Machine learning for practical projects

Ita cirovic Donev (FinMetrika d.o.o.)

2024-02-01

Table of contents

Chapter 1

Introduction

The `finmetrika-ml` library is a machine learning library for practical projects, predominantly for the financial industry.

The library is organized as follows:

```
finmetrika_ml
├── data
    ├── data_processing.py
    └── vizualization.py
├── model
    ├── training.py
    ├── evaluation.py
    └── metrics.py
└── utils.py
```

1.1 Installation

To install use the `pip` command in your virtual environment:

```
pip install finmetrika_ml
```

1.2 Create from templates

To create a Jupyter notebook from template run the following command in terminal:

```
fm_create_nb path/notebook_name.ipynb
```

and replace `path` with the path directory where you wish the notebook to be saved and `notebook_name` with your desired name choice.

Chapter 2

Setting up the machine learning project

Any data analysis and machine learning project requires lots of data exploration and experimentation with different model types along with different model arguments. Things can get pretty messy fast. To make life a bit easier it is best to organize at the start of the project by creating a new repository localy and/or in the cloud. This will ensure version control of your project.

Inevitably, there will be many arguments that we will need to use in the course of our experimentation. Best is to store them in the `config.py` file either as `ArgParse` or `dataclass` objects. Here is an example:

```
from dataclasses import dataclass, asdict, field
from pathlib import Path


# Get the absolute path to the directory where config.py is located
BASE_DIR = Path(__file__).resolve().parent.parent

@dataclass
class ProjectConfig:
    # Documenting experiments
    experiment_version: str = field(
        default="v0",
        metadata={'description': "Name of the training experiment"})

    experiment_description: str = field(
        default="This is test run",
        metadata={'description': "Describe the experiment in couple of sentences"})
```

Chapter 3

Data

Data module consists of:
- `data_read.py`: loading the local txt or csv files
- `create_datasets`: creating HuggingFace datasets from local files
-

Chapter 4

data_processing

For classification tasks, or any categorical data feature we can obtain labels with

`get_labels`

get_labels(df: DataFrame, col_label: str, verbose: bool)

Extract unique labels from the dataframe and save them to a list. Print the number of labels in the dataset as well as the first 5 labels if there are more than five labels in the dataset.

Arguments:

	type	default	description
df	DataFrame	None	Dataframe in which the labels are contained.
col_label	str	None	Name of the column in the dataframe containing labels.
verbose	bool	True	Print the statements. Defaults to True.

```
get_labels(df=my_dataframe, col_label="col_label")
```

`count_tokens`

count_tokens(df: DataFrame, col_input_ids: str, col_attn_mask: str)

Counts the number of tokens in each row of a DataFrame where the attention mask is 1.

Arguments:

	type	default	description
df	DataFrame	None	Dataframe containing the token data.
col_input_ids	str	input_ids	Name of the column in df that contains the input IDs. Defaults to “input_ids”.
col_attn_mask	str	None	Name of the column in df that contains the attention masks. Defaults to None.

Feature Engineering

Extract features from large language models for text classification.

`extract_feature_vector`

> extract_feature_vector(data_sample: DatasetDict, model: PreTrainedModel, tokenizer: PreTrainedTokenizerBase, device: str)

Extract features from large language models for text classification.

Arguments:

	type	default	description
data_sample	DatasetDict	None	Dataset including tokenized inputs. Expected to be a dictionary with keys matching the model's expected input names.
model	PreTrainedModel	None	The model from which to extract the feature vectors. Should be an instance of a class derived from transformers.PreTrainedModel.
tokenizer	PreTrainedTokenizerBase	None	The tokenizer corresponding to the model, used to identify model input names.
device	str	None	Compute engine to which the inputs should be transfered. Define using check_device().

Tokenized dataset means that the `DatasetDict` object has minimally `input_ids` in features for, minimally, `train` split. For some models, like BERT it will also have `attention_mask`. For example:

```
my_dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'input_ids', 'attention_mask'],
        num_rows: 125776
    })
    validation: Dataset({
        features: ['text', 'label', 'input_ids', 'attention_mask'],
        num_rows: 32342
    })
    test: Dataset({
        features: ['text', 'label', 'input_ids', 'attention_mask'],
        num_rows: 21563
    })
    other: Dataset({
        features: ['text', 'label', 'input_ids', 'attention_mask'],
        num_rows: 35399
    })
```

```
    rest: Dataset({
        features: ['text', 'label', 'input_ids', 'attention_mask'],
        num_rows: 1581911
    })
})
```

Example of usage:

```
model_name = "distilbert-base-uncased"
device = check_device()
model = AutoModel.from_pretrained(model_name).to(device)
tokenizer = AutoTokenizer.from_pretrained(model_name)

my_dataset_lhs = extract_feature_vector(my_dataset, model, tokenizer)
```

Chapter 5

data_sampling

Stratified random sampling

Sampling from a HuggingFace-like dataset:

`stratified_sample_from_dataset`

> stratified_sample_from_dataset(data: DatasetDict, by_split: str, random_seed: int, perc_sample: float, return_complement_sample: bool)

Stratified sampling without replacement. Sample a percentage of a dataset given the dataset split. If 'return_complement_sample' is set to True then the function returns the complement sample as well.

