L5: Sequence Labelling

Frederik Evenepoel & Piotr Bonar - Pair K
30/03/2025

The full implementation of the code can also be directly seen in the notebook:
https://colab.research.google.com/drive/1wTHwb-yvpo5zDzwRHktRInO3Npo5MhHR?usp=sharing

Graded exercise 1:

In this exercise we will build a pre-processing pipeline for a NER using a LUT-based tokenizer. We'll implement functions to create a word-to-index mapping, pad sentences to a fixed length, and convert them into index sequences, which we afterwards will test with a small example. Afterward, we'll apply the tokenizer to the actual dataset, identify out-of-vocabulary words in the test set, and analyse them in the context of historical Catalan marriage records.

TODO 1.1: Write the LUT computation. Ensure to incorporate the various additional tokens in the process.

In this step, we implemented the function to create a LUT that assigns a unique integer index to every unique word in the training dataset. The LUT also includes special tokens *<bos>*, *<eos>*, and *<unk>* to handle sentence boundaries, and out-of-vocabulary words, enabling the conversion of raw text into a consistent numerical format for further processing. The output from the implementation/code below looks like this: *{'<bos>': 0, '<eos>': 1, '<unk>': 2, 'Dilluns': 3, 'a': 4, '5': 5, 'rebere': 6, 'de': 7, 'Hyacinto': 8, 'Boneu': 9, 'hortola': 10, 'Bara': 11, 'fill': 12, 'Juan': 13, 'parayre': 14, 'defunct': 15, (...)}.* We have in total 2291 entries in the dictionary.

TODO 1.1 - Code implementation:

```
def create_tokens_lut(train_dataset: EspossallesTextDataset) -> Dict[str, int]:

    # Define special tokens and give them a directly an index
    special_tokens = ["<bos>", "<eos>", "<unk>"]
    lut = {token: idx for idx, token in enumerate(special_tokens)}
    current_index = len(special_tokens)

    # Next step is to loop over training set and assign index to unique words
    for i in range(len(train_dataset)):
        x, _ = train_dataset[i]
        for word in x:
            if word not in lut:
                lut[word] = current_index
                current_index += 1

    return lut


lut = create_tokens_lut(train_loader)
print(lut)
print(len(list(lut)))
```

TODO 1.2: Write the Out-of-vocabulary word checking function.

In this part, we implemented a function to identify out-of-vocabulary words in the test set by comparing each word against the LUT. The function returns a list of unique words from the test set that were not seen during training. The output from the implementation/code below looks like this: *['Gassull', 'Corties', 'Sobrevila', 'Gusman', 'Masseres', 'Celles', 'Sitjar', 'Rotxe', 'Alaverni',…]*. In total we have an amount of 141 unique OOV words.

TODO 1.2 - Code implementation:

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

    oov_words = set()

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

    return list(oov_words)

print(list(check_oov_words(lut, test_loader)))
print(len(list(check_oov_words(lut, test_loader))))
```

TODO 1.3: Write functions to pad sentences and to apply the LUT on the padded sentences.

The following code prepares the training data by padding each sentence to a fixed length of 50 tokens. It adds the special tokens <bos>, <eos>, and <pad> as needed, and truncates longer sentences. Then, it converts the words into numerical indices using the LUT, resulting in a tokenized data set ready for serve as input to a model.

A part of the output looks like this:

Example 1:
Original sentence: ['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']

Padded sentence: ['<bos>', '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', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>']

Tokenized sentence: [0, 3, 4, 5, 6, 7, 8, 9, 10, 7, 11, 12, 7, 13, 9, 14, 15, 16, 7, 17, 18, 19, 20, 21, 7, 22, 23, 24, 7, 11, 15, 16, 7, 25, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

Example 2 :
Original sentence: ['dit', 'dia', 'rebere', 'de', 'Bernat', 'Call', 'pages', 'de', 'St', 'Esteva', 'de', 'Palau', 'tordera', 'fill', 'de', 'Guille', 'Call', 'pages', 'y', 'de', 'Catherina', 'defunct', 'ab', 'Maria', 'donsella', 'filla', 'de', 'Guillem', 'Boix', 'pages', 'de', 'la', 'Mora', 'bisbat', 'de', 'Vich', 'y', 'de', 'Margarida']

Padded sentence: ['<bos>', 'dit', 'dia', 'rebere', 'de', 'Bernat', 'Call', 'pages', 'de', 'St', 'Esteva', 'de', 'Palau', 'tordera', 'fill', 'de', 'Guille', 'Call', 'pages', 'y', 'de', 'Catherina', 'defunct', 'ab', 'Maria', 'donsella', 'filla', 'de', 'Guillem', 'Boix', 'pages', 'de', 'la', 'Mora', 'bisbat', 'de', 'Vich', 'y', 'de', 'Margarida', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>']

Tokenized sentence: [0, 26, 27, 6, 7, 28, 29, 24, 7, 30, 31, 7, 32, 33, 12, 7, 34, 29, 24, 16, 7, 35, 15, 18, 17, 20, 21, 7, 36, 37, 24, 7, 38, 39, 40, 7, 41, 16, 7, 42, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2]

TODO 1.3 - Code implementation:

