DA5401 - Assignment 6

September 21, 2024

Name: Anirudh Rao

Roll No.: BE21B004

We will first have to install the `datasets` library from HuggingFace.

[1]:
```
!pip install transformers datasets
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-
packages (4.44.2)
Collecting datasets
  Downloading datasets-3.0.0-py3-none-any.whl.metadata (19 kB)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers) (3.16.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.23.2 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.24.7)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-
packages (from transformers) (1.26.4)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (24.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2024.9.11)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from transformers) (2.32.3)
Requirement already satisfied: safetensors>=0.4.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.5)
Requirement already satisfied: tokenizers<0.20,>=0.19 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.19.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-
packages (from transformers) (4.66.5)
Collecting pyarrow>=15.0.0 (from datasets)
  Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl.metadata (3.3
kB)
Collecting dill<0.3.9,>=0.3.0 (from datasets)
  Downloading dill-0.3.8-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages
```

```
(from datasets) (2.1.4)
Collecting xxhash (from datasets)
  Downloading
xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata
(12 kB)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.16-py310-none-any.whl.metadata (7.2 kB)
Requirement already satisfied: fsspec<=2024.6.1,>=2023.1.0 in
/usr/local/lib/python3.10/dist-packages (from
fsspec[http]<=2024.6.1,>=2023.1.0->datasets) (2024.6.1)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-
packages (from datasets) (3.10.5)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (2.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->datasets) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (6.1.0)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->datasets) (1.11.1)
Requirement already satisfied: async-timeout<5.0,>=4.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (4.0.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.23.2->transformers) (4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2024.8.30)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas->datasets) (2024.2)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-
packages (from pandas->datasets) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.8.2->pandas->datasets) (1.16.0)
Downloading datasets-3.0.0-py3-none-any.whl (474 kB)
                         474.3/474.3 kB
```

```
31.2 MB/s eta 0:00:00
Downloading dill-0.3.8-py3-none-any.whl (116 kB)
                         116.3/116.3 kB
10.4 MB/s eta 0:00:00
Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl (39.9 MB)
                         39.9/39.9 MB
22.3 MB/s eta 0:00:00
Downloading multiprocess-0.70.16-py310-none-any.whl (134 kB)
                         134.8/134.8 kB
9.9 MB/s eta 0:00:00
Downloading
xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (194 kB)
                         194.1/194.1 kB
13.0 MB/s eta 0:00:00
Installing collected packages: xxhash, pyarrow, dill, multiprocess,
datasets
  Attempting uninstall: pyarrow
    Found existing installation: pyarrow 14.0.2
    Uninstalling pyarrow-14.0.2:
      Successfully uninstalled pyarrow-14.0.2
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
cudf-cu12 24.4.1 requires pyarrow<15.0.0a0,>=14.0.1, but you have pyarrow 17.0.0
which is incompatible.
ibis-framework 8.0.0 requires pyarrow<16,>=2, but you have pyarrow 17.0.0 which
is incompatible.
Successfully installed datasets-3.0.0 dill-0.3.8 multiprocess-0.70.16
pyarrow-17.0.0 xxhash-3.5.0
```

We will also import some additional libraries.

[2]:
```python
# Importing necessary libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[3]:
```python
# Ignoring any warnings that arise

import warnings
warnings.filterwarnings("ignore")
```

1 Task 1

We list the languages that use a Roman script. These will be used as the target labels for classification.

[4]:
```
# Languages with a Roman script

languages =␣
↪["af-ZA","da-DK","de-DE","en-US","es-ES","fr-FR","fi-FI","hu-HU","is-IS","it-IT","jv-ID","l
```

[5]:
```
from datasets import load_dataset
```

For each language, we will create a .txt file containing all the sentences in that language from the MASSIVE dataset. We will not deaccent any characters as this may hinder our algorithm from disciminating between languages.

[6]:
```
# Creating .txt files with sentences from each language

for language in languages:
  print(language)
  dataset = load_dataset("qanastek/MASSIVE",language)
  dummy_df = pd.concat([dataset["train"].
 ↪to_pandas()["utt"],dataset["validation"].to_pandas()["utt"],dataset["test"].
 ↪to_pandas()["utt"]])
  dummy_df.to_csv(f"{language}.txt",index=False,header=False,sep="\t")
```

```
af-ZA

MASSIVE.py:   0%|          | 0.00/32.3k [00:00<?, ?B/s]

README.md:   0%|          | 0.00/34.1k [00:00<?, ?B/s]

The repository for qanastek/MASSIVE contains custom code which must be executed
to correctly load the dataset. You can inspect the repository content at
https://hf.co/datasets/qanastek/MASSIVE.
You can avoid this prompt in future by passing the argument
`trust_remote_code=True`.