Arguments:

	type	default	description
data	DatasetDict	None	
by_split	str	None	Which data subset based on split should we sample from. Example: 'train'.
random_seed	int	None	Project arguments.
perc_sample	float	None	percentage of samples to obtain
return_complement_sample	bool	True	Save the compleent sample as well.

For example, if we have a HuggingFace dataset `emotion`

```
emotion
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
})
```

Say we want to create a smaller sample of 1% of train data but using stratified sampling. We should define which split to sample from and the percentage of samples. Note that due to stratified nature of sampling and depending on how many label examples are present in each label group there can be a possibility that we sample (in count) less or more than what you would get as exact 1% of total dataset.

```
emotion_sub = stratified_sample_from_dataset(
                data=emotion,
                by_split='train',
                random_seed=42,
                perc_sample=0.01)
emotion_sub
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 161
    })
    trainC: Dataset({
        features: ['text', 'label'],
        num_rows: 15839
    })
})
```

Chapter 6

data_vizualization

Categorical data

Plot frequency of classes using the bar chart including labels.

`plot_freq_classes`

plot_freq_classes(df: DataFrame, class_column: str, plot_no_classes: int, bar_color: str)

Create a horizontal bar plot of frequency classes.

Arguments:

	type	default	description
df	DataFrame	None	Dataframe containing the class.
class_column	str	None	Name of the column in df that contains the class label.
plot_no_classes	int	None	Number of classes to plot.
bar_color	str	#1f77b4	Color of the bars as HEX value.

`plot_tokens_per_class`

plot_tokens_per_class(df: DataFrame, class_column: str, tokens_cnt_column: str)

Plot a box-plot of the number of tokens per sequence. All classes are plotted in a decreasing order given by the median value.

Arguments:

	type	default	description
df	DataFrame	None	Dataframe containing the class_column and tokens_cnt_column.
class_column	str	None	Name of the column in df that contains the class label.
tokens_cnt_column	str	None	Name of the column in df that contains the number of tokens per sequence.

Chapter 7

training

Training

`TrainNN`

> TrainNN(model: _empty, training_dataloader: DataLoader, loss_fn: str, optimizer: _empty, num_epochs: int, device: str)

Train a neural network.

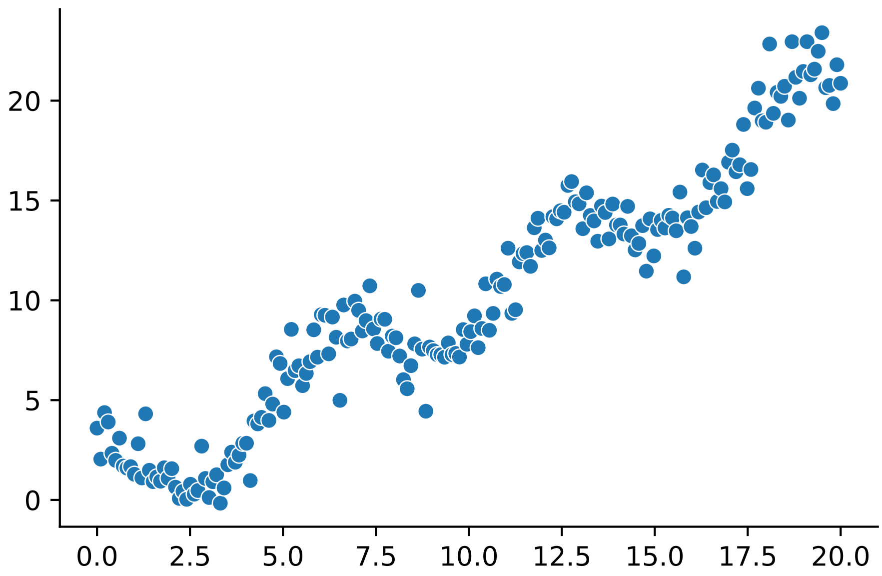
Arguments:

	type	default	description
model	_empty	None	Instantiated model class or a defined model architecture.
training_dataloader	DataLoader	None	Dataloader for training.
loss_fn	str	None	Loss function
optimizer	_empty	None	optimizer
num_epochs	int	None	Number of epochs to train.
device	str	None	Device on which to train the model. Use utils.check_device().

