Problems L5

March 29, 2025

Anna Blanco (1709582), Queralt Salvadó (1706789)

1 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.

1.1 First, we will install the unmet dependencies.

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

[62]:
```
#@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 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 pycrfsuite as crfs
from sklearn_crfsuite import scorers
from sklearn_crfsuite import metrics

from resources.data.dataloaders import EsposallesTextDataset
```

1.2 Data Processing

Loading the dataset

[63]:
```
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

[64]:
```
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:other',
'Bara:location', 'fill:other', 'de:other', 'Juan:name', 'Boneu:surname',
'parayre:occupation', 'defunct:other', 'y:other', 'de:other', 'Maria:name',
'ab:other', 'Anna:name', 'donsella:state', 'filla:other', 'de:other', 't:name',
'Cases:surname', 'pages:occupation', 'de:other', 'Bara:location',
'defunct:other', 'y:other', 'de:other', 'Peyrona:name']
['Divendres:other', 'a:other', '18:other', 'rebere:other', 'de:other',
'Juan:name', 'Torres:surname', 'pages:occupation', 'habitant:other', 'en:other',
'Sabadell:location', 'fill:other', 'de:other', 'Bernat:name', 'Torres:surname',
'pages:occupation', 'de:other', 'Moya:location', 'bisbat:location',
'de:location', 'Vich:location', 'y:other', 'de:other', 'Antiga:name',
'defucts:other', 'ab:other', 'Margarida:name', 'donsella:state', 'filla:other',
'de:other', 'Juan:name', 'Argemir:surname', 'pages:occupation', 'de:other',
'Sabadell:location', 'y:other', 'de:other', 'Aldonsa:name', 'defuncts:other']
```

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

[65]:
```
# Check Dataset

# Prints are commented to avoid long outputs in the report!
```

```
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")
```

1.3 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 succintly 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 unkown 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.

[66]:
```
def create_tokens_lut(train_dataset: EsposallesTextDataset) -> Dict[str, int]:
    """Create a LUT that maps each word to an index.

    Construct a dictionary that takes a word as input and converts this word␣
 ↪into an int
```

```
    index. The indices must be unique and all words in the training dataset␣
 ↪should have
    one. Do not forget to incorporate extra tokens for beginning of sequence␣
 ↪(<bos>),
    end of sequence (<eos>) and unknown token (<unk>).

    Parameters
    ----------
    train_dataset : EsposallesTextDataset
        Train data container that provides words and tokens one at a time.

    Returns
    -------
    Dict[str, int]
        The LUT for words in the dataset.
    """
    vocab = set()

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


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

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

    return lut

lut = create_tokens_lut(train_loader)
```

[67]:
```
def check_oov_words(lut: Dict[str, int], test_set: EsposallesTextDataset) ->␣
 ↪List[str]:
    """Find all out-of-vocabulary words in the test partition.

    Parameters
    ----------
    lut : Dict[str, int]
        LUT you have calculated in the previous exercise with all of the words
        present in the training partition.
    test_set : EsposallesTextDataset
```

```
        Test partition of the Esposalles dataset.

    Returns
    -------
    List[str]
        A list of words that are not found in the training partition.
    """
    oov_words = set()

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

    return sorted(oov_words)
```

[68]:
```
print(list(check_oov_words(lut, test_loader)))
```

