UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Bachelor's degree in Data Science and Engineering

Spoken and Written Language Processing - 270225

Assignment 3 - Language Identification (Sentence Classification)

Sílvia Fàbregas
Eduard Morillo

Professors:
José Adrián Rodríguez
Carlos Escolano

May 23, 2024

Contents

Motivation

Language identification (LI) is a fundamental task in natural language processing with wide-ranging applications from machine translation to sentiment analysis. The motivation behind language identification systems lies in the need to process and understand the vast amount of multilingual content available. In this assignment, various LI models will be trained using Wikipedia paragraphs as data from 235 different languages.

A character-based RNN baseline is provided to begin the assignment. Modifications are introduced and new models are tested in order to conduct a comparative study for the proposed task. These models are described in the following section, together with the code sections that implement them.

Proposed Models

Deep learning approaches

The baseline model is a character classifier, composed of an embedding layer which converts input indices into dense vectors of fixed size (`embedding_size`), followed by an Recurrent Neural Network layer, which depends on the specified model, and, finally, a linear layer that transforms the last hidden state from the RNN to the final output. A schematic of this architecture is to be found below, specifying the dimensions of the input and output vectors and of every layer. T is the maximum number of toquens in a sequence, E is the embedding size, H is the hidden size of the recurrent layer and $O = 235$ is the number of languages, so that $O(i) = P(\text{sentence written in } i \text{ language})$.

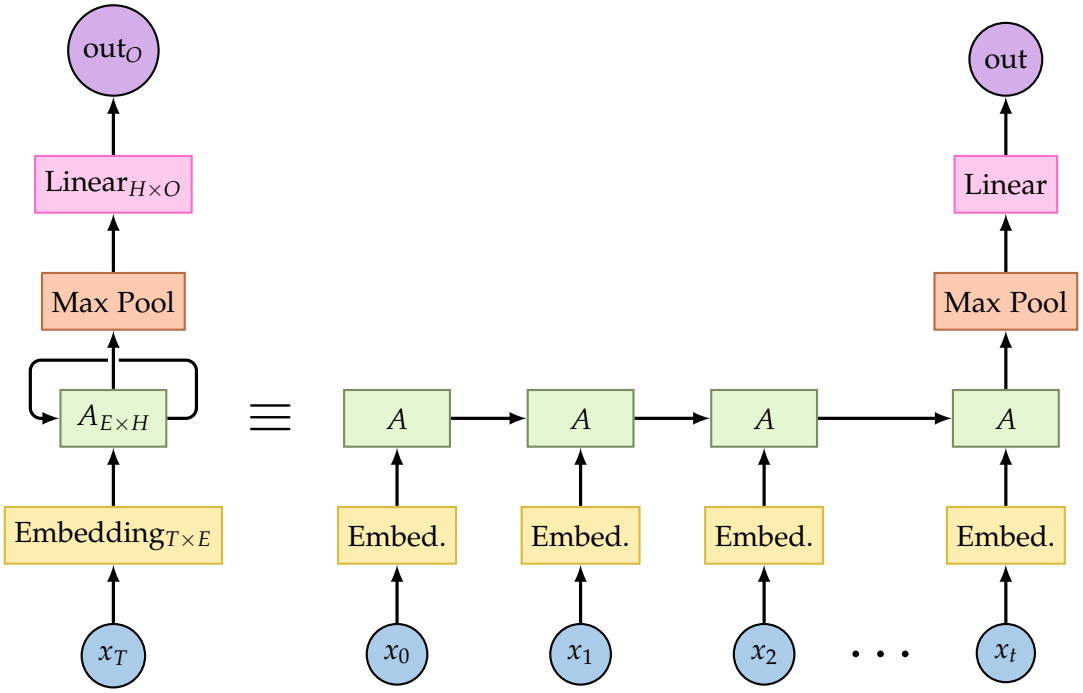

Figure 1: Block diagram of the baseline model.

LSTM

When `model='LSTM'`, block A in figure 1 becomes an LSTM. LSTM, or Long Short-Term Memory, is a type of recurrent neural network designed to tackle the limitations of traditional RNNs, particularly in handling long-term dependencies in sequential data. It was first introduced in [1].

