bc_evaluation_notebook

September 16, 2024

1 Evaluation: translation webapp

For the evaluation phase I have used the provided Hungarian dataset. Before I started the evaluation phase, I considered different scorings and metrics suitable for evaluating the quality of translations: - **BLEU (Bilingual Evaluation Understudy) Score** (most commonly use, briefly: it compares n-grams) - **METEOR (Metric for Evaluation of Translation with Explicit ORdering)** (different than BLEU, considers synonyms, stemming, etc) - **TER (Translation Edit Rate)** (number of edits from candidate to reference, might not catch semantics) - **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)** (widely used, works like BLEU) - **Embedding-based Metrics (BERTScore)** (uses LLM models to compare text, very powerful tool, computationally intensive) *favourite - **Human Evaluation** (non automated, but it might be the gold standard) - **Hybrid model** (some kind of combination of the above metrics) <- choose this

I finally choose a hybrid model which uses BLEU and METEOR. They are really a powerful combination as they complete each other (BLEU is strong in literal word overlap, while METEOR accounts for synonyms and paraphrasing). Both metrics range between 0-1, 1 being a perfect match, therefore combination - again - is very easy to implement. If I had more time I would have involved BERTScore as well (with a higher weight) in this hybrid score as embedding-based metrics are adept at understanding nuanced meanings beyond mere word matches.

I start with the importation of modules.

[1]:
```
# import modules

import warnings
warnings.simplefilter("ignore", UserWarning)

import plotly.express as px
import pandas as pd
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import nltk
from nltk.tokenize import word_tokenize
from custom_modules.chatengine import ChatEngine
from nltk.translate.bleu_score import sentence_bleu
```

```
import re
import os
from typing import List
from nltk.translate.meteor_score import meteor_score
nltk.download('wordnet')
nltk.download('punkt')
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Lori\AppData\Roaming\nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\Lori\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

[1]: True

[7]:
```
from pygoogletranslation import Translator
```

As next step I define the formulas needed for data handling and also score calculations.

[8]:
```
# formula definitions

def google_translator(translation_text:str)->str:
    translator = Translator()
    translated_text=translator.translate(translation_text, dest='hu').text
    return translated_text

def broker_chooser_translator(translation_text:str)->str:
    ce=ChatEngine()
    translated_text=ce.
  ↪run_engine(user_prompt=translation_text,language="Hungarian")
    return translated_text

def clean_text(text:str)->str:
    result=re.sub('[^\w\s]','', text)
    return result

def create_bleu_score(df_row:pd.Series,candidate_name:str)->pd.Series:
    reference=[[word.lower() for word in clean_text(df_row["orig_trans"]).
  ↪split(" ")]]
    candidate=[word.lower() for word in clean_text(df_row[candidate_name]).
  ↪split(" ")]
    score = sentence_bleu(reference, candidate)
    return score

def log_min_max_normalize(data:pd.Series)->pd.Series:
    log_data = np.log10(data + 1e-300)  # Adding a small constant to avoid␣
↪log(0)

    # Min-Max Normalization on log-transformed data
    min_val = np.min(log_data)
    max_val = np.max(log_data)
    normalized_data = (log_data - min_val) / (max_val - min_val)

    return normalized_data

def create_meteor_score(df_row:pd.Series,candidate_name:str)->pd.Series:
    reference_sentence=clean_text(df_row["orig_trans"])
    candidate_sentence=clean_text(df_row[candidate_name])
    reference = word_tokenize(reference_sentence)
    candidate = word_tokenize(candidate_sentence)
    score = meteor_score([reference], candidate)
    return score

def create_custom_score(df_row:pd.Series,is_own_score:bool)->pd.Series:
    if is_own_score:
        result=np.round(df_row["own_bleu_sc"]*df_row["own_meteor_sc"],4)
    else:
        result=np.round(df_row["google_bleu_sc"]*df_row["google_meteor_sc"],4)
    return result
```

Next I load the csv file - containing the Hungarian translations - into a pandas dataframe.

[9]:
```
# load hungarian text file

