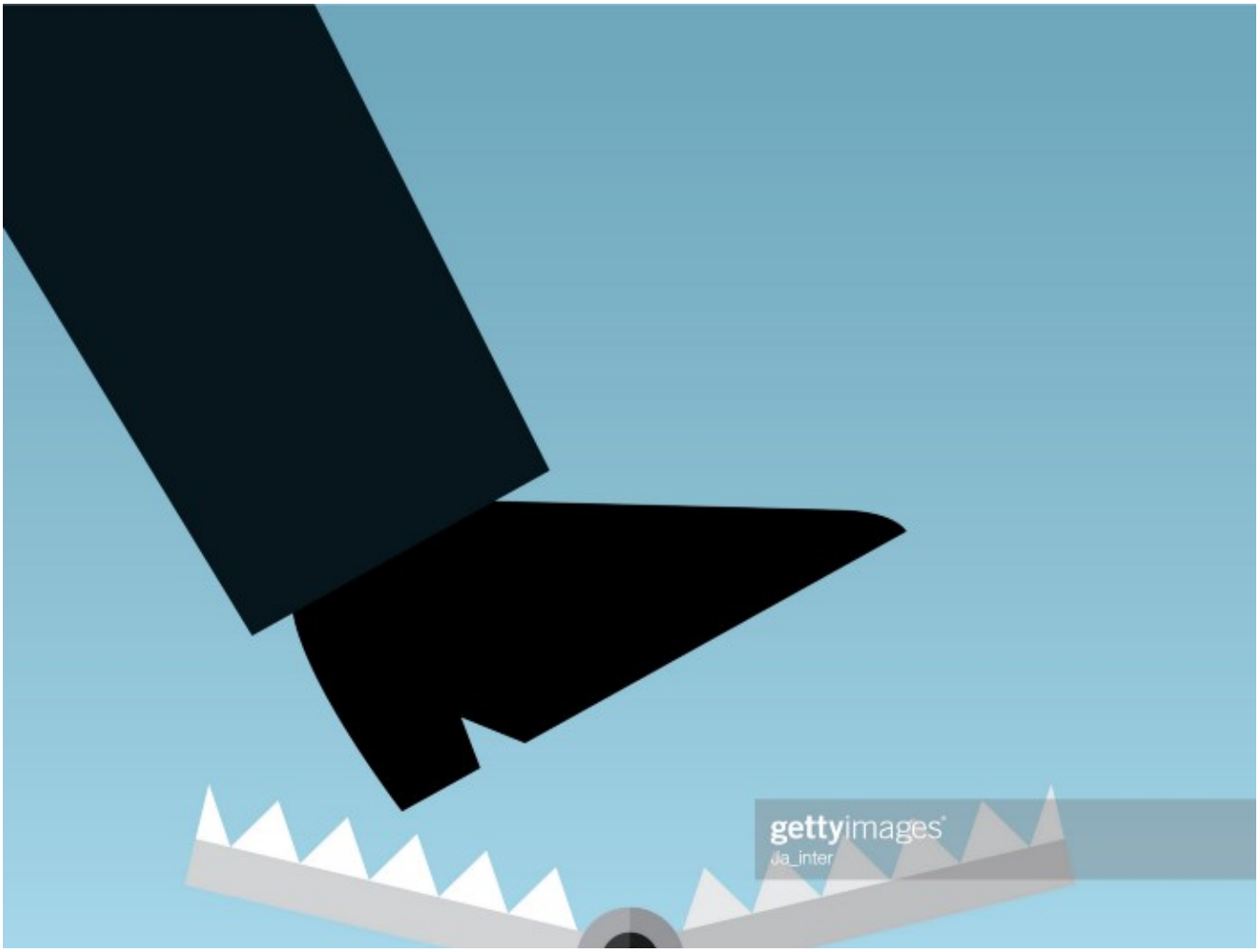
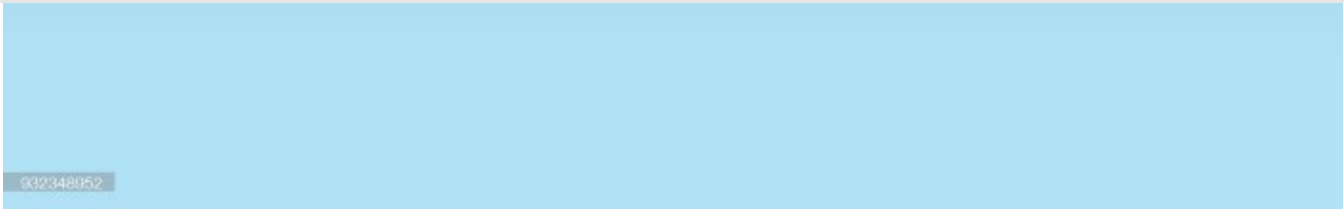
One-Hot-Encoding, Multicollinearity and the Dummy Variable Trap

