Negation and uncertainty detection in medical texts

Sara Martín Núñez[1], Aina Navarro Ràfols[2], Lara Rodríguez Cuenca[3], and Amelia Gómez Grabowska[4]

[1]NIU: 1669812
[2]NIU: 1670797
[3]NIU: 1667906
[4]NIU: 1631745
[*1]Sara Martín Núñez: 1669812@uab.cat
[*2]Aina Navarro Ràfols: 1670797@uab.cat
[*3]Lara Rodríguez Cuenca: 1667906@uab.cat
[*4]Amelia Gómez Grabowska: 1631745@uab.cat

COURSE 2023-2024 (SECOND SEMESTER - Natural Language Processing LESSONS)

Keywords: rule-based approach, machine learning, deep learning, medical records, negations, uncertainties.

1. INTRODUCTION:

The goal of natural language processing (NLP) is to analyze and understand text data automatically. The aim of this project is to work with a sub-problem within NLP which consists of identifying negation cues and their scopes, that is also called negation detection. We apply negation detection to the analysis of medical history reports, written using natural language text, which describes findings and observations of medical health issues by professionals. We have been provided with a JSON file with medical reports that do not include the private information of the patient (it has been masked).

2. OBJECTIVES:

- Identify the negation and uncertainty tag in each sentence of the corpus.
- Recognize the scope: everything that is affected by the negation/uncertainty word.

3. RULE-BASED APPROACH:

This approach is one of the oldest NLP methods and is often used with other methods like machine learning and neural networks (deep learning processes) for more complex tasks and to obtain a more robust and optimized result. The defining characteristic of a rule-based machine learner is the identification and utilization of a set of predefined linguistic rules that collectively represent the analyzed and processed textual data of our interest. These rules can be based on: grammar, syntax patterns, semantic rules, regular expressions... Then they are iteratively redefined by repetitive processing to improve their performance and accuracy.

3.1 Methodology:

Before starting with the code, we read the literature about rule-based methods that were provided to us. First, we read about the original implementation of the NegEx algorithm[1] and then about an implementation of RegEx adapted to Spanish[2]. After reading the documentation, we realized we could use some datasets of negations, uncertain words and medical keywords. We explored the original NegEx's author Github repository[4] and found out that it used the following tags:

- [PREN] for negations that appear before medical terms.
- [POSTN] for negations that appear after medical terms.
- [PREP] for uncertainty words that appear before medical terms.
- [POSTP] for uncertainty words that appear after medical terms.
- [PSEUDO] for words that make double negations.
- [CONJ] for conjunctions.

However, the dataset was in English, but luckily we found that PlanTL-GOB-ES's repository NegEx-MES in GitHub[5] had the translated [PREN] and [POSTN] in Spanish. In addition, we created an uncertainty words dataset in Spanish and Catalan using ChatGPT[6], since we did not found any database source. We also searched for datasets that contained medical keywords. The most famous one is UMLS (Unified Medical Language System), but it is not available without a license and it is not translated to Spanish. Therefore, we found the Centro de Terminologia TERMCAT datasets[7] that contained more than 30.000 medical terms. Once we had a solid base of data we could use, we drew a flow-chart of the data preprocessing procedure.

3.2 Implementation:

3.2.1 Data preprocessing

Data extraction

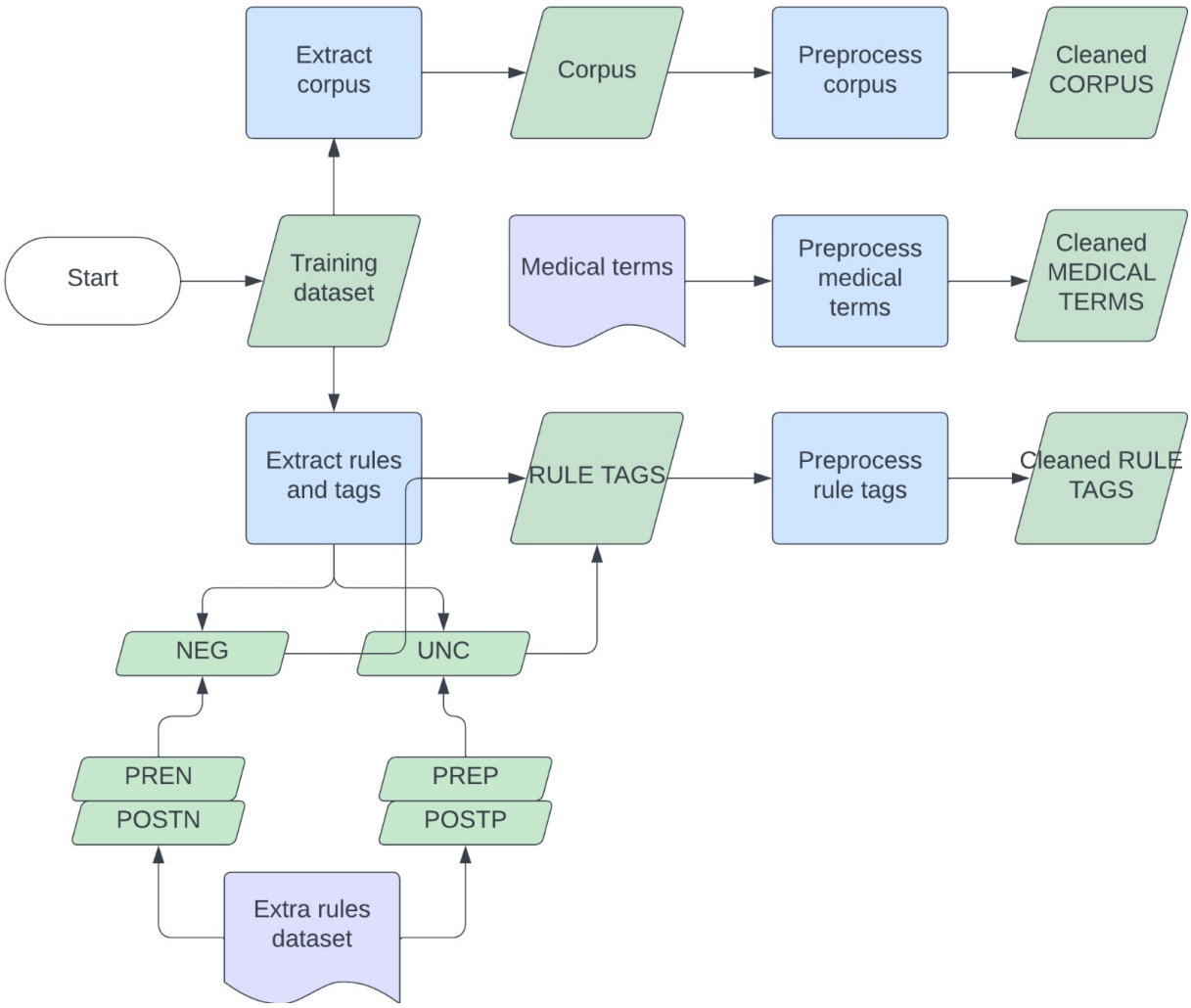


Figure 1: Data preprocessing

First, some preprocessing operations were performed. The data from the files was extracted and transformed into a format suitable for further processing. The texts were extracted from the JSON files (both training and test data) and were stored in two variables: x_train and x_test, each containing a list of strings representing different texts. The ground truth results for both the training and test data were also extracted to facilitate the evaluation of model metrics such as accuracy and precision.

To accomplish the ground truth extraction, a dictionary was created with keys labeled "NEG", "UNC", "NSCO", and "USCO", where the index tuples classified as negation words, uncertainty words, negation scopes, or uncertainty scopes were stored. Next, the text was tokenized, and the index tuples of each token in the text were determined. For each token, it was checked whether its index tuple was present in any of the lists corresponding to the keys in the dictionary. If so, the token was assigned the corresponding label; otherwise, it was labeled as "O". Consequently, the following data variables were obtained:

- x_train / x_test: lists of texts (strings)

- X_train_tokens / X_test_tokens: lists of words for each text

- X_train_tokenindex / X_test_tokenindex: lists of index tuples for each word of each text

- y_train / y_test: lists of true labels for each word of each text

Additionally, for the training data only, all words labeled as "NEG" were stored in a neg list, all words labeled as "UNC" were stored in an unc list, and all words labeled as "NSCO" or "USCO" were stored in a medical terms list.

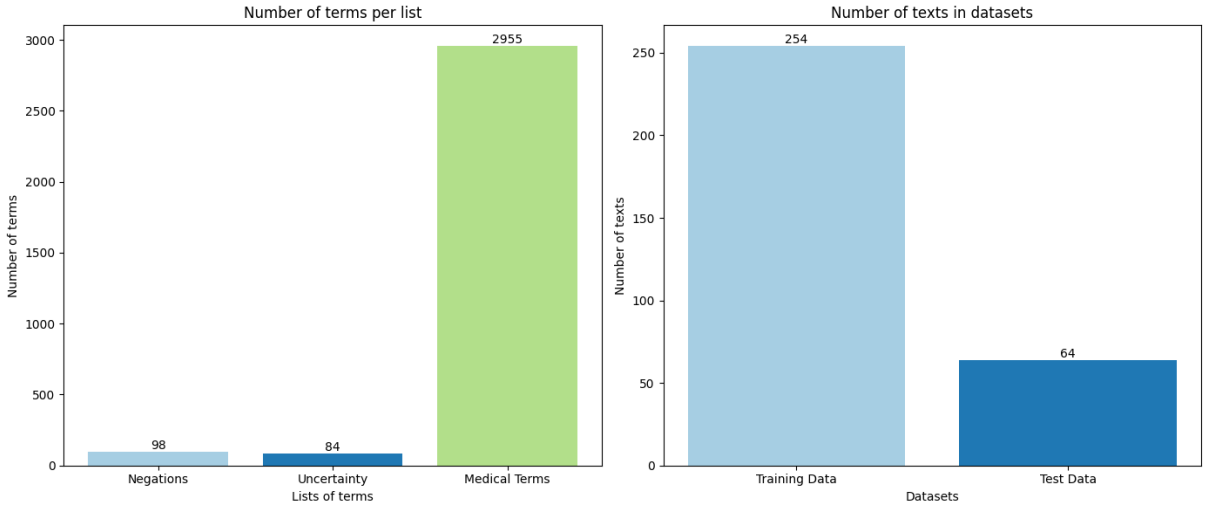


Figure 2: Initial number of terms per list and number of texts per dataset

Data augmentation

The negation/uncertainty detector has been enhanced by incorporating additional tag rules sourced from external datasets. This effort has resulted in more than doubling the number of negation tags, while the number of uncertainty cases remains unchanged. This augmentation was deemed necessary due to the relatively limited availability of negation tags and implementations in Spanish, especially when compared to the vast resources available in English. Additionally, addressing Catalan poses even greater challenges in obtaining a comprehensive and manageable implementation.

In addition to the datasets for negations/uncertainty, consideration was given to medical terminology in Catalan and Spanish. A collection of XML files containing medical words classified by medical specialty was utilized. To supplement the available resources, additional medical keywords were incorporated, ensuring that these lists consist solely of unique words.

All the datasets used in this process are accessible from the Github repository.

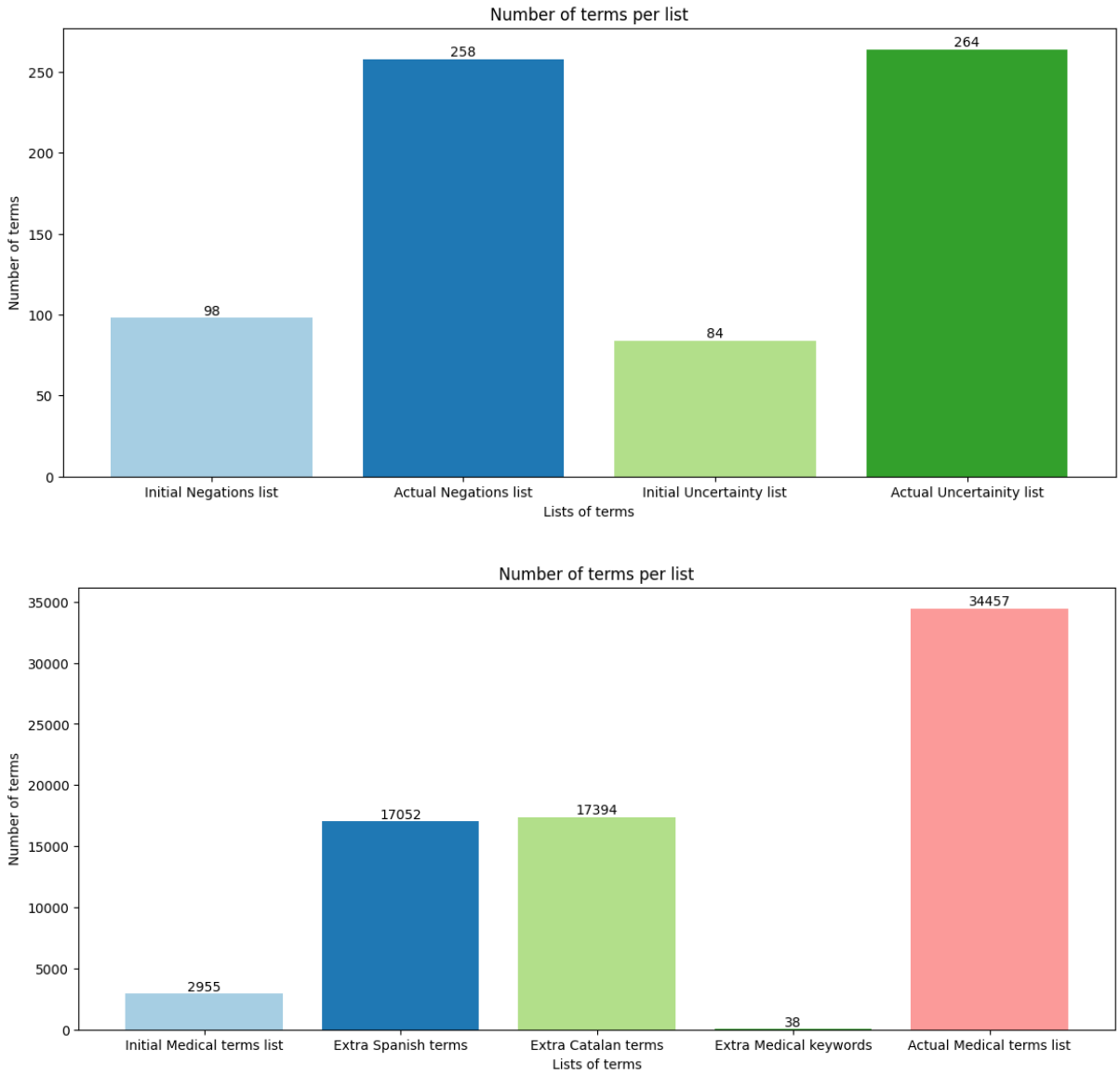


Figure 3: Number of terms per list

Data cleaning

Now that we have a robust database, it's essential to consider how to preprocess this data effectively, both the raw medical texts and the individual words from the datasets. Various preprocessing steps, such as removing punctuation, lemmatization, and correcting spelling mistakes, have presented challenges. We have encountered doubts regarding the potential benefits or drawbacks of altering the original dataset.

Initially, we attempted to correct spelling mistakes, but found it challenging due to the mixed nature of the dataset containing both Spanish and Catalan words, as well as numerous abbreviations and unconventional terms that the spelling corrector struggled to handle appropriately. For instance, the Catalan word "nega" was incorrectly corrected as "nena."

As a result, we settled on the preprocessing steps illustrated on Figure 3.

In summary, we now have cleaned versions of our rule terms (neg, unc, and conj) and our medical words (medicalterms), along with their corresponding regex versions

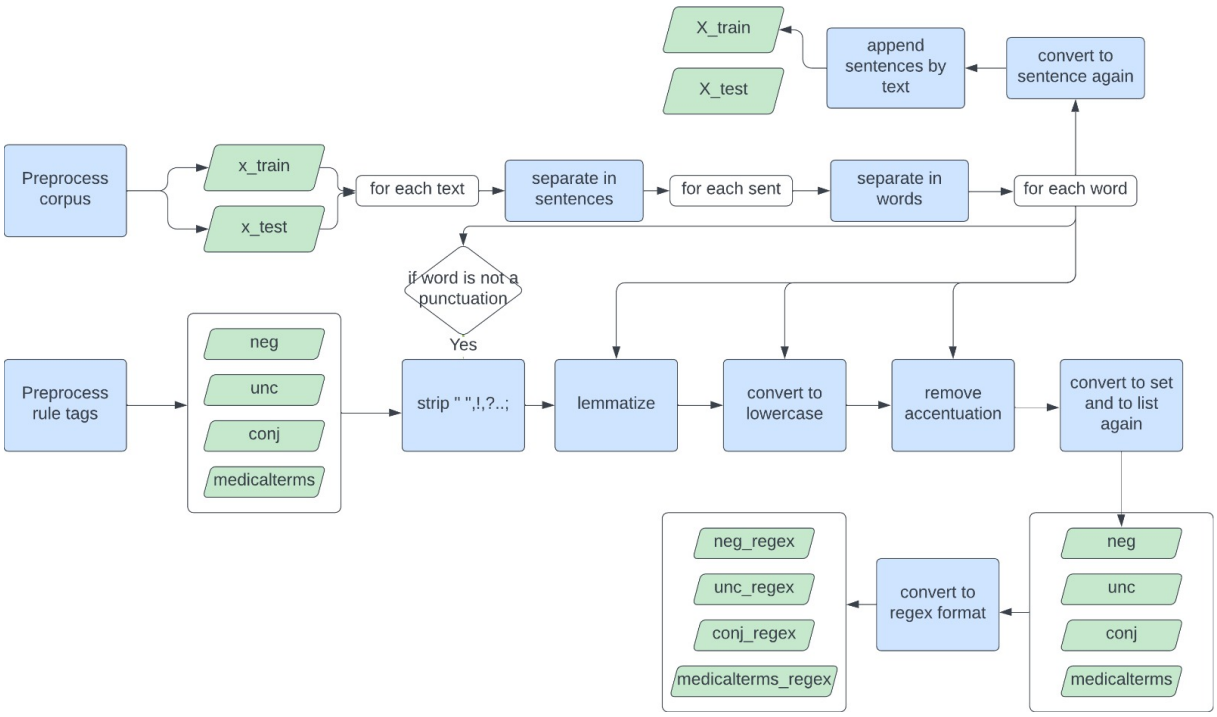


Figure 4: Preprocessing steps

(neg_regex, unc_regex, conj_regex, and medicalterms_regex). Transforming the terms to regex terms (which involves adding word boundaries to each term e.g "necesita" to "/bnecesitab/") was necessary to be able to find patterns in sentences with a regular expression.

Additionally, we have cleaned the words in our corpus without modifying the number of tokens in each sentence.

3.2.2 Algorithm implementation

The algorithm implementation of the negation rule-based system in English draws heavily from NegEx[1], which stands out as the extended and widely adopted tool in this domain. The algorithm takes as input a sentence with indexed diseases/findings and outputs the sentence with indexed negated diseases/findings. The process is as follows: all punctuation is removed but stop words are kept. Then the diseases/findings are indexed using the UMLS dataset, which as it was mentioned before, it is a medical terms datasets where each term has its own unique ID. For example: *"The patient denied experiencing chest pain on exertion"* becomes *"The patient denied experiencing S1459038 on exertion"*. To detect the negations and negations scopes, the method consists of two RegEx rules:

`<negation term>WORDS{0-5}<UMLS term>`

and vice versa (interchanging the negation term and the UMLS term positions). For instance: *"extremities showed no cyanosis, clubbing, or edema"* matches *"no"* as the negation term and both *"cyanosis"* and *"edema"* as the UMLS terms.

After understanding the NegEx algorithm, we tried to replicate our own version of it. We ended up doing three different approaches, each one trying to improve the last

one. Our hypothesis was that we would improve the results by using dependency parsing instead of just using regular expressions because it will take into account the syntax of the sentence.

Model 1 - using a simple NegEx algorithm

As mentioned previously, before starting with our own implementation it was necessary to examine and understand previously made implementations that could serve as a guideline or as a first step for designing our own. For our initial attempt, the MedicalReportTagger made by Chapman, B. E[4] was adapted to our problem. It used the same reasoning as the original NegEx algorithm: using regular expressions to detect negations, uncertainties and its scopes. The MedicalReportTagger takes as input the text and outputs the indices of the detected negations/uncertainties and their scopes. The class has its defined negations, uncertainties, conjunctions and medical terms lists, which we have used the ones we created. The process involves initially detecting the negations and then determining their scope by considering the entire sentence where the negation was detected. The same approach is applied to uncertainties. However, this approach has several drawbacks as it does not take into account the syntax of the sentence to determine the affected scope; instead, it considers the entire sentence. Consequently, while this method yields high recall because all scopes are classified, many words that should not be included in the scope are also classified as if they were.

Model 2 - using only regular expressions

For this version, a custom tagger function was created. It takes as input our lists of negations, uncertainty words and medical terms and each text of the cleaned corpus, which is already segmented into sentences, and outputs the list of labels for each word in the text. For the detection process two regular expressions were defined: one for negations and another for uncertainties. These patterns were crafted to capture instances where negations or uncertainties occur within a certain proximity to medical terms in the text. The expressions can match patterns where negations or uncertainties are followed by up to five words before encountering a medical term, or vice versa:

```
<neg/unc>WORDS{0,5}<medicalterm>
<medicalterm>WORDS{0,5}<neg/unc>
```

In the processing phase, each sentence in the text is scanned for matches using these regular expressions, first for negations and then for uncertainties. For each match, the algorithm identifies the starting indices of the negation/uncertainty and the medical term within the sentence. By examining the words between these indices, the algorithm deduces the scope of the negation/uncertainty.

The resulting output is a nested list, with each sublist corresponding to a different text and containing the labels for each word. For instance, in the example sentence *"un*

ataque al corazón puede ser" the labels would be assigned accordingly: ["O", "USCO", "USCO", "USCO", "UNC", "O"].

To handle cases where labels overlap, such as when a word is both negated and uncertain, the algorithm designates the label as "NEG/USCO". This way we can visualize those exceptions easily and realize if some change is needed in our database or in our implementation.

However, the model has inherent limitations. It relies heavily on the provided datasets; if a negation negates a medical term not stored in the dataset or vice versa, it won't be detected. Additionally, misspelled terms are not identified due to the lack of spelling correction, which is something we also want to take into consideration for future improvements. Furthermore, the regular expressions used are highly specific, potentially leading to incomplete scope detection in complex sentences. For example, if we have the sentence *"el paciente no tiene alergia a penicilina y cloramfenicol"*, it will detect *"no"* as the negation but only *"tiene alergia"* as the negation scope, since our regular expression ends with the first medical term found. We wanted to improve the regular expression so that it could include more than one medical term, however, because of time problems we weren't able to succeed.

In response to these challenges, an alternative approach was considered: implementing a syntactic parser to analyze sentence dependencies and define scopes based on the dependencies of the negation. This prompted the exploration of model 3 as a potential solution.

Model 3 - using dependency parsing

For this algorithm the tagger function was modified so that it tagged scopes based on the sentence's dependency tree. For each sentence in the text, each match was found with the regular expressions (first with negations and then with uncertainties), as it was done in the original function. Once it has found a negation/uncertainty keyword, the dependency tree of the sentence is computed based on this keyword.

To do so, the function dependency_scope has been defined, which takes as input the sentence as well as the keyword, and outputs a list including the words inside the scope of this keyword. It starts by creating the dependency tree of the sentence using spaCy, and finds the head of the keyword (e.g., in *"no tiene fiebre"*, the keyword is *"no"* and its head is *"tiene"*). Once it has the head, it will add to the scope list all words which depend on the head, as well as the words which depend on words already inside the scope (direct and indirect dependence). We try to handle some exceptions: (1) if there is a conjunction dependence with the head we will not add the word unless it is a medical term; (2) we use the word index to distinguish the head from any other occurrence of the same word.

After this, for every word in the sentence it is checked whether it is the negation/uncertainty keyword (or if it is contained in it, because some terms are made up of several words). If it is, its position is labeled as the corresponding tag. If it isn't, it is labeled

as tag scope if it is included in the dependency scope and as 'other' otherwise. This way, the label lists are computed for both the negation and the uncertainty, and join both lists to obtain the final result.

One of the limitations of this model is that it takes only one negation/uncertainty keyword into account each time it computes the dependency tree. An improvement could be to take into consideration all keywords in the sentence when computing the scope for each of them. Another limitation is that some medical terms or Catalan words are not recognized by the model and get wrongly tagged, creating incorrect dependencies. Problems also arise from lack of punctuation in the text, when sentences that should be divided are present together without no pause or clear mechanism to divide them. This creates dependencies between words of different constituents, degrading the performance of the model.

3.2.3 Model evaluation

After computing and executing our model, y_pred_train and y_pred_test are obtained, which are the lists of the predicted labels for each word. Using these predictions and the y_train and y_test we can validate our models and extract some conclusions about different metrics, such as precision, recall, F1-score and accuracy.

3.3 Results:

As for the results of our model we have got the precision, recall, F1, accuracy of each model.It is important to know that we have rows 0,1,2,3,4 in the evaluate table which respectively correspond to: Other ("O"), Negation ("NEG"), Negation Scope ("NSCO"), Uncertainty ("UNC") and Uncertainty Scope ("USCO").

