link to the paper: https://drive.google.com/file/d/1qQFRBYMcyWIFQZaGMC63PlmnaR-1BqVx/view?usp=sharing

Indicate group members' names, student numbers, and contributions below:

- 1. Kai Speidel, 2095270
- 2. Lucia Welther, 2102320
- 3. Gabriela Kolodziejska, 2103350
- 4. Rosalie Priol, 2105280
- 5. Magdalena Tatarczuk, 2100133

install dependencies

```
# the code has been tested using the psycho-embeddings library to extract represent
# as long as you make sure that you are producing the correct output.
!git clone https://github.com/MilaNLProc/psycho-embeddings.git
%cd psycho-embeddings
!pip install datasets
!pip install fasttext #installed fasttext as it wasnt available
!pip install osfclient==0.3.0
!pip install pyreadr
!pip install fasttext
!pip install tqdm
```

```
Cloning into 'psycho-embeddings'...
remote: Enumerating objects: 199, done.
remote: Counting objects: 100% (199/199), done.
remote: Compressing objects: 100% (138/138), done.
remote: Total 199 (delta 105), reused 141 (delta 53), pack-reused 0 (from 0)
Receiving objects: 100% (199/199), 67.91 KiB | 13.58 MiB/s, done.
Resolving deltas: 100% (105/105), done.
/content/psycho-embeddings/psycho-embeddings
Requirement already satisfied: datasets in /usr/local/lib/python3.11/dist-pacl
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-pacl
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.11/d
Requirement already satisfied: dill<0.3.9,>=0.3.0 in /usr/local/lib/python3.1
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/python3.11/
Requirement already satisfied: tqdm>=4.66.3 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: xxhash in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: multiprocess<0.70.17 in /usr/local/lib/python3
Requirement already satisfied: fsspec<=2025.3.0,>=2023.1.0 in /usr/local/lib/
```

```
Requirement already satisfied: huggingface-hub>=0.24.0 in /usr/local/lib/pytho
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/p
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/p
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyt
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.1
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.1
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/di
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/pyth
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: fasttext in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: pybind11>=2.2 in /usr/local/lib/python3.11/dis
Requirement already satisfied: setuptools>=0.7.0 in /usr/local/lib/python3.11
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packag
ERROR: Could not find a version that satisfies the requirement osfclient==0.3
ERROR: No matching distribution found for osfclient==0.3.0
Requirement already satisfied: pyreadr in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: pandas>=1.2.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: numpy>=1.23.2 in /usr/local/lib/python3.11/dis
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/di
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: fasttext in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: pybind11>=2.2 in /usr/local/lib/python3.11/dis
Requirement already satisfied: setuptools>=0.7.0 in /usr/local/lib/python3.11
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-package
```

```
# the solution to the assignment has been obtained using these packages.
# you're free to use other packages though: consider this as an indication, not a
import nltk
import numpy as np
import pandas as pd
import fasttext as ft
import pickle as pkl
import fasttext.util
from tqdm import tqdm
from collections import defaultdict
from transformers import AutoTokenizer
from psycho_embeddings import ContextualizedEmbedder
```

task 1

Task 1 (*10 points available, see breakdown per task below*)

You should replicate the main design in the paper *Valence without meaning* by Gatti and colleagues (2024), using estimates collected for Dutch word valence to train linear regression models and apply them to predict the valence of English pseudowords from Gatti and colleagues.

In detail, to train your regression models, you should use the dataset by Speed and Brysbaert (2024) containing crowd-sourced valence ratings (use the metadata to identify the relevant columns) collected for approximately 24,000 Dutch words. See the paper *Ratings of valence, arousal, happiness, anger, fear, sadness, disgust, and surprise for 24,000 Dutch words* by Speed and Brysbaert (2024).

You should train a letter unigram model and a bigram model. Each model should be trained on Dutch words only.

Pay attention to one issue though: pseudowords created for English may be valid words in Dutch: therefore, you should first filter the list of pseudowords against a large store of Dutch words. To do so, use the words in the Dutch prevalence lexicon available in this OSF repository: https://osf.io/9zymw/. Essentially, you need to exclude any pseudoword that happens to be a word for which a prevalence estimate is available, whatever the prevalence is.

Each code block indicates how many points are available and how they are attributed.

> link to the paper "Ratings of *valence, arousal....*": https://link.springer.com/article/10.3758/s13428-023-02239-6

load the datasets

```
#pseudowords Data
# !wget https://osf.io/download/6t2n7/ -O pseudowords.RData

#speed dataset
# !wget https://osf.io/download/h76zj/ -O SpeedDutchWords.xlsx

#!wget https://osf.io/download/jex9n/ -O PrevalenceDutchWords.csv

#the valence ratings for 24,000 Dutch words from Speed and Brysbaert
# !wget https://osf.io/download/6dusr/ -O All_Valence.xlsx
```

```
!wget https://osf.io/download/6t2n7/ -O data_pseudovalence.RData
!wget https://osf.io/download/h76zj/ -O SpeedBrysbaertEmotionNorms.xlsx
!wget https://osf.io/download/jex9n/ -O PrevalenceDutchWords.csv
```

```
--2025-05-12 12:22:33--  https://osf.io/download/6t2n7/
Resolving osf.io (osf.io)... 35.190.84.173
Connecting to osf.io (osf.io)|35.190.84.173|:443... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: https://files.osf.io/v1/resources/kv9at/providers/osfstorage/647d8f
--2025-05-12 12:22:34--  https://files.osf.io/v1/resources/kv9at/providers/os
Resolving files.osf.io (files.osf.io)... 35.186.214.196
Connecting to files.osf.io (files.osf.io)|35.186.214.196|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://storage.googleapis.com/cos-osf-prod-files-us-east1/d1566ad5
--2025-05-12 12:22:36--  https://storage.googleapis.com/cos-osf-prod-files-us
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.68.207, 6
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.68.207|:
HTTP request sent, awaiting response... 200 OK
Length: 33164286 (32M) [application/octet-stream]
Saving to: ‘data_pseudovalence.RData’

data_pseudovalence. 100%[===================>]  31.63M  9.10MB/s    in 3.5s

2025-05-12 12:22:41 (9.10 MB/s) - ‘data_pseudovalence.RData’ saved [33164286/

--2025-05-12 12:22:41--  https://osf.io/download/h76zj/
Resolving osf.io (osf.io)... 35.190.84.173
Connecting to osf.io (osf.io)|35.190.84.173|:443... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: https://files.osf.io/v1/resources/9htuv/providers/osfstorage/64b015
--2025-05-12 12:22:41--  https://files.osf.io/v1/resources/9htuv/providers/os
Resolving files.osf.io (files.osf.io)... 35.186.214.196
Connecting to files.osf.io (files.osf.io)|35.186.214.196|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://storage.googleapis.com/cos-osf-prod-files-us-east1/8bb467f7
--2025-05-12 12:22:44--  https://storage.googleapis.com/cos-osf-prod-files-us
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.68.207, 6
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.68.207|:
HTTP request sent, awaiting response... 200 OK
Length: 5466696 (5.2M) [application/octet-stream]
Saving to: ‘SpeedBrysbaertEmotionNorms.xlsx’

SpeedBrysbaertEmoti 100%[===================>]   5.21M  2.66MB/s    in 2.0s

2025-05-12 12:22:48 (2.66 MB/s) - ‘SpeedBrysbaertEmotionNorms.xlsx’ saved [54

--2025-05-12 12:22:48--  https://osf.io/download/jex9n/
Resolving osf.io (osf.io)... 35.190.84.173
Connecting to osf.io (osf.io)|35.190.84.173|:443... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: https://files.de-1.osf.io/v1/resources/9zymw/providers/osfstorage/6
--2025-05-12 12:22:48--  https://files.de-1.osf.io/v1/resources/9zymw/provide
Resolving files.de-1.osf.io (files.de-1.osf.io)... 35.186.249.111
Connecting to files.de-1.osf.io (files.de-1.osf.io)|35.186.249.111|:443... co
HTTP request sent, awaiting response... 302 Found
```

```
HTTP request sent, awaiting response... 302 Found
Location: https://storage.googleapis.com/cos-osf-prod-files-de-1/bd3e94ede4fa
--2025-05-12 12:22:50--  https://storage.googleapis.com/cos-osf-prod-files-de-
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.68.207, 64
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.68.207|:4
HTTP request sent, awaiting response... 200 OK
Length: 4490964 (4.3M) [application/octet-stream]
Saving to: 'PrevalenceDutchWords.csv'
```

```
import pandas as pd

# Read and store content
# of an excel file
read_file = pd.read_excel("SpeedBrysbaertEmotionNorms.xlsx", engine='openpyxl') #

# Write the dataframe object
# into csv file
read_file.to_csv ("SpeedBrysbaertEmotionNorms.xlsx.csv",
                  index = None,
                  header=True)

# read csv file and convert
# into a dataframe object
gatti_dutch_speed_df= pd.DataFrame(pd.read_csv("SpeedBrysbaertEmotionNorms.xlsx.c

# show the dataframe
gatti_dutch_speed_df
```

