Multimodal Transformers

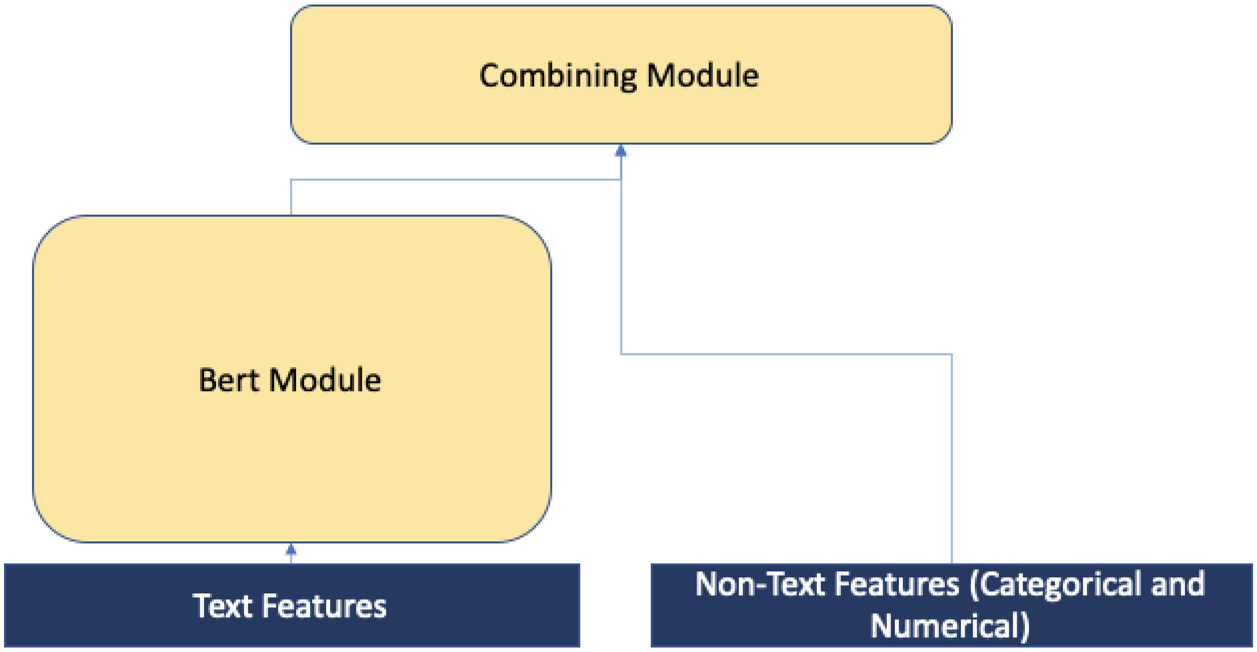
Ken Gu

Jul 27, 2021

NOTES

A toolkit for incorporating multimodal data on top of text data for classification and regression tasks. This toolkit is heavily based off of HuggingFace Transformers. It adds a combining module that takes the outputs of the transformers in addition to categorical and numerical features to produce rich multimodal features for downstream classification/regression layers. Given a pretrained transformer, the parameters of the combining module and transformer are trained based on the supervised task.

See its documentation for specific details regarding HuggingFace transformer models, configs, and tokenizers.



CHAPTER ONE

INSTALLATION

This package was written with **python3.7**. It depends on PyTorch and HuggingFace Transformers 3.0. and up.

1.1 Install

```
pip install multimodal-transformers
```

CHAPTER

TWO

INTRODUCTION BY EXAMPLE

This guide covers how to use the transformer with tabular models in your own project. We use a `BertWithTabular` model as an example.

For a working script see the github repository.

2.1 How to Initialize Transformer With Tabular Models

The models which support tabular features are located in *multimodal_transformers.model.tabular_transformers*. These adapted transformer modules expect the same transformer config instances as the ones from HuggingFace. However, expect a `multimodal_transformers.model.TabularConfig` instance specifying the configs.

Say for example we had categorical features of dim 9 and numerical features of dim 5.

```
from transformers import BertConfig

from multimodal_transformers.model import BertWithTabular
from multimodal_transformers.model import TabularConfig

bert_config = BertConfig.from_pretrained('bert-base-uncased')

tabular_config = TabularConfig(
        combine_feat_method='attention_on_cat_and_numerical_feats',  # change this to␣
↪specify the method of combining tabular data
        cat_feat_dim=9,  # need to specify this
        numerical_feat_dim=5,  # need to specify this
        num_labels=2,   # need to specify this, assuming our task is binary␣
↪classification
        use_num_bn=False,
)

bert_config.tabular_config = tabular_config
```

(continues on next page)

(continued from previous page)

```
model = BertWithTabular.from_pretrained('bert-base-uncased', config=bert_config)
```

In fact for any HuggingFace transformer model supported in *multimodal_transformers.model.tabular_transformers* we can initialize it using `multimodal_transformers.model.AutoModelWithTabular` to leverage any community trained transformer models

```
from transformers import AutoConfig

from multimodal_transformers.model import AutoModelWithTabular
from multimodal_transformers.model import TabularConfig

hf_config = AutoConfig.from_pretrained('ipuneetrathore/bert-base-cased-finetuned-
↪finBERT')
tabular_config = TabularConfig(
        combine_feat_method='attention_on_cat_and_numerical_feats',  # change this to␣
↪specify the method of combining tabular data
        cat_feat_dim=9,  # need to specify this
        numerical_feat_dim=5,  # need to specify this
        num_labels=2,   # need to specify this, assuming our task is binary␣
↪classification
)
hf_config.tabular_config = tabular_config

model = AutoModelWithTabular.from_pretrained('ipuneetrathore/bert-base-cased-
↪finetuned-finBERT', config=hf_config)
```

2.2 Forward Pass of Transformer With Tabular Models

During the forward pass we pass HuggingFace's normal transformer inputs as well as our categorical and numerical features.

