Data Pipelines with Column Selection

Module 2 | Chapter 3 | Notebook 3

In this notebook you will learn how to use the ColumnTransformer to perform `Pipeline` steps only using selected columns. By the end of this notebook you will be able to pack different parts of the data cleaning and feature engineering processes into a pipeline. The goal is to perform as many steps as possible automatically with `sklearn`, which you would otherwise have to do one after the other with `pandas`.

Scenario: You work for an international global logistics company. Due to the tense situation on the labor market, the company is finding it increasingly difficult to attract new talent. For this reason, management has decided to try and limit the number of employees who leave. You should predict which employees are likely to want to leave the company so that measures can be taken to encourage them to stay.

You can find the data about employees who leave in *attrition_train.csv*. You've also been provided test data in the file *attrition_test.csv*.

Simply run the following code cell to import the new test data. The cell:

- Import the file and store it as a `DataFrame` named `df_train`.
- Then `df_train` is split into `features_train` and `target_train` (the `'attrition'` column of `'df_train'`).
- The cell then proceeds in the same way with the test data.

In [1]:
```
# module import
import pandas as pd

# data gathering
df_train = pd.read_csv('attrition_train.csv')
df_test = pd.read_csv('attrition_test.csv')

#extract features and target
features_train = df_train.drop('attrition', axis=1)
target_train = df_train.loc[:,'attrition']

features_test = df_test.drop('attrition', axis=1)
target_test = df_test.loc[:,'attrition']

# look at raw data
df_train.head()
```

Out[1]:

	attrition	age	gender	businesstravel	distancefromhome	education	joblevel	maritalstatus	mont
0	0	30	0	1	5.0	3	2	1	
1	1	33	0	1	5.0	3	1	0	
2	0	45	1	1	24.0	4	1	0	
3	0	28	1	1	15.0	2	1	1	
4	0	30	1	1	1.0	3	1	2	

5 rows × 21 columns

The code for that looks like this:

Column number	Column name	Type	Description
0	`'attrition'`	categorical	Whether the employee has left the company (`1`) or not (`0`)
1	`age`	continuous (`int`)	The person's age in years
2	`'gender'`	categorical (nominal, `int`)	Gender: male (`1`) or female (`0`)
3	`'businesstravel'`	categorical (ordinal, `int`)	How often the employee is on a business trip: often (`2`), rarely (`1`) or never (`0`)
4	`'distancefromhome'`	continuous (`int`)	Distance from home address to work address in kilometers
5	`'education'`	categorical (ordinal, `int`)	Level of education: doctorate (`5`), master (`4`), bachelor (`3`), apprenticeship(`2`), Secondary school qualifications (`1`)
6	`'joblevel'`	categorical (ordinal, `int`)	Level of responsibility: Executive (`5`), Manager (`4`), Team leader (`3`), Senior employee (`2`), Junior employee (`1`)
7	`'gender'`	categorical (nominal, `int`)	Marital status: married (`2`), divorced (`1`), single (`0`)
8	`'monthlyincome'`	continuous (`int`)	Gross monthly salary in EUR
9	`'numcompaniesworked'`	continuous (`int`)	The number of enterprises where the employee worked before their current position
10	`'over18'`	categorical (`int`)	Whether the employee is over 18 years of age (`1`) or not (`0`)
11	`'overtime'`	categorically (`int`)	Whether or not they have accumulated overtime in the past year (`1`) or not (`0`)
12	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months

Column number	Column name	Type	Description
13	`'standardhours'`	continuous (`int`)	contractual working hours per two weeks
14	`'stock option levels'`	categorical (ordinal, `int`)	options on company shares: very many (`4`), many (`3`), few (`2`), very little (`1`), none (`0`)
15	`'trainingtimeslastyear'`	continuous (`int`)	Number of training courses taken in the last 12 months
16	`'totalworkingyears'`	continuous (`int`)	Number of years worked: Number of years on the job market and as an employee
17	`'years_atcompany'`	continuous (`int`)	Number of years at the current company Number of years in the current company
18	`'years_currentrole'`	continuous (`int`)	Number of years in the current position
19	`'years_lastpromotion'`	continuous (`int`)	Number of years since the last promotion
20	`'years_withmanager'`	continuous (`int`)	Number of years working with current manager

Each row in `df_train` represents an employee

The EDA in the last notebook showed that the following columns are strongly correlated. Run the next cell to import the data and save yourself some typing.