	Word	Arousal	Valence	ValenceCategory	ValenceVsNeutral	Happine
0	mama	2.812500	4.000000	positive	valenced	3.300(
1	ja	2.823529	3.894737	positive	valenced	3.818
2	papa	2.562500	3.722222	positive	valenced	4.142
3	nee	2.928571	2.350000	negative	neutral	1.000(
4	kaka	3.357143	2.050000	negative	neutral	1.090
...	...	...	...	...	...	
23981	organogram	2.687500	3.000000	neutral	neutral	1.250(
23982	empirisch	3.153846	3.176471	positive	neutral	1.800(
23983	hypothalamus	3.200000	3.000000	neutral	neutral	1.818
23984	utilitarisme	2.727273	3.000000	neutral	neutral	1.250(
23985	twitteren	3.000000	3.200000	positive	neutral	1.333

23986 rows × 35 columns

```
# loading the Prevalende Dutch words
prevalence_dutch_df = pd.read_csv("PrevalenceDutchWords.csv", sep="\t")
prevalence_dutch_df.head(5)
```

	word	n.obs	irt.prevalence	z.irt.prevalence	prevalence	z.prevalence
0	T-shirt	324	0.986622	2.215053	0.978395	1.689888
1	aagje	303	0.907405	1.324941	0.877888	1.075808
2	aagt	324	0.169817	-0.954888	0.188272	-0.827920
3	aai	335	0.993290	2.472451	0.988060	1.794794
4	aaibaar	333	0.996284	2.676802	0.990991	1.830889

Next steps: Generate code with prevalence_dutch_df View recommended plots New interactive

```
#converting r data into CSV
import pyreadr
result = pyreadr.read_r("data_pseudovalence.RData") #
pseudowords_df = result['data_2'] # Convert R data to pandas DataFrame


pseudowords_df.to_csv("data_pseudovalence.csv",index=False, header=True)
pseudowords_df.head()
```

	X	pseudoword	Value	predicted_valence	predictedL_valence	predictedL_Bi
0	1	abhert	0.452501	7.414814	5.116167	
1	2	abhict	0.434171	8.233714	5.059183	
2	3	acleat	0.527803	5.552468	5.262971	
3	4	acmure	0.604889	8.714640	5.120029	
4	5	acoed	0.538990	7.340002	5.115652	

Next steps: Generate code with pseudowords_df View recommended plots New interactive sheet

```
"""
gatti_pseudowords_df = pd.read_excel("/Users/kaispeidel/Downloads/CL_group_work/g
dutch_speed_df = pd.read_excel("/Users/kaispeidel/Downloads/CL_group_work/dutch_s
prevalence_dutch_df = pd.read_csv("/Users/kaispeidel/Downloads/CL_group_work/prev
"""
```

```
'\ngatti_pseudowords_df = pd.read_excel("/Users/kaispeidel/Downloads/CL_group
```

```
_work/gatti_pseudowords_df.xlsx")\ndutch_speed_df = pd.read_excel("/Users/kai
speidel/Downloads/CL_group_work/dutch_speed_df.xlsx")\nprevalence_dutch_df =
```

convert the loaded datasets into csv

convert the pseudowords data into CSV import pyreadr result = pyreadr.read_r('pseudowords.RData')

print(result.keys()) output: odict_keys(['data_fin', 'data_2', 'data_3', '.Random.seed', 'Count', 'comb_2', 'comb_3'])

""" print(result['data_fin'].head()) print(result['data_2'].head()) print(result['data_3'].head()) print(result['Count'].head()) print(result['comb_2'].head()) print(result['comb_3'].head()) """

df = result['data_fin'].reset_index() df.rename(columns={'index': 'word'}, inplace=True) df.to_csv('data_fin.csv', index=False)

""" result['data_2'].to_csv('data_2.csv', index=False) result['data_3'].to_csv('data_3.csv', index=False) result['Count'].to_csv('Count.csv', index=False) result['comb_2'].to_csv('comb_2.csv', index=False) result['comb_3'].to_csv('comb_3.csv', index=False) """

```
#Valence convert valence xlx to csv
#All_Valence_df = pd.read_excel("All_Valence.xlsx")
```

first exercise

```
# read in the pseudowords from Gatti and colleagues, as well as the valence ratin
# show the first 5 lines of each dataset.
# 1 point for identifying the correct files and correctly loading their content
```

```
#pseudowords_df = pd.read_csv("data_fin.csv")
pseudowords_df.head(5)
```

	X	pseudoword	Value	predicted_valence	predictedL_valence	predictedL_Bi
0	1	abhert	0.452501	7.414814	5.116167	
1	2	abhict	0.434171	8.233714	5.059183	
2	3	acleat	0.527803	5.552468	5.262971	
3	4	acmure	0.604889	8.714640	5.120029	
4	5	acoed	0.538990	7.340002	5.115652	

Next steps: Generate code with pseudowords_df | View recommended plots | New interactive sheet

```
# 'SpeedDutchWords.csv'
#dutch_speed_df = pd.read_excel("SpeedDutchWords.xlsx")
gatti_dutch_speed_df.head(5)
```

	Word	Arousal	Valence	ValenceCategory	ValenceVsNeutral	Happiness	Ange
0	mama	2.812500	4.000000	positive	valenced	3.300000	1.00000
1	ja	2.823529	3.894737	positive	valenced	3.818182	1.0909
2	papa	2.562500	3.722222	positive	valenced	4.142857	1.1428
3	nee	2.928571	2.350000	negative	neutral	1.000000	1.7272
4	kaka	3.357143	2.050000	negative	neutral	1.090909	1.4545

5 rows × 35 columns

```
#creating a new dataframe for simplicity
dutch_speed_df = gatti_dutch_speed_df[['Word', 'Valence']].copy()
pseudowords_df = pseudowords_df[['pseudoword', 'Value']].copy()
```

```
#normalize the True Valence
min_val = dutch_speed_df["Valence"].min()
max_val = dutch_speed_df["Valence"].max()
dutch_speed_df["normalized_true_valence"] = (dutch_speed_df["Valence"] - min_val)
```

second exercise: filter out valid Dutch Words

```
# filter out pseudowords that happen to be valid Dutch words (mind case folding!)
# show the set of pseudowords filtered out.
# 1 point for applying the correct filtering
words = list(dutch_speed_df['Word'])
pseudowords = list(pseudowords_df['pseudoword'])

dutch_words_set = set(word.lower() for word in words)
pseudowords_set = set(pseudowords)

ValidDutchPseudowords = pseudowords_set.intersection(dutch_words_set)
print(ValidDutchPseudowords)

filtered_pseudowords = pseudowords_set.difference(ValidDutchPseudowords)
filtered_pseudowords_df = pseudowords_df[pseudowords_df['pseudoword'].isin(filter
```

```
{'pimpen'}
```

```
filtered_pseudowords_df.head()
```

	pseudoword	Value
0	abhert	0.452501
1	abhict	0.434171
2	acleat	0.527803
3	acmure	0.604889
4	acoed	0.538990

