OOP Guidance 10 - Portfolio BOTH Imputer and Factory

July 24, 2023

1 Guidance 10 - Portfolio Parts 1 & 2

This notebook contains both parts of the mandatory portfolio assignment. Part 1 has been previosuly released as Guidance 5. Part 2 is listed below Part 1. You are encouraged to copy your previous work for pary 1 into this notebook to submit both assignments together.

1.1 Portfolio of Practical work - Part 1, the Strategy Pattern

The tasks in this notebook should be submitted as the first part of the mandatory assignment, Portfolio of Practical work, that is due by 17 November 2022. (**Note: This portfolio will have additional components. The link to hand in will only be published once all requirements have been posted.)**

This notebook requires you to complete the missing parts of the Jupyter notebook. This will include comments in markdown, or completing code for the outline classes that is provided.

1.1.1 Context and Task

1.1.2 Part 1

Most data science projects start by pre-processing a dataset to ensure the data is ready to use for its intended purpose. One of the tasks that a datascientist would typically complete during such a pre-processing phase is to replace missing data values in the dataset using a process known as imputation. A popular toolkit that assist with this task in python is class sklearn.preprocessing.Imputer. A discussion of this class and its properties and methods can be found at http://lijiancheng0614.github.io/scikit-learn/modules/generated/sklearn.preprocessing.Imputer.html .

You are required to design and implement your own version of class Imputer. Your version should make use of the **strategy pattern** to ensure it is extensible and easy to maintain.

1.1.3 Specifications

Initial Parameters Your class Imputer should (initially) accept should accept two parameters, namely strategy and axis with the following options:

strategy : string, optional (default="mean"). The imputation strategy. - If "mean", then replace missing values using the mean along the axis. - If "median", then replace missing values using the median along the axis. - If "mode", then replace missing using the most frequent value along the axis. Important: It should be possible to add an additional strategy without affecting any of the existing strategy implementations.

axis : integer, optional (default=0) The axis along which to impute. - If axis=0, then impute along columns. - If axis=1, then impute along rows. (Not to be implemented now) Note: Initially **we will only impute along columns as the axis you DO NOT need to write any code for dealing with the axis = 1**.

Methods Your class should support two methods, namely fit and transform with the following behaviours:

fit : ***fit(X)***

- fit the imputer on X.

Parameters: - X : {array-like, sparse matrix}, shape (n_samples, n_features) - Input data, where n_samples is the number of samples and n_features is the number of features.

Returns:
- self : object - Returns self.

In other words: Fit receives as input the "matrix" of incomplete data, with the "boundaries" of the area for which we want to impute (calculate) missing values. (i.e. a single column, or the entire matrix) and returns an object containing only the part we want to do the imputation on.

transform : ***transform(X)***

- Impute all missing values in X and returns X with new values.

Parameters: - X : {array-like, sparse matrix}, shape = [n_samples, n_features] - The input data to complete.

Your imputer should work as follows: Firstly, the user should create an instance of the imputer (see example cpommand below). In this case the parameters indicate that the imputation strategy should calculate the mean of the values in each column and replace missing values with the calculated mean. You may assume missing values will always be indicated by the word 'nan'. **(Note: The axis = 0 parameter shown in the example here is not a current requirement.It indicates that the imputation should be by columns. Your version should not have this parameter at all).**

```
imputer = Imputer(strategy = 'mean', axis = 0)
```

Secondly, the user needs to "fit" the imputation to the dataset. This means the user needs to tell the class which rows and columns must be included in the imputation. The statement below specifies that all rows and columns 1 to 3 should be included. You may simplify the syntax your class expects, but it must be documented clearly. `imputer = imputer.fit(X[:,1:3])`

Lastly, the user will invoke the transform method. Transform returns a copy of the input data that has now been imputed. `X = imputer.transform()`

```
array([['France', 44.0, 72000.0],
       ['Spain', 27.0, 48000.0],
       ['Germany', 30.0, 54000.0],
       ['Spain', 38.0, 61000.0],
       ['Germany', 40.0, nan],
       ['France', 35.0, 58000.0],
       ['Spain', nan, 52000.0],
       ['France', 48.0, 79000.0],
       ['Germany', 50.0, 83000.0],
       ['France', 37.0, 67000.0]], dtype=object)
```

Example of data before imputation

```
array([['France', 44.0, 72000.0],
       ['Spain', 27.0, 48000.0],
       ['Germany', 30.0, 54000.0],
       ['Spain', 38.0, 61000.0],
       ['Germany', 40.0, 63777.77777777778],
       ['France', 35.0, 58000.0],
       ['Spain', 38.77777777777778, 52000.0],
       ['France', 48.0, 79000.0],
       ['Germany', 50.0, 83000.0],
       ['France', 37.0, 67000.0]], dtype=object)
```