This article discusses about the Dummy Variable Trap stemming from the multicollinearity problem

Krishna Kumar Mahto Jul 9, 2019 · 11 min read ★





So far, every new topic in Machine Learning I picked up had something which I had never seen before. But it wasn't only the unseen topics, but the topics which I had gone through once had something which had slipped my head. Dummy Variable Trap is one of such details that I had completely forgotten that it exists until I went back to making a classifier 3–4 months before.

Before starting with the exact content on Dummy Variable Trap, here are a few terms that we need to be familiar with. Formal definitions can be googled, so I shall put down an informal description which should be enough for the context of this article.

Categorical Variables

Categorical variables are those which can take values from a finite set. For such variables, the values that they take up can have an intrinsic ordering (for e.g., *speed: {low, medium, high}*). Such variables are called as *Ordinal Categorical Variables*. On the other hand, some categorical variables may not have any intrinsic ordering (for e.g., *Gender: {Male, Female}*). Such categorical variables are called as *Nominal Categorical Variables*.

If you are aware of the common practice of encoding categorical variables into numbers, you know that often it is suggested to get them *One-hot-encoded*. There are two reasons for that:

1. Most Machine Learning Algorithms cannot work with categorical variables directly, they need to be converted to numbers.
2. Even if we find a way to directly work with categorical variables without converting them to numbers, our model shall get biased towards the language we use. For eg, in an animal classification task, if the labels are {'rat', 'dog', 'ant'}, then using such a labelling method would train our model to predict labels only in English, which would put a linguistic restriction upon the possible applications of the model.

Case 1:

The ordinal categorical variable Speed (as quoted above) can be encoded as:

{'Low': 1, 'Medium': 2, 'High': 3}.

Case 2:

For the animal classification task (quoted above) the label variable, which is a nominal categorical variable can be encoded as:

{'rat': 1, 'dog': 2, 'ant': 3}.

> *Let's discuss case 2 first:*

There is one major issue with this- the labels in the animal classification problem should not be encoded to integers (like we have done above) since that would enforce an apparently incorrect natural ordering of: *'rat' < 'dog' < 'ant'*. While we understand no such ordering really exists and that the numbers 1, 2 and 3 do not hold any numerical ordering in the labels we have encoded, our Machine Learning model will not be able to *intuitively understand* that. If we feed these numbers directly into a model, the cost/loss function is likely to get affected by these values. We need to model this understanding of ours mathematically. One-hot-encoding is how we do it.

> *Case 1:*

Speed is an ordinal variable. We may argue that the relation: *'low'< 'medium'<'high'* makes sense and therefore, using labels 1, 2 and 3 should not be an issue. Unfortunately it is not so. Using labels as 100, 101 and 300000 in place of 1, 2 and 3 would still have the same relationship as '*low*', '*medium*' and '*high*' have. There is nothing special about using 1, 2 and 3. In other words, we do not know how greater is a speed of *'medium'* than a speed of '*low'* and how small it is compared to a *'high'* speed. Difference between these labels can potentially affect the model we train. So, we might want to one-hot-encode the variable *'speed'* as well.

> ***At this point, I hope we understand what categorical variables are all about, and why we would like to one-hot-encode them.***

Multicollinearity

can measure the degree and direction of correlation for bivariate cases (more information on measures of correlation), while multicollinearity is generally measured using Variance Inflation Factor (more information on measures of multicollinearity). In a nutshell, multicollinearity is said to exist in a dataset when the independent variables are (nearly) linearly related to each other.

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = x_i$$

Fig. 1. Perfect Multicollinearity (perfect linear dependence between x_i and other independent variables)

Cases like as shown in Fig. 1. are called Perfect Multicollinearity. Likewise, we also have cases of Imperfect Multicollinearity, in which one or more highly linear relationships may be of our concern. These directly impact the linear regression analysis (refer to these lecture notes for more information on this). However, we shall be discussing their impact from the point of view of any general Machine Learning algorithm.

> ***At this point, I hope we understand what multicollinearity is.***

OneHotEncoding in Python

Before we move on to the final part of this article, let us have a look at how we can encode categorical variables.