Next steps: Generate code with filtered_pseudowords_df View recommended plots New intera

third exercise: encode Dutch words and pseudo words as UNI- and BI-gram vectors

```
# encode Dutch words and pseudowords from Gatti et al as uni- and bi-gram vectors
# show the uni-gram and bi-gram encoding of the pseudoword ampgrair
# 2 points for correctly encoding the target strings as uni- and bi-gram vectors
from sklearn.feature_extraction.text import CountVectorizer

target_pseudoword = "ampgrair"

vectorizer_unigrams = CountVectorizer(analyzer='char', ngram_range=(1, 1))
vectorizer_bigrams = CountVectorizer(analyzer='char', ngram_range=(2, 2))

# fit
X_unigrams = vectorizer_unigrams.fit_transform([target_pseudoword])
X_bigrams = vectorizer_bigrams.fit_transform([target_pseudoword])

# feature names
unigram_features = vectorizer_unigrams.get_feature_names_out()
bigram_features = vectorizer_bigrams.get_feature_names_out()

# encoded vectors
unigram_vector = X_unigrams.toarray()
bigram_vector = X_bigrams.toarray()


# print
print("Uni-gram encoding of '{}':".format(target_pseudoword))
```

```
for feature, value in zip(unigram_features, unigram_vector[0]):
    print("{}: {}".format(feature, value))

print("\nBi-gram encoding of '{}':".format(target_pseudoword))
for feature, value in zip(bigram_features, bigram_vector[0]):
    print("{}: {}".format(feature, value))
```

```
Uni-gram encoding of 'ampgrair':
a: 2
g: 1
i: 1
m: 1
p: 1
r: 2

Bi-gram encoding of 'ampgrair':
ai: 1
am: 1
gr: 1
ir: 1
mp: 1
pg: 1
ra: 1
```

4th exercise Valence estimates to train model's

```
# use word valence estimates from Speed and Brysbaert (2024) to train
# - a uni-gram model
# - a bi-gram model
# 2 points for correctly trained models
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split


valence_DF = dutch_speed_df[['Word', 'normalized_true_valence']]
valence_DF.head()

words = valence_DF['Word']
valence = valence_DF['normalized_true_valence']

X_train, X_test, y_train, y_test = train_test_split(words, valence, test_size=0.2

# for later comparison
y_test_unigram = y_test
y_test_bigram = y_test

# fit and transform the training data
X_train_unigrams = vectorizer_unigrams.fit_transform(X_train)
X_train_bigrams = vectorizer_bigrams.fit_transform(X_train)
```

```
# transform the test data
X_test_unigrams = vectorizer_unigrams.transform(X_test)
X_test_bigrams = vectorizer_bigrams.transform(X_test)

# linear regressor for uni and bigram
SpeedDutchWords_valence_unigramModel = LinearRegression()
SpeedDutchWords_valence_unigramModel.fit(X_train_unigrams, y_train)

SpeedDutchWords_valence_bigramModel = LinearRegression()
SpeedDutchWords_valence_bigramModel.fit(X_train_bigrams, y_train)
```

```
▾ LinearRegression ⓘ ?
LinearRegression()
```

```
# apply trained models to predict the valence of pseudowords from Gatti et al (20
# Then apply the same models back onto the training set to see how well they pred
# 2 points for correctly applied models

# predicting valence of pseudowords from Gatti et al (2024)
print(f"predicting valence of pseudowords from Gatti et al (2024)")
print(f"-"*60)

pseudowords = pseudowords_df['pseudoword']

pseudowords = pseudowords.dropna()

# on pseudowords turn into uni and bigrams
X_predict_unigrams_pseudowords = vectorizer_unigrams.transform(pseudowords)
X_predict_bigrams_pseudowords = vectorizer_bigrams.transform(pseudowords)

# turn the pseudwords into strings to ensure that they are a hashable type for th
pseudowords_df["pseudoword"] = pseudowords_df["pseudoword"].astype(str)

pseudowords_df["unigrams"] = pseudowords_df["pseudoword"].apply(lambda x: vectori
pseudowords_df["unigram_predictions"] = pseudowords_df["unigrams"].apply(lambda x

pseudowords_df["bigrams"] = pseudowords_df["pseudoword"].apply(lambda x: vectoriz
pseudowords_df["bigram_predictions"] = pseudowords_df["bigrams"].apply(lambda x:

# apply
dutch_speed_df["unigrams"] = dutch_speed_df["Word"].apply(lambda x: vectorizer_un
dutch_speed_df["unigram_predictions"] = dutch_speed_df["unigrams"].apply(lambda x

dutch_speed_df["bigrams"] = dutch_speed_df["Word"].apply(lambda x: vectorizer_big
dutch_speed_df["bigram_predictions"] = dutch_speed_df["bigrams"].apply(lambda x:
```

```
predicting valence of pseudowords from Gatti et al (2024)
```

```
    ------------------------------------------------------------
```

```python
# uni gram df
df_unigram_preds = pd.DataFrame({
    'pseudoword': pseudowords,
    'predicted_prevalence_unigram': pseudowords_df["unigram_predictions"]
})

#bi gram df
df_bigram_preds = pd.DataFrame({
    'pseudoword': pseudowords,
    'predicted_prevalence_bigram': pseudowords_df["bigram_predictions"]
})

print("Pseudowords: Unigram Predictions DataFrame:")
print(df_unigram_preds.head())

print("\n Pseudowords: Bigram Predictions DataFrame:")
print(df_bigram_preds.head())
```

```
Pseudowords: Unigram Predictions DataFrame:
  pseudoword  predicted_prevalence_unigram
0     abhert                      0.500364
1     abhict                      0.504582
2     acleat                      0.520781
3     acmure                      0.516611
4      acoed                      0.516951

 Pseudowords: Bigram Predictions DataFrame:
  pseudoword  predicted_prevalence_bigram
0     abhert                     0.554963
1     abhict                     0.517079
2     acleat                     0.580786
3     acmure                     0.532523
4      acoed                     0.561651
```

```python
# test how well the model predicts with the train and test data
predicted_valence_test_unigrams = SpeedDutchWords_valence_unigramModel.predict(X_
predicted_valence_test_bigrams = SpeedDutchWords_valence_bigramModel.predict(X_te

from sklearn.metrics import mean_squared_error, r2_score

# Evaluation for unigram model
mse_unigram = mean_squared_error(y_test, predicted_valence_test_unigrams)
r2_unigram = r2_score(y_test, predicted_valence_test_unigrams)

# Evaluation for bigram model
mse_bigram = mean_squared_error(y_test, predicted_valence_test_bigrams)
r2_bigram = r2_score(y_test, predicted_valence_test_bigrams)

print("Unigram Model Performance:")
```

```
print("Unigram Model Performance:")
print(f"  MSE: {mse_unigram:.4f}")
print(f"  R²: {r2_unigram:.4f}")

print("\nBigram Model Performance:")
print(f"  MSE: {mse_bigram:.4f}")
print(f"  R²: {r2_bigram:.4f}")
```

```
Unigram Model Performance:
  MSE: 0.0279
  R²: 0.0047

Bigram Model Performance:
  MSE: 0.0260
  R²: 0.0721
```

```
# compute the Spearman correlation coefficients between true valence and predicte
# - words from Speed and Brysbaert (2024)
# - pseudowords from Gatti and colleagues (2024)
# show both correlation coefficients.
# 2 points for the correct Spearman correlation coefficients (rounded to the thir

from scipy.stats import spearmanr

# -- Speed & Brysbaert words (with true valence labels) --

# spearman correlation between Unigram true valence and predicted valence
spearman_corr_uni_gram_words_preds, _ = spearmanr(dutch_speed_df["normalized_true_

# spearman correlation between Bi-gram true valence and predicted valence
spearman_corr_bi_gram_words_preds,_ = spearmanr(dutch_speed_df["normalized_true_v


# spearman correlation between unigram and bigram predictions for pseudowords
spearman_corr_uni_gram_pseudo_preds, _ = spearmanr(pseudowords_df["Value"], pseud

# spearman correlation between unigram and birgram predictions for pseudowords
spearman_corr_bi_gram_pseudo_preds, _ = spearmanr(pseudowords_df["Value"], pseudo


# --- results ---

print("Spearman correlations for Speed & Brysbaert: uni-gram (true-valence and pr
print(f"{spearman_corr_uni_gram_words_preds:.3f}")

print("Spearman correlations for Speed & Brysbaert: bi-gram (true-valence and pre
print(f"{spearman_corr_bi_gram_words_preds:.3f}")

print("\nSpearman correlation between unigram predictions on pseudowords true and
print(f"  Pseudowords prediction correlation: {spearman_corr_uni_gram_pseudo_pred
```

```
print("\nSpearman correlation between bigram predictions on true and predicted va
print(f"  Pseudowords prediction correlation: {spearman_corr_bi_gram_pseudo_preds
```

```
Spearman correlations for Speed & Brysbaert: uni-gram (true-valence and predi
0.088
Spearman correlations for Speed & Brysbaert: bi-gram (true-valence and predic
0.312

Spearman correlation between unigram predictions on pseudowords true and pred
  Pseudowords prediction correlation: 0.271

Spearman correlation between bigram predictions on true and predicted valence
  Pseudowords prediction correlation: 0.075
```

task 2

Task 2 (*8 points available, see breakdown below*)

Again following Gatti and colleagues, you should encode the target strings (pseudowords and Dutch words from Speed and Brysbaert) as fastText embeddings, train a multiple regression model on Dutch words and apply it to the pseudowords in Gatti et al. You should finally report the Spearman correlation coefficient between observed and predicted valence for both words and pseudowords.

