each category value into a new column and assign a 1 or 0 (True/False) value to the column.

For example: recall, mapping Sunday - Saturday to 0-6 respectively, one-hot encoding will take each categorical value, say “Sunday”, create a column called “Sunday” and in that column will appear 0’s (False) and 1’s (True). It will do this for each day.

There are many libraries out there that support one-hot encoding but the simplest one is using panda’s .getdummies( ) method.

Categorical Data

One-Hot encoding

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,

The second being the columns argument which lets you specify the columns you want to do encoding on,

The third, the prefix argument which lets you specify the prefix for the new columns that will be created after encoding.

Categorical Data

One-Hot encoding

How do we use the .getdummies( ) method?

```
df_onehot = df.copy()
df_onehot = pd.get_dummies(df_onehot, columns=['days'],
prefix = [‘days'])
```

We don’t necessarily have to use df.copy( ) but its better to keep the original as is. pd.get_dummies( ) has the following parameters: (data (dataframe), columns = categorical column you want one-hot encoding)

Categorical Data

One-Hot encoding

Why do we want one-hot encoding?

Recall there are “numerical” values which are categorical data. For example, age, year, etc., are all categorical data. Using one-hot encoding has the benefit of not weighting a value improperly.

Sci-kit learn also supports one hot encoding via LabelBinarizer and OneHotEncoder in its preprocessing module. Just for the sake of practicing we will do the same encoding via LabelBinarizer.

Categorical Data

One-Hot encoding

```
df_onehot_sklearn = df.copy() #just copies df

#import the LabelBinarizer
from sklearn.preprocessing import LabelBinarizer

lb = LabelBinarizer()
lb_results = lb.fit_transform(df_onehot_sklearn['days'])
lb_results_df = pd.DataFrame(lb_results, columns=lb.classes_)

#lb_results will output an array of 0’s and 1’s for each category
#hence we need to concat with the original df to have the same result as
the #one-hot encoder
result_df = pd.concat([df_onehot_sklearn, lb_results_df], axis=1)
```

Preparing Data

Normally distributing and Scaling

When we prepare our data for linear regression we want to make sure that all data is normally distributed.

If they are not normally distributed, one can use the log transformation to normally distribute the data using np.log( )

We can then Scale by: for each “x” in the columns we take

$(x- x_{min})/(x_{max} - x_{min})$. This will scale all your data to make sure that they are within the same range.

Efficient Code

List comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name. Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Efficient Code

List comprehension

With list comprehension you can do all that with only one line of code:

```
fruits =
["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

The syntax is as follows:

newlist = [*expression* for *item* in *iterable* if *condition* == True]

List Comprehension

Condition

The *condition* is like a filter that only accepts the items that valuate to True. For example: Only accept items that are not “apple”:

```
newlist = [x for x in fruits if x != “apple"]
```

The condition “if x!= “apple” will return True for all elements other than "apple", making the new list contain all fruits except “apple”.
The *condition* is optional and can be omitted: With no “if” statement:

```
newlist = [x for x in fruits]
```

List Comprehension

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

You can use the range( ) function to create an iterable:

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

```
newlist = [x for x in range(10) if x < 5]
```

List Comprehension

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list: Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like: Set all values in the new list to ‘hello':

```
newlist = ['hello' for x in fruits]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome: Return "orange" instead of “banana”:

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```