LSTM improves upon vanilla RNNs by introducing a more complex memory cell structure, allowing it to selectively remember or forget information over arbitrary time intervals. This addresses the vanishing/exploding gradient problem, where gradients either become extremely small or large over many time steps, hindering the model's ability to capture long-term dependencies. Let ∇L denote the gradient of the loss function L with respect to the parameters of the network. During backpropagation, the gradients are computed recursively using the chain rule:

$$\nabla L = \frac{\partial L}{\partial w_L} \frac{\partial w_L}{\partial w_{L-1}} \cdots \frac{\partial w_2}{\partial w_1}$$

where w_i represents the parameters of layer i in the network. If the absolute values of the partial derivatives $\frac{\partial w_i}{\partial w_{i-1}}$ are less than 1 for most layers, or greater than 1 for most layers, then the gradients will diminish or grow exponentially as they are propagated backward through the layers, leading to the vanishing/exploding gradient problem.

At its core, an LSTM unit consists of three main components: the input gate, the forget gate, and the output gate. These gates regulate the flow of information through the cell, determining what information to discard, what new information to store, and what information to output based on the current cell state (see figure 2).

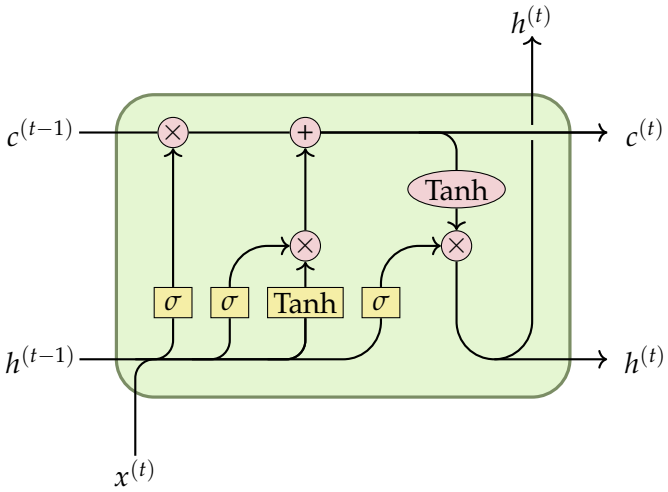

Figure 2: Architecture of an LSTM cell

Behind the scenes, an LSTM cell maintains a cell state $c^{(t)}$ and a hidden state $h^{(t)}$, which are updated at each time step based on the input $x^{(t)}$ and the previous states $c^{(t-1)}$ and $h^{(t-1)}$, as well as the outputs of the gate functions.

Forget gate $f^{(t)}$: Decides what information to discard from the cell state.

$$f^{(t)} = \sigma(W_f \cdot [x^{(t)}; h^{(t-1)}] + b_f)$$

Input gate $i^{(t)}$: Determines what new information to store in the cell state.

$$i^{(t)} = \sigma(W_i \cdot [x^{(t)}; h^{(t-1)}] + b_i)$$

Output gate $o^{(t)}$: Controls what information to output based on the current cell state.

$$o^{(t)} = \sigma(W_o \cdot [x^{(t)}; h^{(t-1)}] + b_o)$$

Cell State update:

$$\tilde{c}^{(t)} = \tanh(W_c \cdot [h^{(t-1)}; x^{(t)}] + b_c)$$
$$c^{(t)} = f_t \cdot c^{(t-1)} + i^{(t)} \cdot \tilde{c}^{(t)}$$

Hidden State update:

$$h^{(t)} = o^{(t)} \cdot \tanh(c^{(t)})$$

The total number of parameters in an LSTM unit is

$$4(\texttt{input\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size})$$

and it scales with the number of LSTM units in the network.

GRU

When `model='GRU'`, block A in figure 1 becomes a GRU. The GRU is a variant of the traditional RNN architecture, like LSTM, designed to improve upon the limitations of vanilla RNNs [2].

GRU consist of two main gates: the update gate and the reset gate. These gates determine how much of the past information to retain and how much new information to incorporate into the current state (see figure 3).

