UNDERSTANDING FEATURE ENGINEERING (PART 2)

# Categorical Data

Strategies for working with discrete, categorical data

Dipanjan (DJ) Sarkar Follow
Jan 6, 2018 · 14 min read

Source: https://pixabay.com

## Introduction

We covered various feature engineering strategies for dealing with structured continuous numeric data in the *previous article in this series*. In this article, we will look at another type of structured data, which is discrete in nature and is popularly termed as categorical data. Dealing with numeric data is often easier than categorical data given that we do not have to deal with additional complexities of the semantics pertaining to each category value in any data attribute which is of a categorical type. We will use a hands-on approach to discuss several encoding schemes for dealing with categorical

data and also a couple of popular techniques for dealing with large scale feature explosion, often known as the *"curse of dimensionality"*.

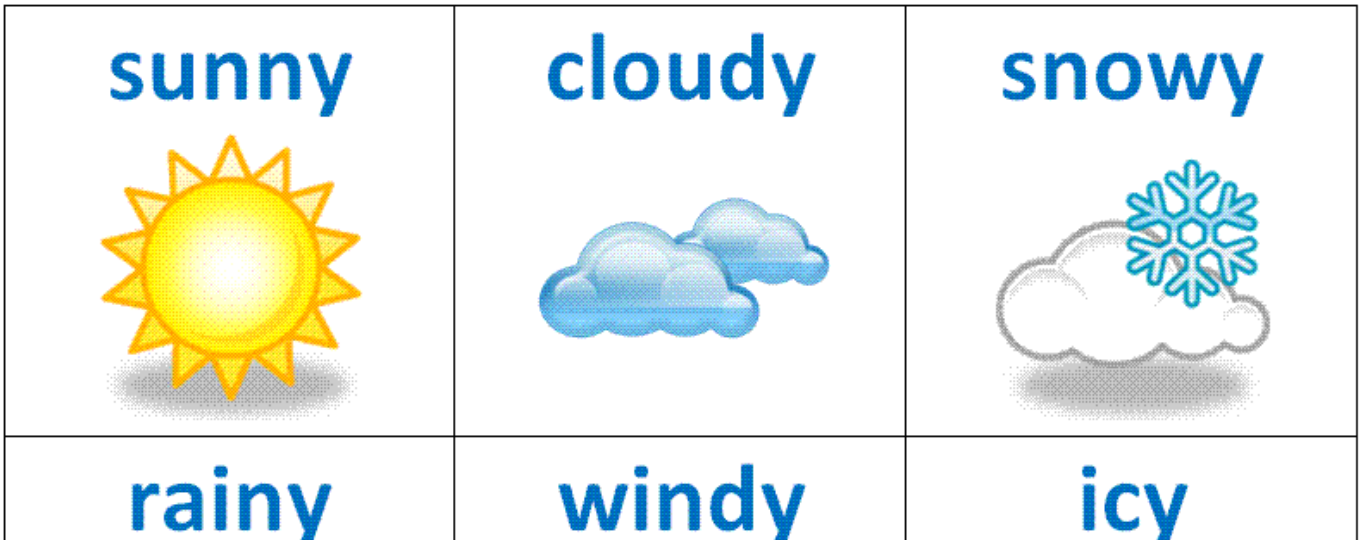
## Motivation

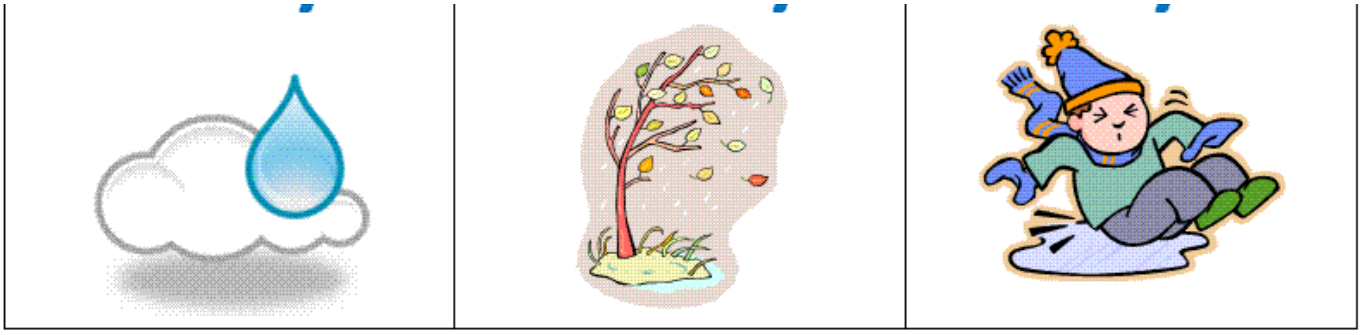
I'm sure by now you must realize the motivation and the importance of feature engineering, we do stress on the same in detail in ***'Part 1'*** of this series. Do check it out for a quick refresher if necessary. In short, machine learning algorithms cannot work directly with categorical data and you do need to do some amount of engineering and transformations on this data before you can start modeling on your data.