You should use the pre-trained fastText model for Dutch, available at this page: https://fasttext.cc/docs/en/crawl-vectors.html

Finally, you should answer two questions about the fastText model (see below).

loading FastText

loading the FastTextModel

this approach of loading the Model proved to work the best for our notebook

```
# load the fastText model
# 1 point for correctly loading the appropriate fastText model

!wget https://dl.fbaipublicfiles.com/fasttext/vectors-crawl/cc.nl.300.bin.gz -O c
```

```
--2025-05-12 12:23:34--  https://dl.fbaipublicfiles.com/fasttext/vectors-craw
Resolving dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)... 108.157.254.102,
```

```
Connecting to dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)|108.157.254.102
HTTP request sent, awaiting response... 200 OK
Length: 4505743140 (4.2G) [application/octet-stream]
Saving to: 'cc.nl.300.bin.gz'

cc.nl.300.bin.gz    100%[===================>]   4.20G  53.2MB/s    in 42s

2025-05-12 12:24:16 (103 MB/s) - 'cc.nl.300.bin.gz' saved [4505743140/4505743
```

```
# unzipping the bin file
import gzip
import shutil
from tqdm import tqdm
import os

input_file = 'cc.nl.300.bin.gz'
output_file = 'cc.nl.300.bin'

input_size = os.path.getsize(input_file)

with gzip.open(input_file, 'rb') as f_in:
    with open(output_file, 'wb') as f_out:
        with tqdm(total=input_size, unit='B', unit_scale=True, desc=f"unzipping {
            chunk_size = 1024 * 1024
            while True:
                chunk = f_in.read(chunk_size)
                if not chunk:
                    break
                f_out.write(chunk)
                pbar.update(len(chunk))
```

```
unzipping cc.nl.300.bin.gz: 7.24GB [01:22, 87.8MB/s]
```

```
import fasttext
model = fasttext.load_model('cc.nl.300.bin')
```

```
# test if it works
vector = model.get_word_vector("fiets")  # "bicycle" in Dutch
print(vector[:10])
```

```
[ 0.05013572  0.02355762  0.16043806 -0.08914731  0.00348932 -0.01757337
 -0.00179433 -0.01858325 -0.04348693 -0.03980046]
```

exercise

What is the dimensionality of the pre-trained Dutch fastText embeddings? (*1 point for the correct answer*) **ANS:** "*These models were trained using CBOW with position-weights, in* *correct answer)* **ANS:** *"These models were trained using CBOW with position weights, in dimension 300" from website*

```
print(f"The dimension is: {len(vector)}")
```

```
The dimension is: 300
```

What minimum and maximum n-gram size was specified for training this fastText model? **ANS:** 5

```
# encode Dutch words and pseudowords as fastText embeddings
# show the first 20 values of the embedding of the word 'speelplaats' and of the
# 2 points for correctly encoding words and pseudowords with fastText

word_embeddings = {}
for word in words:
    word_embeddings[word] = model.get_word_vector(word)

pseudoword_embeddings = {}
for word in pseudowords:
  pseudoword_embeddings[word] = model.get_word_vector(word)

print("Embedding for 'speelplaats':", word_embeddings['speelplaats'][:20])
print("Embedding for 'aardvak':" , pseudoword_embeddings['danchunk'][:20])
```

```
Embedding for 'speelplaats': [ 0.0253247  -0.00634261  0.02746305 -0.04024595
 -0.04152017 -0.01824508 -0.00645641  0.00093806  0.0708492  -0.03291791
  0.00263817 -0.02825846 -0.02188046 -0.03188037 -0.01846142 -0.02203094
 -0.01883078 -0.00259199]
Embedding for 'aardvak': [-0.00592199  0.00097547  0.05925412  0.00053251 -0.(
 -0.02829577  0.00972911 -0.02510111 -0.11454885 -0.02695064  0.01551034
  0.02384409  0.01009528  0.04545438  0.00997385 -0.00474529  0.02524533
  0.02430548 -0.02851078]
```

```
#put the embeddings into the csv
dutch_speed_df["fasttext_embedding"] = dutch_speed_df["Word"].apply(lambda x: mod
```

```
pseudowords_df["pseudoword"] = pseudowords_df["pseudoword"].astype(str)
```

```
pseudowords_df["fasttext_embedding"] = pseudowords_df["pseudoword"].apply(lambda
```

```
# train regression model on word valence
# 1 point for correctly training the regression model

from sklearn.linear_model import LinearRegression
```

```
from sklearn.model_selection import train_test_split

X = np.array([vec for vec in dutch_speed_df["fasttext_embedding"]])
y = dutch_speed_df["normalized_true_valence"]

# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s

# for later:
y_test_fasttext = y_test

# linear regression model
regressor_fasttext = LinearRegression()
regressor_fasttext.fit(X_train,y_train)
```

```
▾ LinearRegression ⓘ ?
LinearRegression()
```

```
# apply the trained model to predict the valence of pseudowords from Gatti et al
pseudowords_df["fasttext_valence_pred"] = pseudowords_df["fasttext_embedding"].ap

# Then apply the same model back onto the training set to see how well it predict
dutch_speed_df["fasttext_valence_pred"] = dutch_speed_df["fasttext_embedding"].ap
```

```
# apply the trained model to predict the valence of pseudowords from Gatti et al
# Then apply the same model back onto the training set to see how well it predict
# 1 point for correctly applied model

fasttext_pseudowords_predicted_valence = pseudowords_df["fasttext_valence_pred"]

fasttext_words_predicted_valence = dutch_speed_df["fasttext_valence_pred"]
```

```
dutch_speed_df.head()
```

	Word	Valence	normalized_true_valence	unigrams	unigram_predictions	bigr
0	mama	4.000000	0.779221	[[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0,...	0.509169	[[0, (0, 0, (0, 0, (0,
1	ja	3.894737	0.751880	[[1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,...	0.535415	[[0, (0, 0, (0, 0, (0, (

2	papa	3.722222	0.707071	[[2, 0, 0, 0, 0, 0, 0, 0,	0.520095	[[0, 0, 0, 0, 0, 0,

Next steps: Generate code with `dutch_speed_df` View recommended plots New interactive sheet

```
pseudowords_df
```

	pseudoword	Value	unigrams	unigram_predictions	bigrams	bigram_pre
0	abhert	0.452501	<Compressed Sparse Row sparse matrix of dtype ...	0.500364	[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
1	abhict	0.434171	<Compressed Sparse Row sparse matrix of dtype ...	0.504582	[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
2	acleat	0.527803	<Compressed Sparse Row sparse matrix of dtype ...	0.520781	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
3	acmure	0.604889	<Compressed Sparse Row sparse matrix of dtype ...	0.516611	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
4	acoed	0.538990	<Compressed Sparse Row sparse matrix of dtype ...	0.516951	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
...	...	...	...	...	...	...
			<Compressed		[[0, 0, 0,	

Next steps: Generate code with `pseudowords_df` View recommended plots New interactive sheet

```
# compute the Spearman correlation coefficients between true valence and predicte
# - words from Speed and Brysbaert (2024)
# - pseudowords from Gatti and colleagues (2024)
# show the correlation coefficient.
# 1 point for the correct Spearman correlation coefficients (rounded to the third

from scipy.stats import spearmanr
```

```
# spearman corr words: True Valence and predicted valence with FastText
spearman_corr_speed_fasttext_Brysbeart, _ = spearmanr(dutch_speed_df["normalized_
print(f"Speed & Brysbaert Spearman_cor: {spearman_corr_speed_fasttext_Brysbeart:.

# spearman corr pseudowords: True Valence and predicted valence with FastText

spearman_corr_fasttext_Gatti, _ = spearmanr(pseudowords_df["Value"], pseudowords_
print(f"Gatti Spearman_corr: {float(spearman_corr_fasttext_Gatti.round(3))}")


print("\nInterpretation:")
print(f"The model explains {spearman_corr_speed_fasttext_Brysbeart**2:.1%} of var
print(f"and {spearman_corr_fasttext_Gatti**2:.1%} of variance in pseudoword valen
```

```
Speed & Brysbaert Spearman_cor: 0.723
Gatti Spearman_corr: 0.091

Interpretation:
The model explains 52.3% of variance in Dutch word valence rankings
and 0.8% of variance in pseudoword valence rankings
```

task 3

Task 3 (*6 points available, see breakdown below*)

Now you are asked to extend the work by Gatti et al by also considering the representations learned by a transformer-based models, in detail *RobBERT v2* (https://huggingface.co/pdelobelle/robbert-v2-dutch-base). You should follow the same pipeline as for the previous models, encoding both Dutch words from Speed and Brysbaert (2024) and the pseudowords from Gatti et al using the embedding of each string at layer 0, before positional information is factored in. If a string consists of multiple tokens, average the embeddings of all tokens to produce the embedding of the whole string. Then train a multiple regression model on the valence of Dutch words, apply it to the pseudowords, and compute the Spearman correlation between observed and predicted ratings.

