Report

Object Oriented Programming Final Project

Merlijn Mulder S5187540, Jerno Beuker S5193559 February 4, 2024

Data manager

For the datamanager, we chose to use pathlib for setting and creating the paths to the data as pathlib allows for use on multiple operating systems as different operating systems use different backslashes in their paths to files. This allows for a more flexible program. The paths that are used in the data manager are provided by the main, where the datasets should be located within the current working directory. Furthermore to improve readability custom type hints using TypeVar are created at the start of the file for data and labels.

The data manager includes one base class called DataLoader. This handles the initialization of the data and provides the data to the main. To improve encapsulation, the data can only be accessed via this class. This class implements an **init** function that takes a loader type, a path to the folder in which the data is stored and the name of the file containing labels. This can also be none when no label file is present. When this is called the data will be initialized in a lazy or eager fashion. This is done by private initialization methods. These are private as there is no need for the user to access them as this could potentially break the program when not executed in the right order. We furthermore chose to make these separate methods so the getter and setter in this function could use the same code to load new data when a new path is provided without directly creating another instance of the class. This allows for a more memory-efficient program when needed. These functions than call the lazy and eager loaders.

This base class furthermore implements the magic function **len** which will return the length of the dataset stored in one of the private attributes in the class. The magic function **getitem** is used to load a specific data point. We have chosen to do the check for a label here which is None when there is no label present. This way the method will always return the same structure, allowing for more consistency in the code and allowing a user in the main to also check whether or not a datapoint has a certain label. Note that this function is the only way to access the data as all attributes in which the data is stored is made private to ensure encapsulation.

Lastly, the base class contains a **train_test_split** function. This function does not return any data as for encapsulation reasons this should only be done by the **get_item** function. It, therefore, shuffles the list of data points and labels in a zip (to keep the right label on the same index as the data point), and sets a private attribute _len_train and _len_labels. These private attributes are returned via the **len** function together with the length of the dataset. Via this function, a user can determine what indexes are part of the test and train set. This allows for the data only to be called via the **getitem** function while still creating a train and test set, ensuring encapsulation. Note that the data stays within the same attribute, ensuring a more memory-efficient program as no copies of the original order have to be kept.

Lazy loader

The lazy loader does most of the work in the data loader. This function takes a path and walks through the entire folder loading all paths. Due to a malfunction in the Walk method from Pathlib, we chose to use the corresponding method from the os library after which the paths were converted to a pathlib path again. This still ensures the possibility of using the paths in different operating systems. While walking trough the folders, the program determines whether the dataset contains multiple/a hierarchy of folders. If this is the case it will store the name of the folders as labels in the data files. If this is not the case it checks if a file containing the labels needs to be loaded if it is encountered, which will be then labels. Note that if this is provided by the user, this is automatically chosen as the right label instead of the folders. This means that the program would also allow for a hierarchical structure while still having a label file. Because the program recognizes this automatically, the user does not have to specify this upfront, achieving easier usability and less need of variables that need to be passed through multiple classes. We chose to make the method in this class private as it should only be called from the data manager base class as there is no need for the user to call this separately, preventing wrong usage. The method returns the paths to the data and the labels. We chose to already load the labels here so that are directly initialized as there is no need to do that multiple times later. Here is made sure that via the private method _match_labels the labels that are found in the label file, if applicable, are matched with the names of the data files. If no matches are found errors are raised. Multiple columns in the CSV are checked in the case that names and labels are switched to allow for more flexible code.

Eager Loader

To prevent code duplicates and a more efficient flow of the program, the eager loader uses the lazy loader to load all the paths to the data. It then uses the private method _load_files to load all the data the paths point to. The method in this class is private as it should only be called via the base class.

LoadData

All data is loaded by this class and eventually returned to the base class in the data manager which is why all methods in this class are private. Furthermore, as with the lazy loader, a datapoint is only loaded when **getitem** is called we also decided to make this a separate class as it allows for code reusability. The loading is first going past the private **load_file** method. This method checks the suffixes of the paths to make sure different files are loaded in the correct way. This allows for the addition of extra file types in the code, without the user having to specify this by defining the file type in the main function for the loading of the data.