## Understanding Categorical Data

Let's get an idea about categorical data representations before diving into feature engineering strategies. Typically, any data attribute which is categorical in nature represents discrete values which belong to a specific finite set of categories or classes. These are also often known as classes or labels in the context of attributes or variables which are to be predicted by a model (popularly known as response variables). These discrete values can be text or numeric in nature (or even unstructured data like images!). There are two major classes of categorical data, nominal and ordinal.

In any nominal categorical data attribute, there is no concept of ordering amongst the values of that attribute. Consider a simple example of weather categories, as depicted in the following figure. We can see that we have six major classes or categories in this particular scenario without any concept or notion of order (*windy* doesn't always occur before *sunny* nor is it smaller or bigger than *sunny*).

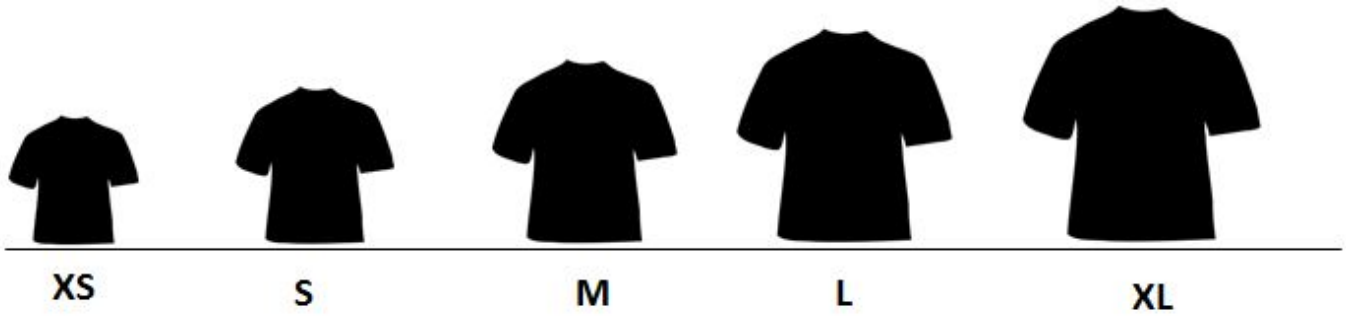




Weather as a categorical attribute

Similarly movie, music and video game genres, country names, food and cuisine types are other examples of nominal categorical attributes.

Ordinal categorical attributes have some sense or notion of order amongst its values. For instance look at the following figure for shirt sizes. It is quite evident that order or in this case *'size'* matters when thinking about shirts (***S*** is smaller than ***M*** which is smaller than ***L*** and so on).




Shirt size as an ordinal categorical attribute

Shoe sizes, education level and employment roles are some other examples of ordinal categorical attributes. Having a decent idea about categorical data, let's now look at some feature engineering strategies.

## Feature Engineering on Categorical Data

While a lot of advancements have been made in various machine learning frameworks to accept complex categorical data types like text labels. Typically any standard workflow in feature engineering involves some form of ***transformation*** of these categorical values into numeric labels and then applying some ***encoding scheme*** on these values. We load up the necessary essentials before getting started.

```
import pandas as pd
import numpy as np
```

## Transforming Nominal Attributes

Nominal attributes consist of discrete categorical values with no notion or sense of order amongst them. The idea here is to transform these attributes into a more representative numerical format which can be easily understood by downstream code and pipelines. Let's look at a new dataset pertaining to video game sales. This dataset is also available on **Kaggle** as well as in my **GitHub** repository.

```
vg_df = pd.read_csv('datasets/vgsales.csv', encoding='utf-8')
vg_df[['Name', 'Platform', 'Year', 'Genre', 'Publisher']].iloc[1:7]
```

| | Name | Platform | Year | Genre | Publisher |
|---|---|---|---|---|---|
| 1 | Super Mario Bros. | NES | 1985.0 | Platform | Nintendo |
| 2 | Mario Kart Wii | Wii | 2008.0 | Racing | Nintendo |
| 3 | Wii Sports Resort | Wii | 2009.0 | Sports | Nintendo |
| 4 | Pokemon Red/Pokemon Blue | GB | 1996.0 | Role-Playing | Nintendo |
| 5 | Tetris | GB | 1989.0 | Puzzle | Nintendo |
| 6 | New Super Mario Bros. | DS | 2006.0 | Platform | Nintendo |