df=pd.read_csv("translated_output.csv")
df.rename(columns={"translated_value":"orig_trans"}, inplace=True)
```

Next, I use the brooker_chooser translation engine to translate the English text to Hungarian. The results are stored in the 'own_trans' column.

[10]:
```
# add own translations

df['own_trans']=df["english"].apply(broker_chooser_translator)
```

As next step, I use the google translator to translate the English text to Hungarian. The results are stored in the 'google_trans' column.

[11]:
```
# add google translated text to new column

df['google_trans']=df["english"].apply(google_translator)
```

[12]:
```
# OPTIONAL: load saved CSV

# df=pd.read_csv("own_and_google_translated.csv")
```

Next I perform the BLEU calculations on the google translations. As some of the values are very very small numbers (1.88747e-253) I also use logarithmic minmax scaling in order to make those small values more humanlike readable.

[13]:
```
# create bleu score for google translation

df["google_bleu"]=df.apply(lambda row:
 ↪create_bleu_score(df_row=row,candidate_name="google_trans"), axis=1)
df["google_bleu_sc"]=log_min_max_normalize(df["google_bleu"])
```

I do the same for the translations performed by the chat engine.

[14]:
```
# create bleu score for own translation

df["own_bleu"]=df.apply(lambda row:
 ↪create_bleu_score(df_row=row,candidate_name="own_trans"), axis=1)
df["own_bleu_sc"]=log_min_max_normalize(df["own_bleu"])
```

Next, I calculate the METEOR scores as well on the google translations. Logarithmic minmax scaling is also applied.

[15]:
```
# create meteor score for google translation

df["google_meteor"]=df.apply(lambda row:
 ↪create_meteor_score(df_row=row,candidate_name="google_trans"), axis=1)
df["google_meteor_sc"]=log_min_max_normalize(df["google_meteor"])
```

METEOR score calculation for the chat engine translations.

[16]:
```
# create meteor score for own translation

df["own_meteor"]=df.apply(lambda row:
 ↪create_meteor_score(df_row=row,candidate_name="own_trans"), axis=1)
df["own_meteor_sc"]=log_min_max_normalize(df["own_meteor"])
```

As a final step I create the hybrid score, which is a multiplication of the BLEU and METEOR scores. Of course defining a more plausible hybrid score would require lot more time and analysis. It should be an iterative process which involves score evaluations by understanding more these scores against different text datasets. In a production project I would be creating different text cohorts as well to examine and identify the pros and cons of the models (and metrics) in case of different textual environments. This would also enable me to improve the models. I also added some calculated columns for easier performance interpretation.

[17]:
```
# create custom scores

df["own_hybrid"]=df.apply(lambda row:
 ↪create_custom_score(df_row=row,is_own_score=True), axis=1)
df["google_hybrid"]=df.apply(lambda row:
 ↪create_custom_score(df_row=row,is_own_score=False), axis=1)
```

```
df["own_win"] = (df["own_hybrid"]>df["google_hybrid"]).astype(int)
df["own_bleu_win"] = (df["own_bleu"]>df["google_bleu"]).astype(int)
df["own_meteor_win"] = (df["own_meteor"]>df["google_meteor"]).astype(int)
df["google_win"] = (df["google_hybrid"]>df["own_hybrid"]).astype(int)

df.to_csv("final_df.csv")
```

I created a smaller dataset for interpreting the results.

[18]:
```
interest_cols=['english', 'orig_trans', 'google_trans',␣
 ↪'own_trans','google_bleu_sc','own_bleu_sc','google_meteor_sc',
               'own_meteor_sc','google_hybrid','own_hybrid', 'own_bleu_win',␣
 ↪'own_meteor_win','own_win', 'google_win']

exp_df=df[interest_cols].copy(deep=True)