```
MAX_SEQUENCE_LENGTH = 50

def pad_sentence(sent: List[str], max_sent_len: int) -> List[str]:
    padded = ["<bos>"] + sent

    if len(padded) >= max_sent_len:
        padded = padded[:max_sent_len - 1]
        padded.append("<eos>")
    else:
        padded.append("<eos>")
        while len(padded) < max_sent_len:
            padded.append("<pad>")

    return padded


def apply_lut(lut: Dict[str, int], sent: List[str]) -> List[int]:
    unk_index = lut.get("<unk>")
    return [lut.get(word, unk_index) 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 idx in range(len(train_loader)):
    sent, _ = train_loader[idx]
    print(f"Original sentence: {sent}")
    sent = pad_sentence(sent, MAX_SEQUENCE_LENGTH)
    print(f"Padded sentence: {sent}")
    sent = apply_lut(lut, sent)
    print(f"Tokenized sentence: {sent}")
    tokenised_train.append(sent)
```

Analysis

1. **Full pipeline working principle with handcrafted example:**

   We demonstrate the full LUT-based tokenization pipeline using a handcrafted sentence that is not randomly chosen but lies in our domain of the dataset: *["Maria", "es", "va", "casar", "amb",* *"Joan", "a", "Barcelona", "l'any", "1850"]*. This sentence is first padded to a fixed length of 15 tokens by adding <bos> at the beginning, <eos> at the end, and <pad> tokens to fill the remaining space. The padded result is then converted to a sequence of integers using the LUT, where each word is replaced by its corresponding index, and unseen words are mapped to the <unk> token. We can verify from output below that it works like intended.

Output:

```
Padded Sentence: ['<bos>', 'Maria', 'es', 'va', 'casar', 'amb', 'Joan', 'a', 'Barcelona', 'l'any', '1850',
'<eos>', '<pad>', '<pad>', '<pad>']
Indexed Sentence: [0, 17, 1946, 2, 2, 2, 197, 4, 1834, 2, 2, 1, 2, 2, 2]
```

Code implementation:

```
#Handcrafted example
example_sent = ["Maria", "es", "va", "casar", "amb", "Joan", "a", "Barcelona",
"l'any", "1850"]
MAX_SENT_LEN = 15

padded_example = pad_sentence(example_sent, MAX_SENT_LEN)
print("Padded Sentence:", padded_example)

indexed_example = apply_lut(lut, padded_example)
print("Indexed Sentence:", indexed_example)
```

2. **Generate the LUT on the training data of the NER task:**

Here we generate the LUT on the training data (again with including the special tokens), and prints its size and a few samples of the entries inside the LUT.

Output:

```
Vocabulary size: 2291
Example tokens in LUT (first 10): [('<bos>', 0), ('<eos>', 1), ('<unk>', 2), ('Dilluns', 3), ('a', 4), ('5',
5), ('rebere', 6), ('de', 7), ('Hyacinto', 8), ('Boneu', 9)]
```

Code implementation:

```
#Generate the LUT on the training data of the NER task.
lut = create_tokens_lut(train_loader)
print("Vocabulary size:", len(lut))
print("Example tokens in LUT:", list(lut.items())[:10])
```

3. **Find all out-of-vocabulary words in the test partition. Study what kinds of words they are:**

In our data set, *Esposalles* (a set of historical Catalan marriage records), most of the OOV words are nouns, such as names of people and places, which naturally vary a lot across records. Since personal names tend to be unique or low frequency, it's rare for the exact same names to appear in both the training and test sets, especially in a historical dataset where standardised spelling wasn't always followed. Additionally, the dataset includes regional place names, archaic terms (e.g., *imaginayre*), and noise/errors (e.g., *Guardi#*, *Juan#),* which therefore also add to the total OOV words.

Code implementation:

```
oov_words = check_oov_words(lut, test_loader)
print("Number of OOV words:", len(oov_words))
print("Sample OOV words:", oov_words[:10]) #Print 10 examples
```

Graded exercise 2:

We will implement a baseline maximum likelihood estimator by computing the probabilities *P(tag| word)* from the training data and using them to predict tags in the test set. We will then analyse the results to determine in which circumstances the model performs well and where it tends to fail, using metrics for NER such as the confusion matrix. We will also examine the most common types of errors and explain how we could handle OOV words more in a different way.

TODO 2.1: Write the 'compute_tag_prob' function.

We made the *compute_tag_prob* function by first counting how many times each tag appears for every word in the training dataset using a defaultdict of Counter. Then, for each word, we computed the probability distribution over tags by normalising the tag counts, resulting in *P(tag|word)*. We ensured that the output is a nested dictionary where each word maps to a dictionary of tag probabilities, as required. We evaluated the system with the hint given in the notebook namely: "The emission dictionary should yield the following probabilities after implementing the *compute_tag_prob* function: *P(Location|Prats) = 18%, P(Surname|Prats) = 72%, P(Others|Prats) = 9%.* "

Our results gave: *{'location': 0.18181818181818182, 'surname': 0.7272727272727273, 'other': 0.09090909090909091}*. Which we therefore can validate as correct.

TODO 2.1 - Code Implementation

```
from collections import defaultdict, Counter

def compute_tag_prob(train_loader: EsposallesTextDataset) -> Dict[str, Dict[str, float]]:
    word_tag_counts = defaultdict(Counter)
    for i in range(len(train_loader)):
        words, tags = train_loader[i]
        for word, tag in zip(words, tags):
            word_tag_counts[word][tag] += 1
    tag_probs = {}
    for word, tag_counter in word_tag_counts.items():
        total = sum(tag_counter.values())
        tag_probs[word] = {
            tag: count / total for tag, count in tag_counter.items()
        }
    return tag_probs

tag_probs = compute_tag_prob(train_loader)
print(tag_probs)

