# Natural Language Processing Lab

# Exercise 5

## Hate Speech Identification

## 1. Develop a model for Named Entity Recognition using Hidden Markov Model.

```
# !pip install hmmlearn==0.2.6
```

```
Defaulting to user installation because normal site-packages is not
writeable
Collecting hmmlearn==0.2.6
  Downloading hmmlearn-0.2.6-cp39-cp39-
manylinux_2_5_x86_64.manylinux1_x86_64.whl (369 kB)
ent already satisfied: numpy>=1.10 in
/opt/anaconda3/lib/python3.9/site-packages (from hmmlearn==0.2.6)
(1.22.4)
Requirement already satisfied: scipy>=0.19 in
/opt/anaconda3/lib/python3.9/site-packages (from hmmlearn==0.2.6)
(1.7.1)
Requirement already satisfied: scikit-learn>=0.16 in
/opt/anaconda3/lib/python3.9/site-packages (from hmmlearn==0.2.6)
(0.24.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/opt/anaconda3/lib/python3.9/site-packages (from scikit-learn>=0.16-
>hmmlearn==0.2.6) (2.2.0)
Requirement already satisfied: joblib>=0.11 in
/opt/anaconda3/lib/python3.9/site-packages (from scikit-learn>=0.16-
>hmmlearn==0.2.6) (1.1.0)
Installing collected packages: hmmlearn
Successfully installed hmmlearn-0.2.6
```

```
import hmmlearn
from hmmlearn import hmm
```

```
# !pip show hmmlearn  # check that the version installed is 0.2.6
```

```
Name: hmmlearn
Version: 0.2.6
Summary: Hidden Markov Models in Python with scikit-learn like API
Home-page: https://github.com/hmmlearn/hmmlearn
Author:
Author-email:
License: new BSD
Location: /home/ai_ds_a1/.local/lib/python3.9/site-packages
```

```
Requires: scikit-learn, numpy, scipy
Required-by:
```

```python
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g.
pd.read_csv)
import seaborn as sns
from tqdm import tqdm
from matplotlib import pyplot as plt  # show graph
import random

#some other libraries
import re
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')

from typing import List

from sklearn.model_selection import GroupShuffleSplit
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score, precision_score, recall_score, \
    f1_score, roc_auc_score
```

```
Matplotlib is building the font cache; this may take a moment.
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/ai_ds_a1/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```python
data = pd.read_csv("ner_dataset.csv", encoding='latin1')

data = data.fillna(method="ffill")

data = data.rename(columns={'Sentence #': 'sentence'})
data.head(5)
```

```
      sentence           Word  POS Tag
0  Sentence: 1      Thousands  NNS   O
1  Sentence: 1             of   IN   O
2  Sentence: 1  demonstrators  NNS   O
3  Sentence: 1           have  VBP   O
4  Sentence: 1        marched  VBN   O
```

```python
def pre_processing(text_column):
    # lowercase all text in the column
    text_column = text_column.str.lower()

    # replacing numbers with NUM token
    text_column = text_column.str.replace(r'\d+', 'NUM')

    # removing stopwords
```

```
    stop_words = set(stopwords.words('english'))
    text_column = text_column.apply(lambda x: ' '.join([word for word
in x.split() if word not in stop_words]))

    return text_column



data_pre_precessed = pre_processing(data.Word)
```

```
/tmp/ipykernel_8512/3849771148.py:6: FutureWarning: The default value
of regex will change from True to False in a future version.
  text_column = text_column.str.replace(r'\d+', 'NUM')
```

```
data_pre_precessed.head(20)
```

```
0         thousands
1
2     demonstrators
3
4           marched
5
6            london
7
8           protest
9
10              war
11
12             iraq
13
14           demand
15
16       withdrawal
17
18          british
19           troops
Name: Word, dtype: object
```

```
data_processed = data
data_processed['Word'] = data_pre_precessed

#removing the rows where word is empty
data_processed = data_processed[(data_processed['Word'] != '') |
(data_processed['Word'].isna())]



data_processed.head(20)
```

```
        sentence            Word  POS    Tag
0   Sentence: 1       thousands  NNS      O
2   Sentence: 1  demonstrators  NNS      O
4   Sentence: 1         marched  VBN      O
6   Sentence: 1          london  NNP  B-geo
8   Sentence: 1         protest   VB      O
10  Sentence: 1             war   NN      O
12  Sentence: 1            iraq  NNP  B-geo
14  Sentence: 1          demand   VB      O
16  Sentence: 1      withdrawal   NN      O
18  Sentence: 1         british   JJ  B-gpe
19  Sentence: 1          troops  NNS      O
22  Sentence: 1         country   NN      O
23  Sentence: 1               .    .      O
24  Sentence: 2        families  NNS      O
26  Sentence: 2        soldiers  NNS      O
27  Sentence: 2          killed  VBN      O
30  Sentence: 2        conflict   NN      O
31  Sentence: 2          joined  VBD      O
33  Sentence: 2      protesters  NNS      O
35  Sentence: 2         carried  VBD      O
```

```
tags = list(set(data.POS.values))  # Unique POS tags in the dataset
words = list(set(data.Word.values))  # Unique words in the dataset
len(tags), len(words)
```

```
(42, 29764)
```

```
words1 = list(set(data_processed.Word.values))  # Unique words in the
dataset
len(words1)
```

```
29763
```

```
y = data.POS
X = data.drop('POS', axis=1)

gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix, test_ix = next(gs.split(X, y, groups=data['sentence']))

data_train = data.loc[train_ix]
data_test = data.loc[test_ix]



data_train.head(5)
```

```
        sentence      Word  POS Tag
24  Sentence: 2  families  NNS   O
25  Sentence: 2             IN   O
```

