CBU5201_miniproject_submission

2024 年 12 月 29 日

This project is exported to PDF by intend to be protected from plagiarism.

CBU5201 mini-project submission

What is the problem?

This year's mini-project considers the problem of predicting whether a narrated story is true or not. Specifically, you will build a machine learning model that takes as an input an audio recording of **3-5 minutes** of duration and predicts whether the story being narrated is **true or not**.

Which dataset will I use?

A total of 100 samples consisting of a complete audio recording, a *Language* attribute and a *Story Type* attribute have been made available for you to build your machine learning model. The audio recordings can be downloaded from:

https://github.com/CBU5201Datasets/Deception

A CSV file recording the *Language* attribute and *Story Type* of each audio file can be downloaded from:

https://github.com/CBU5201Datasets/Deception/blob/main/CBU0521DD_stories_attributes.csv

What will I submit?

Your submission will consist of **one single Jupyter notebook** that should include:

- **Text cells**, describing in your own words, rigorously and concisely your approach, each implemented step and the results that you obtain,
- **Code cells**, implementing each step,
- **Output cells**, i.e. the output from each code cell,

Your notebook **should have the structure** outlined below. Please make sure that you **run all the cells** and that the **output cells are saved** before submission.

Please save your notebook as:

- CBU5201_miniproject.ipynb

How will my submission be evaluated?

This submission is worth 16 marks. We will value:

- Conciseness in your writing.
- Correctness in your methodology.
- Correctness in your analysis and conclusions.
- Completeness.
- Originality and efforts to try something new.

(4 marks are given based on your audio submission from stage 1.)

The final performance of your solutions will not influence your grade. We will grade your understanding. If you have an good understanding, you will be using the right methodology, selecting the right approaches, assessing correctly the quality of your solutions, sometimes acknowledging that despite your attempts your solutions are not good enough, and critically reflecting on your work to suggest what you could have done differently.

Note that **the problem that we are intending to solve is very difficult**. Do not despair if you do not get good results, **difficulty is precisely what makes it interesting** and **worth trying**.

Show the world what you can do

Why don't you use **GitHub** to manage your project? GitHub can be used as a presentation card that showcases what you have done and gives evidence of your data science skills, knowledge and experience. **Potential employers are always looking for this kind of evidence**.

———————————— PLEASE USE THE STRUCTURE BELOW THIS LINE ————————————

Decetion Detection based on AI-Generated Text Assumption

1 Author

Student Name: Cave Nightingale
Student ID: <masked>

[ ]:
```
import numpy as np
import pandas as pd
import git
import os
import transformers
import soundfile as sf
from deep_translator import GoogleTranslator
import traceback
import nltk
import re
import collections
import torch
import opencc
import jieba
import librosa
import matplotlib.pyplot as plt
import sklearn
import scipy
import whisper
import random
import torch
from torch import nn
from torch.nn import functional as F
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import math
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier, RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report

<masked>
<masked>
<masked>
os.environ['TRUST_REMOTE_CODE'] = 'True'
```

```
if not os.path.exists('input'):
    git.Repo.clone_from('git@github.com:CBU5201Datasets/Deception.git', 'input')
```

2 Problem formulation

Given a dataset of audio recordings, which contains the narrated stories visiting a place or visiting a person, in either English or Mandarin, we aim to build a machine learning model that takes as an input an audio recording of 3-5 minutes of duration and predicts whether the story being narrated is true or not.

3 Methodology

3.1 Intuition

Humans are ChatGPTs, so detect GPT-generated stories instead of fake stories.

3.2 Differences between true and false stories

3.2.1 Text aspects

- true stories are sampled from real world experiences R.
- fake stories are sampled from storytellers' generated experiences G.

We don't know how storytellers generate their stories, but the intuition is that **humans are extremely well-trained Auto Regressive Generative Models**, so we can **replace the storytellers with a generative model** in our assumption.

So the assumption changes to:

- true stories are sampled from real world experiences R.
- fake stories are sampled from an Auto Regressive Generative Model G.

Since we have only 90 samples, we have to train extremely small models, this idea greatly reduces the complexity of the problem and proves to be feasible.

3.2.2 Audio aspects

It's observed that native speakers tell a true story speak faster than a fake story, but not the case for non-native speakers. But we can use this feature as a reference, at leat not harmful.

Other audio features, such as mfcc, f0, intensity, formants, are tested and found no evidence of difference between true and fake stories.

3.3 Features

3.3.1 The training process

The auto regressive generative model take some sample R as input, and interpolate it to a distribution I_R.

I_R is expected to be as close to R as possible, so we assume that I_R is a good representation of R.