tag_probs["Prats"]
```

TODO 2.2: Write the predict_test_set, find_common_error and compute_token_precision functions.

In this section, we implemented a MLE algorithm to predict tags for each word in the test set using word-to-tag probabilities learned from the training data. For each word, the model selects the tag with the highest probability, defaulting to the "other" tag for OOV words. The model then generates sentence-level tag predictions for the entire test set.

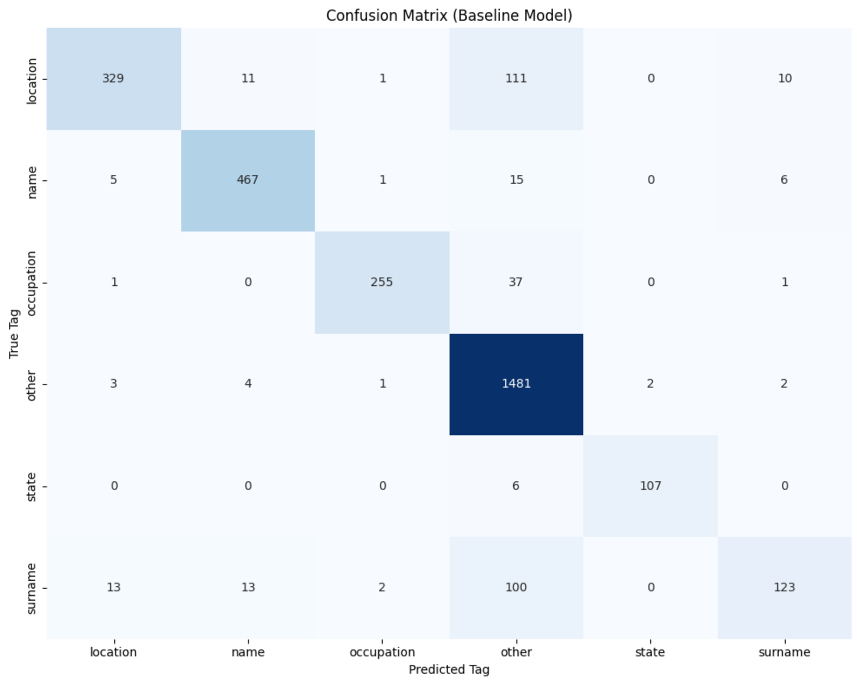
To evaluate its performance, we computed token-level precision by comparing predicted tags against the ground truth. Additionally, we built a confusion matrix to analyse common tagging errors and better understand which tags are most frequently misclassified. The token-level precision of our model is 88,9%, meaning that 88,9% of the predicted tags match the ground truth. This corresponds with the clue in the notebook.

TODO 2.2 - Code Implementation

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

    for i in range(len(test_set)):
        x, _ = test_set[i]
        sentence_prediction = []

        for word in x:
            if word in tag_probs:
                predicted_tag = max(tag_probs[word], key=tag_probs[word].get)
            else:
                predicted_tag = "other"
            sentence_prediction.append(predicted_tag)

        predictions.append(sentence_prediction)

    return predictions

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

    confusion_matrix = defaultdict(lambda: defaultdict(int))

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

        for true_tag, pred_tag in zip(true_tags, predicted_tags):
            confusion_matrix[true_tag][pred_tag] += 1
    return confusion_matrix

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

    total = 0
    correct = 0

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

        for true_tag, pred_tag in zip(true_tags, predicted_tags):
            total += 1
            if true_tag == pred_tag:
                correct += 1

    return correct / total if total > 0 else 0.0

test_predictions = predict_test_set(tag_probs, test_loader)
err1 = find_common_errors(test_loader, test_predictions)

precision = compute_token_precision(test_loader, test_predictions)
print(f"Token-level precision: {precision:.4f}")
```

TODO 2.3: Analyse the results according to the previously stated questions.

We first create our confusion matrix; the implementation is given in the code below. The results from this confusion matrix are shown in the figure below. The sum of the entries is 3017, the same as our sum of words in our test set.



What do all of the mistakes have in common?

All of the mistakes involve confusion between named entity tags such as name, surname, location, and occupation. Most errors occur when the model incorrectly predicts the tag *other* instead of the correct named entity tag.

What kinds of words are the least performers ?

The least accurately predicted categories are:

- surname → other (100 times), and 128 misclassification in total
- location → other (111 times), and 133 misclassification in total
- state → other (107 times), and 113 misclassifications in total

This means that surnames, locations, and state are particularly hard for the model to recognise

What's your solution for out-of-vocabulary words? Can you provide a prediction for those?

The current implementation is that all OOV words are predicted as "other" by default. This leads to a large number of false negatives, especially for location, surname, and name tags, as seen in the confusion matrix. Instead of always assigning "other" to OOV words, assign them the tag with the highest prior probability based on the tag distribution in the training data. So, in this way, the model makes a bit smarter guess for unknowns.

What words are usually the best performers?

Looking at the confusion matrix, the tag "other" has the highest number of correct predictions, with 1481 out of 1493 tokens correctly labeled. However, this is a bit misleading since "other" is the most common tag in the dataset, so the model naturally learns to default to it. Focusing on the named entity tags, "name" is the best performer. The model correctly identifies 467 out of around 494 instances, giving it an accuracy of about 94.5%. "Occupation" also performs quite well, with 255 correct predictions and only 39 being misclassified, most of them as "other". The reason these tags perform better is likely because the words associated with names and occupations appear more frequently and consistently in the training data.