Example of data after imputation

1.2 Imputer Classes

[114]:
```python
# CODE YOUR IMPLEMENTATION OF IMPUTER CLASSES HERE
# REMEMBER TO COMMENT WELL
import numpy as np
import pandas as pd

#Super class
class Imputer:
    #Importing the dataset
    def __init__(self, strategy="mean"):
        self.strategy = strategy

    #Fitting the imputer to the dataset
    def fit(self, X):
        if self.strategy == "mean": #Mean Imputation
            self.imputer = MeanImputer() #MeanImputer class
        elif self.strategy == "median": #Median Imputation
            self.imputer = MedianImputer() #MedianImputer class
        elif self.strategy == "most_frequent": #Most Frequent Imputation
            self.imputer = MostFrequentImputer() #MostFrequentImputer class
        else:
            raise ValueError("Strategy not supported") #Raise an error if the␣
 ↪strategy is not supported
        self.imputer.fit(X) #Fit the imputer to the dataset
```

```
    #Transforming the dataset
    def transform(self, X):
        return self.imputer.transform(X) #Transform the dataset

#Mean Imputer
class MeanImputer: #MeanImputer class
    def fit(self, X):
        self.mean = X.mean() #Calculate the mean of the dataset

    def transform(self, X):
        return X.fillna(self.mean) #Fill the missing values with the mean
#Median Imputer
class MedianImputer: #MedianImputer class
    def fit(self, X):
        self.median = X.median() #Calculate the median of the dataset

    def transform(self, X):
        return X.fillna(self.median) # Fill the missing values with the median
#Most Frequent Imputer
class MostFrequentImputer: #MostFrequentImputer class
    def fit(self, X):
        self.most_frequent = X.mode().iloc[0] #Calculate the most frequent␣
 ↪value of the dataset

    def transform(self, X):
        return X.fillna(self.most_frequent) #Fill the missing values with the␣
 ↪most frequent value
```

1.3 Example of use

[149]:
```
## Code an example to show how the imputer classes is used
## Remember to comment
# Importing the dataset with some missing values
df = pd.DataFrame({"a": [44.0, 27.0, 30.0, 38.0, 40.0, 35.0, np.nan, 48.0, 50.
 ↪0, 37.0],
                   "b": [72000.0, 48000.0, 54000.0, 61000.0, np.nan, 58000.0,␣
 ↪52000.0, 79000.0, 83000.0, 67000.0]
                   })

#Create the imputer object
imputer = Imputer(strategy="most_frequent")
#Fit the imputer to the dataset
imputer.fit(df)
#Transform the dataset
imputer.transform(df)
```

[149]:

	a	b
0	44.0	72000.0
1	27.0	48000.0
2	30.0	54000.0
3	38.0	61000.0
4	40.0	48000.0
5	35.0	58000.0
6	27.0	52000.0
7	48.0	79000.0
8	50.0	83000.0
9	37.0	67000.0

1.3.1 Part 2

Consider the possibility of having to add a new strategy and/or having to change your imoplementation to also support imputing along axis 1. Explain how the strategy pattern makes your design resistent to the impact of such changes.

Write a brief reflection on the strategy pattern and how its utility is demonstarted in the above code here. 150 - 300 words.

Reflection: Imputer is the class for the imputer. It takes the strategy as an input and saves it as an attribute. Imputer has two methods: fit and transform. The fit method takes the data as an input and fits the imputer. The transform method takes the data as an input and returns the imputed data. MeanImputer, Median Imputer, and MostFrequentImputer are three different imputers. They all have two methods: fit and transform. The fit method takes the data as an input and fits the imputer. The transform method takes the data as an input and returns the imputed data. The fit method for MeanImputer, MedianImputer and MostFrequentImputer calculates the mean, median and most frequent value of the data and save it as an attribute. The transform method for MeanImputer, Medialmputer and MostFrequentImputer fills the missing values with the mean, median and most frequent value of the data respectively. The same interface unifies a family of algorithms that are included in different classes and made interchangeable. Each family member has the ability to create new behaviors, but they all inherit the behaviors of the superclass, which can be overridden to create new behaviors. A common default interface is enforced by defining abstract methods in the superclass and requiring any subclass to implement the specific behavior.

1.4 Portfolio of Practical work - Part 2, the Factory Pattern

1.4.1 Context

One possible problem with the use of the strategy pattern is the reliance on the client to compose the used object with the correct "strategy class" to ensure the required behaviour. The previous specification for Part 1 of this task already required the use of a parameter (for example "strategy = 'mean' ") to determine which strategy the class should be composed with. However, this leaves the code to instantiate with a specific strategy inside your imputer class. Ideally we want to keep the imputer open for extension but closed for modification. From a software usage point of view, it would be more convenient, and less error prone, to simply specify the behaviour that would be desirable as a parameter and have an external "factory" take care of the instantiation.