This process does not seem to create flaws that can be exploited.

3.3.2 The temperature scaling process

The temperature parameter is used to control the randomness of the generation process. Basically if we have a temperature t between 0 and 1, then the model do the following to get a new distrubution S_R that it actually samples from:

$$S_R(x) = \frac{I_R(x)^{\frac{1}{t}}}{\sum_y I_R(y)^{\frac{1}{t}}}$$

If you observe the formula, the model have more tendency to have a higher probability for the most probable words, in other worlds, **have low entropy** than samples from I_R and R.

So we have determine one predictor of our model: **entropy**.

3.3.3 The generation process

The generation process is tricky. Assume there are n words in the vocabulary, and the story have a maximum length of m. So the generative model have to etimate a distribution of n^m stories, which is time-wasting and infeasible. Instead, the model can estimate only a small sequence or a single word at a time, and then generate the next word based on the previous words.

It's said that the generated stories are not exactly sampled from S_R, but a distribution G_R that is close to S_R but lack of long-term considerations.

This short-sighted behavior inspires us the idea that fake stories does not have a very good overall structure. To be exact, it's observed that sentiment changes across these structures, while fake stories have a more stable sentiment. Because the loss function focus on local prediction and is lack of mechanism to ensure they generate stories with varying sentiment which has actually high probability in the real world.

So we have determined another predictor of our model: **sentiment**.

3.3.4 The narrating process

When the storyteller narrates a story, they speak faster when they are native speakers and tell a true story. This is because they don't have to think about the content of the story.

When they tell true stories, they are expected to speak many specific nouns, such as the name of dishes, the name of tourist attractions, but if they do not experience the story, they will have trouble to tell these nouns.

All in all, features related to narrator's personal language usage but not much about the content of the story are grouped in to language features. These informations are directly extracted from raw content, translation to English may have the risk of losing information.

So we have determined the last predictor of our model: **language**.

3.4 Methodology Summary

Although the intuition asserts that humans are well-trained generative models, there is no straightforward mathematical explanation of this, but somehow the idea proves to make sense in the following experiments and you will see later.

So finally, the predictors of our model are:

- entropy
  - Entropy per token
  - Entropy per unit time
- sentiment
  - Max sentiment
  - Min sentiment
  - Overall sentiment
- language
  - Audio length over text length (Speaking speed)
  - Unique word count (Word diversity)

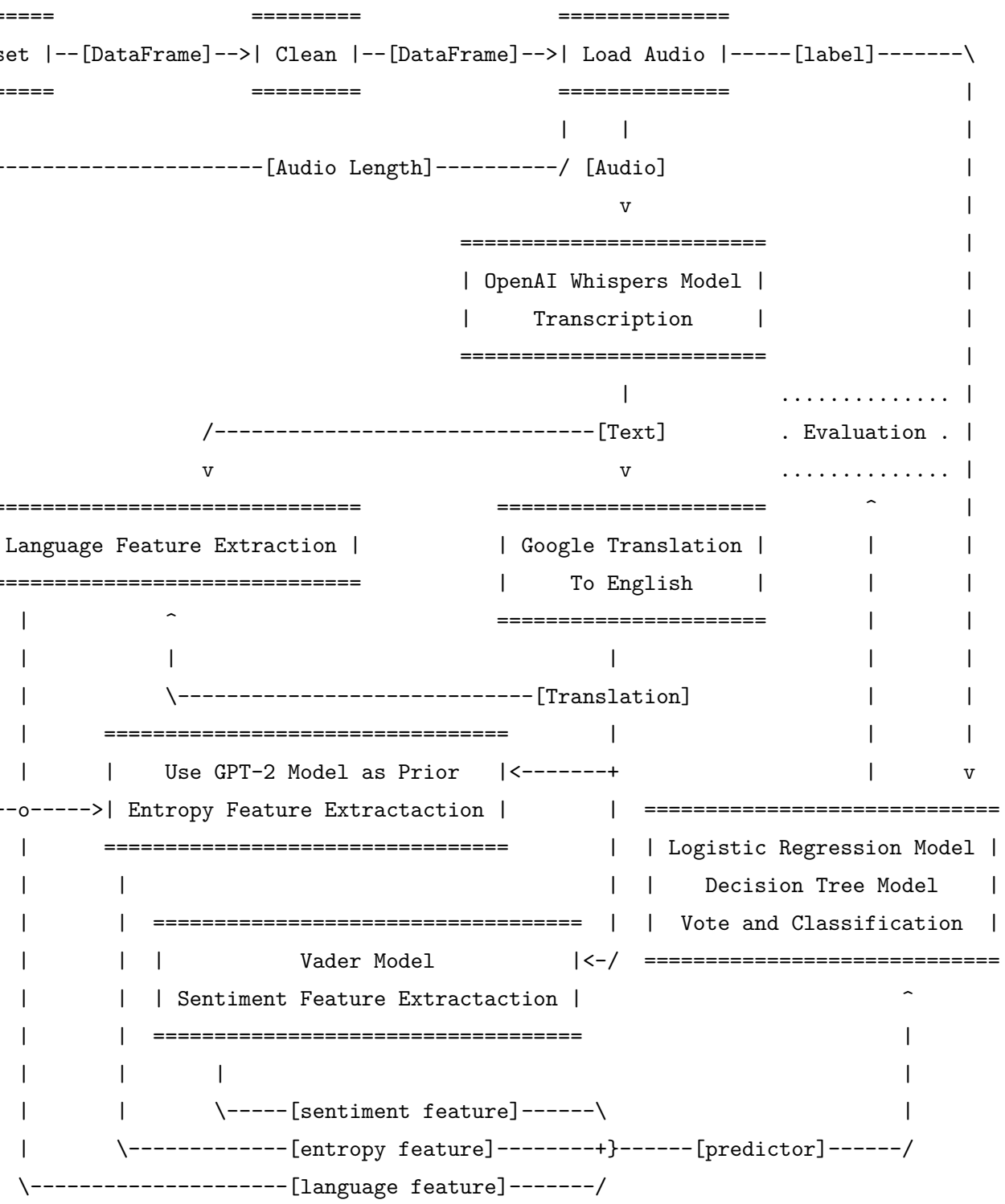
and the label of the model is:

- true or fake

and we will build a model simple enough to train on this dataset.

4 Implemented ML prediction pipelines

You can see the overall pipeline in the ASCII art below, make sure you use monospaced font to view it correctly.



4.1 Transformation stage

4.1.1 Audio Loading

Load the audio into the memory.

[2]:
```
std_rate = 24000

def load_audio(filename):
    audio, rate = sf.read(f'input/CBU0521DD_stories/{filename}')
    if len(audio.shape) != 2:
        assert len(audio.shape) == 1
        audio = np.stack([audio, audio], axis=1)
    if rate != std_rate:
        audio = np.stack([librosa.resample(audio[:, i], orig_sr=rate,
                          target_sr=std_rate) for i in range(audio.shape[1])],␣
 ↪axis=1)
    return audio
```

4.1.2 Transcription

The audio files are transcribed to text using OpenAI Whispers Model.

[3]:
```
whisper_model = whisper.load_model('medium')
converter = opencc.OpenCC('t2s')

def transcript(filename, native):
    result = whisper_model.transcribe(f'input/CBU0521DD_stories/{filename}')
    text = result['text']
    if native:
        text = converter.convert(text)
    return text
```

```
<masked>/Projects/ML_MiniProject/.venv/lib/python3.12/site-
packages/whisper/__init__.py:150: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
```

```
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  checkpoint = torch.load(fp, map_location=device)
```

4.1.3 Translation

The text is translated to English using Google Translation. (if it's not in English)

[4]:
```
translator = GoogleTranslator(source='zh-CN', target='en')
nltk.download('punkt')

def try_infinite(func, *args, **kwargs):
    while True:
        try:
            return func(*args, **kwargs)
        except Exception:
            # traceback.print_exc()
            pass

def translate(text, native):
    translated = try_infinite(translator.translate, text) if native else text
    return translated

examples = [
    ('如果同时使用行和列的规范化经测试上述做法将完全不奏效程序仍然跑得飞快我们换一种
构造方法', True),
    ('这是一段由 Whisper 转义产生的没有标点符号的文本我们将尝试使用 Google 翻译将其
翻译成英文以便测试谷歌能否处理没有标点符号的文本', True),
    ('If both row and column normalization are used, the above method will not␣
 ↪work at all. The program still runs very fast. We will try another␣
 ↪construction method.', False),
```

```
    ('This is a piece of text generated by Whisper without punctuation. We will␣
 ↪try to use Google Translate to translate it into English for testing whether␣
 ↪Google can handle text without punctuation.', False),
]

for example, native in examples:
    print(f"Text: {example}")
    print(f"Translated: {translate(example, native)}")
```

```
[nltk_data] Downloading package punkt to <masked>/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

Text: 如果同时使用行和列的规范化经测试上述做法将完全不奏效程序仍然跑得飞快我们换一种构造方法
Translated: If we use both row and column normalization, the above approach will not work at all. The program still runs very fast. Let's change the construction method.
Text: 这是一段由 Whisper 转义产生的没有标点符号的文本我们将尝试使用 Google 翻译将其翻译成英文以便测试谷歌能否处理没有标点符号的文本
Translated: This is a text without punctuation generated by Whisper. We will try to translate it into English using Google Translate to test whether Google can handle text without punctuation.
Text: If both row and column normalization are used, the above method will not work at all. The program still runs very fast. We will try another construction method.
Translated: If both row and column normalization are used, the above method will not work at all. The program still runs very fast. We will try another construction method.
Text: This is a piece of text generated by Whisper without punctuation. We will try to use Google Translate to translate it into English for testing whether Google can handle text without punctuation.
Translated: This is a piece of text generated by Whisper without punctuation. We will try to use Google Translate to translate it into English for testing whether Google can handle text without punctuation.

4.1.4 Language Feature Extraction

We take the word diversity and sentence count as features.

[5]:
```
def language_features(text, translated, native):
    tokens = list(jieba.lcut(re.sub(r'[^\w\s]', '', text))) if native \
        else [x.lower() for x in re.findall(r'\w+', text)]