Do you wish to run the custom code? [y/N] y

Downloading data:   0%|          | 0.00/39.5M [00:00<?, ?B/s]

Generating train split: 0 examples [00:00, ? examples/s]

Generating validation split: 0 examples [00:00, ? examples/s]

Generating test split: 0 examples [00:00, ? examples/s]

da-DK

Generating train split: 0 examples [00:00, ? examples/s]

Generating validation split: 0 examples [00:00, ? examples/s]
```

```
Generating test split: 0 examples [00:00, ? examples/s]
de-DE
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
en-US
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
es-ES
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
fr-FR
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
fi-FI
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
hu-HU
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
is-IS
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
it-IT
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
```

```
Generating test split: 0 examples [00:00, ? examples/s]
jv-ID
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
lv-LV
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
ms-MY
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
nb-NO
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
nl-NL
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
pl-PL
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
pt-PT
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
ro-RO
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
```

```
Generating test split: 0 examples [00:00, ? examples/s]
ru-RU
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
sl-SL
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
sv-SE
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
sq-AL
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
sw-KE
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
tl-PH
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
tr-TR
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
Generating test split: 0 examples [00:00, ? examples/s]
vi-VN
Generating train split: 0 examples [00:00, ? examples/s]
Generating validation split: 0 examples [00:00, ? examples/s]
```

```
Generating test split: 0 examples [00:00, ? examples/s]

cy-GB

Generating train split: 0 examples [00:00, ? examples/s]

Generating validation split: 0 examples [00:00, ? examples/s]

Generating test split: 0 examples [00:00, ? examples/s]
```

2 Task 2

We combine the sentences from the files created into train, validation, and test sets based on their partition in the original MASSIVE dataset.

[7]:
```
# Defining train, validation, and test sets

train_df = pd.DataFrame()
val_df = pd.DataFrame()
test_df = pd.DataFrame()

train_size = 11514
val_size = 2033
test_size = 2974

for language in languages:
  f = open(f"{language}.txt","r")
  sentences = [line.strip() for line in f.readlines()]
  f.close()
  dummy_df = pd.DataFrame()
  dummy_df["utt"] = sentences
  dummy_df["language"] = language
  train_df = pd.concat([train_df,dummy_df.iloc[:train_size]])
  val_df = pd.concat([val_df,dummy_df.iloc[train_size:train_size+val_size]])
  test_df = pd.concat([test_df,dummy_df.iloc[train_size+val_size:]])
```

To predict which language a sentence is in, we will use a Multinomial Naive Bayes model that makes use of word frequencies computed by a `TfidfVectorizer`.

[8]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfVectorizer
```

[9]:
```
# Performing TFIDF vectorization of the sentences

vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(train_df["utt"])
X_val = vectorizer.transform(val_df["utt"])
X_test = vectorizer.transform(test_df["utt"])
```

[10]:
```python
X_train.shape
```

[10]: (310878, 152098)

[11]:
```python
# Training the Multinomial Naive Bayes algorithm

model = MultinomialNB()
model.fit(X_train,train_df["language"])
```

[11]: MultinomialNB()

We now evaluate the baseline performance of the model. Based on this, we can fine-tune using the validation set.

[12]:
```python
# Importing classification metrics

from sklearn.metrics import accuracy_score, precision_score, recall_score
```

[13]:
```python
print(f"Baseline validation accuracy: {accuracy_score(val_df['language'],model.
 ↪predict(X_val))}")
print(f"Baseline validation precision:␣
 ↪{precision_score(val_df['language'],model.predict(X_val),average='macro')}")
print(f"Baseline validation recall: {recall_score(val_df['language'],model.
 ↪predict(X_val),average='macro')}")
```

```
Baseline validation accuracy: 0.9842050609389518
Baseline validation precision: 0.9845450428266429
Baseline validation recall: 0.9842050609389518
```

The baseline performance metrics are already quite high. We can see if we can improve this by using a different Laplace smoothing factor α.