TODO 2.3 - Code Implementation

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
from typing import List


def build_confusion_matrix(
    test_set: EsposallesTextDataset,
    predictions: List[List[str]]
) -> pd.DataFrame:
    confusion = defaultdict(lambda: defaultdict(int))

    for i in range(len(test_set)):
        _, true_tags = test_set[i]
        pred_tags = predictions[i]
        for true_tag, pred_tag in zip(true_tags, pred_tags):
            confusion[true_tag][pred_tag] += 1

    all_tags = sorted({tag for row in confusion.values() for tag in row} |
set(confusion.keys()))
    df = pd.DataFrame(index=all_tags, columns=all_tags).fillna(0).astype(int)

    for true_tag in confusion:
        for pred_tag in confusion[true_tag]:
            df.loc[true_tag, pred_tag] = confusion[true_tag][pred_tag]

    return df

confusion_df = build_confusion_matrix(test_loader, test_predictions)

plt.figure(figsize=(10, 8))
sns.heatmap(confusion_df, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title("Confusion Matrix (Baseline Model)")
plt.xlabel("Predicted Tag")
plt.ylabel("True Tag")
plt.tight_layout()
```

Graded exercise 3:

Where does the HMM perform better than the baseline approach?

The HMM-based model outperforms the baseline approach in contexts where word ambiguity and sequence structure are important. The baseline method assigns tags only based on the probability of a tag given a word (*P(tag|word)*), while the HMM considers both the current word and the surrounding tag context (sequential transitions), which allows it to model contextual dependencies. This is useful in our data that is based on historical marriage records, where tokens appear in predictable patterns.

For example, names like "Juan", "Maria", or "Boneu" can appear as either a first name or a surname depending on their position and surrounding context. The baseline is unable to distinguish these cases and always assigns the most frequent tag, leading to misclassifications. In contrast, the HMM captures the flow of the text (e.g., name → surname → occupation) and can therefore make more accurate predictions based on surrounding words and tag history.

Another thing is that HMM can work better with OOV words. While the baseline defaults to tagging all unknown words as "other", the HMM uses sequence information and learned patterns to infer tags, for example, classifying unknown words as "surname" if they appear after a known "name".

These improvements result in a token-level precision of 95%, an increase over the 88% baseline approach.

What words does it still struggle with?

Despite its high overall precision, the HMM model stays struggling with ambiguous or OOV words, especially when they appear in rare or penalized tag transitions. For example, transitions like location → name or occupation → surname have low or negative weights, meaning the model has learned to discourage these sequences. So, when such patterns do occur in real sentences, despite being valid, the model is less likely to predict them correctly, leading to misclassifications. This reflects the model's bias toward the most common tag sequences seen in the training data.

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

The baseline model handles all OOV words by assigning them the default tag "other", regardless of their position or surrounding context. This results in a high number of misclassifications, especially for OOVs that are actually names, surnames, or locations which is common in historical documents with unique or rare terms.

In contrast, the HMM model uses contextual features and tag transitions to infer more accurate labels even for unseen words. For example, if an unknown word appears after a known "name" or in a typical "name → surname" pattern, the model can correctly guess it as a "surname" which improves performance on OOV tokens.

We evaluated how both models handle OOV words by computing token-level precision specifically for unseen words in the test set, finding that the baseline model which always tags OOVs as "other"achieves only 4.40% precision (so only 4,4% of OOV words that are labelled as "other" are really "other"), while the HMM model, which uses contextual and transition-based features, significantly improves performance with an OOV precision of 77%.

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

Yes, the transitions learned by the HMM model show patterns consistent with the structure of historical marriage records. For instance, the most likely transitions such as name → surname, surname → occupation, and location → location, align with common sentence structures like "Joan Boneu, hortolà de Bara." These transitions capture how individuals are typically introduced with a name, followed by a surname, then a profession, and possibly a place of origin. Conversely, unlikely transitions like location → name or occupation → surname have large negative weights, which also makes sense because these tag sequences are rare or unnatural in the dataset, and the model learns to avoid them.