    word_diversity = len(set(tokens)) / len(tokens)
    # Native text transcripted by Whisper is not punctuated, so use translated␣
 ↪text to estimate sentence count
    # Usually translatio does not change the number of sentences
    sentence_count = len(nltk.sent_tokenize(translated))
    return word_diversity, sentence_count

examples = [
    ('If I go to Beijing, I will go to Beijing, otherwise I will not go to␣
 ↪Beijing.',
     'If I go to Beijing, I will go to Beijing, otherwise I will not go to␣
 ↪Beijing.', False),
    ('如果我去北京，我会去北京，否则我不会去北京。',
     'If I go to Beijing, I will go to Beijing, otherwise I will not go to␣
 ↪Beijing.', True),
    ('If I go to Beijing, I am going to visit the Summar Palace, or I may␣
 ↪participate in the Olympic Games.',
     'If I go to Beijing, I am going to visit the Summer Palace, or I may␣
 ↪participate in the Olympic Games.', False),
    ('如果我去北京，我会去参观颐和园，或者我可能参加奥运会。',
     'If I go to Beijing, I am going to visit the Summer Palace, or I may␣
 ↪participate in the Olympic Games.', True),
]

for example, translated, native in examples:
    print(f"Text: {example}")
    print(f"Translated: {translated}")
    print(f"Word diversity, sentence count: {language_features(example,␣
 ↪translated, native)}")
```

```
Building prefix dict from the default dictionary …
Loading model from cache /tmp/jieba.cache
```

```
Text: If I go to Beijing, I will go to Beijing, otherwise I will not go to
Beijing.
Translated: If I go to Beijing, I will go to Beijing, otherwise I will not go to
Beijing.
Word diversity, sentence count: (0.47058823529411764, 1)
Text: 如果我去北京，我会去北京，否则我不会去北京。
Translated: If I go to Beijing, I will go to Beijing, otherwise I will not go to
Beijing.

Loading model cost 0.453 seconds.
Prefix dict has been built successfully.

Word diversity, sentence count: (0.5833333333333334, 1)
Text: If I go to Beijing, I am going to visit the Summar Palace, or I may
participate in the Olympic Games.
Translated: If I go to Beijing, I am going to visit the Summer Palace, or I may
participate in the Olympic Games.
Word diversity, sentence count: (0.8095238095238095, 1)
Text: 如果我去北京，我会去参观颐和园，或者我可能参加奥运会。
Translated: If I go to Beijing, I am going to visit the Summer Palace, or I may
participate in the Olympic Games.
Word diversity, sentence count: (0.8461538461538461, 1)
```

4.1.5 Entropy Feature Extraction

We take the entropy, entropy per token as features.

[6]:
```python
def entropy_features(translated):
    model_name = 'gpt2-large'
    tokenizer = GPT2Tokenizer.from_pretrained(model_name)
    model = GPT2LMHeadModel.from_pretrained(model_name)
    model.eval()

    inputs = tokenizer(translated, return_tensors='pt')
    input_ids = inputs['input_ids']

    with torch.no_grad():
        outputs = model(input_ids)
        logits = outputs.logits
```

```
    entropy = 0.0
    for i in range(logits.size(1) - 1):
        probs = torch.softmax(logits[0, i], dim=-1)
        target_id = input_ids[0, i + 1].item()
        target_prob = probs[target_id].item()
        entropy -= math.log(target_prob)

    return entropy, entropy / (logits.size(1) - 1) # Persplexity = exp(entropy␣
 ↪per token)

examples = [
    'I won gold medal! I am so happy!',
    'I won gold medal! I am so sad!',
    'I won gold medal! You cheaters get imprisoned!',
    'I won gold medal! Wonderful Beijing Huawei Laboratory!',
]

for example in examples:
    print(f"Text: {example}")
    print(f"Entropy: {entropy_features(example)}")
```