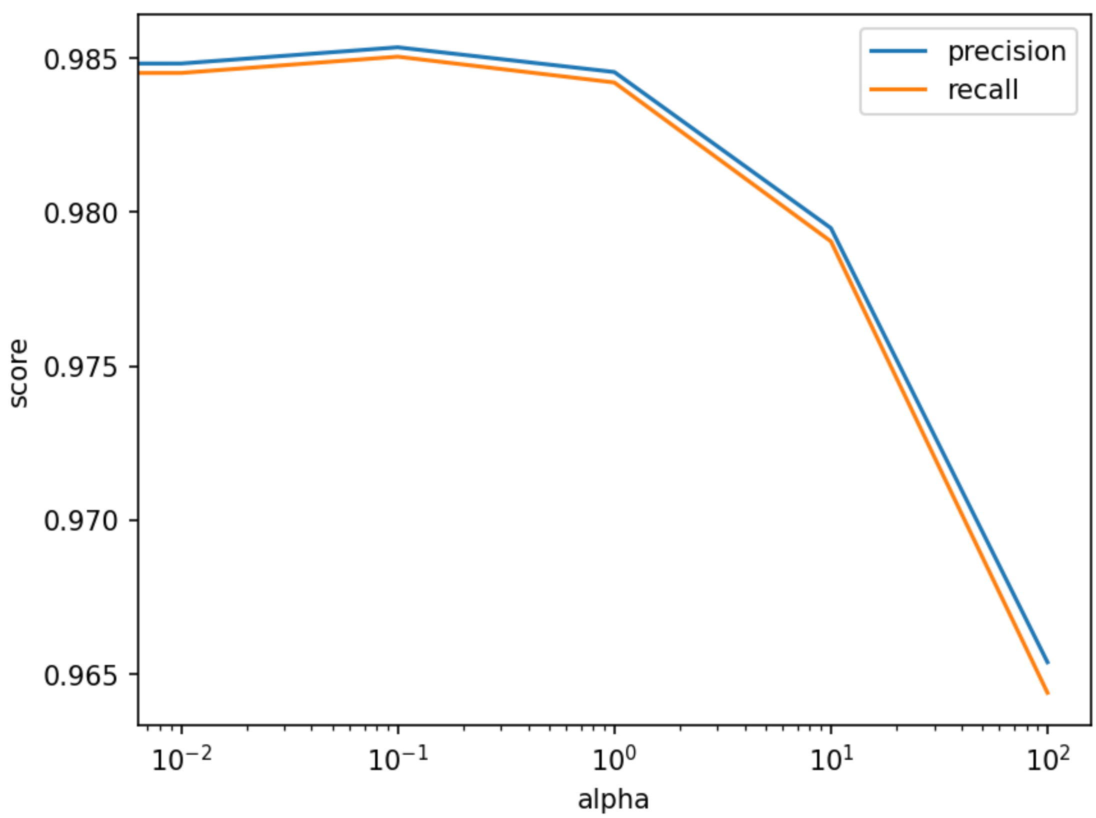
[14]:
```python
# Using the validation set to fine-tune the hyperparameter of the model

performance = pd.DataFrame()

alphas = [0,0.01,0.1,1,10,100]

for alpha in alphas:
  model = MultinomialNB(alpha=alpha)
  model.fit(X_train,train_df["language"])
  predictions = model.predict(X_val)
  accuracy = accuracy_score(val_df["language"],predictions)
  precision = precision_score(val_df["language"],predictions,average="macro")
  recall = recall_score(val_df["language"],predictions,average="macro")
  performance = pd.concat([performance,pd.DataFrame({"alpha":alpha,"accuracy":
 ↪accuracy,"precision":precision,"recall":recall},index=[alphas.
 ↪index(alpha)])])
```

[15]:
```
performance
```

[15]:

	alpha	accuracy	precision	recall
0	0.00	0.982055	0.982417	0.982055
1	0.01	0.984515	0.984821	0.984515
2	0.10	0.985043	0.985347	0.985043
3	1.00	0.984205	0.984545	0.984205
4	10.00	0.979049	0.979481	0.979049
5	100.00	0.964402	0.965395	0.964402

[16]:
```
plt.figure(dpi=150)
plt.plot(performance["alpha"],performance["precision"],label="precision")
plt.plot(performance["alpha"],performance["recall"],label="recall")
plt.xlabel("alpha")
plt.ylabel("score")
plt.xscale("log")
plt.legend()
plt.show()
```



[17]:
```
best_alpha = performance.loc[performance["precision"].idxmax()]["alpha"]
best_alpha
```

[17]: 0.1

The best value of α to use is found to be 0.1. We now evaluate the performance of the model using this α.

[19]:
```
# Final performance metrics

best_model = MultinomialNB(alpha=best_alpha)
best_model.fit(X_train,train_df["language"])

print("Train set performance")
print(f"Accuracy: {accuracy_score(train_df['language'],best_model.
  ↪predict(X_train))}")
print(f"Precision: {precision_score(train_df['language'],best_model.
  ↪predict(X_train),average='macro')}")
print(f"Recall: {recall_score(train_df['language'],best_model.
  ↪predict(X_train),average='macro')}")
print("\nValidation set performance")
print(f"Accuracy: {accuracy_score(val_df['language'],best_model.
  ↪predict(X_val))}")
print(f"Precision: {precision_score(val_df['language'],best_model.
  ↪predict(X_val),average='macro')}")
print(f"Recall: {recall_score(val_df['language'],best_model.
  ↪predict(X_val),average='macro')}")
print(f"\nTest set performance")
print(f"Accuracy: {accuracy_score(test_df['language'],best_model.
  ↪predict(X_test))}")
print(f"Precision: {precision_score(test_df['language'],best_model.
  ↪predict(X_test),average='macro')}")
print(f"Recall: {recall_score(test_df['language'],best_model.
  ↪predict(X_test),average='macro')}")
```

```
Train set performance
Accuracy: 0.9932803221842652
Precision: 0.9933001545353906
Recall: 0.9932803221842653

Validation set performance
Accuracy: 0.9850430853874041
Precision: 0.9853465715773398
Recall: 0.985043085387404

Test set performance
Accuracy: 0.9847443273805077
Precision: 0.9852551769764829
Recall: 0.9847443273805077
```

The model performs quite well on all three partitions. There is only a marginal improvement from the baseline model by changing the hyperparameter `alpha`.

3 Task 3

We now combine languages based on their continent of origin. Our goal is now to predict the continent from the sentence.

[20]:
```
# Combining languages from different continents

asia_languages = ["jv-ID","ms-MY","tl-PH","vi-VN"]
africa_languages = ["af-ZA","sw-KE"]
europe_languages =␣
↪["da-DK","de-DE","es-ES","fr-FR","fi-FI","hu-HU","is-IS","it-IT","lv-LV","nb-NO","nl-NL","p
north_america_languages = ["en-US"]
```

We will create a .txt file for each continent.

[21]:
```
# Creating .txt files for each continent

asia_df = pd.DataFrame()
africa_df = pd.DataFrame()
europe_df = pd.DataFrame()
north_america_df = pd.DataFrame()

for language in languages:

  continent = "asia" if language in asia_languages else "africa" if language in␣
↪africa_languages else "europe" if language in europe_languages else␣
↪"north_america"
  f = open(f"{language}.txt","r")
  sentences = [line.strip() for line in f.readlines()]
  f.close()
  dummy_df = pd.DataFrame()
  dummy_df["utt"] = sentences

  if continent == "asia":
    asia_df = pd.concat([asia_df,dummy_df])
  elif continent == "africa":
    africa_df = pd.concat([africa_df,dummy_df])
  elif continent == "europe":
    europe_df = pd.concat([europe_df,dummy_df])
  else:
    north_america_df = pd.concat([north_america_df,dummy_df])

asia_df.to_csv("asia.txt",index=False,header=False,sep="\t")
africa_df.to_csv("africa.txt",index=False,header=False,sep="\t")
europe_df.to_csv("europe.txt",index=False,header=False,sep="\t")
north_america_df.to_csv("north_america.txt",index=False,header=False,sep="\t")
```

Using the .txt files created, we define the train, validation, and test sets based on the original partitions in MASSIVE.

[22]:
```
# Defining the train, validation, and test sets

train_df = pd.DataFrame()
val_df = pd.DataFrame()
test_df = pd.DataFrame()

for continent in ["asia","africa","europe","north_america"]:
  f = open(f"{continent}.txt","r")
  content = f.readlines()
  sentences = [line.strip() for line in content]
  f.close()

  languages = asia_languages if continent == "asia" else africa_languages if␣
↪continent == "africa" else europe_languages if continent == "europe" else␣
↪north_america_languages

  for i in range(len(languages)):
    dummy_df = pd.DataFrame()
    dummy_df["utt"] = sentences[i*(train_size+val_size+test_size):
↪(i+1)*(train_size+val_size+test_size)]
    dummy_df["continent"] = continent

    train_df = pd.concat([train_df,dummy_df.iloc[:train_size]])
    val_df = pd.concat([val_df,dummy_df.iloc[train_size:train_size+val_size]])
    test_df = pd.concat([test_df,dummy_df.iloc[train_size+val_size:]])
```

For this classification tasks, we will use Regularized Discriminant Analysis, a combination of Linear and Quadratic Discriminant Analysis. The class probabilities from the two models will be combined using a hyperparameter λ as $ P_{RDA} = P_{LDA} + (1-) P_{QDA}$. Here $\lambda \in [0, 1]$. The class with the highest P_{RDA} will be returned as the prediction.

[23]:
```
# Importing LDA and QDA from sklearn

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
↪QuadraticDiscriminantAnalysis
```

[24]:
```
# Defining the regularized discriminant analysis model

def RDA_predict(X,LDA_model,QDA_model,param=1):

  LDA_prob = LDA_model.predict_proba(X)
  QDA_prob = QDA_model.predict_proba(X)
  RDA_prob = param*LDA_prob + (1-param)*QDA_prob

  return LDA_model.classes_[np.argmax(RDA_prob,axis=1)]
```

To reduce the training time of the disciminant models, we use low frequency pruning to limit the number of words in the feature space.

[25]:
```
# TF-IDF vectorization with low frequency pruning

vectorizer = TfidfVectorizer(min_df=1e-3)
X_train = vectorizer.fit_transform(train_df["utt"])
X_val = vectorizer.transform(val_df["utt"])
X_test = vectorizer.transform(test_df["utt"])
```

[26]:
```
X_train.shape
```

[26]:
```
(310878, 856)
```

We can see that the number of features has drastically come down.

[27]:
```
# We define and fit the LDA and QDA models. It should be noted that LDA takes a␣
↪long time to fit.

LDA = LinearDiscriminantAnalysis()
QDA = QuadraticDiscriminantAnalysis()

LDA.fit(X_train.toarray(),train_df["continent"])
QDA.fit(X_train.toarray(),train_df["continent"])
```

[27]:
```
QuadraticDiscriminantAnalysis()
```

We test the performance of the RDA model for different values of λ.

[28]:
```
# Testing RDA performance for different lambdas

performance = pd.DataFrame()

lambdas = [0,0.25,0.5,0.75,1]

for param in lambdas:
  predictions = RDA_predict(X_val.toarray(),LDA,QDA,param)
  accuracy = accuracy_score(val_df["continent"],predictions)
  precision = precision_score(val_df["continent"],predictions,average="macro")
  recall = recall_score(val_df["continent"],predictions,average="macro")
  performance = pd.concat([performance,pd.DataFrame({"lambda":param,"accuracy":
↪accuracy,"precision":precision,"recall":recall},index=[lambdas.
↪index(param)])])
```

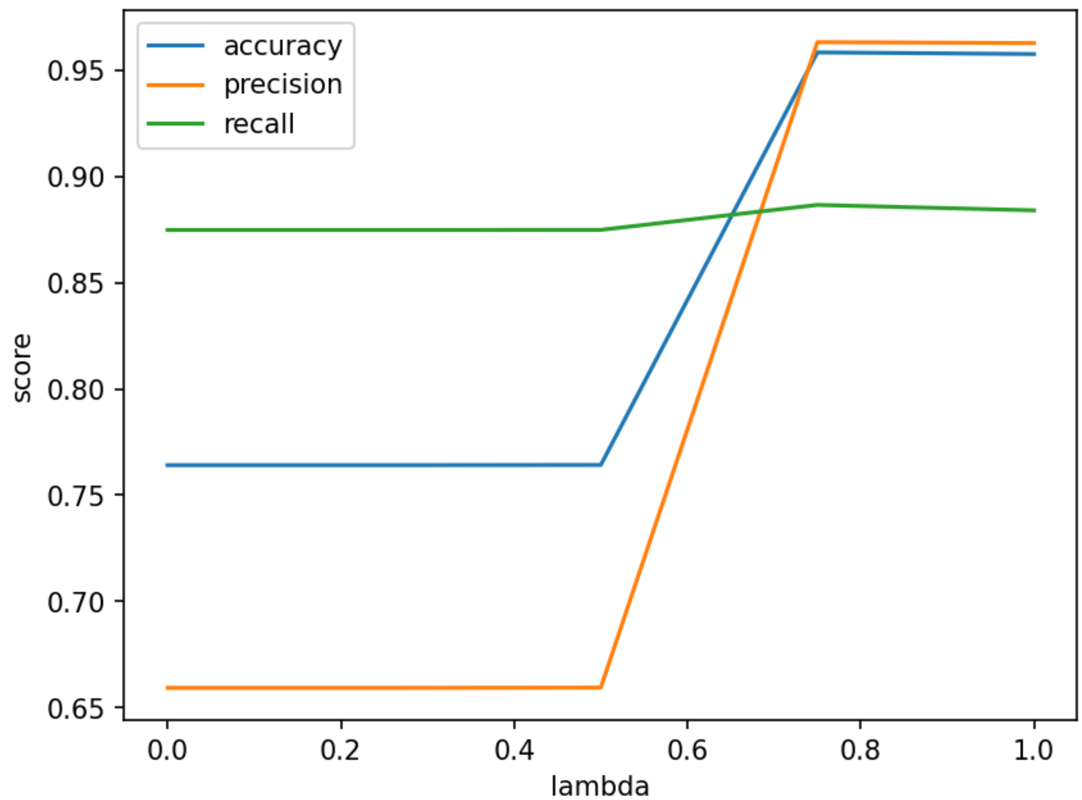
[29]:
```
performance
```

[29]:

	lambda	accuracy	precision	recall
0	0.00	0.764133	0.659306	0.874834

```
1    0.25  0.764133   0.659306  0.874834
2    0.50  0.764224   0.659434  0.874865
3    0.75  0.958408   0.963258  0.886621
4    1.00  0.957625   0.962781  0.884075
```

[30]:
```python
plt.figure(dpi=150)
plt.plot(performance["lambda"],performance["accuracy"],label="accuracy")
plt.plot(performance["lambda"],performance["precision"],label="precision")
plt.plot(performance["lambda"],performance["recall"],label="recall")
plt.xlabel("lambda")
plt.ylabel("score")
plt.legend()
plt.show()
```



[31]:
```python
best_lambda = performance.loc[performance["precision"].idxmax()]["lambda"]
best_lambda
```

[31]: 0.75

We can see that the performance is best when $\lambda = 0.75$. Using this, we report the final metrics using the validation and test sets.

[32]:
```
print("Validation performance")
print(f"Accuracy:␣
 ↪{float(performance[performance['lambda']==best_lambda]['accuracy'])}")
print(f"Precision:␣
 ↪{float(performance[performance['lambda']==best_lambda]['precision'])}")
print(f"Recall:␣
 ↪{float(performance[performance['lambda']==best_lambda]['recall'])}")
```

```
Validation performance
Accuracy: 0.9584084822648522
Precision: 0.963257742666094
Recall: 0.8866207575012297
```

[33]:
```
test_predictions = RDA_predict(X_test.toarray(),LDA,QDA,best_lambda)
print("Test performance")
print(f"Accuracy: {accuracy_score(test_df['continent'],test_predictions)}")
print(f"Precision:␣
 ↪{precision_score(test_df['continent'],test_predictions,average='macro')}")
print(f"Recall:␣
 ↪{recall_score(test_df['continent'],test_predictions,average='macro')}")
```

```
Test performance
Accuracy: 0.9576328177538669
Precision: 0.9614559847380274
Recall: 0.8838937457969065
```

The RDA model is able to predict the continent of origin quite well, even with low frequency pruning.