The forward pass returns

- `torch.FloatTensor` of shape `(1,)`: The classification (or regression if tabular_config.num_labels==1) loss
- `torch.FloatTensor` of shape `(batch_size, tabular_config.num_labels)`: The classification (or regression if tabular_config.num_labels==1) scores (before SoftMax)
- `list` of `torch.FloatTensor` The outputs of each layer of the final classification layers. The 0th index of this list is the combining module's output

The following example shows a forward pass on two data examples

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

text_1 = "HuggingFace is based in NYC"
text_2 = "Where is HuggingFace based?"
model_inputs = tokenizer([text_1, text_2])

# 5 numerical features
numerical_feat = torch.rand(2, 5).float()
# 9 categorical features
```

(continues on next page)

(continued from previous page)

```
categorical_feat = torch.tensor([[0, 0, 0, 1, 0, 1, 0, 1, 0],
                                 [1, 0, 0, 0, 1, 0, 1, 0, 0]]).float()
labels = torch.tensor([1, 0])

model_inputs['cat_feats'] = categorical_feat
model_inputs['num_feats'] = numerical_feat
model_inputs['labels'] = labels

loss, logits, layer_outs = model(**model_inputs)
```

We can also pass in the arguments explicitly

```
loss, logits, layer_outs = model(
    model_inputs['input_ids'],
    token_type_ids=model_inputs['token_type_ids'],
    labels=labels,
    cat_feats=categorical_feat,
    numerical_feats=numerical_feat
)
```

2.3 Modifications: Only One Type of Tabular Feature or No Tabular Features

If there are no tabular features, the models basically default to the ForSequenceClassification models from HuggingFace. We must specify `combine_feat_method='text_only'` in `multimodal_transformers.model.TabularConfig`. During the forward pass we can simply pass the text related inputs

```
loss, logits, layer_outs = model(
    model_inputs['input_ids'],
    token_type_ids=model_inputs['token_type_ids'],
    labels=labels,
)
```

If only one of the features is available, we first must specify a `combine_feat_method` that supports only one type of feature available. See supported methods for more details. When initializing our tabular config we specify the dimensions of the feature we have. For example if we only have categorical features

```
tabular_config = TabularConfig(
    combine_feat_method='attention_on_cat_and_numerical_feats',  # change this to␣
↪specify the method of combining tabular data
    cat_feat_dim=9,  # need to specify this
    num_labels=2,   # need to specify this, assuming our task is binary classification
)
```

During the forward pass, we also pass only the tabular data that we have.

```
loss, logits, layer_outs = model(
    model_inputs['input_ids'],
    token_type_ids=model_inputs['token_type_ids'],
    labels=labels,
    cat_feats=categorical_feat,
)
```

2.4 Inference

During inference we do not need to pass the labels and we can take the logits from the second output from the forward pass of the model.