Table 1: Evaluation model 1 Train set

	precision	**recall**	**f1-score**	**support**
O	0.9910	0.9048	0.9459	224531
NEG	0.8004	0.9996	0.8890	4517
NSCO	0.5030	0.8390	0.6289	13981
UNC	0.4786	0.8523	0.6130	684
USCO	0.1611	0.8244	0.2696	2074
accuracy	0.9020			
macro avg	0.5869	0.8840	0.6693	245787
weighted avg	0.9513	0.9020	0.9202	245787

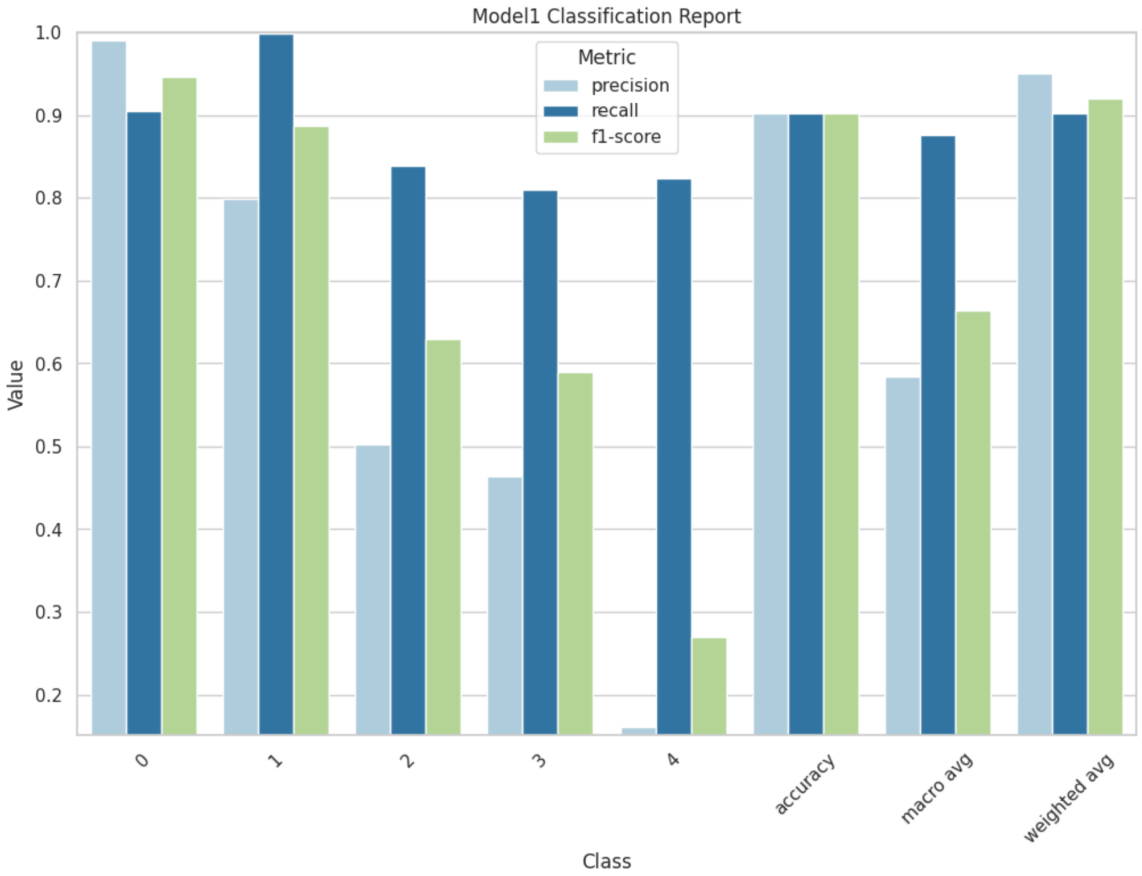


Figure 5: Evaluation metrics for the training set

Table 2: Evaluation model 1 Test set

	precision	**recall**	**f1-score**	**support**
O	0.9900	0.9099	0.9483	58388
NEG	0.8	0.9830	0.8821	1180
NSCO	0.5295	0.8266	0.6455	3570
UNC	0.4967	0.77	0.6039	200
USCO	0.1581	0.8095	0.2645	567
accuracy	0.9053			
macro avg	0.5948	0.8598	0.6688	63905
weighted avg	0.9518	0.9053	0.9230	63905

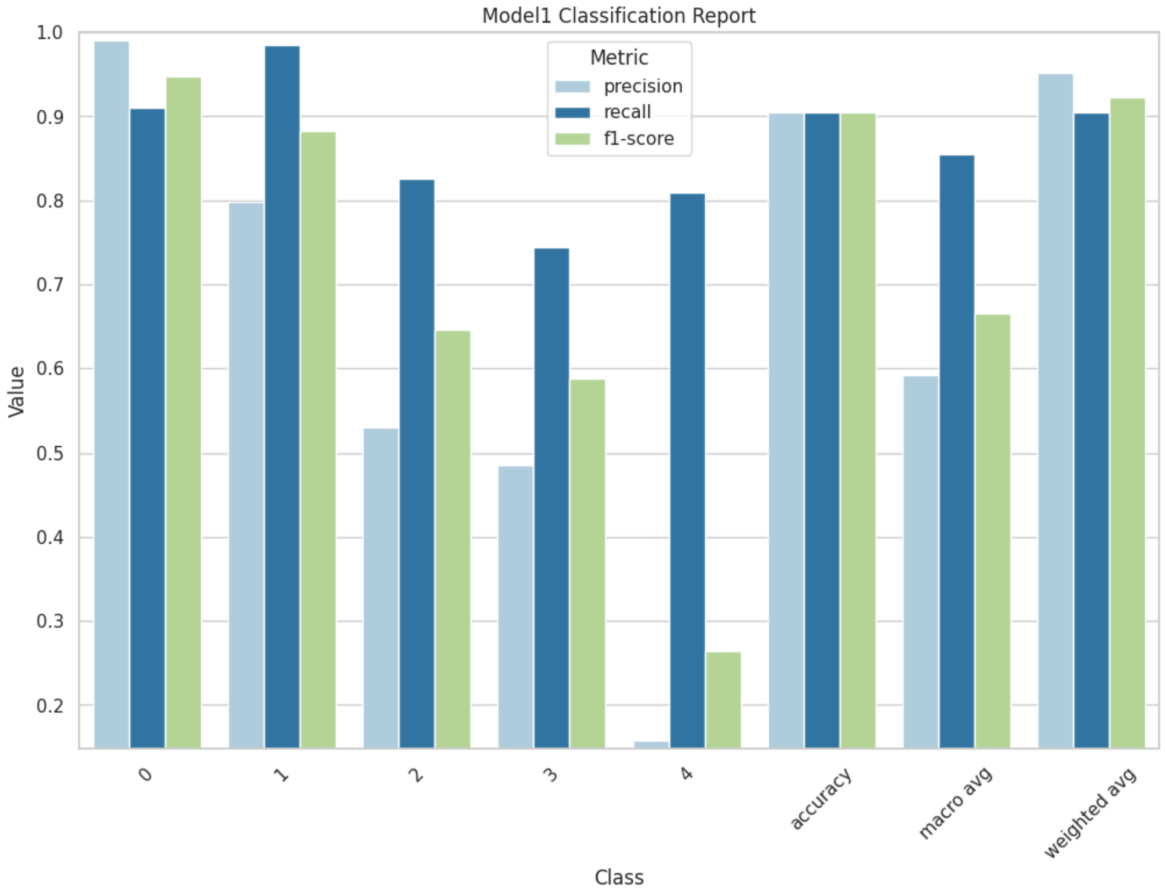


Figure 6: Evaluation metrics for the test set

Table 3: Evaluation model 2 Train set

	precision	**recall**	**f1-score**	**support**
O	0.9358	0.9702	0.9527	224531
NEG	0.8858	0.4724	0.6162	4517
NSCO	0.4009	0.2185	0.2828	13981
UNC	0.5230	0.5146	0.5187	684
USC	0.1857	0.2049	0.1948	2074
accuracy	0.9106			
macro avg	0.5862	0.4761	0.5131	245787
weighted avg	0.8970	0.9106	0.9008	245787

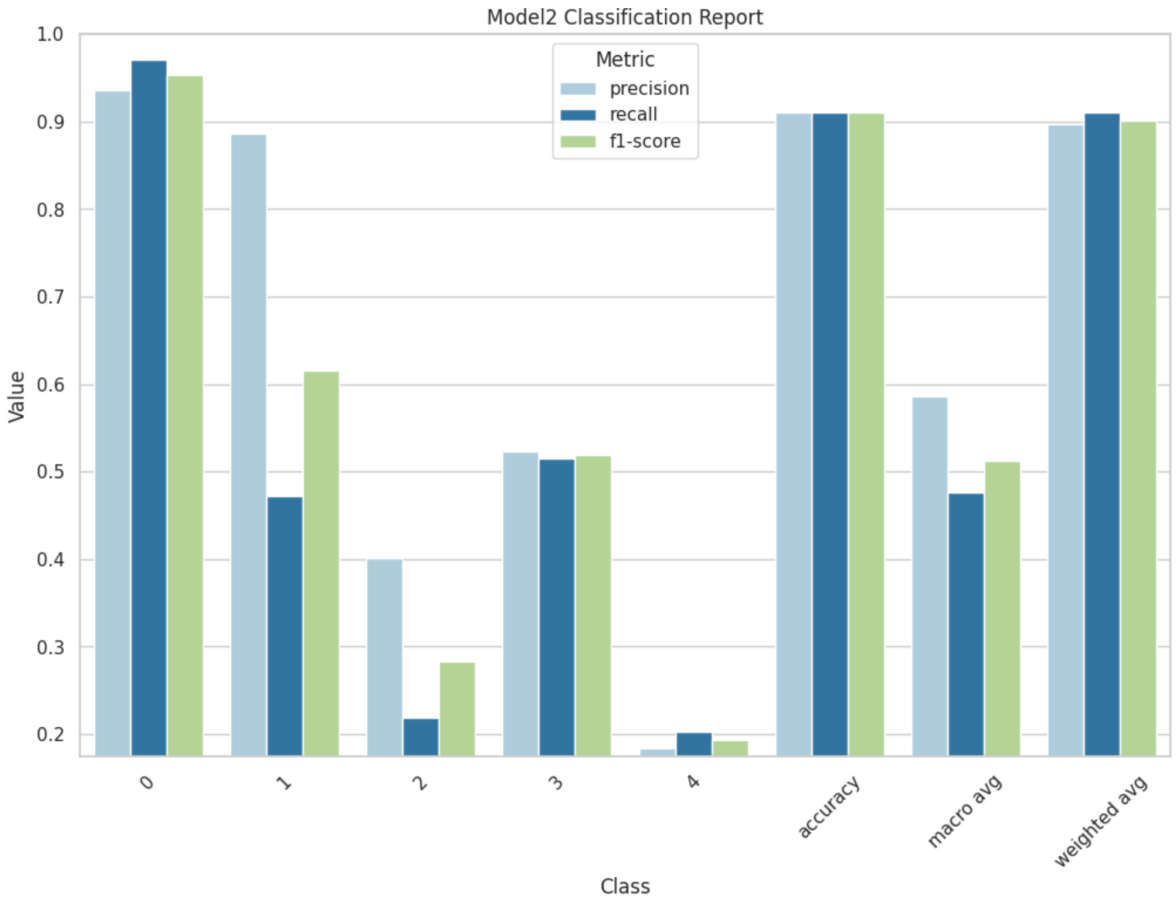


Figure 7: Evaluation metrics for the training set

Table 4: Evaluation model 2 Test set

	precision	**recall**	**f1-score**	**support**
O	0.9351	0.9738	0.9540	58388
NEG	0.9118	0.4474	0.6003	1180
NSCO	0.4188	0.2117	0.2813	3570
UNC	0.5272	0.435	0.4767	200
USCO	0.2094	0.2028	0.2060	567
accuracy	0.9130			
macro avg	0.6005	0.4541	0.5037	63905
weighted avg	0.8981	0.9130	0.9018	63905

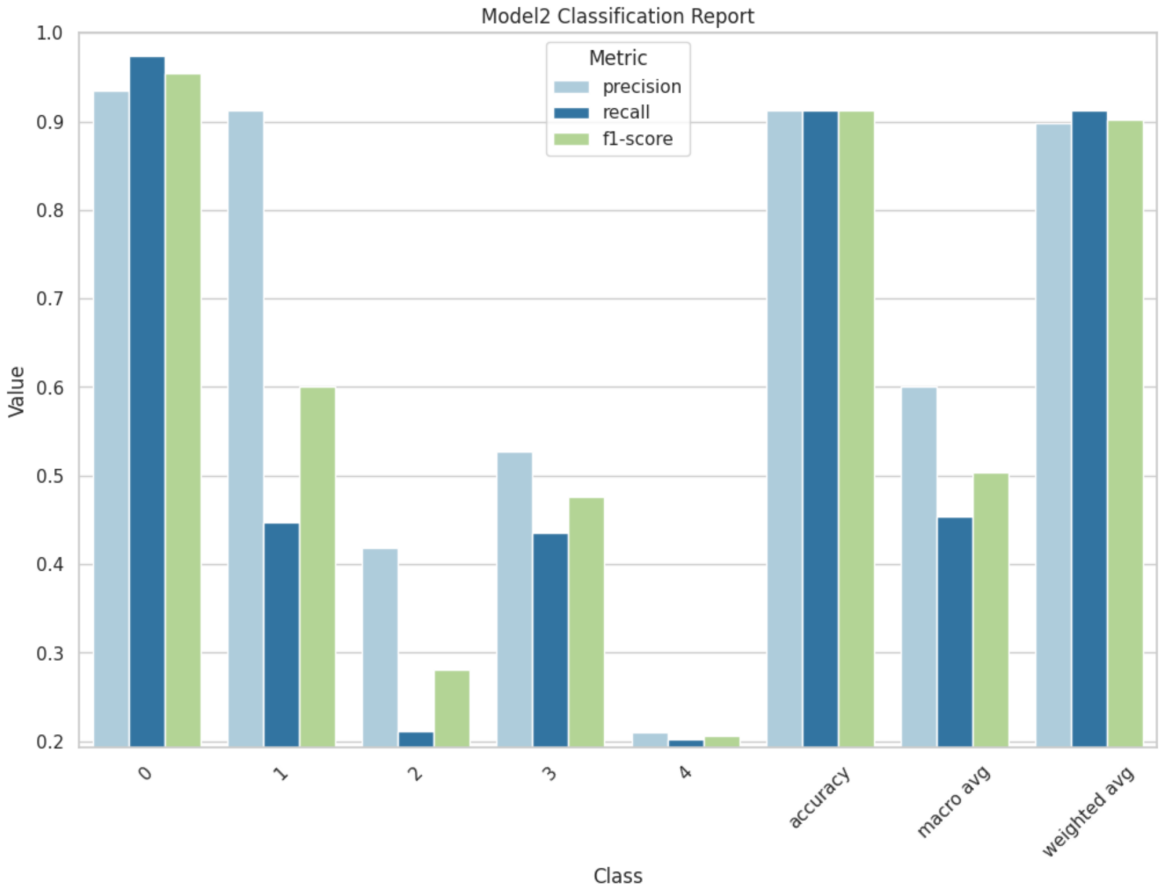


Figure 8: Evaluation metrics for the test set

Table 5: Evaluation model 3 Train set

	precision	**recall**	**f1-score**	**support**
O	0.9423	0.9051	0.9234	224531
NEG	0.8269	0.4770	0.6050	4517
NSCO	0.2321	03403	0.2760	13981
UNC	0.5405	0.6038	0.5704	684
USCO	0.1450	0.4378	0.2178	2074
accuracy	0.8603			
macro avg	0.5374	0.5528	0.5185	245787
weighted avg	0.8920	0.8603	0.8737	245787

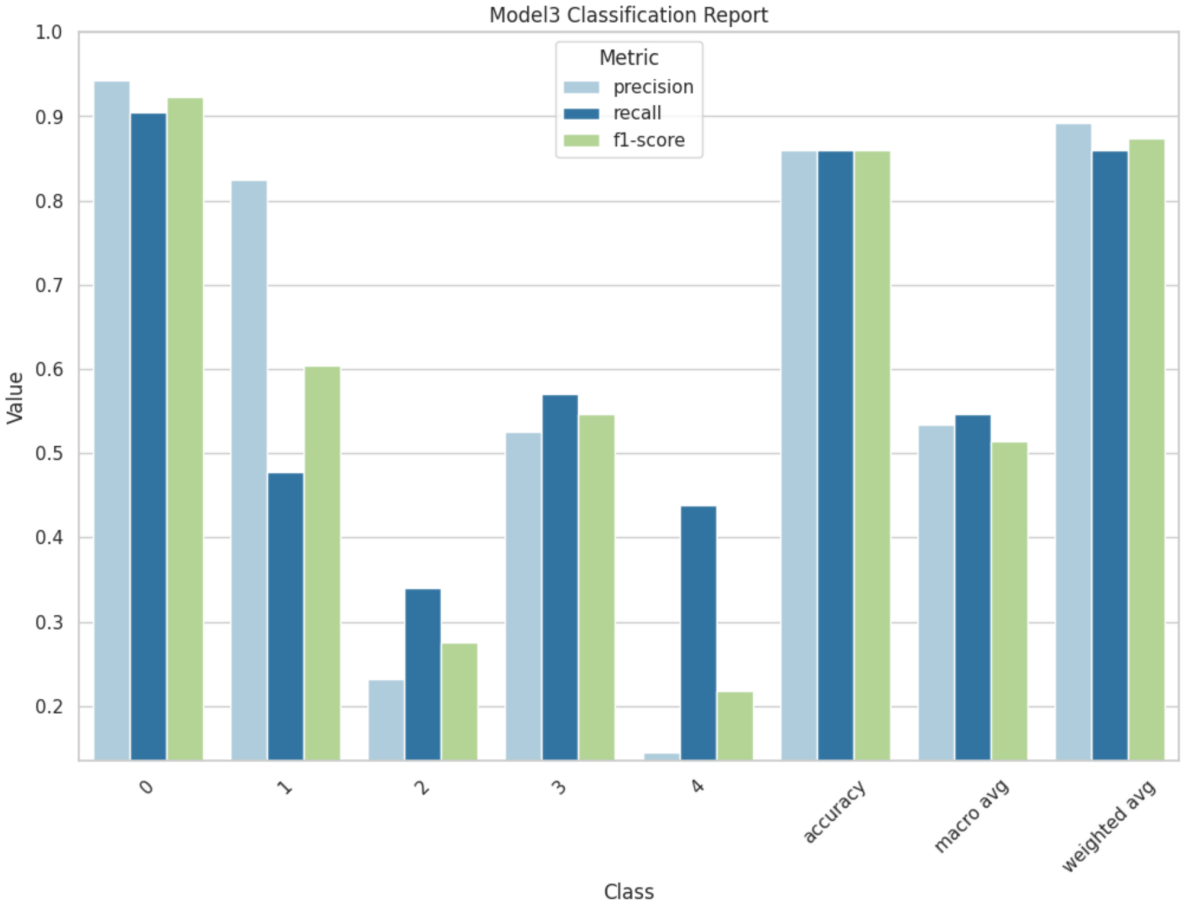


Figure 9: Evaluation metrics for the training set

Table 6: Evaluation model 3 Test set

	precision	**recall**	**f1-score**	**support**
O	0.9436	0.9224	0.9329	58388
NEG	0.8349	0.45	0.5848	1180
NSCO	0.2859	0.3509	0.3151	3570
UNC	0.5425	0.51	0.5257	200
USCO	0.1556	0.4444	0.2305	567
accuracy	0.8762			
macro avg	0.5525	0.5355	0.5178	63905
weighted avg	0.8966	0.8762	0.8844	63905

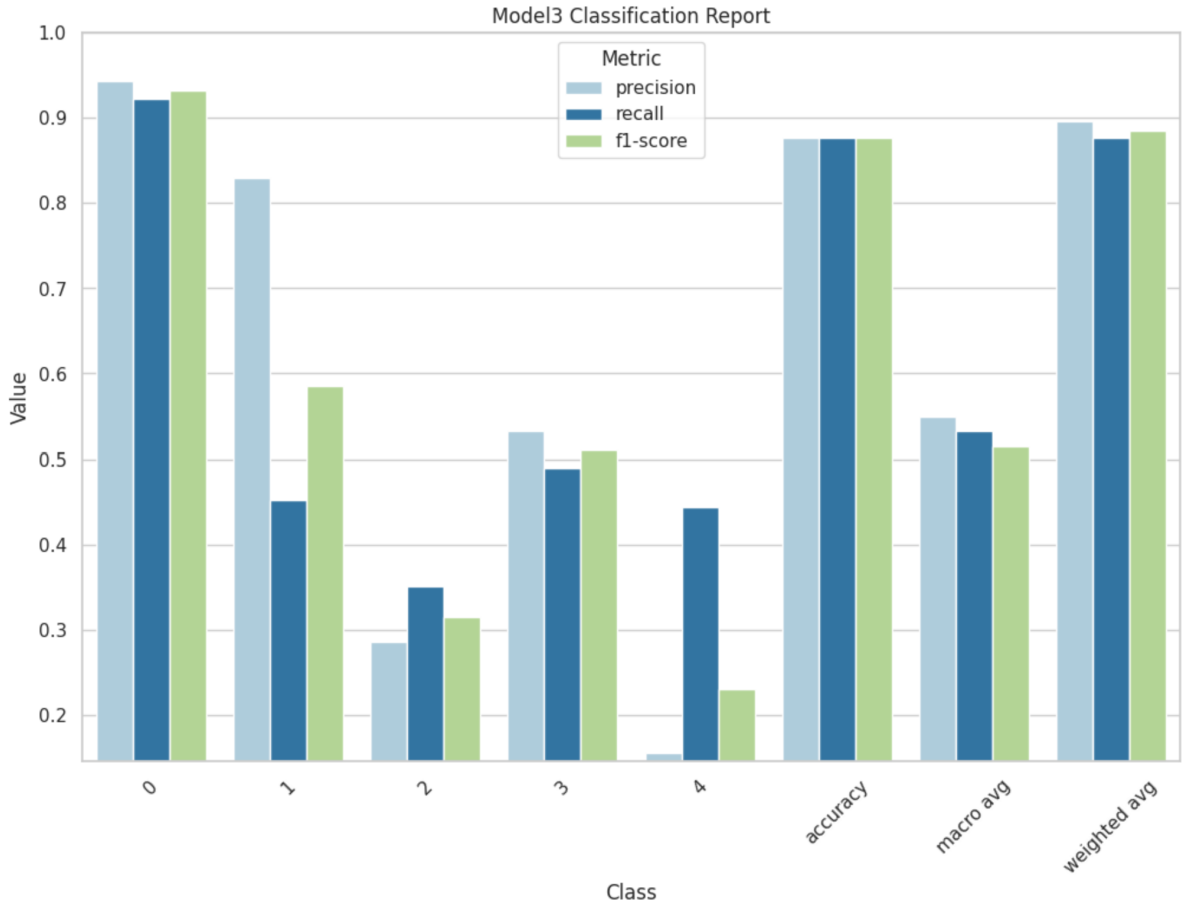
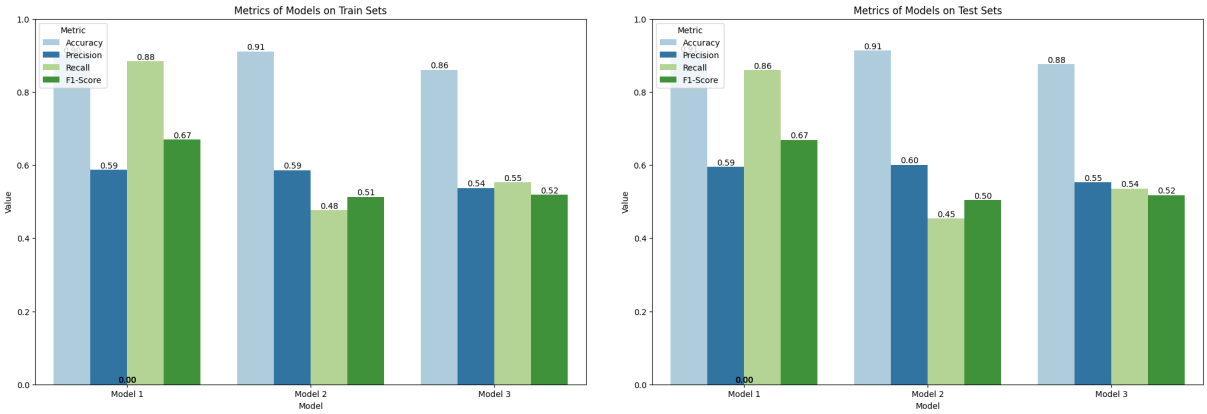


Figure 10: Evaluation metrics for the test set

In Figure 11, we make a comparison of the models, taking into account Accuracy, Precision, Recall and F1-score on the training and test sets.



(a) (b)

Figure 11: Comparison for: (a) training set, (b) test set

Finally, to illustrate the performance of the different models, we consider a piece of text extracted from the training data and compare the ground truth with the outputs for each model in Figures 12-15.

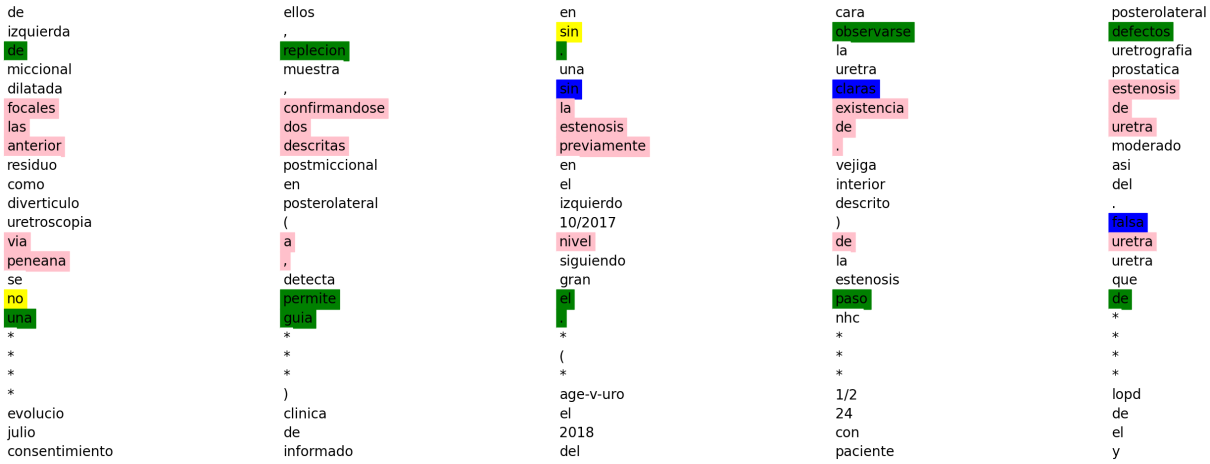


Figure 12: Ground truth sentence

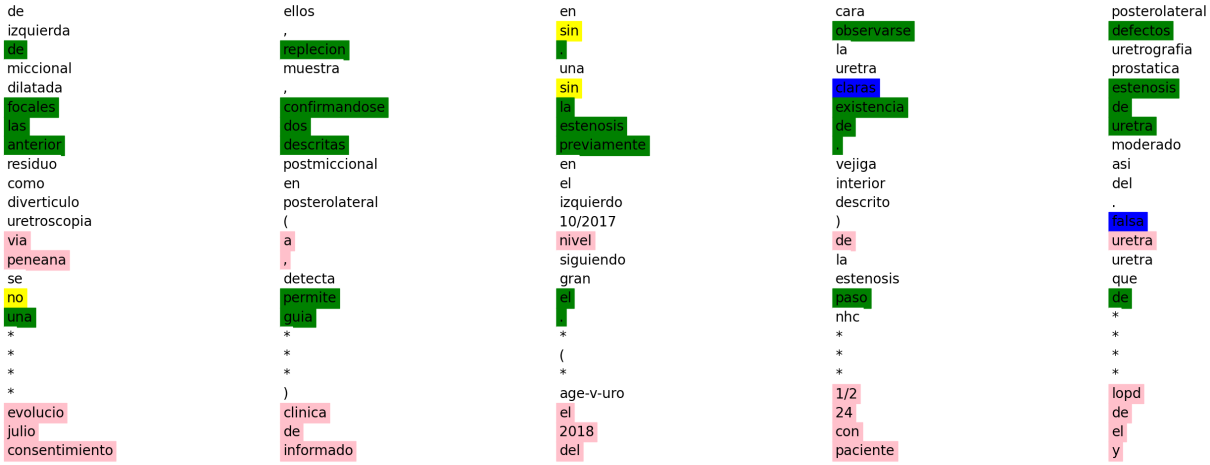


Figure 13: Model 1 output

Figure 14: Model 2 output

Figure 15: Model 3 output

From the results obtained we can conclude that our hypothesis that stated that using syntactic parsing on top of regular expressions would improve significantly our results is not completely true. We can observe a higher accuracy on our Model 2 (91% compared to 88% of Model 3 on test set results) and also higher precision (60% versus 55% also on test results). However, we do see an improvement in the recall and f1-score in the Model 3. In addition, as opposed as what we thought, model 1 obtained the best metrics of all three models.