In [2]:
```
col_correlated = ['totalworkingyears',
                  'years_atcompany',
                  'years_currentrole',
                  'years_lastpromotion',
                  'years_withmanager']
```

Create a pipeline that handles features differently

In the last notebook you successfully cleaned the data from `features_train` "by hand" and followed the following steps to the goal:

1. Carry out a PCA on all the columns in `col_correlated` with `std_pca` .
2. Append the result columns of `std_pca` to the `DataFrame` .
3. Delete `col_correlated` and `['over18', 'standardhours']` .

Now we can copy the same steps again and apply them to `features_test` and all the other data sets our model is supposed to make predicttions for in the future. If our model goes into production later, copying these steps would no longer be a viable way of doing things. This is why data scientists and data engineers use pipelines for this. For us as data scientists, pipelines have the advantage that we only have to define our steps once and that the data is always processed in the same way. In addition, a well-designed pipeline minimizes the risk that we accidentally give information such as mean values or standard deviations from the test set to our model. Remember: **Only ever fit to the training set!** Pipelines make this easy for us. You used a `Pipeline` in the last notebook - `std_pca`:

In [3]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

std_pca = Pipeline([('std', StandardScaler()),
                    ('pca', PCA(n_components=0.8))])
```

But so far, we've always applied pipelines to the entire dataset by just using the `my_pipe.transform()` method. However, we only want to apply our PCA to `col_correlated`. `sklearn` offers us the `ColumnTransformer` from the `sklearn.compose` module, which makes this possible. Import it directly.

In [4]:
```
from sklearn.compose import ColumnTransformer
```

We can pass `ColumnTransformer()` a `list` containing tuples like `Pipeline()`, which need a name in the as a `str` and a transformer. This tuple should then also contain a `list` of column names or index numbers which we want to apply the transformer to:

```
ColumnTransformer(transformers=list, # List of Tuples like
('step_name',transformer,[columns to apply transformer to])
                  remainder=str, # Strategy of what to do with remainder;
                                 # - 'drop': delete,
                                 # - 'passthrough': append to output,
                                 # - transformer: pipe to another
transformer and append result
                  n_jobs=int)                 # number of CPU cores to
use
                 )
```

Use `ColumnTransformer()` to apply `std_pca` in the `'pca'` step to the columns in `col_correlated`. Name the object whatever you want - we're just using it for a demonstration. Then apply the `.fit_transform()` method to `features_test` and examine the *shape* of the output.

In [5]:
```
colTransformer = ColumnTransformer([('pca', std_pca, col_correlated)])
colTransformer.fit_transform(features_train)
df_ColTrans = colTransformer.transform(features_test)
df_ColTrans.shape
```

Out[5]:
```
(441, 2)
```

I get a Numpy array with 441 rows and 2 columns. This corresponds to `'pca_years_0'` and `'pca_years_1'` and we have completed the first point of our procedure "Carry out a PCA on all the columns in `col_correlated` with `std_pca`". However, all other columns of `df_train` got lost, because the `remainder` parameter is set to the `'drop'` by default.

We can also pass another *transformer2* to `remainder`, which is then applied to the remaining columns. The result of *transformer2* is then tagged onto the result of our actual *transformer*. So if we can define and pass *transformer2*, it will tick off the second point "Append the resulting columns of `std_pca` to the `DataFrame`.

Now let's consider what function our *transformer2* should have. This should keep all the columns except those in `col_correlated` and `['over18', 'standardhours']` and output them without changing them. The *remainder* then has to be discarded accordingly. So we can write a pipeline step that filters out cells.

To define a pipeline step that simply returns the input data without changing it, you pass a tuple to the list of transformers, with the string `'passthrough'`, e.g. `('do_nothing_step', 'passthrough')` as its second entry. This works in both the regular `Pipeline` and the `ColumnTransformer`.

Execute the next code cell to define all the columns that should be retained by the second transformer:

In [6]:
```
keep_cols = ['age',
             'gender',
             'businesstravel',
             'distancefromhome',
             'education',
             'joblevel',
             'maritalstatus',
             'monthlyincome',
             'numcompaniesworked',
             'overtime',
             'percentsalaryhike',
             'stockoptionlevels',
             'trainingtimeslastyear']
```

Now create a ColumnTransformer with the following functionality: "*Delete all columns except* `keep_cols`"

Assign this object to the variable `col_dropper`. Name the pipeline step you define in `col_dropper` `"drop_unused_cols"`.

In [7]:
```
col_dropper = ColumnTransformer([('drop_unused_cols', 'passthrough', keep_cols)])
```

Now let's check if `col_dropper` has the functionality we want. Are all the values the same once we use pandas to select columns, compared to when we use `col_dropper`? Run the next cell. If all the values are equal, as we expect, the output should be `True`.

In [8]:
```
# Check if all the values are the same
col_dropper.fit_transform(features_train)
(features_test[keep_cols].values == col_dropper.transform(features_test)).all()
```

Out[8]:
```
True
```

Now we can put our pipeline steps together.

The finished pipeline should then correspond to the following pattern:

Define the final *ColumnTransformer* named `corr_transformer` exactly as you defined your first `ColumnTransformer`, except that this time you pass the `remainder` parameter to the `col_dropper` transformer.

In [9]:
```
corr_transformer = ColumnTransformer([('pipe_std_pca_corrcols', std_pca, col_correlate
```

To test `corr_transformer`, apply `.fit_transform()` from `corr_transformer` to `features_train` and name the output `piped_out_arr`. Then check the *shape* of `piped_out_arr`. Does it match the *shape* of `features_train` that we transformed by hand in the last notebook? The shape then was `(1029, 15)`.

In [10]:
```
print("Manual:  (1029, 15)")
piped_out_arr = corr_transformer.fit_transform(features_train)
print("Piped : ", piped_out_arr.shape)
```

```
Manual:  (1029, 15)
Piped :  (1029, 15)
```

You should get the same number of rows and columns as you got for the `features_train` `DataFrame` you transformed yourself from the last notebook - so you should get `(1029, 15)` again.

We can pass the output of `corr_transformer` directly to a prediction model by expanding our pipeline accordingly. However, things become difficult if we want to analyze the data further, because pandas DataFrames are much better suited to this than the numpy array that is output. We should therefore convert our output into a `DataFrame`.

For this we still need a table with the column names. Since we performed the PCA first, the `'pca_years_0'` and `'pca_years_1'` columns are in the first two columns of the pipeline output. All other columns are in the same order as in `keep_cols`.

First create a *list* called `col_names` where you store all column names. Then create a DataFrame from `piped_out_arr`, whose columns you rename to `col_names`. Call this `DataFrame` `output`.

In [11]:
```
col_names = ['pca_years_0','pca_years_1'] + keep_cols
output = pd.DataFrame(piped_out_arr, columns=col_names)
output.head()
```

Out[11]:

	pca_years_0	pca_years_1	age	gender	businesstravel	distancefromhome	education	joblevel	ma
0	0.385171	-0.156575	30.0	0.0	1.0	5.0	3.0	2.0	
1	-2.348248	-0.406330	33.0	0.0	1.0	5.0	3.0	1.0	
2	-0.781200	-0.233330	45.0	1.0	1.0	24.0	4.0	1.0	
3	-1.181156	-0.535303	28.0	1.0	1.0	15.0	2.0	1.0	
4	-1.447056	0.019780	30.0	1.0	1.0	1.0	3.0	1.0	

So to summarize, we took the following steps to build our pipeline:

In [12]:

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

col_correlated = ['totalworkingyears',
                  'years_atcompany',
                  'years_currentrole',
                  'years_lastpromotion',
                  'years_withmanager']

keep_cols = ['age',
             'gender',
             'businesstravel',
             'distancefromhome',
             'education',
             'joblevel',
             'maritalstatus',
             'monthlyincome',
             'numcompaniesworked',
             'overtime',
             'percentsalaryhike',
             'stockoptionlevels',
             'trainingtimeslastyear']

std_pca = Pipeline([('std', StandardScaler()),
                    ('pca', PCA(n_components=0.8))])

col_dropper = ColumnTransformer([('drop_unused_cols', 'passthrough', keep_cols)],
                                remainder='drop')

corr_transformer = ColumnTransformer([('pipe_std_pca_corrcols', std_pca, col_correlate
                                     remainder=col_dropper)

col_names = ['pca_years_0','pca_years_1'] + keep_cols
```