```
with torch.no_grad():
    _, logits, classifier_outputs = model(
        model_inputs['input_ids'],
        token_type_ids=model_inputs['token_type_ids'],
        cat_feats=categorical_feat,
        numerical_feats=numerical_feat
    )
```

CHAPTER THREE

COMBINE METHODS

This page explains the methods that are supported by `multimodal_transformers.tabular_combiner.TabularFeatCombiner`. See the table for details.

If you have rich categorical and numerical features any of the `attention`, `gating`, or `weighted sum` methods are worth trying.

The following describes each supported method and whether or not it requires both categorical and numerical features.

This table shows the the equations involved with each method. First we define some notation

- equation denotes the combined multimodal features
- equation denotes the output text features from the transformer
- equation denotes the categorical features
- equation denotes the numerical features
- equation denotes a MLP parameterized by equation
- equation denotes a weight matrix
- equation denotes a scalar bias

CHAPTER FOUR

COLAB EXAMPLE

Here is an example of a colab notebook for running the **toolkit** involving data preparation, training, and evaluation:

1. Training a BertWithTabular Model for Clothing Review Recommendation Prediction

CHAPTER

FIVE

MULTIMODAL_TRANSFORMERS.MODEL

Contents

5.1 Tabular Feature Combiner

class TabularFeatCombiner(*tabular_config*)

Bases: `torch.nn.modules.module.Module`

Combiner module for combining text features with categorical and numerical features The methods of combining, specified by `tabular_config.combine_feat_method` are shown below. $\mathbf{m}$ denotes the combined multimodal features, $\mathbf{x}$ denotes the output text features from the transformer, $\mathbf{c}$ denotes the categorical features, $\mathbf{t}$ denotes the numerical features, $h_{\mathbf{\Theta}}$ denotes a MLP parameterized by Θ, W denotes a weight matrix, and b denotes a scalar bias

- **text_only**

$$\mathbf{m} = \mathbf{x}$$

- **concat**

$$\mathbf{m} = \mathbf{x} \, \| \, \mathbf{c} \, \| \, \mathbf{n}$$

- **mlp_on_categorical_then_concat**

$$\mathbf{m} = \mathbf{x} \, \| \, h_{\mathbf{\Theta}}(\mathbf{c}) \, \| \, \mathbf{n}$$

- **individual_mlps_on_cat_and_numerical_feats_then_concat**

$$\mathbf{m} = \mathbf{x} \, \| \, h_{\mathbf{\Theta_c}}(\mathbf{c}) \, \| \, h_{\mathbf{\Theta_n}}(\mathbf{n})$$

- **mlp_on_concatenated_cat_and_numerical_feats_then_concat**

$$\mathbf{m} = \mathbf{x} \, \| \, h_{\mathbf{\Theta}}(\mathbf{c} \, \| \, \mathbf{n})$$

- **attention_on_cat_and_numerical_feats** self attention on the text features

$$\mathbf{m} = \alpha_{x,x}\mathbf{W}_x\mathbf{x} + \alpha_{x,c}\mathbf{W}_c\mathbf{c} + \alpha_{x,n}\mathbf{W}_n\mathbf{n}$$

  where $\mathbf{W}_x$ is of shape `(out_dim, text_feat_dim)`, $\mathbf{W}_c$ is of shape `(out_dim, cat_feat_dim)`, $\mathbf{W}_n$ is of shape `(out_dim, num_feat_dim)`, and the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^{\top}[\mathbf{W}_i\mathbf{x}_i \, \| \, \mathbf{W}_j\mathbf{x}_j]\right)\right)}{\sum_{k \in \{x,c,n\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^{\top}[\mathbf{W}_i\mathbf{x}_i \, \| \, \mathbf{W}_k\mathbf{x}_k]\right)\right)}.$$

- **gating_on_cat_and_num_feats_then_sum** sum of features gated by text features. Inspired by the gating mechanism introduced in Integrating Multimodal Information in Large Pretrained Transformers

$$\mathbf{m} = \mathbf{x} + \alpha\mathbf{h}$$

$$\mathbf{h} = \mathbf{g_c} \odot (\mathbf{W}_c\mathbf{c}) + \mathbf{g_n} \odot (\mathbf{W}_n\mathbf{n}) + b_h$$

$$\alpha = \mathrm{min}(\frac{\|\mathbf{x}\|_2}{\|\mathbf{h}\|_2} * \beta, 1)$$

  where β is a hyperparamter, $\mathbf{W}_c$ is of shape `(out_dim, cat_feat_dim)`, $\mathbf{W}_n$ is of shape `(out_dim, num_feat_dim)`. and the gating vector $\mathbf{g}_i$ with activation function R is defined as

$$\mathbf{g}_i = R(\mathbf{W}_{gi}[\mathbf{i} \, \| \, \mathbf{x}] + b_i)$$

  where $\mathbf{W}_{gi}$ is of shape `(out_dim, i_feat_dim + text_feat_dim)`

- **weighted_feature_sum_on_transformer_cat_and_numerical_feats**

$$\mathbf{m} = \mathbf{x} + \mathbf{W}_{c'} \odot \mathbf{W}_c\mathbf{c} + \mathbf{W}_{n'} \odot \mathbf{W}_n\mathbf{t}$$

Parameters **`tabular_config`** (`TabularConfig`) – Tabular model configuration class with all the parameters of the model.

forward(*text_feats*, *cat_feats=None*, *numerical_feats=None*)

Parameters

- **text_feats** (`torch.FloatTensor` of shape `(batch_size, text_out_dim)`) – The tensor of text features. This is assumed to be the output from a HuggingFace transformer model
- **cat_feats** (`torch.FloatTensor` of shape `(batch_size, cat_feat_dim)`, *optional*, defaults to `None`)) – The tensor of categorical features
- **numerical_feats** (`torch.FloatTensor` of shape `(batch_size, numerical_feat_dim)`, *optional*, defaults to `None`) – The tensor of numerical features

Returns A tensor representing the combined features

Return type `torch.FloatTensor` of shape `(batch_size, final_out_dim)`

5.2 Tabular Config

class TabularConfig(*num_labels*, *mlp_division=4*, *combine_feat_method='text_only'*, *mlp_dropout=0.1*, *numerical_bn=True*, *use_simple_classifier=True*, *mlp_act='relu'*, *gating_beta=0.2*, *numerical_feat_dim=0*, *cat_feat_dim=0*, ***kwargs*)

Bases: `object`

Config used for tabular combiner

Parameters

- **mlp_division** (*int*) – how much to decrease each MLP dim for each additional layer
- **combine_feat_method** (*str*) – The method to combine categorical and numerical features. See `TabularFeatCombiner` for details on the supported methods.
- **mlp_dropout** (*float*) – dropout ratio used for MLP layers
- **numerical_bn** (*bool*) – whether to use batchnorm on numerical features
- **use_simple_classifier** (*bool*) – whether to use single layer or MLP as final classifier
- **mlp_act** (*str*) – the activation function to use for finetuning layers
- **gating_beta** (*float*) – the beta hyperparameters used for gating tabular data see the paper Integrating Multimodal Information in Large Pretrained Transformers for details
- **numerical_feat_dim** (*int*) – the number of numerical features
- **cat_feat_dim** (*int*) – the number of categorical features

5.3 AutoModel with Tabular

class AutoModelWithTabular

Bases: `object`

classmethod from_config(*config*)

Instantiates one of the base model classes of the library from a configuration.

Note: Only the models in multimodal_transformers.py are implemented

Parameters config (`PretrainedConfig`) –

The model class to instantiate is selected based on the configuration class: see multimodal_transformers.py for supported transformer models

Examples:

```
config = BertConfig.from_pretrained('bert-base-uncased')    # Download␣
↪configuration from S3 and cache.
model = AutoModelWithTabular.from_config(config)  # E.g. model was saved␣
↪using `save_pretrained('./test/saved_model/')`
```

classmethod from_pretrained(*pretrained_model_name_or_path*, **model_args*, ***kwargs*)

Instantiates one of the sequence classification model classes of the library from a pre-trained model configuration. See multimodal_transformers.py for supported transformer models

The *from_pretrained()* method takes care of returning the correct model class instance based on the *model_type* property of the config object, or when it's missing, falling back to using pattern matching on the *pretrained_model_name_or_path* string:

The model is set in evaluation mode by default using *model.eval()* (Dropout modules are deactivated) To train the model, you should first set it back in training mode with *model.train()*

Parameters

- **pretrained_model_name_or_path** – either:
  - a string with the *shortcut name* of a pre-trained model to load from cache or download, e.g.: `bert-base-uncased`.
  - a string with the *identifier name* of a pre-trained model that was user-uploaded to our S3, e.g.: `dbmdz/bert-base-german-cased`.
  - a path to a *directory* containing model weights saved using `save_pretrained()`, e.g.: `./my_model_directory/`.
  - a path or url to a *tensorflow index checkpoint file* (e.g. *./tf_model/model.ckpt.index*). In this case, `from_tf` should be set to True and a configuration object should be provided as `config` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
- **model_args** – (*optional*) Sequence of positional arguments: All remaining positional arguments will be passed to the underlying model's `__init__` method
- **config** – (*optional*) instance of a class derived from `PretrainedConfig`: Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

  - the model is a model provided by the library (loaded with the `shortcut-name` string of a pretrained model), or
  - the model was saved using `save_pretrained()` and is reloaded by suppling the save directory.
  - the model is loaded by suppling a local directory as `pretrained_model_name_or_path` and a configuration JSON file named *config.json* is found in the directory.
- **`state_dict`** – (*optional*) dict: an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file. This option can be used if you want to create a model from a pretrained configuration but load your own weights. In this case though, you should check if using `save_pretrained()` and `from_pretrained()` is not a simpler option.
- **`cache_dir`** – (*optional*) string: Path to a directory in which a downloaded pre-trained model configuration should be cached if the standard cache should not be used.
- **`force_download`** – (*optional*) boolean, default False: Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
- **`resume_download`** – (*optional*) boolean, default False: Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.
- **`proxies`** – (*optional*) dict, default None: A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}. The proxies are used on each request.
- **`output_loading_info`** – (*optional*) boolean: Set to `True` to also return a dictionary containing missing keys, unexpected keys and error messages.
- **`kwargs`** – (*optional*) Remaining dictionary of keyword arguments: These arguments will be passed to the configuration and the model.

Examples:

```
model = AutoModelWithTabular.from_pretrained('bert-base-uncased')    #␣
↪Download model and configuration from S3 and cache.
model = AutoModelWithTabular.from_pretrained('./test/bert_model/')  # E.g.␣
↪model was saved using `save_pretrained('./test/saved_model/')`
assert model.config.output_attention == True
# Loading from a TF checkpoint file instead of a PyTorch model (slower)
config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
model = AutoModelWithTabular.from_pretrained('./tf_model/bert_tf_checkpoint.
↪ckpt.index', from_tf=True, config=config)
```

5.4 Transformers with Tabular

class AlbertWithTabular(*hf_model_config*)

Bases: `transformers.modeling_albert.AlbertForSequenceClassification`

ALBERT Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Roberta pooled output

Parameters **`hf_model_config`** (`AlbertConfig`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a `TabularConfig` instance specifying the configs for `TabularFeatCombiner`

forward(*input_ids=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=None*, *class_weights=None*, *cat_feats=None*, *numerical_feats=None*)

The `AlbertWithTabular` forward method, overrides the `__call__()` special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **input_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using `transformers.AlbertTokenizer`. See `transformers.PreTrainedTokenizer.encode()` and `transformers.PreTrainedTokenizer()` for details.

  What are input IDs?

- **attention_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.

  What are attention masks?

- **token_type_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token

  What are token type IDs?

- **position_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range `[0, config.max_position_embeddings - 1]`.

  What are position IDs?

- **head_mask** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.

- **inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.

- **output_attentions** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.

- **labels** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `config.num_labels > 1` a classification loss is computed (Cross-Entropy).

`class BertWithTabular`(*hf_model_config*)

Bases: `transformers.modeling_bert.BertForSequenceClassification`

Bert Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Bert pooled output

Parameters **`hf_model_config`** (`BertConfig`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a `TabularConfig` instance specifying the configs for `TabularFeatCombiner`

`forward`(*input_ids=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *class_weights=None*, *output_attentions=None*, *output_hidden_states=None*, *cat_feats=None*, *numerical_feats=None*)

The `BertWithTabular` forward method, overrides the `__call__()` special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **`input_ids`** (`torch.LongTensor` of shape `(batch_size, sequence_length)`) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using `transformers.BertTokenizer`. See `transformers.PreTrainedTokenizer.encode()` and `transformers.PreTrainedTokenizer.__call__()` for details.

  What are input IDs?

- **`attention_mask`** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.

  What are attention masks?

- **`token_type_ids`** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token

  What are token type IDs?

- **`position_ids`** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range `[0, config.max_position_embeddings - 1]`.

  What are position IDs?

- **`head_mask`** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.

- **inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.
- **encoder_hidden_states** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if the model is configured as a decoder.
- **encoder_attention_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in the cross-attention if the model is configured as a decoder. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.
- **output_attentions** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.
- **class_weights** (`torch.FloatTensor` of shape `(tabular_config.num_labels,)`, *optional*, defaults to `None`) – Class weights to be used for cross entropy loss function for classification task
- **labels** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `tabular_config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `tabular_config.num_labels > 1` a classification loss is computed (Cross-Entropy).
- **cat_feats** (`torch.FloatTensor` of shape `(batch_size, tabular_config.cat_feat_dim)`, *optional*, defaults to `None`) – Categorical features to be passed in to the TabularFeatCombiner
- **numerical_feats** (`torch.FloatTensor` of shape `(batch_size, tabular_config.numerical_feat_dim)`, *optional*, defaults to `None`) – Numerical features to be passed in to the TabularFeatCombiner

Returns

loss (`torch.FloatTensor` of shape `(1,)`, *optional*, returned when `label` is provided): Classification (or regression if tabular_config.num_labels==1) loss.

logits (`torch.FloatTensor` of shape `(batch_size, tabular_config.num_labels)`): Classification (or regression if tabular_config.num_labels==1) scores (before SoftMax).

classifier_layer_outputs(`list` of `torch.FloatTensor`): The outputs of each layer of the final classification layers. The 0th index of this list is the combining module's output

Return type `tuple` comprising various elements depending on configuration and inputs

class DistilBertWithTabular(*hf_model_config*)

Bases: `transformers.modeling_distilbert.DistilBertForSequenceClassification`

DistilBert Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Roberta pooled output

Parameters **`hf_model_config`** (`DistilBertConfig`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a `TabularConfig` instance specifying the configs for `TabularFeatCombiner`

`forward`(*input_ids=None*, *attention_mask=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *output_attentions=None*, *output_hidden_states=None*, *class_weights=None*, *cat_feats=None*, *numerical_feats=None*)

The `DistilBertWithTabular` forward method, overrides the `__call__()` special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **`input_ids`** (`torch.LongTensor` of shape `(batch_size, sequence_length)`) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using `transformers.DistilBertTokenizer`. See `transformers.PreTrainedTokenizer.encode()` and `transformers.PreTrainedTokenizer.__call__()` for details.

  What are input IDs?

- **`attention_mask`** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.

  What are attention masks?

- **`head_mask`** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.

- **`inputs_embeds`** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.

- **`output_attentions`** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.

- **`class_weights`** (`torch.FloatTensor` of shape `(tabular_config.num_labels,)`,`optional`, defaults to `None`) – Class weights to be used for cross entropy loss function for classification task

- **`labels`** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `tabular_config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `tabular_config.num_labels > 1` a classification loss is computed (Cross-Entropy).

- **cat_feats** (torch.FloatTensor of shape (batch_size, tabular_config.cat_feat_dim),`optional`, defaults to None) – Categorical features to be passed in to the TabularFeatCombiner
- **numerical_feats** (torch.FloatTensor of shape (batch_size, tabular_config.numerical_feat_dim),`optional`, defaults to None) – Numerical features to be passed in to the TabularFeatCombiner

Returns

loss (torch.FloatTensor of shape (1,), *optional*, returned when label is provided): Classification (or regression if tabular_config.num_labels==1) loss.