3.4 Problems:

- Difficulties to find datasets in Spanish and Catalan for both cases: medical terms and negation terms.
- Lots of research to try to focus on the implementation the best way.
- Doubtful about how to preprocess our data. The mechanism and how adequate the different words are. Whether to correct the misspelling, work in sentence or words label,..
- Imagine what rules could we add so our implementation has better performance.
- The amount of time that the training set requires to be executed at the end while using CPU (around 40 min).

3.5 Conclusions on Rule-based Approach:

At the beginning, after all the research we have done, preprocessing the data we created and the corpus, we end up with the first idea of doing a model only with regular expressions.

First we did a model based on the NegEx algorithm of the Github repository[4]. For this model, as the algorithm is implemented in English, we have used our own rules to

do so. It works well too, but with an accuracy of 90.2% for training dataset and 90.6% for test dataset. With these results of this model we can conclude that our implemented rules work well, as we have a high accuracy. However, for lack of time and simplicity of this model, we were not able to improve these results.

After that we decided to create our own model from scratch. With this model we have obtained quite good results, with an accuracy of 91%. However, as we have said before, the huge limitation of only working with this dataset and that it only works with two different structures of sentences, make it a very basic but good first attempt of detecting negations and uncertainties in this medical notes dataset.

Finally, we had the idea of using dependency parsing for detecting more variety of sentences with negations, uncertainties and medicalterms. But it only takes into account the negation or uncertainty word as the most important, which can lead to errors in classification and in detecting the scopes. Also, as we have said before, there are some words (medical terms and words in catalan) that our model does not detect, which lead to a wrong result.

However, the accuracy we have obtained with this model is good too, but less than with the first model, 86%. We think that the accuracy is lower due to as it only takes into account the negation/uncertainty words and does not correctly detect medical terms and catalan words correctly, even if it has more flexibility with the kind of sentences the corpus may have, those limitations are important.

In summary, the three methods we have implemented give a reasonably high accuracy and good results. But we think that with more time and resources, we would be able to make a better algorithm.

In general we can say that this first implementation has given us the chance to get an idea of how hard it is to create a model that using basic Regex and rules is able to cover all the different aspects that a language can have. Our advantage is that for our project we can find some patterns for negation and uncertainty words, but as it gets harder (idioms, conditional sentences....) this method may not be enough.

Besides this, we have been able to use some of the concepts learnt in class and we have managed to incorporate them correctly in the process (tokenize, lemmatize,...). Nonetheless, it has been quite a challenge to understand how to manage the data and make our code a correct interpreter of the different words in the text.

4. MACHINE LEARNING APPROACH:

Machine Learning approaches for negation and uncertainty detection give more robustness, effectiveness and interpretability than a rule-based method. It consists of preprocessing the data so that it can be fed to an algorithm i.e., decision trees, Hidden Markov Model, Support Vector Machines.... However, even if it improves the results of rule-based approaches, it still gives worse results than a Deep Learning approach.

4.1 Methodology:

First of all, we read many papers relating these two topics, negation and uncertainty detection and Machine Learning. After we had a clear idea of what and how to do them, we started the code. Before applying algorithms, we needed to clean our data. We extracted the data from the JSON file and applied some functions, we splitted it into a train and test set, and obtained the labels for each word by doing some tagging (BIO and BIESO). Once we had obtained the finals datasets for working with the different algorithms, we implemented CRF, HMM, SVM, Naïve Bayes and Logistic Regression.

Finally, after applying all the algorithms, we have done an evaluation for each of them and arrived to a final conclusion.

4.2 Implementation:

4.2.1 Data preprocessing

Data extraction

First, some preprocessing operations were performed. The data from the files was extracted and transformed into a format suitable for further processing. The texts were extracted from the JSON files (both training and test data) and were stored in two variables: x_train and x_test, each containing a list of strings representing different texts. The ground truth results for both the training and test data were also extracted to facilitate the evaluation of model metrics such as accuracy and precision.

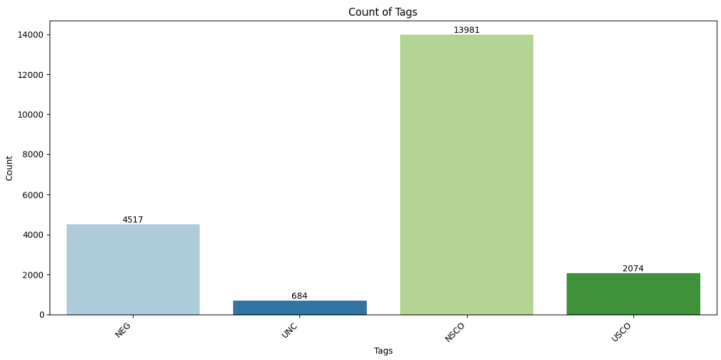


Figure 16: Tagger words plot

Tagging

It is a classification method that may be defined as the automatic assignment of description to the tokens. In this case those descriptors are called tags, which may represent some semantic information.

BIO Tagging

The BIO tagging is a a common tagging format for tagging tokens in a chunking task in computational linguistics (ex. named-entity recognition), based on identifying and adding prefixes to the word tags, and it consists of these tags:

- B → Beginning
- I → Inside
- O → Other

These tags are general, so we have adapted them for our task:

- B_NEG → beginning negation
- I_NEG → inside negation
- B_UNC → beginning uncertainty
- I_UNC → inside uncertainty
- B_NSCO → beginning negation scope
- I_NSCO → inside negation scope
- B_USCO → beginning uncertainty scope
- I_USCO → inside uncertainty scope
- O → other

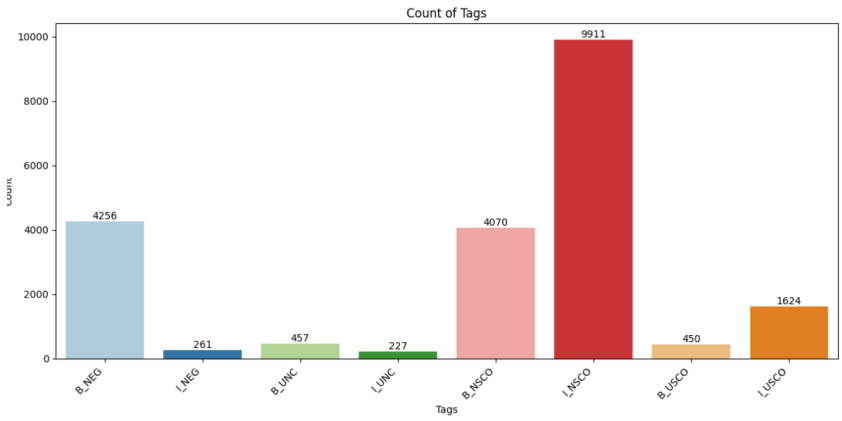


Figure 17: BIO Tagger words plot

BIESO Tagging

The BIESO tagging is another tagging scheme used in NER (Name Entity Recognition), which expands the prefixes for the tags:

- B → Beginning
- I → Inside
- E → End
- S → Single
- O → Other

As this tagging method has more tags, the data resulting would have more information than the BIO tagging, which means that it is more useful. These tags are general, so we have adapted them for our task:

- B_NEG → beginning negation
- I_NEG → inside negation
- E_NEG → end negation
- S_NEG → single negation
- B_UNC → beginning uncertainty
- I_UNC → inside uncertainty
- E_UNC → end uncertainty
- S_UNC → single uncertainty
- B_NSCO → beginning negation scope
- I_NSCO → inside negation scope
- E_NSCO → end negation scope
- B_USCO → beginning uncertainty scope
- I_USCO → inside uncertainty scope
- E_USCO → end uncertainty scope
- O → other

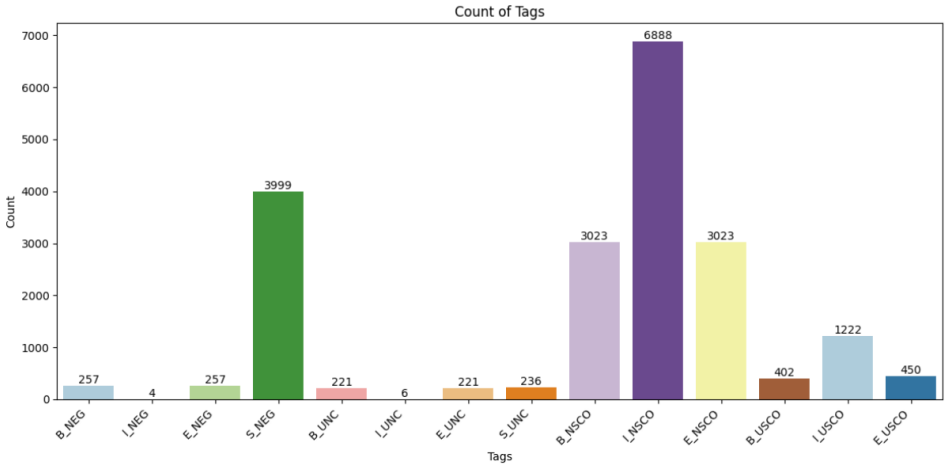


Figure 18: BIESO Tagger words plot

4.2.2 Algorithm implementation

Model 1 - CRF

CRFs use the observed data to predict the labels of the sequence, while taking into account the dependencies between neighboring labels. During training, the model learns to identify patterns and features in the input text that are associated with named entities, such as the presence of specific words or phrases, syntactic structures, or contextual information.

For the CRF implementation, three methods have been applied, one using the original tags, with BIO Tagging and with BIESO tagging. All of them used the same hyperparameters when creating the model (algorithm used='lbfgs', c1=0.01, c2=0.1, max_iterations=100, all_possible_transitions=True), which gave the best result from the options explored.

Model 2 - HMM

For this algorithm, some preprocessing tasks were made. Firstly, to extract the points we had in the training set, then extract each word and each token for each tuple so that we obtain the vocabulary of the dataset and the number of tags.

After these preprocessing tasks are done, the model is ready to be trained. Firstly, the emission probabilities (probability of a word given a tag) are computed. For this we have created a function that returns the counts of a word given a tag and the tag. Another function was created for calculating the transition probabilities (probability of getting a tag given the tag of the previous word) by doing a function that returns the counts of the second tag given the first one, along with the counts of the first tag.

After computing these two probabilities, we have done some visualizations to see which are the frequencies of each tag.

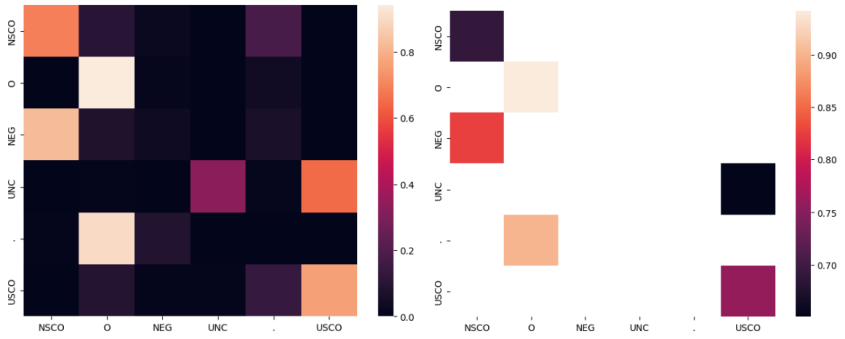


Figure 19: Conditional probability plots

Finally, the Viterbi algorithm was implemented, which consists of finding the most likely sequence of hidden states for a given sequence of observations. By doing this, we have a prediction of the tags of sequences that we enter in the function of Viterbi, the following steps were made: for all the unique tags in the sequence that enter the function, the transition and emission probability was computed, the maximum state probability and the corresponding tag was found, and finally it returned a list of all the maximum states probabilities and the corresponding tag for all the input words in the function.

Model 3 - SVM

Support Vector Machine (SVM) model is a powerful algorithm for classification tasks, particularly in scenarios with a large number of features. Let's see how it can be implemented for the negation and uncertainty words detection in the medical field.

The initial step involves preparing the data in the correct format for training the model. It flattens the different datasets into lists of features (X_train_flat, X_test_flat) and lists of labels (y_train_flat, y_test_flat). This flattening process aggregates the feature representations and labels across all sentences in the datasets.

However, now we have the raw text which can not be processed using machine learning, so we use the function DictVectorizer to adapt dictionaries of feature names to feature values into a sparse matrix representation. Besides, it uses some transforming functions to ensure consistency in feature representation across both training and testing datasets.

After that, the LinearSVC was called, which is a linear Support Vector Machine (SVM) classifier from the sklearn.svm module that is particularly effective when you have a large number of features and require a fast training time. Lastly the model does its predictions and evaluations.

Model 4 - Multinomial Naïve Bayes

Let's now consider this model as it is known due to its simplicity, efficiency, and effectiveness in handling text data. First the data is preprocessed. It has been done in two different ways as we have applied DictVectorizer that as before required a concrete structured data, and also Tf-idfVectorizer was implemented as we discovered that they were quite similar and we wanted to find which performed better.

In this implementation it is important to consider the use of the Pipeline. This element orchestrates a cohesive workflow, encompassing feature vectorization (using the previous vector) and model training in a unified setting (multinomial_NB). Through the invocation of the fit method, the classifier imbibes the intricacies of the training data, learning the underlying patterns and relationships crucial for effective classification. Then it will be used for prediction and consequently it will be evaluated.

Model 5 - Logistic Regression

Lastly an array methodology was considered. Logistic regression stands out as a robust and versatile technique, renowned for its interpretability and efficacy in handling text data. Even if this method is quite similar in implementation like the previous one (the only variation is that the model in the Pipeline is the Logistic Regression), in this case we can highlight the time consumption of the methods.

Multinomial_NB is actually faster than Logistic in all cases. Besides, the time increases as the tagging gets more complex (Baseline in both algorithms are faster than BIESO for instance). In addition, as the vector used is different even if they help to implement the algorithms, DictVectorizer is actually faster than Tf-idfVectorizer no matter the algorithm implemented.

4.3 Results:

As for the results of our model we have got the precision, recall, F1, accuracy of each model. It is important to know that we have rows 0,1,2,3,4 in the evaluate table which respectively correspond to: Other ("O"), Negation ("NEG"), Negation Scope ("NSCO"), Uncertainty ("UNC") and Uncertainty Scope ("USCO").

4.3.1 CRF model

CRF Baseline model

Table 7: Test set evaluation on CRF with Baseline Tagging

	precision	recall	f1-score	support
NEG	0.9819	0.9652	0.9735	1180
NSCO	0.9651	0.9294	0.9469	3570
O	0.9949	0.9972	0.9961	58388
UNC	0.9894	0.935	0.9614	200
USCO	0.9428	0.9894	0.9656	567
accuracy	0.9926			
macro avg	0.9748	0.9633	0.9687	63095
weighted avg	0.9925	0.99216	0.9925	63095

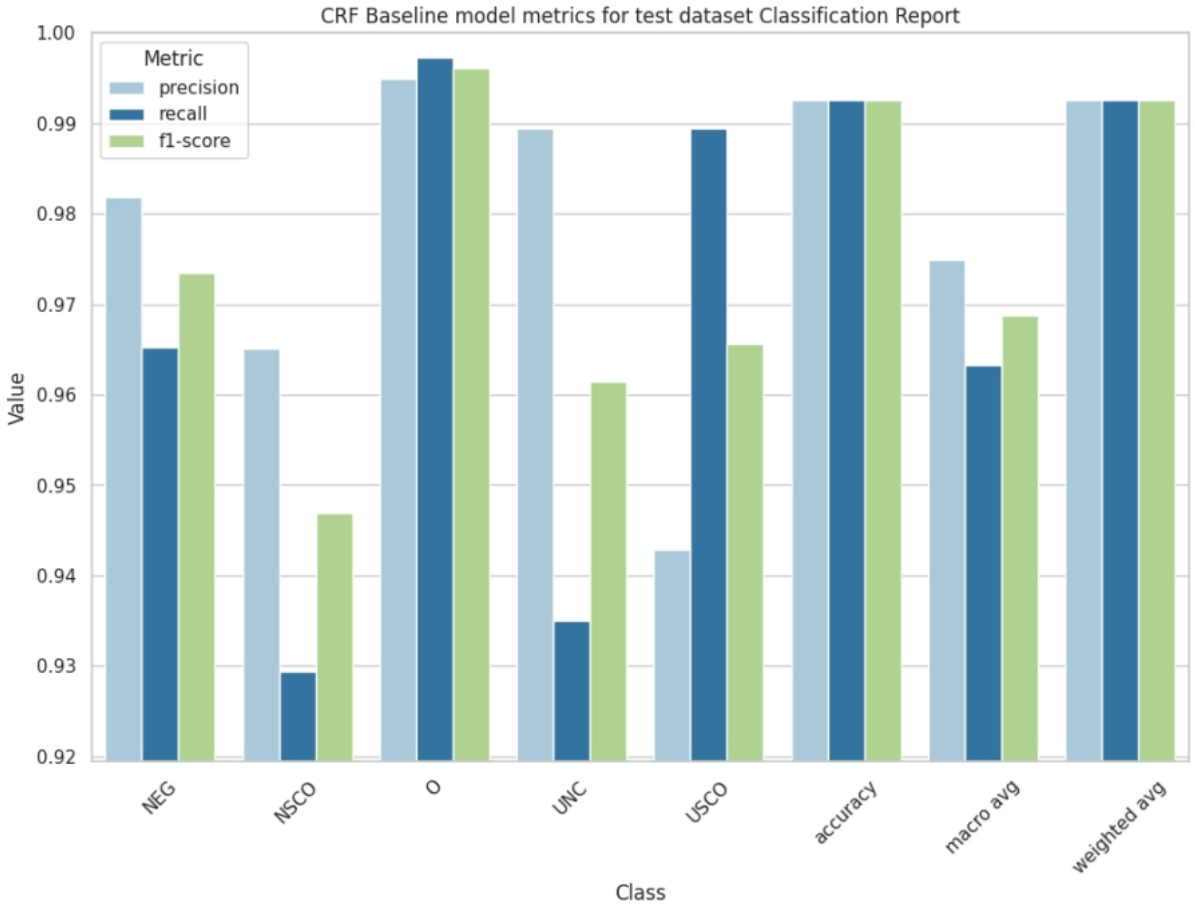


Figure 20: CRF

CRF BIO model

Table 8: Test set evaluation on CRF with BIO Tagging

	precision	recall	f1-score	support
B_NEG	0.9946	0.9901	0.9924	1116
B_NSCO	0.9924	0.9784	0.9853	1065
B_UNC	0.9847	0.9772	0.981	132
B_USCO	0.9923	1	0.9961	129
I_NEG	0.6774	0.6562	0.6667	64
I_NSCO	0.9591	0.9166	0.9373	2505
I_UNC	0.9851	0.9706	0.9778	68
I_USCO	0.9475	0.9886	0.9676	438
O	0.9954	0.9974	0.9964	58388
accuracy	0.9933			
macro avg	0.9476	0.9417	0.9445	63095
weighted avg	0.9932	0.9933	0.9933	63095

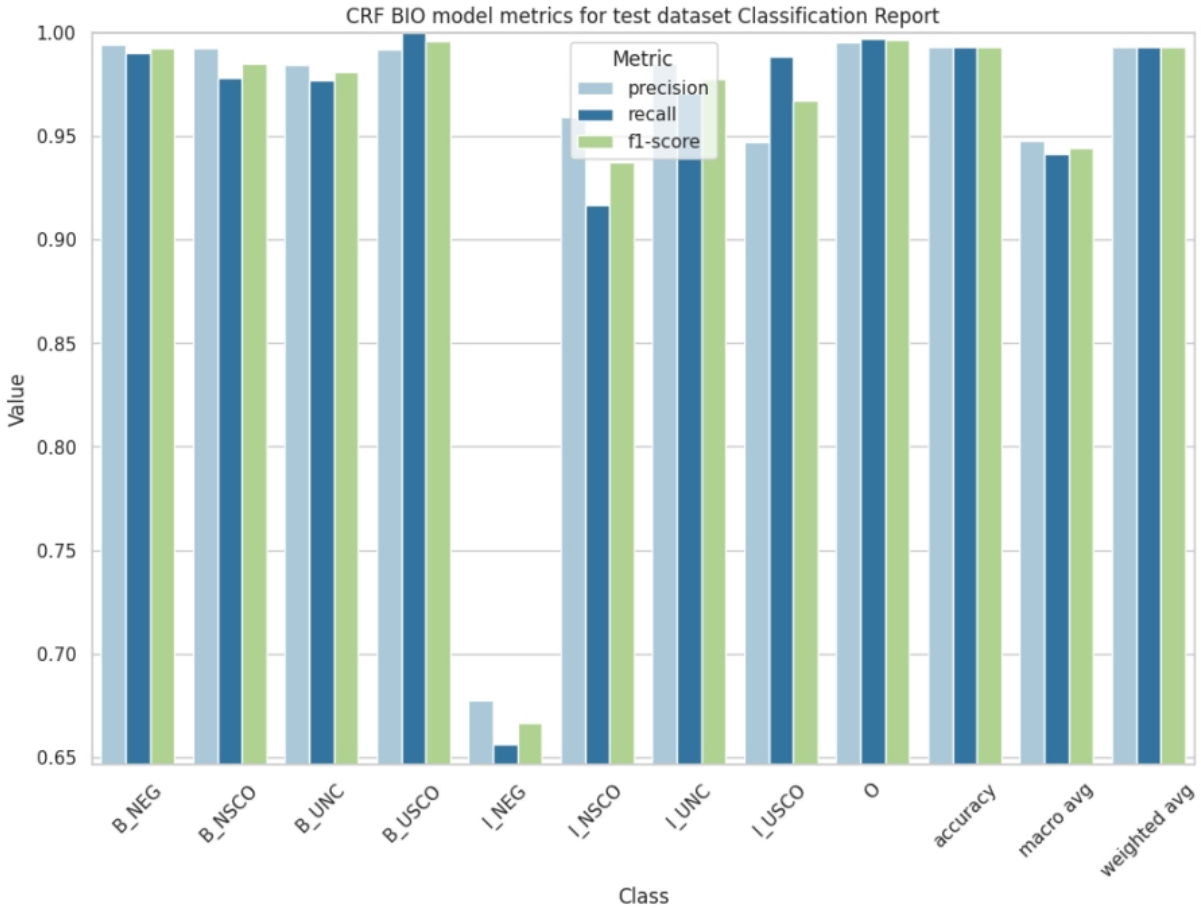


Figure 21: CRF

CRF BIESO model

Table 9: Test set evaluation on CRF with BIESO Tagging

	precision	**recall**	**f1-score**	**support**
B_NEG	1	1	1	62
B_NSCO	1	1	1	789
B_UNC	0.9836	0.9375	0.96	64
B_USCO	0.9914	1	0.9957	115
I_NEG	1	0.5	0.6667	2
I_NSCO	1	1	1	1716
I_UNC	1	0.75	0.8571	4
I_USCO	1	1	1	323
E_NEG	0.9839	0.9839	0.9839	62
E_NSCO	1	1	1	789
E_UNC	1	0.9687	0.9841	64
E_USCO	0.9845	0.9845	0.9845	129
S_NEG	0.9952	0.9905	0.9929	1054
S_NSCO	0.9807	0.9203	0.9495	276
S_UNC	0.9853	0.9853	0.9853	68
O	0.9993	0.9998	0.9996	58388
accuracy	0.9991			
macro avg	0.994	0.9388	0.9599	63095
weighted avg	0.9991	0.9991	0.9991	63095

