Lab 1: One model to rule them all? Multilingual NLP

Manon Fleury
22410952

October 9, 2024

1 Appetizers: What are we looking for?

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?

The type-token ratio (TTR) is a measurement of lexical diversity. It is defined as the ratio of unique tokens divided by the total number of tokens within a written text or a person's speech. If the text has no repetition, the TTR is 1, and if it has infinite repetition, it will tend to 0.
A high TTR indicates a large amount of lexical variation, namely a rich morphology, and a low TTR indicates relatively little lexical variation. Thus, TTR is a 'good' measure of the morphological complexity.
Another metric to quantify morphological complexity could be to measure word lengths as languages with richer morphology would have longer words due to inflections and affixes.

2. Why do we consider a language modeling task?

We consider a language modeling task because we're evaluating models, and we're doing that across languages, so we need a consistent task across languages to measure performance.

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

Perplexity measures the quality of a language model by measuring how well a model predicts the next token in a sentence. In short, it measures how confused the model is for predicting the next word. The lower the perplexity, the more confident the model is at predicting words in the given context, it actually assigns high probabilities to the correct words.
We can't directly compare perplexity across corpora and across languages:
Across corpora, they differ in size and subject matter. It can be very easy to predict the next words in a very logical text, like in a recipe, while in a fantastic novel for example there can be incongruous events that are difficult to predict, leading to a higher perplexity.
Across languages, they vary in morphological richness, so some languages can have many word forms, namely a larger vocabulary, leading to higher perplexity because the model has more possible words to predict, on the contrary of morphologically simpler languages with smaller vocabulary.
To conclude, perplexity can't be compared directly across corpora or languages, meaning the same model for all languages will not have the same performance across languages. However, perplexity comparisons can make more sense if the vocabulary size is normalized, if the corpora are of similar size and same domain, and if we group languages by morphological similarity.

4. What conclusions can you draw from Figure 1.

We can conclude from Figure 1 that the performance is not the same, it varies with language. This is the main conclusion, but we can also notice a linear dependency with the complexity of language and performance, and also see different families of language.

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

The study [1] may have focused on a smaller number of languages, particularly those that are already well-represented in NLP resources. However, our goal is to analyze the performance of language models across a diverse set of languages (including resource-poor languages).
Also, the corpus might not be publicly available or have restrictions, so we would not be able to replicate their study using the same dataset.

The FAIR principles (Findable, Accessible, Interoperable and Reusable) ensure that data are accessible, shareable and reusable in the long term.
Data must be easily findable with proper indexing, accessible (open access licenses), interoperable (compatible with other datasets or systems) and reusable (well-documented data, to promote transparency).

2 Main Course

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

The data_dir parameter indicates where to search and load the dataset. Here the data are loaded from internet, with 'gs' that is short for "Google Cloud Storage", indicating the dataset is stored and accessible from a cloud storage bucket rather than from a local directory; in there, tfds-data is a bucket where the datasets are stored.

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

The islice function provides an iterator that slices an iterable given as argument (the dataset here). It takes a specific number of elements from the dataset without creating an intermediate list. Here, it slices the shuffled dataset and only takes the first 100 examples from it. Its role is to limit the selection of items retrieved from the dataset.

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

When looking at the size of the train/validation/test sets of the wiki40b dataset (on TensorFlow.org), we notice that the test set is much smaller than the train set (160 000 vs 2 900 000) and since we are only going to extract 40 000 + 3 000 sentences, the test set is sufficient enough for this amount, we don't need to load all the train set to use only a tiny amount.

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

ds.shuffle(buffer_size=10_000) allows to randomly select items from the first 10 000 elements, it randomizes the order of examples. Shuffling an entire dataset that is very large would be very heavy so buffer_size allows to shuffle only a subset (a buffer) of the dataset. The larger the buffer_size, the more thoroughly the dataset is shuffled. The choice of the value of buffer_size depends on the size of the dataset, the available memory and how much randomness we aim at. So 10 000 provides a decent shuffle without overwhelming the system.

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

Wikipedia articles are a rich source of raw data, but to exploit it we have to extract the sentences, namely to break down the text into smaller units that are more suitable for model training. It ensures the data is clean, standardized and suitable for language modeling.

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

All train and test sets must have the same size because we want a fair and controlled comparison of model performance. Indeed, providing more training data in one language can improve the performance and more testing data can provide a more reliable estimate of model performance.

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

Polyglot support the Unicode Text Segmentation algorithm that relies on Unicode character properties to determine whether a character is a letter, punctuation mark, whitespace, number or other symbol. This algorithm detects punctuation marks as indicators of sentence boundaries. But for a dot that means an abbreviation for example, Polyglot uses context to make a decision, so it uses linguistic rules. However, Unicode may face challenges with complex sentence structures or ambiguous punctuation.

13. For each languages of the corpus extract a train set of 40,000 sentences and a test set of 3,000 sentences. Sentences must be unique (i.e. if there are several occurrences of an identical sentence, they must all be removed but one).

(See question 15. where the code answers both questions)

14. Why do we have to remove duplicate sentences?

Removing duplicate sentences ensures that the model is trained and evaluated on a diverse and representative dataset, leading to more reliable and meaningful performance outcomes across different languages.

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).

```
import tensorflow_datasets as tfds
from sklearn.model_selection import train_test_split
import nltk
from polyglot.text import Text
from itertools import islice

# extract the languages
languages = [config.name for config in tfds.builder("wiki40b").
    builder_configs.values()]

for language in languages:
  ds = tfds.load(f"wiki40b/{language}", split="test", data_dir="gs://tfds-
      data/datasets")
  sample = [ex["text"].numpy().decode("utf-8") for ex in islice(ds.shuffle
      (buffer_size=20_000), 3_000)]

  # Remove duplicates
```

```
  sample = list(set(sample))

  # Split into train and test sets
  train_sample, test_sample = train_test_split(sample, test_size=0.085,
     random_state=42)

  train_sample = ' '.join(train_sample)
  test_sample = ' '.join(test_sample)

  train_sample = Text(train_sample, hint_language_code = language)
  test_sample = Text(test_sample, hint_language_code = language)

  with open(f'{language}_train_tokenized.txt', 'w') as f:
    for sentence in train_sample.sentences:
      # Tokenize the sentence into words
      if sentence:
        words = sentence.words
        f.write(str(words) + ' ')
      f.write('\n')

  # same for the test set
  with open(f'{language}_test_tokenized.txt', 'w') as f:
    for sentence in test_sample.sentences:
      if sentence:
        words = sentence.words
        f.write(str(words) + ' ')
      f.write('\n')
```

16. Compute the TTR for all datasets.

```
import json
from collections import Counter

# function to compute the TTR
def compute_ttr(words):
  types = len(set(words))  # nb of unique words
  tokens = len(words)      # total nb of words
  return types / tokens if tokens > 0 else 0

# to store the TTRs
ttr_results = {}

# compute the TTR for each dataset
for language in languages:
  for split in ['train', 'test']:
    tokenized_file = f'{language}_{split}_tokenized.txt'

    with open(tokenized_file, 'r') as f:
      words = []
      for line in f:
        words_in_line = line.strip().split()  # extract the words of the
           line
        words.extend(words_in_line)
```

```
    # compute the TTR
    ttr = compute_ttr(words)

    # save the TTR
    if language not in ttr_results:
      ttr_results[language] = {}
    ttr_results[language][split] = ttr

# save the results
with open('ttr_results.json', 'w') as f:
  json.dump(ttr_results, f, indent=4)
```

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?

We can store the TTR results in a Json file.

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

```
language_types = {
    'zh-cn': 'Isolating', 'zh-tw': 'Isolating', 'vi': 'Isolating', 'th': '
       Isolating', 'id': 'Isolating',
    'en': 'Fusional', 'ar': 'Introflexive', 'nl': 'Fusional', 'fr': '
       Fusional', 'de': 'Fusional', 'it': 'Fusional',
    'pl': 'Fusional', 'pt': 'Fusional', 'ru': 'Fusional', 'es': 'Fusional'
       , 'bg': 'Fusional', 'ca': 'Fusional',
    'cs': 'Fusional', 'da': 'Fusional', 'el': 'Fusional', 'fa': 'Fusional'
       , 'he': 'Introflexive', 'hi': 'Fusional',
    'hr': 'Fusional', 'hu': 'Fusional', 'lt': 'Fusional', 'lv': 'Fusional'
       , 'no': 'Fusional', 'ro': 'Fusional',
    'sk': 'Fusional', 'sl': 'Fusional', 'sr': 'Fusional', 'sv': 'Fusional'
       , 'uk': 'Fusional', 'ja': 'Agglutinative',
    'ko': 'Agglutinative', 'tr': 'Agglutinative', 'et': 'Agglutinative', '
       fi': 'Agglutinative', 'tl': 'Agglutinative',
    'ms': 'Agglutinative'
}
```

19. What structure should you use to store this information?

We can store it in a dictionary (as above) or in a dataframe (as the code below).

20. Install kenlm using the instructions provided in Figure 3.

```
!git clone https://github.com/kpu/kenlm.git
%cd kenlm
!python setup.py develop
!mkdir -p build
%cd build
```

```
!cmake ..
!make -j 4

# + this line of code to make the following code work
!pip install https://github.com/kpu/kenlm/archive/master.zip
```

21. Why do some shell commands in Figure 3 starts with a ! and others with a %?

! is to run regular shell commands. It tells the notebook to execute the following command in a shell (like in a terminal). And % is a magic command, used to enhance functionality, for example to change the working directory.

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 is primarly designed to query language models that have already been trained but it doesn't support training or estimating the model parameters. It is because kenlm is implemented in C++. The Python interafe is limited to querying tasks because they are lighter and faster, and don't require the heavy computational power of C++.

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

```
import subprocess

for language in languages:
  input_file = f'{language}_train_tokenized.txt'
  output_file = f'{language}_train.arpa'

  cmd = f"./bin/lmplz -o 5 < {input_file} > {output_file}"

  subprocess.run(cmd, shell=True, check=True)

  print(f'Language model for {language} has been saved to {output_file}')
```

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

```
import kenlm

dico_ppl = {}
for language in languages:
  m = kenlm.Model(f'{language}_train.arpa')
  L_ppl = [] # to store the perplexities for a language
  with open(f'{language}_train_tokenized.txt', 'r') as f:
    for sentence in f: # one line = one sentence
      sentence = sentence.strip() # Remove spaces at the beginning and end
          of each sentence
      if sentence: # ignore the empty lines
        ppl = m.perplexity(sentence)
        L_ppl.append(ppl)

```

```
  # compute the average of the perplexities for this language
  avg_perplexity = sum(L_ppl) / len(L_ppl) if L_ppl else float('inf')

  # store the average ppl in the dictionary
  dico_ppl[language] = avg_perplexity

print(dico_ppl)
```

Output = {'en': 29.05156204000618, 'ar': 36.8745336959087, 'zh-cn': 19.88877410787131, 'zh-tw': 24.348876469401972, 'nl': 38.945491515193694, 'fr': 27.526078083933026, 'de': 30.071701266684716, 'it': 29.481774320371635, 'ja': 19.94985426583456, 'ko': 66.2587912241133, 'pl': 32.56052008784197, 'pt': 24.64313129549713, 'ru': 31.849623706624467, 'es': 14.943390108611126, 'th': 17.959560445375566, 'tr': 37.24675375006474, 'bg': 29.623085628892827, 'ca': 27.81961970842046, 'cs': 39.46300780654927, 'da': 47.75560707248816, 'el': 32.16079365726815, 'et': 54.40587893035072, 'fa': 37.921421478713825, 'fi': 57.27230762219702, 'he': 35.47104896919734, 'hi': 29.753955989929366, 'hr': 36.0215952746041, 'hu': 35.06999908133554, 'id': 27.557403788940558, 'lt': 41.15669292495179, 'lv': 33.52513612218895, 'ms': 22.70757823195075, 'no': 46.366246045735366, 'ro': 29.75384055594458, 'sk': 44.7814447799377, 'sl': 39.7725833539788, 'sr': 30.193224565452795, 'sv': 43.30117497663664, 'tl': 26.613296914791164, 'uk': 37.66573075569507, 'vi': 16.262597176850612}

25. Plot the results of the previous two sections to reproduce the result reported in Figure 1.

```
import json
import matplotlib.pyplot as plt
import seaborn as sns

# Load the results of TTR
with open('ttr_results.json', 'r') as f:
  ttr_results = json.load(f)

# to store the ttr and ppl
ttr_perplexity = {lang: {"ttr": 0, "ppl": 0} for lang in ttr_results.keys
    ()}

# Extract the ttr
for language, splits in ttr_results.items():
  ttr_perplexity[language]["ttr"] = (splits['train'] + splits['test']) / 2
       # average of the ttr

# Add the perplexities to the dictionary
for language, perplexity in dico_ppl.items():
  ttr_perplexity[language]["ppl"] = perplexity

# Create a df for the plot
data = {
    "Language": [],
    "TTR": [],
    "Perplexity": [],
    "Family": []
}

# Fill the data
for lang, values in ttr_perplexity.items():
  data["Language"].append(lang)
```

```
    data["TTR"].append(values["ttr"])
    data["Perplexity"].append(values["ppl"])
    data["Family"].append(language_types.get(lang, "Unknown"))

# Convert in df
import pandas as pd
df = pd.DataFrame(data)

# Plot
plt.figure(figsize=(12, 8))
palette = {'Isolating': 'blue', 'Fusional': 'orange', 'Agglutinative': '
    green', 'Introflexive': 'red'}

# Scatter plot
sns.scatterplot(data=df, x='TTR', y='Perplexity', hue='Family', palette=
    palette, s=100)

# Labels for each language
for i in range(df.shape[0]):
    plt.text(df['TTR'][i], df['Perplexity'][i], df['Language'][i], fontsize
        =9, ha='right')

plt.title('Perplexity vs TTR by Language')
plt.xlabel('Type-Token Ratio (TTR)')
plt.ylabel('Perplexity')
plt.legend(title='Language Family')
plt.grid()
plt.tight_layout()
plt.savefig('perplexity_vs_ttr.png')  # save the graph
plt.show()
```

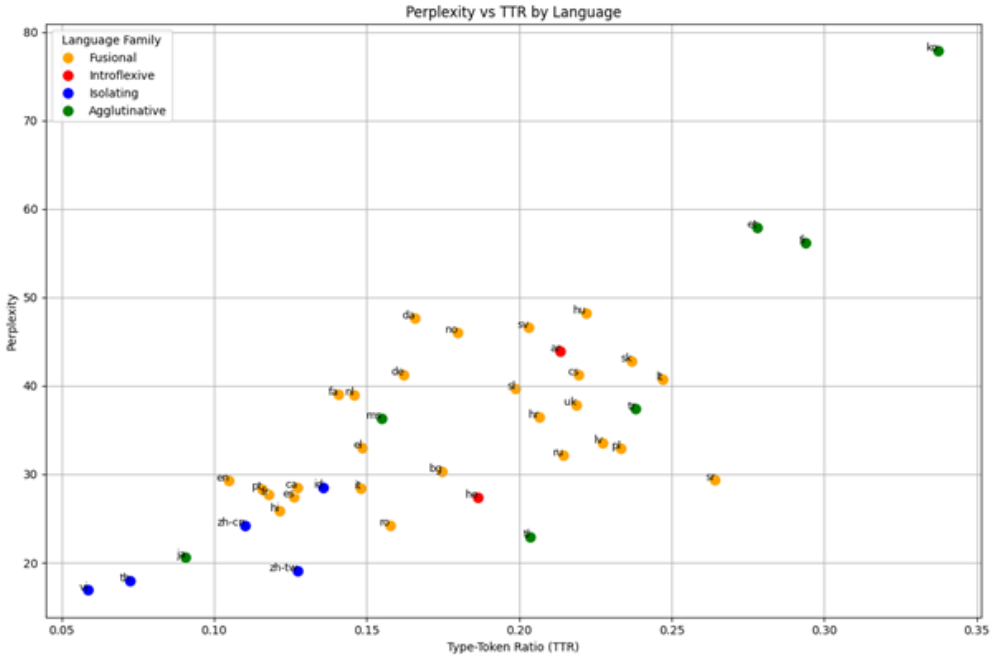

Figure 1: Perplexity of a language model in relation to the TTR for different kind of language and morphological systems.

26. Install sentencepiece and tokenize all datasets using a vocabulary size of 32,000 tokens.

```
!pip install sentencepiece

import sentencepiece as spm

vocab_size = 13000 # because error when above 14 000

# Train a sentencepiece model for each language
for language in languages:
  input_file = f'{language}_train_tokenized.txt'

  # Train the sentencepiece model
  spm.SentencePieceTrainer.Train(f'--input={input_file} --model_prefix={
     language}_spm --vocab_size={vocab_size} --character_coverage=1.0')

# Use the model to tokenize the datasets
for language in languages:
  sp = spm.SentencePieceProcessor(model_file=f'{language}_spm.model')

  # Tokenize the train set
  with open(f'{language}_train_tokenized.txt', 'r') as f:
    train_sentences = f.readlines()

  train_tokenized = [sp.EncodeAsPieces(sentence.strip()) for sentence in
     train_sentences]
  with open(f'{language}_train_sentencepiece.txt', 'w') as f:
    for sentence in train_tokenized:
```

```
      f.write(' '.join(sentence) + '\n')

  # Tokenise the test set
  with open(f'{language}_test_tokenized.txt', 'r') as f:
    test_sentences = f.readlines()

  test_tokenized = [sp.EncodeAsPieces(sentence.strip()) for sentence in
     test_sentences]
  with open(f'{language}_test_sentencepiece.txt', 'w') as f:
    for sentence in test_tokenized:
      f.write(' '.join(sentence) + '\n')
```

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

```
import subprocess
import kenlm

dico_ppl_bpe = {}

# Train language models for each language using the SentencePiece
   tokenized files
for language in languages:
    # Use the file tokenized by SentencePiece
    input_file = f'{language}_train_sentencepiece.txt'
    output_file = f'{language}_train_bpe.arpa'

    # Train the language model
    cmd = f"./bin/lmplz -o 5 < {input_file} > {output_file}"

    # Execute the command to train the model
    try:
        subprocess.run(cmd, shell=True, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Error while training language model for {language}: {e}")
        continue  # Go to next line if error

    model = kenlm.Model(output_file)
    L_ppl = []

    # Add ppl to the list L_ppl
    with open(input_file, 'r') as f:
        for sentence in f:
            sentence = sentence.strip()
            if sentence:
                ppl = model.perplexity(sentence)
                L_ppl.append(ppl)

    # Compute the average ppl for this language
    avg_perplexity_bpe = sum(L_ppl) / len(L_ppl) if L_ppl else float('inf')
       )
    dico_ppl_bpe[language] = avg_perplexity_bpe

```

```
print(dico_ppl_bpe)
```

Output =
{'en': 4.241112960558111, 'ar': 3.9732698611363277, 'zh-cn': 4.5825268909478885, 'zh-tw': 4.249055830560958, 'nl': 4.067590098553929, 'fr': 3.877005831083878, 'de': 4.393046165139333, 'it': 3.876855931148883, 'ja': 3.732958265185048, 'ko': 3.9852912257938655, 'pl': 3.7602259580333244, 'pt': 4.04784244879665, 'ru': 3.820490680672048, 'es': 3.8473432518029855, 'th': 3.4017239814988955, 'tr': 3.6492685827600737, 'bg': 3.8573564694343627, 'ca': 3.659682715096509, 'cs': 3.996184392693485, 'da': 4.264450074759901, 'el': 3.889731314536124, 'et': 3.874470236685818, 'fa': 3.954205654666491, 'fi': 3.952928752608692, 'he': 4.42603897973344, 'hi': 4.193528497813784, 'hr': 3.7444551210636936, 'hu': 3.920775639115839, 'id': 4.355121970730566, 'lt': 3.561173781443256, 'lv': 3.6311733422210657, 'ms': 3.888799693951005, 'no': 4.182365406246318, 'ro': 3.9058647509308946, 'sk': 3.5113149186661334, 'sl': 3.9858474861657203, 'sr': 3.143839986790953, 'sv': 3.969842565145596, 'tl': 3.394430642726207, 'uk': 3.697063557362604, 'vi': 3.575903946781156}

We notice that the perplexities are much lower than previously (before: en: 29.05, fr: 27.52 VS now: en: 4.24, fr: 3.87). Before, the language models faced greater uncertainty, likely due to the diversity of word forms and long character sequences. After applying SentencePiece, which segments words into frequent subunits, perplexity decreases for all languages, showing that subunit tokenization simplifies sentence modeling by reducing token diversity, making the models more confident in their predictions.
We can conclude that the SentencePiece tokenization (BPE - Byte Pair Encoding) make language models more efficient, significantly lowering perplexity.