```
Text: I won gold medal! I am so happy!
Entropy: (33.80992141345526, 3.756657934828362)
Text: I won gold medal! I am so sad!
Entropy: (39.4239305236529, 4.380436724850322)
Text: I won gold medal! You cheaters get imprisoned!
Entropy: (57.63893993780777, 5.763893993780777)
Text: I won gold medal! Wonderful Beijing Huawei Laboratory!
Entropy: (72.35215950479618, 8.039128833866242)
```

4.1.6 Sentiment Feature Extraction

We use Vadar model to extract sentiment features.

[7]:
```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

analyzer = SentimentIntensityAnalyzer()
```

```python
def sentiment_features(translated):
    results = []
    for sentence in nltk.sent_tokenize(translated):
        sentiment = analyzer.polarity_scores(sentence)
        results.append(sentiment['compound'])

    results = np.array(results)
    overall = analyzer.polarity_scores(translated)

    return overall['compound'], results.min().item(), results.max().item()

examples = [
    'I won gold medal! Ten thousand dollars!',
    'Honorable mention! Worse than a pig! I\'m to suicide!',
]

for example in examples:
    print(f"Text: {example}")
    print(f"Sentiment: {sentiment_features(example)}")
```

```
Text: I won gold medal! Ten thousand dollars!
Sentiment: (0.8118, 0.0, 0.7959)
Text: Honorable mention! Worse than a pig! I'm to suicide!
Sentiment: (-0.7163, -0.6996, 0.5848)
```

4.2 Model stage

We use logistic regression model and decision tree to classify the stories. This is mainly due to the small size of the dataset.

- We don't have enough data to train a deep learning model.
- Logistic regression is simple and interpretable, 9 parameters in total. Decision tree is simple as well, 11 float-valued parameters in total. They will be trainable even we only have 75 samples.
- We are have high confidence that these two distributions have different mean and variance in the visualized data. Logistic regression can at least tell some kind of difference between these two distributions.
- Some features, such as mean entropy [1], solely classify obviously fake stories, this matches

decision tree's intuition that it splits the data into two parts each time by a predictor. (I guess these stories are really generated by a generative model but not a human.)

- Both models suffers from distribution shift from training data to testing data since the dataset is small, but the ensemble model can alleviate this problem.
- We don't use SVMs because these two distributions overlap a lot in the visualized data, and SVM behaves strangely in this case.
- Naive Bayes is not used because there is nothing todo with normal distribution in this case.

Trainable parameters summary:

Model	Float-valued parameters	Other parameters
Logistic Regression	8 `weights`, 1 `bias`	
Decision Tree	11 `thresholds`	`tree topology`
	20	

[1] It's observed that in training data, if the mean entropy is too low, the story is definitely fake.

[220]:
```
model1 = LogisticRegression(penalty='l1', max_iter=1000, C=1e2,␣
↪solver='liblinear')
model2 = DecisionTreeClassifier(max_depth=4, max_leaf_nodes=12,␣
↪criterion='log_loss')
```

4.3 Ensemble stage

We let the two models vote and classify the stories. The two models have different features and may have different opinions on the same story. This greatly reduces the instability of the prediction.

[222]:
```
model = VotingClassifier([
    ('lr', model1),
    ('dt', model2),
], voting='soft', weights=[1, 3])
```

5 Dataset

The dataset is build on top of https://github.com/CBU5201Datasets/Deception, with data cleanning and feature extraction and train test splitting.

5.1 Cleaning

We listen to all the audio files remove non-story audio files, ridiculous stories due to the extremely poor quality of the dataset.

Blacklist criteria:

- If even humans can't audibly understand the story, it is blacklisted.
- If the audio is describing an item or supporting a football team instead of telling a story, it is blacklisted.
- If there is the place or person is not mentioned, it is blacklisted.
- If the audio not correctly labeled as English or Mandarin, it is blacklisted.
- If the story in the audio is 'obviously' impossible, it is blacklisted. (one of the discarded samples is even labeled as true)

The dataset after cleaning is almost balanced. So we don't have to worry about it.

Used samples:

	True	Fake
Native	22	22
English	22	23

Samples discarded during data cleaning:

	True	Fake
Native	3	3
English	3	2

We need to ensure the balance of the testing dataset, so we systematicly select 3 samples from each of the 4 classes, 12 samples in total, into testing dataset, and leave the rest in the training dataset.

[10]:
```
blacklisted_audios = [
    # Don't know what he is talking
    '00017.wav',