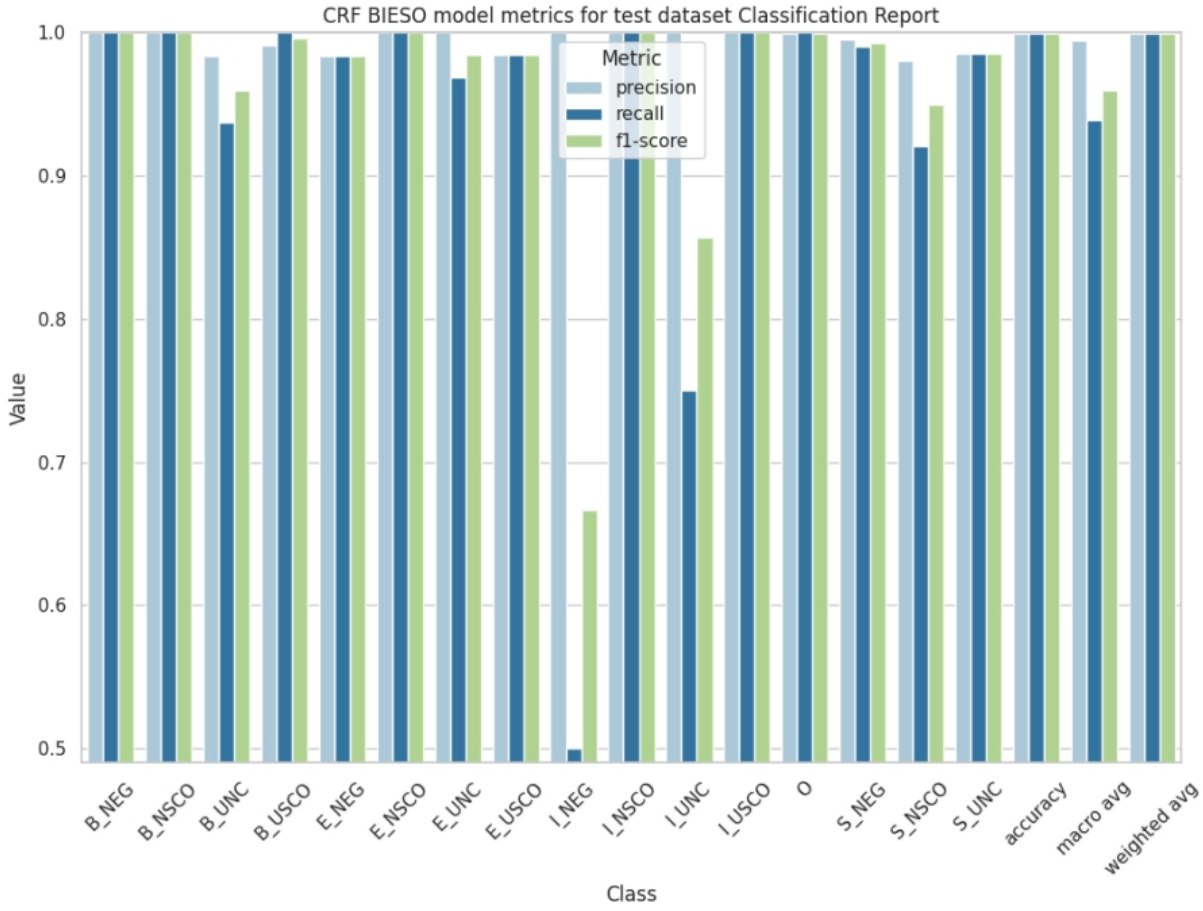


Figure 22: CRF

4.3.2 HMM model

Table 10: Test set evaluation on HMM

	precision	**recall**	**f1-score**	**support**
.	1	1	1	208
NEG	0.9821	0.9322	0.9565	118
NSCO	0.7618	0.7595	0.7606	341
O	0.9794	0.8866	0.9307	3801
UNC	0.5	0.5714	0.5333	7
USCO	0.0235	0.5294	0.045	17
accuracy	0.8816			
macro avg	0.7078	0.7799	0.7044	4492
weighted avg	0.9595	0.8816	0.9177	4492

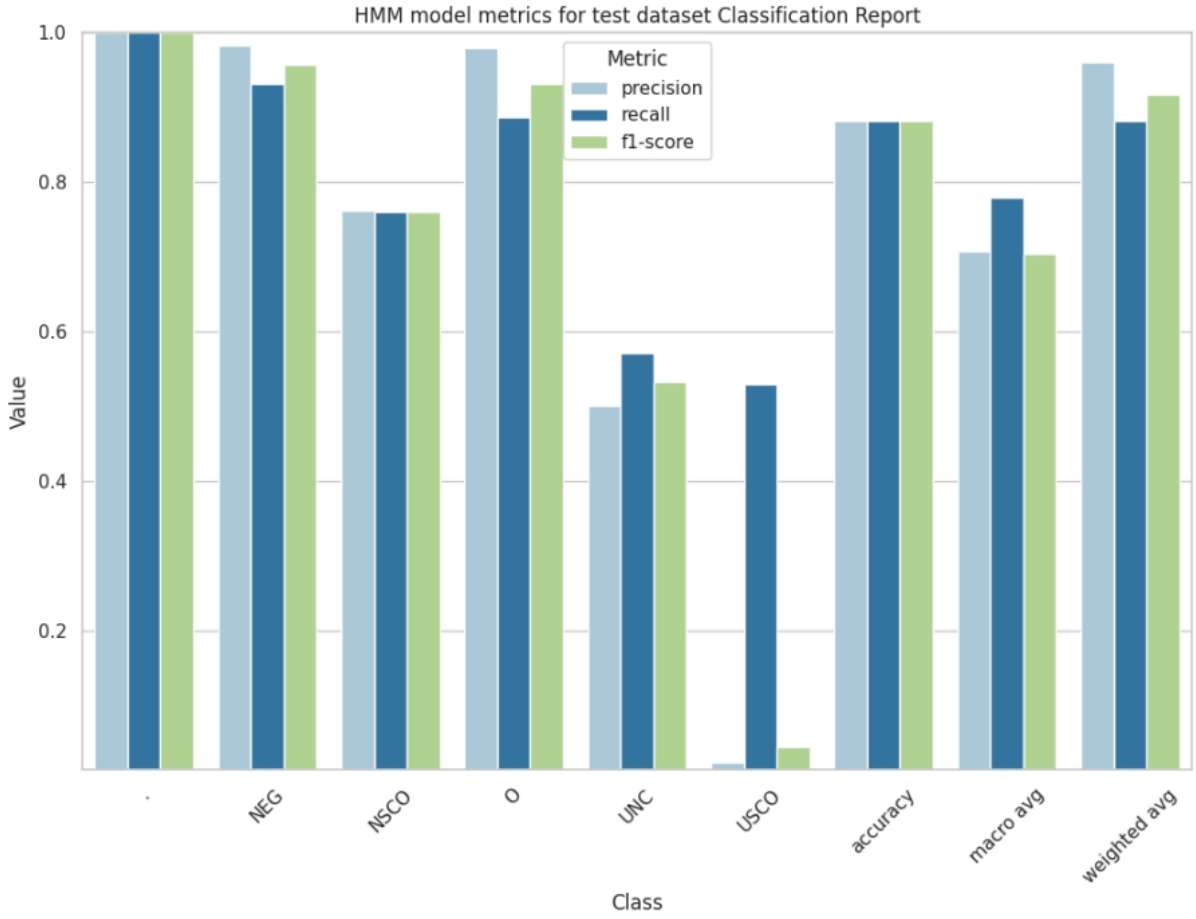


Figure 23: HMM

4.3.3 SVM model

SVM Baseline model

Table 11: Test set evaluation on SVM with Baseline Tagging

	precision	**recall**	**f1-score**	**support**
NEG	0.9769	0.9661	0.9714	1180
NSCO	0.9576	0.9291	0.9431	3570
O	0.9948	0.9967	0.9957	58388
UNC	0.9686	0.925	0.9463	200
USCO	0.9437	0.9753	0.9592	567
accuracy		0.9919		
macro avg	0.9683	0.9584	0.9632	63905
weighted avg	0.9918	0.9919	0.9919	63905

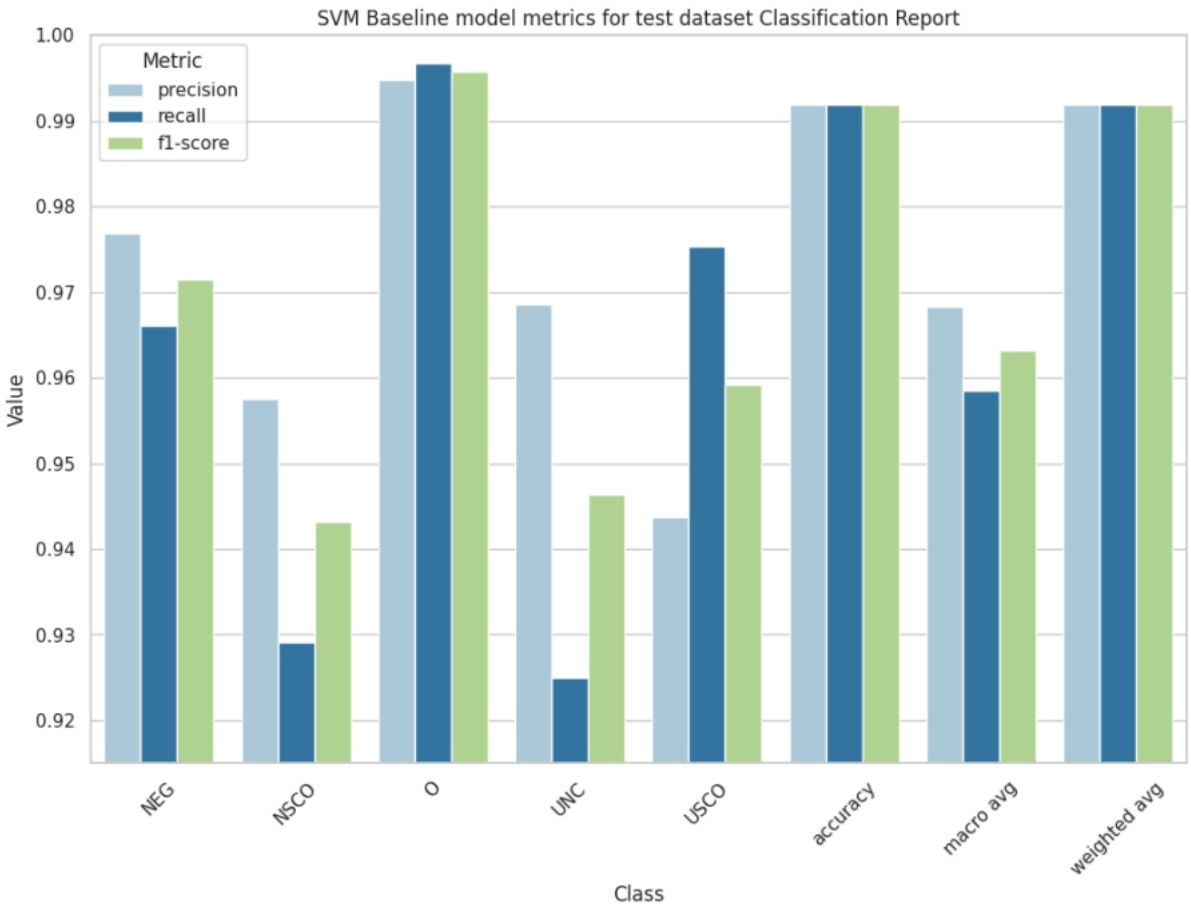


Figure 24: SVM

SVM BIO model

Table 12: Test set evaluation on SVM with BIO Tagging

	precision	**recall**	**f1-score**	**support**
B_NEG	0.9937	0.9892	0.9915	1116
B_NSCO	0.9933	0.9775	0.9853	1065
B_UNC	1	0.9924	0.9962	132
B_USCO	0.9621	0.9845	0.9732	129
I_NEG	0.6866	0.7187	0.7023	64
I_NSCO	0.9504	0.9178	0.9338	2505
I_UNC	0.9706	0.9706	0.9706	68
I_USCO	0.9493	0.984	0.9664	438
O	0.9954	0.9969	0.9961	58388
accuracy	0.9929			
macro avg	0.9446	0.948	0.9461	63905
weighted avg	0.9929	0.9929	0.9929	63905

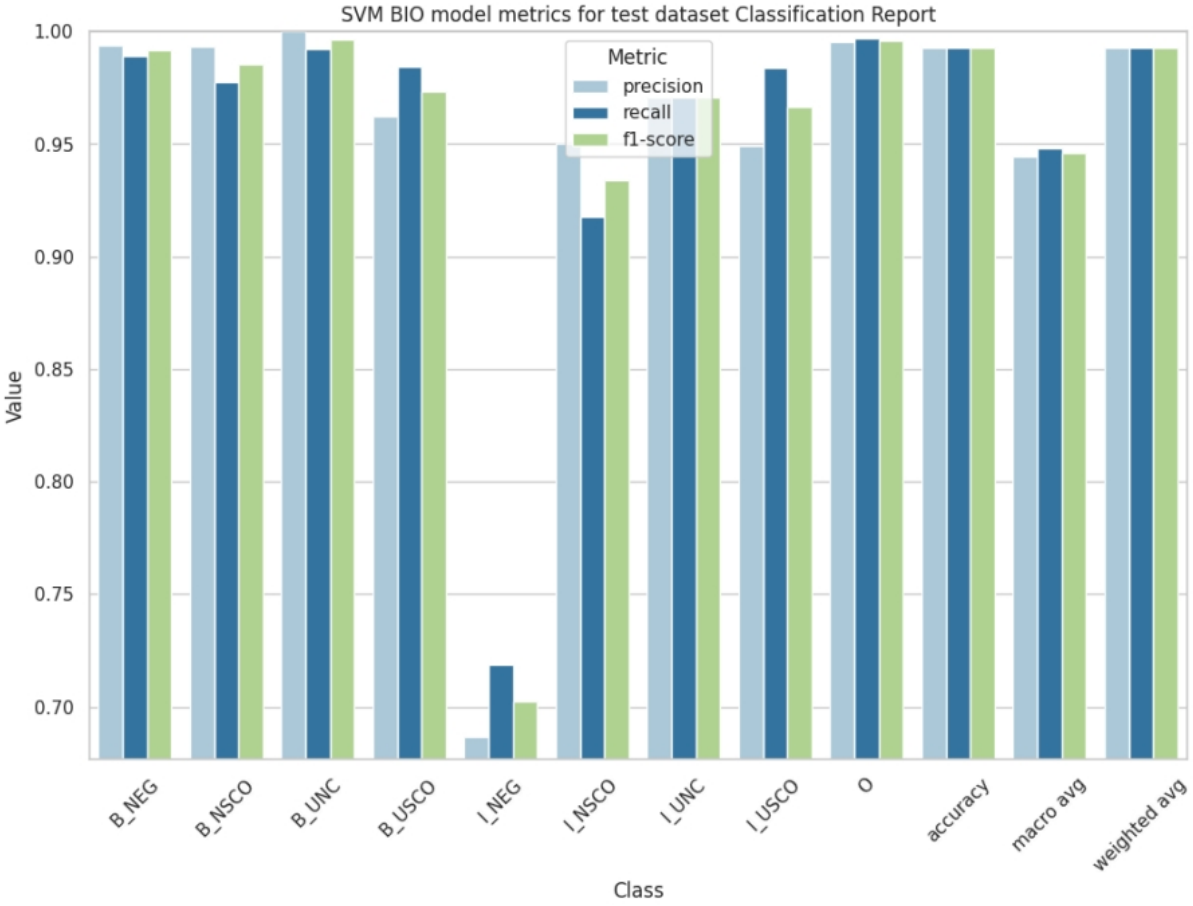


Figure 25: SVM

SVM BIESO model

Table 13: Test set evaluation on SVM with BIESO Tagging

	precision	**recall**	**f1-score**	**support**
B_NEG	1	1	1	62
B_NSCO	0.9975	1	0.9987	789
B_UNC	0.9846	1	0.9922	64
B_USCO	1	1	1	115
I_NEG	1	1	1	2
I_NSCO	1	1	1	1716
I_UNC	1	0.75	0.8571	4
I_USCO	1	1	1	323
E_NEG	1	1	1	62
E_NSCO	1	1	1	789
E_UNC	1	1	1	64
E_USCO	0.9624	0.9922	0.9771	129
S_NEG	0.9952	0.9877	0.9914	1054
S_NSCO	0.9774	0.942	0.9594	276
S_UNC	1	0.9853	0.9926	68
O	0.9995	0.9997	0.9996	58388
accuracy	0.9992			
macro avg	0.9948	0.9785	0.9855	63905
weighted avg	0.9992	0.9992	0.9992	63905

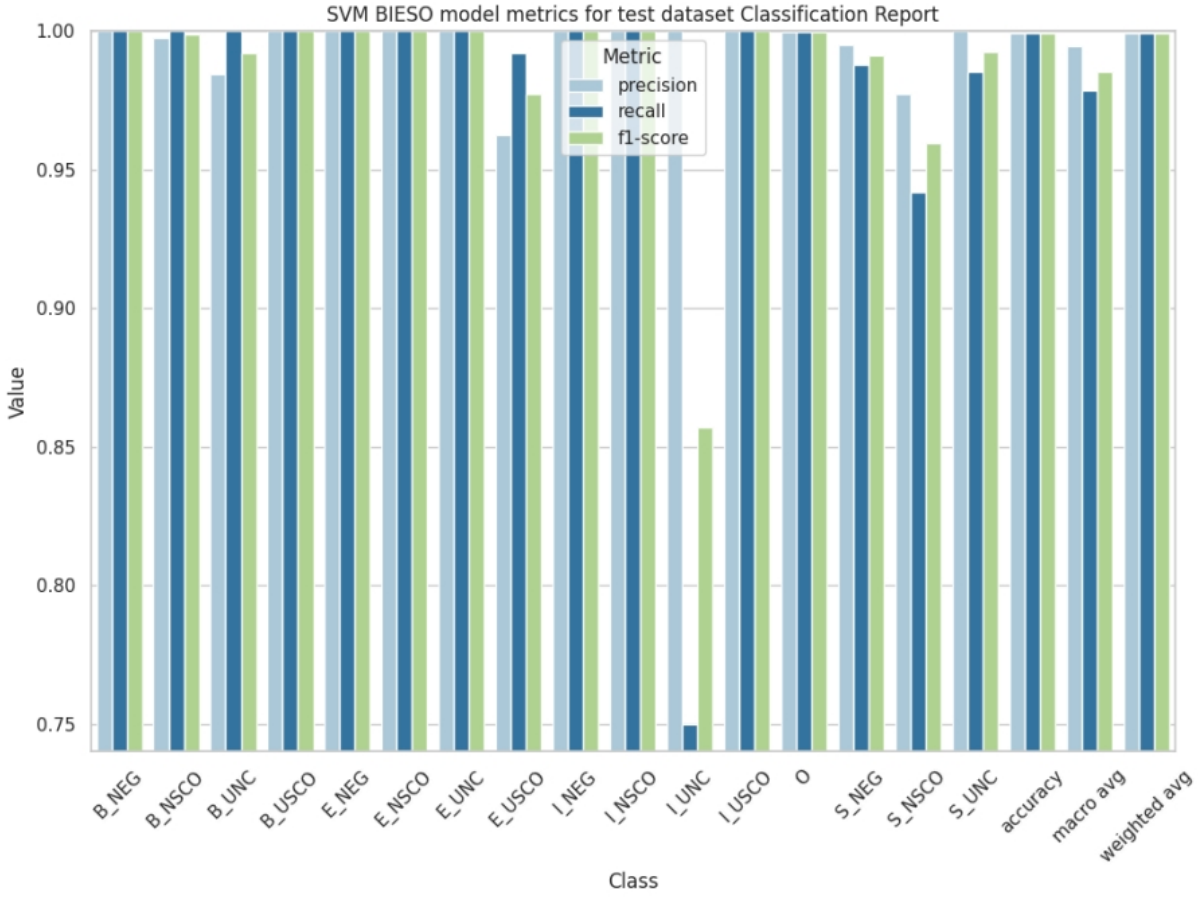


Figure 26: SVM

4.3.4 Naïve Bayes model

Naïve Bayes Baseline model

Table 14: Test set evaluation of Naïve Bayes with Baseline Tagging (DictVectorizer)

	precision	**recall**	**f1-score**	**support**
NEG	0.9089	0.922	0.9154	1180
NSCO	0.7523	0.8745	0.8088	3570
O	0.9883	0.9822	0.9853	58388
UNC	0.9478	0.635	0.7605	200
USCO	0.9194	0.6437	0.7573	567
accuracy	0.971			
macro avg	0.9033	0.8115	0.8454	63095
weighted avg	0.9729	0.971	0.9714	63095

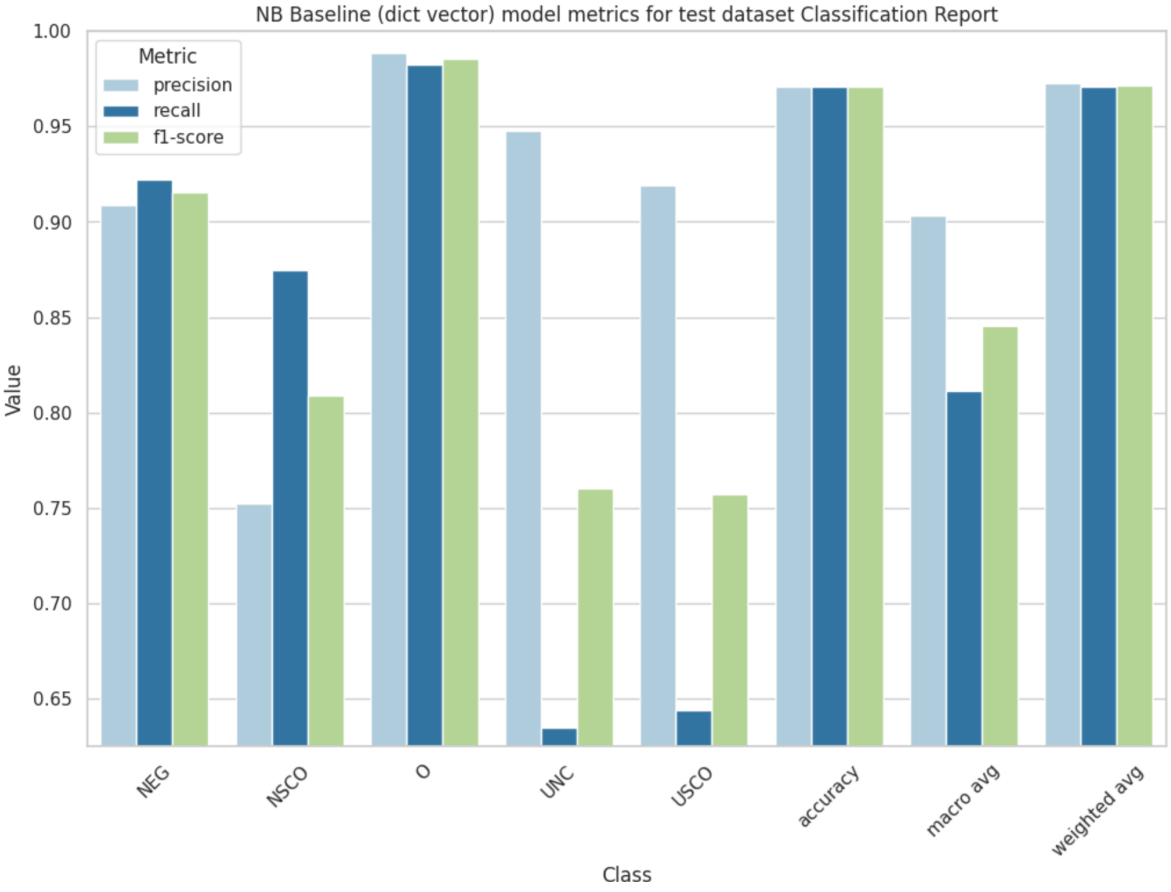


Figure 27: Naïve Bayes Baseline (DictVectorizer)

Table 15: Test set evaluation of Naïve Bayes with Baseline Tagging (Tf-IdfVectorizer)

	precision	**recall**	**f1-score**	**support**
NEG	0.9577	0.8830	0.9188	1180
NSCO	0.8815	0.5481	0.6759	3570
O	0.9592	0.9952	0.9769	58388
UNC	1	0	0	200
USCO	0.875	0.0246	0.0480	567
accuracy	0.9564			
macro avg	0.9346	0.4902	0.5239	63905
weighted avg	0.9542	0.9564	0.9477	63905

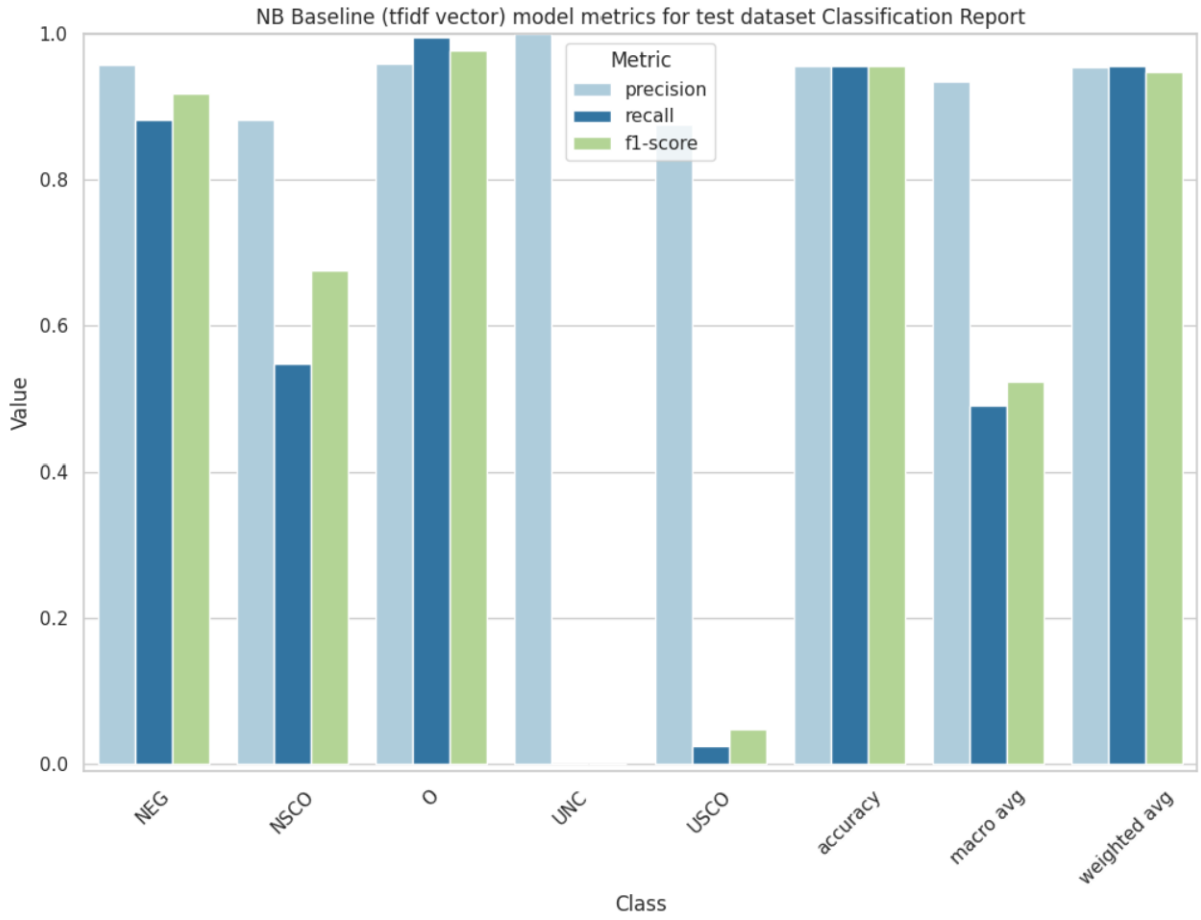


Figure 28: Naïve Bayes Baseline (Tf-IdfVectorizer)

Naïve Bayes BIO model

Table 16: Test set evaluation of Naïve Bayes with BIO Tagging (DictVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	0.9196	0.9749	0.9464	1116
B_NSCO	0.7535	0.9417	0.8372	1065
B_UNC	0.9428	0.75	0.8354	132
B_USCO	0.9701	0.5038	0.6632	129
I_NEG	1	0	0	64
I_NSCO	0.8245	0.8910	0.8564	2505
I_UNC	1	0.4264	0.5979	68
I_USCO	0.9175	0.5844	0.7140	438
O	0.9899	0.9868	0.9884	58388
accuracy	0.9763			
macro avg	0.9242	0.6732	0.7154	63905
weighted avg	0.9777	0.9763	0.9757	63905

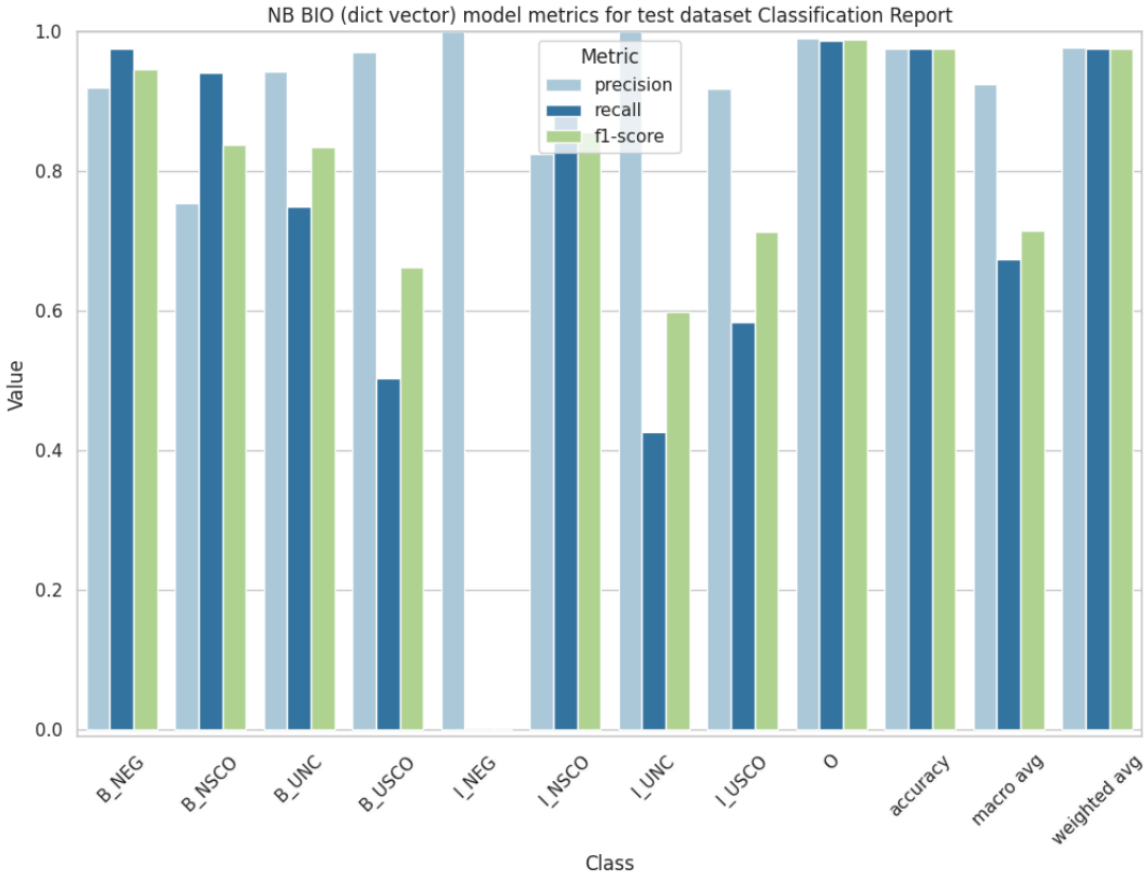


Figure 29: Naïve Bayes BIO (DictVectorizer)

Table 17: Test set evaluation of Naïve Bayes with BIO Tagging (Tf-IdfVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	0.9581	0.9444	0.9512	1116
B_NSCO	0.9592	0.5530	0.7016	1065
B_UNC	1	0.0075	0.0150	132
B_USCO	1	0	0	129
I_NEG	1	0	0	64
I_NSCO	0.8998	0.4554	0.6048	2505
I_UNC	1	0	0	68
I_USCO	0.6666	0.0045	0.0090	438
O	0.9563	0.9978	0.9766	58388
accuracy	0.9553			
macro avg	0.9378	0.3291	0.3620	63905
weighted avg	0.9525	0.9553	0.9444	63905

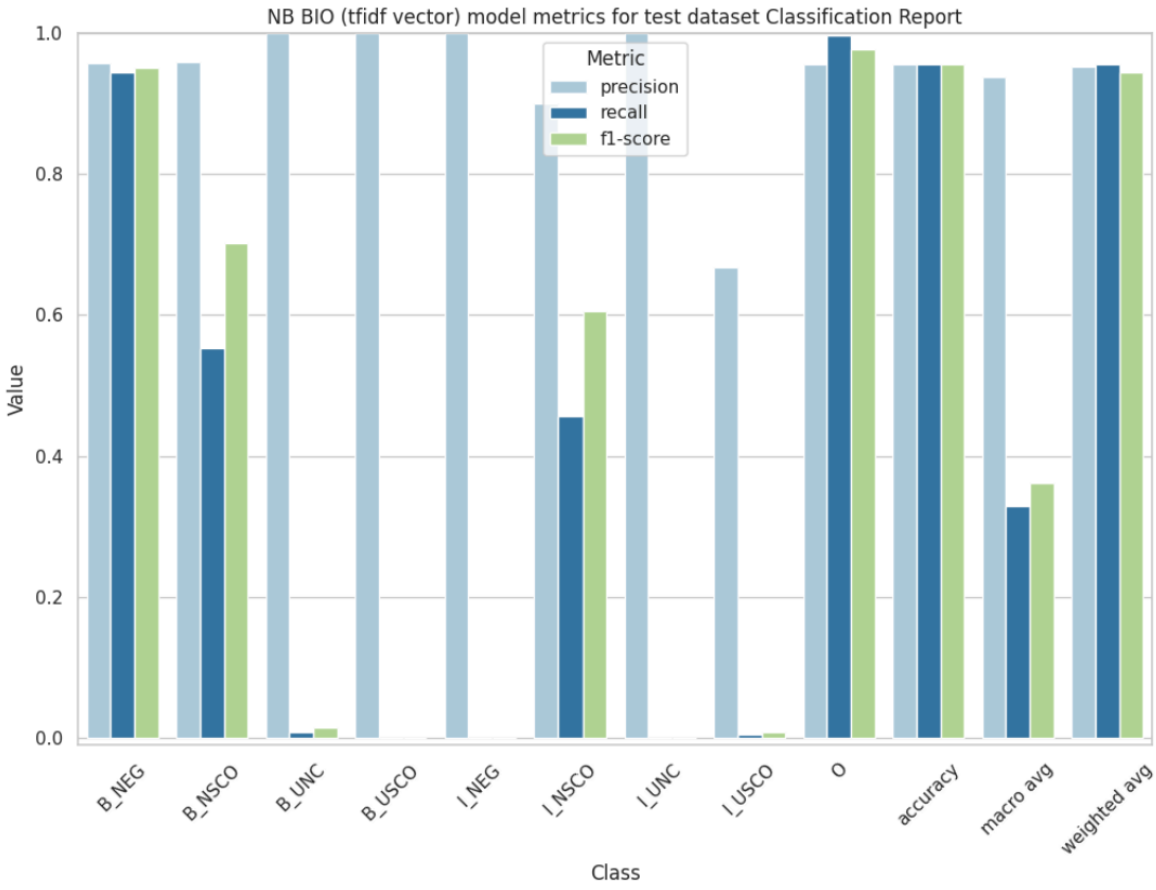


Figure 30: Naïve Bayes BIO (Tf-IdfVectorizer)

Naïve Bayes BIESO model

Table 18: Test set evaluation of Naïve Bayes with BIESO Tagging (DictVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	1	0.2741	0.4303	62
B_NSCO	0.8600	0.9581	0.9064	789
B_UNC	0.9545	0.6562	0.7777	64
B_USCO	0.9843	0.5478	0.7039	115
E_NEG	1	0	0	62
E_NSCO	0.8295	0.7959	0.8124	789
E_UNC	0.9666	0.4531	0.6170	64
E_USCO	0.25	0.0077	0.0150	129
I_NEG	1	0	0	2
I_NSCO	0.9219	0.9842	0.9520	1716
I_UNC	1	0	0	4
I_USCO	0.9464	0.8204	0.8789	323
O	0.9931	0.9964	0.9947	58388
S_NEG	0.8906	0.9810	0.9336	1054
S_NSCO	0.7551	0.5362	0.6271	276
S_UNC	0.9649	0.8088	0.88	68
accuracy	0.9843			
macro avg	0.8948	0.5512	0.5955	63905
weighted avg	0.9829	0.9843	0.9822	63905

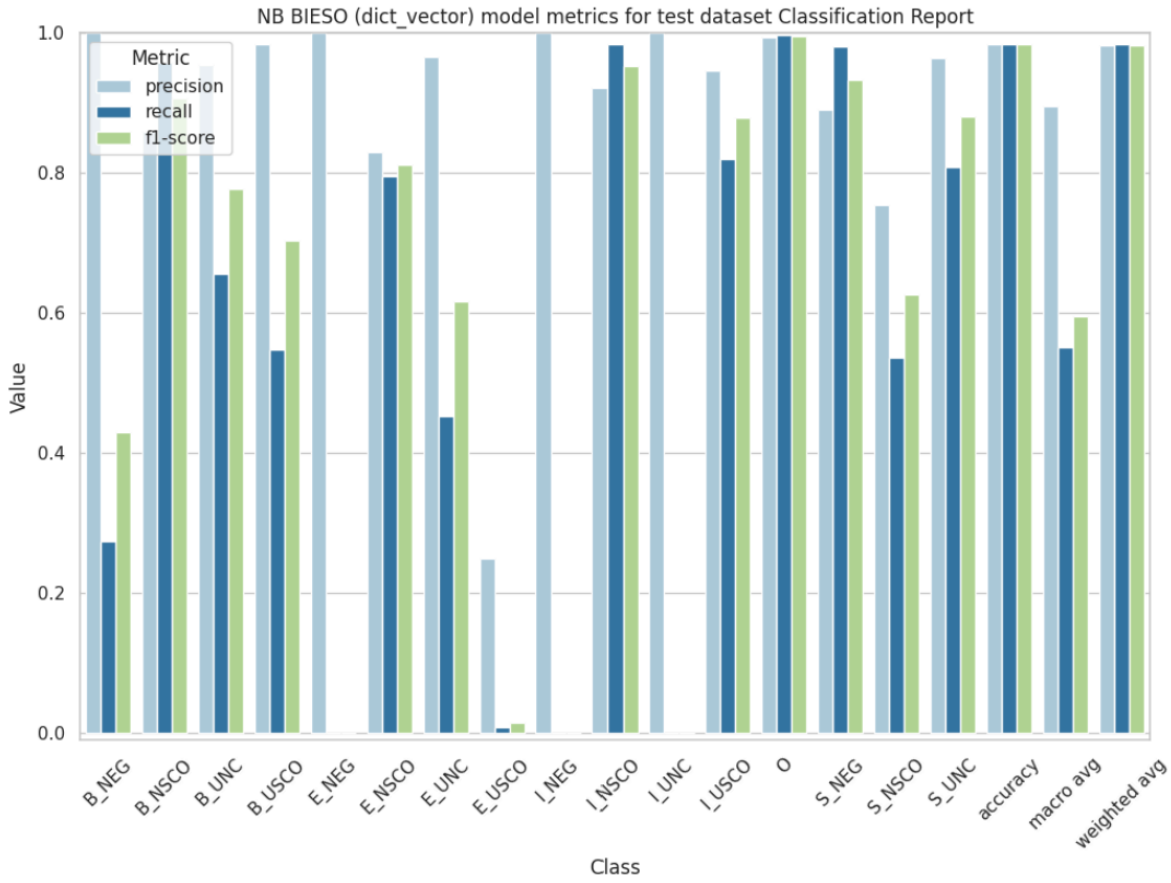


Figure 31: Naïve Bayes BIESO (DictVectorizer)

Table 19: Test set evaluation of Naïve Bayes with BIESO Tagging (Tf-IdfVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	1	0	0	62
B_NSCO	0.8505	0.5627	0.6773	789
B_UNC	1	0	0	64
B_USCO	1	0	0	115
E_NEG	1	0	0	62
E_NSCO	0.9701	0.0823	0.1518	789
E_UNC	1	0	0	64
E_USCO	1	0	0	129
I_NEG	1	0	0	2
I_NSCO	0.9116	0.7092	0.7977	1716
I_UNC	1	0	0	4
I_USCO	1	0.0309	0.0600	323
O	0.9577	0.9989	0.9779	58388
S_NEG	0.9315	0.9421	0.9367	1054
S_NSCO	1	0.0217	0.0425	276
S_UNC	1	0	0	68
accuracy	0.9555			
macro avg	0.9763	0.2092	0.2277	63905
weighted avg	0.9557	0.9555	0.9411	63905

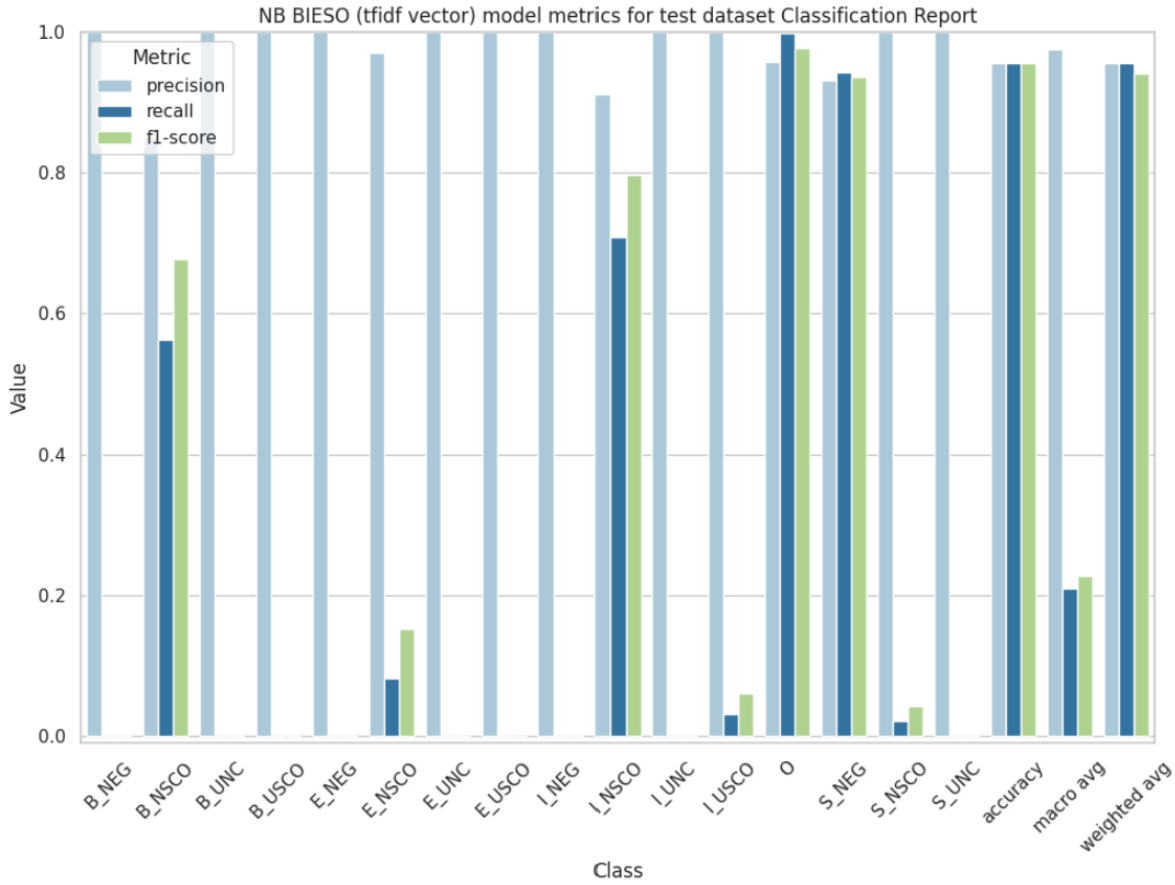


Figure 32: Naïve Bayes BIESO (Tf-IdfVectorizer)

4.3.5 Logistic regression model

Logistic regression Baseline model

Table 20: Test set evaluation of Logistic Regresion with Baseline Tagging (DictVectorizer)

	precision	recall	f1-score	support
NEG	0.9818	0.9610	0.9713	1180
NSCO	0.9635	0.9249	0.9438	3570
O	0.9944	0.9971	0.9957	58388
UNC	0.9786	0.915	0.9457	200
USCO	0.9393	0.9841	0.9612	567
accuracy	0.9920			
macro avg	0.9715	0.9564	0.9635	63905
weighted avg	0.9919	0.9920	0.9919	63905

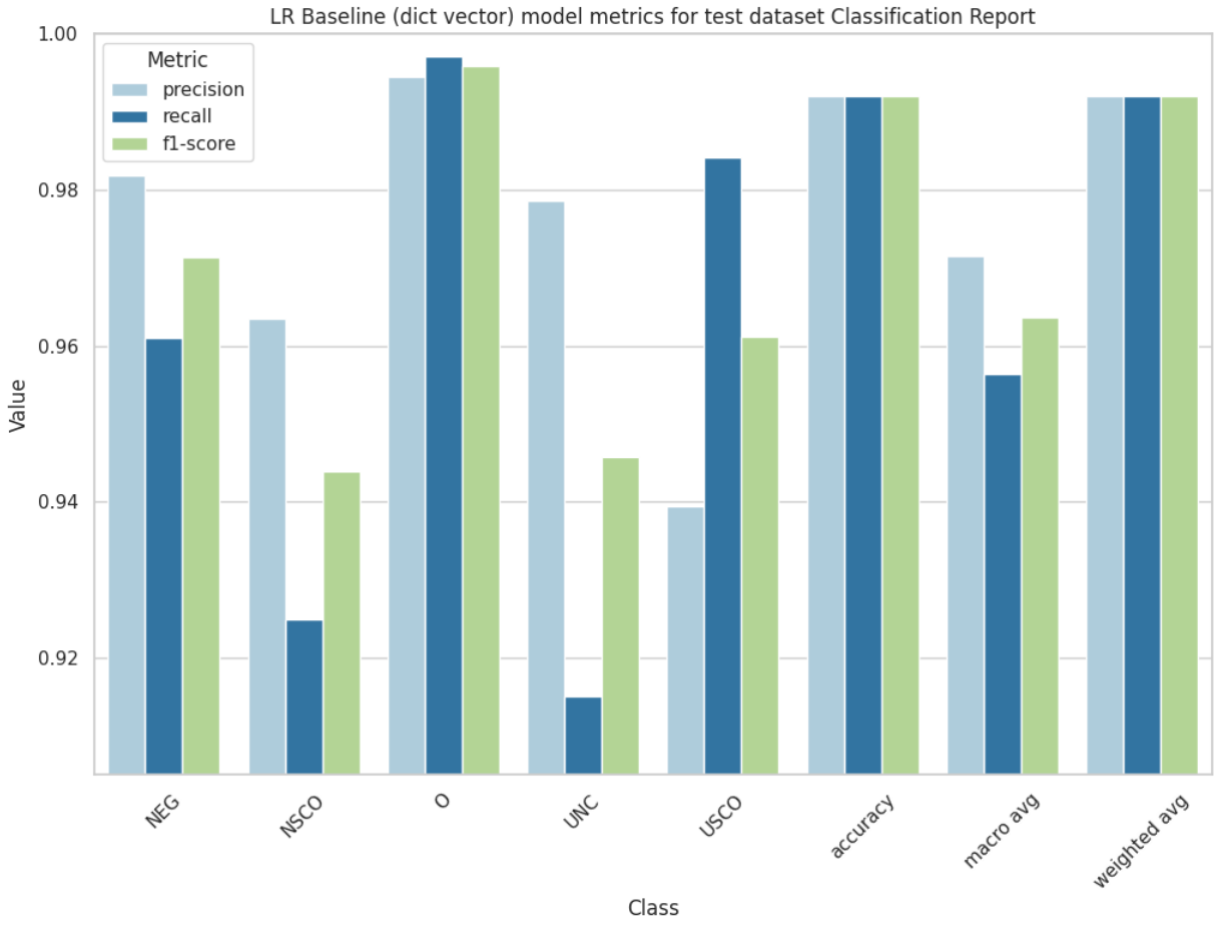


Figure 33: Logistic regression Baseline (DictVectorizer)

Table 21: Test set evaluation of Logistic Regresion with Baseline Tagging (TfIdfVectorizer)

	precision	**recall**	**f1-score**	**support**
NEG	0.9645	0.9466	0.9555	1180
NSCO	0.9555	0.9176	0.9361	3570
O	0.9933	0.9963	0.9948	58388
UNC	0.9883	0.845	0.9110	200
USCO	0.9160	0.9435	0.9296	567
accuracy	0.9900			
macro avg	0.9635	0.9298	0.9454	63905
weighted avg	0.9899	0.9900	0.9899	63905

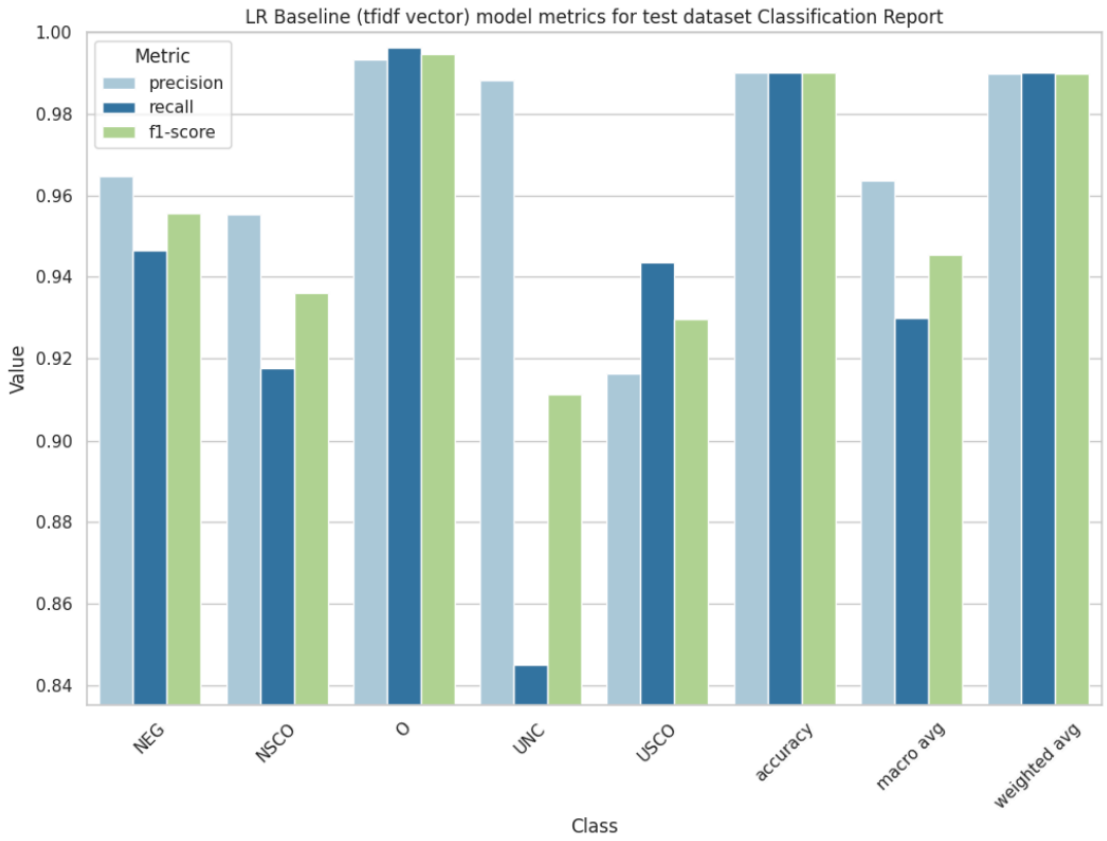


Figure 34: Logistic regression Baseline (Tf-IdfVectorizer)

Logistic regression BIO model

Table 22: Test set evaluation of Logistic Regresion with BIO Tagging (DictVectorizer)

	precision	recall	f1-score	support
B_NEG	0.9945	0.9892	0.9919	1116
B_NSCO	0.9923	0.9765	0.9843	1065
B_UNC	0.9923	0.9848	0.9885	132
B_USCO	1	0.9767	0.9882	129
I_NEG	0.6507	0.6406	0.6456	64
I_NSCO	0.9612	0.9121	0.9360	2505
I_UNC	1	0.9558	0.9774	68
I_USCO	0.9413	0.9885	0.9643	438
O	0.9951	0.9974	0.9962	58388
accuracy	0.9930			
macro avg	0.9475	0.9357	0.9414	63905
weighted avg	0.9930	0.9930	0.9930	63905

