MultiLingual NLP Lab1

XU SUN 22200118

September 2024

Part 1

Question 1 How is the TTR (type-token ratio) defined? Why is this a 'good' measure of the morphological complexity? Which other metric could you use to quantify morphological complexity?

1.1

Type-Token Ratio (TTR) is a measure of lexical variety in a given language sample. It is calculated by dividing the number of unique words (types) by the total number of words (tokens) in a text:

$$TTR = \frac{Numberofuniqueword}{Totalnumberofwords}$$

1.2

Morphological complexity refers to the richness of inflectional forms in a language. For instance, Hungarian exhibits high morphological complexity with 38-42 different possible forms for a basic inflected noun. TTR is more closer to 1, meaning the language has greater lexical richness[1]. TTR serves as an effective measure of morphological complexity because:

1. Languages with higher morphological complexity tend to have a greater variety of word forms.

2. This increased variety results in a higher number of unique words within a given text length.

3. Consequently, morphologically complex languages often yield higher TTR values.

4. TTR performs consistently across different language corpora (EU Law & Leipzig), indicating it is a stable and reliable measurement. This consistency suggests that TTR captures inherent morphological properties of languages rather than corpus-specific features [2].

1.3

An alternative method for approximating morphological complexity was proposed by Patrick Juola (1998, 2008), based on the Kolmogorov complexity and compression algorithms[2]. This approach involves the following steps:

1. Distorting word structures by assigning a unique random number to each word type.

2. Compressing both the original and distorted text data.

3. Calculating the ratio of the compressed original file size to the compressed distorted file size.

Question 2 Why do we consider a language modeling task?

Language modeling is considered in relation to quantifying morphological complexity through TTR methodology because both are unsupervised tasks, requiring no explicit gold labels. Language modeling aims to predict the probability of a given sequence of words or the next word in a sequence, which indirectly captures the morphological structure of a language.

Question 3 How is the perplexity defined? Can we compare perplexity across corpora or languages? Conclude.

Perplexity is a crucial metric for evaluating the performance of language models. It measures the model's ability to predict a given text, with lower perplexity indicating better predictive power.

$$\text{Perplexity}(W) = P(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} \tag{1}$$

Where $W = w_1, w_2, \ldots, w_N$ is a sequence of N words, and $P(w_1, w_2, \ldots, w_N)$ is the probability assigned to this sequence by the language model.
In practice, log perplexity is often used during training for computational efficiency and to avoid numerical underflow:

$$\log\left(\text{Perplexity}(W)\right) = -\frac{1}{N}\sum_{i=1}^{N} P(w_1, w_2, \ldots, w_N) \tag{2}$$

Question 4 What conclusions can you draw from Figure 1.

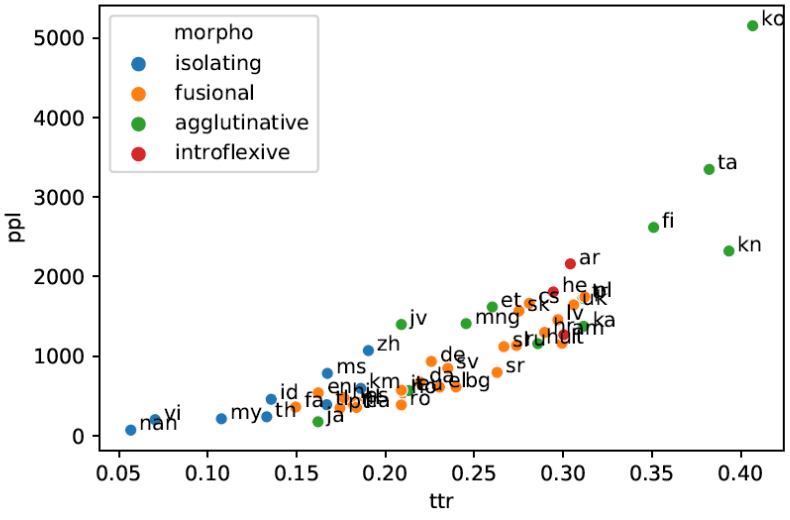

图 1: Perplexity of a language model in relation to the TTR for different kind of language and morphological systems.