logits (torch.FloatTensor of shape (batch_size, tabular_config.num_labels)): Classification (or regression if tabular_config.num_labels==1) scores (before SoftMax).

classifier_layer_outputs(list of torch.FloatTensor): The outputs of each layer of the final classification layers. The 0th index of this list is the combining module's output

Return type tuple comprising various elements depending on configuration and inputs

class RobertaWithTabular(*hf_model_config*)

Bases: transformers.modeling_roberta.RobertaForSequenceClassification

Roberta Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Roberta pooled output

Parameters hf_model_config (RobertaConfig) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a TabularConfig instance specifying the configs for TabularFeatCombiner

forward(*input_ids=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *output_attentions=None*, *output_hidden_states=None*, *class_weights=None*, *cat_feats=None*, *numerical_feats=None*)

The RobertaWithTabular forward method, overrides the __call__() special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **input_ids** (torch.LongTensor of shape (batch_size, sequence_length)) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using transformers.RobertaTokenizer. See transformers.PreTrainedTokenizer.encode() and transformers.PreTrainedTokenizer.__call__() for details.

  What are input IDs?

- **attention_mask** (torch.FloatTensor of shape (batch_size, sequence_length), *optional*, defaults to None) – Mask to avoid performing attention on padding token indices. Mask values selected in [0, 1]: 1 for tokens that are NOT MASKED, 0 for MASKED tokens.

  What are attention masks?

- **token_type_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token

  What are token type IDs?

- **position_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range `[0, config.max_position_embeddings - 1]`.

  What are position IDs?

- **head_mask** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.

- **inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.

- **output_attentions** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.

- **class_weights** (`torch.FloatTensor` of shape `(tabular_config.num_labels,)`, *optional*, defaults to `None`) – Class weights to be used for cross entropy loss function for classification task

- **labels** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `tabular_config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `tabular_config.num_labels > 1` a classification loss is computed (Cross-Entropy).

- **cat_feats** (`torch.FloatTensor` of shape `(batch_size, tabular_config.cat_feat_dim)`, *optional*, defaults to `None`) – Categorical features to be passed in to the TabularFeatCombiner

- **numerical_feats** (`torch.FloatTensor` of shape `(batch_size, tabular_config.numerical_feat_dim)`, *optional*, defaults to `None`) – Numerical features to be passed in to the TabularFeatCombiner

Returns

loss (`torch.FloatTensor` of shape `(1,)`, *optional*, returned when `label` is provided):
Classification (or regression if tabular_config.num_labels==1) loss.

logits (`torch.FloatTensor` of shape `(batch_size, tabular_config.num_labels)`):
Classification (or regression if tabular_config.num_labels==1) scores (before SoftMax).

classifier_layer_outputs(`list` of `torch.FloatTensor`): The outputs of each layer of the final classification layers. The 0th index of this list is the combining module's output

Return type `tuple` comprising various elements depending on configuration and inputs

class XLMRobertaWithTabular(*hf_model_config*)

Bases: *multimodal_transformers.model.tabular_transformers.RobertaWithTabular*

This class overrides *RobertaWithTabular*. Please check the superclass for the appropriate documentation alongside usage examples.

config_class

alias of `transformers.configuration_xlm_roberta.XLMRobertaConfig`

class XLMWithTabular(*hf_model_config*)

Bases: `transformers.modeling_xlm.XLMForSequenceClassification`

XLM Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Roberta pooled output

Parameters hf_model_config (`XLMConfig`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a `TabularConfig` instance specifying the configs for `TabularFeatCombiner`

forward(*input_ids=None*, *attention_mask=None*, *langs=None*, *token_type_ids=None*, *position_ids=None*, *lengths=None*, *cache=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=None*, *class_weights=None*, *cat_feats=None*, *numerical_feats=None*)

The `XLMWithTabular` forward method, overrides the `__call__()` special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **input_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using `transformers.BertTokenizer`. See `transformers.PreTrainedTokenizer.encode()` and `transformers.PreTrainedTokenizer.__call__()` for details.

  What are input IDs?

- **attention_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.

  What are attention masks?

- **langs** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – A parallel sequence of tokens to be used to indicate the language of each token in the input. Indices are languages ids which can be obtained from the language names by using two conversion mappings provided in the configuration of the model (only provided for multilingual models). More precisely, the *language name -> language id* mapping is in *model.config.lang2id* (dict str -> int) and the *language id -> language name* mapping is *model.config.id2lang* (dict int -> str).

  See usage examples detailed in the multilingual documentation.

- **token_type_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token

What are token type IDs?

- **`position_ids`** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range `[0, config.max_position_embeddings - 1]`.

  What are position IDs?

- **`lengths`** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Length of each sentence that can be used to avoid performing attention on padding token indices. You can also use *attention_mask* for the same result (see above), kept here for compatbility. Indices selected in `[0, ..., input_ids.size(-1)]`:
- **`cache`** (`Dict[str, torch.FloatTensor]`, *optional*, defaults to `None`) – dictionary with `torch.FloatTensor` that contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model (see *cache* output below). Can be used to speed up sequential decoding. The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
- **`head_mask`** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.
- **`inputs_embeds`** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.
- **`output_attentions`** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.
- **`labels`** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `config.num_labels > 1` a classification loss is computed (Cross-Entropy).

class XLNetWithTabular(*hf_model_config*)

Bases: `transformers.modeling_xlnet.XLNetForSequenceClassification`

XLNet Model transformer with a sequence classification/regression head as well as a TabularFeatCombiner module to combine categorical and numerical features with the Roberta pooled output

Parameters **`hf_model_config`** (`XLNetConfig`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a `TabularConfig` instance specifying the configs for `TabularFeatCombiner`

forward(*input_ids=None*, *attention_mask=None*, *mems=None*, *perm_mask=None*, *target_mapping=None*, *token_type_ids=None*, *input_mask=None*, *head_mask=None*, *inputs_embeds=None*, *labels=None*, *use_cache=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=None*, *class_weights=None*, *cat_feats=None*, *numerical_feats=None*)