```
['Alaverni', 'Angli', 'Arevig', 'Argemir', 'Arisart', 'Assensio', 'Auger',
'Bachs', 'Begas', 'Berenguer', 'Blanquart', 'Bonastra', 'Box', 'Brasil',
'Broquets', 'Busquet', 'Buyra', 'Cabaner', 'Cabrer', 'Cabus', 'Campanya',
'Campderos', 'Campprecios', 'Cani', 'Carantela', 'Castellterçol', 'Castigaleu',
'Cebriana', 'Celles', 'Comi', 'Conflent', 'Constansa', 'Conteso', 'Corties',
'Crich', 'Darder', 'Deseny', 'Despi', 'Dimarts', 'Dimas', 'Garces', 'Gassull',
'Gatuellas', 'Ge', 'Glandina', 'Guardi#', 'Gusman', 'Honorat', 'Idrach',
'Islla', 'Juan#', 'Llobre', 'Llondra', 'Llorenci', 'Luciana', 'Macip', 'Majol',
'Manader', 'Mandri', 'Masseres', 'Melcior', 'Mercader', 'Miro', 'Monblanch',
'Monllor', 'Morros', 'Munmany', 'Muntells', 'Noguera', 'Novell', 'Pachs',
'Pallissa', 'Payas', 'Per', 'Peramon', 'Perris', 'Plans', 'Planta', 'Poses',
'Quart', 'Ratx', 'Ribagossa', 'Ritoreta', 'Rius', 'Rosa', 'Rotxe', 'Sengermes',
'Sitjar', 'Sobrevila', 'Spa', 'Sto', 'Tamuyell', 'Tarafa', 'Tatare', 'Terre',
'Testa', 'Theodora', 'Thome', 'Tibau', 'Valta', 'Victo', 'Vilademaser',
'Villaro', 'aguller', 'archebisbat', 'ayguardenter', 'basso', 'caxaler',
'clavetayre', 'comptat', 'deBarca', 'fabrega', 'fabres', 'faja', 'faliu',
'faneca', 'fargayre', 'febres', 'felis', 'fontanilles', 'francesa', 'galeras',
'gat', 'habitants', 'hor', 'imaginayre', 'jonqueres', 'more', 'moreno', 'mso',
'nyella', 'ortiz', 'pas', 'pinya', 'pobla', 'raguera', 'rianna', 'scrivent',
'tisser', 'tola', 'vivints']
```

[69]:
```
MAX_SEQUENCE_LENGTH = 50

def pad_sentence(sent: List[str], max_sent_len: int) -> List[str]:
    """Insert <bos>, <eos> in the sentence and add <pad> until max length.

    If ``sent`` is lengthier than ``max_sent_len``, truncate the sentence and␣
 ↪add the
```

```
    final <eos>.

    Parameters
    ----------
    sent : List[str]
        An arbitrary length list of words representing a sentence.
    max_sent_len : int
        The desired max padding length.

    Returns
    -------
    List[str]
        The sentence with additional tokens and fixed width.
    """
    padded = ['<bos>'] + sent[:max_sent_len - 2] + ['<eos>']
    pad_len = max_sent_len - len(padded)
    padded += ['<pad>'] * pad_len
    return padded

def apply_lut(lut: Dict[str, int], sent: List[str]) -> List[int]:
    """Convert words to indices using the LUT.

    Parameters
    ----------
    lut : Dict[str, int]
        LUT you have calculated in the previous exercise with all of the words
        present in the training partition.
    sent : List[str]
        An arbitrary length list of words representing a padded sentence.

    Returns
    -------
    List[int]
        Same sentence as the input after applying the LUT.
    """
    return [lut.get(word, lut.get('<unk>', 0)) for word in sent]

# Since this exercise is for illustratory purposes, we will not care much about␣
 ↪the
# tags for now. If you wanted to do this for a real application, the tags would␣
 ↪need
# their own lut as well and the same padding would need to be applied on the␣
 ↪source and
# target sequences.
tokenised_train = []
for x, _ in train_loader:
    # You could incorporate processing at this stage as well if you want
```

```
    # x = process(x)
    sent = pad_sentence(x, MAX_SEQUENCE_LENGTH)
    sent = apply_lut(lut, sent)

    tokenised_train.append(sent)
```

1.4 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

[70]:
```
from collections import defaultdict
from typing import Dict

def compute_tag_prob(train_loader: EsposallesTextDataset) -> Dict[str,␣
 ↪Dict[str, float]]:
    """Compute the probability of each tag conditioned on the observed word.

    The dictionary should look like:

    >>> output = {
    >>>     "word": {
    >>>         "other": 0.1,
    >>>         "name": 0.8,
    >>>         "surname": 0.8,
    >>>     },
    >>> }

    Parameters
    ----------
    train_loader : EsposallesTextDataset
        Train data container that provides words and tokens one at a time. It␣
 ↪returns the
        list of words and tags of each sentence:

        >>> x, y = train_dataset[index]
        x = ['Dilluns', 'hyacinto', ...]
        y = ['other', 'name', ...]

    Returns
    -------
    Dict[str, Dict[str, float]]
```

```
        A dictionary whose keys are the words in the training corpus and whose␣
↪values
        are the probabilities of each tag inserted in a dictionary.
    """
    # Initialize a dictionary to store counts of tags for each word
    word_tag_counts = defaultdict(lambda: defaultdict(int))
    tag_counts = defaultdict(int)

    # Iterate over all sentences in the training dataset
    for i in range(len(train_loader)):
        x, y = train_loader[i]
        for word, tag in zip(x, y):
            word_tag_counts[word][tag] += 1
            tag_counts[tag] += 1

    # Convert counts to probabilities
    word_tag_prob = {}
    for word in word_tag_counts:
        total_occurrences = sum(word_tag_counts[word].values())
        word_tag_prob[word] = {
            tag: count / total_occurrences
            for tag, count in word_tag_counts[word].items()
        }

    return word_tag_prob
```

[71]:
```
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` $) = 9\%$

[72]:
```
tag_probs["Prats"]
```

[72]:
```
{'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.

[73]:
```
# Code to find the most frequent tag in the train set -> used to handle OOW

from collections import defaultdict

def find_most_frequent_tag(train_set):
    tag_count = defaultdict(int)  # Initialize a dictionary to count tag␣
 ↪occurrences

    # Iterate through the training set and count the tags
    for i in range(len(train_set)):
      x, y = train_set[i]

    for tag in y:
            tag_count[tag] += 1

    # Find the tag with the highest count
    most_frequent_tag = max(tag_count, key=tag_count.get)

    return most_frequent_tag

print("Most frequent tag:", find_most_frequent_tag(train_loader))
```

```
Most frequent tag: other
```

[74]:
```
def predict_test_set(
    tag_probs: Dict[str, Dict[str, float]],
    test_set: EsposallesTextDataset,
) -> List[List[str]]:
    """Implement the MLE algorithm for sequence tagging using the training tag␣
 ↪probs.

    Remember to consider cases such as OOV words.

    Parameters
    ----------
    tag_probs : Dict[str, Dict[str, float]]
        Dictionary of probabilities of each word to be of a certain tag.
    test_set : EsposallesTextDataset
```

```
        Test data container that provides words and tokens one at a time. It␣
 ↪returns the
        list of words and tags of each sentence:

        >>> x, y = train_dataset[index]
        x = ['Dilluns', 'hyacinto', ...]
        y = ['other', 'name', ...]

    Returns
    -------
    List[List[str]]
        A list of sentence-level predictions, which are in turn modeled as a␣
 ↪list of
        tags for each word in the sentence.
    """
    test_predictions = []

    # Iterate through the test set
    for i in range(len(test_set)):
      x, y = test_set[i]
      predictions = []

      for word in x:

        if word in tag_probs.keys():
          #find tag with highest prob
          probabilities = tag_probs[word]
          predicted_tag = max(probabilities, key=probabilities.get)

        else:
          predicted_tag = "other"

        predictions.append(predicted_tag)

      test_predictions.append(predictions)

    return test_predictions

def find_common_errors(
    test_set: EsposallesTextDataset,
    test_predictions: List[str],
) -> Any:
    """Count how many times does some kind of error happen.

    You can copy this function signature as many times as you want to create␣
 ↪functions
    that evaluate specific errors in the output.
```

```
    Returns
    -------
    Any
        RETURN WHAT YOU WANT THAT YOU CAN USE TO EVALUATE THE MODEL.
        Some suggestions:
        - Dict[str, Dict[str, int]] -> Counts of every time a tag is confused␣
 ↪by another
        - Dict[str, int] -> Counts of which word is wrong
        - Dict[str, int] -> Counts of which tag is wrong
    """
    common_errors = defaultdict(int) # initialize with default vale of 0

    # Iterate through the test set
    for i in range(len(test_set)):
      x_true, y_true = test_set[i]
      y_pred = test_predictions[i]

      for real_tag, predicted_tag in zip(y_true, y_pred):

        # compare the predicted tag with the real one
        if predicted_tag != real_tag:
          # if incorreclty predicted increase dictionary count
          common_errors[real_tag] += 1

    # return a dictionary with the counts of which tag is wrong
    return dict(common_errors)

def compute_token_precision(
    test_set: EsposallesTextDataset,
    test_predictions: List[str],
) -> float:
    """Compute how many times the prediction of the model is the same as the GT.

    Parameters
    ----------
    test_set : EsposallesTextDataset
        Test data container that provides words and tokens one at a time. It␣
 ↪returns the
        list of words and tags of each sentence:

        >>> x, y = train_dataset[index]
        x = ['Dilluns', 'hyacinto', ...]
        y = ['other', 'name', ...]
    test_predictions : List[str]
        List of labels predicted by our MLE model.
```

```
    Returns
    -------
    float
        Precision computation for the full dataset.
    """
    # initialize counts to 0
    correct_predictions = 0
    total_predictions = 0

    # Iterate through the test set
    for i in range(len(test_set)):
      x_true, y_true = test_set[i]
      y_pred = test_predictions[i]

      for real_tag, predicted_tag in zip(y_true, y_pred):
        total_predictions += 1

        # compare the predicted tag with the real one
        if predicted_tag == real_tag:
          correct_predictions += 1

    precision = correct_predictions / total_predictions

    return precision

test_predictions = predict_test_set(tag_probs, test_loader)
err1 = find_common_errors(test_loader, test_predictions)
print("Most common errors:", err1)
precision = compute_token_precision(test_loader, test_predictions)
print("Precision:", precision)
```

```
Most common errors: {'location': 133, 'surname': 128, 'name': 27, 'other': 12,
'occupation': 39, 'state': 6}
Precision: 0.8889604119729643
```

After finishing implementing the requested functions we obtained a **precision of 88%**, and concluded that the most commmonly missclassified tag was `location`. However, to further investigate the algorithm we will implement a function to create a **confusion matrix** from the test set and answer the requested questions.

[75]:
```
import pandas as pd

def generate_confusion_matrix(test_set, test_predictions, all_tags):

    cm = defaultdict(lambda: defaultdict(int))

    for i in range(len(test_set)):
      x_true, y_true = test_set[i]
```

```
      y_pred = test_predictions[i]

      for y_true, y_pred in zip(y_true, y_pred):
        cm[y_true][y_pred] +=1

    # Convert to DataFrame for readability
    cm_df = pd.DataFrame.from_dict(cm, orient='index').fillna(0).astype(int)
    # Ensure all tags are included (even if zero)
    cm_df = cm_df.reindex(index=all_tags, columns=all_tags, fill_value=0)

    return cm_df


all_tags = ["location", "name", "occupation", "other", "state", "surname"]
cm = generate_confusion_matrix(test_loader, test_predictions, all_tags)
print("Confusion Matrix:")
print(cm)
```

```
Confusion Matrix:
            location  name  occupation  other  state  surname
location         329    11           1    111      0       10
name               5   467           1     15      0        6
occupation         1     0         255     37      0        1
other              3     4           1   1481      2        2
state              0     0           0      6    107        0
surname           13    13           2    100      0      123
```

1. What do all of the mistakes have in common?

By looking at the off-diagonals of the confusion matrix, we see that many tags (e.g., surname, location, name, occupation) are often confused with `other`. This makes sense, since our approach to classify out-of-vocabulary words consisted on using the most frequent tag (`other`) for every unknown word. Therefore, in all categories (tags) we have missclassifications with the tag `other`.

Additionally, we see frequent confusion between name and surname. This is due to the fact that our classification relies solely on prior probabilities rather than contextual information. Since some names and surnames can overlap or be used interchangeably, distinguishing between them requires understanding the surrounding words in the sentence, something our current approach does not account for.

2. What kinds of words are the least performers? Looking at the confusion matrix, we can determine that the `state` and `surname` categories seem to be the least performers:

> `State` has few correct predictions (107 on the diagonal), and it's frequently misclassified as `other` or other categories.
>
> `Surname` also has significant misclassifications (13 to name, 13 to surname, 100 to other).

The fact that `state` and `surname` are the least well-performing categories, can be due to the fact that:

The dataset includes a wide variety of personal and historical names, where state tags are less

frequent and more difficult to identify. In historical texts or records, such as the one we are using, there can be different ways of referring to locations or statuses, which can lead to confusion between state and location or other tags.

Many surnames are commonly used as first names. For example, in cases where a person's full name includes a surname that could also be a common first name, the model struggles to differentiate between the two categories.

It is also important to note that `location` also shows some confusion, this is because some locations can be mislabelled as names or surnames, for instance "Bara" or "Vila" can be both a surname or a location, the only way we have of differentiating it is using the context.

3. What's your solution for out-of-vocabulary words? Can you provide a prediction for those? To deal with out-of-vocabulary words, we assigned them the most common tag in the data, therefore being able to provide a prediction. However, this method is not very accurate since many words end up being misclassified as `other`. Despite this, we were still able to obtain a proper accuracy. A better approach could be using word embeddings, which help understand word meanings based on their context, to predict the correct tags more accurately.

4. What words are usually the best performers? The best-performing words in the confusion matrix are those with the highest correct classifications (diagonal values) and the lowest misclassifications (off-diagonal values).

The `other` category performs the best, with 1481 correct classifications and very few misclassifications. This is explained by the solution that we provided to OOV words, for example, words like "fill" that are not part of the common tags in the dataset, are correctly labeled as `other`, helping improve the performance in this category.

Similarly, `name` (467 correct) and `occupation` (255 correct) also perform well. For instance, in our dataset, names like "Joan" and "Pere" are frequently classified correctly as `name`. Likewise, words like "pastisser" are correctly identified as occupation. These categories perform well because they follow clear patterns in the dataset, with names and occupations being easily recognizable and distinct from other types of words.

1.5 HMM Approach

As demonstrated in the previous experiment, using just the priors have not enough expresivity 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?

[76]:
```python
def get_word_to_hmm_features(sent: List[Tuple[str, str]], i: int) -> List[str]:
    """This function computes CRF features to generate a HMM from a word.

    Parameters
    ----------
    sent : List[str]
        Sentence to extract features from.
    i : int
        Index of the word whose features should be extracted.

    Returns
    -------
    List[str]
        A list of features for the i-th word.
    """
    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]
```

[77]:
```
# 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]
```

[78]:
```
trainer = crfs.Trainer(verbose=False)  # Instance a CRF trainer

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.

[79]:
```
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,
        "num_memories": 6,  # Number of previous updates kept for convergence␣
 ↪detection
        "epsilon": 1e-5,  # Tolerance for stopping criteria
    }
)
```

We added `num_memories` so that the optimizer retains some memory of the previous 6 updates, if increased too much it can lead to a smoother convergence but to a slower computation, which is why we set it to *6* in order to find a trade-off between the smoother convergence and a reasonable computation time.

We also added `epsilon` which controls when the optimization algorithm should stop training. Specifically, it defines the minimum change in the objective function between iterations before stopping. If the model function is lower than epsilon, the algorithm assumes the model has converged and stops early.

[80]:
```
%%time
trainer.train('npl_ner_hmm.crfsuite') # Train the model and save it locally.
```

```
CPU times: user 243 ms, sys: 3.11 ms, total: 246 ms
Wall time: 246 ms
```

[81]:
```
tagger = crfs.Tagger()
tagger.open("npl_ner_hmm.crfsuite")  # Load the inference API
```

[81]:
```
<contextlib.closing at 0x1571a4450>
```

[82]:
```
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 pages de Bara defunct y de
Peyrona

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

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

[83]:
```
from collections import Counter

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
```

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

The most likely transition is the one from `surname` to `occupation` (5.01). This makes sense given the structured nature (as mentioned above) of historical marriage records. Which typically follow a predictable format, often listing a person's full name first (name -> surname), followed by their occupation (surname -> occupation).

It also has a high probability transition from `location` to `location`(4.25) and `occupation` to `occupation` (4.19). This means that if a word is identified as a location or occupation, the HMM is less likely to switch the label mid-sentence. In contrast, CRF relies more on individual word features, so it sometimes misclassifies long sequences.

> For instance, in the sentence "… ciutat de Barcelona." we would want to label "ciutat de Barcelona", all together, as a location (location -> location).
>
> Another example would be in the sentence: "Pere Roca, mestre de cases de Barcelona.", where "mestre de cases" should be labelled as a single occupation (occupation -> occu-

pation), rather than tagging “mestre” as an occupation and “de”, “cases” as something else.

The HMM also has a strong probability transition from `name` to `surname` (3.9) which means that it has learned that surnames often follow names. Which is logical due to the dataset that we are using, where names are listed before surnames.

Other transitions that are also pretty common are the ones from `surname`to `surname`, `name`to `name`, or `other`to `name`.

For instance, “Joan Ferrer i Vila” contains two consecutive surnames, reinforcing the surname -> surname transition.

First names frequently appear in succession when listing family members, leading to the name -> name transition.

Lastly, the other -> name transition is expected since generic words like “fill de” or “amb” often precede a person’s name.

[84]:
```
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
```

2. Where does the HMM perform better than the baseline approach? The HMM model outperforms the baseline approach when predicting unknown words in our dataset, which consists of structured and predictable texts.

Since the HMM approach relies on **transition probabilities** rather than specific word features, it can generalize better to **unseen words**, making it more robust in historical records, such as the one we are using (historical record of marriages), where spelling variations and rare terms are common.

Additionally, our dataset follows a **fixed structure**, where names, surnames, occupations, and locations often appear in a structured sequence. Therefore, allowing the HMM to strengthen sequence patterns for more accurate predictions.

> *Each marriage license contains information about the husband's occupation, husband's and wife's former marital status, socioeconomic position signaled by the fee imposed on them, and in some cases, fathers' occupations, place of residence or geographical origin.*
> *- extracted from: http://dag.cvc.uab.es/the-esposalles-database/*

In contrast, the NER model, which heavily depends on word features, struggles with rare or unseen words and lacks the same ability to enforce structured labeling.

3. What words does it struggle with? The HMM struggles with words that do not follow the predicted structure or transitions learned. As well as unknown words or words that do not appear that frequently in the dataset, such as "menat" or "pastisser".

Additionally, it can also struggle with alternative spelling of names or locations. For instance, "Esposalles" might sometimes be written as "Esposallès" or "Esposalles", which could cause the model to misclassify it as a location.

Lastly, common words like “de”, “i”, or “amb” that appear frequently in these records are difficult for the HMM to classify, as they don’t carry significant meaning or have clear transitions to another tag.

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

The HMM model handles Out-of-Vocabulary words better than our baseline approach because it focuses on transitions between labels rather than relying on specific word features. When the HMM encounters a new word, it can use the surrounding context to make a good guess based on patterns learned during training. For example, if it comes across a new name or occupation, it can apply its knowledge of past transitions (like surname -> occupation) to predict the label accurately.

In contrast, with our first approach, any unknown word had to be classified as `other`, which limited the model’s ability to make accurate predictions and led to a lot of mislabeling, with many words incorrectly tagged as `other`.

The HMM is particularily useful for out-of-vocabulary words in this dataset, since as mentioned above, words appear in a more or less structured sequence. Therefore, the transitions learned are really useful to predict them.

1.6 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.

[85]:
```python
def get_word_to_crf_features(sent: List[Tuple[str, str]], i: int) -> List[str]:
    """This function computes CRF features from a word.

    Parameters
    ----------
    sent : List[str]
        Sentence to extract features from.
    i : int
        Index of the word whose features should be extracted.

    Returns
    -------
    List[str]
        A list of features for the i-th word.
    """
    word, _ = sent[i]

    features = [
        "bias",
        "word.lower=" + word.lower(),
        "word[-2:]=" + word[-2:],
        "word.istitle=%s" % word.istitle(),
        "word.isdigit=%s" % word.isdigit(),
    ]
```

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

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

[86]:
```
crf_trainer = crfs.Trainer(verbose=False)  # Instance a CRF trainer

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

[87]:
```
# Set CRF training parameters
crf_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,
    }
)
```

[88]:
```
%%time
trainer.train('npl_ner_crf.crfsuite') # Train the model and save it locally.
```

```
CPU times: user 241 ms, sys: 2.54 ms, total: 243 ms
Wall time: 243 ms
```

The two functions, `get_word_to_hmm_features` and `get_word_to_crf_features`, are designed to

extract features from words in a sentence, but they differ in complexity and purpose. The HMM (Hidden Markov Model) version is much simpler: it only includes the lowercase version of the current word and flags if it's the beginning or end of a sentence. This minimalistic approach is quite fast and that's reflected in the CPU and wall times.

In contrast, the CRF (Conditional Random Field) version adds much more context. It not only uses the current word's lowercase form but also checks if the word is capitalized, if it's numeric, and includes the last two letters of the word. It also looks at features from neighboring words (previous and next), which increases both its contextual richness and computational load. This richer feature set helps CRFs make more informed predictions.

Surprisingly, despite CRF performing more work, it is slightly faster. While the improvement isn't huge in this example, it still outperforms the HMM implementation.

[89]:
```
tagger = crfs.Tagger()
tagger.open("npl_ner_crf.crfsuite")

y_pred = [tagger.tag(xseq) for xseq in X_test]

from sklearn_crfsuite import metrics

print("CRF Accuracy:", metrics.flat_accuracy_score(y_test, y_pred))
print(metrics.flat_classification_report(y_test, y_pred, digits=3))
```

```
CRF Accuracy: 0.9732724902216427
              precision    recall  f1-score   support

    location      0.952     0.952     0.952       416
        name      0.982     0.982     0.982       487
  occupation      0.929     0.973     0.950       296
       other      0.988     0.981     0.984      1486
       state      0.966     0.982     0.974       114
     surname      0.966     0.944     0.955       269

    accuracy                          0.973      3068
   macro avg      0.964     0.969     0.966      3068
weighted avg      0.974     0.973     0.973      3068
```