One-hot-encoder returns a vector for each unique value of the categorical column. Each such vector contains only one '*1*' while all other values in the vector are '0' (find fig. 4 below), so the name *one-hot*-encoder.

Probably, there should be packages in the language of your choice as well, but since I have been using Python for all Machine Learning and Data Science related stuffs, I am including snippets from Python console only. In the following discussion, I shall quote Python's *pandas* library by its popular nickname, *'pd'* and *numpy* as '*np*'.

I am using a self-made *pd.DataFrame* object that looks like:

```
>>> df
    speed  numerical
0     low      1.110
1    high      2.100
```

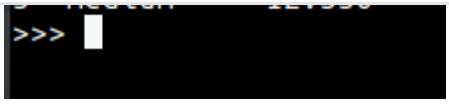


Fig. 2. An example pandas.DataFrame object

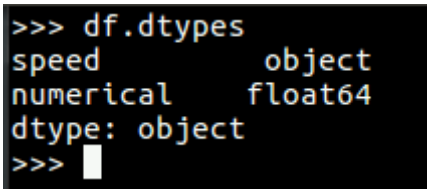


Fig. 3. Datatypes of respective columns

'*speed*' is a categorical variable, while the '*numerical*' column is a non-categorical column.

Let's assume that we are working on a project and we decide to one-hot-encode the column 'speed'.

Method 1: pd.get_dummies

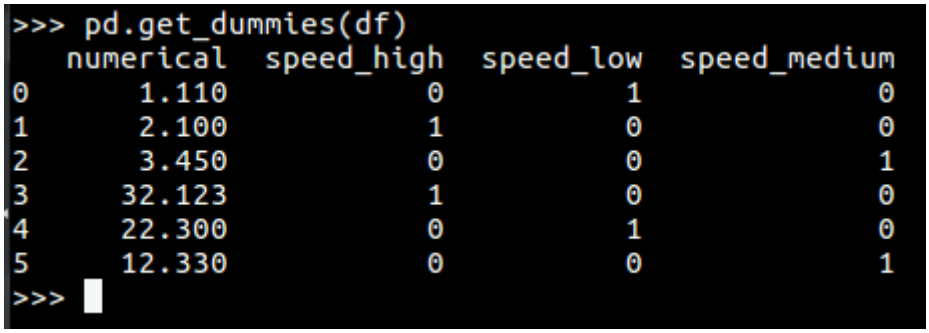


Fig. 4. pd.get_dummies returns a dataframe with one-hot-encoded columns

pd.get_dummies (documentation) returns a new dataframe that contains one-hot-encoded columns. We can observe that not all the columns were encoded. This is because, if no columns are passed to *pd.get_dummies* so as to tell which columns to one-hot-encode, by default it takes the columns with data-type *'object'*. It then encodes them and returns a new dataframe with new columns that replace the old categorical column. In fig. 3, we can see that 'speed' column is of type '*object*'. Fig. 4. shows that this column does not exist anymore in the returned dataframe and has been replaced by new columns. We can also observe that '*low*' has been mapped to a vector: *low -> [0, 1, 0]* , and similarly, *medium -> [0, 0, 1]* and *high -> [1, 0, 0]*. Notice that each vector has only one '1' in it. Also notice that each vector is 3-dimensional. This is because

the dataset. Here, encoding has been done so that 1 in the first place of a vector means '*speed=high*', 1 in the second plae means '*speed=low*' and so forth.

Method 2: sklearn.preprocessing.OneHotEncoder

I prefer using *sklearn.preprocessing.OneHotEncoder* instead of *pd.get_dummies* This is because *sklearn.preprocessing.OneHotEncoder* returns an object of *sklearn.preprocessing.OneHotEncoder* class. We can fit this object on the training set and then use the same object to transform the test set. On the other hand, *pd.get_dummies* returns a dataframe with encodings based on the values in the dataframe we pass to it. This might be good for a quick analysis, but for an extended model building project where you train on training set and will be later testing on a test set, I would suggest using *sklearn.preprocessing.OneHotEncoder*.