The `XLNetWithTabular` forward method, overrides the `__call__()` special method.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Parameters

- **input_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using `transformers.BertTokenizer`. See `transformers.PreTrainedTokenizer.encode()` and `transformers.PreTrainedTokenizer.__call__()` for details.

  What are input IDs?

- **attention_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`: `1` for tokens that are NOT MASKED, `0` for MASKED tokens.

  What are attention masks?

- **mems** (`List[torch.FloatTensor]` of length `config.n_layers`) – Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model (see *mems* output below). Can be used to speed up sequential decoding. The token ids which have their mems given to this model should not be passed as input ids as they have already been computed. *use_cache* has to be set to *True* to make use of *mems*.

- **perm_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length, sequence_length)`, *optional*, defaults to `None`) – Mask to indicate the attention pattern for each input token with values selected in `[0, 1]`: If `perm_mask[k, i, j] = 0`, i attend to j in batch k; if `perm_mask[k, i, j] = 1`, i does not attend to j in batch k. If None, each token attends to all the others (full bidirectional attention). Only used during pretraining (to define factorization order) or for sequential decoding (generation).

- **target_mapping** (`torch.FloatTensor` of shape `(batch_size, num_predict, sequence_length)`, *optional*, defaults to `None`) – Mask to indicate the output tokens to use. If `target_mapping[k, i, j] = 1`, the i-th predict in batch k is on the j-th token. Only used during pretraining for partial prediction or for sequential decoding (generation).

- **token_type_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token. The classifier token should be represented by a `2`.

  What are token type IDs?

- **input_mask** (`torch.FloatTensor` of shape `(batch_size, sequence_length)`, *optional*, defaults to `None`) – Mask to avoid performing attention on padding token indices. Negative of *attention_mask*, i.e. with 0 for real tokens and 1 for padding. Kept for compatibility with the original code base. You can only uses one of *input_mask* and *attention_mask* Mask values selected in `[0, 1]`: `1` for tokens that are MASKED, `0` for tokens that are NOT MASKED.

- **head_mask** (`torch.FloatTensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*, defaults to `None`) – Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`: `1` indicates the head is **not masked**, `0` indicates the head is **masked**.
- **inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*, defaults to `None`) – Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.
- **use_cache** (`bool`) – If *use_cache* is True, *mems* are returned and can be used to speed up decoding (see *mems*). Defaults to *True*.
- **output_attentions** (`bool`, *optional*, defaults to `None`) – If set to `True`, the attentions tensors of all attention layers are returned. See `attentions` under returned tensors for more detail.
- **labels** (`torch.LongTensor` of shape `(batch_size,)`, *optional*, defaults to `None`) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. If `config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `config.num_labels > 1` a classification loss is computed (Cross-Entropy).

CHAPTER

SIX

MULTIMODAL_TRANSFORMERS.DATA

The data module includes two functions to help load your own datasets into `multimodal_transformers.data.tabular_torch_dataset.TorchTabularTextDataset` which can be fed into a `torch.utils.data.DataLoader`. The `multimodal_transformers.data.tabular_torch_dataset.TorchTabularTextDataset`'s `__getitem__` method's outputs can be directly fed to the forward pass to a model in *`multimodal_transformers.model.tabular_transformers`*.

Note: You may still need to move the `__getitem__` method outputs to the right gpu device.

6.1 Module contents

class TorchTabularTextDataset(*encodings*, *categorical_feats*, *numerical_feats*, *labels=None*, *df=None*, *label_list=None*, *class_weights=None*)

Bases: `torch.utils.data.dataset.Dataset`

`TorchDataset` wrapper for text dataset with categorical features and numerical features

Parameters

- **encodings** (`transformers.BatchEncoding`) – The output from encode_plus() and batch_encode() methods (tokens, attention_masks, etc) of a transformers.PreTrainedTokenizer
- **categorical_feats** (`numpy.ndarray`, of shape `(n_examples, categorical feat dim)`, *optional*, defaults to `None`) – An array containing the preprocessed categorical features
- **numerical_feats** (`numpy.ndarray`, of shape `(n_examples, numerical feat dim)`, *optional*, defaults to `None`) – An array containing the preprocessed numerical features
- **(** (*`labels`*) – class: list` or *numpy.ndarray*, *optional*, defaults to `None`): The labels of the training examples
- **class_weights** (`numpy.ndarray`, of shape (n_classes), *optional*, defaults to `None`) – Class weights used for cross entropy loss for classification
- **df** (`pandas.DataFrame`, *optional*, defaults to `None`) – Model configuration class with all the parameters of the model. This object must also have a tabular_config member variable that is a TabularConfig instance specifying the configs for TabularFeatCombiner

get_labels()

returns the label names for classification

load_data(*data_df*, *text_cols*, *tokenizer*, *label_col*, *label_list=None*, *categorical_cols=None*, *numerical_cols=None*, *sep_text_token_str=' '*, *categorical_encode_type='ohe'*, *numerical_transformer=None*, *empty_text_values=None*, *replace_empty_text=None*, *max_token_length=None*, *debug=False*)

Function to load a single dataset given a pandas DataFrame

Given a DataFrame, this function loads the data to a `torch_dataset.TorchTextDataset` object which can be used in a `torch.utils.data.DataLoader`.

Parameters

- **data_df** (`pd.DataFrame`) – The DataFrame to convert to a TorchTextDataset
- **text_cols** (`list` of `str`) – the column names in the dataset that contain text from which we want to load
- **tokenizer** (`transformers.tokenization_utils.PreTrainedTokenizer`) – HuggingFace tokenizer used to tokenize the input texts as specifed by text_cols
- **label_col** (*`str`*) – The column name of the label, for classification the column should have int values from 0 to n_classes-1 as the label for each class. For regression the column can have any numerical value
- **label_list** (`list` of `str`, optional) – Used for classification; the names of the classes indexed by the values in label_col.
- **categorical_cols** (`list` of `str`, optional) – The column names in the dataset that contain categorical features. The features can be already prepared numerically, or could be preprocessed by the method specified by categorical_encode_type
- **numerical_cols** (`list` of `str`, optional) – The column names in the dataset that contain numerical features. These columns should contain only numeric values.
- **sep_text_token_str** (*`str, optional`*) – The string token that is used to separate between the different text columns for a given data example. For Bert for example, this could be the [SEP] token.
- **categorical_encode_type** (*`str, optional`*) – Given categorical_cols, this specifies what method we want to preprocess our categorical features. choices: ['ohe', 'binary', None] see encode_features.CategoricalFeatures for more details
- **numerical_transformer** (`sklearn.base.TransformerMixin`) – The sklearn numeric transformer instance to transform our numerical features
- **empty_text_values** (`list` of `str`, optional) – Specifies what texts should be considered as missing which would be replaced by replace_empty_text