Performance varies significantly across languages in language modeling tasks, as illustrated by Figure 1. This graph demonstrates the relationship between Perplexity and Type-Token Ratio (TTR) for various languages, categorized by their morphological types.

1. Correlation: There is a clear positive correlation between TTR and perplexity. As TTR increases, indicating higher morphological complexity, perplexity tends to rise, suggesting decreased model performance.

2. Language Distribution:

- Isolating languages (e.g., Vietnamese) exhibit very low perplexity and TTR, indicating better model performance.
- Fusional languages (e.g., English) show moderate perplexity and TTR.
- Agglutinative and introflexive languages (e.g., Korean) demonstrate high perplexity and TTR, presenting significant challenges for language models.

3. Morphological Impact:

- Isolating languages, with distinct and repetitive words and generally fixed word order, are more easily captured by language models.
- Fusional languages present increased difficulty due to their morphology yielding rarer words and potentially more flexible word order.
- Introflexive languages, with complex internal morphology, are predicted with even greater difficulty.
- Agglutinative languages, characterized by numerous affixes around noun and verb stems, appear to be the most challenging for language models.

Question 5 Why will we not consider the corpus used in 1? Why are the FAIR principles a solution to this problem?

The datasets that used in the Daniela Gerz et al 's research are not widely applicable to other tasks. Their data is prepocessed by polygot tokenizer, which may not be suitable for all application, and their datasets

were custom-created for their study so it may lack the standardization of more established benchmarks. FAIR principles requires data should be easily Findable Accessible Interoperable Reusable, which is not really the case for the dataset they used.

Part 2

Question 6 What is the role of the data_dir parameter in line 4 of Figure 2? What is the meaning of its value?

data_dir parameter means where the dataset is stored, `"gs://tfds-data/datasets"` is redirected to the Google Cloud to fetch the desired datasets, wiki40bfr, for example.

Question 7 Why do we use the islice function in line 6 of Figure 2?

First, ds.shuffle(buffer_size=10_000) is a transformation designed to handle extremely large datasets that are too big to fit entirely in memory.

Instead of shuffling the entire dataset at once, it maintains a buffer of size 10,000. This operation returns an ***iterable*** dataset.

Then, we use the ***islice*** method from itertools. We use ***islice*** to fetch the first 100 data points from the shuffled dataset. It's important to note that these 100 elements are not taken from a fixed shard of 10,000 data points, but rather from the entire dataset as it's being dynamically shuffled.

The shuffle operation with a buffer creates a moving window of randomization across the entire dataset, and ***islice*** then selects the first 100 elements

from this randomized stream. This approach allows for efficient randomization and sampling from very large datasets without requiring the entire dataset to be loaded into memory at once.

Question 8 Why do we consider the test set (see the split parameter)? Is this a good idea?

Generally Speaking, it is not a good idea if we involve the test data in our workflow, expect using it in evaluation & cross validtion. For most purpose, it should be better if we sample from train split.

1. Test data is reserved for final model evaluation after training phase finished, which is supposed to provided an unbiased evaluation. 2. There are some potential concerns for this: if the test data is used in any part of model development, it might risk introducing the bias into our final evaluation, thus making it easier to overfit.

Question 9 Why do we have to specify a buffer_size parameter for the shuffle method? How do we choose its value?

If buffer_size is not specified, TensorFlow defaults to using the entire dataset size. This ensures maximum randomness but may cause memory overload for large datasets, potentially leading to system crashes or performance issues.

Choosing an appropriate buffer_size involves balancing effectiveness and resource limitations. Ideally, it should be set as large as possible to maximize data shuffling randomness. However, memory constraints must be carefully considered to avoid system overload.

The nature of the data itself also influences this decision. If the dataset is already sufficiently random, a large buffer_size may not be necessary. In such cases, a smaller buffer can be used without significantly impacting data randomness.

We specify the buffer_size parameter for several reasons: it ensures memory efficiency for large datasets, enables stream processing for extensive data, and provides flexibility in adjusting randomization based on specific requirements and resource constraints.
When choosing the buffer_size value, consider factors such as dataset size, available memory, randomness requirements, and performance needs. Smaller values may improve processing speed but could reduce randomness. Experimental tuning is often required to find the optimal balance.
Essentially, the choice of buffer_size is a trade-off between achieving optimal randomness and effectively managing computational resources. It often requires finding the best balance for your specific use case and system capabilities.

