Vorbereitung

Am Beginn werden die Daten eingelesen und die `pandas` Library geladen. weiters wird das `transformers` Modul installiert.

```
import pandas as pd
```

```
df = pd.read_excel("TCM.xlsx")
```

```
/usr/local/lib/python3.9/dist-packages/openpyxl/styles/stylesheet.py:226: UserWarning: Workbook contains no default style, apply openpyxl's default
  warn("Workbook contains no default style, apply openpyxl's default")
```

```
pip install transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting transformers
  Downloading transformers-4.27.4-py3-none-any.whl (6.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 87.9 MB/s eta 0:00:00
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (from transformers) (6.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (1.22.4)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from transformers) (3.10.7)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.9/dist-packages (from transformers) (4.65.0)
Collecting huggingface-hub<1.0,>=0.11.0
  Downloading huggingface_hub-0.13.3-py3-none-any.whl (199 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.8/199.8 KB 26.3 MB/s eta 0:00:00
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading tokenizers-0.13.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.6/7.6 MB 42.2 MB/s eta 0:00:00
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (2022.10.31)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from transformers) (2.27.1)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from transformers) (23.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/dist-packages (from huggingface-hub<1.0,>=0.11.0->transformers) (4.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (1.26.15)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (3.4)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2.0.12)
Installing collected packages: tokenizers, huggingface-hub, transformers
Successfully installed huggingface-hub-0.13.3 tokenizers-0.13.2 transformers-4.27.4
```

Die relevanten Variablen werden aus dem gesamten Dataframe extrahiert. Zur Übersicht werden die ersten Einträge angezeigt.

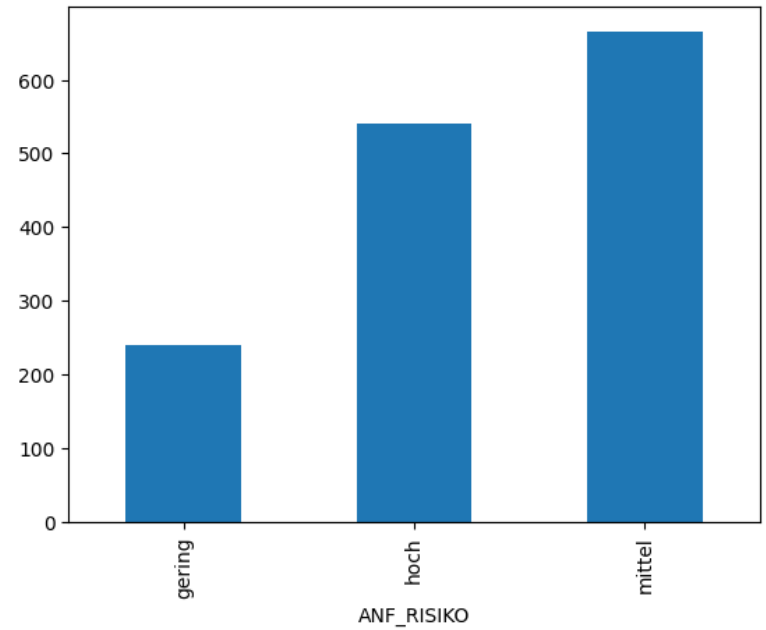
```
df = df[["ANF_BESCHREIBUNG", "ANF_RISIKO"]]
df.head()
```

	ANF_BESCHREIBUNG	ANF_RISIKO
0	In der Formularansicht können über den Befehl ...	mittel
1	Testfälle können innerhalb des Systemordners "...	hoch
4	Der Ablauf für die Erstellung einer Kopie eine...	hoch

Um die Balance der Daten zu überprüfen, werden die absoluten Häufigkeiten jeder Ausprägung in folgenden Diagramm dargestellt. Keine der Klassen ist unterrepräsentiert, daher ist keine weitere Anpassung des Datensets notwendig.

```
df.groupby(['ANF_RISIKO']).size().plot.bar()
```

```
<Axes: xlabel='ANF_RISIKO'>
```



Tokenization

Im `transformers` Modul gibt es einen eigenen Tokenizer für die Bert-Klassifikation. Damit werden aus einem Eingabetext 3 Tensoren erstellt. Anhand eines Beispielsatzes werden die Tensoren ausgegeben.

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-german-cased')

example_text = 'Hier steht ein kurzer Text zur Erklärung'
bert_input = tokenizer(example_text,padding='max_length', max_length = 10,
```

```
                       truncation=True, return_tensors="pt")


print(bert_input['input_ids'])
print(bert_input['token_type_ids'])
print(bert_input['attention_mask'])
```

```
Downloading                                255k/255k [00:00<00:00,
(…)solve/main/vocab.txt: 100%              365kB/s]
Downloading                                29.0/29.0 [00:00<00:00,
(…)okenizer_config.json: 100%              1.34kB/s]
Downloading                                433/433 [00:00<00:00,
```

Zum Verständnis wird der erste Tensor wieder zurück transformiert und als Folge einzelner Token ausgegeben:

```
example_text = tokenizer.decode(bert_input.input_ids[0])

print(example_text)
```

```
[CLS] Hier steht ein kurzer Text zur Erklärung [SEP] [PAD]
```

```
import torch
import numpy as np
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-german-cased')
labels = {'gering':3,
          'mittel':2,
          'hoch':1
          }

class Dataset(torch.utils.data.Dataset):

    def __init__(self, df):

        self.labels = [labels[label] for label in df['ANF_RISIKO']]
        self.texts = [tokenizer(text,
                                padding='max_length', max_length = 512, truncation=True,
                                 return_tensors="pt") for text in df['ANF_BESCHREIBUNG']]

    def classes(self):
        return self.labels

    def __len__(self):
        return len(self.labels)

    def get_batch_labels(self, idx):
        # Fetch a batch of labels
        return np.array(self.labels[idx])
```

```
    def get_batch_texts(self, idx):
        # Fetch a batch of inputs
        return self.texts[idx]

    def __getitem__(self, idx):

        batch_texts = self.get_batch_texts(idx)
        batch_y = self.get_batch_labels(idx)

        return batch_texts, batch_y
```

Train- / Testsplit

Der vorliegende Dataframe wird in Trainings-, Test- und Validierungssets unterteilt im Verhältnis 80:10:10. Anschließend werden die neuen Sets wieder anhand der Verteilung der Zielvariablen dargestellt.

```
np.random.seed(1234)
df_train, df_val, df_test = np.split(df.sample(frac=1, random_state=42),
                                     [int(.8*len(df)), int(.9*len(df))])

print(len(df_train), len(df_val), len(df_test))
```

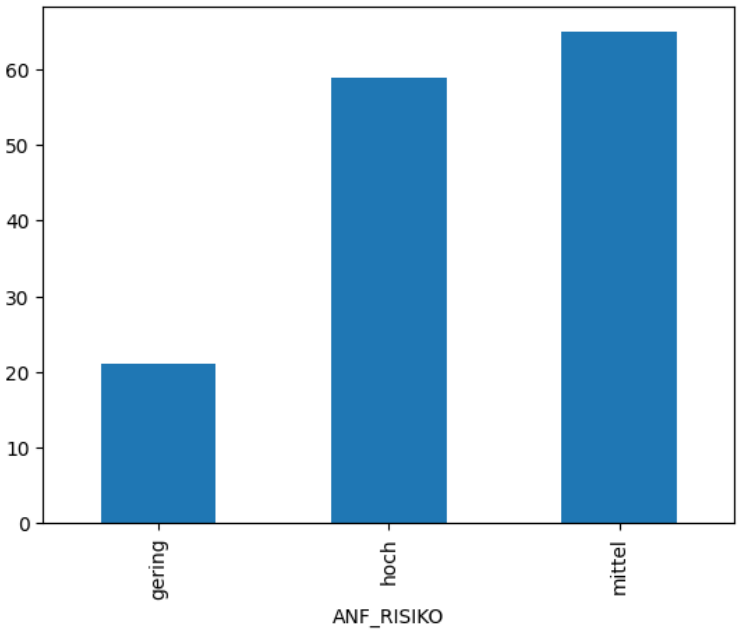
```
1156 145 145
```

```
df_train.groupby(['ANF_RISIKO']).size().plot.bar()
```

```
<Axes: xlabel='ANF_RISIKO'>
```


```
df_test.groupby(['ANF_RISIKO']).size().plot.bar()
```

```
<Axes: xlabel='ANF_RISIKO'>
```




```
df_val.groupby(['ANF_RISIKO']).size().plot.bar()
```

```
<Axes: xlabel='ANF_RISIKO'>
```

Zur Veranschaulichung wird ein Asuzug der Trainingsdaten angezeigt. Es ist zu beachten, dass alle Sonder- und Satzzeichen weiterhin im Text enthalten sind.

```
df_train
```

	ANF_BESCHREIBUNG	ANF_RISIKO
413	Für das Verschieben von Testfallzuordnungen in...	gering
316	Mit dem Typ „Resultat" können Filterkriterien ...	mittel
554	In der Toolbar von Formularansichten gibt es i...	gering
65	Ein in der Resultatshistorie ausgewähltes TF-R...	mittel
1380	Für Anforderungen gibt es unterschiedliche Sym...	mittel
...	...	...
517	Bei Testfällen ohne Resultat wird immer das (n...	gering
1069	Für die Zuordnung eines PTARs zu einem Testfal...	hoch
476	Für das Löschen von Versionen gibt es die folg...	mittel
157	Das Layout einer Komponente kann im Register „...	mittel
16	Das Layout einer Komponente kann im Register „...	mittel

1156 rows × 2 columns

Erstellung des Netzes

Die Klasse `BertClassifier` legt den Aufbau des Netzes fest. An erster Stelle steht das BertModel mit den erzeugten Tensoren. Danach werden in einem Dropout Layer einige Werte vernachlässigt. Es folgen die 768 hidden Layer. Am Ende wird eine Rectified Linear Unit als Aktivierungsfunktion eingesetzt.

```
from torch import nn
from transformers import BertModel

class BertClassifier(nn.Module):

    def __init__(self, dropout=0.5):
```

```
        super(BertClassifier, self).__init__()

        self.bert = BertModel.from_pretrained('bert-base-german-cased')
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(768, 5)
        self.relu = nn.ReLU()

    def forward(self, input_id, mask):

        _, pooled_output = self.bert(input_ids= input_id, attention_mask=mask,return_dict=False)
        dropout_output = self.dropout(pooled_output)
        linear_output = self.linear(dropout_output)
        final_layer = self.relu(linear_output)

        return final_layer
```

Training

Im folgenden Abschnitt wird die Funktion für das Training des Modells implementiert. Es werden einige Colab-spezifische Parameter gesetzt.
Als Kriterium wird `CrossEntropyLoss` verwendet, als Optimierung ist `Adam` eingesetzt. Als Rückgabe des Trainings werden `train_loss`, `train_acc`, `val_loss` und `val_acc` ausgegeben.

```
from torch.optim import Adam
from tqdm import tqdm

def train(model, train_data, val_data, learning_rate, epochs):

    train, val = Dataset(train_data), Dataset(val_data)

    train_dataloader = torch.utils.data.DataLoader(train, batch_size=2, shuffle=True)
    val_dataloader = torch.utils.data.DataLoader(val, batch_size=2)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    criterion = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr= learning_rate)

    if use_cuda:

            model = model.cuda()
            criterion = criterion.cuda()

    train_loss = []
    train_acc = []
    val_loss = []
    val_acc = []
```

```
    for epoch_num in range(epochs):

            total_acc_train = 0
            total_loss_train = 0

            for train_input, train_label in tqdm(train_dataloader):

                train_label = train_label.to(device)
                mask = train_input['attention_mask'].to(device)
                input_id = train_input['input_ids'].squeeze(1).to(device)

                output = model(input_id, mask)

                batch_loss = criterion(output, train_label.long())
                total_loss_train += batch_loss.item()

                acc = (output.argmax(dim=1) == train_label).sum().item()
                total_acc_train += acc

                model.zero_grad()
                batch_loss.backward()
                optimizer.step()

            total_acc_val = 0
            total_loss_val = 0

            with torch.no_grad():

                for val_input, val_label in val_dataloader:

                    val_label = val_label.to(device)
                    mask = val_input['attention_mask'].to(device)
                    input_id = val_input['input_ids'].squeeze(1).to(device)

                    output = model(input_id, mask)

                    batch_loss = criterion(output, val_label.long())
                    total_loss_val += batch_loss.item()

                    acc = (output.argmax(dim=1) == val_label).sum().item()
                    total_acc_val += acc


            train_loss = np.append(train_loss, (total_loss_train / len(train_data)))
            train_acc = np.append(train_acc, (total_acc_train / len(train_data)))
            val_loss = np.append(val_loss, (total_loss_val / len(val_data)))
            val_acc = np.append(val_acc, (total_acc_val / len(val_data)))

    return train_loss, train_acc, val_loss, val_acc
```

Mit Angabe der Zahl an Trainingsepochen und gewünschter Lernrate wird das Training ausgeführt.

```
EPOCHS = 2
model = BertClassifier()
LR = 1e-5

loss_tr, acc_tr, loss_val, acc_val = train(model, df_train, df_val, LR, EPOCHS)
```

```
Downloading pytorch_model.bin:  100%  439M/439M [00:01<00:00, 324MB/s]
Some weights of the model checkpoint at bert-base-german-cased were not used when
- This IS expected if you are initializing BertModel from the checkpoint of a mode
- This IS NOT expected if you are initializing BertModel from the checkpoint of a
100%|██████████| 578/578 [02:03<00:00,  4.67it/s]
```

Evaluierung

Nach dem Training werden für jede Epoche die Loss- und Accuracy-Werte angegeben.

```
print("loss_tr: ", loss_tr)
print("acc_tr: ", acc_tr)
print("loss_val: ", loss_val)
print("acc_val: ", acc_val)
```

```
loss_tr:  [0.29577902 0.05370395]
acc_tr:  [0.79152249 0.97577855]
loss_val:  [0.15991058 0.09121912]
acc_val:  [0.92413793 0.96551724]
```

Für eine detailliertere Evalierung wird im Folgenden die `evaluate` Funktion definiert. Darin wird anhand der Testdaten überprüft, wieviele Datensätze korrekt und wieviele zu hoch oder zu niedrig klassifiziert werden.

```
def evaluate(model, test_data):

    test = Dataset(test_data)

    test_dataloader = torch.utils.data.DataLoader(test, batch_size=1)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:

        model = model.cuda()

    total_acc_test = 0
    zuhochkl = 0
    zuniedrigkl = 0
    richtigkl = 0
```

```
    with torch.no_grad():

        for test_input, test_label in test_dataloader:

              test_label = test_label.to(device)
              mask = test_input['attention_mask'].to(device)
              input_id = test_input['input_ids'].squeeze(1).to(device)

              output = model(input_id, mask)

              pred = output.argmax(dim=1)[0].item()
              trcl = test_label[0].item()

              if (pred < trcl):
                  zuhochkl = zuhochkl + 1
              if (pred > trcl):
                  zuniedrigkl = zuniedrigkl + 1
              if (pred == trcl):
                  richtigkl = richtigkl + 1

              acc = (output.argmax(dim=1) == test_label).sum().item()
              total_acc_test += acc

    print(f'Test Accuracy: {total_acc_test / len(test_data): .3f}')

    checksum = zuhochkl + zuniedrigkl + richtigkl
    print("zu hoch klassifiziert: ", zuhochkl)
    print("zu niedrig klassifiziert: ", zuniedrigkl)
    print("richtig klassifiziert: ", richtigkl)
```

```
print(df_test.shape)
evaluate(model, df_test)
```

```
(145, 2)
Test Accuracy:  0.952
zu hoch klassifiziert:  1
zu niedrig klassifiziert:  6
richtig klassifiziert:  138
```

Zur grafischen Veranschaulichung werden in den folgenden Abbildungen die Trends von Loss und Accuracy in den Trainings- und Validierungsdaten gezeigt.

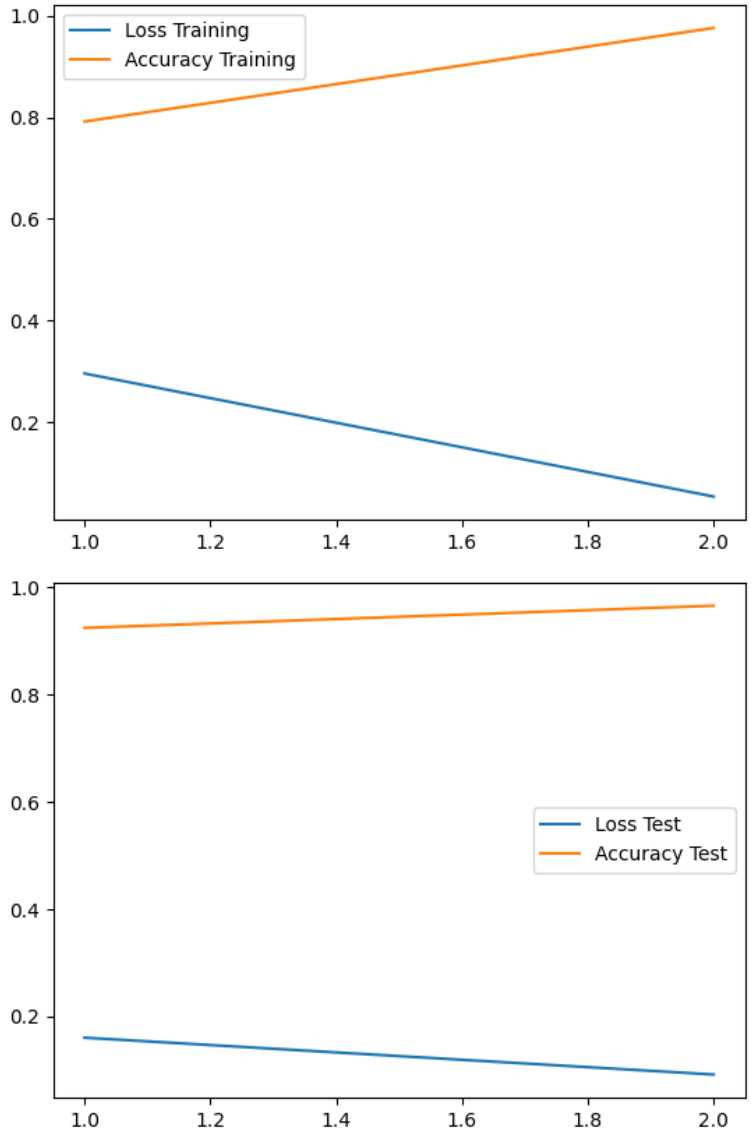
```
p1 = pd.DataFrame({
    'Loss Training': loss_tr,
    'Accuracy Training': acc_tr
    }, index=[1,2])

p2 = pd.DataFrame({
    'Loss Test': loss_val,
    'Accuracy Test': acc_val
    }, index=[1,2])

p1.plot.line()
```

```
p1.plot.line()
p2.plot.line()
```

```
<Axes: >
```



Beispiel

Um eine einzelne Vorhersage ausgeben zu lassen, wird eine `get_pred` Funktion implementiert. Damit kann zu einem Beispieltext der vorhergesagte Wert ausgegeben werden.

```
def get_pred(model, test_data):

    test = Dataset(test_data)

    test_dataloader = torch.utils.data.DataLoader(test, batch_size=1)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:

        model = model.cuda()

    with torch.no_grad():

        pred = []

        for test_input, test_label in test_dataloader:

              test_label = test_label.to(device)
              mask = test_input['attention_mask'].to(device)
              input_id = test_input['input_ids'].squeeze(1).to(device)

              output = model(input_id, mask)

              if output.argmax(dim=1)[0].item() == 3:
                pred = np.append(pred, 'gering')
              if output.argmax(dim=1)[0].item() == 2:
                pred = np.append(pred, 'mittel')
              if output.argmax(dim=1)[0].item() == 1:
                pred = np.append(pred, 'hoch')

    test_data['Vorhersage'] = pred
    print(test_data)
```

```
var = pd.DataFrame({'ANF_BESCHREIBUNG': [
    "ein text mit informationsdialog ist vielleicht richtig",
    "Die Sonne lacht vom Himmel doch die Software stürzt ab"
    ],
    'ANF_RISIKO': ["hoch", "mittel"]})
var.head()
```

	ANF_BESCHREIBUNG	ANF_RISIKO
0	ein text mit informationsdialog ist vielleicht...	hoch
1	Die Sonne lacht vom Himmel doch die Software s...	mittel

```
get_pred(model, var)
```

```
                                  ANF_BESCHREIBUNG ANF_RISIKO Vorhersage
0  ein text mit informationsdialog ist vielleicht...       hoch     gering
1  Die Sonne lacht vom Himmel doch die Software s...     mittel     mittel
```

```
torch.save(model.state_dict(), 'mdl_ds3.pt')
```

✓ 0 s Abgeschlossen um 21:43