exp_df.describe()
```

[18]:
```
       google_bleu_sc  own_bleu_sc  google_meteor_sc  own_meteor_sc  \
count       23.000000    23.000000         23.000000      23.000000
mean         0.432380     0.535966          0.955092       0.688286
std          0.253814     0.398355          0.208204       0.370610
min          0.000000     0.000000          0.000000       0.000000
25%          0.230713     0.166661          0.997481       0.609810
50%          0.486343     0.665382          0.998542       0.820046
75%          0.614986     0.998656          0.999183       0.969328
max          1.000000     1.000000          1.000000       1.000000

       google_hybrid  own_hybrid  own_bleu_win  own_meteor_win    own_win  \
count      23.000000   23.000000     23.000000       23.000000  23.000000
mean        0.431874    0.480617      0.782609        0.826087   0.478261
std         0.253806    0.367417      0.421741        0.387553   0.510754
min         0.000000    0.000000      0.000000        0.000000   0.000000
25%         0.230100    0.125050      1.000000        1.000000   0.000000
50%         0.485500    0.535600      1.000000        1.000000   0.000000
75%         0.614500    0.775200      1.000000        1.000000   1.000000
max         0.999600    1.000000      1.000000        1.000000   1.000000

       google_win
count   23.000000
mean     0.521739
std      0.510754
min      0.000000
25%      0.000000
50%      1.000000
```

```
75%      1.000000
max      1.000000
```

As we can see in the table above (and charts below) in case of the BLEU score the custom chat engine performs slightly better, as in case of the METEOR score the google translator seems to be way more accurate than the chat engine. When we compare the hybrid score we can see that in 12 cases the google translator got a higher score while in 11 cases the own model was performing better.

[19]:
```python
# comparing BLEU score means against each other

data = exp_df[["google_bleu_sc", "own_bleu_sc"]].mean()

df = data.reset_index()
df.columns = ['Metric', 'Value']

fig = px.bar(df, x='Metric', y='Value', title='Comparison of BLEU Scores',
             labels={'Value': 'Mean Value', 'Metric': 'Source'},
             color='Metric',
             text=df['Value'].apply(lambda x: f'{x:.4f}'))

fig.update_layout(
    xaxis_title='Source',
    yaxis_title='Mean Value',
    yaxis=dict(tickformat=".4f"),
    showlegend=False
)

fig.show()
```

[20]:
```python
# comparing METEOR score means against each other

data = exp_df[["google_meteor_sc", "own_meteor_sc"]].mean()

df = data.reset_index()
df.columns = ['Metric', 'Value']

fig = px.bar(df, x='Metric', y='Value', title='Comparison of METEOR Scores',
             labels={'Value': 'Mean Value', 'Metric': 'METEOR scores'},
             color='Metric',
             text=df['Value'].apply(lambda x: f'{x:.4f}'))

fig.update_layout(
    xaxis_title='Source',
    yaxis_title='Mean Value',
    yaxis=dict(tickformat=".4f"),
    showlegend=False
)
```

```
fig.show()
```

[21]:
```
# comparing hybrid scoring performance against each other

df = exp_df[['google_win','own_win']]

counts = df.sum().reset_index()
counts.columns = ['Category', 'Count']

fig = px.bar(counts, x='Category', y='Count', title='Counts for Each Category',
             labels={'Count': 'Total Count', 'Category': 'Category'},
             color='Category',
             text='Count')

fig.update_layout(
    xaxis_title='Category',
    yaxis_title='Total Count',
    showlegend=False
)

fig.show()
```

[22]:
```
df = exp_df.describe()

def highlight_means(row):
    styles = [''] * len(row)
    if row.name == 'mean':
        google_hybrid_mean = row['google_hybrid']
        own_hybrid_mean = row['own_hybrid']
        if google_hybrid_mean > own_hybrid_mean:
            styles[df.columns.get_loc('google_hybrid')] = 'background-color:␣
↪lightgreen'
            styles[df.columns.get_loc('own_hybrid')] = 'background-color:␣
↪lightyellow'
        else:
            styles[df.columns.get_loc('google_hybrid')] = 'background-color:␣
↪lightyellow'
            styles[df.columns.get_loc('own_hybrid')] = 'background-color:␣
↪lightgreen'

        google_meteor_mean = row['google_win']
        own_meteor_mean = row['own_win']
        if google_meteor_mean > own_meteor_mean:
            styles[df.columns.get_loc('google_win')] = 'background-color:␣
↪lightgreen'
```