1.4.2 Task

Your task consist of thrree parts, the first two are short written discussions (you may use diagrams as part of the discussions).

1. Explain how you can use a factory to take care of the strategy instantiation. This explanation should take the form of a discussion of the suggested design for the overall collection of classes for the imputer and any clients that will use it.
2. Discuss the benefits and/or negatives of the above design
3. Provide all the code for the suggested design. Including new versions for any classes you already wrote in part 1 of the protfolio assignment. Also add code to showcase how the classes work

1.5 Suggested Design

The ImputerFactory class has only one method get_imputer: takes strategy as an argument and returns an object of the class MeanImputer, MedianImputer or MostFrequentmputer according to the strategy passed as an argument.

1.6 Pros and Cons of Design

1.6.1 Factory Pattern

A factory pattern is a creational pattern. It is used to create objects without exposing the instantiation logic to the client and refer to newly created objects using a common interface.

Some advantages are that the factory pattern allows the sub-classes to choose the type of objects to create, and it promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extend that abstract class. The main disadvantage of the factory pattern is that the classes of the factory are tightly coupled to the classes of the product. That means that any change in the product interface will require a change in the factory interface. The factory pattern requires a lot of subclasses since the factory class is responsible for instantiating all the different objects in the product hierarchy. The factory pattern adds complexity to the code because you need to introduce a lot of new subclasses to implement it. The best scenario to use the factory pattern is when the client doesn't know what exact class it requires or when the client should be able to choose the type of object to create.

1.6.2 Strategy Pattern

A Strategy pattern is a behavioral pattern. It is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

Some advantages are that the strategy pattern allows you to change the behavior of an object at run-time and that clients must know the existence of different strategies and that clients must understand how the strategies differ. Some disadvantage of the strategy pattern is that it requires a lot of similar classes. For example, if you have 4 strategies, then you need to have 4 concrete classes of the strategy. But the main disadvantage of the strategy pattern is that it makes the code more complicated because you need to introduce a lot of new classes to implement it. The best scenario to use the strategy pattern is when you have a lot of similar classes that only differ in behavior.

[150]:
```
# Your code for the complete solution goes here.
```

[182]:
```
import numpy as np
import pandas as pd

#Super Class
class Imputer:
    def __init__(self, strategy="mean"): #Default strategy is mean
        self.strategy = strategy #Set the strategy

    def fit(self, X):
        self.imputer = ImputerFactory.get_imputer(self.strategy) #Get the␣
 ↪imputer
        self.imputer.fit(X) #Fit the imputer to the dataset

    def transform(self, X):
        return self.imputer.transform(X) #Transform the dataset
#Mean Imputer
class MeanImputer: #MeanImputer class
    def fit(self, X):
        self.mean = X.mean() #Calculate the mean of the dataset

    def transform(self, X):
        return X.fillna(self.mean) #Fill the missing values with the mean
# Median Imputer
class MedianImputer: #MedianImputer class
    def fit(self, X):
        self.median = X.median() #Calculate the median of the dataset

    def transform(self, X):
        return X.fillna(self.median) #Fill the missing values with the median
# Most Frequent Imputer
class MostFrequentImputer: #MostFrequentImputer class
    def fit(self, X):
        self.most_frequent = X.mode().iloc[0] #Calculate the most frequent␣
 ↪value of the dataset

    def transform(self, X):
        return X.fillna(self.most_frequent) #Fill the missing values with the␣
 ↪most frequent value

    #Factory Class
class ImputerFactory: #ImputerFactory class
    @staticmethod #Static method
    def get_imputer(strategy): #Get the imputer
        if strategy == "mean": #Mean Imputation
            return MeanImputer() #MeanImputer class
```

```
        elif strategy == "median": #Median Imputation
            return MedianImputer() #MedianImputer class
        elif strategy == "most_frequent": #Most Frequent Imputation
            return MostFrequentImputer() #MostFrequentImputer class
        else:
            raise ValueError("Strategy not supported") #Raise an error if the␣
 ↪strategy is not supported
```

[186]:
```
#Importing the dataset with some missing values
df = pd.DataFrame({"a": [44.0, 27.0, 30.0, 38.0, 40.0, 35.0, np.nan, 48.0, 50.
 ↪0, 37.0], "b": [
                 72000.0, 48000.0, 54000.0, 61000.0, np.nan, 58000.0, 52000.0,␣
 ↪79000.0, 83000.0, 67000.0]})

#Create the imputer object
imputer = Imputer(strategy="median")
#Fit the imputer to the dataset
imputer.fit(df)
#Transform the dataset
imputer.transform(df)
```

[186]:

	a	b
0	44.0	72000.0
1	27.0	48000.0
2	30.0	54000.0
3	38.0	61000.0
4	40.0	61000.0
5	35.0	58000.0
6	38.0	52000.0
7	48.0	79000.0
8	50.0	83000.0
9	37.0	67000.0

[ ]: