Notes and Proposals for mlmodel's Data Loader

Data-loading needs

Four aspects of the mlmodels model zoo influence its data-loading needs:

- mlmodels is multipurpose. Since it contains models for supervised and unsupervised learning on various kinds of data, a data loader cannot always load the data as present in the data file, and often needs to reformat the data after loading.
- mlmodels aims to provide a full pipeline for training and inference from raw data rather than saved preprocessed data. Thus the data loader needs to expose options for preprocessing the data.
- mlmodels is multiframework. Thus, the data loader needs to be able to format the loaded data as per the needs of the framework used by any one given model in the zoo.
- mlmodels is built with maximum extensibility and modularity in mind. Thus, new methods for performing the three above-mentioned capabilities (reformatting the data, preprocessing the data, and outputting the data in a format that can be used as an input for the given model's framework) need to be easily addable to the data-loader with minimal new code and minimal to no refractoring.

Use case examples and assumptions

A flexible data loader needs to be able to load the required data with minimal information provided, while at the same time being able to accept additional informatin on how to interpet, preprocess, and output the data. Such flexibility is made possible by making assumptions on the structure of the data and the preprocesssing and output needs of the target model when such information is not explicitly provided to the data loader. The following are the examples of the loading of various datasets for various models in the mlmodels zoo, and the assumptions that a flexible data loader must make in the process of loading them.

Maximum-assumption use case. In the most basic case of data loading within the model zoo, the dataset to be loaded and used for supervised learning is stored in a csv file with named columns in which one column represents the labels, the data doesn't require preprocessing, and the entire dataset can easily fit in memory. *GOOG-year.csv* and *titanic_train_preprocessed.csv* are examples of such datasets currently used in mlmodels. For such datasets:

- All necessary reformatting (most often, simply interpreting the first row of the csv as column names rather than a datapoint) is done by pandas's **pd.read_csv**, and the subsequent selection of data and label columns.
- Preprocessing is done by directly using a specified preprocessor method or method alias on the loaded data, with the possibility of storing the fitted preprocessor for later preprocessing test data. If no such method or alias is provided, preprocessing isn't done at all.

In such cases, the following assumptions are made:

- At the most fundamental level, it is assumed that the csv file represents a tabular view of a dataset that is used for a supervised learning task, with some columns representing data and some columns representing labels.
- The assumption on the reformatting method is done based one the file extension: the .csv file extension suggests using **pd.read_csv**.
  - Assumptions on which columns represent data and which columns represent labels can also be made if this information is not supplied.
- It's assumed that preprocessing can be done with default parameters and the resultant fitted preprocessor can be stored away for later use trivially.

Medium-assumption use case. In relatively complex cases of data loading within the model zoo, the data stored in a file cannot be read directly. Datasets that store sentence corpora, such as *ag_news_csv*, *ner_dataset.csv*, or embeddings, such as *imdb.npz*, are examples of such cases. Image datasets such as MNIST or CIFAR are also examples of such cases. For such datasets:

- Data can still be inferred to be loaded with the appropriate method based on the file extension (e.g. **pd.read_csv** for csv and **np.load** for npz), but default arguments for the method cannot be used or an additional step is required after loading. For example, an encoding might need to be specified, or instead of loading the entire dataset into memory a generator that reads the data file chunk-by-chunk might need to be generated. Additionally, the location of the data file might be remote (e.g. stored remotely on AWS, or on the repository), or the location might be an image directory.
- After loading, data might need to be preprocessed on a per-column/per-axis basis, with a different encoder being specified for each column/axis.

In such cases, fundamentally the dataset is still assumed to have the basic dataset structure of data points that contain features, rather than being raw text file corpuses, graphs, or other structures. Assumption on the reformatting method is still made based on the file path extension.

Minimum-assumption use case. In even more complex cases of data loading, the only option to load the data is to use a custom method for reading the data files. For example, text corpuses might need to be loaded, tokenized, vectorized, and saved as arrays. The use of *rt-polaritydata* dataset for movie review sentiment analysis as test data for *no_03_textcnn* model is an example of such case in mlmodels. For such cases:

- A specific method and method arguments for loading the data need to be specified.
- A specific method for method arguments for preprocessing the loaded data need to be specified.

Proposed class structure and dictionary schemas

Class structure. At the end of the data loading pipeline, the data is always passed to a framework's model-fitting, predicting, or evaluating method. Since each model is aware of the framework it is using, the format the the loaded data needs to assume is always known a priori within the model code. The entire data loading pipeline can be interpreted as being made out of the following two steps:

- The data is loaded, reformatted, and preprocessed according to the specified methods.
- The processed data is then put into a format that can be passed to a framework's data-fitting function.

Since the first step is universal regardless of which framework a model uses, the following class structure can be used for data loading:
AbstractDataLoader: this abstract class's constructor has two arguments:

- *input_pars*: input parameters dictionary
- *processing_pars* (*None* by default): preprocessing parameters dictionary

The constructor uses an internal method *_interpret_input_pars(self, input_pars)* to interpet *input_pars* and return an interpreted representation of the data. Then it uses an internal method *_interpret_processing_pars(self, processing_pars, interpreted_data)* for preprocessing and saving the fitted preprocessor, and assigns the returned preprocessed data to *self._processed_data*. *_interpret_processing_pars(self, processing_pars, interpreted_data)* returns *interpreted_data* if **processing_pars** is None.
A subclass of **AbstractDataLoader** will exist for each framework, (i.e. **TfDataLoader**, **PyTorchDataLoader**, **GluonDataLoader**, etc.), each exposing methods that transform the saved *self._processed_data* into the data representation that can be directly used by the framework's model-fitting functions. For example, **PyTorchDataLoader** will have a method for returning a torch.utils.data.DataLoader object, while **TfDataLoader** will have a method for returning a tf.data.Dataset object.

Dictionary schemas.
input_pars:

- *location*. File/folder/network location.
- *location_type* (file/folder/aws/etc.) Can be inferred from *location* if not specified.
- *data_type* (csv,npy,npz,h5,image,etc.). Can be inferred from *location* if not specified.
- *generator* (T/F). Whether to load the data fully within memory or whether to return a generator/iterator object.
- *X_cols*: data column indices/names. Can be assumed to be all columns other than the final one if unspecified. Will not be used if a *data_loader* that isn't certain to output tabular data is used.
- *y_cols*: label column indices/names. Can be assumed to be the final column if unspecified. Will not be used if a *data_loader* that isn't certain to output tabular data is used.
- *data_loader*. Data loader method. If unspecified, inferred using all of the above data.
- Others. Any other dictionary keys/labels are passed to the *data_loader* as arguments.

processing_pars:

- *data_preprocessor* (method or a list of dictionaries). If a list of dictionaries is provided, then the entries are interpreted as a per-column preprocessing specifications. Each dictionary will have the following entries.
  - *index* (string/int/list of strings/list of ints).
  - *encoder*. Method to preprocess the specified column/columns.
  - *Others*. Any other arguments will be passed to encoder as arguments.
- *Others*. Any other dictionary keys/labels are passed to the *data_preprocessor* as arguments.