Fine-Tuning DistilBERT for Emotion Classification

Ahmet Taşdemir · Following
8 min read · Jun 14

In this post, we will walk through the process of fine-tuning the DistilBERT model for emotion classification. Emotion classification is a common task in natural language processing (NLP) where we aim to classify text into different emotion categories such as joy, sadness, love, anger, fear, and surprise. We will use the Hugging Face library and the “emotions” dataset for training and evaluation.

[Github repo]

Model Link Hugging Face

Dataset Exploration

Before diving into the fine-tuning process, let's explore the dataset and understand its structure. We will start by loading the "emotions" dataset and examining its properties.

```
from datasets import load_dataset

emotions = load_dataset("emotion")

train_ds = emotions["train"]

print(train_ds.features)
print(train_ds[:5])
print(train_ds["text"][:5])
```

The above code snippet loads the "emotions" dataset and retrieves the training split. We print the dataset's features, which include the "text" and "label" columns. Next, we display the first five examples from the dataset and the corresponding text. This helps us get a sense of the data we'll be working with.

Converting to DataFrames

To facilitate data manipulation and visualization, we convert the dataset into a pandas DataFrame. This allows us to perform various operations easily.

```
import pandas as pd

emotions.set_format(type="pandas")
```

```
df = emotions["train"][:]
df.head()
```

We use the `set_format` method to convert the dataset to a pandas DataFrame format. Then, we create a DataFrame `df` from the "train" split of the dataset. We print the first few rows of the DataFrame to examine the structure.

Examining Class Distribution

Understanding the class distribution in a text classification problem is crucial. We want to ensure that our dataset is balanced across different emotion categories. If the class distribution is imbalanced, it might affect the training process and the evaluation metrics.

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Frequency of Classes")
plt.show()
```

In this code snippet, we plot a horizontal bar chart to visualize the frequency of each emotion class. This helps us identify any class imbalance issues. In our case, we observe that the dataset is heavily imbalanced, with “joy” and “sadness” classes appearing more frequently than “love” and “surprise.” Dealing with imbalanced data is important, and there are several techniques available, such as oversampling, undersampling, or gathering more labeled data for underrepresented classes. However, for simplicity, we will work with the raw, unbalanced class frequencies in this blog post.

Analyzing Text Length

Understanding the distribution of text lengths in our dataset can provide insights into the nature of the data and potential challenges we may encounter during training.

```
df["Words Per Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Words Per Tweet", by="label_name", grid=False, showfliers=False, col
plt.suptitle("")
plt.xlabel("")
plt.show()
```

In the above code snippet, we calculate the number of words per tweet by splitting the text on whitespace and applying the `len` function. We then create a boxplot to visualize the distribution of the number of words per tweet for each emotion class. This helps us understand if there are any significant differences in text lengths across classes.

Tokenization

Tokenization is a crucial step in NLP tasks, where we convert text into a sequence of tokens that can be processed by the model. In this case, we use the DistilBERT tokenizer to tokenize our text data.

```
from transformers import AutoTokenizer

model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

def tokenize(batch):
    return tokenizer(batch["text"], padding=True, truncation=True)
```

```
emotions_encoded = emotions.map(tokenize, batched=True)
```

We initialize the DistilBERT tokenizer using the "distilbert-base-uncased" checkpoint. We define a `tokenize` function that takes a batch of texts and applies tokenization with padding and truncation. We then use the `map` method from the `emotions` dataset to tokenize the text data in batches.

Model Initialization and Configuration

To perform fine-tuning, we need to initialize the pre-trained DistilBERT model with a classification head. We also define the device to use for training, the number of labels, and the mapping between label indices and emotion names.

```
from transformers import AutoModelForSequenceClassification
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_labels = 6
id2label = {
    "0": "sadness",
    "1": "joy",
    "2": "love",
    "3": "anger",
    "4": "fear",
    "5": "surprise"
}
label2id = {
    "sadness": 0,
    "joy": 1,
    "love": 2,
    "anger": 3,
    "fear": 4,
    "surprise": 5
}
```

```
model = AutoModelForSequenceClassification.from_pretrained(model_ckpt, num_label
```

In the above code, we create an instance of the DistilBERT model for sequence classification using the `AutoModelForSequenceClassification` class from the transformers library. We pass the pre-trained checkpoint "distilbert-base-uncased" as the `from_pretrained` argument. We also specify the number of labels and the mapping between label indices and emotion names.

Training Configuration and Initialization

To fine-tune the model, we need to define the training configuration and initialize the `Trainer` object.

```
from transformers import Trainer, TrainingArguments

batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
model_name = f"{model_ckpt}-finetuned-emotion"
training_args = TrainingArguments(
    output_dir=model_name,
    num_train_epochs=2,
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    disable_tqdm=False,
    logging_steps=logging_steps,
    push_to_hub=True,
    log_level="error"
)

trainer = Trainer(
    model=model,
    args=training_args,
```

```
    compute_metrics=compute_metrics,
    train_dataset=emotions_encoded["train"],
    eval_dataset=emotions_encoded["validation"],
    tokenizer=tokenizer
)
```

In the code snippet above, we define the training arguments, including the output directory, the number of training epochs, the learning rate, batch sizes, weight decay, evaluation strategy, and logging settings. We create a `Trainer` object with the model, training arguments, metric computation function, training and validation datasets, and tokenizer.

We are now ready to start the fine-tuning process using the `Trainer` object

```
trainer.train()
```

Fine-Tuning DistilBERT for Emotion Classification

In this notebook, we will walk through the process of fine-tuning the DistilBERT model for emotion classification. Emotion classification is a common task in natural language processing (NLP) where we aim to classify text into different emotion categories such as joy, sadness, love, anger, fear, and surprise. We will use the Hugging Face library and the "emotions" dataset for training and evaluation.

Dataset Exploration

Before diving into the fine-tuning process, let's explore the dataset and understand its structure. We will start by loading the "emotions" dataset and examining its properties.

```
from datasets import load_dataset
```

```
emotions = load_dataset("emotion")

train_ds = emotions["train"]

print(train_ds.features)
print(train_ds[:5])
print(train_ds["text"][:5])
```

The above code snippet loads the “emotions” dataset and retrieves the training split. We print the dataset’s features, which include the “text” and “label” columns. Next, we display the first five examples from the dataset and the corresponding text. This helps us get a sense of the data we’ll be working with.

Converting to DataFrames

To facilitate data manipulation and visualization, we convert the dataset into a pandas DataFrame. This allows us to perform various operations easily.

```
import pandas as pd
```

```
emotions.set_format(type="pandas")
df = emotions["train"][:]
df.head()
```

We use the `set_format` method to convert the dataset to a pandas DataFrame format. Then, we create a DataFrame `df` from the "train" split of the dataset. We print the first few rows of the DataFrame to examine the structure.

Examining Class Distribution

Understanding the class distribution in a text classification problem is crucial. We want to ensure that our dataset is balanced across different emotion categories. If the class distribution is imbalanced, it might affect the training process and the evaluation metrics.

```
import matplotlib.pyplot as plt
```

```
df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Frequency of Classes")
plt.show()
```

In this code snippet, we plot a horizontal bar chart to visualize the frequency of each emotion class. This helps us identify any class imbalance issues. In our case, we observe that the dataset is heavily imbalanced, with "joy" and "sadness" classes appearing more frequently than "love" and "surprise." Dealing with imbalanced data is important, and there are several techniques available, such as oversampling, undersampling, or gathering more labeled data for underrepresented classes. However, for simplicity, we will work with the raw, unbalanced class frequencies in this blog post.

Analyzing Text Length

Understanding the distribution of text lengths in our dataset can provide insights into the nature of the data and potential challenges we may encounter during training.

```
df["Words Per Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Words Per Tweet", by="label_name", grid=False, showfliers=False, col
plt.suptitle("")
plt.xlabel("")
plt.show()
```

In the above code snippet, we calculate the number of words per tweet by splitting the text on whitespace and applying the `len` function. We then create a boxplot to visualize the distribution of the number of words per tweet for each emotion class. This helps us understand if there are any significant differences in text lengths across classes.

Tokenization

Tokenization is a crucial step in NLP tasks, where we convert text into a sequence of tokens that can be processed by the model. In this case, we use the DistilBERT tokenizer to tokenize our text data.

```
from transformers import AutoTokenizer
```

```
model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

def tokenize(batch):
    return tokenizer(batch["text"], padding=True, truncation=True)

emotions_encoded = emotions.map(tokenize, batched=True)
```

We initialize the DistilBERT tokenizer using the “distilbert-base-uncased” checkpoint. We define a `tokenize` function that takes a batch of texts and applies tokenization with padding and truncation. We then use the `map` method from the `emotions` dataset to tokenize the text data in batches.

Model Initialization and Configuration

To perform fine-tuning, we need to initialize the pre-trained DistilBERT model with a classification head. We also define the device to use for training, the number of labels, and the mapping between label indices and emotion names.

```
from transformers import AutoModelForSequenceClassification
import torch
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_labels = 6
id2label = {
    "0": "sadness",
    "1": "joy",
    "2": "love",
    "3": "anger",
    "4": "fear",
    "5": "surprise"
}
label2id = {
    "sadness": 0,
    "joy": 1,
    "love": 2,
    "anger": 3,
    "fear": 4,
    "surprise": 5
}

model =
AutoModelForSequenceClassification.from_pretrained(model_ckpt,
num_labels=num_labels, id2label=id2label,
label2id=label2id).to(device)
```

In the above code, we create an instance of the DistilBERT model for sequence classification using the `AutoModelForSequenceClassification` class from the transformers library. We pass the pre-trained checkpoint "distilbert-base-uncased" as the `from_pretrained` argument. We also specify the number of labels and the mapping between label indices and emotion names.

Training Configuration and Initialization

To fine-tune the model, we need to define the training configuration and initialize the `Trainer` object.

```
from transformers import Trainer, TrainingArguments
```

```
batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
model_name = f"{model_ckpt}-finetuned-emotion"
training_args = TrainingArguments(
    output_dir=model_name,
    num_train_epochs=2,
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    disable_tqdm=False,
    logging_steps=logging_steps,
    push_to_hub=True,
    log_level="error"
)

trainer = Trainer(
    model=model,
    args=training_args,
    compute_metrics=compute_metrics,
    train_dataset=emotions_encoded["train"],
    eval_dataset=emotions_encoded["validation"],
```

```
        tokenizer=tokenizer
    )
```

In the code snippet above, we define the training arguments, including the output directory, the number of training epochs, the learning rate, batch sizes, weight decay, evaluation strategy, and logging settings. We create a `Trainer` object with the model, training arguments, metric computation function, training and validation datasets, and tokenizer.

Training and Evaluation

We are now ready to start the fine-tuning process using the `Trainer` object.

```
trainer.train()
```

Calling the `train` method on the `Trainer` object initiates the fine-tuning process. The model will be trained on the training dataset and evaluated on the validation dataset for the specified number of epochs.

Evaluation Metrics

After training, we can evaluate the performance of our model using various metrics such as accuracy and F1 score.

```
preds_output = trainer.predict(emotions_encoded["validation"])
preds_output.metrics
```

The `predict` method on the `Trainer` the object is used to generate predictions on the validation dataset. We then retrieve the evaluation metrics from the `preds_output` object, which include metrics such as test loss, test accuracy, and test F1 score.

Summary

In this notebook, we have covered the process of fine-tuning the DistilBERT model for emotion classification. We started by exploring the dataset and examining the class distribution and text lengths. We then performed tokenization using the DistilBERT tokenizer and initialized the model for sequence classification. After configuring the training settings and initializing the `Trainer` object, we trained the model and evaluated its performance using various metrics. Fine-tuning models like DistilBERT allows us to leverage pre-trained language representations for specific downstream tasks, enabling effective text classification in various applications.

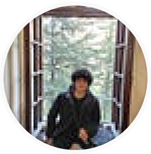
Machine Learning | NLP | Deep Learning | AI

Characteristics of Word Embeddings
11 min read · Sep 4, 2021

The Word2vec Hyperparameters
6 min read · Sep 3, 2021

The Word2ve
15 min rea

View list



Written by Ahmet Taşdemir

Following

84 Followers

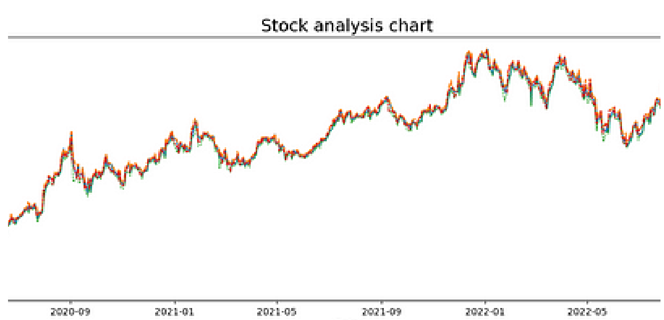
I'm trying to learn and teach. https://www.linkedin.com/in/ahmet-tasdemir/ https://ahmettasdemir.blog/

More from Ahmet Taşdemir

Ahmet Taşdemir

End-to-End Machine Learning Project Guide

The first question to ask your boss is what exactly is the business objective; building a...



Ahmet Taşdemir

STOCK PRICE PREDICTION Using Regression and Deep Learning...

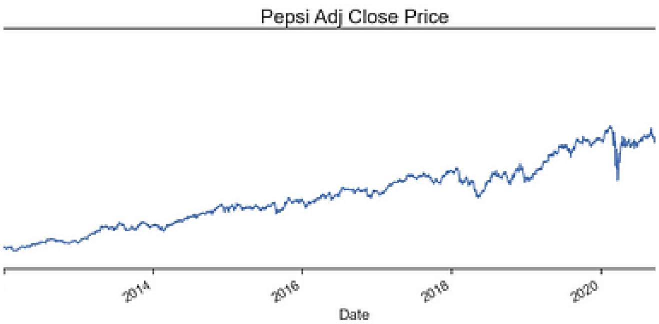
In this tutorial, we will access the stock information of Apple using yfinance API. Thi...

8 min read · Mar 17

142 1

10 min read · May 21

45



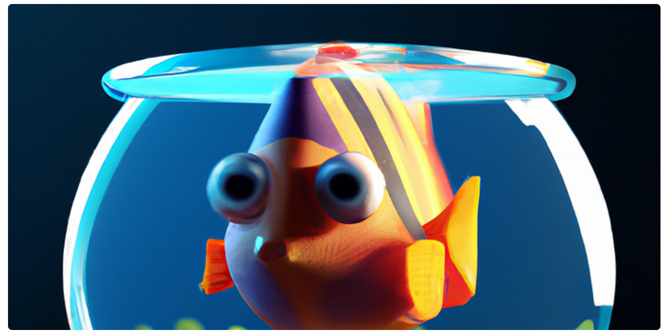
Ahmet Taşdemir

Stock Price Prediction with Technical Analysis(Tutorial)

In this tutorial, we will access the stock information of Pepsi company using yfinanc…

16 min read · May 21

31



Ahmet Taşdemir

Document classification with ELMo(Embeddings from Languag…

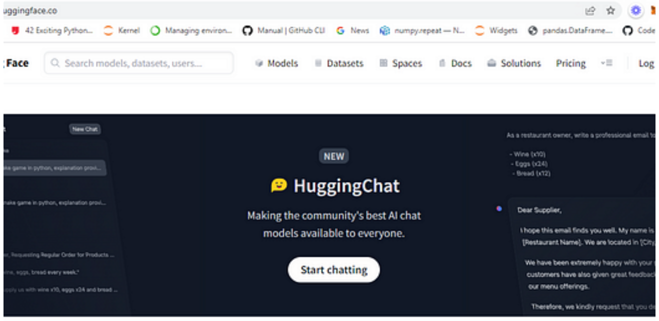
Word embeddings serve as the feature representation for words in various…

7 min read · May 16

5

See all from Ahmet Taşdemir

Recommended from Medium



Bright Eshun

Sentiment Analysis (Part 1): Finetuning DistilBert for Text...

I. Introduction

9 min read · May 8

2

Nimrita Koul

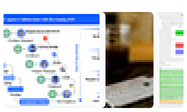
Natural Langauge Processing with Python Part 5: Text Classification

This article is the fifth in the series of my articles covering the sessions I delivered for...

8 min read · May 4

1

Lists

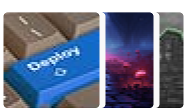


Natural Language Processing

669 stories · 283 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves

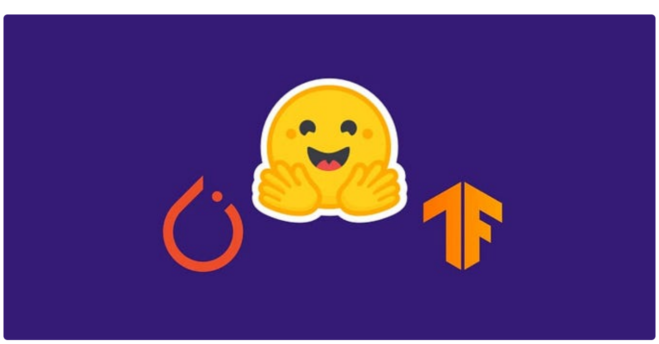


Predictive Modeling w/ Python

20 stories · 452 saves

Practical Guides to Machine Learning

10 stories · 519 saves



Alidu Abubakari in AI Science

Taking Sentiment Analysis to the Next Level with Huggingface's...

Introduction

17 min read · May 31

104 1

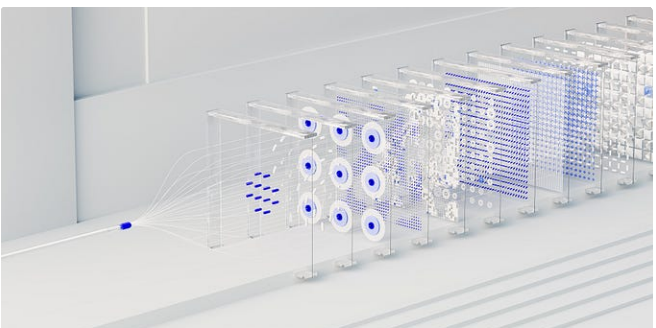
Lukas Niederhäuser

Exploratory Text Analysis of Swiss and German companies from...

The objective of this article is to gather and analyse text from Wikipedia pages that...

8 min read · Jul 5

39

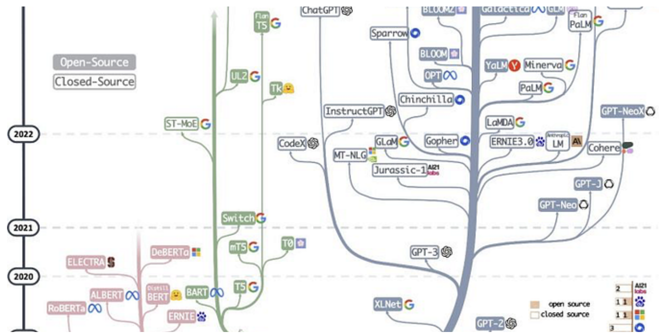


Lefteris Charteros

Building a Sentiment Analysis Classifier using PyTorch Lightning

The purpose of this tutorial is to build a complete machine learning system that will...

17 min read · Jul 22



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

See more recommendations