Problems L5

Fundamentals of Natural Language

Authors:

```
Sonia Espinilla, 1708919
Agustina Inés Lazzati, 1707964
```

Autonomous University of Barcelona

March 30, 2025

Sequence Labeling - NER

Based on work by Adri Molina.

The extraction of relevant information from historical handwritten document collections is one of the key steps in order to make these manuscripts available for access and searches. In this context, instead of a pure transcription, the objective is to move towards document understanding. Concretely, the aim is to detect the named entities and assign each of them a semantic category, such as family names, places, occupations, etc.

A typical application scenario of named entity recognition are demographic documents, since they contain people's names, birthplaces, occupations, etc. In this scenario, the extraction of the key contents and its storage in databases allows the access to their contents and envision innovative services based in genealogical, social or demographic searches.

Usage of Google Colab is not mandatory, but highly recommended as most of the behaviors are expected for a Linux VM with IPython bindings.

First, we will install the unmet dependencies.

This will download some packages and the required data, it may take a while.

```
#@title
from IPython.display import clear_output

!git clone https://github.com/EauDeData/nlp-resources
!cp -r nlp-resources/ resources/
!rm -rf nlp-resources/

%pip install nltk
%pip install scikit-learn
%pip install python-crfsuite
%pip install sklearn-crfsuite
%pip install git+https://github.com/MeMartijn/updated-sklearn-crfsuite
clear_output()

from typing import *

import nltk
import numpy as np
import copy
import random
from collections import Counter

import sklearn_crfsuite as crfs
from sklearn_crfsuite import scorers
from sklearn_crfsuite import metrics

from resources.data.dataloaders import EsposallesTextDataset
```

Graded Exercise 1 (4 points)

The first part of this exercise consists of pre-processing the data. In order to get acquainted with the NER dataset we will be using throughout this exercise, you are tasked to do the following:

- **Write a Look-up Table-based (LUT) tokeniser**. This includes:
    - The LUT generator itself.
    - A sentence padding function.

    - A sentence-to-LUT conversion function.

After the LUT-based tokeniser is written:

- Generate the LUT on the training data of the NER task.
- Find all out-of-vocabulary words in the test partition.
- Study what kinds of words they are.

Contextualise all of these answers with the domain of the data being analysed.

Data Processing

Loading the dataset

```
random.seed(42)
train_loader = EsposallesTextDataset('resources/data/esposalles/')
test_loader = copy.deepcopy(train_loader)
test_loader.test()
```

Example of data from each loader:

> Format string: `word:label`

This is a simple IO tagging format which, for the task at hand, should be more than enough.

```
print([f"{x}:{y}" for x,y in zip(*train_loader[0])])
print([f"{x}:{y}" for x,y in zip(*test_loader[0])])
```

```
['Dilluns:other', 'a:other', '5:other', 'rebere:other', 'de:other', 'Hyacinto:name', 'Boneu:surname', 'hortola:occupation', 'de:othe
['Divendres:other', 'a:other', '18:other', 'rebere:other', 'de:other', 'Juan:name', 'Torres:surname', 'pages:occupation', 'habitant
```

If the dataset is correctly downloaded you will see two different samples above, and both tests passed below.

```
# Check Dataset

for idx in range(len(train_loader)):

    x, y = train_loader[idx]
    if len(x) != len(y):
        print("train_set test not passed")
        break
    else:
        print("train_set test passed")

for idx in range(len(test_loader)):

    x, y = test_loader[idx]
    if len(x) != len(y):
        print("test_set test not passed")
        break

else:
    print("test_set test passed")
```

Mostrar salida oculta

An example of a typical pre-processing pipeline

Let's do a quick exercise to get acquainted with the data.

Most efficient implementations of these kinds of algorithms do not use words encoded as strings directly. Usually, one would first convert each word into an integer index that is stored succinctly and contiguously in memory and perform all computations with it. The way this is usually done is through a Look Up Table (LUT), which can be implemented very easily in Python using a map (Dictionary).

Furthermore, many packages will process multiple sentences at once. However, the fact that there are variable length sentences makes indexing in parallel algorithms complicated. To this avail, we sometimes want to pad all sentences to a fixed sentence length to ensure they can be stored in contiguous matrices easily.

Lastly, as a common practice, we will want to create three new tokens `<bos>` and `<eos>` for the start and the end of a given sequence and `<unk>` for unknown tokens in the application (test) layer or 0 padding during the training. These tokens must also be stored in the LUT.

> TODO: Write the LUT computation. Ensure to incorporate the various additional tokens in the process.

> TODO: Write the Out-of-vocabulary word checking function

> TODO: Write functions to pad sentences and to apply the LUT on the padded sentences.

```
def create_tokens_lut(train_dataset: EsposallesTextDataset) -> Dict[str, int]:
    vocabulary = set()

    for i in range(len(train_loader)):
        words, _ = train_loader[i]
        vocabulary.update(words)

    lut = {
        '<pad>': 0,
        '<bos>': 1,    # start
        '<eos>': 2,    # End
        '<unk>': 3,    # Unknown
    }

    for idx, word in enumerate(sorted(vocabulary), start=4):
        lut[word] = idx

    return lut

lut = create_tokens_lut(train_loader)
```

```
def check_oov_words(lut: Dict[str, int], test_set: EsposallesTextDataset) -> List[str]:

    oov_words = set()

    known_words = set()
    special_tokens = {'<pad>', '<bos>', '<eos>', '<unk>'}
    for word in lut.keys():
        if word not in special_tokens:
            known_words.add(word)

    for i in range(len(test_set)):
        words, _ = test_set[i]
        for word in words:
            if word not in known_words:
                oov_words.add(word)

    return sorted(oov_words)

test_set = test_loader
oov_words = check_oov_words(lut, test_set)
print("OOV words in test:", oov_words)
```

```
OOV words in test: ['Alaverni', 'Angli', 'Arevig', 'Argemir', 'Arisart', 'Assensio', 'Auger', 'Bachs', 'Begas', 'Berenguer', 'Blanqu
```

The out-of-vocabulary (OOV) words in your test set are primarily personal names, surnames, and place names from historical Catalan marriage records. They are OOV because of spelling variations and influences from past languages (Latin). Also, names of villages, towns, or regions of Catalunya that can be compound names may appear. On another note, we can find professions or social roles that can be modified by gender or terms that originated from Latin.

Additionally, if we normalize, we wouldn't find the names of the days of the week.

```
MAX_SEQUENCE_LENGTH = 50

def pad_sentence(sent: List[str], max_sent_len: int) -> List[str]:
    padded_sent = ["<bos>"] + sent + ["<eos>"]
    #sentences too long, truncate and add eos at final
    if len(padded_sent) > max_sent_len:
        padded_sent = padded_sent[:max_sent_len - 1] + ["<eos>"]
    #sentence to short, we add pad at the final
    elif len(padded_sent) < max_sent_len:
        padded_sent = padded_sent + ["<pad>"] * (max_sent_len - len(padded_sent))

    return padded_sent

def apply_lut(lut: Dict[str, int], sent: List[str]) -> List[int]:

    index_sent = []
    for word in sent:
        index = lut.get(word, lut['<unk>'])
```

```
        index_sent.append(index)
    return index_sent


tokenised_train = []
print(f"Total sentences in loader: {len(train_loader)}")
tokenised_train = []
for i in range(len(train_loader)):
    sent, _ = train_loader[i]  # Access via index
    padded_sent = pad_sentence(sent, MAX_SEQUENCE_LENGTH)
    indexed_sent = apply_lut(lut, padded_sent)
    tokenised_train.append(indexed_sent)

print(tokenised_train[:2])  # Print the first two processed sentences
```

```
Total sentences in loader: 871
[[1, 523, 1621, 32, 2141, 1745, 706, 264, 1952, 1745, 175, 1859, 1745, 743, 264, 2097, 1753, 2283, 1745, 864, 1622, 98, 1803, 1861,
```

Show a simple input and output example for the full pipeline (you can handcraft an example of your own with 5-10 words and pad it to 15 words).

```
train_data = [
    (["Hello", "we", "are", "Agustina", "and", "Sonia"], ["other", "other", "other","other", "other", "other"]),
    (["This", "is", "a", "test"], ["other", "other", "other", "other"])
]
test_data = [
    (["Hi", "we", "are", "Lucia", "and", "Carlos"], ["other", "other", "other", "other", "other", "other"]),
    (["This", "is", "another", "experiment"], ["other", "other", "other", "other"])
]
def create_lut(train_data: list) -> Dict[str, int]:
    vocabulary = set()
    for words, _ in train_data:
        vocabulary.update(words)

    lut = {
        '<pad>': 0,
        '<bos>': 1,
        '<eos>': 2,
        '<unk>': 3,
    }
    for idx, word in enumerate(sorted(vocabulary), start=4):
        lut[word] = idx
    return lut

lut = create_lut(train_data)
print("LUT:", lut)
```

```
LUT: {'<pad>': 0, '<bos>': 1, '<eos>': 2, '<unk>': 3, 'Agustina': 4, 'Hello': 5, 'Sonia': 6, 'This': 7, 'a': 8, 'and': 9, 'are': 10,
```

```
MAX_SEQUENCE_LENGTH = 15

tokenised_train = []
for words, _ in train_data:
    padded = pad_sentence(words, MAX_SEQUENCE_LENGTH)
    tokenised = apply_lut(lut, padded)
    tokenised_train.append(tokenised)

for sent in tokenised_train:
    print(sent)
```

```
[1, 5, 13, 10, 4, 9, 6, 2, 0, 0, 0, 0, 0, 0, 0]
[1, 7, 11, 8, 12, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
oov_words = check_oov_words(lut, test_data)
print("OOV words in test:", oov_words)

tokenised_test = []
for words, _ in test_data:
    padded = pad_sentence(words, MAX_SEQUENCE_LENGTH)
    tokenised = apply_lut(lut, padded)
    tokenised_test.append(tokenised)

for sent in tokenised_test:
    print(sent)
```

```
OOV words in test: ['Carlos', 'Hi', 'Lucia', 'another', 'experiment']
[1, 3, 13, 10, 3, 9, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 7, 11, 3, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Graded Exercise 2 (3 points)

Implement a Baseline maximum likelihood estimator from the training data. To do so, you will need to calculate probabilities P(tag|word) from the training dataset and then use these probabilities to predict the tags on the test partition. Analyse these results.

Contextualise all of the answers with the domain of the data being analysed.

NER - Baseline Approach

Let's actually start implementing sequence labelling algorithms. The first approach we will try is based on computing the probability of each tag for each word in our training corpus. With this simple approach we can obtain a baseline accuracy score.

Since this is a very simple algorithm and we have very little data, we can work directly on strings to spare us a few headaches.

> TODO: Write the `compute_tag_prob` function

```
def compute_tag_prob(train_loader: EsposallesTextDataset) -> Dict[str, Dict[str, float]]:
    word_tag_counts = {}
    word_counts = Counter()
    tag_counts = Counter()

    most_common_tag = None

    output={}

    for i in range (len(train_loader)):
        x, y = train_loader[i]
        for word, tag in zip(x, y):
            if word not in word_tag_counts:  # Here we are handling the missing missing words
                word_tag_counts[word] = {}
            if tag not in word_tag_counts[word]:  # Here missing keys
                word_tag_counts[word][tag] = 0
            word_tag_counts[word][tag] += 1
            tag_counts[tag] += 1
            word_counts[word] += 1

    for word, tag_counts in word_tag_counts.items():
        total = sum(tag_counts.values())
        output[word] = {}
        for tag, count in tag_counts.items():
            output[word][tag] = count / total   # Computing P(tag|word), which means # of times a word is associated
                                                #with a specific tag divided by the total occurrences of that word
    return output
```

```
tag_probs = compute_tag_prob(train_loader)
```

At this point, as an example, your emissions dictionary should yield the following probabilities:

$P(location|$ `Prats` $) = 18\%$

$P(surname|$ `Prats` $) = 72\%$

$P(other|$ `Prats` $) = 9\%$

```
tag_probs["Prats"]
```

```
{'location': 0.18181818181818182,
 'surname': 0.7272727272727273,
 'other': 0.09090909090909091}
```

This method is, of course, quite limited by the fact that **it cannot model context**.
It will output the most likely class for each word. However, you will find that this method works quite well in spite of its simplicity. Note that you will have to do a few assumptions for edge cases such as finding an out-of-vocabulary word.

You have to analyse the results of this algorithm. You must implement a confusion matrix generator from the test set to aid you in this endeavour. Then, you can answer the following questions.

- What do all of the mistakes have in common?
- What kinds of words are the least performers?
- What's your solution for out-of-vocabulary words? Can you provide a prediction for those?
- What words are usually the best performers?

CLUE: You should be getting around 88% precision if you do everything correctly.

> TODO: Write the `predict_test_set`, `find_common_errors` and `compute_token_precision` functions.

> TODO: Analyse the results according to the previously stated questions. - ALSO EVALUATE WHERE IS NOT FUNCTIONING

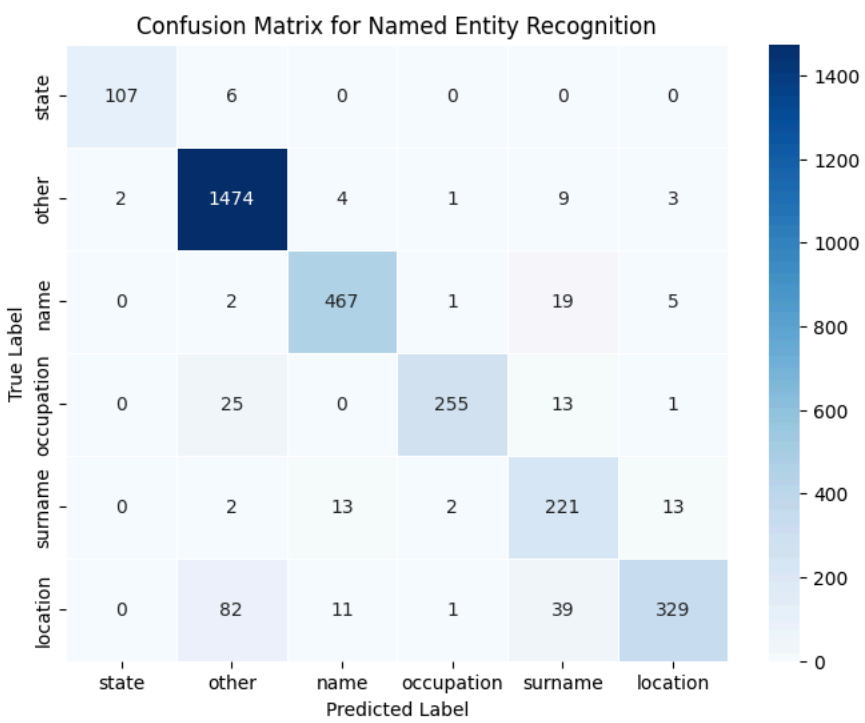
```
def predict_test_set(
    tag_probs: Dict[str, Dict[str, float]],
    test_set: EsposallesTextDataset,
) -> List[List[str]]:

    # TO DEAL WITH UNSEEN WORDS/ OOV (out-of-vocabulary)
    # We decide to asign the tag that appeared the most
    tag_counts = Counter()
    for word_tags in tag_probs.values():
        for tag, prob in word_tags.items():
            tag_counts[tag] += prob
    most_common_tag = max(tag_counts, key=tag_counts.get)

    test_predictions = []  # will be list of lists

    for i in range(len(test_set)):
        test_words,_ = test_set[i]
        sentence_predictions = [] # predictions for this sentence

        for word in test_words:
            if word not in tag_probs:
                sentence_predictions.append(most_common_tag)
            else:
                sentence_predictions.append(max(tag_probs[word], key=tag_probs[word].get))
        test_predictions.append(sentence_predictions)

    return test_predictions



def find_common_errors(
    test_set: EsposallesTextDataset,
    test_predictions: List[List[str]],
) -> Any:

    #WE WILL TRY TO TRACK THE TREE WAYS OF EVALUATION PROPOSED

    confusion_counts = {}
    wrong_words_counts = Counter()
    wrong_tags_counts = Counter()

    for i in range(len(test_set)):
        test_words, true_tags = test_set[i]
        predicted_tags = test_predictions[i]

        for word, true_tag, predicted_tag in zip(test_words, true_tags, predicted_tags):
            if true_tag != predicted_tag:  # If prediction is incorrect

                if true_tag not in confusion_counts:
                    confusion_counts[true_tag] = {}
                if predicted_tag not in confusion_counts[true_tag]:
                    confusion_counts[true_tag][predicted_tag] = 0

                confusion_counts[true_tag][predicted_tag] += 1
                wrong_words_counts[word] += 1
                wrong_tags_counts[true_tag] += 1

    return confusion_counts, wrong_words_counts, wrong_tags_counts


def compute_token_precision(
    test_set: EsposallesTextDataset,
    test_predictions: List[str],
) -> float:
    total_tokens = 0
    correct_predictions = 0

    for i in range(len(test_set)):
        test_words, true_tags = test_set[i]
```

```
        predicted_tags = test_predictions[i]

        for true_tag, predicted_tag in zip(true_tags, predicted_tags):
            total_tokens += 1
            if true_tag == predicted_tag: #Tag predicted is the correct one
                correct_predictions += 1

    return correct_predictions / total_tokens if total_tokens > 0 else 0.0  # Avoid division by zero (when empty dataset)
```

```
test_predictions = predict_test_set(tag_probs, test_loader)

for sentence in test_predictions[:5]:  # Print first 5 predicted sentences
    print(sentence)
```

```
['other', 'other', 'other', 'other', 'other', 'name', 'surname', 'occupation', 'other', 'other', 'location', 'other', 'other', 'name
['other', 'other', 'other', 'other', 'other', 'name', 'surname', 'occupation', 'other', 'surname', 'other', 'other', 'name', 'locati
['other', 'other', 'other', 'other', 'other', 'name', 'name', 'occupation', 'other', 'location', 'other', 'other', 'name', 'name',
['other', 'other', 'other', 'other', 'name', 'name', 'surname', 'occupation', 'other', 'location', 'other', 'other', 'name', 'surnam
['other', 'other', 'other', 'other', 'other', 'name', 'surname', 'occupation', 'other', 'location', 'location', 'other', 'location',
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(true_labels, predicted_labels, tag_list, title):
    # Compute the confusion matrix
    cm = confusion_matrix(true_labels, predicted_labels, labels=tag_list)

    # Plot the confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=tag_list, yticklabels=tag_list, linewidths=0.5)
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(title)
    plt.show()

def generate_confusion_matrix(test_set, test_predictions, title):
    true_labels = []
    predicted_labels = []

    for i in range(len(test_set)):
        test_words, true_tags = test_set[i]
        predicted_tags = test_predictions[i]

        true_labels.extend(true_tags)
        predicted_labels.extend(predicted_tags)

    unique_tags = list(set(true_labels))  # Get all unique tags
    plot_confusion_matrix(true_labels, predicted_labels, unique_tags, title)

generate_confusion_matrix(test_loader, test_predictions, "Confusion Matrix for Named Entity Recognition")
```



Quick Answers!

1. **Which circumstances lead to failures?**

The confusion matrix shows that locations followed by occupations misclassified the most.

2. **What are the most typical errors?**

The most common misclassifications where locations mistaken for other (82 times), or surnames(39 times), ocupation mistaken for other and names for surnames.

3. **Which words are the worst performers?**

The words worst performers where the words standing for locations being incorrectly classified 133 times, in particular the word "de" with 91 errors.

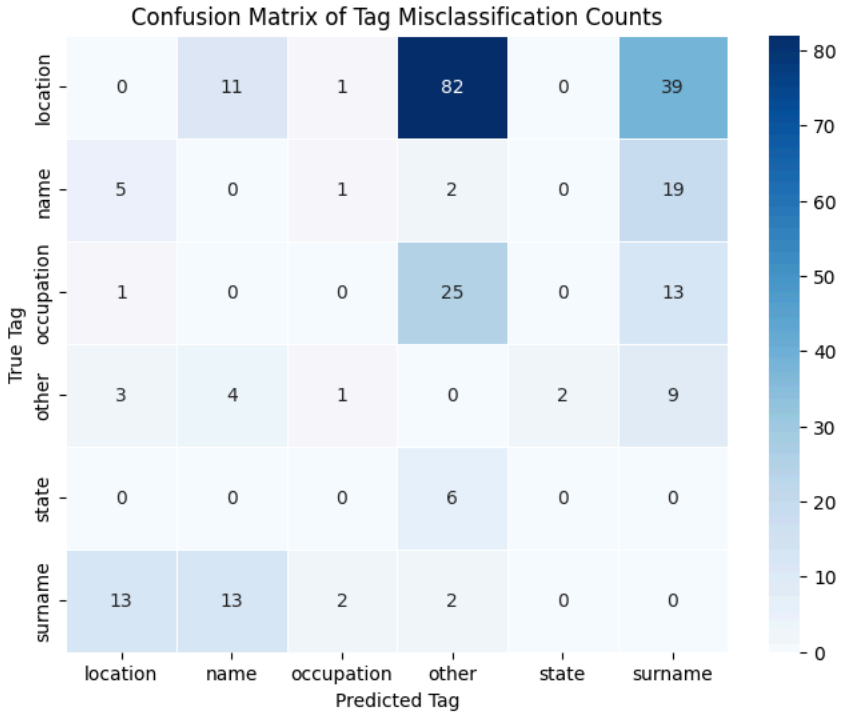
4. **Which words perform best?**

The words that perform best was other, being correctly classified 1474 times.

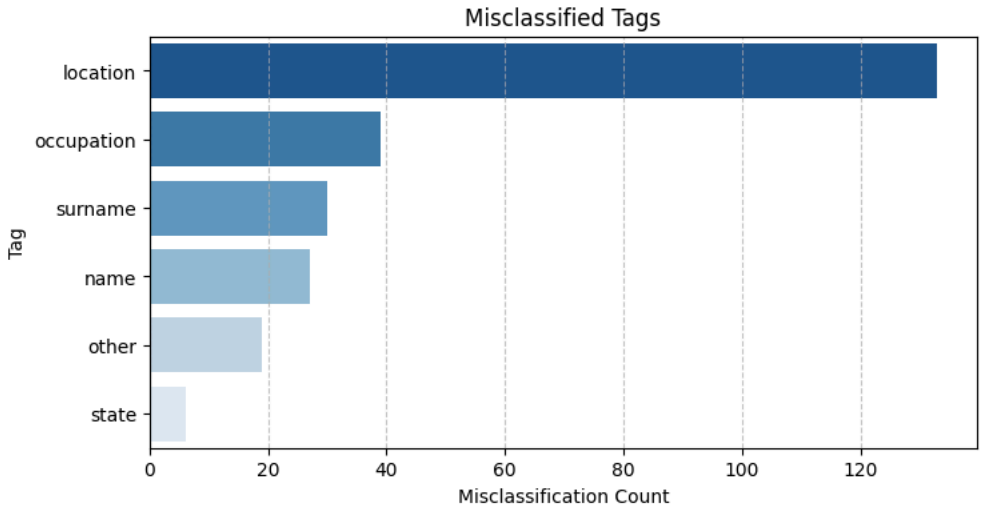
```
confusion_counts, wrong_words_counts, wrong_tags_counts = find_common_errors(test_loader, test_predictions)

# Print confusion matrix (tag misclassification counts)
print("Tag Confusion Counts:")
for true_tag, mistaken_tags in confusion_counts.items():
    print(f"{true_tag} → {mistaken_tags}")
```

```
Tag Confusion Counts:
location → {'other': 82, 'surname': 39, 'name': 11, 'occupation': 1}
surname → {'location': 13, 'name': 13, 'other': 2, 'occupation': 2}
name → {'location': 5, 'surname': 19, 'occupation': 1, 'other': 2}
other → {'surname': 9, 'name': 4, 'location': 3, 'state': 2, 'occupation': 1}
occupation → {'other': 25, 'surname': 13, 'location': 1}
state → {'other': 6}
```

```
# Important first convert the dictionary
conf_matrix_df = pd.DataFrame.from_dict(confusion_counts, orient='index').fillna(0).astype(int)
conf_matrix_df = conf_matrix_df.sort_index(axis=0).sort_index(axis=1)

plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_df, annot=True, fmt="d", cmap="Blues", linewidths=0.5)

plt.xlabel("Predicted Tag")
plt.ylabel("True Tag")
plt.title("Confusion Matrix of Tag Misclassification Counts")

plt.show()
```



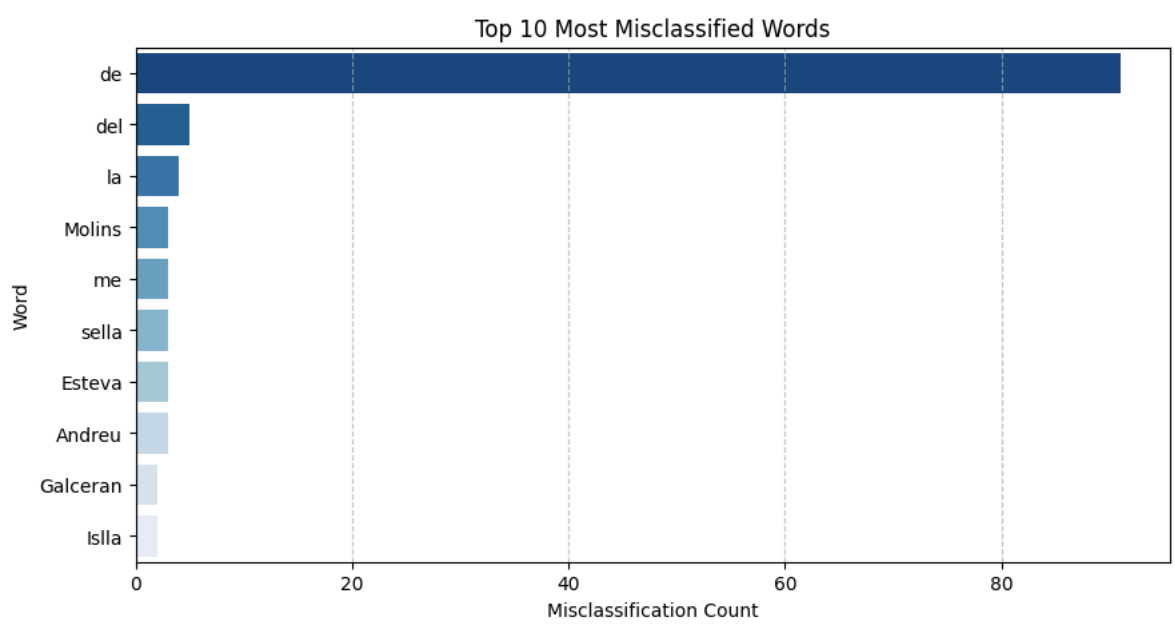

```
tags, tag_counts = zip(*wrong_tags_counts.most_common(6))
tags_df = pd.DataFrame({"Tag": tags, "Count": tag_counts})

plt.figure(figsize=(8, 4))
sns.barplot(data=tags_df, x="Count", y="Tag", hue="Tag", palette="Blues_r", legend=False)

plt.xlabel("Misclassification Count")
plt.ylabel("Tag")
plt.title("Misclassified Tags")
plt.grid(axis="x", linestyle="--", alpha=0.7)

plt.show()

# see exact value
print("\nMisclassified Tags:")
for tag, count in wrong_tags_counts.most_common(6):
    print(f"{tag}: {count} errors")
```




```
Misclassified Tags:
location: 133 errors
occupation: 39 errors
surname: 30 errors
name: 27 errors
other: 19 errors
state: 6 errors
```

```
words, word_counts = zip(*wrong_words_counts.most_common(10))
words_df = pd.DataFrame({"Word": words, "Count": word_counts})

plt.figure(figsize=(10, 5))
```

```
sns.barplot(data=words_df, x="Count", y="Word", hue="Word", palette="Blues_r", legend=False)

plt.xlabel("Misclassification Count")
plt.ylabel("Word")
plt.title("Top 10 Most Misclassified Words")
plt.grid(axis="x", linestyle="--", alpha=0.7)

plt.show()

# To see exact value
print("\nMost Misclassified Words:")
for word, count in wrong_words_counts.most_common(10):
    print(f"{word}: {count} errors")
```




```
Most Misclassified Words:
de: 91 errors
del: 5 errors
la: 4 errors
Molins: 3 errors
me: 3 errors
sella: 3 errors
Esteva: 3 errors
Andreu: 3 errors
Galceran: 2 errors
Islla: 2 errors
```

```
precision = compute_token_precision(test_loader, test_predictions)
print(f"Precision: {precision * 100:.2f}%")
```

```
Precision: 91.82%
```

Our conclusions refering to Baseline Approach:

1. **Examine in what circumstances it fails and what circumstances it works.**

This model proposed performs well when tagging words that frequently appear in our training set. Surprisingly, this model performed pretty good with states, misclasifing 6 times as other (but this may be because there weren't to many states to classify). However, it struggles in cases where words have multiple possible tags, leading to confusion. The confusion matrix shows that locations are often misclassified as other (82 times) even sometimes mistaken for surnames (39 times). Similarly, occupations are frequently tagged as other (25 times), this could suggest that jobs might not have enough distinct examples in the training data. As this model uses maximum likelihood estimation (MLE) and it cannot model context, will cause failures since word frequency alone is insufficient for correct classification.

2. **What are the most typical errors?**

The most misclassified word is "de" with 91 errors, likely because is a word that appears in both surnames and locations, leading to incorrect tagging. Wiht significantly less degree, the next common errors include "Molins" (3 errors) and "Esteva" (3 errors), so we deduce the model struggles to differentiate between names and locations. Moreover, surnames are often mistaken for locations (13 times), and occupations are confused with other (25 times). With respect to tags, location takes the lead with 133 errors, misclassified tags.

3. **What are you doing to address out-of-vocabulary words?**

Our idea to handle out-of-vocabulary (OOV) words was assigns them the most common tag in the dataset, ensuring that every word gets a tag even if it has never appeared in the training data. However, we understood that this approach introduced some errors (which justifies the

previous results). As 'other' was the most common tag, most of the misclasified tags (especially location) where assing to this tag. For future approaches, we could implement word embeddings, character-level models, or using external linguistic resources to improve recognition.

Graded Exercise 3 (3 points)

Use the implemented HMM to perform the same task on the test partition and compare the performance.

Contextualise with the domain of the data being analysed.

You can obtain up to 3 extra points for using Conditional Random Fields to re-do the same as in exercise 3.

HMM Approach

As demonstrated in the previous experiment, using just the priors have not enough expressivity for managing both out of vocabulary words and polysemic words. Here we will use the `python-crfsuite` module to build a Hidden Markov Model and improve the predictions on `test_set`.

Check here the `python-crfsuite` documentation.

First, we will set up the parameters for our HMM model.

The HMM will be implemented using CRFs, because at the end of the day a HMM is more or less equivalent to a CRF that uses only the current emitted word and the previous tag as features.

> TODO: Train the HMM and compare it to the baseline approach. Where is it better?

> TODO: Check which tag transitions are most common. Rationalise why this is the case and contextualise it with the type of input data you have. Do they make sense?

```
def get_word_to_hmm_features(sent: List[Tuple[str, str]], i: int) -> List[str]:

    word, _ = sent[i]

    # The features we care about are the current emitted word and whether this word is
    # at the beginning or the end of a sentence.
    features = [
        "bias",
        "word.lower=" + word.lower(),
    ]
    if i == 0:
        features.append("bos")

    if i == len(sent) - 1:
        features.append("eos")

    return features


def get_sent_to_hmm_features(sent: List[Tuple[str, str]]) -> List[List[str]]:
    """Extract HMM-CRF features for a full sentence."""
    return [get_word_to_hmm_features(sent, i) for i in range(len(sent))]


def sent2labels(sent: List[Tuple[str, str]]) -> List[str]:
    """Get labels from the sentence tuple format."""
    return [label for token, label in sent]


def sent2tokens(sent: List[Tuple[str, str]]) -> List[str]:
    """Get words from the sentence tuple format."""
    return [token for token, label in sent]
```

```
# Transform the dataset to the (token, gt) tuple format
train_sents = [
    [(x, y) for x, y in zip(*train_loader[idx])] for idx in range(len(train_loader))
]
test_sents = [
    [(x, y) for x, y in zip(*train_loader[idx])] for idx in range(len(test_loader))
]

X_train = [get_sent_to_hmm_features(s) for s in train_sents]
y_train = [sent2labels(s) for s in train_sents]
```

```
X_test = [get_sent_to_hmm_features(s) for s in test_sents]
y_test = [sent2labels(s) for s in test_sents]
```

```
import pycrfsuite
trainer = pycrfsuite.Trainer(verbose=False)
for xseq, yseq in zip(X_train, y_train):
    trainer.append(xseq, yseq)  # Stack the data
```

You can modify these hyperparameters if you wish. Check the documentation to see if there are some additions you can make here.

```
trainer.set_params(
    {
        "c1": 1.0,  # coefficient for L1 penalty
        "c2": 1e-3,  # coefficient for L2 penalty
        "max_iterations": 50,  # Max Number of iterations for the iterative algorithm
        # include transitions that are possible, but not observed (smoothing)
        "feature.possible_transitions": True,
    }
)
```

```
%%time
trainer.train('npl_ner_hmm.crfsuite') # Train the model and save it locally.
```

```
CPU times: user 903 ms, sys: 6.27 ms, total: 910 ms
Wall time: 3 s
```

```
tagger = pycrfsuite.Tagger()
tagger.open("npl_ner_hmm.crfsuite")  # Load the inference API
```

Mostrar salida oculta

```
example_sent = test_sents[0]
print(" ".join(sent2tokens(example_sent)), end="\n\n")

print("Predicted:", " ".join(tagger.tag(get_sent_to_hmm_features(example_sent))))
print("Correct:  ", " ".join(sent2labels(example_sent)))  # Inference
```

```
Dilluns a 5 rebere de Hyacinto Boneu hortola de Bara fill de Juan Boneu parayre defunct y de Maria ab Anna donsella filla de t Cases

Predicted: other other other other other name surname occupation other location other other name surname occupation other other othe
Correct:   other other other other other name surname occupation other location other other name surname occupation other other othe
```

In the following code, you have a way of checking which of the most common state transitions are present in the model.

```
info = tagger.info()


def print_transitions(trans_features):
    for (label_from, label_to), weight in trans_features:
        print("%-6s -> %-7s %0.6f" % (label_from, label_to, weight))


print("Top likely transitions:")
print_transitions(Counter(info.transitions).most_common(15))

print("\nTop unlikely transitions:")
print_transitions(Counter(info.transitions).most_common()[-15:])
```

```
Top likely transitions:
surname -> occupation 5.019786
location -> location 4.249178
occupation -> occupation 4.194908
name    -> surname 3.918272
surname -> surname 3.294527
other   -> name    2.642988
name    -> name    2.280106
other   -> location 1.814986
occupation -> other   1.799188
state   -> state   1.515564
name    -> state   1.182575
other   -> other   1.181984
state   -> other   0.735261
occupation -> state   0.467426
location -> other   0.403282

Top unlikely transitions:
state   -> occupation 0.214062
```

```
surname -> state   0.070542
location -> occupation -0.072833
name   -> other   -0.506374
occupation -> name    -0.617364
other  -> surname -0.626089
occupation -> location -0.735001
surname -> name    -0.998662
other  -> occupation -1.030786
other  -> state   -1.296538
name   -> occupation -1.650451
occupation -> surname -2.111763
name   -> location -2.166658
location -> surname -2.771394
location -> name    -3.351583
```

```
def print_state_features(state_features):
    for (attr, label), weight in state_features:
        print("%0.6f %-6s %s" % (weight, label, attr))


print("Top positive:")
print_state_features(Counter(info.state_features).most_common(20))

print("\nTop negative:")
print_state_features(Counter(info.state_features).most_common()[-20:])
```

```
Top positive:
10.471530 state  word.lower=viudo
9.201264 state  word.lower=donsella
9.171407 other  word.lower=ab
9.073835 other  word.lower=fill
9.043569 other  word.lower=defuncts
8.975021 other  word.lower=#
8.538504 state  word.lower=viuda
8.204700 other  word.lower=defunct
7.974212 other  word.lower=y
7.971931 other  word.lower=defuncta
7.816721 location word.lower=frances
7.655355 state  word.lower=dosella
7.522998 other  word.lower=rebere
7.259508 other  word.lower=habitant
7.223392 location word.lower=bara
7.056602 other  word.lower=a
6.963770 other  word.lower=de
6.655697 other  word.lower=filla
6.586878 other  word.lower=habitat
6.535491 occupation word.lower=llana

Top negative:
0.009387 occupation word.lower=pastisser
0.006814 surname word.lower=pere
-0.000147 name   word.lower=sr
-0.000667 location word.lower=dels
-0.015598 location word.lower=menat
-0.061897 surname word.lower=vila
-0.086558 surname word.lower=del
-0.118397 surname word.lower=toni
-0.285311 occupation bias
-0.398388 location word.lower=habitant
-0.422007 surname eos
-0.499756 location word.lower=pages
-0.504381 surname word.lower=texidor
-0.558585 surname word.lower=y
-0.694049 surname word.lower=#
-0.757626 state  bias
-1.027974 occupation word.lower=del
-1.104222 location word.lower=en
-1.170696 surname word.lower=serrat
-1.568957 location word.lower=domiciliat
```

Our conclusions refering to HMM approach:

1. **Where does the HMM perform better than the baseline approach?**

The HMM performs significantly better than the baseline method, especially in keeping label sequences consistent. For instance, it maintains location labels effectively (with a strong 5.06 weight for location-to-location transitions), while the baseline makes 133 location errors. It also excels at recognizing special historical terms like "donsella" (weight 10) and "viudo" (weight 9) that the baseline often misclassifies. The model's handling of locations and its transition patterns match well with actual notarial document structures.

2. **What words does it still struggle with?**

The model still has difficulty with common connecting words ("de", "y"), terms that could be either surnames or locations, and uncommon last names (which get strong negative scores).

3. **How differently does the model perform w.r.t. Out-of-Vocabulary words?**

For out-of-vocabulary in its original training data, the HMM works better than the baseline when dealing with location, occupational terms and morphological variants. However, it struggles with rare personal names (which get the worst negative scores) and some preposition combinations like "dels".

4. **Study the kinds of transitions the model has learnt. Do they make sense?**

The patterns the model learned make good sense for historical documents. It particularly recognizes important sequences like surname-to-job-title and firstname-to-surname transitions. These patterns accurately reflect the typical features found in 18th-19th century Catalan records, including family naming traditions, property descriptions, and indicators of social status.

Optional: Use arbitrary CRF Features

You can try to extract finer-grained features if you want using a CRF model. If you do so, provide the same analysis of results as with the HMM. We can use gradient descent or forward-backward.

```
import pycrfsuite
from collections import Counter
def get_word_to_crf_features(sent: List[Tuple[str, str]], i: int) -> List[str]:
    word, _ = sent[i]

    features = [
        "bias",
        "word.lower=" + word.lower(),
        "word[-3:]=" + word[-3:],  # Suffix
        "word[:3]=" + word[:3],    # Prefix
        "word.isupper=%s" % word.isupper(),
        "word.istitle=%s" % word.istitle(),
        "word.isdigit=%s" % word.isdigit(),
    ]

    if i > 0:
        prev_word = sent[i - 1][0]
        features.extend([
            "prev_word.lower=" + prev_word.lower(),
            "prev_word.isupper=%s" % prev_word.isupper(),
            "prev_word.istitle=%s" % prev_word.istitle(),
        ])

    if i == 0:
        features.append("bos")

    if i < len(sent) - 1:
        next_word = sent[i + 1][0]
        features.extend([
            "next_word.lower=" + next_word.lower(),
            "next_word.isupper=%s" % next_word.isupper(),
            "next_word.istitle=%s" % next_word.istitle(),
        ])

    if i == len(sent) - 1:
        features.append("eos")

    return features


def get_sent_to_crf_features(sent: List[Tuple[str, str]]) -> List[List[str]]:
    """Extract HMM-CRF features for a full sentence."""
    return [get_word_to_crf_features(sent, i) for i in range(len(sent))]
```

```
# Again we transform dataset for CRF training
X_train = [get_sent_to_crf_features(s) for s in train_sents]
y_train = [sent2labels(s) for s in train_sents]

X_test = [get_sent_to_crf_features(s) for s in test_sents]
y_test = [sent2labels(s) for s in test_sents]

crf_trainer = pycrfsuite.Trainer(verbose=False)  # Instance a CRF trainer

for xseq, yseq in zip(X_train, y_train):
    crf_trainer.append(xseq, yseq)  # Stack the data
```

```
crf_trainer.set_params(
    {
       "c1": 0.1,  # L1 regularization
        "c2": 1e-3,  # L2 regularization
```

```
        "max_iterations": 50,
        "feature.possible_transitions": True,
    }
)
```

```
%%time
crf_trainer.train('npl_ner_crf.crfsuite') # Train the model and save it locally.
```

```
CPU times: user 1.6 s, sys: 8.95 ms, total: 1.6 s
Wall time: 2.11 s
```

```
tagger_crf = pycrfsuite.Tagger()
tagger_crf.open("npl_ner_crf.crfsuite")  # Load the inference API
```

Mostrar salida oculta

```
example_sent = test_sents[0]
print(" ".join(sent2tokens(example_sent)), end="\n\n")

print("Predicted:", " ".join(tagger_crf.tag(get_sent_to_crf_features(example_sent))))
print("Correct:  ", " ".join(sent2labels(example_sent)))  # Inference
```

```
Dilluns a 5 rebere de Hyacinto Boneu hortola de Bara fill de Juan Boneu parayre defunct y de Maria ab Anna donsella filla de t Cases

Predicted: other other other other other name surname occupation other location other other name surname occupation other other othe
Correct:   other other other other other name surname occupation other location other other name surname occupation other other othe
```

```
info_crf = tagger_crf.info()

print("Top likely transitions:")
print_transitions(Counter(info_crf.transitions).most_common(15))

print("\nTop unlikely transitions:")
print_transitions(Counter(info_crf.transitions).most_common()[-15:])
```

```
Top likely transitions:
location -> location 2.872853
surname -> occupation 2.636955
name   -> surname 2.611067
occupation -> occupation 2.335944
surname -> surname 2.207959
other  -> other   2.169349
state  -> state   1.610926
other  -> name    1.419328
name   -> name    1.170603
other  -> location 0.408166
state  -> other   0.360116
state  -> occupation 0.346188
occupation -> other   0.180175
name   -> state   0.072084
surname -> other   -0.014152

Top unlikely transitions:
surname -> location -0.887503
other  -> occupation -0.904658
state  -> surname -0.913424
other  -> state   -1.378395
location -> other   -1.643350
other  -> surname -2.051393
location -> occupation -2.248958
occupation -> location -3.047927
name   -> occupation -3.149253
occupation -> name    -3.591631
name   -> location -4.179267
location -> surname -4.762295
surname -> name    -5.028702
location -> name    -5.121750
occupation -> surname -5.413854
```

```
print("Top positive:")
print_state_features(Counter(info_crf.state_features).most_common(20))

print("\nTop negative:")
print_state_features(Counter(info_crf.state_features).most_common()[-20:])
```

```
Top positive:
11.351549 other  prev_word.lower=lomar
8.595487 name   prev_word.lower=ab
8.340087 occupation word[-3:]=yre
7.834604 other  prev_word.lower=bruneta
7.318131 location prev_word.lower=pedro
7.240423 other  bos
```

```
7.049330 other  word[:3]=def
6.882273 name   word[-3:]=eth
6.537952 occupation next_word.lower=corredor
6.105554 surname word[:3]=Bos
6.096146 other  word[:3]=fil
6.054197 location word[:3]=mal
5.975986 occupation word[-3:]=ter
5.960622 other  prev_word.lower=heredia
5.894165 location prev_word.lower=lletget
5.700476 occupation word[-3:]=tor
5.692651 other  next_word.lower=bara
5.666594 occupation word[-3:]=dor
5.660061 occupation word.lower=dosell
5.504777 other  word.lower=vallenti

Top negative:
-2.307631 state  prev_word.lower=clara
-2.314522 other  prev_word.lower=defucta
-2.350244 name   next_word.lower=y
-2.375177 location word[:3]=Con
-2.426815 other  prev_word.lower=pere
-2.450522 surname word[:3]=Jua
-2.505918 other  prev_word.lower=bisbat
-2.598547 other  prev_word.lower=anna
-2.638708 surname word.lower=valls
-2.922583 surname word[:3]=Gir
-2.998828 other  prev_word.lower=desvern
-3.231384 other  prev_word.lower=viudo
-3.274620 other  next_word.lower=ab
-3.293265 occupation prev_word.lower=bassa
-3.445616 other  prev_word.lower=de
-3.543070 other  prev_word.lower=sella
-3.613124 occupation prev_word.lower=roca
-3.699285 occupation prev_word.lower=guitart
-3.791194 occupation prev_word.lower=bertran
-5.399048 occupation prev_word.lower=pedro
```

Our conclusions refering to CRF approach:

1. **Where does the CRF perform better than the baseline approach?**

As we can see, the CRF outperforms the baseline in keeping the label sequences consistent, such as location-to-location (2.87) and surname-to-occupation (2.64). This is similar to the HMM, which also performed well in handling location labels (5.06 weight for location-to-location). The CRF also excels at recognizing historical terms like "lomar" (positive weight 11.35), just as the HMM did with words like "viudo" and "donsella".

2. **What words does the CRF still struggle with?**

The CRF still has trouble with common words like "de" and rare personal names like "viudo" (-3.23). This is similar to the HMM, which also struggled with names and words that could be either locations or surnames. The CRF is a bit better at handling location and occupation terms, but still struggles with ambiguous words.

3. **How differently does the model perform w.r.t. Out-of-Vocabulary words?**

The CRF handles OOV words better than the baseline, especially for location and occupation terms. This is similar to the HMM's good performance on location and occupation words. However, rare words like "viudo" are still a challenge for the CRF, just like they were for the HMM.