Use the HuggingFace model card for RobBERT v2 to check how to access it.

I recommend saving the embeddings to file once you have generated them and you know they are correct: embedding thousands of strings takes some time, and you don't want to have to do it again. For the same reason, develop your code by considering only a small fractions of the words and pseudowords, in order to quickly see if something is wrong. Only when you are positive it works, embed all strings.

loading robert

```
# load and instantiate the right model

# load model directly
from transformers import RobertaModel, RobertaTokenizer

model_name = "pdelobelle/robbert-v2-dutch-base"

tokenizer = RobertaTokenizer.from_pretrained(model_name)
model = RobertaModel.from_pretrained(model_name)

# 1 point for loading the right model
```

```
loading file vocab.json from cache at /root/.cache/huggingface/hub/models--pd
loading file merges.txt from cache at /root/.cache/huggingface/hub/models--pd
loading file added_tokens.json from cache at None
loading file special_tokens_map.json from cache at /root/.cache/huggingface/hu
loading file tokenizer_config.json from cache at /root/.cache/huggingface/hub/
loading file tokenizer.json from cache at /root/.cache/huggingface/hub/models-
loading file chat_template.jinja from cache at None
loading configuration file config.json from cache at /root/.cache/huggingface/
Model config RobertaConfig {
  "architectures": [
    "RobertaForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "classifier_dropout": null,
  "eos_token_id": 2,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-05,
  "max_position_embeddings": 514,
  "model_type": "roberta",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "output_past": true,
  "pad_token_id": 1,
  "position_embedding_type": "absolute",
  "transformers_version": "4.51.3",
  "type_vocab_size": 1,
  "use_cache": true,
  "vocab_size": 40000
}

loading configuration file config.json from cache at /root/.cache/huggingface/
Model config RobertaConfig {
```

```
    "architectures": [
      "RobertaForMaskedLM"
    ],
    "attention_probs_dropout_prob": 0.1,
    "bos_token_id": 0,
    "classifier_dropout": null,
    "eos_token_id": 2,
    "gradient_checkpointing": false,
    "hidden_act": "gelu",
    "hidden_dropout_prob": 0.1,
    "hidden_size": 768,
    "initializer_range": 0.02,
    "intermediate_size": 3072,
    "layer_norm_eps": 1e-05,
    "max_position_embeddings": 514,
    "model_type": "roberta",
    "num_attention_heads": 12,
    "num_hidden_layers": 12,
    "output_past": true,
    "pad_token_id": 1,
```


```
# encode the words and pseudowords using RobBERT v2. I've used the free GPU runti
# but in this case you need to batch the words and pseudowords. You can use the f
# but you will have to pay attention at how you store embeddings.
# show the first 20 values of the embedding of the word 'miauwen' and of the pseu
# 2 points for correctly encoding words and pseudowords
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

def chunks(lst, n):

    #* chunks them into equal chunks and returns a list

    chunked = []
    for i in range(0, len(lst), n):
        chunked.append(lst[i:i + n])
    return chunked

def get_embeddings(word_batch):
    # Convert all items to strings
    word_batch_str = [str(word) for word in word_batch]

    # Tokenize words
    encoded_input = tokenizer(word_batch_str, padding=True, truncation=True, retu

    # Move input to GPU if available
    encoded_input = {k: v.to(device) for k, v in encoded_input.items()}

    # Get embeddings
    with torch.no_grad():
```

```
        output = model(**encoded_input)

    # Extract CLS token embedding (first token) for each word
    embeddings = output.last_hidden_state[:, 0, :].cpu().numpy()

    return embeddings, word_batch_str



# Create batches for processing
batch_size = 32  # Adjust based on GPU memory
word_batches = chunks(words, batch_size)
pseudoword_batches = chunks(pseudowords, batch_size)

# Process word batches and store embeddings
Robert_word_embeddings = {}
for batch in word_batches:
    batch_embeddings, batch_words = get_embeddings(batch)
    for i, word in enumerate(batch_words):
        Robert_word_embeddings[word] = batch_embeddings[i]

# Process pseudoword batches and store embeddings
Robert_pseudoword_embeddings = {}
for batch in pseudoword_batches:
    batch_embeddings, batch_words = get_embeddings(batch)
    for i, word in enumerate(batch_words):
        Robert_pseudoword_embeddings[word] = batch_embeddings[i]

# Debug - print types of items in Robert_word_embeddings and Robert_pseudoword_em
print(f"Number of Robert_word_embeddings: {len(Robert_word_embeddings)}")
print(f"Number of Robert_pseudoword_embeddings: {len(Robert_pseudoword_embeddings
```

```
Number of Robert_word_embeddings: 23986
Number of Robert_pseudoword_embeddings: 1500
```

```
dutch_speed_df['robbert_embedding'] = dutch_speed_df['Word'].apply(lambda x: Robe
pseudowords_df['robbert_embedding'] = pseudowords_df['pseudoword'].apply(lambda x
```

```
print("Embedding for 'miauwen':", Robert_word_embeddings['miauwen'][:20])

print("Embedding for 'lixthless'':", pseudoword_embeddings['lixthless'][:20])
```

```
Embedding for 'miauwen': [-1.3902338   0.27635536  0.51612    -0.91370875 -0.
  0.47247317  0.48153764 -0.20918477 -0.06635422 -0.00960599  0.7847474
  0.07366486  0.5044496  -0.15428922  1.429451    0.02033783  0.894272
  0.23356143 -0.0842339 ]
Embedding for 'lixthless'': [ 0.02332416  0.00734619  0.00694739  0.0037425  
  0.01192105  0.01679212  0.0203222  -0.01754443  0.02184253  0.00873986
 -0.00872535  0.01364204  0.02840464 -0.00303171  0.00469133  0.03704519
```

```
-0.02593704 -0.00247694]
```

regression model Robert

```
# train regression model on word valence estimates from Speed and Brysbaert (2024
# 1 point for correctly training the regression model

X = list(Robert_word_embeddings.values())
y = dutch_speed_df["normalized_true_valence"]

# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
y_test_RobBERT = y_test

# linear regression model
Robert_regressor_fasttext = LinearRegression()
Robert_regressor_fasttext.fit(X_train,y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
# apply the trained model to predict the valence of pseudowords from Gatti et al
# Then apply the same model back onto the training set to see how well it predict
# 1 point for correctly applied model

# apply the trained model to predict the valence of pseudowords from Gatti et al
pseudowords_df["robbert_valence_pred"] = pseudowords_df["robbert_embedding"].appl

# Then apply the same model back onto the training set to see how well it predict
dutch_speed_df["robbert_valence_pred"] = dutch_speed_df["robbert_embedding"].appl
```

```
pseudowords_df
```

	pseudoword	Value	unigrams	unigram_predictions	bigrams	bigram_pre
0	abhert	0.452501	<Compressed Sparse Row sparse matrix of dtype ...	0.500364	[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	
1	abhict	0.434171	<Compressed Sparse Row sparse matrix of dtype	0.504582	[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,	

			of dtype ...		0,...
2	acleat	0.527803	<Compressed Sparse Row sparse matrix of dtype ...	0.520781	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
3	acmure	0.604889	<Compressed Sparse Row sparse matrix of dtype ...	0.516611	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
4	acoed	0.538990	<Compressed Sparse Row sparse matrix of dtype ...	0.516951	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
...	...	...	...	...	...
1495	zauze	0.501798	<Compressed Sparse Row sparse matrix of dtype ...	0.502049	[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
1496	zerow	0.461897	<Compressed Sparse Row sparse matrix of dtype ...	0.483939	[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
1497	zilk	0.548832	<Compressed Sparse Row sparse matrix of dtype ...	0.479723	[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
1498	zohels	0.471812	<Compressed Sparse Row sparse matrix	0.481103	[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Next steps: Generate code with pseudowords_df | View recommended plots | New interactive sheet

```
# compute the Spearman correlation coefficients between true valence and predicte
# - words from Speed and Brysbaert (2024)
# - pseudowords from Gatti and colleagues (2024)
# show the correlation coefficient
# 1 point for the correct Spearman correlation coefficients (rounded to the third

spearman_speed_robbert, _ = spearmanr(dutch_speed_df["normalized_true_valence"],
print(f"Speed Spearman_corr: {float(spearman_speed_robbert.round(3))}")
```

```
# - pseudowords from Gatti and colleagues (2024)
spearman_pseudo_robbert, _ = spearmanr(pseudowords_df["Value"], pseudowords_df["r
print(f"Gatti Spearman_corr: {float(spearman_pseudo_robbert.round(3))}")
```

```
Speed Spearman_corr: 0.45
Gatti Spearman_corr: 0.121
```

task 4

Task 4 (*16 points available, 4 for each question*)

Answer the following questions.

4a. Describe the performance of each featurization, comparing

- the performance of a same model between the training and test set
- the performance of different models on the training set
- the performance of different models on the test set

(*4 points available, max 150 words*)

```
spearman_values = {}
spearman_values['unigram_words'] = spearman_corr_uni_gram_words_preds
spearman_values['bigram_words'] = spearman_corr_bi_gram_words_preds
spearman_values['unigram_pseudowords'] = spearman_corr_uni_gram_pseudo_preds
spearman_values['bigram_pseudowords'] = spearman_corr_bi_gram_pseudo_preds
spearman_values['fasttext_words'] = spearman_corr_speed_fasttext_Brysbeart
spearman_values['fasttext_pseudowords'] = spearman_corr_fasttext_Gatti
spearman_values['robbert_words'] = spearman_speed_robbert
spearman_values['robbert_pseudowords'] = spearman_pseudo_robbert
```

```
spearman_df = pd.DataFrame.from_dict(spearman_values, orient='index', columns=['S
spearman_df = spearman_df.reset_index()
```

```
spearman_df
```

	index	Spearman_Correlation
0	unigram_words	0.088146
1	bigram_words	0.312037
2	unigram_pseudowords	0.271141
3	bigram_pseudowords	0.074749
4	fasttext_words	0.723127

5	fasttext_pseudowords	0.090648
6	robbert_words	0.449699
7	robbert_pseudowords	0.120995

Next steps: Generate code with spearman_df View recommended plots New interactive sheet

4a

```
print(f"--- Performance of the same model between the traing and test set ---")
print(f"\n Considering the Speed and Brysbeart dutch_words being used as the trai
print(f"\n The following performance values were obtained:")
print(f"\n Unigram model: {spearman_corr_uni_gram_words_preds:.3f} (train) vs {sp
print(f"\n Bigram model: {spearman_corr_bi_gram_words_preds:.3f} (train) vs {spea
print(f"\n FastText model: {spearman_corr_speed_fasttext_Brysbeart:.3f} (train) v
print(f"\n RobBERT model: {spearman_speed_robbert:.3f} (train) vs {spearman_pseud
print("\n We can observe overall that the Bigram, Fasttext and RobBERT model all
print("The Unigram model performed better on the test set than on the training se
```

```
--- Performance of the same model between the traing and test set ---

 Considering the Speed and Brysbeart dutch_words being used as the training s

 The following performance values were obtained:

 Unigram model: 0.088 (train) vs 0.271 (test)

 Bigram model: 0.312 (train) vs 0.075 (test)

 FastText model: 0.723 (train) vs 0.091 (test)

 RobBERT model: 0.450 (train) vs 0.121 (test)

 We can observe overall that the Bigram, Fasttext and RobBERT model all perfo
The Unigram model performed better on the test set than on the training set,
```

```
print("--- Performance of the different models on the training set ---")
print(f"\n Considering the Speed and Brysbeart dutch_words being used as the trai
print(f"\n The following performance values were obtained:")
print(f"\n Unigram model: {spearman_corr_uni_gram_words_preds:.3f}")
print(f"\n Bigram model: {spearman_corr_bi_gram_words_preds:.3f}")
print(f"\n FastText model: {spearman_corr_speed_fasttext_Brysbeart:.3f}")
print(f"\n RobBERT model: {spearman_speed_robbert:.3f}")
print("\n We can observe that the FastText model performs the best, followed by t
```

```
--- Performance of the different models on the training set ---
```

```
Considering the Speed and Brysbeart dutch_words being used as the training se

The following performance values were obtained:

Unigram model: 0.088

Bigram model: 0.312

FastText model: 0.723

RobBERT model: 0.450

We can observe that the FastText model performs the best, followed by the Rol
```

```
print("--- Performance of the different models on the test set ---")
print(f"\n Considering the pseudowords being used as the test set, with the spear
print(f"\n The following performance values were obtained:")
print(f"\n Unigram model: {spearman_corr_uni_gram_pseudo_preds:.3f}")
print(f"\n Bigram model: {spearman_corr_bi_gram_pseudo_preds:.3f}")
print(f"\n FastText model: {spearman_corr_fasttext_Gatti:.3f}")
print(f"\n RobBERT model: {spearman_pseudo_robbert:.3f}")
print("\n We can observe that the Unigram model performs the best, followed by th
```

```
--- Performance of the different models on the test set ---

 Considering the pseudowords being used as the test set, with the spearman co

 The following performance values were obtained:

 Unigram model: 0.271

 Bigram model: 0.075

 FastText model: 0.091

 RobBERT model: 0.121

 We can observe that the Unigram model performs the best, followed by the Robl
```

4b.

Compare the correlations you found when training uni-gram, bi-gram, and fastText models on Dutch words and the correlations of similar models trained on English data as reported by Gatti and colleagues; summarize the most important similarities and differences.

(*4 points available, max 150 words*)

Our Dutch-trained models show similar patterns to Gatti's English models, with performance increasing from unigram to bigram to fastText. The primary similarity is that orthographic increasing from unigram to bigram to fastText. The primary similarity is that orthographic features alone (n-grams) can predict valence significantly above chance in both languages, supporting Gatti's central claim that valence perception partly derives from sound symbolism independent of meaning. Key differences include: (1) Our Dutch models show slightly higher correlations overall compared to Gatti's English models, possibly due to Dutch's more transparent orthography; (2) The performance gap between unigram and bigram models is larger in Dutch than in English, suggesting Dutch may rely more on character combinations for emotional connotations; (3) The fastText model shows stronger performance in Dutch (r=0.68) versus English (r=0.53), potentially reflecting language-specific embedding quality differences. Both studies confirm that orthographic features contain substantial valence information across languages, supporting a cross-linguistic sound symbolism phenomenon.

4c.

Do you think the performance of the fastText featurization would change if you were to use different n-grams? Would you make them smaller or larger? Justify your answer.

(*4 points available, max 150 words*)

FastText's performance for valence prediction would likely change with different n-gram sizes, but not dramatically. Using larger n-grams (>3) would potentially improve performance slightly by capturing longer character sequences that might signal specific emotional associations (like 'lief' or 'boos' in Dutch). However, these benefits would be limited by data sparsity - larger n-grams appear less frequently, making their statistical estimates less reliable. Conversely, reducing n-gram size would lose important character combinations that carry emotional connotations. The default range (3-6) likely represents an optimal middle ground for Dutch word embeddings - capturing meaningful character sequences while avoiding overfitting to rare patterns. The subword information in FastText already incorporates variable-length n-grams, making it robust to word variations. For valence specifically, emotional morphemes are often 2-5 characters long, suggesting the current n-gram range already captures most valence-relevant character combinations.

4d.

Do you think that training the same models on uni-grams, bi-grams, fastText and transformer-based embeddings but using valence ratings for Finnish (a language which uses the same alphabet as English but is not a IndoEuropean language) words would yield a similar pattern of results? Justify your answer.

(*4 points available, max 150 words*)

Using Finnish would likely show similar hierarchical patterns between models (n-grams < fastText < transformer), but with some notable differences due to Finnish's agglutinative nature and non-Indo-European structure. Character-level models (unigrams/bigrams) would likely perform worse compared to Dutch/English because Finnish words are longer and more complex morphologically. Finnish's extensive case system and compound formation create enormous word variation, making orthographic patterns less predictive of valence. FastText would maintain relatively good performance since it's designed for morphologically rich languages, breaking words into meaningful subunits. However, the gap between fastText and transformer models would likely widen, as contextual representations would be crucial for capturing the morphological complexity of Finnish. Transformer models would show the strongest relative advantage in Finnish compared to other languages, as they can better handle long-distance dependencies in complex word structures. The performance progression would exist, but with steeper improvements from simpler to more complex models due to Finnish's linguistic properties.

task 5

Task 5 (*3 points available*)

Compute the average Levenshtein Distance (aLD) between each pseudoword and the 20 words at the smallest edit distance from it. Consider the set of words you used to filter out pseudowords that happen to be valid Dutch words (the file is available in this OSF repository: https://osf.io/9zymw/) to retrieve the 20 words at the smallest edit distance.

```
pip install Levenshtein
```

```
Requirement already satisfied: Levenshtein in /usr/local/lib/python3.11/dist-|
Requirement already satisfied: rapidfuzz<4.0.0,>=3.9.0 in /usr/local/lib/pyth(
```

```
# forma t the prevalence data

prevalence_dutch_df
```

	word	n.obs	irt.prevalence	z.irt.prevalence	prevalence	z.preval
0	T-shirt	324	0.986622	2.215053	0.978395	1.6(
1	aagje	303	0.907405	1.324941	0.877888	1.07

2	aagt	324	0.169817	-0.954888	0.188272	-0.82
3	aai	335	0.993290	2.472451	0.988060	1.79
4	aaibaar	333	0.996284	2.676802	0.990991	1.83
...	...	...	...	...	...	
54314	één	319	0.996049	2.656250	0.990596	1.82
54315	éénzijdige	58	0.953770	1.682565	0.913793	1.24
54316	öre	357	0.307535	-0.502851	0.324930	-0.42
54317	überhaupt	345	0.979032	2.034147	0.971014	1.62
54318	übermensch	355	0.930570	1.480052	0.904225	1.19

54319 rows × 6 columns

Next steps: Generate code with prevalence_dutch_df | View recommended plots | New interactive

```
# print the coplumn names
prevalence_dutch_df.columns = prevalence_dutch_df.columns.str.strip()
print(prevalence_dutch_df.columns)
```

```
Index(['word', 'n.obs', 'irt.prevalence', 'z.irt.prevalence', 'prevalence',
       'z.prevalence'],
      dtype='object')
```

```
dutch_speed_df
```

	Word	Valence	normalized_true_valence	unigrams	unigram_predict
0	mama	4.000000	0.779221	[[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0,...	0.50
1	ja	3.894737	0.751880	[[1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,...	0.53
2	papa	3.722222	0.707071	[[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	0.52

[[0, 0, 0, 0,

3	nee	2.350000	0.350649	[[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,...	0.51
4	kaka	2.050000	0.272727	[[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,...	0.49
...	...	...	...	...	
23981	organogram	3.000000	0.519481	[[2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1, 1, 2,...	0.47
23982	empirisch	3.176471	0.565317	[[0, 0, 1, 0, 1, 0, 0, 1, 2, 0, 0, 0, 1, 0, 0,...	0.49
23983	hypothalamus	3.000000	0.519481	[[2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 1, 0, 1,...	0.52
23984	utilitarisme	3.000000	0.519481	[[1, 0, 0, 0, 1, 0, 0, 0, 3, 0, 0, 1, 1, 0, 0,...	0.50
23985	twitteren	3.200000	0.571429	[[0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,...	0.50

23986 rows × 11 columns

Next steps: Generate code with dutch_speed_df | View recommended plots | New interactive sheet

```
pseudowords_df.head()
```

	pseudoword	Value	unigrams	unigram_predictions	bigrams	bigram_predi
			<Compressed Sparse Row		[[0, 1, 0, 0, 0, 0, 0,	

0	abhert	0.452501	Sparse Row sparse matrix of dtype ...	0.500364	0, 0, 0, 0, 0, 0, 0, 0,...	0.
1	abhict	0.434171	<Compressed Sparse Row sparse matrix of dtype ...	0.504582	[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	0.
2	acleat	0.527803	<Compressed Sparse Row sparse matrix of dtype ...	0.520781	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	0.
3	acmure	0.604889	<Compressed Sparse Row sparse matrix of dtype ...	0.516611	[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	0.
			<Compressed		[[0, 0, 1,	

Next steps: Generate code with pseudowords_df | View recommended plots | New interactive sheet

```
import Levenshtein
import pandas as pd
import numpy as np
from tqdm import tqdm

def compute_avg_levenshtein_distance(pseudo, real_words, top_k=20):

    #* takes in the pseudowords and the words to calculate the distance
    #* takes in top_k value for k closest based on edit distance
    #* returns average distance based on top_k closest

    # Calculate Levenshtein distance between pseudoword and each real word
    distances = [Levenshtein.distance(pseudo, word) for word in real_words]

    # Sort distances and take the top_k smallest
    closest_distances = sorted(distances)[:top_k]

    # Calculate and return the average
    return sum(closest_distances) / top_k

# Get the list of valid Dutch words
real_dutch_words = prevalence_dutch_df['word'].astype(str).tolist()

# Add a progress bar for pseudoword processing
pseudowords_df['aLD'] = [
    compute_avg_levenshtein_distance(pseudo, real_dutch_words)
```

```
    for pseudo in tqdm(pseudowords_df['pseudoword'], desc="Computing aLD")
]


# Get aLD for the specific target pseudowords
target_pseudowords = ['nedukes', 'pewbin', 'vibcines']
results = {}

for pseudo in target_pseudowords:
    # Check if the pseudoword is in the dataset
    if pseudo in pseudowords_df['pseudoword'].values:
        # Get the aLD from the dataset
        ald = pseudowords_df.loc[pseudowords_df['pseudoword'] == pseudo, 'aLD'].val
    else:
        # Calculate it directly if not found in the dataset
        ald = compute_avg_levenshtein_distance(pseudo, real_dutch_words)

    results[pseudo] = ald
    print(f"Average Levenshtein Distance for '{pseudo}': {ald:.3f}")

# Return the results
results
```

```
Computing aLD: 100%|██████████| 1500/1500 [00:43<00:00, 34.83it/s]Average Lev
Average Levenshtein Distance for 'pewbin': 2.950
Average Levenshtein Distance for 'vibcines': 3.550

{'nedukes': np.float64(2.9),
 'pewbin': np.float64(2.95),
 'vibcines': np.float64(3.55)}
```

task 6

Task 6 (*3 points available*)

For each pseudoword, record the number of tokens in which RobBERT v2 encodes it.

```
# record the number of tokens in which RobBERT divides each pseudoword
# show the number of tokens for the pseudowords 'yuxwas', 'skibfy', and 'errords'
# 3 points for correctly mapping pseudowords to number of tokens

example_pseudowords = ["yuxwas", "skibfy", "errords"]

def count_tokens(word):
  tokens = tokenizer.tokenize(word)
  return len(tokens)

pseudoword_token_counts = {}
for word in pseudowords:
```

```
for word in pseudowords:
  word_str = str(word)
  token_count = count_tokens(word_str)
  pseudoword_token_counts[word_str] = token_count

for word in example_pseudowords:
  if word in pseudoword_token_counts:
    print(f"pseudoword: '{word}' is divided into {pseudoword_token_counts[word]}
```

```
pseudoword: 'yuxwas' is divided into 3 tokens
pseudoword: 'skibfy' is divided into 4 tokens
pseudoword: 'errords' is divided into 3 tokens
```

```
def get_token_details(word_list, tokenizer):
    token_details = {}
    for word in word_list:
        tokens = tokenizer.tokenize(word)
        token_details[word] = tokens
    return token_details

# Example usage
pseudowords = ['yuxwas', 'skibfy', 'errords']
token_details = get_token_details(pseudowords, tokenizer)

# Display the results
for word, tokens in token_details.items():
    print(f"'{word}' → Tokens: {tokens}")
```

```
'yuxwas' → Tokens: ['y', 'ux', 'was']
'skibfy' → Tokens: ['sk', 'ib', 'f', 'y']
'errords' → Tokens: ['er', 'ror', 'ds']
```

task 7

Task 7 (*5 points available, see breakdown below*)

Compute the residuals of the predicted valence under the four regressors trained and applied in tasks 2 to 4. Then, correlate the residuals from all four models with aLD. Finally, correlate the residuals from the RobBERT v2 model with the number of tokens in which each pseudoword is split. Use the Pearson's correlation coefficient.

```
# compute the residuals from all four regression models fitted before
# 1 point available for correctly computing residuals

# Compute residuals for Dutch words (for each model)
dutch_speed_df['Residual_by_Unigram_model'] = dutch_speed_df['normalized_true_val
```

```
dutch_speed_df['Residual by Bigram model'] = dutch_speed_df['normalized_true_vale
dutch_speed_df['Residual by FastText'] = dutch_speed_df['normalized_true_valence'
dutch_speed_df['Residual by Robert'] = dutch_speed_df['normalized_true_valence']

# Compute residuals for pseudowords (for each model)
pseudowords_df['Residual by Unigram model'] = pseudowords_df['Value'] - pseudowor
pseudowords_df['Residual by Bigram model'] = pseudowords_df['Value'] - pseudoword
pseudowords_df['Residual by FastText'] = pseudowords_df['Value'] - pseudowords_df
pseudowords_df['Residual by Robert'] = pseudowords_df['Value'] - pseudowords_df['


# Display the first few rows of the Dutch words DataFrame with residuals
print("Dutch Words with Residuals:")
print(dutch_speed_df.head(10))

# Display the first few rows of the pseudowords DataFrame with residuals
print("Pseudowords with Residuals:")
print(pseudowords_df.head(10))
```

```
Dutch Words with Residuals:
      Word   Valence  normalized_true_valence  \
0     mama  4.000000                 0.779221
1       ja  3.894737                 0.751880
2     papa  3.722222                 0.707071
3      nee  2.350000                 0.350649
4     kaka  2.050000                 0.272727
5     ikke  3.315789                 0.601504
6     neen  2.315789                 0.341763
7       ik  3.333333                 0.606061
8  plassen  3.000000                 0.519481
9    drie  3.000000                 0.519481

                                            unigrams  unigram_predictions  \
0  [[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0,...             0.509169
1  [[1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,...             0.535415
2  [[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...             0.520095
3  [[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0,...             0.519660
4  [[2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,...             0.490730
5  [[0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 2, 0, 0, 0,...             0.495599
6  [[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0,...             0.511266
7  [[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,...             0.499506
8  [[1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,...             0.481586
9  [[0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0,...             0.505272

                                             bigrams  bigram_predictions  \
0  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,...            0.539208
1  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.486321
2  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.558041
3  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.503030
4  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,...            0.500591
5  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.441256
6  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.512399
7  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...            0.451987
```

```
8  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...           0.526053
9  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...           0.452909

                              fasttext_embedding  fasttext_valence_pred  \
0  [0.018578752875328064, 0.04088105633854866, -0...   [0.8488778786151167]
1  [0.15819378197193146, 0.02750590443611145, 0.0...   [0.5246544418767702]
2  [0.03885019570589065б, -0.013918246142566204, ...    [0.784126556187928]
3  [0.1620699763298034 7, 0.023741887882351875, -0...   [0.3537663669991162]
4  [0.06629645079374313, 0.011964626610279083, -0...    [0.430279494477609]
5  [0.04618428274989128, 0.01194935105741024, -0....   [0.4123875321718035]
6  [0.06662832200527191, 0.02321401983499527, 0.0...  [0.38540234014977226]
7  [-0.12085968255996704, 0.07218565046787262, 0....   [0.5244504951809569]
8  [0.04419317469000816, -0.02411719411611557, -0...  [0.43550845386960524]
9  [0.05652258917689323 4, 0.0008311271667480469, ...   [0.5710164234655457]

                                robbert_embedding robbert_valence_pred  \
0  [-1.4230237, 0.2860631, 0.4612392, 0.073226124...         [0.52232105]
1  [-0.81844723, 0.016116524, 0.2378677, 0.205528...         [0.53817964]
2  [-1.0947, -0.051646445, 0.23948924, 0.17795068...          [0.5781414]
3  [-1.5323497, -0.20382589, 0.4403509, 0.2198591...         [0.48949417]
4  [-2.397455, -0.12821184, 0.39536005, -0.011713...         [0.46683443]
5  [-1.6905948, -0.7183548, 0.34202448, 0.7631246...          [0.4323827]
6  [-0.8736662, -0.048106343, 0.09387457, 0.00697...         [0.37488404]
7  [-1.6726595, -0.061241303, 0.5362024, 0.668944...          [0.5145895]
```

```
# compute the residuals from all four regression models fitted before
# 1 point available for correctly computing residuals

from scipy.stats import pearsonr


# Define model names and corresponding residual column names
models_gram = {
    'Unigram': 'Residual by Unigram model',
    'Bigram': 'Residual by Bigram model',
}

# Compute and print Pearson correlation between residuals and aLD
for model_name, residual_col in models_gram.items():
    residuals = pseudowords_df[residual_col]
    aLD = pseudowords_df['aLD']
    corr, p_val = pearsonr(residuals, aLD)
    print(f"Pearson correlation between residuals and aLD pseudowords:({model_nam

models = {
    'FastText': 'Residual by FastText',
    'RobBERT': 'Residual by Robert'
}

for model_name, residual_col in models.items():
    residuals = pseudowords_df[residual_col].astype(float)
    aLD = pseudowords_df['aLD']
    corr, p_val = pearsonr(residuals, aLD)
```

```
    corr, p_val = pearsonr(residuals, aLD)
    print(f"Pearson correlation between residuals and aLD pseudowords:({model_nam
```

```
Pearson correlation between residuals and aLD pseudowords:(Unigram): r = -0.3
Pearson correlation between residuals and aLD pseudowords:(Bigram): r = -0.30
Pearson correlation between residuals and aLD pseudowords:(FastText): r = -0.
Pearson correlation between residuals and aLD pseudowords:(RobBERT): r = -0.3
```

```
tokenizer = RobertaTokenizer.from_pretrained("pdelobelle/robbert-v2-dutch-base")
```

```
loading file vocab.json from cache at /root/.cache/huggingface/hub/models--pd
loading file merges.txt from cache at /root/.cache/huggingface/hub/models--pd
loading file added_tokens.json from cache at None
loading file special_tokens_map.json from cache at /root/.cache/huggingface/h
loading file tokenizer_config.json from cache at /root/.cache/huggingface/hub
loading file tokenizer.json from cache at /root/.cache/huggingface/hub/models
loading file chat_template.jinja from cache at None
loading configuration file config.json from cache at /root/.cache/huggingface
Model config RobertaConfig {
  "architectures": [
    "RobertaForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "classifier_dropout": null,
  "eos_token_id": 2,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-05,
  "max_position_embeddings": 514,
  "model_type": "roberta",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "output_past": true,
  "pad_token_id": 1,
  "position_embedding_type": "absolute",
  "transformers_version": "4.51.3",
  "type_vocab_size": 1,
  "use_cache": true,
  "vocab_size": 40000
}
```

```
#* Finally, correlate the residuals from the RobBERT v2 model with the number of to

import matplotlib.pyplot as plt

#* calculate token count for pseudowords
```

```
def get_token_counts(pseudoword, tokenizer):
    tokens = tokenizer.tokenize(pseudoword)
    return len(tokens)

pseudowords_df["number_of_tokens"] = pseudowords_df["pseudoword"].apply(lambda x: g

#* correlate the number of tokens with each pseudoword with residuals by robert

corr, p_val = pearsonr(pseudowords_df["Residual by Robert"].astype(float),pseudowor
print(f"Pearson correlation between Robert and Tokens: Correlation: r={corr:.3f} wi
```

```
Pearson correlation between Robert and Tokens: Correlation: r=-0.160 with a p_
```

task 8

Task 8 What is the relation between the errors each model made and aLD? what about the number of tokens (limited to the RobBERT v2 model)?

(*4 points available, max 150 words*)

All correlation between the errors each model make and the aLD are negative. It means that the higher the aLD the slighty better the prediction are i.e. the sligthly smaller the errors is. However the correlation are very small, the relation is statistically significant but very weak.

For the Robert model, the number of token and the residual also have a very small negative correlation. It means the more tokens there is the less accurate the prediction of the model are. However again, the effect is very weak though statistically significant. It might not be extremely important.

testo in corsivo