    # Not story
    '00021.wav',
```

```
    '00051.wav',
    '00087.wav',
    '00012.wav',
    '00068.wav',

    # Not visiting a place or person
    '00041.wav',
    '00088.wav',

    # Not in native language
    '00094.wav',

    # Ridiculous story
    '00001.wav',
    '00030.wav',
]
```

5.2 Preprocessing

We load the audio, extract features and save it into a CSV file. This take half an hour, you don't want to run this again. So you can recover the preprocessed data from the CSV file.

[ ]:
```
def process_sample(filename, language, story_type):
    audio = load_audio(filename)
    audio_length = audio.shape[0] / std_rate
    native = language.strip() != 'English'
    fake = story_type != 'True Story'
    text = transcript(filename, native)
    translated = translate(text, native)

    word_diversity, sentence_count = language_features(text, translated, native)
    word_diversity_per_second = word_diversity / audio_length
    sentence_per_second = sentence_count / audio_length
    entropy, entropy_mean = entropy_features(translated)
    entropy_per_second = entropy / audio_length
    sentiment_f = sentiment_features(translated)
```

```
    return {
        'filename': filename,
        'audio_length': audio_length,
        'native': native,
        'fake': fake,
        'text': text,
        'text_length': len(text),
        'translated': translated,
        'translated_length': len(translated),
        'audio_over_text': audio_length / len(text),
        'word_diversity': word_diversity,
        'sentence_count': sentence_count,
        'sentence_per_second': sentence_per_second,
        'entropy': entropy,
        'entropy_mean': entropy_mean,
        'entropy_per_second': entropy_per_second,
        'sentiment': sentiment_f[0],
        'sentiment_min': sentiment_f[1],
        'sentiment_max': sentiment_f[2],
        'word_diversity_per_second': word_diversity_per_second,
    }


def load_dataset():
    data = pd.read_csv('input/CBU0521DD_stories_attributes.csv')
    filtered_data = data[~data['filename'].isin(blacklisted_audios)]
    data_list = list(filtered_data.itertuples(index=False, name=None))
    return data_list

dataset = load_dataset()
preprocessed = [process_sample(*x) for x in dataset]
frame = pd.DataFrame(preprocessed)
frame.to_csv('preprocessed.csv', index=False)
```

5.3 Train Test Splitting and Normalization

Select 3 samples in each of 4 classes in to testing dataset and leave the rest in the training set.

Testing dataset:

	True	Fake
Native	3	3
English	3	3

Training dataset:

	True	Fake
Native	19	19
English	19	20

About the normalization, we choose to divide the features by mean value instead of min-max scaling, because the the min-max are much more prone to outliers, and we have two languages, it's hard to balance the two languages. We don't choose standard scaling because these features have nothing to do with normal distribution.

[261]:
```
N_TEST = 3

# These features are language dependent, normalize them with scaler with␣
 ↪respect to language
native_features = ['word_diversity', 'audio_over_text']
# These features do not need to be normalized with respect to language
global_features = ['entropy', 'entropy_mean', 'entropy_per_second',␣
 ↪'sentiment', 'sentiment_min', 'sentiment_max']

random.shuffle(preprocessed)
train_samples = []
test_samples = []
test_count_for_each_class = collections.defaultdict(int)
for sample in preprocessed:
    if test_count_for_each_class[(sample['native'], sample['fake'])] < N_TEST:
        test_samples.append(sample)
        test_count_for_each_class[(sample['native'], sample['fake'])] += 1
    else:
        train_samples.append(sample)
```

```
train_samples_native = [x for x in train_samples if x['native']]
train_samples_english = [x for x in train_samples if not x['native']]
test_samples_native = [x for x in test_samples if x['native']]
test_samples_english = [x for x in test_samples if not x['native']]

native_mean = np.array([[x[f] for f in native_features] for x in␣
 ↪train_samples_native]).mean(axis=0)
english_mean = np.array([[x[f] for f in native_features] for x in␣
 ↪train_samples_english]).mean(axis=0)

train_samples_native_normalized = np.array([[x[f] for f in native_features] for␣
 ↪x in train_samples_native]) / native_mean
train_samples_english_normalized = np.array([[x[f] for f in native_features]␣
 ↪for x in train_samples_english]) / english_mean
train_samples_normalized = np.concatenate([
    np.concatenate([train_samples_native_normalized,␣
 ↪train_samples_english_normalized], axis=0),
    np.array([[x[f] for f in global_features] for x in train_samples])
], axis=1)
test_samples_native_normalized = np.array([[x[f] for f in native_features] for␣
 ↪x in test_samples_native]) / native_mean
test_samples_english_normalized = np.array([[x[f] for f in native_features] for␣
 ↪x in test_samples_english]) / english_mean
test_samples_normalized = np.concatenate([
    np.concatenate([test_samples_native_normalized,␣
 ↪test_samples_english_normalized], axis=0),
    np.array([[x[f] for f in global_features] for x in test_samples])
], axis=1)