Let's see a simple example of randomly generated data:

```
# Define data
X = np.linspace(0,20, num=200)
y = X + np.cos(X)*2 + np.random.normal(size=X.shape)

# Create a dataset & dataloader
dataset_reg = RegressionDataset1D(X,y)
training_dataloader = DataLoader(dataset_reg, shuffle =True)
```



Let's fit a simple 2 layer linear model with a `tanh` activation function:

```
model = nn.Sequential(
    nn.Linear(1, 10),
```

```
    nn.Tanh(),
    nn.Linear(10, 1),
)

train = TrainNN(
    model=model,
    training_dataloader=training_dataloader,
    loss_fn=nn.MSELoss(),
    optimizer=torch.optim.SGD(model.parameters(), lr=0.001),
    num_epochs=20,
    device=check_device()
)

# train the model
print(f'Training ... ')
train.train()
```

```
Using mps device!
Training ...
```

```
  0%|          | 0/20 [00:00<?, ?it/s]100%|██████████| 20/20 [00:00<00:00, 390167.81it/s]
```

Fine tuning

Feature extraction

`FineTuneFtsExtraction`

FineTuneFtsExtraction(model_name_hf: _empty, dataset_hf: DatasetDict, use_hf: bool)

Fine tune a model using feature extraction. Training is done on the hidden states as features, without modifying the pretrained model.

Arguments:

	type	default	description
model_name_hf	_empty	None	Model name as shown on HuggingFace
dataset_hf	DatasetDict	None	Dataset dictionary with minimal splits:
use_hf	bool	True	Use transformers library for training.

Describing the model architecture

`model_size`

> model_size(model: _empty)

Count the number of parameters in the model

Arguments:

	type	default	description
model	_empty	None	Instantiated model class.

Chapter 8

utils

Various utility functions for checking and defining compute engine, logging and creating the experimentation documentation.

Reproducibility

Reproducibility is one of the most important aspects of proper project development and management, for ourselves, as well as for other people to whom we will share the project and possibly need to make decisions based on the results.

`set_all_seeds`

set_all_seeds(seed: int)

Set the seed for all packages: python, numpy, torch, torch.cuda, and mps.

Arguments:

	type	default	description
seed	int	None	Any positive integer value.

We can set the seed for most of the libraries that we use in machine learning like: `numpy`, `torch`, `torch.cuda`, `mps` as well as for Python in general.

```
set_all_seeds(seed=42)
```

If you are using `FLAGS` then simply replace the value of the seed for the data class defined for the reproducibility. For example, if my data class is called `seed` then I would use:

```
set_all_seeds(seed=FLAGS.seed)
```

Computation engine

`check_device`

check_device(verbose: bool)

Check which compute device is available on the machine.

Arguments:

	type	default	description
verbose	bool	True	Show all print statements.

We can use the function as follows, which if the argument `verbose` is `True` it will print out the compute device currently available.

```
device = check_device()
```

```
Using mps device!
```

`moveTo`

moveTo(obj: _empty, device: str)

Move an object to a specified device. It is a recursive function which checks iteratively for every element of obj. The device is determined by the function check_device(). Ref: Inside Deep Learning by Raff E. page 15

Arguments:

	type	default	description
obj	_empty	None	object
device	str	None	name of the device to move the obj to. Examples are "cuda", "mps,"cpu".

System information

`get_python_version`

get_python_version()

Return the current running Python version.

Arguments:

type	default	description

`get_package_version`

get_package_version(package_name: _empty)

Print the version of the Python package.

Arguments:

	type	default	description
package_name	_empty	None	Name of the package.

Creating experiment information document

`update_config`

update_config(FLAGS: _empty)

Update config arguments if any change was done via CLI when running "sh run.sh".

Arguments:

	type	default	description
FLAGS	_empty	None	Instantiation of the `config` dataclass.

`create_experiment_descr_file`

create_experiment_descr_file(config: _empty)

Create a txt file to include information on experiment including all the parameters used.

Arguments:

	type	default	description
config	_empty	None	Python script defining project parameters.

`add_runtime_experiment_info`

add_runtime_experiment_info(start_time: _empty, config: _empty)

Create structure of the experiment info file.

Arguments:

	type	default	description
start_time	_empty	None	*description*
config	_empty	None	*description*