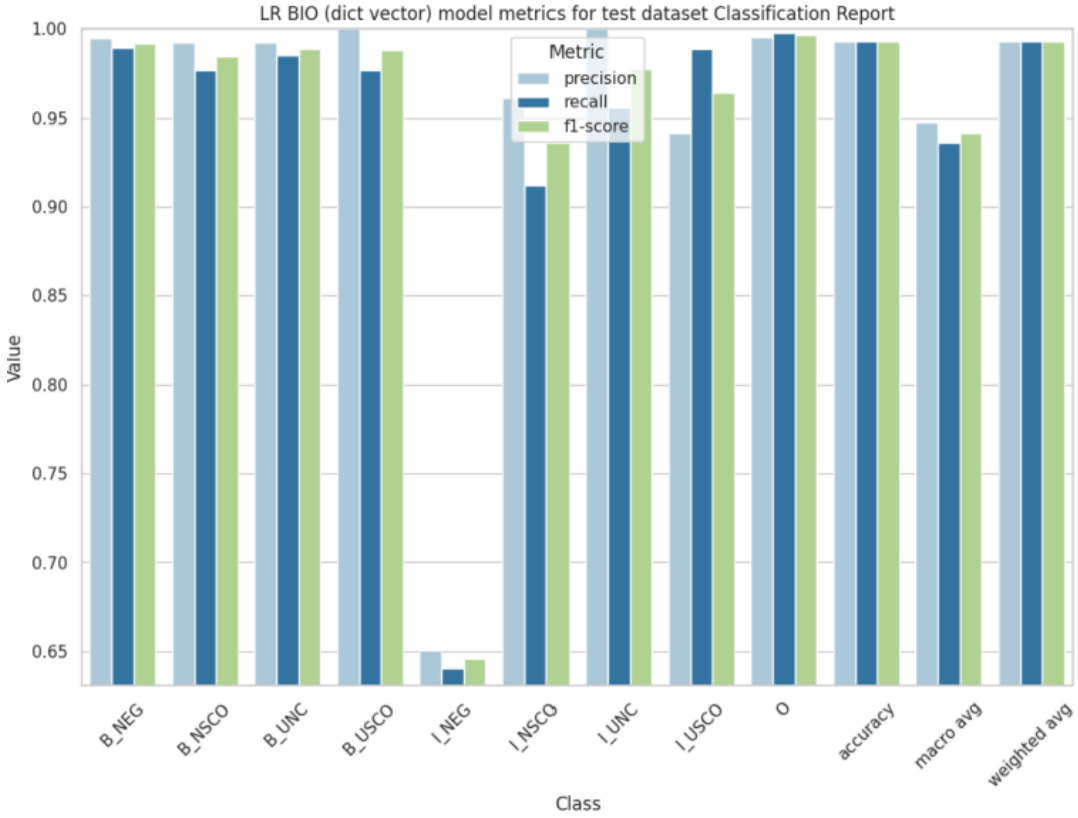


Figure 35: Logistic Regression BIO (dictVectorizer)

Table 23: Test set evaluation of Logistic Regresion with BIO Tagging (Tf-IdfVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	0.9846	0.9793	0.9820	1116
B_NSCO	0.9910	0.9408	0.9653	1065
B_UNC	0.9596	0.9015	0.9296	132
B_USCO	1	0.9069	0.9512	129
I_NEG	0.6734	0.5156	0.5840	64
I_NSCO	0.9498	0.9073	0.9281	2505
I_UNC	1	0.8823	0.9375	68
I_USCO	0.9187	0.9292	0.9239	438
O	0.9932	0.9967	0.9950	58388
accuracy	0.9906			
macro avg	0.9411	0.8844	0.9107	63905
weighted avg	0.9904	0.9906	0.9904	63905

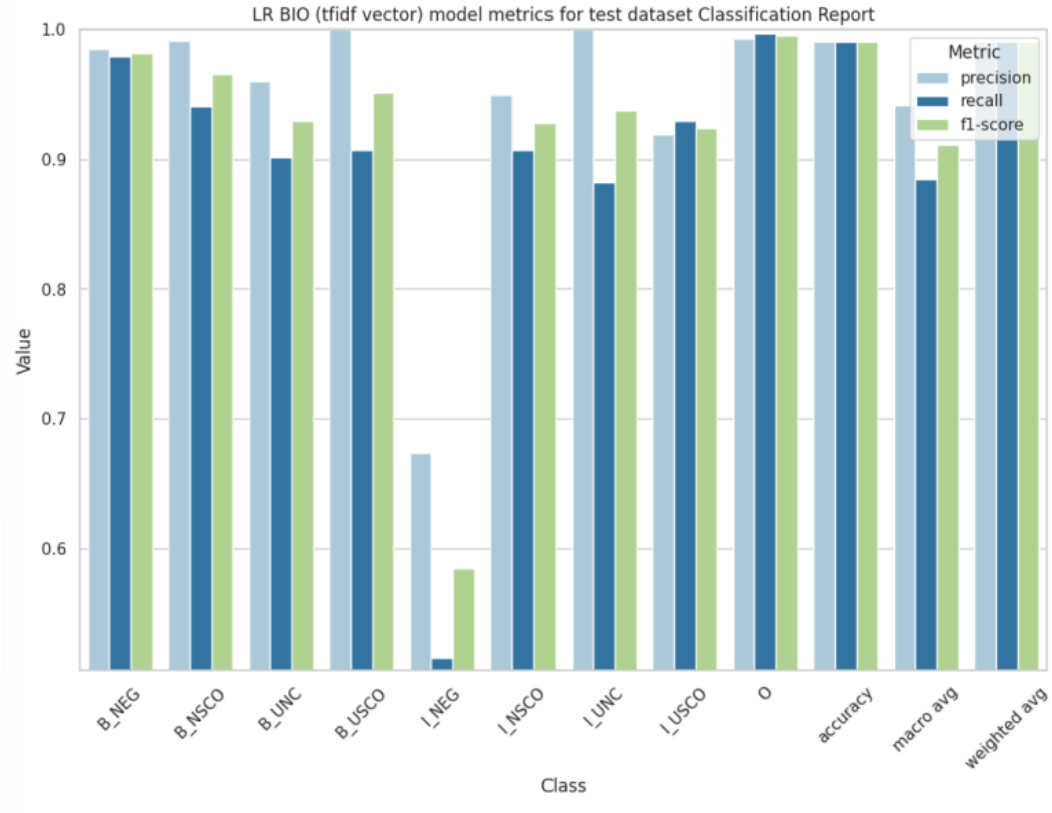


Figure 36: Logistic Regression BIO (Tf-IdfVectorizer)

Logistic regression BIESO model

Table 24: Test set evaluation of Logistic Regresion with BIESO Tagging (DictVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	1	0.9838	0.9918	62
B_NSCO	1	0.9987	0.9993	789
B_UNC	0.9836	0.9375	0.96	64
B_USCO	0.9829	1	0.9913	115
E_NEG	0.9836	0.9677	0.9756	62
E_NSCO	1	1	1	789
E_UNC	1	0.9843	0.9921	64
E_USCO	0.9920	0.9689	0.9803	129
I_NEG	1	0.5	0.6666	2
I_NSCO	1	1	1	1716
I_UNC	1	0.75	0.8571	4
I_USCO	1	1	1	323
O	0.9992	0.9998	0.9995	58388
S_NEG	0.9923	0.9867	0.9895	1054
S_NSCO	0.9766	0.9094	0.9418	276
S_UNC	0.9852	0.9852	0.9852	68
accuracy	0.9989			
macro avg	0.9934	0.9357	0.9581	63905
weighted avg	0.9989	0.9989	0.9989	63905

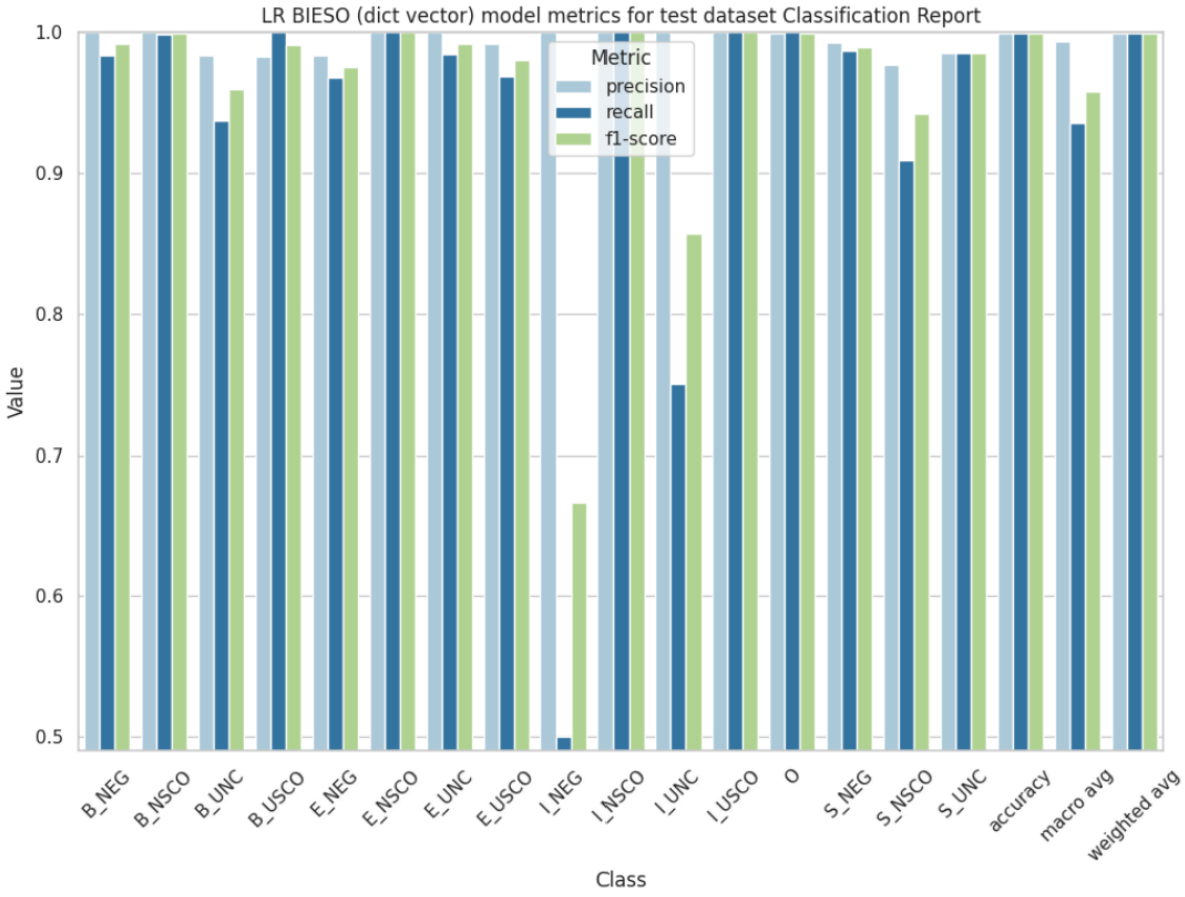


Figure 37: Logistic Regression BIESO (dictVectorizer)

Table 25: Test set evaluation of Logistic Regresion with BIESO Tagging (Tf-IdfVectorizer)

	precision	**recall**	**f1-score**	**support**
B_NEG	1	0.9354	0.9666	62
B_NSCO	1	0.9759	0.9878	789
B_UNC	0.9310	0.8437	0.8852	64
B_USCO	0.9913	0.9913	0.9913	115
E_NEG	0.9672	0.9516	0.9593	62
E_NSCO	1	0.9873	0.9936	789
E_UNC	1	0.9375	0.9677	64
E_USCO	1	0.7286	0.8430	129
I_NEG	1	0	0	2
I_NSCO	0.9976	1	0.9988	1716
I_UNC	1	0	0	4
I_USCO	1	1	1	323
O	0.9973	0.9996	0.9984	58388
S_NEG	0.9781	0.9781	0.9781	1054
S_NSCO	0.9734	0.7971	0.8764	276
S_UNC	0.9830	0.8529	0.9133	68
accuracy	0.9968			
macro avg	0.9887	0.8112	0.8350	63905
weighted avg	0.9968	0.9968	0.9967	63905

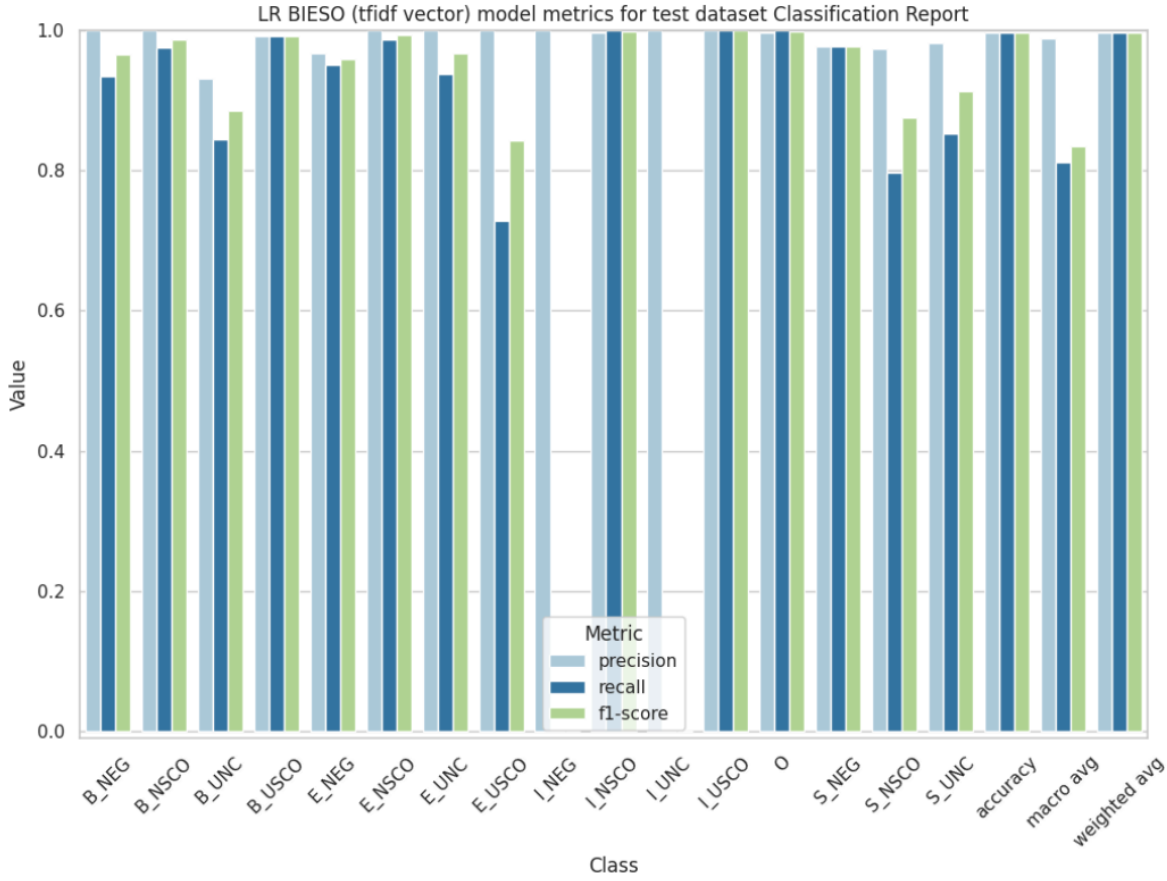


Figure 38: Logistic Regression BIESO (Tf-idf Vectorizer)

4.3.6 Comparison plots

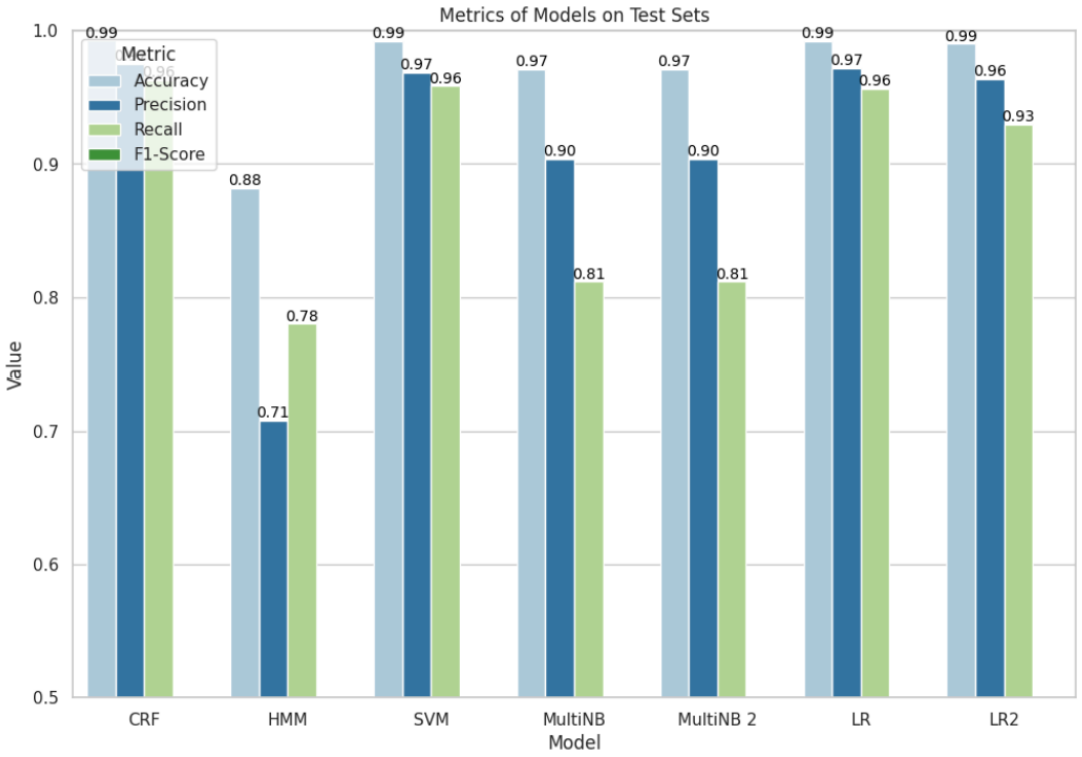


Figure 39: BASELINE

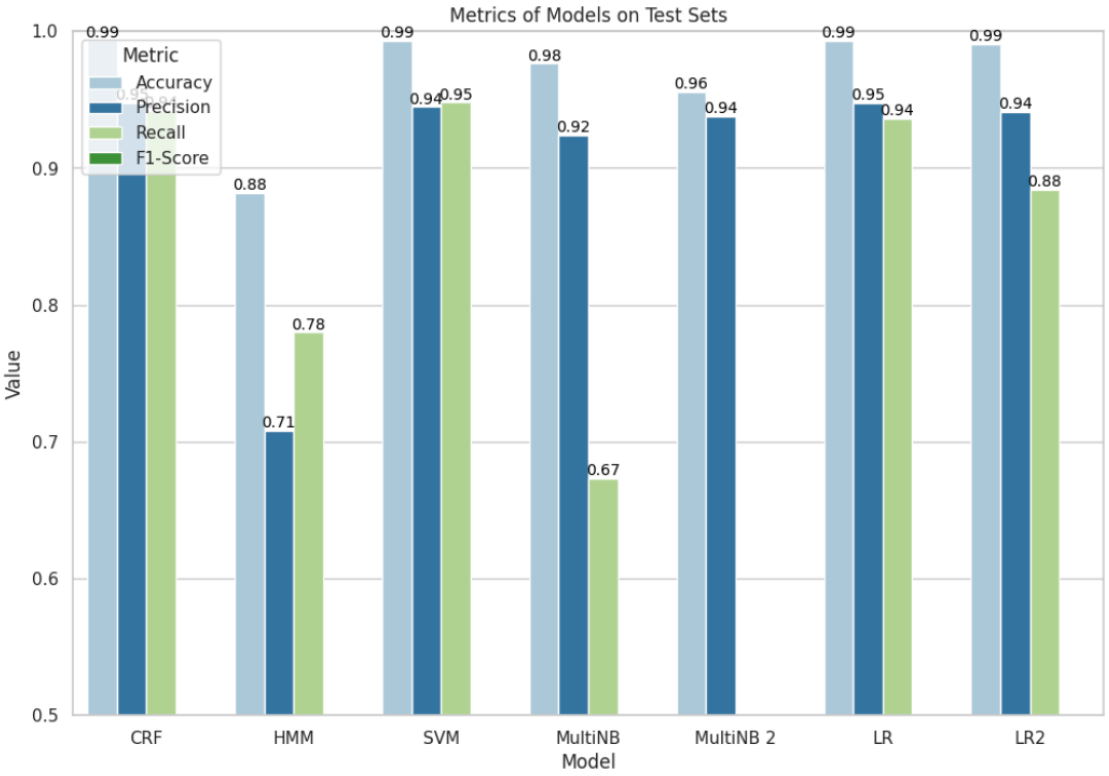


Figure 40: BIO

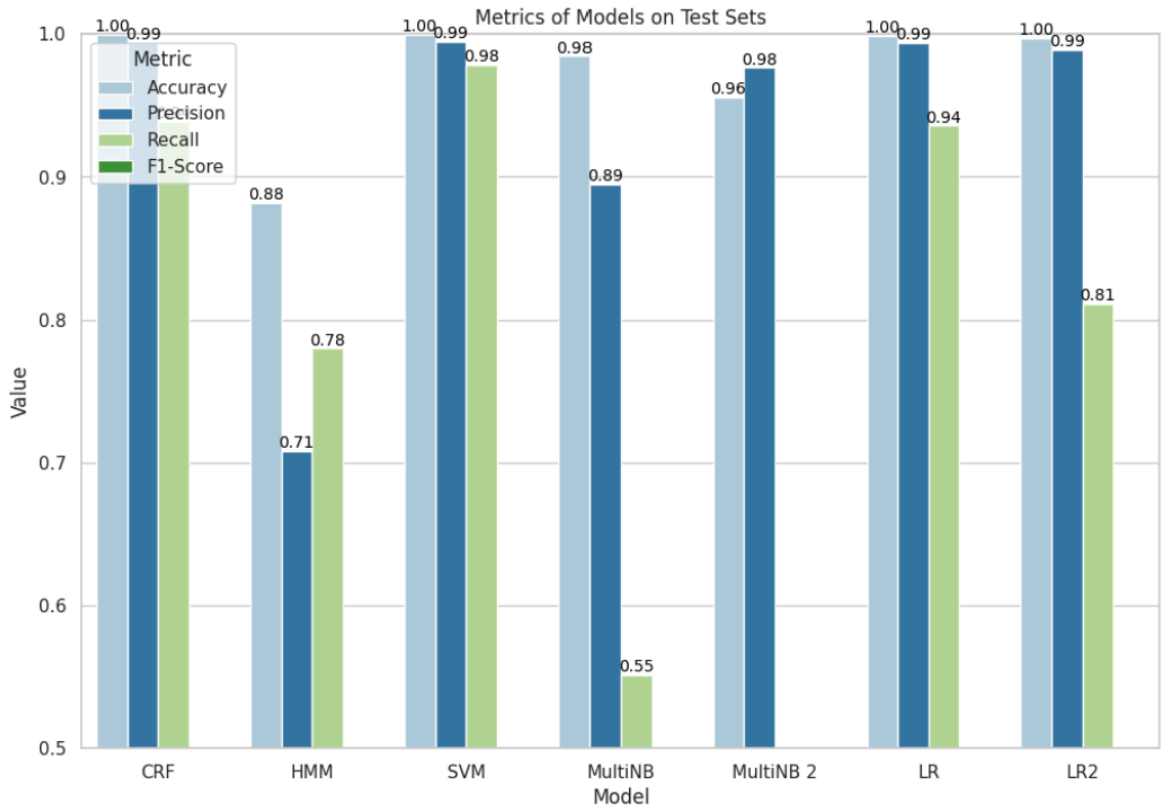


Figure 41: BIESO

4.4 Problems:

- Deciding which models to implement
- Understand what kind of input expected each model
- Create the correct taggers
- Some models require a lot of time to train with CPU

4.5 Conclusions on Machine-Learning Approach:

At the beginning, after all the research about machine learning implementation in NLP, we have adapted our datasets. We have created some tagging algorithms which take from the simplest and basic ones to use beginning and end tags of the labeled word. We decided to begin with the CRF and it was one that was more recurrently mentioned in our research and it was quite easy.

For this model the results obtained were very favorable. The accuracy of this model in any of the cases (Baseline, BIO or BIESO) is 99% for the test set. The other matrices are also above 92%. We were actually very satisfied. Nevertheless we thought that we could try other methods to see if we could get even better results.

Our second choice was HMM. This algorithm actually can be used to describe the evolution of observable events that depend on internal factors, which are not directly

observable and was said to be quite effective in Machine Learning processes. As a result of this method it is true that the accuracy (how many target values matches the predicted values) was quite high, 88%, while some other matrices failed to improve the wanted results. The recall, which is the correctly predicted positive instances, was around 73%, which could mean that some words were tagged wrongly while predicting for the test set. An important drawback from this model is that the Viterbi algorithm required hours to run on the entire dataset. That is why we decided to evaluate only on a few texts chosen at random, but we believe increasing the test size could yield better results.

After more exploration we decided to work out with SVM. This implementation actually was the best overall (considering the plots above). All the different measurements taken to evaluate the model were above 94%. So, it was performing correctly when measuring the accuracy of positive predictions, and also in identifying the words correctly (tagging them with their expected label).

Not content with these algorithms, it occurred to us to try simpler models such as Logistic Regression or Naïve Bayes. For the last one, we need to consider that it is a probabilistic classifier which assumes that the features are conditionally independent given the target class. The strength (naivety) of this assumption is what gives the classifier its name. This model actually did not work out. Firstly, as it mentioned in the implementation explanation, we tried to see how different vectorizers could condition our results. Tf-idf was useless as some metrics could not even be considered. In contrast, DictVectorizer was actually acceptable for Baseline cases (recall was above 80% and the accuracy and precision were higher than 90%). However, with both vectorizers when we tried to use more complex tagging methods like BIESO it did not work.

On the other hand, Logistic Regression, while it has some limitations, such as the assumption of a linear relationship between the input features and the class labels, it remains a useful and practical approach to text classification as it is simple, understandable and versatile. This model required some time to predict its model and the amount of time needed got higher as the tagging used was more complex. In this case both vectorizers actually got decent evaluations. It is confirmed that DictVectorizer works better than TF-IdfVectorizer - last bar. Nonetheless, the use of DictVectorizer actually has been conditioned on the number of iterations for this model to train. We believe that the values will change if more time was given, maybe getting similar results to the Tf-IdfVectorizer. At the end the performance of this model overall was above 94%.

In conclusion, the methods did actually perform well in general. They give reasonably high accuracy and good results. But we believe that to improve the obtained conclusions maybe it will require having a more computationally expensive model, such as HMM using the Viterbi algorithm or SVM models, and because the amount of data that we have may force us to use GPU or a similar tool with those characteristics. In the end, we can say that CRF and SVM are the best models based on their evaluations.

One remarkable thing is that in most of the cases the methods implemented with

BIESO is the one with higher results, followed by the same model with no tagging (baseline) and finally with the BIO approach. We think that BIESO is the one with maximum results because as it differentiates between beginning, inside, end, single-word entities and non-word entities they provide more context so the model understands better the structure of entities in a text, which leads to better performance. Also, comparing it with BIO, if an error occurs the BIESO scheme can potentially limit the impact of such errors by providing more opportunities for the model to correct itself.

Regarding why the baseline model has better performance than BIO could be for different factors. It could happen that for this dataset, as it contains a remarkable number of single-word entities, the baseline model could detect and/or identify them better. Also the other way around, it has plenty of multi-word entities, so those two things combined could lead to errors with BIO tagging. That is why the baseline model is a more robust model than the algorithm combined with BIO tagging

In summary, we can say that this second implementation has given us the chance to see how an algorithm evolves from something like basic RegEx to more complex methods and advanced methods like machine learning ones. Our main advantage is that for our previous steps (preprocessing data) was well designed based on the results that we obtained so the models have good datasets to work with.

Moreover we have been able to use some of the concepts learnt in class and we have managed to incorporate them correctly in the process (CRF, HMM, SVM...). Nonetheless, it has been quite difficult to decide which method to implement and how it could be more beneficial for the kind of corpus we are working with. At the end, the code needed to interpret and learn accordingly the different words in the text.

5. DEEP LEARNING APPROACH:

Deep learning approaches excel in negation and uncertainty detection due to their ability to learn complex language patterns from large datasets. Unlike traditional rule-based systems, deep learning models, such as neural networks, can capture the nuances and variability of natural language, which makes them highly effective for accurately identifying negation and uncertainty.

5.1 Methodology:

The first task for this approach was to do some research on the most common Deep Learning methods for negation and uncertainty detection. Most of the papers that addressed the same problem mentioned LSTMs, a type of recurrent neural network (RNN) architecture designed to better capture long-term dependencies in sequential data.

After setting the LSTM as the starting point, we still had to define its architecture, how to preprocess the data, the different hyperparameters to try and how to evaluate our results.

As in the Machine Learning approach, experiments were performed on three types of tagging: the original tags, BIO and BIESO.

5.2 Implementation:

5.2.1 Data preprocessing

First, some preprocessing operations were performed. The data from the files was extracted and transformed into a suitable format for further processing. The texts were extracted from the JSON files (both training and test data) and were stored in two variables: x_train and x_test, each containing a list of strings representing different texts. The ground truth results for both the training and test data were also extracted to facilitate the evaluation of model metrics such as accuracy and precision. As in the machine learning approach, the ground truth tags were also converted with BIO and BIESO tagging. Therefore, we had y_train and y_test for normal tags, y_train_bio and y_test_bio for BIO tags and y_train_bieso and y_test_bieso for BIESO tags. Moreover, we passed the text and tags to a format that was suitable for the neural networks. New datasets with the format [(text,labels),...,(text,labels)] were created for each type of tagging. On top of that, since the algorithms only understand numerical data, we created tag_to_ix and ix_to_tag dictionaries for each type of tagging, where each tag had been mapped to a different number. We also obtained a word_to_ix dictionary, where every unique word in all the medical texts from the training set is associated with a number, including a special "<OOV>" token to later handle unseen words.

5.2.2 Algorithm implementation

RNNs (Long Short-Term Memory networks (LSTMs)) are designed to handle sequential data effectively, making them well-suited for processing sentences where word order is crucial. They maintain a hidden state that captures information from previous time steps, allowing them to leverage context from earlier words to inform the tagging of the current word.

Moreover, they can capture long-range dependencies over extended sequences. By processing words one at a time, it incrementally builds an understanding of the sentence, which is essential for tasks where a word's meaning depends on its context. **LSTM**

architecture

The LSTM architecture is kept as simple as possible for computational reasons and to avoid overfitting. It consists of the following layers:

- Embedding layer: the input (a sequence of word indices) is passed through the embedding layer to get the word embeddings (dense vector representations of words that capture their meanings, semantic relationships, and syntactic roles in a low-dimensional space).

- LSTM layer: The word embeddings are reshaped and fed into the LSTM. The LSTM outputs the hidden states (lstm_out) for each time step in the sequence, and updates its hidden states (self.hidden). This allows to capture temporal dependencies.

- Linear layer: the output of the LSTM (lstm_out) is reshaped and passed through the linear layer to produce tag scores for each word in the sentence.

- Log-Softmax: the raw scores from the linear layer are converted to log-probabilities using F.log_softmax, which are then returned as tag_scores.

Fine-tuning

Once we had the base architecture for our model, we had to try multiple hyperparameters combinations. This step allows us to control the behavior of the training process, since the parameters that we choose will condition how our model learns. It is important to be able to improve the model's performance on unseen data, speed up the training process and reduce the computational time and cost. Also, choosing good hyperparameters ensures that the model converges to an optimal solution, rather than getting stuck in a local minima. This step helped us to prevent overfitting and underfitting too. We decided to test the hyperparameters only with the Baseline tags because the computational cost of training multiple times was very high. The final best hyperparameters will also be used with the BIO and BIESO tags.

For the loss function, we had two options: NLLLoss (Negative Log Likelihood loss) or Cross Entropy Loss. Both are used for classification tasks, but NLLLoss is used for tasks where the outputs are probabilities obtained from a softmax layer, whereas Cross Entropy Loss combines a softmax layer and NLLLoss in a single step, applying the softmax

function to the raw scores to obtain probabilities and then computing the negative log likelihood loss.

For the optimizers, we had four in mind: SGD, Adam, RMSProp and AdaGrad. SGD (Stochastic Gradient Descent) updates the model parameters using the gradient of the loss function with respect to each parameter. However, it can be slow to converge and can get stuck in local minima. But it is simple and effective for many problems. AdaGrad (Adaptative Gradient Algorithm): it adapts the learning rate to the parameters by doing smaller updates to recurrent features and larger updates to infrequent features. The problem is that since the learning rate continuously decreases, it can make the model stop training too early. RMSProp (Root Mean Square Propagation): it also adapts the learning rate for each parameter by dividing the gradient by an average of the magnitudes of recent gradients. This optimizer require a careful tuning of hyperparameters. Adam (Adaptative Moment Estimation): It combines the advantages of AdaGrad and RMSprop: it maintains an adaptive learning rate for each parameter and also adjusts the learning rate based on the first and second moments of the gradients. By first impressions, the Adam optimizer seems like the best option, but we still have to put it into test.

Finally, regarding the learning rate, we decided to try with two different values: 0.1 and 0.01. The learning rate determines the step size at each iteration while moving towards the minimum of the loss function. A high learning rate (0.1) leads to a faster convergence, but with the risk of overshooting the minima, which leads to divergence instead of convergence. A moderate learning rate (0.01) is more stable. In addition, a low learning rate (0.001) can lead to a more precise convergence, but it is much slower. That's why, since we observed that the high learning rate was not overshooting in most cases and the moderate learning rate worked well, we discarded the low one because of the computational cost.

The following graphs, created using the Weights and Biases website with the wandb library for python, show the loss and accuracy evolution for each hyperparameter combination. Apart from the data stored in Weight and Biases, we also have stored more specific metric for each combination, such as precision, recall, f1-score, etc, overall and for each different tag. However, we did not include it in the report as we thought it was not relevant, since the final decision on which were the best hyperparameters was based on the final train and test accuracy.

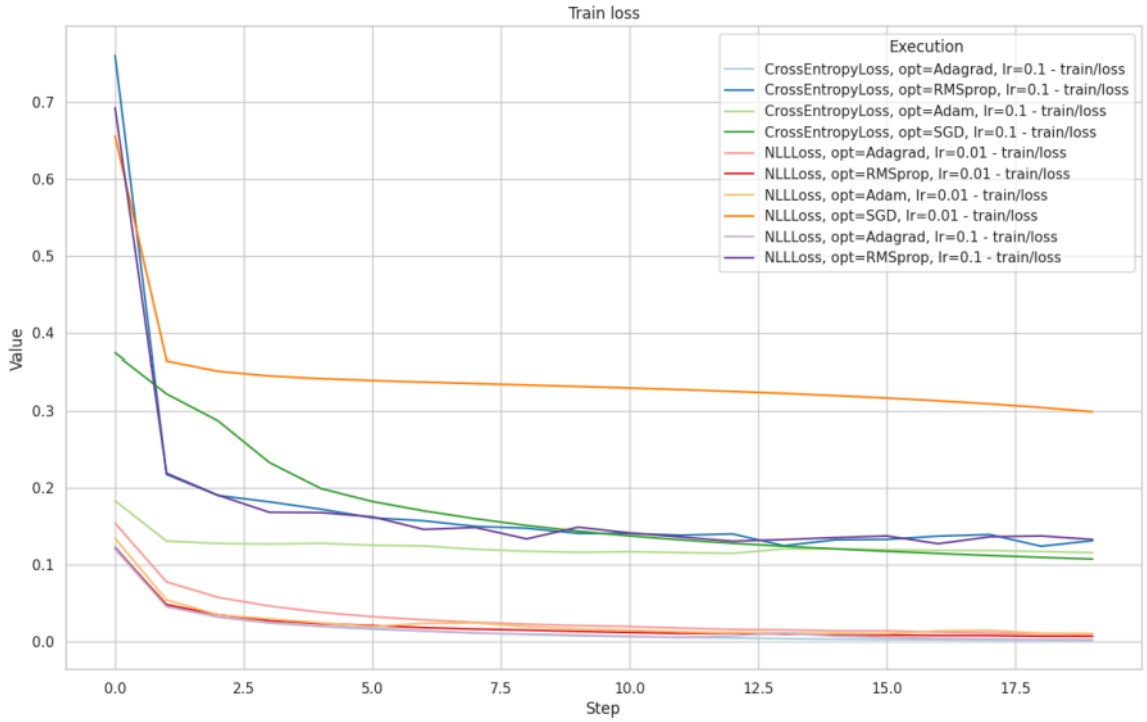


Figure 42: Train loss for different hyperparameter's combinations

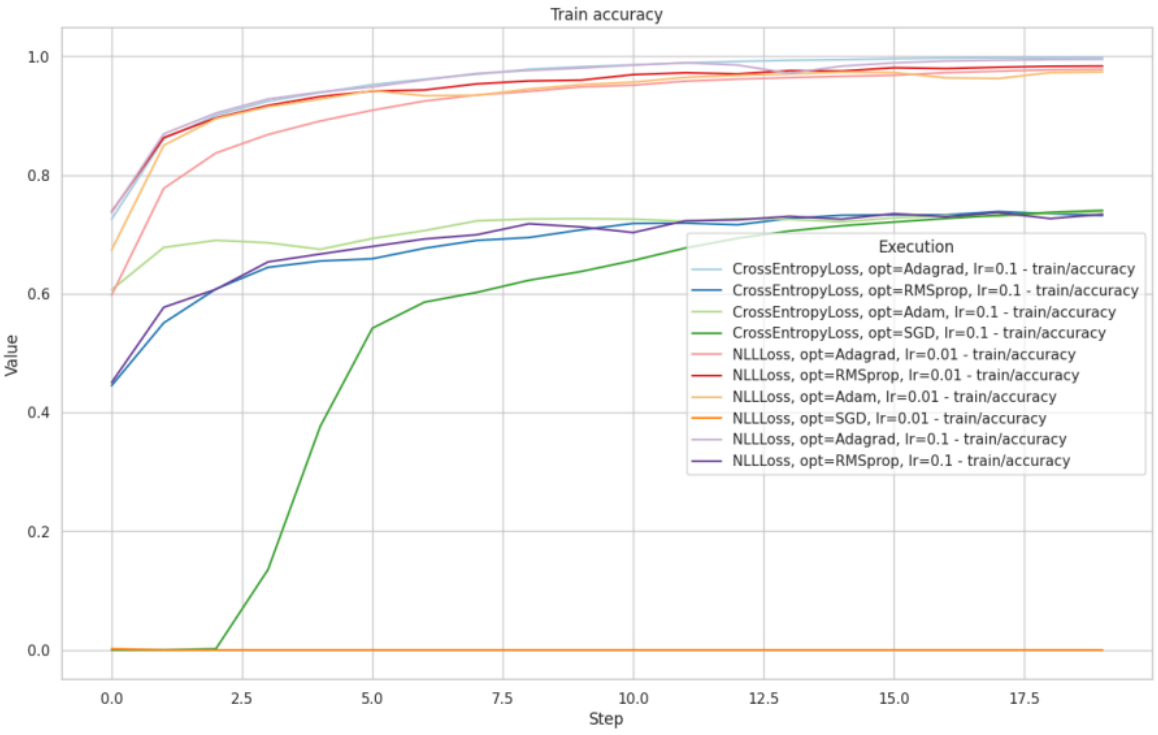


Figure 43: Train accuracy for different hyperparameter's combinations

Table 26: Hyperparameters evaluation (from highest to lowest)

Loss Function	Optimizer	Learning Rate	Final Loss	Final Accuracy
CrossEntropyLoss	Adagrad	0.1	0.0009	0.9979
NLLLoss	Adagrad	0.1	0.0021	0.995
NLLLoss	RMSprop	0.01	0.0064	0.9836
NLLLoss	Adagrad	0.01	0.0094	0.9781
NLLLoss	Adam	0.01	0.0096	0.9737
CrossEntropyLoss	SGD	0.1	0.1068	0.7404
CrossEntropyLoss	Adam	0.1	0.1152	0.7384
NLLLoss	RMSprop	0.1	0.1308	0.7339
CrossEntropyLoss	RMSprop	0.1	0.1324	0.7384
NLLLoss	SGD	0.01	0.2984	0

There are some events that can happen with the model and its hyperparameters:

- Overfitting: when a model learns the training data too well. It can be detected if the training accuracy is very high and the validation accuracy significantly lower.

As we do not have the validation plots of the different combinations of hyperparameters, we cannot see directly which of them overfit. However, models that converge quickly and reach almost perfect accuracy might be overfitting, mainly if the training process continues with more epochs without seeing the validation performance.

- Overshooting: when the learning rate is too high, which causes the model to jump over the optimal solution (usually if the model has a high learning rate). Detected by fluctuations or diverging losses.

With the graphics we can say that CrossEntropyLoss with Adam and lr=0.1 shows initial fluctuations and instability in the training loss plot, which could potentially be overshooting due to a high learning rate. Also the NLLLoss with Adam and lr=0.01 in addition shows some initial fluctuations which are also a sign of overshooting, but it stabilizes over time.

- Underfitting: when a model is too simple to capture the underlying patterns in the data, which result in a poor performance on training and validation data. It's detected if the model has low training accuracy, or if the model converges too slowly or fails to reduce the training loss significantly.

We can say that the combination of hyperparameters CrossEntropyLoss with SGD and lr=0.1 could present underfitting, due to its showing slower convergence (compared to the others combinations).

- Convergence: process of the training loss steadily decreasing and training accuracy increasing towards an optimal value.

Given the plots of the Weights and Bias webpage, there are multiple combinations of hyperparameters that seem to converge, for example CrossEntropyLoss with Adagrad and lr=0.1 or NLLLoss with Adagrad and lr=0.01, among others.

In summary, the final hyperparameters chosen to obtain the final model results are: Loss function: Cross Entropy Loss. Optimizer: Adagrad Learning rate: 0.1 However, we could also have chosen any of the top 5 best hyperparameters with the best accuracies and obtained very good results.

5.3 Results:

5.3.1 LSTM with Baseline Tags

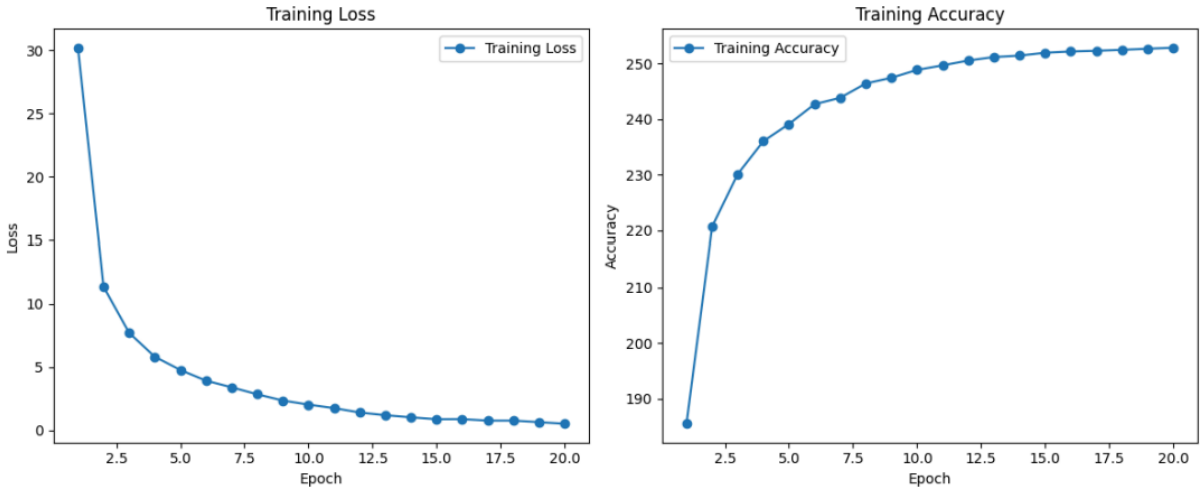


Figure 44: Curve of loss in Baseline tagging

Table 27: LSTM Baseline train set evaluation

	precision	recall	f1-score	support
NEG	0.9947	0.9977	0.9962	4517
NSCO	0.9612	0.9991	0.9798	13981
O	0.9999	0.9974	0.9986	224531
UNC	0.99912	0.9883	0.9897	684
USCO	0.9990	0.9956	0.9973	2074
accuracy	0.9974			
macro avg	0.9892	0.9956	0.9923	245787
weighted avg	0.9975	0.9974	0.9975	245787

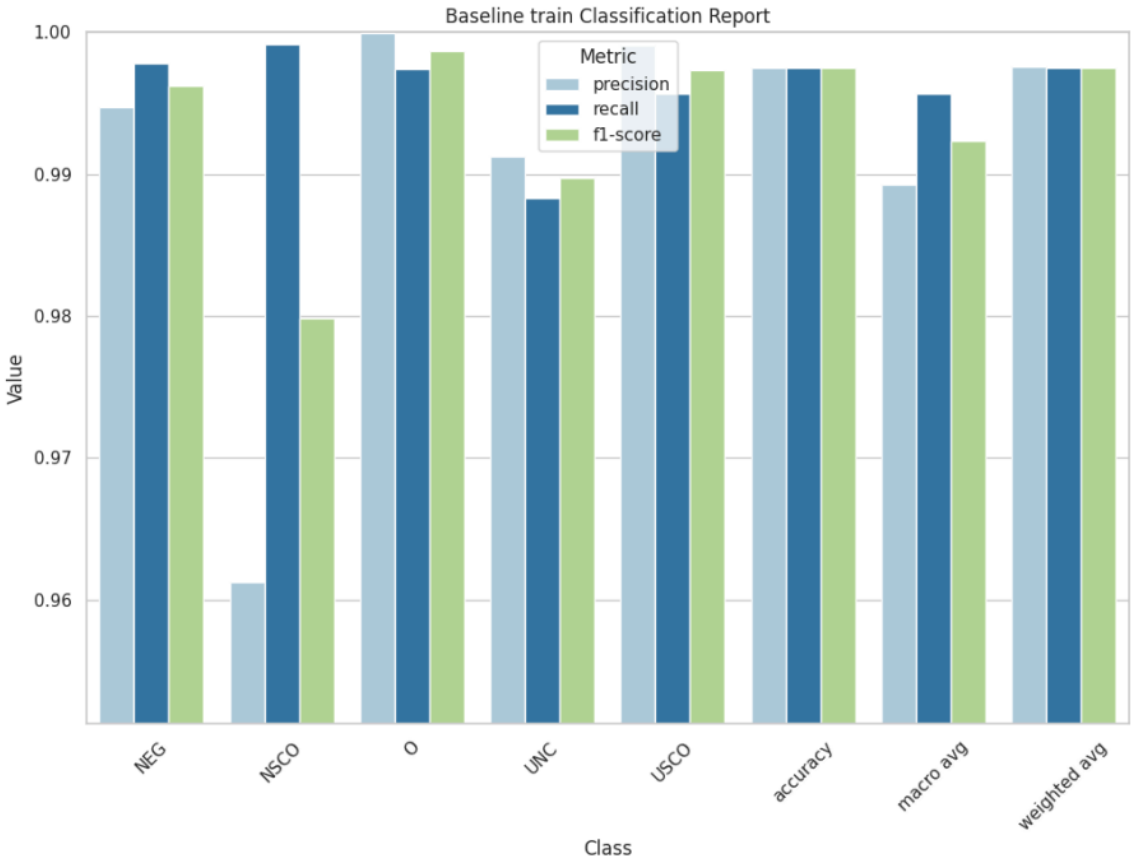


Figure 45: Report Baseline tagging model train with best hyperparameters

Table 28: LSTM Baseline test set evaluation

	precision	**recall**	**f1-score**	**support**
NEG	0.9325	0.9483	0.9403	1180
NSCO	0.7754	0.8910	0.8292	3570
O	0.9905	0.9816	0.9860	58388
UNC	0.8010	0.745	0.7720	200
USCO	0.7859	0.7707	0.7782	567
accuracy	0.9733			
macro avg	0.8571	0.8673	0.8611	63905
weighted avg	0.9750	0.9733	0.9739	63905

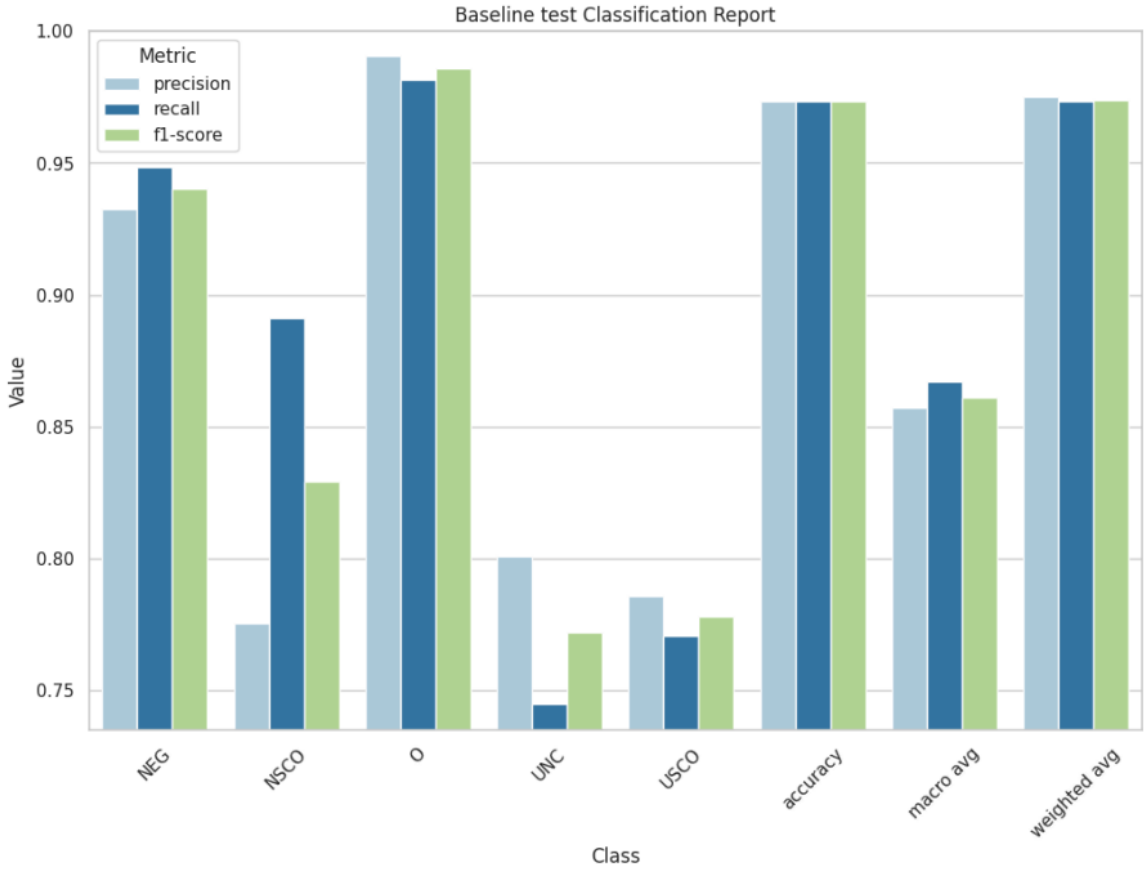


Figure 46: Report Baseline tagging model test with best hyperparameters

5.3.2 LSTM with BIO Tags

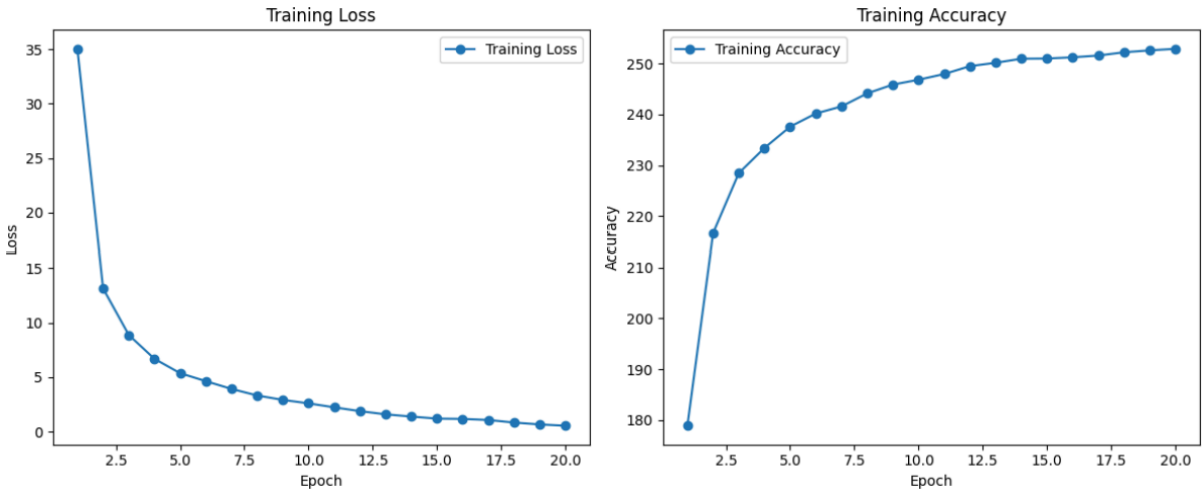


Figure 47: Curve of loss in BIO tagging

Table 29: LSTM BIO train set evaluation

	precision	recall	f1-score	support
B_NEG	0.9934	0.9995	0.9964	4256
B_NSCO	0.9960	0.9982	0.9971	4070
B_UNC	1	0.9649	0.9821	457
B_USCO	0.9955	0.9933	0.9844	450
I_NEG	0.9178	0.9846	0.9500	261
I_NSCO	0.9855	0.9984	0.9919	9911
I_UNC	1	0.9911	0.9955	227
I_USCO	0.9975	1	0.9987	1164
O	0.9999	0.9991	0.9995	224531
accuracy	0.9990			
macro avg	0.9873	0.9921	0.9895	63095
weighted avg	0.9990	0.9990	0.9990	63095

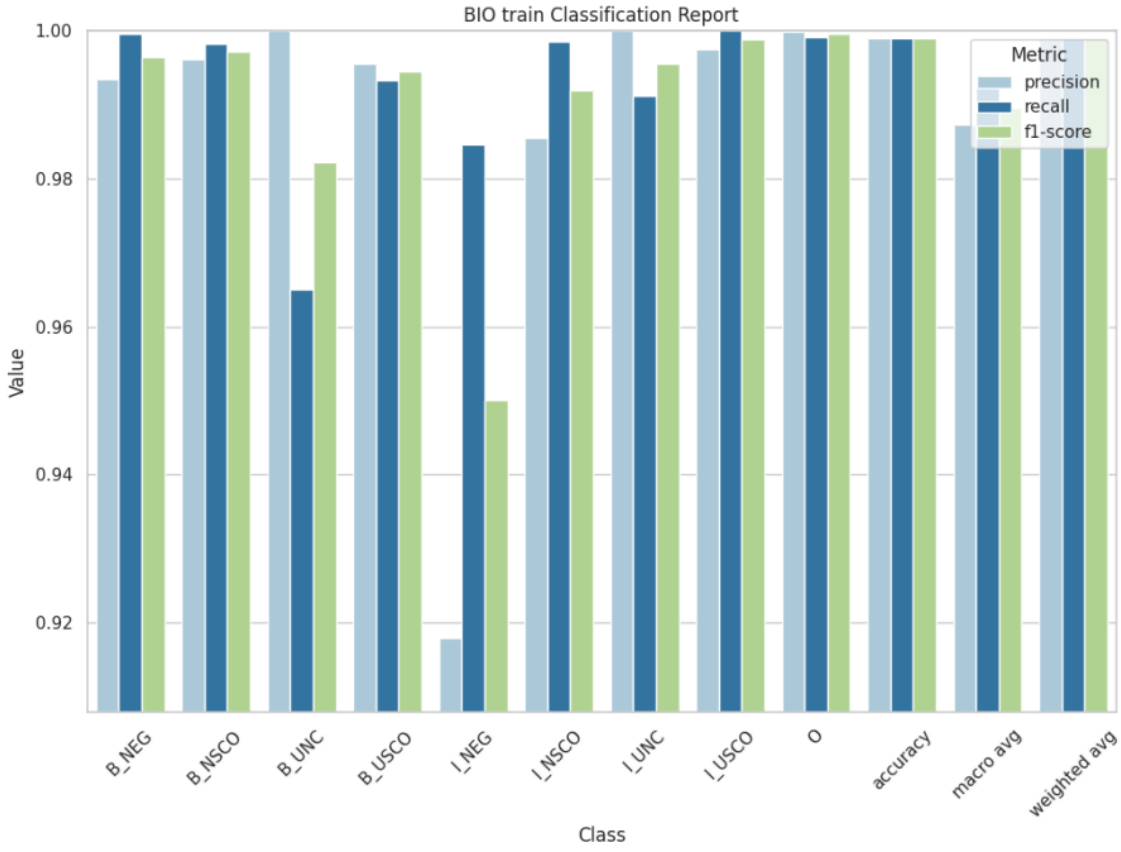


Figure 48: Report BIO tagging model train with best hyperparameters

Table 30: LSTM BIO test set evaluation

	precision	recall	f1-score	support
B_NEG	0.9573	0.9659	0.9616	1116
B_NSCO	0.9090	0.9107	0.9099	1065
B_UNC	0.8083	0.7348	0.7698	132
B_USCO	0.9285	0.8062	0.8630	129
I_NEG	0.4941	0.6562	0.5637	64
I_NSCO	0.7651	0.8495	0.8051	2505
I_UNC	0.8620	0.7352	0.7936	68
I_USCO	0.7380	0.7397	0.7388	438
O	0.9894	0.9848	0.9871	58388
accuracy	0.9748			
macro avg	0.8280	0.8203	0.8214	63095
weighted avg	0.9758	0.9748	0.9752	63095

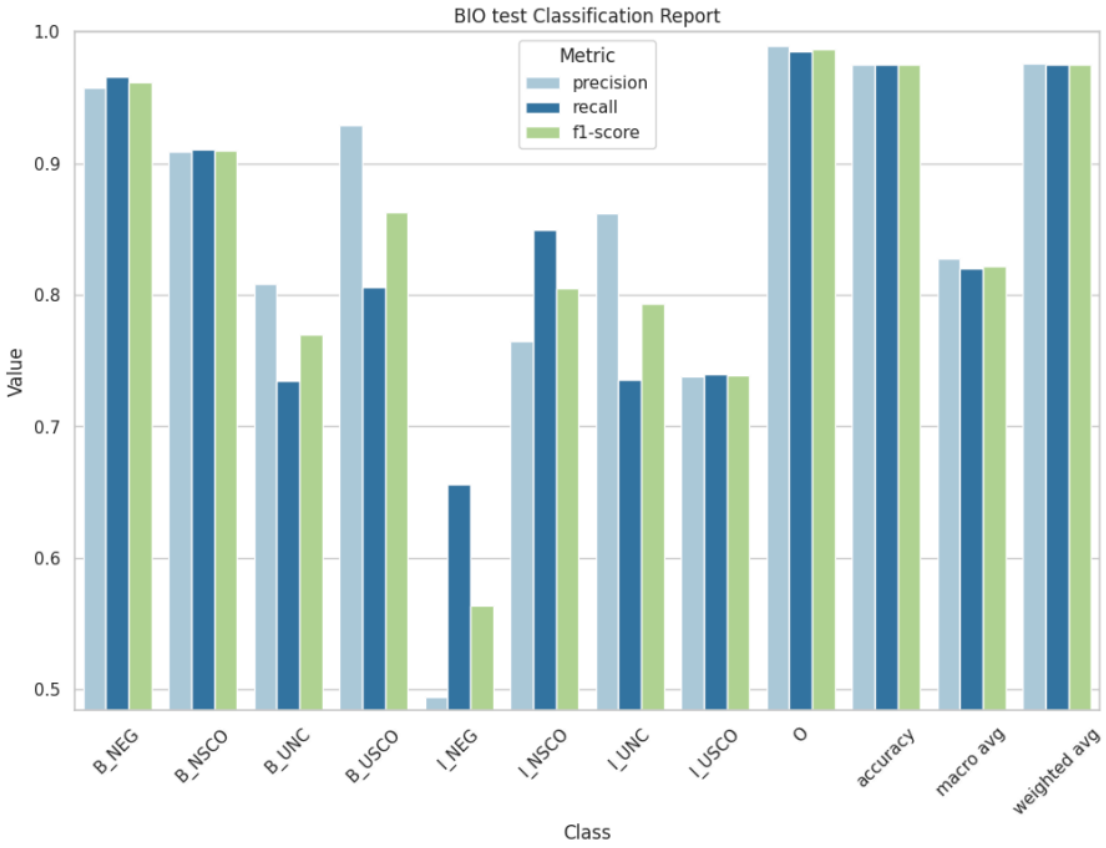


Figure 49: Report BIO tagging model test with best hyperparameters

5.3.3 LSTM with BIESO Tags

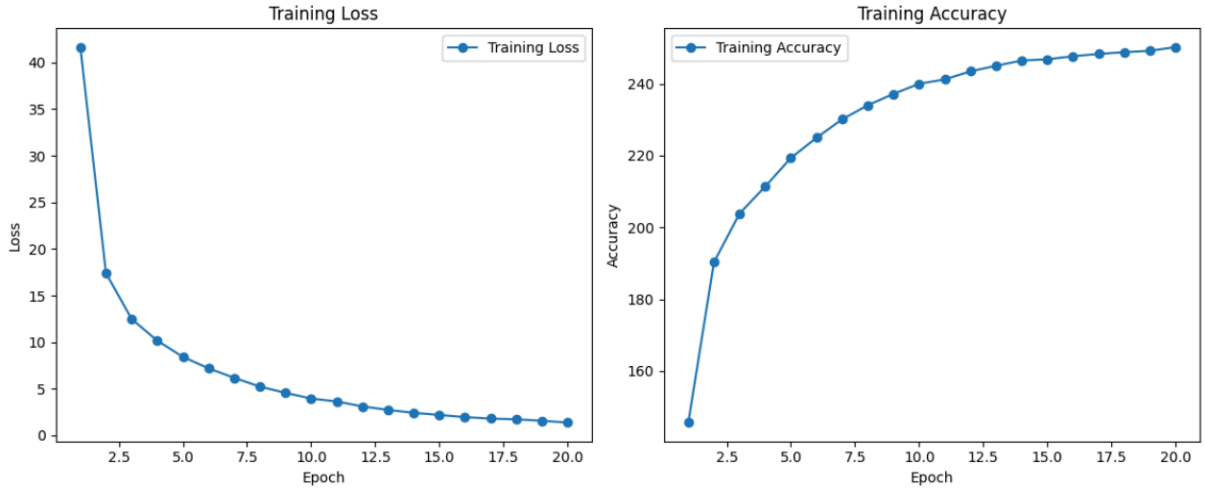


Figure 50: Curve of loss in BIESO tagging

Table 31: LSTM BIESO train set evaluation

	precision	recall	f1-score	support
<OOV>	0.9573	0.9610	0.9592	257
B_NSCO	0.9791	0.9937	0.9863	3023
B_UNC	0.9634	0.9547	0.9590	221
B_USCO	0.9974	0.9850	0.9912	402
E_NEG	0.9654	0.9844	0.9749	257
E_NSCO	0.9121	0.9209	0.9165	3023
E_UNC	0.9954	0.9864	0.9909	221
E_USCO	0.9954	0.98	0.9876	450
I_NEG	1	0.5	0.6666	4
I_NSCO	0.9636	0.9923	0.9777	6888
I_UNC	1	0.5	0.6666	6
I_USCO	0.9934	0.9950	0.9942	1222
O	0.9997	0.9950	0.9992	224531
S_NEG	0.9935	0.9969	0.9952	3999
S_NSCO	0.9813	0.9532	0.9670	1047
S_UNC	0.9911	0.9533	0.9719	236
accuracy	0.9970			
macro avg	0.9805	0.9160	0.9378	245787
weighted avg	0.9971	0.9970	0.9970	245787

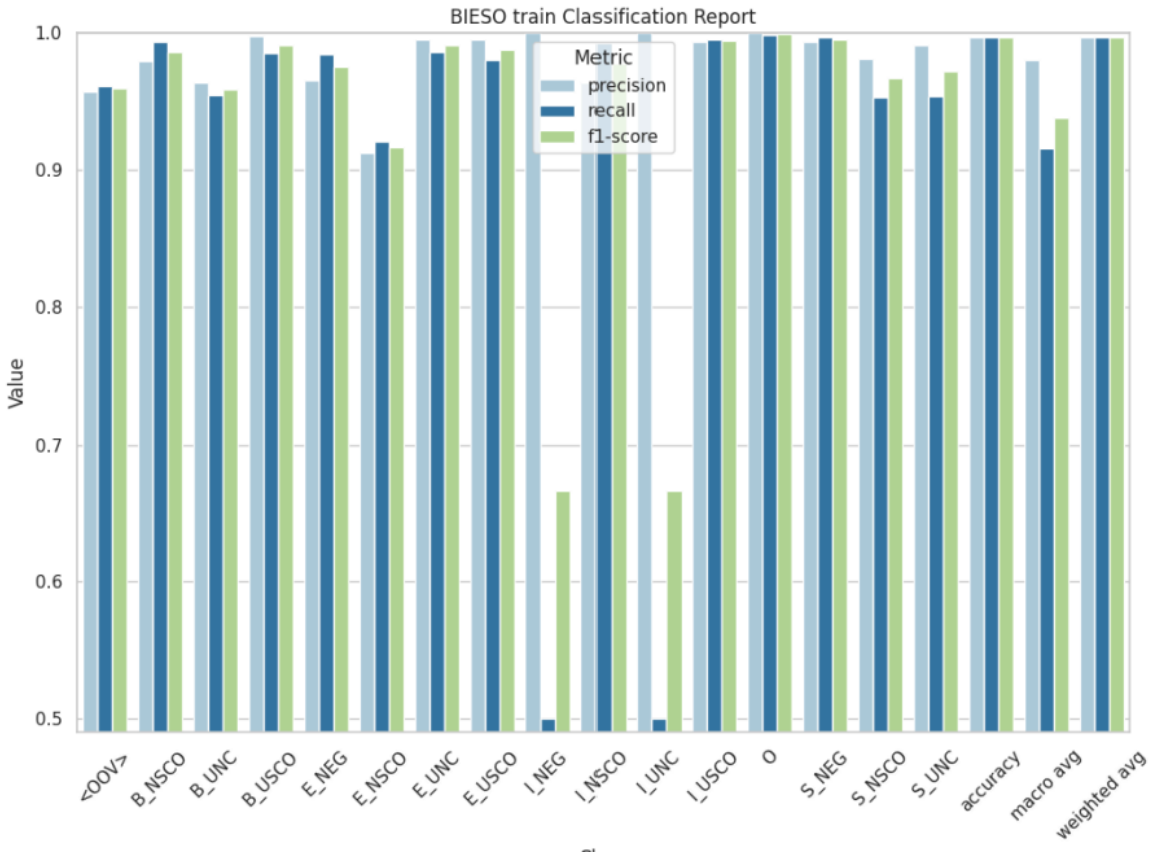


Figure 51: Report BIESO tagging model train with best hyperparameters

Table 32: LSTM BIESO test set evaluation

	precision	**recall**	**f1-score**	**support**
<OOV>	0.5479	0.6451	0.5925	62
B_NSCO	0.7542	0.8517	0.8	789
B_UNC	0.7118	0.6562	0.6829	64
B_USCO	0.84	0.7304	0.7813	115
E_NEG	0.56	0.6774	0.6131	62
E_NSCO	0.3802	0.4245	0.4011	789
E_UNC	0.8524	0.8125	0.832	64
E_USCO	0.4444	0.3720	0.4050	129
I_NEG	1	0	0	2
I_NSCO	0.6845	0.7867	0.7321	1716
I_UNC	1	0.25	0.4	4
I_USCO	0.6532	0.6532	0.6532	323
O	0.9883	0.9827	0.9855	58388
S_NEG	0.9481	0.9373	0.9427	1054
S_NSCO	0.5879	0.4239	0.4926	276
S_UNC	0.9193	0.8382	0.8769	68
accuracy	0.9611			
macro avg	0.7420	0.6276	0.6369	63905
weighted avg	0.9630	0.9611	0.9618	63905

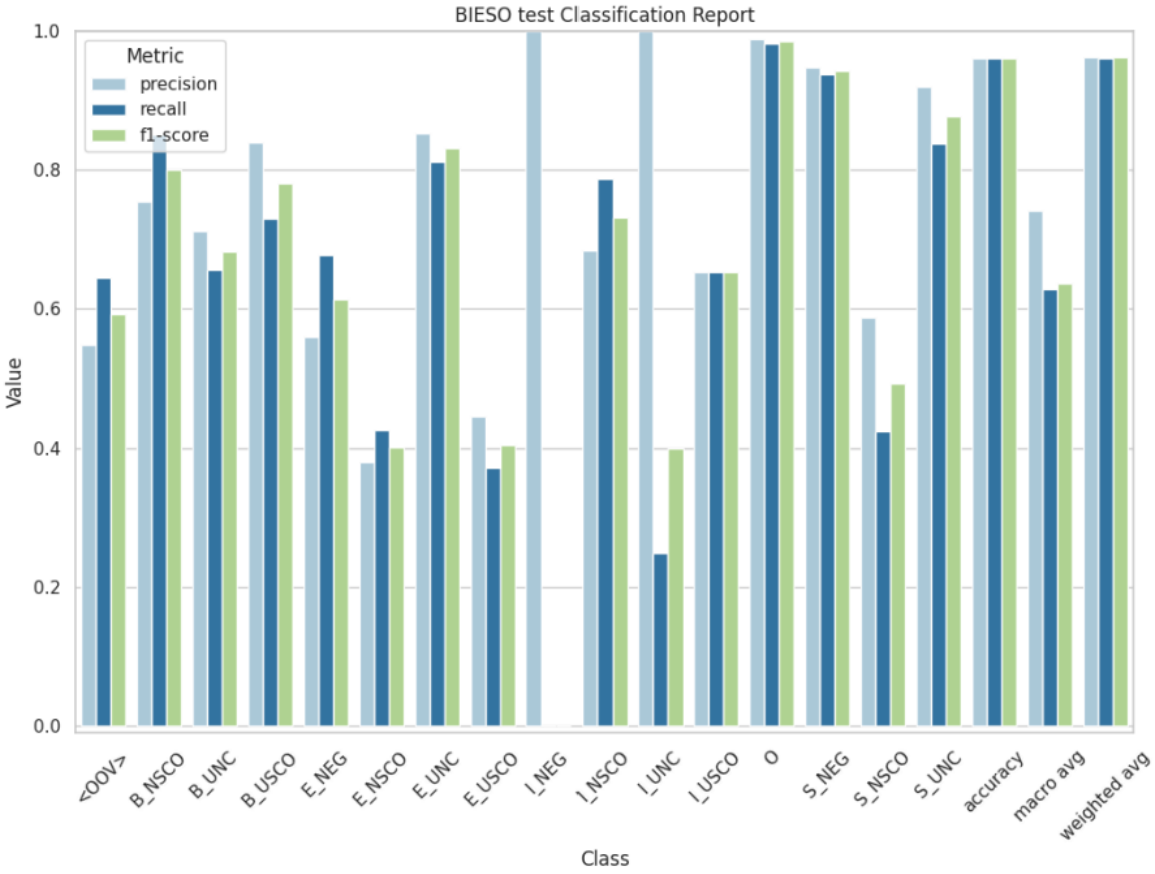


Figure 52: Report BIESO tagging model test with best hyperparameters

5.4 Problems:

- Defining the architecture
- Finding the optimal hyperparameters
- Computational resources

5.5 Conclusions on Deep-Learning Approach:

At the beginning, after all the research about deep learning implementation in NLP, we have adapted our datasets. We have created some tagging algorithms which take from the simplest and basic ones to use beginning and end tags of the labeled word. We decided to do the LSTM algorithm, which was the one more recurrently mentioned in our research.

One important remark is that for the training process of the three different approaches of LSTM (Baseline, BIO and BIESO) the train accuracy was above 99%.

The first implementation of the LSTM consisted of using the baseline tags. We obtained an accuracy of 97,33% for the test data, which is very high. In general, all the metrics are above 0.80, except UNC and USCO because the data does not contain a lot of instances of them.

Regarding the LSTM with BIO tagging, the accuracy obtained was 97,48%, which is higher than the Baseline. The variance of the metrics vary quite a lot (maximum 98.71

and minimum of 56.37). The lower ones are because there are not enough examples of that tag in the data, which means that if the model does not classify well one instance, the metric result varies significantly.

Finally, talking about the LSTM model with BIESO tagging, the accuracy obtained was 96,11%. As this approach of tagging has many more tags than the BIO one, there are some tags with practically no instance of them, for example Inside Negation or Inside Uncertainty. This leads to the same problem as with BIO tagging; if one of the tags with not much more examples has been misclassified, the error and final metrics vary a lot.

For knowing which of the three approaches of LSTM has better results, we would take into account the accuracy, the metrics and the instances of each tag. The BIESO approach is the one with worse results in accuracy, however we think that if the dataset would have more instances of the less tags, the model would learn better and would lead this model to be the best approach of LSTM. Now, regarding BIO and Baseline, both of them have similar results and metrics, however we think that the best model is the BIO one, as it classifies with more precision and variety the data than the Baseline model. But, it is remarkable to say the problem of having a few instances of some tags significantly affects the performance of this model. That's why this approach is the best one even if the Baseline model is the one with more flexibility.

The structure, as we said before, was simple as we do not have the time and/or resources to implement a more complex one. So, we think that if we had them the results obtained for all the three models would improve.

6. IMPROVEMENTS FOR THE FUTURE:

- Combining methods: e.g. LSTM + CRF
- More ambitious implementations: Transformers and BERT (Bidirectional Encoder Representations from Transformers). They are able to handle parallelization and capturing very long-range dependencies.

7. GENERAL CONCLUSIONS:

For the final conclusion of the project, we are going to tackle each approach individually and then express our opinion of which is the best approach for detecting negation and uncertainty in medical text based on the results we obtained.

Firstly, we had done the rule-based approach. For this, we had done three different methods, NER implementation, our own model from scratch (using regex) and dependency parsing. The one with better results was the model we created. The reason for that is that it is mostly based on the dataset we had, detecting in a range of 5 words for each size of the negation/uncertainty word found. However, if there is a word that the model has not previously seen it won't detect it, so it is very specific (for words in catalan and

spanish). Furthermore, the model could be improved with more vocabulary or adding different sentence structure so it would detect the scopes better.

Secondly, the Machine Learning approach. For this model we have done different algorithms to see which of them performs better and we have combined the model with some tagging algorithms, BIO and BIESO. Among all, the best one, or, in this case, the best ones, are CRF, SVM and LR (with dict vector). They perform very well, with all the metrics above 94%.

Finally, the Deep Learning approach. For this model we have done a type of Recurrent Neural Network, an LSTM (Long-Short Term Memory) with BIO and BIESO tagging, apart from the Baseline model. Comparing these three approaches, the best one was the BIO model. It gives quite good results with an accuracy of 97.48%. However, we think that with more time and resources and if the dataset could have more instances of the less tags, the performance of this model (and the BIESO one) would be much better.

After doing three different approaches for detecting negation and uncertainty in medical text, we can conclude that, given the limited time and resources we had for doing this project, we end up with the machine learning approach that gives us the best results for this task. However, we think that the deep learning one would be the best one, but the resources we had and/or the structure of our model (even if we try to find the best hyperparameters) were not enough to back up our theory.

For a general conclusion of this project, we can say that we have learned overall how to put the theory into practice. Leading to a more comprehensive and reasoning of the lectures provided by this subject, as we have gone from the initial steps, regex, lemmatization...; to the most innovative and complex ones, deep learning approaches which are composed by LSTM, BERT... among others.

8. REFERENCES:

8.1 Rule-based

1. Original NegEx algorithm paper. Chapman, W. W., Bridewell, W., Hanbury, P., Cooper, G. F., & Buchanan, B. G. (2002). A Simple Algorithm for Identifying Negated Findings and Diseases in Discharge Summaries. Journal of Biomedical Informatics, 35(5-6), 305-312.

2. NegEx adaptation to Spanish paper. Costumero, R., Lopez, F., Gonzalo-Martín, C., Millan, M., & Menasalvas, E. (2014). An Approach to Detect Negation on Medical Documents in Spanish. In Proceedings of the International Conference on Health Informatics (pp. 123-135). Madrid, Spain.

3. Original NegEx algorithm authors expand NegEx to other languages. Chapman, W. W., Hilert, D., Velupillai, S., Kvist, M., Skeppstedt, M., Chapman, B. E., Conway,

M., Tharp, M., Mowery, D. L., & Deleger, L. (2013). Extending the NegEx Lexicon for Multiple Languages. Journal of Biomedical Informatics, 45(5), 973-981.

4. Original NegEx algorithm. Chapman, B. E. (2009). negex. GitHub.

5. Spanish datasets for negations, uncertainties and conjunctions. PlanTL-GOB-ES. (2018). NegEx-MES. GitHub.

6. Uncertainties dataset. OpenAI. (2024, April 25). Conversation about obtaining uncertainty words in Spanish and Catalan [ChatGPT prompt]. OpenAI.

7. Centro de Terminología TERMCAT. (2020). Medical Terminologies. ELRC-SHARE Repository. European Language Resource Coordination.

8.2 Machine-learning

1. Visited website for implementing SVM and Naive Bayes in NLP. Bedi, Gunjit. (2018). SVM and Naïve Bayes

2. NegEx adaptation to Spanish paper. Costumero, R., Lopez, F., Gonzalo-Martín, C., Millan, M., & Menasalvas, E. (2014). An Approach to Detect Negation on Medical Documents in Spanish. In Proceedings of the International Conference on Health Informatics (pp. 123-135). Madrid, Spain.

3. Original SVM algorithm. Cico, David. (2020). SVM. GitHub.

4. Original Logistic Regression and Naïve Bayes algorithm. Akhiiksare. (2020). Logistic Regression and Naïve Bayes. GitHub.

5. Original CRF algorithm. Ruthu S Sanketh. (2020). CRF.

6. Original HMM algorithm. (2022). HMM.

8.3 Deep-learning

1. Basic knowledge dor sequence modelsand LSTM LSTM.Pytorch Official Web.

2. Original LSTM algorithm in keras. Pavez, Juan. (2016). LSTM. Github.

3. More information research in LSTM LSTM. Community Notebook.

4. Original LSTM algorithm in Medium webpage. Loem, Mengsay. (2021) LSTM.Medium.