| | | | | |
|---|---|---|---|---|
| 26 | Sentence: 2 | soldiers | NNS | O |
| 27 | Sentence: 2 | killed | VBN | O |
| 28 | Sentence: 2 | | IN | O |

```
data_test.head(5)
```

| | sentence | Word | POS | Tag |
|---|---|---|---|---|
| 0 | Sentence: 1 | thousands | NNS | O |
| 1 | Sentence: 1 | | IN | O |
| 2 | Sentence: 1 | demonstrators | NNS | O |
| 3 | Sentence: 1 | | VBP | O |
| 4 | Sentence: 1 | marched | VBN | O |

```
#using preprocessed data

y1 = data_processed.POS
X1 = data_processed.drop('POS', axis=1)
data_processed.reset_index(drop=True, inplace=True)
gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix1, test_ix1 = next(gs.split(X1, y1,
groups=data_processed['sentence']))

data_train1 = data_processed.loc[train_ix1]
data_test1 = data_processed.loc[test_ix1]

data_train1.head()
```

| | sentence | Word | POS | Tag |
|---|---|---|---|---|
| 13 | Sentence: 2 | families | NNS | O |
| 14 | Sentence: 2 | soldiers | NNS | O |
| 15 | Sentence: 2 | killed | VBN | O |
| 16 | Sentence: 2 | conflict | NN | O |
| 17 | Sentence: 2 | joined | VBD | O |

```
data_test1.head()
```

| | sentence | Word | POS | Tag |
|---|---|---|---|---|
| 0 | Sentence: 1 | thousands | NNS | O |
| 1 | Sentence: 1 | demonstrators | NNS | O |
| 2 | Sentence: 1 | marched | VBN | O |
| 3 | Sentence: 1 | london | NNP | B-geo |
| 4 | Sentence: 1 | protest | VB | O |

```
dfupdate = data_train.sample(frac=.15, replace=False, random_state=42)
dfupdate.Word = 'UNKNOWN'
data_train.update(dfupdate)
words = list(set(data_train.Word.values))
# Convert words and tags into numbers
word2id = {w: i for i, w in enumerate(words)}
tag2id = {t: i for i, t in enumerate(tags)}
```

```
id2tag = {i: t for i, t in enumerate(tags)}
len(tags), len(words)
```

```
(42, 23607)
```

```
count_tags = dict(data_train.POS.value_counts())  # Total number of
POS tags in the dataset
# Now let's create the tags to words count
count_tags_to_words = data_train.groupby(['POS']).apply(
    lambda grp: grp.groupby('Word')
['POS'].count().to_dict()).to_dict()
# We shall also collect the counts for the first tags in the sentence
count_init_tags =
dict(data_train.groupby('sentence').first().POS.value_counts())

# Create a mapping that stores the frequency of transitions in tags to
it's next tags
count_tags_to_next_tags = np.zeros((len(tags), len(tags)), dtype=int)
sentences = list(data_train.sentence)
pos = list(data_train.POS)
for i in tqdm(range(len(sentences)), position=0, leave=True):
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        prevtagid = tag2id[pos[i - 1]]
        nexttagid = tag2id[pos[i]]
        count_tags_to_next_tags[prevtagid][nexttagid] += 1
```

```
100%|██████████| 702936/702936 [00:00<00:00,
1125320.05it/s]
```

```
startprob = np.zeros((len(tags),))
transmat = np.zeros((len(tags), len(tags)))
emissionprob = np.zeros((len(tags), len(words)))
num_sentences = sum(count_init_tags.values())
sum_tags_to_next_tags = np.sum(count_tags_to_next_tags, axis=1)
for tag, tagid in tqdm(tag2id.items(), position=0, leave=True):
    floatCountTag = float(count_tags.get(tag, 0))
    startprob[tagid] = count_init_tags.get(tag, 0) / num_sentences
    for word, wordid in word2id.items():
        emissionprob[tagid][wordid] = count_tags_to_words.get(tag,
{}).get(word, 0) / floatCountTag
    for tag2, tagid2 in tag2id.items():
        transmat[tagid][tagid2] = count_tags_to_next_tags[tagid]
[tagid2] / sum_tags_to_next_tags[tagid]
```

```
100%|██████████| 42/42 [00:00<00:00,
94.18it/s]
```

```
count_words = {}
for word in data_train.Word.values:
    count_words[word] = count_words.get(word, 0) + 1
```

```python
# then count the number of times a word appears after another word
count_word_transitions = {}
for sentence in data_train.groupby('sentence'):
    words = sentence[1]['Word'].values
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i+1]
        if w1 not in count_word_transitions:
            count_word_transitions[w1] = {}
        count_word_transitions[w1][w2] =
count_word_transitions[w1].get(w2, 0) + 1

# convert the counts to probabilities
word_transition_matrix = np.zeros((len(word2id)+1, len(word2id)+1))
sum_words_to_next_words = np.sum([count_word_transitions[w1][w2] for
w1 in count_word_transitions for w2 in count_word_transitions[w1]])
for w1, w1id in word2id.items():
    for w2, w2id in word2id.items():
        word_transition_matrix[w1id][w2id] =
count_word_transitions.get(w1, {}).get(w2, 0) /
sum_words_to_next_words
print(word_transition_matrix.shape)
```

```
(23608, 23608)
```

```python
def calculate_log_likelihood(sentence: List[str],
word_transition_matrix) -> float:
    """
    Given a sentence and word_transition_matrix, returns the log-
likelihood of the sentence.
    """
    # converting the sentence to a list of word IDs
    sentence_ids = [word2id.get(w, word2id['UNKNOWN']) for w in
sentence]

    # calculating the log-likelihood using the word transition matrix
    log_likelihood = np.log(word_transition_matrix[sentence_ids[0]]
[sentence_ids[1]])
    for i in range(1, len(sentence_ids) - 1):
        log_likelihood +=
np.log(word_transition_matrix[sentence_ids[i]][sentence_ids[i+1]] +
1e-10)
    return log_likelihood

calculate_log_likelihood(["This", "is", "a", "test", "sentence"],
word_transition_matrix)
```

```
-41.259970813020175
```

```python
model = hmm.MultinomialHMM(n_components=len(tags),
algorithm='viterbi', random_state=42)
model.startprob_ = startprob
```

```
model.transmat_ = transmat
model.emissionprob_ = emissionprob
```

```
data_test.loc[~data_test['Word'].isin(words), 'Word'] = 'UNKNOWN'
word_test = list(data_test.Word)
samples = []
for i, val in enumerate(word_test):
    samples.append([word2id[val]])

# TODO use panda solution
lengths = []
count = 0
sentences = list(data_test.sentence)
for i in tqdm(range(len(sentences)), position=0, leave=True):
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        count += 1
    elif i > 0:
        lengths.append(count)
        count = 1
    else:
        count = 1
```

```
100%|██████████████████████████████| 345639/345639 [00:00<00:00,
2775254.11it/s]
```

```
pos_predict = model.predict(samples, lengths)
pos_predict
```

```
array([23, 26,  7, ..., 25,  2, 16], dtype=int32)
```

## 2. Remove the labels from the Corpus and use Baum Welch algorithm to estimate the learning parameters.

```
import numpy as np


def baum_welch(observations, observations_vocab, n_hidden_states):
    """
    Baum-Welch algorithm for estimating the HMM parameters
    :param observations: observations
    :param observations_vocab: observations vocabulary
    :param n_hidden_states: number of hidden states to estimate
    :return: a, b (transition matrix and emission matrix)
    """

    def forward_probs(observations, observations_vocab,
n_hidden_states, a_, b_) -> np.array:
        """
        forward pass to calculate alpha
        :param observations: observations
```

```
        :param observations_vocab: observation vocabulary
        :param n_hidden_states: number of hidden states
        :param a_: estimated alpha
        :param b_: estimated beta
        :return: refined alpha_
        """
        a_start = 1 / n_hidden_states
        alpha_ = np.zeros((n_hidden_states, len(observations)),
dtype=float)
        alpha_[:, 0] = a_start
        for t in range(1, len(observations)):
          for j in range(n_hidden_states):
            calc = observations_vocab == observations[t]
            for i in range(n_hidden_states):
              alpha_[j, t] = sum(alpha_[i, t-1]*a_[i,j] * b_[j,
np.where(calc)[0][0]] for i in range(n_hidden_states))

        return alpha_

    def backward_probs(observations, observations_vocab,
n_hidden_states, a_, b_) -> np.array:
        """
        backward pass to calculate alpha
        :param observations: observations
        :param observations_vocab: observation vocabulary
        :param n_hidden_states: number of hidden states
        :param a_: estimated alpha
        :param b_: estimated beta
        :return: refined beta_
        """
        beta_ = np.zeros((n_hidden_states, len(observations)),
dtype=float)
        beta_[:, -1:] = 1
        for t in range(len(observations) -2, -1, -1):
          for i in range(n_hidden_states):
            calc2 = observations_vocab == observations[t+1]
            beta_[i,t] = sum(a_[i,j] * b_[j, np.where(calc2)[0]
[0]]*beta_[j, t+1] for j in range(n_hidden_states))
        return beta_

    def compute_gamma(alfa, beta, observations, vocab, n_samples, a_,
b_) -> np.array:
        """

        :param alfa:
        :param beta:
        :param observations:
        :param vocab:
        :param n_samples:
        :param a_:
```

```
        :param b_:
        :return:
        """
        # gamma_prob = np.zeros(n_samples, len(observations))
        gamma_prob = np.multiply(alfa, beta) / sum(np.multiply(alfa,
beta))
        return gamma_prob

    def compute_sigma(alfa, beta, observations, vocab, n_samples, a_,
b_) -> np.array:
        """

        :param alfa:
        :param beta:
        :param observations:
        :param vocab:
        :param n_samples:
        :param a_:
        :param b_:
        :return:
        """
        sigma_prob = np.zeros((n_samples, len(observations) - 1,
n_samples), dtype=float)
        denomenator = np.multiply(alfa, beta)
        for i in range(len(observations) - 1):
            for j in range(n_samples):
                for k in range(n_samples):
                    index_in_vocab = np.where(vocab == observations[i
+ 1])[0][0]
                    sigma_prob[j, i, k] = (alfa[j, i] * beta[k, i + 1]
* a_[j, k] * b_[k, index_in_vocab]) / sum(
                        denomenator[:, j])
        return sigma_prob

    # initialize A ,B
    a = np.ones((n_hidden_states, n_hidden_states)) / n_hidden_states
    b = np.ones((n_hidden_states, len(observations_vocab))) /
len(observations_vocab)
    for iter in tqdm(range(2000), position=0, leave=True):

        # E-step caclculating sigma and gamma
        alfa_prob = forward_probs(observations, observations_vocab,
n_hidden_states, a, b)  #
        beta_prob = backward_probs(observations, observations_vocab,
n_hidden_states, a, b)  # , beta_val
        gamma_prob = compute_gamma(alfa_prob, beta_prob, observations,
observations_vocab, n_hidden_states, a, b)
        sigma_prob = compute_sigma(alfa_prob, beta_prob, observations,
observations_vocab, n_hidden_states, a, b)
```

```
        # M-step caclculating A, B matrices
        a_model = np.zeros((n_hidden_states, n_hidden_states))
        for j in range(n_hidden_states):  # calculate A-model
            for i in range(n_hidden_states):
                for t in range(len(observations) - 1):
                    a_model[j, i] = a_model[j, i] + sigma_prob[j, t, 
i]
                normalize_a = [sigma_prob[j, t_current, i_current] for 
t_current in range(len(observations) - 1) for
                               i_current in range(n_hidden_states)]
                normalize_a = sum(normalize_a)
                if normalize_a == 0:
                    a_model[j, i] = 0
                else:
                    a_model[j, i] = a_model[j, i] / normalize_a

        b_model = np.zeros((n_hidden_states, len(observations_vocab)))

        for j in range(n_hidden_states):
            for i in range(len(observations_vocab)):
                indices = [idx for idx, val in enumerate(observations) 
if val == observations_vocab[i]]
                numerator_b = sum(gamma_prob[j, indices])
                denominator_b = sum(gamma_prob[j, :])
                if denominator_b == 0:
                    b_model[j, i] = 0
                else:
                    b_model[j, i] = numerator_b / denominator_b

        a = a_model
        b = b_model
    return a, b


import random

hidden_states = ['healthy', 'sick']
observable_states = ['sleeping', 'eating', 'pooping']
observable_map = {'sleeping': 0, 'eating': 1, 'pooping': 2}
observations = []
for i in range(40):

observations.append(observable_map[random.choice(observable_states)])

A, B = baum_welch(observations=observations, 
observations_vocab=np.array(list(observable_map.values())),
                  n_hidden_states=2)

hidden_state_sequence = model(startprob, transmat, emissionprob, 
observations)
```

```
print("Observations:", observations)
print("Viterbi sequence:", model)
```