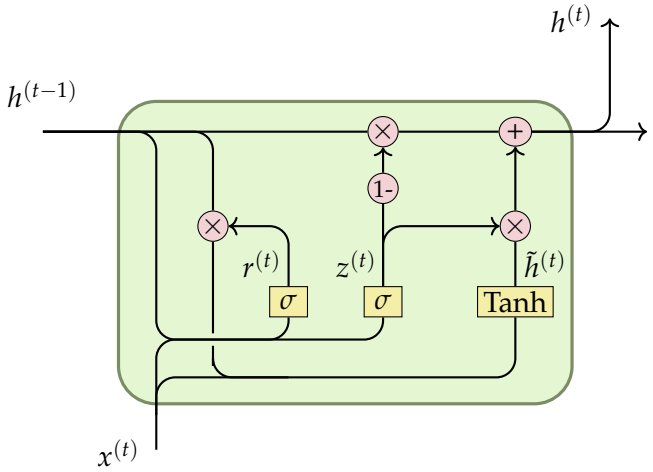

Figure 3: Architecture of a GRU cell

The update gate $z^{(t)}$ and reset gate $r^{(t)}$ are computed using sigmoid activation functions, and the new candidate state $\tilde{h}^{(t)}$ is computed using a hyperbolic tangent function.

$$z^{(t)} = \sigma(W_z \cdot [x^{(t)}; h^{(t-1)}])$$
$$r^{(t)} = \sigma(W_r \cdot [x^{(t)}; h^{(t-1)}])$$
$$\tilde{h}^{(t)} = \tanh(W_h \cdot [x^{(t)}; r^{(t)} \odot h^{(t-1)}])$$

Here, W_z, W_r, and W_h are weight matrices, $h^{(t-1)}$ is the previous hidden state, $x^{(t)}$ is the current input, and $\odot$ represents element-wise multiplication.
The update gate is then used to blend the previous hidden state h_{t-1} with the new candidate state $\tilde{h}^{(t)}$ to produce the current hidden state $h^{(t)}$:

$$h^{(t)} = (1 - z^{(t)}) \odot h^{(t-1)} + z^{(t)} \odot \tilde{h}^{(t)}$$

GRUs yield similar performance as LSTMs, with one gate less. Therefore, one GRU has

$$3(\texttt{input\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size})$$

parameters, that of course scale with the number of units in the network.

BERT

Pretrained models such as BERT have been assessed, fine-tuning the general model to adequately confront the required task. BERT (Bidirectional Encoder Representations from Transformers) is a state-of-the-art pre-trained language representation model developed by Google. It's based on the Transformer architecture and has achieved remarkable performance on a wide range of natural language processing (NLP) tasks.

Machine learning approaches

In addition to exploring various input features, we also evaluated classical classification algorithms, including Random Forests, Naive Bayes and Support Vector Machines (SVMs). Notably, only SVMs have been incorporated into this report due to their unparalleled performance. It's worth mentioning that for SVMs, the inputs consist of n-gram counts for $n = 3$ and $n = 4$.

Input data

The machine learning algorithms proposed have been inputed matrices of data built with count-based methods together with n-grams, counting how many times each n-gram appeared in every sentence of the data. For the sentences below and $n = 3$:

1. `"The cat sat on the mat."`
2. `"The dog ate my homework."`
3. `"I love cats and dogs."`

The following matrix would be inputed to the model.

Index	the cat sat	cat sat on	sat on the	on the mat	the dog ate	dog ate my	...
0	1	1	1	1	0	0	
1	1	0	0	0	1	1	
2	0	0	0	0	0	0	

In Python this is done with the function `CountVectorizer`, as follows.

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import Normalizer

vectorizer = CountVectorizer(ngram_range=(3,4), analyzer="char")
X = vectorizer.fit_transform(x_train)
X_val = vectorizer.transform(x_val)

# scale features
normalizer = Normalizer()

# Fit and transform the training data
X_train = normalizer.fit_transform(X)

# Transform the testing data
X_val = normalizer.transform(X_val)
```

SVM

Support Vector Machines or SVMs are a type of supervised learning algorithm. In the context of classification, SVMs find the hyperplane that best separates different classes of data points. The hyperplane is chosen in such a way that it maximizes the margin between the classes, where the margin is the distance between the hyperplane and the nearest data point of each class, called a support vector. This maximization of margin leads to better generalization and robustness of the classifier.

SVMs is particularly well-suited for binary classification problems but can also be extended to handle multi-class classification tasks through various strategies like one-vs-one or one-vs-all approaches. In our dataset, we'll be classifiying sentences in 235 languages, following a one-vs-all approach for the problem to be tractable.

Mathematically, given a training dataset $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i represents the feature vector and y_i represents the class label ($y_i \in \{-1, 1\}$ for binary classification), SVM seeks to solve the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i$$

subject to:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0, \ \forall i$$

Here, $\mathbf{w}$ is the weight vector perpendicular to the hyperplane, b is the bias term, ξ_i are slack variables that allow for misclassification, and C is a regularization parameter that balances the margin maximization and the classification error.

The parameter C controls the trade-off between allowing training errors and forcing rigid margins. A smaller C encourages a larger margin and allows more training errors, potentially leading to better generalization on unseen data but might underfit the training data. Conversely, a larger C penalizes errors more heavily, leading to narrower margins but potentially better fitting the training data, albeit with a risk of overfitting.

```
from sklearn.svm import LinearSVC

# define parameter grid
param_grid = {
    "C": [0.1, 1, 10, 100],  # Regularization parameter
    "loss": ["hinge", "squared_hinge"]  # Loss function
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(LinearSVC(max_iter=10000), param_grid, \
            cv=KFold(n_splits=3), verbose=3)

grid_search.fit(X_train, y_train)
```

The best parameters turned out to be `{'C': 1, 'loss': 'squared_hinge'}`, which are the default ones. Both the best Naive Bayes model and the best random forest model, scored 92% accuracy in the validation set.

Naïve Bayes

A Naive Bayes classifier, as its name announces, is based on Bayes' Theorem. Bayes' Theorem states that

$$Pr(A|B) = \frac{Pr(B|A)Pr(A)}{Pr(B)}$$

It has several assumptions:

- Independence of the different features of the model. Note: Since we will be using as input features the counts of various n-gram, we cannot assure here. For instance some languages employ different alphabets.
- Continuous features are normally distributed.
- Discrete features have multinomial distributions. Note: This could not be the case since our continuous features are counts and thus will probably follow Poissons.
- Features are equally important.

Modifications to the baseline model

Having explained the trained models, we shall now go back to the baseline model, which used a Recurrent Neural Network.

Starting from this initial model, we will optimize various hyperparameters in order to achieve superior performances, which include:

- Increasing the embedding size
- Increasing the RNN hidden size
- Increasing the batch size
- Trying out different optimizers: Adam, RMSProp, AdaDelta

Moreover, variations of the architecture were also trained, including:

- Combination of the output of the max-pool layer with the output of a mean-pool layer, addition

```
    def forward(self, input, input_lengths):
        """Forward method of the CharRNNClassifier class with added
        max + mean pooling layers."""

        encoded = self.embed(input)
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, \
        input_lengths)
        output, _ = self.rnn(packed)
        padded, _ = torch.nn.utils.rnn.pad_packed_sequence(output, \
        padding_value=float(0))
        output_1, _ = padded.max(dim=0)
        output_2 = torch.mean(padded, 0)
        output = output_1 + output_2
        output = self.h2o(output)
        return output
```

- Combination of the output of the max-pool layer with the output of a mean-pool layer, concatenation

```
        # In CharRNNClassifier __init__
        self.h2o = torch.nn.Linear(hidden_size*2, output_size)
```

```
        # In CharRNNClassifier forward
        output = torch.cat((output_1, output_2), dim=1)
```

- Addition of a dropout layer to avoid overfitting to the training data right before the linear layer

```
    # In CharRNNClassifier __init__
    def __init__(self, input_size, embedding_size, hidden_size, \
    output_size, dropout_p = 0.5, model="lstm", num_layers=1,\
    bidirectional=False, pad_idx=0):

        # (...)
        self.dropout = torch.nn.Dropout(dropout_p)
```

```
        # In CharRNNClassifier forward
        output = self.dropout(output)
```

- Concatenating two or more RNNs (see example with 2 RNNs in the code below)

```
    self.rnn1 = torch.nn.LSTM(embedding_size, hidden_size, num_layers, \
    bidirectional=bidirectional)
    self.rnn2 = torch.nn.LSTM(hidden_size, hidden_size, num_layers,\
    bidirectional=bidirectional)
```

Alternative architectures

Alternative architectures and formats for the input data have undergone testing. We shall enumerate and elaborate on the set of architectures that have been examined.

Rather than encoding and inputting individual characters, we employed character bigrams as input features. This approach aimed to capture a more comprehensive representation of the language sentences. For instance, many languages share alphabet and have similar character frequencies, for which it could be harder to deduce just from that information the language, whereas bigrams encode more information about the language and usual dispositions of characters.

The following code has been added to the baseline code.