Run the following code cell to test the pipeline on the uncleaned data set. Keep the golden rule in mind: **Only ever fit to the training set!** We can then apply the `.transform()` method to the test set and all the other data sets.

In [15]:

```
#Load data
df_train = pd.read_csv('attrition_train.csv')
```

```python
df_test = pd.read_csv('attrition_test.csv')

#split into features
features_train = df_train.drop('attrition', axis=1)
features_test = df_test.drop('attrition', axis=1)

#clean data
corr_transformer.fit_transform(features_train)
pd.DataFrame(corr_transformer.transform(features_test), columns=col_names)
```

Out[15]:

	pca_years_0	pca_years_1	age	gender	businesstravel	distancefromhome	education	joblevel
0	1.278545	-1.011343	36.0	1.0	0.0	10.0	4.0	3.0
1	-1.173493	-0.238435	33.0	1.0	1.0	25.0	3.0	2.0
2	-1.003990	-0.545141	35.0	0.0	2.0	18.0	4.0	2.0
3	1.933813	-0.922284	40.0	1.0	1.0	20.0	4.0	3.0
4	-2.221669	-0.527632	29.0	1.0	2.0	24.0	2.0	1.0
...	...	...	...	...	...	...	...	...
436	1.653629	0.456788	36.0	1.0	0.0	18.0	4.0	2.0
437	-1.054577	-0.656605	31.0	1.0	1.0	1.0	2.0	1.0
438	-1.257373	1.652822	40.0	1.0	0.0	28.0	3.0	3.0
439	0.377438	2.139758	52.0	1.0	1.0	3.0	3.0	3.0
440	1.736067	-0.213745	33.0	1.0	1.0	7.0	3.0	3.0

441 rows × 15 columns

Your test data should now look like this:

Now we only have to save our fitted pipeline and the corresponding column names as a *pickle* so that we can use it in the next notebooks. To do this, import `pickle` and save `corr_transformer` as *pipeline.p* and `col_names` as *col_names.p*

Tip: Use `pickle.dump(my_object, open("my_filename.p","wb")` to create a *pickle* file.

In [16]:

```python
import pickle
pickle.dump(corr_transformer, open("pipeline.p","wb"))
pickle.dump(col_names, open("col_names.p","wb"))
```

Congratulations: You have successfully prepared the data of this chapter and used knowledge from previous chapters. So now we can turn to a new classification algorithm: Decision trees.

Remember:

- With `ColumnTransformer` you can apply *transformer* to selected columns.

- If you use the string `'passthrough'` instead of a *transformer* when defining an `sklearn` pipeline, you can output data exactly as it was output.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