Dataset for video game sales

Let's focus on the video game `Genre` attribute as depicted in the above data frame. It is quite evident that this is a nominal categorical attribute just like `Publisher` and `Platform`. We can easily get the list of unique video game genres as follows.

```
genres = np.unique(vg_df['Genre'])
genres

Output
------
array(['Action', 'Adventure', 'Fighting', 'Misc', 'Platform',
       'Puzzle', 'Racing', 'Role-Playing', 'Shooter', 'Simulation',
       'Sports', 'Strategy'], dtype=object)
```

This tells us that we have 12 distinct video game genres. We can now generate a label encoding scheme for mapping each category to a numeric value by leveraging `scikit-learn`.

```
from sklearn.preprocessing import LabelEncoder

gle = LabelEncoder()
genre_labels = gle.fit_transform(vg_df['Genre'])
genre_mappings = {index: label for index, label in
                  enumerate(gle.classes_)}
genre_mappings
```

```
Output
------
{0: 'Action', 1: 'Adventure', 2: 'Fighting', 3: 'Misc',
 4: 'Platform', 5: 'Puzzle', 6: 'Racing', 7: 'Role-Playing',
 8: 'Shooter', 9: 'Simulation', 10: 'Sports', 11: 'Strategy'}
```

Thus a mapping scheme has been generated where each genre value is mapped to a number with the help of the `LabelEncoder` object `gle`. The transformed labels are stored in the `genre_labels` value which we can write back to our data frame.

```
vg_df['GenreLabel'] = genre_labels
vg_df[['Name', 'Platform', 'Year', 'Genre', 'GenreLabel']].iloc[1:7]
```



Video game genres with their encoded labels

These labels can be used directly often especially with frameworks like `scikit-learn` if you plan to use them as response variables for prediction, however as discussed earlier, we will need an additional step of encoding on these before we can use them as features.

## Transforming Ordinal Attributes

Ordinal attributes are categorical attributes with a sense of order amongst the values. Let's consider our ***Pokémon dataset*** which we used in **Part 1** of this series. Let's focus more specifically on the `Generation` attribute.

```
poke_df = pd.read_csv('datasets/Pokemon.csv', encoding='utf-8')
poke_df = poke_df.sample(random_state=1,
                         frac=1).reset_index(drop=True)

np.unique(poke_df['Generation'])

Output
------
array(['Gen 1', 'Gen 2', 'Gen 3', 'Gen 4', 'Gen 5', 'Gen 6'],
         dtype=object)
```

Based on the above output, we can see there are a total of **6** generations and each Pokémon typically belongs to a specific generation based on the video games (when they were released) and also the television series follows a similar timeline. This attribute is typically ordinal (domain knowledge is necessary here) because most Pokémon belonging to *Generation 1* were introduced earlier in the video games and the television shows than *Generation 2* as so on. Fans can check out the following figure to remember some of the popular Pokémon of each generation (views may differ among fans!).

Popular Pokémon based on generation and type (source: https://www.reddit.com/r/pokemon/comments/2s2upx/heres_my_favorite_pokemon_by_type_and_gen_chart)

Hence they have a sense of order amongst them. In general, there is no generic module or function to map and transform these features into numeric representations based on order automatically. Hence we can use a custom encoding\mapping scheme.

```
gen_ord_map = {'Gen 1': 1, 'Gen 2': 2, 'Gen 3': 3,
               'Gen 4': 4, 'Gen 5': 5, 'Gen 6': 6}

poke_df['GenerationLabel'] = poke_df['Generation'].map(gen_ord_map)
poke_df[['Name', 'Generation', 'GenerationLabel']].iloc[4:10]
```



Pokémon generation encoding

It is quite evident from the above code that the `map(…)` function from `pandas` is quite helpful in transforming this ordinal feature.

## Encoding Categorical Attributes

If you remember what we mentioned earlier, typically feature engineering on categorical data involves a transformation process which we depicted in the previous section and a compulsory encoding process where we apply specific encoding schemes to create dummy variables or features for each category\value in a specific categorical attribute.

You might be wondering, we just converted categories to numerical labels in the previous section, why on earth do we need this now? The reason is quite simple. Considering video game genres, if we directly fed the `GenreLabel` attribute as a feature in a machine learning model, it would consider it to be a continuous numeric feature thinking value **10** (*Sports*) is greater than **6** (*Racing*) but that is meaningless because the *Sports* genre is certainly not bigger or smaller than *Racing*, these are essentially different

values or categories which cannot be compared directly. Hence we need an additional layer of encoding schemes where dummy features are created for each unique value or category out of all the distinct categories per attribute.

## One-hot Encoding Scheme

Considering we have the numeric representation of any categorical attribute with ***m*** labels (after transformation), the one-hot encoding scheme, encodes or transforms the attribute into ***m*** binary features which can only contain a value of 1 or 0. Each observation in the categorical feature is thus converted into a vector of size ***m*** with only one of the values as **1** (indicating it as active). Let's take a subset of our Pokémon dataset depicting two attributes of interest.

```
poke_df[['Name', 'Generation', 'Legendary']].iloc[4:10]
```



Subset of our Pokémon dataset

The attributes of interest are Pokémon `Generation` and their `Legendary` status. The first step is to *transform* these attributes into numeric representations based on what we learnt earlier.

```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

# transform and map pokemon generations
gen_le = LabelEncoder()
gen_labels = gen_le.fit_transform(poke_df['Generation'])
poke_df['Gen_Label'] = gen_labels

# transform and map pokemon legendary status
leg_le = LabelEncoder()
```

```
leg_labels = leg_le.fit_transform(poke_df['Legendary'])
poke_df['Lgnd_Label'] = leg_labels

poke_df_sub = poke_df[['Name', 'Generation', 'Gen_Label',
                       'Legendary', 'Lgnd_Label']]
poke_df_sub.iloc[4:10]
```



Attributes with transformed (numeric) labels

The features `Gen_Label` and `Lgnd_Label` now depict the numeric representations of our categorical features. Let's now apply the one-hot encoding scheme on these features.

```
# encode generation labels using one-hot encoding scheme
gen_ohe = OneHotEncoder()
gen_feature_arr = gen_ohe.fit_transform(
                              poke_df[['Gen_Label']]).toarray()
gen_feature_labels = list(gen_le.classes_)
gen_features = pd.DataFrame(gen_feature_arr,
                            columns=gen_feature_labels)

# encode legendary status labels using one-hot encoding scheme
leg_ohe = OneHotEncoder()
leg_feature_arr = leg_ohe.fit_transform(
                                poke_df[['Lgnd_Label']]).toarray()
leg_feature_labels = ['Legendary_'+str(cls_label)
                          for cls_label in leg_le.classes_]
leg_features = pd.DataFrame(leg_feature_arr,
                            columns=leg_feature_labels)
```

In general, you can always encode both the features together using the `fit_transform(…)` function by passing it a two dimensional array of the two features together (Check out the documentation!). But we encode each feature separately, to make things easier to

understand. Besides this, we can also create separate data frames and label them accordingly. Let's now concatenate these feature frames and see the final result.

```
poke_df_ohe = pd.concat([poke_df_sub, gen_features, leg_features],
axis=1)
columns = sum([['Name', 'Generation', 'Gen_Label'],
               gen_feature_labels, ['Legendary', 'Lgnd_Label'],
               leg_feature_labels], [])
poke_df_ohe[columns].iloc[4:10]
```



One-hot encoded features for Pokémon generation and legendary status

Thus you can see that **6** dummy variables or binary features have been created for `Generation` and **2** for `Legendary` since those are the total number of distinct categories in each of these attributes respectively. ***Active*** state of a category is indicated by the **1** value in one of these dummy variables which is quite evident from the above data frame.

Consider you built this encoding scheme on your training data and built some model and now you have some new data which has to be engineered for features before predictions as follows.

```
new_poke_df = pd.DataFrame([['PikaZoom', 'Gen 3', True],
                            ['CharMyToast', 'Gen 4', False]],
                      columns=['Name', 'Generation', 'Legendary'])
new_poke_df
```



Sample new data

You can leverage `scikit-learn's` excellent API here by calling the `transform(…)` function of the previously build `LabeLEncoder` and `OneHotEncoder` objects on the new data. Remember our workflow, first we do the ***transformation***.

```
new_gen_labels = gen_le.transform(new_poke_df['Generation'])
new_poke_df['Gen_Label'] = new_gen_labels

new_leg_labels = leg_le.transform(new_poke_df['Legendary'])
new_poke_df['Lgnd_Label'] = new_leg_labels

new_poke_df[['Name', 'Generation', 'Gen_Label', 'Legendary',
             'Lgnd_Label']]
```



Categorical attributes after transformation

Once we have numerical labels, let's apply the encoding scheme now!

```
new_gen_feature_arr =
gen_ohe.transform(new_poke_df[['Gen_Label']]).toarray()
new_gen_features = pd.DataFrame(new_gen_feature_arr,
                                columns=gen_feature_labels)

new_leg_feature_arr =
leg_ohe.transform(new_poke_df[['Lgnd_Label']]).toarray()
new_leg_features = pd.DataFrame(new_leg_feature_arr,
                                columns=leg_feature_labels)

new_poke_ohe = pd.concat([new_poke_df, new_gen_features,
new_leg_features], axis=1)
columns = sum([['Name', 'Generation', 'Gen_Label'],
               gen_feature_labels,
               ['Legendary', 'Lgnd_Label'], leg_feature_labels], [])

new_poke_ohe[columns]
```

Categorical attributes after one-hot encoding

Thus you can see it's quite easy to apply this scheme on new data easily by leveraging `scikit-learn's` powerful API.

You can also apply the one-hot encoding scheme easily by leveraging the `to_dummies(…)` function from `pandas`.

```
gen_onehot_features = pd.get_dummies(poke_df['Generation'])
pd.concat([poke_df[['Name', 'Generation']], gen_onehot_features],
          axis=1).iloc[4:10]
```



One-hot encoded features by leveraging pandas

The above data frame depicts the one-hot encoding scheme applied on the `Generation` attribute and the results are same as compared to the earlier results as expected.

## Dummy Coding Scheme

The dummy coding scheme is similar to the one-hot encoding scheme, except in the case of dummy coding scheme, when applied on a categorical feature with **m** distinct labels, we get ***m - 1*** binary features. Thus each value of the categorical variable gets converted into a vector of size ***m - 1***. The extra feature is completely disregarded and thus if the category values range from ***{0, 1, …, m-1}*** the ***0th*** or the ***m - 1th*** feature column is dropped and corresponding category values are usually represented by a vector of all zeros ***(0)***. Let's try applying dummy coding scheme on Pokémon `Generation` by dropping the first level binary encoded feature (`Gen 1`).

```
gen_dummy_features = pd.get_dummies(poke_df['Generation'],
                                    drop_first=True)
pd.concat([poke_df[['Name', 'Generation']], gen_dummy_features],
          axis=1).iloc[4:10]
```



Dummy coded features for Pokémon g `eneration`

If you want, you can also choose to drop the last level binary encoded feature ( `Gen 6` ) as follows.

```
gen_onehot_features = pd.get_dummies(poke_df['Generation'])
gen_dummy_features = gen_onehot_features.iloc[:,:-1]
pd.concat([poke_df[['Name', 'Generation']], gen_dummy_features],
          axis=1).iloc[4:10]
```



Dummy coded features for Pokémon g `eneration`

Based on the above depictions, it is quite clear that categories belonging to the dropped feature are represented as a vector of zeros (**0)** like we discussed earlier.

## Effect Coding Scheme

The effect coding scheme is actually very similar to the dummy coding scheme, except during the encoding process, the encoded features or feature vector, for the category values which represent all **0** in the dummy coding scheme, is replaced by **-1** in the effect coding scheme. This will become clearer with the following example.

```
gen_onehot_features = pd.get_dummies(poke_df['Generation'])
gen_effect_features = gen_onehot_features.iloc[:,:-1]
gen_effect_features.loc[np.all(gen_effect_features == 0,
                               axis=1)] = -1.
pd.concat([poke_df[['Name', 'Generation']], gen_effect_features],
          axis=1).iloc[4:10]
```



Effect coded features for Pokémon g `eneration`

The above output clearly shows that the Pokémon belonging to `Generation` 6 are now represented by a vector of -1 values as compared to zeros in dummy coding.