train_labels = np.array([x['fake'] for x in train_samples])
test_labels = np.array([x['fake'] for x in test_samples])
```

6 Experiments and results

6.1 Fit and Predict

We simply fit the model and print accuracy on training dataset and testing dataset.

[295]:
```
model.fit(train_samples_normalized, train_labels)

print("Train accuracy:", model.score(test_samples_normalized, test_labels))
predicted = model.predict(test_samples_normalized)
print(classification_report(test_labels, predicted))

print("Train accuracy:", model.score(train_samples_normalized, train_labels))
predicted = model.predict(train_samples_normalized)
print(classification_report(train_labels, predicted))

for test_sample, pred in zip(test_samples, predicted):
    print(f"File: {test_sample['filename']}, Predicted: {'Fake' if pred else␣
 ↪'True'}"
          f", Actual: {'Fake' if test_sample['fake'] else 'True'}")
```

```
Train accuracy: 0.75
              precision    recall  f1-score   support

       False       0.80      0.67      0.73         6
        True       0.71      0.83      0.77         6

    accuracy                           0.75        12
   macro avg       0.76      0.75      0.75        12
weighted avg       0.76      0.75      0.75        12

Train accuracy: 0.7922077922077922
              precision    recall  f1-score   support

       False       0.89      0.66      0.76        38
        True       0.73      0.92      0.82        39

    accuracy                           0.79        77
```

```
   macro avg       0.81      0.79      0.79        77
weighted avg       0.81      0.79      0.79        77

