PoliStance_Supervised_Training

March 28, 2024

0.1 PoliStance: Supervised Training

This tutorial demonstrates how to train an NLI classifier as a supervised classifier. It uses the PoliStance model on the HuggingFace Hub, a DeBERTAv3 model trained for political stance classification. The base model should be sufficient for such tasks, although the large model may provide a benefit in instances with a low number of training samples.

[ ]:
```
from datasets import Dataset, DatasetDict # The datasets library allows us to␣
↪import the data directly from the huggingface hub and puts it in an␣
↪efficient format for training
from transformers import AutoTokenizer, AutoModelForSequenceClassification,␣
↪TrainingArguments, Trainer # Transformers has the classes and functions for␣
↪training the model
import torch # Transformers is built on top of the pytorch library. This also␣
↪allows us to interact with the GPU and check its availability.

# numpy, pandas, and sklearn are used for data manipulation and performance␣
↪metrics.
import numpy as np
from sklearn.metrics import balanced_accuracy_score,␣
↪precision_recall_fscore_support, accuracy_score, classification_report
import pandas as pd
```

We have three different components for this task:

1. The dataset
2. The tokenizer
3. The model

The dataset contains our training and testing data. The tokenizer will convert the dataset into numeric representations of the tokens that will be passed to the model during training. The tokenizer doesn't need to be trained, and is just for preparing the dataset to be passed to the model.

[ ]:
```
# in this block we are just defining variables that will later be passed to␣
↪other functions.
# first we define the model. the name of the model on the HuggingFace directory
modname = "XXXXX/deberta-v3-base-polistance-affect-v1.0"
# the directory where you want the model checkpoints to save
```

```
training_directory ='training_base'
# use GPU if one is available, else CPU. You will want GPU access for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Device: {device}")
```

[ ]:
```
# import example train and test data as pandas dataframes. Generally you want a␣
↪train, validate, and test set
train = pd.read_csv('https://raw.githubusercontent.com/XXXXX/
↪stance_detection_tutorials/main/data/train.csv')
validate = pd.read_csv('https://raw.githubusercontent.com/XXXXX/
↪stance_detection_tutorials/main/data/test.csv')

# convert the data to a huggingface dataset for ease of use
tr_ds = Dataset.from_pandas(train)
val_ds = Dataset.from_pandas(validate)
ds = DatasetDict()
ds['train'] = tr_ds
ds['validate'] = val_ds
```

[ ]:
```
# import the tokenizer
tokenizer = AutoTokenizer.from_pretrained(modname)

# define a generic tokenizing function
def tokenize_function(docs):
    return tokenizer(docs['text'], padding = 'max_length', truncation = True)

# tokenize the dataset
dstok = ds.map(tokenize_function)
```

[ ]:
```
# import the model. Change the num_labels variable to match the number of␣
↪classes in your dataset. If the number of labels is different than 2
# then the ignore_mismatched_sizes must be true. This tells the model to␣
↪replace the classifier head of the neural network with a new one that has␣
↪the appropriate number of labels.
# id2label makes sure the model associates 0 with 'not support' and 1 with␣
↪'support'. Change this to whatever is appropriate for how many labels you␣
↪have and what they represent.
model = AutoModelForSequenceClassification.from_pretrained(modname, num_labels␣
↪= 2, ignore_mismatched_sizes=True, id2label = {0:'not support', 1:'support'})
```

Now that we've imported the model we can set our training parameters and define how the model will be evaluated during training.

[ ]:
```
# Now we define all of the training parameters
training_args = TrainingArguments(output_dir=training_directory,
    logging_dir=f'{training_directory}/logs',
```

```
    lr_scheduler_type= "linear", # The algorithm that will adjust the learning␣
↪rate while training
    group_by_length=False,  # If set to True, can increase speed with dynamic␣
↪padding, by grouping similar length texts
    learning_rate = 2e-5, # the initial learning rate
    per_device_train_batch_size = 16, # batch size controls how many documents␣
↪are passed through the model at once. Higher batch sizes train faster but␣
↪demand more memory. lower the batch size if you are running out of memory
    per_device_eval_batch_size = 16,
    gradient_accumulation_steps= 1,  # Number of batches to pass through the␣
↪model before updating the weights of the neural network. Can be useful when␣
↪using very small batch sizes like 2 or 4.
    num_train_epochs=3, # number of times to pass the entire training set␣
↪through the model
    warmup_ratio=0.06, # warmup length before learning rate scheduler kicks in
    weight_decay=0.01, # weight regularization
    fp16=True, # the data type that the model's weights are stored in. fp16␣
↪stands for floating point 16 and will make the model much smaller and faster.
    fp16_full_eval=True,
    evaluation_strategy="epoch", # evaluate the model every n steps or epochs.
    seed=1,
    #eval_steps=50,  # how many steps between evaluations if using steps␣
↪evaluation strategy. 1 step = 1 gradient update
    save_strategy="epoch",  # Save after each epoch or after n steps
    #save_steps=100,  # Number of updates steps before two checkpoint saves.
    dataloader_num_workers = 1, # number of cpu workers passing data to the the␣
↪GPU
)
```

Below is a custom function that can be passed to the trainer and will report a battery of metrics to report while training.

[ ]:
```
# this function will be used to calculate performance metrics during training
def compute_metrics(eval_pred, label_text_alphabetical=list(model.config.
↪id2label.values())):
    # Extract labels
    labels = eval_pred.label_ids
    pred_logits = eval_pred.predictions
    preds_max = np.argmax(pred_logits, axis=1)

    # Compute the metrics
    precision_macro, recall_macro, f1_macro, _ =␣
↪precision_recall_fscore_support(labels, preds_max, average='macro')
    precision_micro, recall_micro, f1_micro, _ =␣
↪precision_recall_fscore_support(labels, preds_max, average='micro')
    acc_balanced = balanced_accuracy_score(labels, preds_max)
    acc_not_balanced = accuracy_score(labels, preds_max)
```

```
    # Pass computed metrics to a dictionary for printing
    metrics = {'f1_macro': f1_macro,
            'f1_micro': f1_micro,
            'accuracy_balanced': acc_balanced,
            'accuracy': acc_not_balanced,
            'precision_macro': precision_macro,
            'recall_macro': recall_macro,
            'precision_micro': precision_micro,
            'recall_micro': recall_micro,
            }

    # Print results
    print("Aggregate metrics: ", {key: metrics[key] for key in metrics if key␣
 ↪not in ["label_gold_raw", "label_predicted_raw"]} )
    print("Detailed metrics: ", classification_report(
        labels, preds_max, labels=np.sort(pd.factorize(label_text_alphabetical,␣
 ↪sort=True)[0]),
        target_names=label_text_alphabetical, sample_weight=None,
        digits=2, output_dict=True, zero_division='warn'),
    "\n")

    return metrics
```

Now that we've prepared everything, we just pass the model, tokenizer, dataset, training parameters, and metrics function to the trainer. Then we simply call the trainer to start the process.

[ ]:
```
# instantiate the trainer by passing our model, tokenizer, training parameters,␣
 ↪data, and metrics function to it
trainer = Trainer(
    model=model,
    tokenizer=tokenizer,
    args=training_args,
    train_dataset=dstok['train'],
    eval_dataset=dstok['validate'],
    compute_metrics=lambda x: compute_metrics(x,␣
 ↪label_text_alphabetical=list(model.config.id2label.values()))
)
```

[ ]:
```
# call the trainer to train the model
trainer.train()
```