BatchLoader

The batch loader is responsible for creating batches. This is done according to the variables passed into the **init** method. It is passed an instance of the data manager here, so it is able to determine the length of the dataset. The length of the dataset is returned in the form of a tuple, with the length of a test and a train set. This way the batch loader can determine whether or not train and test sets exist and to make batches based on those sets with specific lengths. This is done in the private **set_parameters** function. We decided to make this a separate function because when the parameters are changed via the **getter** function, it is possible the length has to be initiated again and no other instance of the class has to be created this way. Furthermore, it prevents code duplicates in the **init** and the **getter**. With the help of the magic **iter** method that calls the **next** method, the data can be loaded in batches by calling the **getitem** function in the data manager, which will be returned to the main. The data is called based on the indexes in a batch created by the batch loader. As a test set always starts at the index at which the train set ends, the length of the test set is added to the index initialized by the batch loader to make sure the correct data is called when applicable. The creation of the batches themselves is done via the public splits method. This is public as a user can decide here whether to splits a batch and create it. Note that this function does not return anything, but just creates new batches when the user would want to. The batches in this method are created based on the length of the dataset, which corresponds to the indexes of the data in an array that need to be assigned to a certain batch.

Data preprocessing

We split the data preprocessing up into 4 files. The first file is an Abstract Base Class specifies the structure of the preprocessing methods, which are two other files, namely the preprocImag and the preprocAudio. We decided to split the preprocessing steps up into two files since the usage of libraries and the usage of preprocessing methods that needed to be implemented are very different, so they should be placed in separate files. The last file is the pipeline that needs to apply the preprocessing methods to batches of data. This is also a separate file since the functionality is drastically different. In the Abstract Base Class, we put two methods, since these are the only two methods that are shared in every preprocessing method (except for the mel spectogram since it does not take any arguments). We made an Abstract Base Class to have the same structure in all four preprocessing types. The structure for all preprocessing methods is quite similar, all hyperparam- eters are specified in the constructor in which they are put in private attributes (they are private methods because from this point the user should not be able to access them, the user only needs to be able to access the methods from the Abstract Base Class). Since all attributes are private, we implemented a private getter for each of them, this is also a private method since it is only needed in the special call method of the preprocessing, so the user should not use it. The last important method is the special method ’call’. This method triggers when an instance of an object is called using the brackets(). This is when the prepro- cessing method should apply the preprocess. Any further methods are private since they are meant to remove some bulk from the special method call and the user should not use them. We opted for such a structure since the assignment hinted us in this direction and this makes the preprocesses similar and simple to use. Furthermore, in the pipeline file, we added another custom typehint using Type- Var, namely pro. This typehint is meant to hint towards the preprocesses that need to be implemented. They are tuplets that contain the preprocess- ing method and a list of corresponding hyperparameters. The addition of this new typehint massively improves readability as the single typehint is very large because of the possibility of either one preprocess, a list of preprocess, and the possibility of having no hyperparameters, or a list of hyperparameters.

Main

The main method is called three public methods. One for each data set. For each method, the correct dataset is loaded via the data manager. Furthermore, instances of the Batch_loader and the pipeline are provided to these functions, which prevents the need to initiate multiple instances in multiple methods. To avoid code duplication we have added two more methods: **process_data_with_batch** and **procees_data_without_batch** which load the data based on the need for batches via the batchloader. This prevents code duplication as these functions can be used for all different methods used for different datasets. At this point, the name of the data point is not needed as the label will be attached, which is therefore removed from the data, so the data can be more easily processed. Furthermore, all possibilities for variables are defined as constants in the main and the other methods. By enforcing the use of these constants, users will use the right variables needed to run the code. They furthermore allow the user to quickly change a parameter when needed.