## Bin-counting Scheme

The encoding schemes we discussed so far, work quite well on categorical data in general, but they start causing problems when the number of distinct categories in any feature becomes very large. Essential for any categorical feature of ***m*** distinct labels, you get ***m*** separate features. This can easily increase the size of the feature set causing problems like storage issues, model training problems with regard to time, space and memory. Besides this, we also have to deal with what is popularly known as the ***'curse of dimensionality'*** where basically with an enormous number of features and not enough representative samples, model performance starts getting affected often leading to overfitting.

Hence we need to look towards other categorical data feature engineering schemes for features having a large number of possible categories (like IP addresses). The bin-counting scheme is a useful scheme for dealing with categorical variables having many categories. In this scheme, instead of using the actual label values for encoding, we use probability based statistical information about the value and the actual target or response value which we aim to predict in our modeling efforts. A simple example would be based on past historical data for IP addresses and the ones which were used in DDOS attacks; we can build probability values for a DDOS attack being caused by any of the IP addresses. Using this information, we can encode an input feature which depicts that if the same IP address comes in the future, what is the probability value of a DDOS attack being caused. This scheme needs historical data as a pre-requisite and is an elaborate one. Depicting this with a complete example would be currently difficult here but there are several resources online which you can refer to for the same.

## Feature Hashing Scheme

The feature hashing scheme is another useful feature engineering scheme for dealing with large scale categorical features. In this scheme, a hash function is typically used with the number of encoded features pre-set (as a vector of pre-defined length) such that the hashed values of the features are used as indices in this pre-defined vector and values are updated accordingly. Since a hash function maps a large number of values into a small finite set of values, multiple different values might create the same hash which is termed as collisions. Typically, a signed hash function is used so that the sign of the value obtained from the hash is used as the sign of the value which is stored in the final feature vector at the appropriate index. This should ensure lesser collisions and lesser accumulation of error due to collisions.

Hashing schemes work on strings, numbers and other structures like vectors. You can think of hashed outputs as a finite set of ***b*** bins such that when hash function is applied on the same values\categories, they get assigned to the same bin (or subset of bins) out of the ***b*** bins based on the hash value. We can pre-define the value of ***b*** which becomes the final size of the encoded feature vector for each categorical attribute that we encode using the feature hashing scheme.

Thus even if we have over **1000** distinct categories in a feature and we set ***b=10*** as the final feature vector size, the output feature set will still have only **10** features as compared to **1000** binary features if we used a one-hot encoding scheme. Let's consider the `Genre` attribute in our video game dataset.

```
unique_genres = np.unique(vg_df[['Genre']])
print("Total game genres:", len(unique_genres))
print(unique_genres)

Output
------
Total game genres: 12
['Action' 'Adventure' 'Fighting' 'Misc' 'Platform' 'Puzzle' 'Racing'
 'Role-Playing' 'Shooter' 'Simulation' 'Sports' 'Strategy']
```

We can see that there are a total of 12 genres of video games. If we used a one-hot encoding scheme on the `Genre` feature, we would end up having 12 binary features. Instead, we will now use a feature hashing scheme by leveraging `scikit-learn's FeatureHasher` class, which uses a signed 32-bit version of the *Murmurhash3* hash function. We will pre-define the final feature vector size to be **6** in this case.

```
from sklearn.feature_extraction import FeatureHasher

fh = FeatureHasher(n_features=6, input_type='string')
hashed_features = fh.fit_transform(vg_df['Genre'])
hashed_features = hashed_features.toarray()
pd.concat([vg_df[['Name', 'Genre']], pd.DataFrame(hashed_features)],
          axis=1).iloc[1:7]
```

Feature Hashing on the Genre attribute

Based on the above output, the `Genre` categorical attribute has been encoded using the hashing scheme into **6** features instead of **12**. We can also see that rows **1** and **6** denote the same genre of games, ***Platform*** which have been rightly encoded into the same feature vector.

## Conclusion

These examples should give you a good idea about popular strategies for feature engineering on discrete, categorical data. If you read **Part 1** of this series, you would have seen that it is slightly challenging to work with categorical data as compared to continuous, numeric data but definitely interesting! We also talked about some ways to handle large feature spaces using feature engineering but you should also remember that there are other techniques including *feature selection* and *dimensionality reduction* methods to handle large feature spaces. We will cover some of these methods in a later article.

Next up will be feature engineering strategies for unstructured text data. Stay tuned!

. . .

To read about feature engineering strategies for continuous numeric data, check out **Part 1** of this series!

All the code and datasets used in this article can be accessed from my **GitHub**

The code is also available as a **Jupyter notebook**

Thanks to Ludovic Benistant.