```
            styles[df.columns.get_loc('own_win')] = 'background-color:␣
↪lightyellow'
        else:
            styles[df.columns.get_loc('google_win')] = 'background-color:␣
↪lightyellow'
            styles[df.columns.get_loc('own_win')] = 'background-color:␣
↪lightgreen'
    return styles


styled_df = df.style.apply(highlight_means, axis=1)


styled_df
```

```
[22]: <pandas.io.formats.style.Styler at 0x2ae01b51c60>
```

1.1 Improvements

First of all: there are plenty areas for improvement! Due to lack of time I tried to rather show some of the methodologies I used,than striving for model optimization. Model optimization is a long way, with lots of iterations and backfalls. The optimization of a model for production involves a lot of time/effort but also lot of people (domain knowledge experts, business decision makers, etc) as it involved multiple areas and disciplines. I will gladly expand on this topic if you are open to go forward with me in the application process. There are at least 2 main topics in case of model optimization: 1. **Model evaluation methodology** As I described at the beginnig of this chapter, this even by itself is a huge topic. Even before getting to the heavy topic of which metric to use, first it must be well defined the reference and candidate texts formatting. Should we remove special characters? What about hyphens (in some languages they can alter the meanig of the word)? What about the numbers in the text? How to treat non latinic characters/languages? Should we apply stemming? Or maybe we should compare embeddings? So many questions, so many decisions need to be evaluated and made and we did not even touch the topic of the translation models. So, before even getting to decide on which metrics to use, we must first decide on the "cleanliness" and format of the reference and candidate texts. Only after this important step can we start considering evaluation methodologies and metrics. Of course as I stated at the beginning of this document this is another heavy topic. Based on my experience, in most of the cases, usually, I ended up using mix of metrics by adding different weights. In almost every case I arrived to these decisions by involving subject matter experts, business decision makers and other stakeholders who also shared their view on the relevan topic. Usually choosing metric is a trade-off decision. 2. **LLM optimisation** Once we have developed our scoring metrics we can start on the improvements of the model. This is an even bigger topic than the evaluation was. Without going in too much details (as this would require a long conversation) I would like to point out the main topics which could add to the better performance of the model:

- experimenting with different chat models (the list is huge, but its worth trying out open source models: llama, mistral, falcon, bloom, bert, flan etc and also closed source models: chatgpt, claude, gemini and many more); note: in case of open source models there is an added layer of risk as the license of these models are not always very clear!

- experimenting with different versions of a chat model (different versions perform differently)
- model fine tuning (after choosing the best performing model, training it further on specific brokerage-related corpus)
  - fine tuning involves gathering large corpus of multi language texts: this by itself is a huge topic with plenty steps
  - data augmentation: back-translation to create more data
- experimenting with different LLM agents types (reflection, reflexion, react, tools, etc)
- multi agent environments (for example: first translate (one LLM) then rephrase (another LLM))
- prompt engineering (chain of thought, chain of verification, COSTAR, etc)
- zero-shot/few shot prompting
- active learning: to continuously improve the model based on user feedback
- integration of a glossary of brokerage terms
- custom model development, I left this one as left resort as it is very time, computation, memory and knowledge intesive

Each of these topics are quite extensive and have many options.

As you can see there are lot's of options in order to improve a models performance. Optimization of LLM models is an iterative, non linear process, which needs expertise, persevearance and communication skills. If you choose me you will get all of these! ;)