```
class Dictionary_twogram(object):
    def __init__(self):
        self.twogram2idx = {}
        self.idx2twogram = []

    def add_twogram(self, twogram):
        if twogram not in self.twogram2idx:
            self.idx2twogram.append(twogram)
            self.twogram2idx[twogram] = len(self.idx2twogram) - 1
        return self.twogram2idx[twogram]

    def __len__(self):
        return len(self.idx2twogram)

```

```
char_vocab = Dictionary_twogram()
text_total = ' '.join(x_train_full)
for i in range(1, len(text_total) - 1):
    char_vocab.add_twogram(text_total[i-1: i+1])

x_train_idx = []
for line in x_train_full:
    one_line = []
    for i in range(1, len(line)-1):
        one_line.append(char_vocab.twogram2idx[line[i-1:i+1]])
    x_train_idx.append(np.asarray(one_line))

y_train_idx = np.asarray([lang_vocab.token2idx[l] for l in y_train_full])
```

Results

Table 1 (see next page) shows the comparison in accuracies of the variations of the first model. The test accuracies range from 92.9% to 94.5%, given a baseline of 93.5%. See also the IDs of the models.

ID	*Model description*	*num parameters*	*Validation accuracy*	*Test Accuracy*
1	Baseline	1081835	92.7%	93.5%
2	Embedding size x4	3353579	92.9%	93.1%
3	Embedding size x4 dropout	3353579	93.2%	93.6%
4	Hidden size x2	1996011	93.9%	94.5 %
5	Hidden size x2 dropout	1996011	93.8%	94.3%
6	Suma de poolings embedding size x3	2596331	92%	92.9%
7	Batch size x2	1081835	92.6%	93.1%
8	Concat de poolings	1141995	92.3%	92.9%
9	2 RNNs amb dropout	1700075	94.1%	94.5%
10	3 RNNs droptout Hidden Size x2	4851947	93.7%	93.9%
11	RMSProp	1081835	91%	91.6%
12	AdaDelta	1081835	92.6%	93.0%
13	GRU	999403	92.5%	93.2%
14	2-gram input	80869867	93.8%	93.3 %

Table 1: Accuracies obtained with variations of the baseline model.

Figure 4 shows the effect on the training and validation accuracies due to increasing the hidden state size, and how we can avoid severe overfitting to the training data by

introducing a dropout layer. The train accuracy rapidly increases when doubling the hidden state size, whereas the validation accuracy stays flat at barely 92%-93% accuracy. When adding a dropout layer, the validation accuracy stays the same but the training accuracy does not skyrocket. Probably, performing some more epochs would have allowed for a better fit.

This effect can also be noticed in Figure 5, which portrays the train and validation accuracy caused by multiplying by four the `embedding_size`, with and without dropout layer. In the former case, the test accuracy drops due to overfitting, whereas in the latter one, the accuracy remains the same. Thus, augmenting the embedding size does not seem to have an impact on the overall predictions.

In terms of accuracy, the two best models turned out to be the one with increased hidden state size and the one with 2 RNNs. Including more parameters tends to increase the accuracy, as long as we do not overfit to the training data and we allow the model to train for enough epochs. It must be noted that, in fact, after adding a third RNN to the architecture (model 10), the performance actually worsened.

Changing the optimizer, the model from LSTM to GRU (which has less parameters and thus is more quick to train but tends not to improve the accuracy), or combining the max-pool layer with an avg-pool layer did not show any improvements from the baseline model. Not only that, but using 2-gram counts instead of directly encoding the characters did not provide better results either.