- **replace_empty_text** (*`str, optional`*) – The value of the string that will replace the texts that match with those in empty_text_values. If this argument is None then the text that match with empty_text_values will be skipped
- **max_token_length** (*`int, optional`*) – The token length to pad or truncate to on the input text
- **debug** (*`bool, optional`*) – Whether or not to load a smaller debug version of the dataset

Returns The converted dataset

Return type `tabular_torch_dataset.TorchTextDataset`

load_data_from_folder(*folder_path*, *text_cols*, *tokenizer*, *label_col*, *label_list=None*, *categorical_cols=None*, *numerical_cols=None*, *sep_text_token_str=' '*, *categorical_encode_type='ohe'*, *numerical_transformer_method='quantile_normal'*, *empty_text_values=None*, *replace_empty_text=None*, *max_token_length=None*, *debug=False*)

Function to load tabular and text data from a specified folder

Loads train, test and/or validation text and tabular data from specified folder path into TorchTextDataset class and does categorical and numerical data preprocessing if specified. Inside the folder, there is expected to be a train.csv, and test.csv (and if given val.csv) containing the training, testing, and validation sets respectively

Parameters

- **folder_path** (*str*) – The path to the folder containing *train.csv*, and *test.csv* (and if given *val.csv*)
- **text_cols** (`list` of `str`) – The column names in the dataset that contain text from which we want to load
- **tokenizer** (`transformers.tokenization_utils.PreTrainedTokenizer`) – HuggingFace tokenizer used to tokenize the input texts as specifed by text_cols
- **label_col** (*str*) – The column name of the label, for classification the column should have int values from 0 to n_classes-1 as the label for each class. For regression the column can have any numerical value
- **label_list** (`list` of `str`, optional) – Used for classification; the names of the classes indexed by the values in label_col.
- **categorical_cols** (`list` of `str`, optional) – The column names in the dataset that contain categorical features. The features can be already prepared numerically, or could be preprocessed by the method specified by categorical_encode_type
- **numerical_cols** (`list` of `str`, optional) – The column names in the dataset that contain numerical features. These columns should contain only numeric values.
- **sep_text_token_str** (*str, optional*) – The string token that is used to separate between the different text columns for a given data example. For Bert for example, this could be the [SEP] token.
- **categorical_encode_type** (*str, optional*) – Given categorical_cols, this specifies what method we want to preprocess our categorical features. choices: ['ohe', 'binary', None] see encode_features.CategoricalFeatures for more details
- **numerical_transformer_method** (*str, optional*) – Given numerical_cols, this specifies what method we want to use for normalizing our numerical data. choices: ['yeo_johnson', 'box_cox', 'quantile_normal', None] see https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html for more details
- **empty_text_values** (`list` of `str`, optional) – specifies what texts should be considered as missing which would be replaced by replace_empty_text
- **replace_empty_text** (*str, optional*) – The value of the string that will replace the texts that match with those in empty_text_values. If this argument is None then the text that match with empty_text_values will be skipped
- **max_token_length** (*int, optional*) – The token length to pad or truncate to on the input text
- **debug** (*bool, optional*) – Whether or not to load a smaller debug version of the dataset

Returns This tuple contains the training, validation and testing sets. The val dataset is `None` if there is no *val.csv* in folder_path

Return type `tuple` of *tabular_torch_dataset.TorchTextDataset*

`load_data_into_folds`(*data_csv_path*, *num_splits*, *validation_ratio*, *text_cols*, *tokenizer*, *label_col*, *label_list=None*, *categorical_cols=None*, *numerical_cols=None*, *sep_text_token_str=' '*, *categorical_encode_type='ohe'*, *numerical_transformer_method='quantile_normal'*, *empty_text_values=None*, *replace_empty_text=None*, *max_token_length=None*, *debug=False*)

Function to load tabular and text data from a specified folder into folds

Loads train, test and/or validation text and tabular data from specified csv path into num_splits of train, val and test for Kfold cross validation. Performs categorical and numerical data preprocessing if specified. *data_csv_path* is a path to

Parameters

- **`data_csv_path`** (*`str`*) – The path to the csv containing the data
- **`num_splits`** (*`int`*) – The number of cross validation folds to split the data into.
- **`validation_ratio`** (*`float`*) – A float between 0 and 1 representing the percent of the data to hold as a consistent validation set.
- **`text_cols`** (`list` of `str`) – The column names in the dataset that contain text from which we want to load
- **`tokenizer`** (`transformers.tokenization_utils.PreTrainedTokenizer`) – HuggingFace tokenizer used to tokenize the input texts as specifed by text_cols
- **`label_col`** (*`str`*) – The column name of the label, for classification the column should have int values from 0 to n_classes-1 as the label for each class. For regression the column can have any numerical value
- **`label_list`** (`list` of `str`, optional) – Used for classification; the names of the classes indexed by the values in label_col.
- **`categorical_cols`** (`list` of `str`, optional) – The column names in the dataset that contain categorical features. The features can be already prepared numerically, or could be preprocessed by the method specified by categorical_encode_type
- **`numerical_cols`** (`list` of `str`, optional) – The column names in the dataset that contain numerical features. These columns should contain only numeric values.
- **`sep_text_token_str`** (*`str, optional`*) – The string token that is used to separate between the different text columns for a given data example. For Bert for example, this could be the [SEP] token.
- **`categorical_encode_type`** (*`str, optional`*) – Given categorical_cols, this specifies what method we want to preprocess our categorical features. choices: ['ohe', 'binary', None] see encode_features.CategoricalFeatures for more details
- **`numerical_transformer_method`** (*`str, optional`*) – Given numerical_cols, this specifies what method we want to use for normalizing our numerical data. choices: ['yeo_johnson', 'box_cox', 'quantile_normal', None] see https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html for more details
- **`empty_text_values`** (`list` of `str`, optional) – specifies what texts should be considered as missing which would be replaced by replace_empty_text

- **`replace_empty_text`** (*`str, optional`*) – The value of the string that will replace the texts that match with those in empty_text_values. If this argument is None then the text that match with empty_text_values will be skipped
- **`max_token_length`** (*`int, optional`*) – The token length to pad or truncate to on the input text
- **`debug`** (*`bool, optional`*) – Whether or not to load a smaller debug version of the dataset

Returns This tuple contains three lists representing the splits of training, validation and testing sets. The length of the lists is equal to the number of folds specified by *num_splits*

Return type `tuple` of *list* of *tabular_torch_dataset.TorchTextDataset*

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX