Sentiment Analizáló

Gosztolya Máté FZRNZW

Tartalomjegyzék

Bevezetés

A projekt célja a szövegek gépi tanuláson alapuló osztályozása, amely kiemelt fontossággal bír sok különböző területen, például a termék- vagy filmkritikák értékelésében, érzelmek felismerésében a közösségi médiában vagy akár hírek kategorizálásában. Az automatizált osztályozás lehetővé teszi a nagy mennyiségű szöveges adatok hatékony elemzését és kezelését, ami számos felhasználási lehetőséget kínál az üzleti, társadalmi és tudományos területeken.

A projektben a Reddit és Twitter platformokról származó szöveges adatokat használjuk fel, amelyeket gépi tanulási modellek segítségével kategorizálunk. Az alkalmazott módszerek magukban foglalják az adatok előkészítését, szövegvektorok létrehozását, modell kialakítását és tanítását, valamint a modell teljesítményének értékelését. Ennek a dokumentációnak a célja az, hogy részletesen bemutassa ezeket a lépéseket, és átfogó betekintést nyújtson abba, hogyan lehet gépi tanulási modellek segítségével hatékonyan osztályozni szöveges adatokat.

Modell előállítása

Adatok Betöltése és Előkészítése

Az adatok betöltése és előkészítése során a program két különböző forrásból származó adatokat olvas be CSV fájlokból: egy Reddit adatfájlból és egy Twitter adatfájlból. Az adatok betöltése pandas DataFrame-ekbe történik, ami lehetővé teszi az adatok könnyű kezelését és manipulációját.

A dataset forrása: https://www.kaggle.com/datasets/cosmos98/twitter-and-reddit-sentimentalanalysis-dataset?resource=download

A Reddit adatfájlban a szöveges kommentek és azokhoz tartozó kategóriák találhatók, míg a Twitter adatfájlban hasonlóképpen a tiszta szöveges adatok és az azokhoz rendelt kategóriák szerepelnek. Az adatfájlok struktúrája lehet például a következő

This is a positive comment.,1

This is a neutral comment.,0

This is a negative comment., -1

Első lépésben betöltjük a CSV fájlokat a pandas read_csv függvényével, majd kiválasztjuk a szöveges adatokat és azokhoz tartozó címkéket. Ha valamelyik adatfájlban hiányzó értékek vannak, ezeket üres stringekkel helyettesítjük a „fillna” metódus segítségével. Ezáltal az adatok betöltése és előkészítése sikeresen megtörténik a további feldolgozáshoz.

```
# Lekérdezzük a könyvtár abszolút elérési helyét, a fájlok bekéréséhez
script_dir = os.path.dirname(os.path.abspath(__file__))

# Betöltjük a „Reddit_Data.csv” fájlból származó adatokat
reddit_data_file_path = os.path.join(script_dir, "Reddit_Data.csv")
redditData = pd.read_csv(reddit_data_file_path)
```

```
# Betöltjük a „Twitter_Data.csv” fájlból származó adatokat
twitter_data_file_path = os.path.join(script_dir, "Twitter_Data.csv")
data2 = pd.read_csv(twitter_data_file_path)

# Külön válasszuk az adatokat a címkéktől
X = redditData['clean_comment']
y = redditData['category']
X2 = data2['clean_text']
y2 = data2['category']

# Ellenőrizzük van e NaN (üres) érték, és felülírjuk az értékét
X.fillna('', inplace=True)
X2.fillna('', inplace=True)
```

Adatok felosztása és a címkék átmappázása

Az `X` és `y` változókat felosztjuk tanító- és tesztadatokra a `train_test_split` függvény segítségével. A `test_size` paraméter meghatározza a tesztadatok arányát az összes adathoz képest (ebben az esetben 10%), míg a `random_state` paraméter biztosítja a véletlenszerű megosztást, hogy a megosztás minden futtatáskor ugyanaz legyen.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
```

Az adatok felosztása után hozzáadjuk a Twitter adatokat is a tanítóadatokhoz a `pd.concat` függvény segítségével. Ez növeli a tanítóadatok méretét és diverzitását.

```
X_train = pd.concat([X_train, X2], ignore_index=True)
y_train = pd.concat([y_train, y2], ignore_index=True)
```

A címkéket átmappázzuk nem-negatív egész számokra a `label_map` szótárral. Ez lehetővé teszi, hogy a címkék könnyen kezelhetőek legyenek a gépi tanulási modell számára.

```
label_map = {-1: 0, 0: 1, 1: 2}
y_train_mapped = y_train.map(label_map)
y_test_mapped = y_test.map(label_map)
```

Adatok szövegszerűsítése és vektorizálása

Az Adatok Szövegszerűsítése és Vektorizálása rész a szöveges adatokat tokenizálja és vektorizálja, hogy azokat könnyen kezelhető formába alakítsa a gépi tanulási modell számára. A tokenizálás folyamata során a szöveges adatokat tokenekre (pl. szavakra vagy karakterekre) bontja, míg a vektorizálás létrehoz egy numerikus reprezentációt ezekből a tokenekből.

Ebben a programban a CountVectorizer osztályt használjuk, ami a szövegeket bag-of-words model alapján vektorizálja, vagyis minden szóhoz egy számlálót rendel. Ez a módszer szótárat hoz létre az összes szövegből, majd minden dokumentumot egy olyan vektorral reprezentál, amelynek elemei a szótárhoz tartozó számlálók.

Létrehozunk egy CountVectorizer objektumot, amelynek a max_features paramétere korlátozza a szótárat a leggyakoribb 25000 szóra.

Ezt követően alkalmazzuk a fit_transform metódust az X_train adatokon, hogy a CountVectorizer létrehozza a szótárat és vektorizálja a tanítóadatokat. A transform metódust pedig az X_test adatokon alkalmazzuk, hogy ugyanazt a szótárat használva vektorizáljuk a tesztadatokat is. A vektorizált adatokat tenzorok formájában kapjuk meg, amelyeket továbbíthatunk a gépi tanulási modellnek a továbbiakban.

```
# Alkalmazza a CountVectorizert a szövegek tokenizálására és megjelenítésére
vectorizer = CountVectorizer(max_features=25000)
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)
```

PyTorch adatkészlet létrehozása

Ezután a PyTorch adatkészlet létrehozását valósítjuk meg, amely lehetővé teszi a tanító- és tesztadatok kezelését és betöltését a PyTorch keretrendszerbe. Az adatkészletet úgy tervezték, hogy kompatibilis legyen a PyTorch Dataset API-jával, ami egyszerűsíti az adatok kezelését és betöltését a modell tanítása és értékelése során.

A SentimentDataset osztályt definiáljuk, ami örököl az Dataset osztályból, amelyet a PyTorch nyújt.

```
class SentimentDataset(Dataset):
```

Az inicializáló metódusban a tanítóadatokat (X) és a hozzájuk tartozó címkéket (y) fogadjuk el. Az X adatokat tenzor formátumban tároljuk el a torch.tensor segítségével, konvertálva azokat Float32 adattípusba. A címkéket (y) is tenzor formátumban tároljuk, de Long adattípusba konvertálva. A length változóban eltároljuk az adatkészlet méretét.

```
def __init__(self, X, y):
    self.X = torch.tensor(X.toarray(), dtype=torch.float32)
    self.y = torch.tensor(y.values, dtype=torch.long)
    self.length = len(self.X)
```

A len metódus visszaadja az adatkészlet méretét, ami a length változóból származik.

```
def __len__(self):
    return self.length
```

A getitem metódus lehetővé teszi az adatkészlet egy elemének lekérését adott index alapján. Ha az index a megadott hosszúságon kívül esik, egy IndexError kivételt dob. Ellenkező esetben az X és y tenzorokból az adott indexű elemet kinyerjük és visszaadjuk.

```
def __getitem__(self, idx):
    if idx >= self.length:
        raise IndexError(f"Index {idx} is out of bounds for dimension 0 with size {self.length}")
    return self.X[idx], self.y[idx]
```

Hiányzó címkékkel rendelkező adatok szűrése

Ez a kódrészlet egy függvényt definiál, amely segít szűrni az adatokat olyan esetekben, amikor hiányzó címkékkel rendelkező adatokkal találkozunk. A hiányzó címkékkel rendelkező adatok problémát okozhatnak a gépi tanulási modellek számára, mivel nem tudják megfelelően értelmezni ezeket az adatokat a tanítás során. Ezért fontos ezeket az adatokat kiszűrni a tanító- és tesztadatokból.

```
def filter_data(X, y):
    filtered_indices = [i for i, label in enumerate(y) if not pd.isnull(label)]
    return X[filtered_indices], y.iloc[filtered_indices]
```

A filter data függvény elfogadja az X és y adatokat, majd kiszűri azokat a pozíciókat, ahol a y címkék hiányoznak (NaN értéket vesznek fel). Ehhez felhasználja a Python listakomprehenziós módszerét és az enumerate függvényt, hogy meghatározza azokat a pozíciókat, ahol a címkék hiányoznak. Ezután visszaadja az X és y adatokat csak azokkal a pozíciókkal, ahol a címkék értéke nem hiányzik.

```
X_train_vec_filtered, y_train_mapped_filtered = filter_data(X_train_vec, y_train_mapped)
X_test_vec_filtered, y_test_mapped_filtered = filter_data(X_test_vec, y_test_mapped)
```

Adatok betöltése DataLoader segítségével

A PyTorch DataLoader osztályát használjuk az adatkészletek betöltésére és batch-ek formájában való feldolgozására. A kódrészletben két fő lépést hajtunk végre:

Az adatkészletek létrehozásához a SentimentDataset osztályt használjuk az X_train_vec_filtered és y_train_mapped_filtered (tanítóadatokhoz) és az X_test_vec_filtered és y_test_mapped_filtered (tesztadatokhoz) adatok beolvasására és előkészítésére.

```
train_dataset = SentimentDataset(X_train_vec_filtered, y_train_mapped_filtered)
test_dataset = SentimentDataset(X_test_vec_filtered, y_test_mapped_filtered)
```

A DataLoader-k létrehozására a létrehozott adatkészleteket DataLoader objektumokká alakítjuk át, amelyek batch-ek formájában képesek az adatokat betölteni és feldolgozni. A batch_size paraméter meghatározza, hogy egy adott batch-ben hány adatpont legyen. A shuffle paraméterrel beállítjuk, hogy a DataLoader véletlenszerűen keverje-e meg a betöltött adatok sorrendjét.

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Modell létrehozása és konfigurálása

Itt a modell létrehozását és konfigurálását valósítjuk meg a gépi tanulási feladathoz. A modell létrehozása során egy osztályt definiálunk, amely örököl az nn.Module osztályból, ami a PyTorch által nyújtott modell interfész.

```
class SentimentClassifier(nn.Module):
```

A modell definiálása az __init__ metódusban történik, ahol a modell architektúráját határozzuk meg. Ebben az esetben a modell egyetlen teljesen összekapcsolt rétegből áll (nn.Linear), amelynek bemeneti mérete az input_size, kimeneti mérete pedig az output_size. Ez az architektúra megfelel egy egyszerű neurális hálónak, amelyet használhatunk szentiment analízisre.

```
def __init__(self, input_size, output_size):
    super(SentimentClassifier, self).__init__()
    self.fc = nn.Linear(input_size, output_size)
```

A modell forward metódusa határozza meg a bemeneti adatok áthaladását a modellen. Ebben az esetben a bemenet egyszerűen áthalad a teljesen összekapcsolt rétegen (fc), majd a kimenetet visszaadjuk.

```
  def forward(self, x):
    x = self.fc(x)
    return x
```

A modell konfigurációját (bemeneti és kimeneti méret, valamint a használt szótár mérete) egy JSON fájlba mentjük. Ez lehetővé teszi a modell újrahasználását és betöltését későbbi műveletek során.

```
input_size = X_train_vec.shape[1]
output_size = len(y.unique())
model = SentimentClassifier(input_size, output_size)

data = {
  "input_size": len(train_dataset),
  "output_size": output_size,
  "max_features": vectorizer.max_features
}

json_file_path = os.path.join(script_dir, "model_config.json")

with open(json_file_path, "w") as json_file:
  json.dump(data, json_file)
```

Modell tanítása és optimalizálása

Ez a kódrészlet a gépi tanulási modell tanításáért és optimalizálásáért felelős. A modellt egy adott adatkészleten (tanítóadatokon) tanítjuk a három alapvető lépés végrehajtásával: előrejelzés, veszteség kiszámítása és visszaterjesztés.

Veszteségfüggvény és Optimalizáló algoritmus kiválasztása: A nn.CrossEntropyLoss() segítségével inicializáljuk a veszteségfüggvényt, ami a keresztentrópia veszteségfüggvényt jelenti. Az optim.Adam() függvény segítségével inicializáljuk az optimalizáló algoritmust, ami az Adam optimalizáló algoritmust jelenti.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

A train() függvény felelős a modell tanításáért az adatkészleten. A modell train() metódusával az előrejelzési folyamatot aktiváljuk, majd az adott epochokon keresztül iterálva a tanítóadatokon keresztül futtatjuk az optimalizáló algoritmust. A veszteség értékét minden epoch után összegezzük, hogy megfigyeljük a tanítás során bekövetkező változásokat.

```
def train(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}")
```

Meg hívjuk a korábban már definiált train() függvényt a modell, a tanító adatkészlet, a veszteségfüggvény és az optimalizáló algoritmus paramétereivel. Ez elindítja a modell tanítási folyamatát.

```
train(model, train_loader, criterion, optimizer)
```

Ezután az optimalizáló algoritmus és a veszteségfüggvény állapotát mentjük egy adott elérési úton elhelyezett fájlba. A torch.save() függvény segítségével egy Python dictionary-t mentünk, amely tartalmazza a modell paramétereit (state_dict), az optimalizáló algoritmus állapotát (state_dict), valamint a veszteségfüggvény állapotát (state_dict). Ez az eljárás lehetővé teszi, hogy később visszatöltsük és folytassuk a tanítást vagy az értékelést, anélkül hogy újra kellene tanítani a modellt. A PATH változóban tárolt elérési út határozza meg, hogy hova kerül mentésre a modell és az állapotok.

```
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'criterion_state_dict': criterion.state_dict()
}, PATH)
```

Modell értékelése

Itt a tanított modell teljesítményét értékeljük a tesztadatokon. A modellt először "eval" módba állítjuk a model.eval() hívással, ami jelzi a PyTorch-nak, hogy értékelési módban fog működni, és nem fogja változtatni a súlyokat a modellben.

```
def evaluate(model, test_loader):
    model.eval()
```

Ezután létrehozunk két üres listát, y_true és y_pred, amelyekbe az igazi és a prediktált címkéket fogjuk tárolni.

```
    y_true = []
```

```
y_pred = []
```

A torch.no_grad() blokk hatására a PyTorch nem fogja követni a gradienseket az értékelés során, ami gyorsabb és kevesebb memóriát igénylő értékelést tesz lehetővé.

```
with torch.no_grad():
```

Ezután iterálunk a test_loader DataLoader-en keresztül, és minden egyes batch adaton előrejelzéseket készítünk a modell segítségével (outputs = model(inputs)). Az torch.max() függvénnyel meghatározzuk az előrejelzett címkét a kimeneti valószínűségek alapján, majd ezeket hozzáadjuk a y_pred listához.

```
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        y_true.extend(labels.tolist())
        y_pred.extend(predicted.tolist())
```

Miután végzett az iterációval, az accuracy_score() függvény segítségével kiszámítjuk az osztályozás pontosságát az igazi és a prediktált címkék alapján. Végül kiírjuk az eredményt az aktuális pontosság mellett.

```
    accuracy = accuracy_score(y_true, y_pred)
    print(f"Accuracy: {accuracy}")

#A kiértékelő függvényt meghívjuk a tesztadatokra
evaluate(model, test_loader)
```

Szótár mentése JSON fájlba

Ez a kódrészlet egy adott szótárat ment JSON formátumban egy fájlba. Ez egy hatékony módszert kínál arra, hogy egy adott szótárat JSON formátumban mentünk el, ami szükséges lehet például a későbbi újrahasználhatóság vagy megosztás során.

A vocab_dict változó egy Python szótárt tartalmaz, amelyet a vectorizer objektum vocabulary_ attribútuma alapján hozunk létre. Ez a szótár tartalmazza a CountVectorizer által létrehozott szó-index párokat, ahol a kulcsok a szavak, az értékek pedig azok indexei a vektorban.

```
vocab_dict = {str(key): int(value) for key, value in vectorizer.vocabulary_.items()}
```

Az elérési útvonalat a script_dir változó és a 'vectorizer_vocabulary.json' fájlnév összefűzésével kapjuk meg, amely megadja, hogy hova mentjük el a szótárat JSON formátumban.

```
vocab_file_path = os.path.join(script_dir, 'vectorizer_vocabulary.json')
```

A json.dump() függvény segítségével a vocab_dict szótárat kiírjuk a korábban meghatározott fájlba. Ez a függvény átalakítja a Python szótárat JSON formátummá, majd kiírja azt a fájlba.

```
with open(vocab_file_path, 'w') as f:
    json.dump(vocab_dict, f)
```

Forráskód

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
import numpy as np
import json

script_dir = os.path.dirname(os.path.abspath(__file__))

reddit_data_file_path = os.path.join(script_dir, "Reddit_Data.csv")
redditData = pd.read_csv(reddit_data_file_path)

twitter_data_file_path = os.path.join(script_dir, "Twitter_Data.csv")
data2 = pd.read_csv(twitter_data_file_path)

PATH = os.path.join(script_dir, "model.pth")

X = redditData['clean_comment']
y = redditData['category']
X2 = data2['clean_text']
y2 = data2['category']

X.fillna('', inplace=True)
X2.fillna('', inplace=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

X_train = pd.concat([X_train, X2], ignore_index=True)
y_train = pd.concat([y_train, y2], ignore_index=True)

label_map = {-1: 0, 0: 1, 1: 2}
y_train_mapped = y_train.map(label_map)
y_test_mapped = y_test.map(label_map)

vectorizer = CountVectorizer(max_features=25000)
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)

class SentimentDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.tensor(X.toarray(), dtype=torch.float32)
        self.y = torch.tensor(y.values, dtype=torch.long)
        self.length = len(self.X)

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        if idx >= self.length:
```

```
            raise IndexError(f"Index {idx} is out of bounds for dimension 0 with size {self.length}")
        return self.X[idx], self.y[idx]

def filter_data(X, y):
    filtered_indices = [i for i, label in enumerate(y) if not pd.isnull(label)]
    return X[filtered_indices], y.iloc[filtered_indices]

X_train_vec_filtered, y_train_mapped_filtered = filter_data(X_train_vec, y_train_mapped)
X_test_vec_filtered, y_test_mapped_filtered = filter_data(X_test_vec, y_test_mapped)

train_dataset = SentimentDataset(X_train_vec_filtered, y_train_mapped_filtered)
test_dataset = SentimentDataset(X_test_vec_filtered, y_test_mapped_filtered)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

class SentimentClassifier(nn.Module):
    def __init__(self, input_size, output_size):
        super(SentimentClassifier, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.fc(x)
        return x

input_size = X_train_vec.shape[1]
output_size = len(y.unique())
model = SentimentClassifier(input_size, output_size)

data = {
    "input_size": len(train_dataset),
    "output_size": output_size,
    "max_features": vectorizer.max_features
}

json_file_path = os.path.join(script_dir, "model_config.json")

with open(json_file_path, "w") as json_file:
    json.dump(data, json_file)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

def train(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}")
```

```
train(model, train_loader, criterion, optimizer)

torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'criterion_state_dict': criterion.state_dict()
}, PATH)

def evaluate(model, test_loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            y_true.extend(labels.tolist())
            y_pred.extend(predicted.tolist())
    accuracy = accuracy_score(y_true, y_pred)
    print(f"Accuracy: {accuracy}")

evaluate(model, test_loader)

vocab_dict = {str(key): int(value) for key, value in vectorizer.vocabulary_.items()}
vocab_file_path = os.path.join(script_dir, 'vectorizer_vocabulary.json')
with open(vocab_file_path, 'w') as f:
    json.dump(vocab_dict, f)
```

Modell alkalmazása

Modell konfiguráció és szövegválaszték vektorizálása

Első lépésben a modell konfigurációs beállításainak betöltését és a szövegválaszték vektorizálását végezzük el.

A script_dir változó segítségével megkapjuk a jelenlegi szkript könyvtárának elérési útját. Ez a könyvtár tartalmazza a konfigurációs és szókincs fájlokat.

```
script_dir = os.path.dirname(os.path.abspath(__file__))
```

A config_file_path változóban összeállítjuk a konfigurációs fájl elérési útját a script_dir és a fájl neve alapján. Ezután megnyitjuk a fájlt olvasásra, és betöltjük annak tartalmát a config_data változóba.

```
config_file_path = os.path.join(script_dir, "model_config.json")
with open(config_file_path, "r") as json_file:
    config_data = json.load(json_file)
```

A konfigurációs adatokból kinyerjük a max_features értéket, amely meghatározza a CountVectorizer legnagyobb szókincsméretét.

```
max_features = config_data["max_features"]
```

Ezután inicializáljuk a vectorizer objektumot a CountVectorizer segítségével, beállítva a max_features értékét.

```
vectorizer = CountVectorizer(max_features=max_features)
```

A szókincs fájl elérési útját összeállítjuk a vocabulary_file_path változóban, majd megnyitjuk ezt a fájlt olvasásra. Betöltjük a szókincset a vocabulary változóba, és beállítjuk azt a CountVectorizer vocabulary_ attribútumához.

```
vocabulary_file_path = os.path.join(script_dir, 'vectorizer_vocabulary.json')
with open(vocabulary_file_path, 'r') as f:
    vocabulary = json.load(f)

vectorizer.vocabulary_ = vocabulary
```

A modell osztály és a szöveg előfeldolgozó függvény

A SentimentClassifier osztály egy PyTorch nn.Module leszármazott osztálya, ami deffiniálja a modell által használt neurális hálózatot.

```
class SentimentClassifier(nn.Module):
    def __init__(self, input_size, output_size):
        super(SentimentClassifier, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.fc(x)
        return x
```

A preprocess_text függvény szöveg előfeldolgozására szolgál a gépi tanulási modell számára. Ellenőrzi, hogy a szöveg nem üres-e, majd a transform metódussal átalakítja a szöveget egy vektorrá a megadott vectorizer segítségével. Ezután a vektort torch.tensor objektummá alakítja át és visszaadja, megfelelő adattípus és forma mellett a további feldolgozáshoz. Ha a bemenet üres, a függvény None értéket ad vissza.

```
def preprocess_text(text, vectorizer):
    if text:
        text_vector = vectorizer.transform([text])
        return torch.tensor(text_vector.toarray(), dtype=torch.float32)
    else:
        return None
```

Szöveg érzelmi töltetének előrejelzése

Ez a függvény a kapott szöveg alapján előrejelzi annak érzelmi töltetét a modell segítségével.

```
def predict_sentiment(text, model, vectorizer):
```

A függvény első lépése, hogy megvizsgálja, hogy a kapott szöveg üres-e. Ha nem üres, akkor a modell állapotát "eval" módba állítja, hogy ne végezzen tanítást vagy súlyok frissítését a

predikció során. Ezután a szöveget előfeldolgozza a megadott vektorizáló segítségével, hogy megfelelő formába hozza az inputként való kezeléshez.

```
if text:
    model.eval()
    with torch.no_grad():
      text_tensor = preprocess_text(text, vectorizer)
      if text_tensor is not None:
```

A következő lépésben a függvény az előfeldolgozott szöveget átadja a modellnek, és a modell kimenetét használva meghatározza a szöveg érzelmi töltetét. A predikció eredményét egyetlen számként adja vissza, amely reprezentálja az érzelmi töltetet. Ha a szöveg üres, vagy a modell nem tudott predikciót generálni, a függvény None értéket ad vissza.

```
output = model(text_tensor)
          _, predicted = torch.max(output, 1)
          sentiment = predicted.item()
          return sentiment
  return None
```

Modell betöltése és inicializálása

Ebben a részletben a cél az előre betanított modell betöltése és inicializálása a memóriába. Először létrehozunk egy SentimentClassifier objektumot, amely a modellt fogja reprezentálni.

```
model = SentimentClassifier(max_features, 3)
```

Ezután definiáljuk a modell fájl elérési útvonalát a model_file_path változóban, amely a korábban mentett modell fájlra mutat.

```
model_file_path = os.path.join(script_dir, "model.pth")
```

A torch.load függvény segítségével betöltjük a modell állapotát a megadott fájlból. A betöltött állapotban a modell súlyai és paraméterei vannak tárolva.

```
model_state_dict = torch.load(model_file_path, map_location=torch.device('cpu'))
```

A state_dict változóban csak a modell súlyait tartalmazó részt kiválasztjuk a teljes állapotból, és ezt töltjük be a modellbe a load_state_dict függvény segítségével. A strict=False beállítás lehetővé teszi számunkra, hogy figyelmen kívül hagyjuk az esetleges méreteltéréseket a modell súlyai között, ami hasznos lehet, ha az új modellnek eltérő méretei vannak a betanított modelltől. Így a kód biztosítja, hogy a betanított modell súlyait sikeresen betöltsük a modellbe, amelyet ezután használhatunk a predikciók generálásához.

```
state_dict = model_state_dict['model_state_dict']
model.load_state_dict(state_dict, strict=False)
```

Felhasználói interakció: kommentek értékelése

Ez a kódrészlet egy végtelen ciklust hoz létre, amely lehetővé teszi a felhasználó számára, hogy beírjon egy kommentet, és megkapja annak értékelését a modell segítségével.

```
while True:
```

Amikor a ciklus elindul, a program bekéri a felhasználótól a kommentet egy input prompt segítségével. Ha a felhasználó nem ír semmit, és csak entert nyom, a ciklus megszakad, és a program véget ér.

```
user_input = input("Enter a comment (press Enter to exit): ")
    if user_input.strip() == "":
        break
```

Ha a felhasználó beírt valamit, akkor a program meghívja a predict_sentiment függvényt a beírt kommenttel, a modellünkkel és a vektorizálóval. Ez a függvény az adott komment érzelmét predikálja a modell alapján.

```
    sentiment = predict_sentiment(user_input, model, vectorizer)
```

A predikció eredményét a sentiment változóban tároljuk, majd azt ellenőrizzük, hogy a predikció nem nulla. Ha nem nulla, akkor a program kiírja a prediktált érzelmi töltetet a kommentre: "Negative", "Neutral" vagy "Positive", attól függően, hogy a modell az adott érzelmi kategóriába sorolta a kommentet.

```
    if sentiment is not None:
        if sentiment == 0:
            print("Negative")
        elif sentiment == 1:
            print("Neutral")
        else:
            print("Positive")
```

Forráskód

```
import os
import torch
import torch.nn as nn
from sklearn.feature_extraction.text import CountVectorizer
import json

script_dir = os.path.dirname(os.path.abspath(__file__))

config_file_path = os.path.join(script_dir, "model_config.json")
with open(config_file_path, "r") as json_file:
    config_data = json.load(json_file)

max_features = config_data["max_features"]

vectorizer = CountVectorizer(max_features=max_features)

vocabulary_file_path = os.path.join(script_dir, 'vectorizer_vocabulary.json')
```

```
with open(vocabulary_file_path, 'r') as f:
    vocabulary = json.load(f)

vectorizer.vocabulary_ = vocabulary

class SentimentClassifier(nn.Module):
    def __init__(self, input_size, output_size):
        super(SentimentClassifier, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.fc(x)
        return x

def preprocess_text(text, vectorizer):
    if text:
        text_vector = vectorizer.transform([text])
        return torch.tensor(text_vector.toarray(), dtype=torch.float32)
    else:
        return None

def predict_sentiment(text, model, vectorizer):
    if text:
        model.eval()
        with torch.no_grad():
            text_tensor = preprocess_text(text, vectorizer)
            if text_tensor is not None:
                output = model(text_tensor)
                _, predicted = torch.max(output, 1)
                sentiment = predicted.item()
                return sentiment
    return None

model = SentimentClassifier(max_features, 3)

model_file_path = os.path.join(script_dir, "model.pth")

model_state_dict = torch.load(model_file_path, map_location=torch.device('cpu'))

state_dict = model_state_dict['model_state_dict']

model.load_state_dict(state_dict, strict=False)

while True:
    user_input = input("Enter a comment (press Enter to exit): ")
    if user_input.strip() == "":
        break
    sentiment = predict_sentiment(user_input, model, vectorizer)
    if sentiment is not None:
        if sentiment == 0:
            print("Negative")
        elif sentiment == 1:
            print("Neutral")
        else:
            print("Positive")
```

Projekt összefoglalása és tapasztalatok

A modell sikeresen tanult az adatokon a kijelölt 5 epoch alatt. Az epoch-ok során a veszteség folyamatosan csökkent, ami azt mutatja, hogy a modell képes volt adaptálni a tanító adatokon és javítani a teljesítményén.

Epoch 1/5, Loss: 0.6656809394656933
Epoch 2/5, Loss: 0.3989075378083282
Epoch 3/5, Loss: 0.3055350371978835
Epoch 4/5, Loss: 0.25513382695156755
Epoch 5/5, Loss: 0.2232207206324427
Accuracy: 0.9210738255033557

A modell nagyon magas pontossággal értékelte ki a tesztadatokat, a kapott pontossági érték 0.921. Ez azt jelenti, hogy a modell 92.1%-os pontossággal képes megjósolni a kommentek érzelmi kategóriáját.

Hatékonyan és gyorsan futott, az epoch-ok futási ideje elfogadható volt, és a modell rendkívül gyorsan képes volt kiértékelni a tesztadatokat a tanítás után.

Lehetséges további fejlesztési irányok

Az érzelemanalízis továbbfejlesztési lehetőségei számos irányba mutathatnak. Az egyik lehetséges irány a modellek finomítása és tökéletesítése. Ez magában foglalhatja a nagyobb és változatosabb adathalmazokkal való tanítást, amelyek segíthetnek a modell pontosságának és megbízhatóságának növelésében. Emellett a modell további tanítása olyan specifikusabb és árnyaltabb érzelmi kifejezésekkel is javíthatja annak teljesítményét, amelyek az adatokban lehetnek kevésbé gyakoriak.

Mivel az analizáló jelenleg az angol nyelvű tartalmakra van optimalizálva, a modell további nyelvek támogatása lehetővé tenné az érzelemfelismerés globális alkalmazását. Ezáltal a különböző kultúrák és nyelvek érzelmi reakcióinak elemzése is megvalósítható lenne.

Az interaktív felhasználói felületek fejlesztése is fontos lehetőség. Egy felhasználóbarát és intuitív felület lehetővé tenné a felhasználók számára, hogy könnyedén és hatékonyan használják az érzelemanalízis eszközt. Az ilyen típusú felületek lehetőséget adhatnak az adatok vizualizációjára, az eredmények könnyebb értelmezésére, valamint az interakcióra az elemzési folyamat során.

Github repository

A teljes projekt elérhető, és letölthető beleértve ezen dokumentációt, és a specifikálólapot a következő linken: https://github.com/mategm88/SentimentAnalyser

Ismeretszerzés forrásai

"PyTorch for Deep Learning & Machine Learning – Full Course" - https://www.youtube.com/watch?v=V_xro1bcAuA

"PYTORCH DOCUMENTATION" - https://pytorch.org/docs/stable/index.html

"T5-BASE MODEL FOR SUMMARIZATION, SENTIMENT CLASSIFICATION, AND TRANSLATION" - https://pytorch.org/text/main/tutorials/t5_demo.html

"Text classification with the torchtext library" - https://pytorch.org/tutorials/beginner/text_sentiment_ngrams_tutorial.html

"Tutorial on Sentimental Analysis using Pytorch for Beginners" - https://bhadreshpsavani.medium.com/tutorial-on-sentimental-analysis-using-pytorch-b1431306a2d7

"A Tutorial on Torchtext" - https://anie.me/On-Torchtext/