Question 10 Why do we have to extract sentences out of Wikipedia articles?

Extracting sentences from Wikipedia articles provides a rich and diverse dataset for training. Sentences represent natural, coherent units of language that capture grammatical structures, contextual relationships, and semantic meanings. By using individual sentences, we can create a large number of training examples from a relatively smaller set of articles, enhancing the model's ability to learn varied language patterns and improve its generalization capabilities. This approach also offers practical benefits for model training. Working with sentences allows for more efficient processing and memory management compared to handling entire articles at once. It enables easier batch processing and can help in creating more balanced training sets. Additionally, sentence-level data is often easier to tokenize and preprocess, which is crucial for preparing input for language models. This method also naturally aligns with many language modeling objectives, such as predicting the next word in a sentence or understanding sentence-level context.

Question 11 Why do we enforce that all train and test sets have the same size?

When evaluating a model's generalization ability, it's common practice to allocate data between training and testing sets in a 6:4 ratio. However, enforcing equal sizes for training and testing sets (i.e., a 5:5 ratio) offers distinct advantages. This approach increases the test set size, providing more diverse cases to assess the model's performance on unseen data. Furthermore, maintaining this equal proportion across different languages ensures consistency in the evaluation process. It guarantees that the model is trained on an equivalent volume of data for each language and subsequently tested on an equal amount of unseen data. This standardization allows for more reliable cross-linguistic comparisons of the model's performance and generalization capabilities.

Question 12 What kind of information polyglot is using to identify sentence boundaries and segment sentences into tokens. Comment.

In polyglot lib, Tokenization is the process that identifies the text boundaries of words and sentences. It relies on the Unicode Text Segmentation algorithm as implemented by the ICU Project

```
# Tokenization Task
from polyglot.text import Text
text_en = u"I need an Internship. I'm tired"
text_zh = u"我需要一份实习!累拥了"
text_fr = u"J'ai besoins d'un stage!!Je suis fatigué!"
tokens_en, tokens_zh, tokens_fr = Text(text_en).tokens, Text(
    text_zh).tokens, Text(text_fr).tokens
sentences_en, sentences_zh, sentences_fr = Text(text_en).
    sentences, Text(text_zh).sentences, Text(text_fr).sentences
print(f"tokens_en: {tokens_en}, tokens_zh: {tokens_zh},
    tokens_fr: {tokens_fr}")
print(f"sentences_en: {sentences_en}, sentences_zh: {
    sentences_zh}, sentences_fr: {sentences_fr}")
tokens_en: ['I', 'need', 'an', 'Internship', '.', "I'm", 'tired
```

```
    '], tokens_zh: ['我', '需要', '一份', '实习', '!', '累', '
    拥', '了'], tokens_fr: ["J'ai", 'besoins', "d'un", 'stage',
     '!', '!', 'Je', 'suis', 'fatigué', '!']
sentences_en: [Sentence("I need an Internship."), Sentence("I'm
     tired")], sentences_zh: [Sentence("我需要一份实习!"),
    Sentence("累拥了")], sentences_fr: [Sentence("J'ai besoins
    d'un stage!!"), Sentence("Je suis fatigué!")]
```

Question 13 For each languages of the corpus extract a train set of 40,000 sentences and a test set of 3000 sentences. Sentences must be unique(i.e if there are several occurrences of an identical sentence, they must be removed but one

.

```
from collections import deque
def filter_paragraph(text_list):
    return [item for item in text_list if item != "
        _START_PARAGRAPH_" and text_list[text_list.index(item)
        - 1] == "_START_PARAGRAPH_"]
def get_limited_sentences(sentences_list:list, num_sentences:
    int = 43000) -> list[str]:
    sentences_splited = [i.split("\n") for i in sentences_list
        if i]
    buffer = [sentence for sublist in sentences_splited for
        sentence in filter_paragraph(sublist)]

    sentences_return = set()
    for paragraph in buffer:
        sentences_return.update(str(sent) for sent in Text(
            paragraph).sentences)
    sentences_return = deque(sentences_return, maxlen=
        num_sentences)

    if len(sentences_return) < num_sentences:
        print(f"Currently, it only stores {len(sentences_return
            )}. Please consider changing to a bigger dataset.")
```

```
    return list(sentences_return)
# lang_list
languages = [
    'en', 'ar', 'zh-cn', 'zh-tw', 'nl', 'fr', 'de', 'it', 'ja',
        'ko', 'pl', 'pt', 'ru', 'es',
    'th', 'tr', 'bg', 'ca', 'cs', 'da', 'el', 'et', 'fa', 'fi',
        'he', 'hi', 'hr', 'hu', 'id',
    'lt', 'lv', 'ms', 'no', 'ro', 'sk', 'sl', 'sr', 'sv', 'tl',
        'uk', 'vi'
]
# save the data of each lang into a dict
from time import time
from collections import deque
time_start = time()
dict_lang = {}
for lang in languages:
  ds = tfds.load("wiki40b/"+lang,split="train",data_dir = "gs
      ://tfds-data/datasets")
  sample = [ex['text'].numpy().decode('utf-8') for ex in islice
      (ds.shuffle(buffer_size=100_000),10_000)] # regardless of
       original size of ds, this code will generate 100 decoded
       samples
  dict_lang[lang] = get_limited_sentences(sample,43000)
  print(f"{lang} has been processed, sentences now stored in
      dict")
  time_end_lang = time()
  total_time = time_end_lang - time_start
  print(f"total time: {total_time}")

# split 40000 / 3000 sentences train / test
import random
random.seed(42)
for lang in dict_lang:
  random.shuffle(dict_lang[lang])
  dict_lang[lang] = {
      "total_data":dict_lang[lang],
      "train":dict_lang[lang][:40000],
      "test":dict_lang[lang][40000:]
  }
```

Question 14 Why do we have to remove duplicate sentences?

If duplicate sentences appear frequently in the training data, then the model may assign them disproportionate weight, which may lead to overfitting on specific items, and by removing them, we also reduce bias in the datasets. Ensuring sentences are not duplicate can also enhance the data's diversity, so the model is exposed to more pattern and structures.

Question 15 Using polyglot tokenize all datasets into words. Save each dataset into a separate text file (one sentence per line) using consistent names to be able to automate the LM estimation (see Section 2.3).

```
for lang in languages:
  dict_lang[lang]['tokens'] = [list(Text(i).tokens) for i in
      dict_lang[lang]['total_data']]
  print(f"{lang} has been processed, tokens now stored in dict"
      )
  with open(f"{lang}_tokens.txt","w",encoding='utf-8') as f:
    for sentence in dict_lang[lang]['tokens']:
      f.write(" ".join(sentence) + "\n")
  print(f"Tokens for {lang} have been written to {lang}_tokens.
      txt")
```

Part 2.2 Extracting morpholofical information

Question 16 Compute the TTR for all datasets

```
for i in pathlib.Path("/content/").rglob("*.txt"):
  file_name = i.stem
  lang_code = file_name.split("_")[0]
  data = ''
  print(lang_code)
  with open(i,'r',encoding='utf-8') as f:
    data = f.read()
```

```
    data_processed = re.split(r"\s+|\n",data)
  TTR = len(set(data_processed))/len(data_processed)
  dict_lang[lang_code]['TTR'] = TTR
  print(TTR)
```

Question 17 Knowing that you will have to compute for each language its TTR and its perplexity, what is the best way to store the TTR?

Dictionary, this structure will allow you to easily add more information if needed.

Question 18 Classify each language of the wiki40b corpus into one of the following four categories to characterize its morphology: isolating, fusional, introflexive and agglutinative. You can/should use a typological database such as the WALS.4

```
isolating = ['zh-cn', 'zh-tw', 'th', 'vi', 'id', 'ms']
fusional = [
    'en', 'nl', 'fr', 'de', 'it', 'pl', 'pt', 'ru', 'es',
    'bg', 'ca', 'cs', 'da', 'el', 'lt', 'lv', 'no', 'ro',
    'sk', 'sl', 'sr', 'sv', 'uk', 'hr', 'hi'
]
introflexive = ['ar', 'he', 'fa']
agglutinative = [
    'ja', 'ko', 'tr', 'et', 'fi', 'hu', 'tl'
]
```

Question 19 What structure should you use to store this information? Why?

To store this information, it would be beneficial to use both a DataFrame and extend the existing dictionary structure. This information will be further needed for plot or to show its TTR & perplexity scores, it is better if

stored in Dataframe. At the same time, It is also good if we saved them as one extra keys in the existing dictionary, for easy access and reuse.

2.3 Training and Evaluating language models

Question 20 Install kenlm using the instructions provided in Fig3

Question 21 Why do some shell commands in Fig3 starts with a ! and others with a %?

Since we are executing these commands within JupyterLab or Google Colab:

- The ! symbol is used to execute shell commands directly from a code cell. It tells the notebook environment to run the rest of the line as a shell command in the system's command line interface.
- The % symbol denotes IPython magic commands. In this case, %cd is a special command to change the current working directory within the notebook environment.

If we are working on normal shell like bash, we need to omit the ! and %

Question 22.Why the python interface of kenlm allows you to compute the perplexity of a model but not to estimate its parameters?

The Python interface of KenLM allows computation of perplexity but not parameter estimation due to its design philosophy and performance considerations. KenLM is primarily a C++ based language model toolkit, which offers significant speed advantages through multi-threaded processing. The Python interface serves as a wrapper around this C++ core, focusing on providing convenient access to pre-trained models rather than model training capabilities. This design choice reflects the typical workflow where model

training, a computationally intensive task, is performed infrequently using the optimized C++ implementation. In contrast, model usage tasks like perplexity computation are more common and benefit from the ease of integration that Python provides.

Question 23 Using a for-loop, train a language model for each languages

First, it loops over all the lang_token.txt, then pass each of them one by one into training, each language model will be saved under one specific folder

```
!for file in /content/*_tokens.txt; do basename=$(basename "
    $file" _tokens.txt); /content/kenlm/build/bin/lmplz -o 5 --
    verbose_header --text "$file" --arpa "/content/kenlm_models
    /${basename}_lm.arpa"; done
```

Question 24 Estimate for each language, the perplexity of the LM and store it into a well-chosen structure.

```
def calculate_perplexity(model_folder:str, data_folder:str,
    language:list,dict_lang:dict):
  model_folder = Path(model_folder)
  data_folder = Path(data_folder)
  perplexity_dict = {}
  for lang in language:
    model_path =model_folder/f"{lang}_lm.arpa"
    model = kenlm.Model(str(model_path))
    data_length = len(dict_lang[lang]['tokens'])
    total_perplexity = 0
    for i in dict_lang[lang]['tokens']:
      perplexity = model.perplexity(" ".join(i))
      total_perplexity += perplexity
    average_perplexity = total_perplexity/data_length
    dict_lang[lang]['perplexity'] = average_perplexity
    perplexity_dict[lang] = average_perplexity
    print(f"{lang} average perplexity: {average_perplexity}")
  return perplexity_dict
```

```
perplexity = calculate_perplexity("/content/kenlm_models","/
    content",languages,dict_lang)
```

2.4 Desert

Question 25 Plot the results of the previous two sections to reproduce the result reported in Figure1

```
selected_keys = ["perplexity","TTR","label"]
df = pd.DataFrame({lang:{col:dict_lang[lang][col]for col in
    selected_keys}for lang in dict_lang}).T
df = df.reset_index().rename(columns={"index":"language"})
unique_labels = df['label'].unique()
colors = plt.cm.rainbow(np.linspace(0, 1, len(unique_labels)))
label_color_map = dict(zip(unique_labels, colors))
plt.figure(figsize=(12, 8))
for label in unique_labels:
    mask = df['label'] == label
    plt.scatter(df.loc[mask, 'TTR'], df.loc[mask, 'perplexity'
        ],
                c=[label_color_map[label]], label=label, alpha
                    =0.7)
for i, row in df.iterrows():
    plt.annotate(row['language'], (row['TTR'], row['perplexity'
        ]),
                 xytext=(5, 5), textcoords='offset points')
plt.xlabel('TTR')
plt.ylabel('Perplexity')
plt.title('TTR vs Perplexity across Languages')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

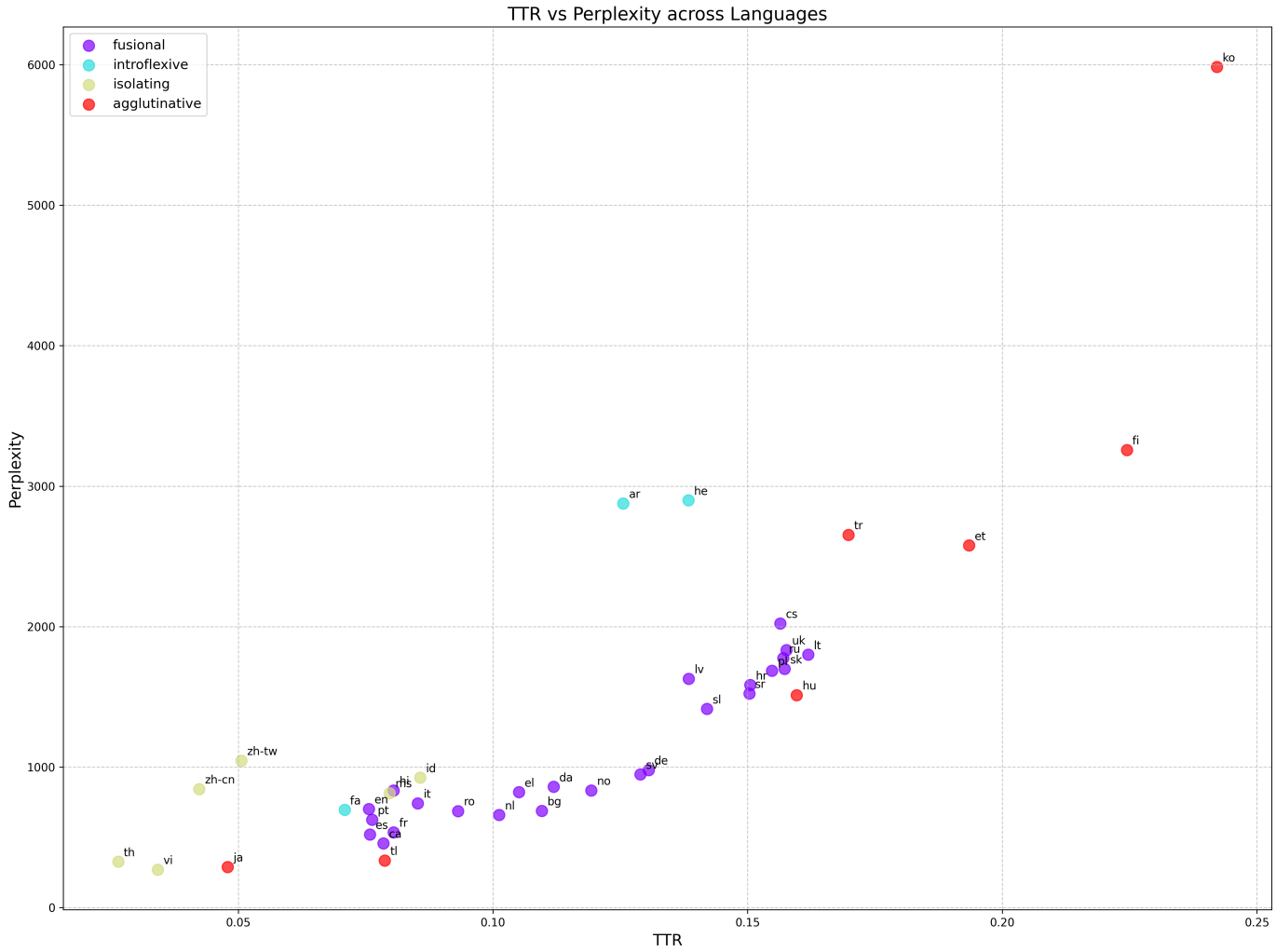

图 2: TTR vs Perplexity across Languages

Question 26 Install Sentencepiece and tokenize all datasets using a vocabulary size of 32000 tokens

When training SentencePiece models for our datasets using a vocabulary size of 32000 tokens, I encountered errors for certain languages that didn't have enough unique tokens to support this size. This occurs because SentencePiece cannot create a model with more tokens than exist in the input data. To address this issue, I manually set three different vocabulary sizes: 32000, 21757, and 10000. These sizes were determined through trial and error and successfully accommodated all the languages we were processing.

```
!mkdir -p /content/sentencepiece_models
model_folder = Path("/content/sentencepiece_models")
data_folder = Path("/content")
for lang in languages:
  data_path =data_folder/f"{lang}_tokens.txt"
  model_path = model_folder/f"{lang}_sp.model"
    try:
    spm.SentencePieceTrainer.train(f"--input={data_path} --
        model_prefix={model_path} --vocab_size=32000")
  except:
    print(f"error in {lang}, saying it should be smaller than
        21757")
    try:
      spm.SentencePieceTrainer.train(f"--input={data_path} --
          model_prefix={model_path} --vocab_size=21757")
    except:
      print(f"error in {lang}, even smaller, i just set it to
          10000")
      spm.SentencePieceTrainer.train(f"--input={data_path} --
          model_prefix={model_path} --vocab_size=10000")
      sp = spm.SentencePieceProcessor()
  sp.load(str(model_path))
  data = []
  with open(data_path,'r',encoding='utf-8') as f:
    data = f.read()
  tok = sp.encode_as_pieces(data)
  TTR = len(set(tok))/len(tok)
  dict_lang[lang]['TTR_sp'] = TTR
```

```
  dict_lang[lang]['Tokens_sp'] = tok
```

Question 27 Train a language model on these new datasets and compute the new perplexity. What can you conclude?

```
!mkdir -p /content/sentencepiece_tokenized_txt
data_folder = Path("/content/sentencepiece_tokenized_txt")
for lang in languages:
  with open(data_folder/f"{lang}_sp.txt",'w',encoding='utf-8')
      as f:
    for sentence in dict_lang[lang]['Tokens_sp_per_line']:
      f.write(" ".join(sentence) + "\n")
!mkdir -p /content/kenlm_sentencepiece_models
!for file in /content/sentencepiece_tokenized_txt/*_sp.txt; do
    basename=$(basename "$file" _tokens.txt); /content/kenlm/
    build/bin/lmplz -o 5 --verbose_header --text "$file" --arpa
     "/content/kenlm_sentencepiece_models/${basename}_lm.arpa";
     done
for lang in languages:
  perplexity = 0
  with open(f"/content/sentencepiece_tokenized_txt/{lang}_sp.
      txt",'r',encoding='utf-8') as f:
    lines = f.read().splitlines()
    model = kenlm.Model(f"/content/kenlm_sentencepiece_models/{
        lang}_sp.txt_lm.arpa")
    for i, sent in enumerate(lines):
      perplexity += model.perplexity(" ".join(sent))
    dict_lang[lang]['perplexity_sp'] = perplexity/len(lines)
```

The second plot reveals significant differences from the first, showcasing a less linear relationship between TTR and perplexity, with fewer outliers. This shift underscores the crucial impact of token quality and tokenizer selection in language model training. Despite these changes, some consistent patterns persist, notably the clustering of many European languages in the lower right quadrant, suggesting similar linguistic properties in terms of TTR and perplexity across both fusional and introflexive language families.

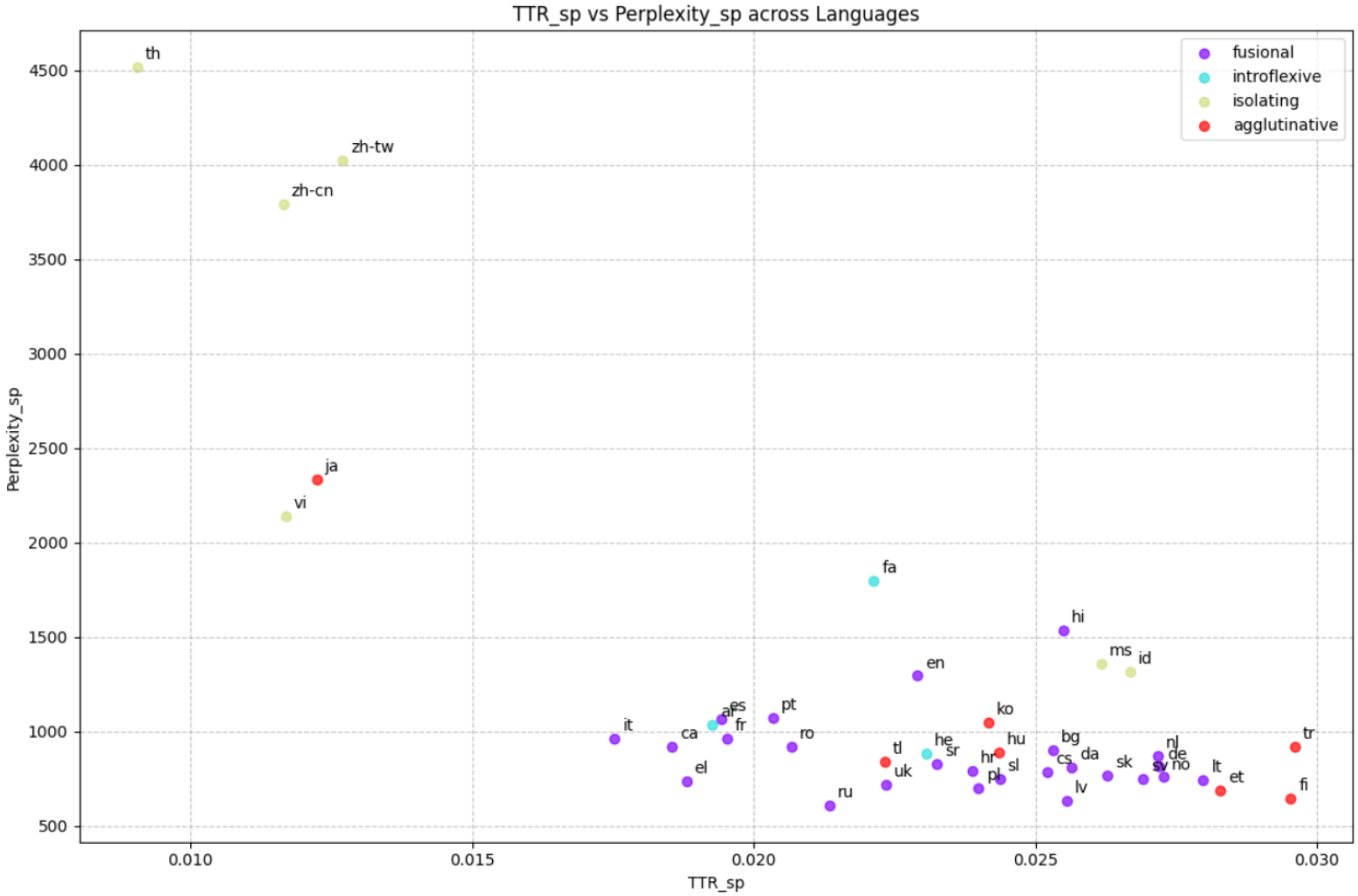

图 3: TTR_sp vs Perplexity_sp

Interestingly, languages such as Thai and Chinese maintain their position as outliers, characterized by lower TTR and higher perplexity. This consistency across both plots reinforces the notion that these languages possess more complex structures or larger vocabularies, potentially posing greater challenges for language models. The persistence of these patterns, despite the differences in tokenization approaches, highlights fundamental linguistic characteristics that transcend specific preprocessing techniques, offering valuable insights for natural language processing tasks and model development across diverse language families.

References

[1] Ameeta Agrawal et al. "On the Role of Corpus Ordering in Language Modeling". In: *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*. Ed. by Nafise Sadat Moosavi et al. Virtual: Association for Computational Linguistics, Nov. 2021, pp. 142–154. DOI: 10.18653/v1/2021.sustainlp-1.15. URL: https://aclanthology.org/2021.sustainlp-1.15.

[2] Kimmo Kettunen. "Can Type-Token Ratio be Used to Show Morphological Complexity of Languages?" In: *Journal of Quantitative Linguistics* 21 (June 2014), pp. 223–245. DOI: 10.1080/09296174.2014.911506.