Code.Hub

Week 2 exercise

You will be given another csv file, this time containing both numeric and categorical values. The goal of this exercise is to extend the class you created in week 1, in order to be able to process this data.

For example a the table we you will receive could look like this:

38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female

Our final goal is to represent the data in a numpy array. Even though numpy does support strings, representing the data as an array of strings would not be effective [1]. A better option would be to convert the string values to numbers and store the table as a numeric numpy array. The procedure of converting a column with categorical values to numeric is called **encoding** and will be discussed in future lessons in more detail.

Note that this process needs to be done **for each column separately**.

Encoding

We'll examine 3 different strategies for encoding.

- Simple mapping

Here we just map each **unique** value of the column to an integer (e.g for the 8th column of the table we could define the mapping "not-in-family" → 0, "husband" → 1, "wife" → 2). Then we replace the values with their mapping.

For example:

original values	**encoded values**
Not-in-family	0
Husband	1
Wife	2
Wife	2
Husband	1
...	...

- One-hot encoding

This strategy involves creating **a new column** for each unique value of the original column. Each one of these new columns **corresponds to one of the values** of the original column. In the previous example we'd have one column for "not-in-family", one for "husband" and one for "wife". For each row of the column we want to encode, **we set its corresponding column to 1 and all the rest to zero**.

The result looks like this:

original values	**encoded values**		
	Not-in-family	Husband	Wife
Not-in-family	1	0	0
Husband	0	1	0
Wife	0	0	1
Wife	0	0	1
Husband	0	1	0
...	...	...	...

Note: the encoding of each value is referred to as the **one-hot vector**. E.g. the vector for "husband" is [0, 1, 0].

- Binary encoding

The final strategy is similar to one-hot encoding, but rather than creating a one-hot vector with multiple zeros and only one 1, we create the vector with the **binary representation** of its mapping.

E.g. “wife”, which has a mapping of 2 would have a binary-encoding vector of [1, 0], because 10 is the binary representation of 2.

original values	**encoded values**	
	b1	b0
Not-in-family	0	0
Husband	0	1
Wife	1	0
Wife	1	0
Husband	0	1
...	...	...

Note: the number of columns required to encode the values obviously depends on the number of unique values in the original column. E.g. if we had 5 unique values in the original column, we’d need 3 columns to represent it. [2]

Scaling

Another operation we want to perform on the columns is to scale their values. We’ll examine two different strategies for this as well.

Note: each column is scaled independently!

- Normalization

This involves subtracting the minimum value of the column from each of the original values and dividing this with the range of the column. I.e. if x_i are the original values of the column x , then its normalized value y_i can be calculated like this:

$$y_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

where $\min(x)$ and $\max(x)$ are the minimum and maximum values of the original column x .

- ## Standardization

This involves subtracting the mean of the column from each of the original values and dividing by the standard deviation of the column. I.e. if x_i are the original values of the column x , then its standardized value z_i can be calculated like this:

$$z_i = \frac{x_i - \mu}{\sigma}$$

where μ and σ are the minimum and maximum values of the original column x .

In this exercise we'll attempt to craft these preprocessing steps on our own and apply them to the given dataset. For this reason, **the use of any external library besides numpy is prohibited**.

The tasks are the following:

1. Create a function that takes a column with categorical values and returns an encoded version, just by **mapping** its unique values to integers.
    - This can be done either with built-in python functions or with the help of numpy.

   [Bonus 1]: Write a function for one-hot encoding.
   [Bonus 2]: Write a function for binary encoding.

2. Now that we have a way of encoding columns, extend the class of the previous exercise to **identify** if a column is numerical or categorical and if it is categorical, **encode** it. Finally, store the data into a **numpy array**.
    - This is easier to do with the list-of-lists representation of the previous exercise.
    - Hint: the built-in string method `string.isdigit()` might be of use.
    - You may need to create a way to add new columns to the representation.
    - Store the resulting numpy array into an instance variable.

   [Bonus 1]: Rewrite the methods that compute the statistics so that they work with numpy arrays instead of the list-of-lists format of the previous exercise.
   [Bonus 2]: Incorporate all this functionality into the class (methods not functions).
   [Bonus 3]: Add the other two encoding options into the class. Add an argument to the class's `__init__()` to control which encoding strategy to follow. E.g. `dset = Dataset(data_path, encoder='one_hot')` to use the one-hot encoder.

3. Answer the following questions.
    - Since strings can be used to represent both characters and numbers, why isn't it effective to store the data as an array of strings? [1]

    - How many columns are needed encode a column with each N unique values, with each of the 3 encoding strategies? [2]

4. Create a method for scaling the columns of the dataset.
    - Write a method for **normalizing** the columns of the dataset.

[Bonus 1]: Write the method for standardizing the columns. Add an argument to the class's `__init__()` to control which scaling strategy to use. E.g. `dset = Dataset(data_path, scaling='standardization')` to standardize the columns.