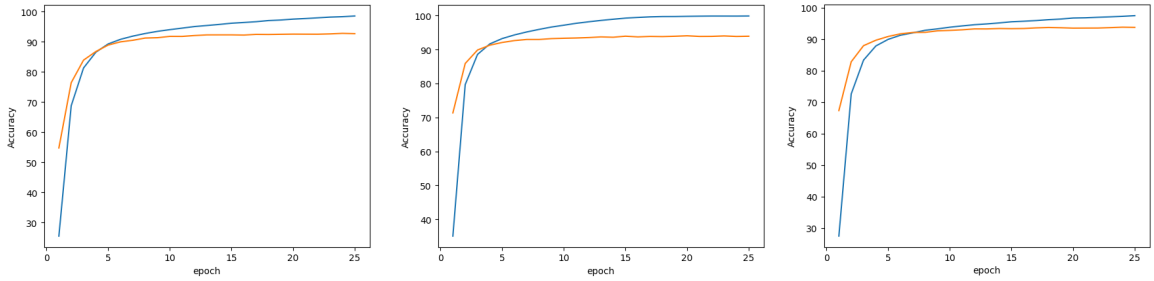

Figure 4: Train and Validation accuracies for Baseline (left), model 4 (center) and model 5 (right).

On the other hand, the more classical classification models had unmatched accuracies. Two main models were tested: Naive Bayes and SVMs, as described in the previous section. Table 2 shows the validation and test accuracies obtained by these models, in comparison to the baseline.

Last but not least, we have tried fine-tuning pre-trained models such as BERT. However, the required memory employed was too big and therefore training for many epochs was not possible and we reduced the number of epochs to three. After these three epochs, the validation accuracy had risen to 92%. However, it was not high enough to compete against the other models. We believe these results are mainly atributeable to the following three aspects:

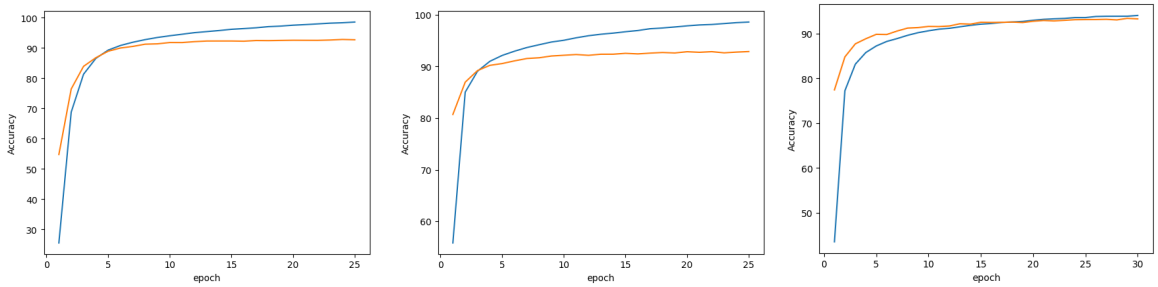

Figure 5: Train and Validation accuracies for Baseline (left), model 2 (center) and model 3 (right).

ID	*Model description*	*Validation accuracy*	*Test Accuracy*
1	Baseline	92.7%	93.5%
15	Naive Bayes	92%	94.9%
16	SVM (3x3)	95.4%	95.9%
17	SVM (3x3) grid search	96%	96.3%
18	SVM (3x4)	96.3%	96.7%

Table 2: Accuracies obtained with variations of the baseline model.

- BERT has been trained with 104 languages. Learning 131 more in 3 epochs is practically impossible.
- BERT has a maximum sequence length: 512. The given dataset contained longer sentences, which had to be clipped (shortened).
- Moreover, memory constraints forced our inputs to be of maximum length 384, having to discard relevant parts of the sentences.

Conclusions

On the one hand, we started from the baseline architecture that was given to us, and from there we developed variations in order to try to improve the accuracy. On the other hand, we used traditional machine learning models like SVMs and Naive Bayes, adapting the input to be able to fit the models, using n-gram counts instead of directly passing the encoded input sentences.

We have seen that increasing the number of parameters generally helps to obtain more accurate results, although it can sometimes lead to overfitting or to requiring more epochs to adequately train them. In order to avoid overfitting, we may add a dropout layer, which ensures that the distance between the training accuracy and the validation accuracy is not too large. Other alterations of the original architecture did not yield promising results.

After several tests, we have been able to verify that the best precisions and results have been those where we increased the hidden state size or added an RNN to the architecture.

However, the most surprising results have come when testing other models, more specifically machine learning models. As it has been shown in the Results section, these had significantly less training time and achieved superior results. The reasons we attribute to SVM's success can mainly be summarised as:

- SVMs work well in high-dimensional spaces, which is crucial for language processing where each feature can represent a different aspect of the language.
- SVMs can effectively handle non-linear data through the use of kernel functions. This is beneficial for language classification, as linguistic data often exhibit complex, non-linear relationships.
- SVMs have regularization parameters that help prevent overfitting. This was key for our particular task.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[2] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).