File: 00044.wav, Predicted: Fake, Actual: Fake
File: 00098.wav, Predicted: Fake, Actual: True
File: 00015.wav, Predicted: True, Actual: Fake
File: 00013.wav, Predicted: Fake, Actual: Fake
File: 00020.wav, Predicted: True, Actual: True
File: 00069.wav, Predicted: Fake, Actual: Fake
File: 00082.wav, Predicted: Fake, Actual: True
File: 00070.wav, Predicted: Fake, Actual: Fake
File: 00090.wav, Predicted: Fake, Actual: Fake
File: 00049.wav, Predicted: True, Actual: True
File: 00037.wav, Predicted: Fake, Actual: True
File: 00055.wav, Predicted: True, Actual: True
```

6.2 Evaluation

The model's accuracy is much dependent on train test splitting. In most cases, the model give a test accuracy of $\frac{8}{12} = 0.67$ (27% chances), $\frac{9}{12} = 0.75$ (23% chances) or $\frac{7}{12} = 0.58$ (19% chances) [1] on different train test splitting. But in rare cases, the accuracy can be as low as 0.25 or as high as 0.92. The f1 score is also around 0.67, which is good.

Since the dataset is small and we have only 3 samples for each class in the test set, distribution always shifts.

I have listened to the audios and classified them by myself, and the accuracy is around 0.65, so I believe the model is doing a good job. It gets **almost same accuracy as me** with much less effort, no millions of parameters and hours of training, only 20 **parameters and** 1 **millisecond of training**[2].

Unlike neural networks, since both logistic regression and decision tree model have good explainability, let's see what they have learned.

[1] The chance of getting each accuracy is calculated by running 10000 train test splitting, training the model and record the accuracy.
[2] The training time is calculated by running 10000 training and averaging the time.
[3] Some students propose the idea to stop training when the test accuracy drops, or in another words, to take the maximum test accuracy during training process instead of the converged accuracy. I completely OBJECT this idea, because **if you look at the testing accuracy to determine**

when to stop training, you are leaking the testing data into the training process, which make no sense. So all the accuracy mentioned above is the converged accuracy.

[296]:
```
model.estimators_[0].coef_, model.estimators_[0].intercept_
```

[296]:
```
(array([[-5.62118396e+00,  6.26005131e-01,  6.08006757e-04,
         -1.20026742e+00, -1.08222542e-01,  5.14939333e+00,
         -1.94097978e+00,  2.90775810e+00]]),
 array([0.57221014]))
```

The logistic regression model tells us that fake stories have:

- high entropy but low entropy per token or per second (low perplexity)
- positive and stable sentiment
- low speaking speed and low word diversity

This basically agree with our assumptions.

[297]:
```
model.estimators_[1].feature_importances_
```

[297]:
```
array([0.21408523, 0.        , 0.19270556, 0.17658963, 0.        ,
       0.41661958, 0.        , 0.        ])
```

The decision tree model tells us that it's more important to look at:

- sentiment first
- entropy second
- language third

This mainly swaps the order of sentiment feature and entropy feature in our assumption. There are some guess below about this phenomenon:

- Estimating entropy by GPT-2 may be not very accurate, thus include noise in the model.
- Low entropy may means stable sentiment, so they are not independent as we thought.
- There may be indeed some difference between humans and auto regressive generative models.
- The dataset is noise and small, so maybe there are overfitting.

but it's very unclear and need more research to confirm.

7 Conclusions

7.1 Summary

According to what out models have learned, we can conclude that fake stories have (in descending order of importance):

- positive and stable sentiment
- high entropy sum but low perplexity
- low speaking speed and diversity

And the lesson we learned from this project is that humans are displaying chatbot-like behavior when they are telling fake stories.

7.2 Suggestions for improvements

7.2.1 Low audio feature utilization

The only used audio feature is audio length, and the audio is completely discarded after the transcription.

We are meant to use more audio or even mainly work on audio features, but I personally don't know audio well and I didn't find any evidence of difference between true and fake stories in common audio features, like mfcc, f0, intensity, formants.

In the futures works, we may explore more audio features. Probably wav2vec2 is a good choice if we have more data.

7.2.2 Poor sentiment analysis model

Vader model does not behave well in this task. It is basically a rule-based model, and can not capture complex expressions (like sarcasm, or some long phrases that are generally negative but does not have negative words in it). It's output is discrete and not continuous, but we assume that sentiment is continuous.

It should be better if we train another deep-learning model based on another dataset.

7.2.3 Poor dataset

This dataset is awfully small and low-quality. 89 samples are far from enough to train a deep learning model.

And most recordings are reading stories in a steady tone, and seems to have practiced for many times which is contradict to the task statement that just pretend to talk to your friend.

I suggest in future mini-projects, we would have a dataset:

- with more samples
  - at least 3600 samples.
- less dirty
  - Stories should be stories, not other things.
- labeled correctly
  - We don't want to hear a 'true' story where a 'high school student' is complaining about his 'job' again!
  - And 'Mandarin' story should be in Mandarin, not in English!
  - We don't know how many potential problematics samples not that obvious are not discovered.

8 References

8.1 Knowledge

The project is mostly based on lecture content in Machine Learning and Computational Creativity, and prior life experiences. Nothing special to reference.

8.2 Libraries

Models are built on top of:

- OpenAI Whispers
- Google Translation
- Vader
- GPT-2
- Scikit-learn

Other used libraries can be found on the top of the notebook.

8.3 Miscellaneous

My classmate <masked> greatly inspired me to solve this problem in this way, although his approach to this task is completely different. I thank him very much.

He used a completely different approach on the task, which primarily works on audio features and assisted with few text features. He used a big deep learning model with 10K parameters and achieved a test accuracy of 0.75. I don't know how he managed to train such a big model on such a small dataset up to now, maybe he will reveal it in after the deadline of this coursework. It's amazing!

At that time, I was struggling working with text CNN and RNN models to extract text features, but ended up with severe overfitting where training loss went to 10^{-6} and test loss went to 10^{3} and accuracy was exactly 0.5. So I had a talk with my friend <masked>. (He is also a student in this course.)

He persuaded me to give up text features.

He said, "Can you tell the true stories from the fake stories by these text?"

"Basically I have tried to. Manually reading the stories and classifying them gives me an accuracy of 0.6. But most samples are too hard to classify."

"Then I don't think it's possible for the model tell the difference from text if even you are struggling to do so. Because you know, **you are basically a generative model but much more well trained than those large language models**. It's no use to ask those pre-trained language models to do the what you can't do.", he said.

I have thought about the idea that humans are well-trained generative models for a few days. I love this metaphor.

At a certain moment, I suddenly realized that telling fake stories from nothing is actually an unconditional generation task. We know nothing about humans but too much about generative models. If humans are generative models, then features to tell model-generated stories from real stories should more or less make sense in telling fake stories from true stories.

This is the fundamental idea of this project, replacing humans with generative models in the assumption that humans are generative models. Based on this idea, I successfully extracted text features and built a model with only 20 parameters and achieved a test accuracy of 0.67.

This is completely out of <masked>'s expectation: I neither give up text features nor find any audio features useful. Actually, I visualized many audio features, such as f0, intensity, mfcc, but I can tell no difference on the distribution. I'm not sure how <masked> make use of audios.