Using *sklearn.preprocessing.OneHotEncoder* is not as straightforward as using *pd.get_dummies*. We can see this in fig. 5 below.

```
>>> data_np
array([['low', 1.11],
       ['high', 2.1],
       ['medium', 3.45],
       ['high', 32.123],
       ['low', 22.3],
       ['medium', 12.33]], dtype=object)
>>> from sklearn.preprocessing import LabelEncoder
>>> labelencoder = LabelEncoder()
>>> labelencoder = labelencoder.fit(data_np[:, 0])
>>> data_np[:, 0] = labelencoder.transform(data_np[:, 0])
>>> data_np
array([[1, 1.11],
       [0, 2.1],
       [2, 3.45],
       [0, 32.123],
       [1, 22.3],
       [2, 12.33]], dtype=object)
>>> onehot_encoder = OneHotEncoder(categorical_features=[0])
>>> onehot_encoder = onehot_encoder.fit(data_np)
>>> data_np = onehot_encoder.transform(data_np)
>>> data_np
<6x4 sparse matrix of type '<class 'numpy.float64'>'
        with 12 stored elements in COOrdinate format>
>>> data_np = data_np.toarray()
>>> data_np
array([[ 0.   ,  1.   ,  0.   ,  1.11 ],
       [ 1.   ,  0.   ,  0.   ,  2.1  ],
       [ 0.   ,  0.   ,  1.   ,  3.45 ],
       [ 1.   ,  0.   ,  0.   , 32.123],
       [ 0.   ,  1.   ,  0.   , 22.3  ],
```

Fig. 5. One-hot encoding using sklearn.preprocessing.OneHotEncoder

You may have observed that we first did integer-encoding of categorical column using the *LabelEncoder*. This is because the *OneHotEncoder* requires the categorical columns to contain numerical labels. The *fit* as well as *transform* methods require *np.array* objects with shape *(m, n)* to be passed. Finally, *fit* method returns a *OneHotEncoder* object that is fitted on the data passed to it. The process is lengthy, but you end up with a fitted object, which can be used later on the test set. Scikit-learn comes with a combined version for the methods *fit* and *transform- fit_transform* that helps reduce a line or two from your code (see documentation).

Dummy Variable Trap

The dummy variable trap manifests itself directly from one-hot-encoding applied on categorical variables. As discussed earlier, size of one-hot vectors is equal to the number of unique values that a categorical column takes up and each such vector contains exactly one '1' in it. This ingests multicollinearity into our dataset. From the encoded dataset in fig. 4 (which is equivalent to the encoded dataset in fig. 5), we can observe the following linear relationship (fig. 6):

$$speed_{low} + speed_{medium} + speed_{high} = 1$$

Fig. 6. Perfect Multicollinearity after one-hot encoding

Fig. 6 is a case of perfect multicollinearity. The vectors that we use to encode the categorical columns are called '*Dummy Variables*'. We intended to solve the problem of using categorical variables, but got *trapped* by the problem of Multicollinearity. This is called the *Dummy Variable Trap*.

As mentioned earlier, this directly impacts the linear regression analysis because linear regression assumes non-existence of multicollinearity in the dataset. However, it also poses some other problems in Machine Learning tasks. Let us say, we train a logistic regression model on the dataset. We would expect our model to learn weights for the following equation:

Fig. 7. Sigmoid function

In particular, for the features we have in our dataset, following are the weights that logistic regression would learn:

$$w = \left(w_{speed_{low}},\ w_{speed_{medium}},\ w_{speed_{high}},\ w_{numerical} \right)$$

And the feature vector **X** is:

$$X = \left(speed_{low},\ speed_{medium},\ speed_{high} \right)$$

Clearly, it is the power on the exponential function in the denominator of the sigmoid function that actually affects the value of *y_hat* and contains trainable weights. This expression actually expands to:

$$w^T X = w_{speed_{low}} \times speed_{low} + w_{speed_{medium}} \times speed_{medium} + w_{speed_{high}} \times speed_{high} + w_{numerical} \times numerical$$

Equation- 1: Expanding the power term in sigmoid function

From the relationship in fig. 6, we can express any one of the three independent variables in terms of the other two, let us take *speed_low* in LHS and express it in terms of *speed_medium* and *speed_high:*

$$speed_{low} = 1 - speed_{medium} - speed_{high}$$

Equation-2: Expressing speed_low in terms of speed_medium and speed_high

> ***Why is it bad?***

1. We can substitute *speed_low* in equation-1 with its value in equation-2. This actually means that (atleast) one of the features we are working with is redundant- that feature could be any one of the three, since equation-2 could be written with any one of them in the LHS. So, we are making our model learn an additional weight which is not really needed. This consumes computational power and time. This also gives an optimisation objective that might not be very reasonable and might also be difficult to work with. Too many independent variables may also lead to Curse of Dimensionality. If multicollinearity also comes alongwith that, things become worse.

2. We not only want our model to predict well, but we also want it to be interpretable. For e.g., Logistic Regression is expected to learn relatively higher values for weights corresponding to relatively more important features. More important features have a greater impact on the final prediction. But if features are correlated, then it becomes hard to judge which feature has more "say" in the final decision because their values are actually dependent on one another. This affects the values of the weights. In other words, the weights not only get decided based on how an independent variable correlates to the dependent variable, they also get influenced by how independent variables correlate with one another. For e.g., if *speed_high* has higher values than others, then *speed_low* and *speed_medium* have to be lower so that the sum is always 1. Let's assume that *importance(speed_high) = importance(speed_medium) = importance(speed_low)*. But since *speed_high* has higher values than the other independent variables, the learned weight corresponding to it will be much lower than the other two. In reality, we would want their respective weights to be (almost) equal.

Since, one-hot-encoding directly induces perfect multicollinearity, we drop one of the columns from the encoded features. For e.g., we may choose to drop *speed_medium* in this case, but the choice is completely arbitrary.

Following is how we can handle this (dropping one of the encoded columns):

1. When we use *pd.get_dummies*, we can pass an additional argument, *drop_first=True* (documentation) to drop the first new column that we get after encoding (a.k.a., the first dummy variable) (fig. 7):

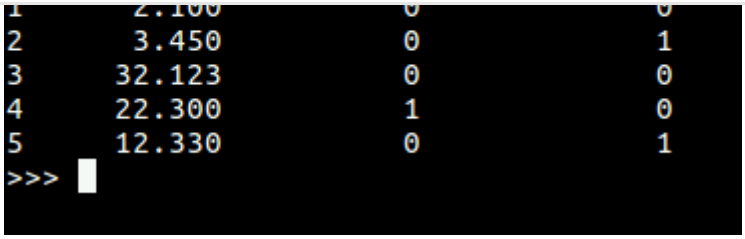


Fig. 7. drop_first=True

Compare the returned dataframe in fig. 7 with the output we got in fig. 4. We find that *speed_high* is dropped from the encoded dataframe. Not using *drop_first=True* will make sklearn use the default assignment for this parameter, which is *False*.

2. When we use *sklearn.preprocessing.OneHotEncoder* and want to drop one of the new columns, we pass the argument *drop='first'* constructor of *OneHotEncoder* class (documentation). ***However this does not seem to support all the versions of sklearn (I got an error since my version of sklearn.preprocessing.OneHotEncoder does not support the 'drop' parameter), so you may have to update the python package before you can use it on your system***. Although you can manually drop one of the dummy variables by manually writing one line or two (fig. 8), it may get difficult to keep track of the changes made to your dataframe in cases when you have more than one categorical columns and too many categories in each categorical column to work with.

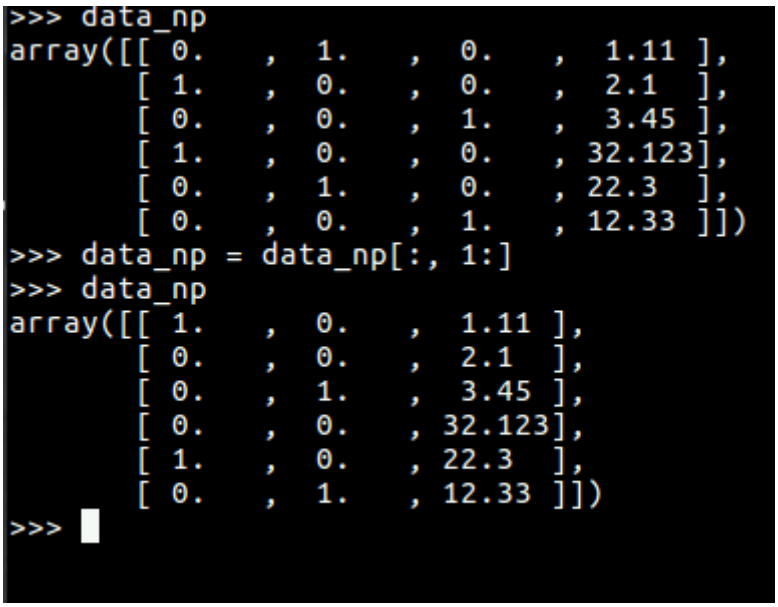


Fig. 8. Manually dropping one of the dummy variables

slicing as shown in fig. 8. However, doing this may become clumsy as already mentioned.

Conclusion

I hope this article was able to give a comprehensive description of Multicollinearity, One-Hot-Encoding and Dummy Variable Trap. The article is entirely based on my personal experience with facts taken from reliable sources (links already mentioned alongside each such concept). So, kindly comment below if you find any discrepancies in the article. Your feedback would help me write and describe better. Looking forward to hearing from you.

Thank you.