Big Data and Artificial Intelligence

Lab Assignment (TP) 2: Sentiment Analysis, a Text Classification Task

Supervisor: Montmaure Antoine

Written by : *Othmane Cherai & Hamza Houbbane*

Instructor: *Son Vu*
Department: Département Informatique
Date: March 30, 2023

Contents

1 Toying Around

1.1 Pre-processing

Data preprocessing is a fundamental step in any machine learning project, which aims to normalize the data for easier processing and better results. The Big Data Lab has already addressed this step, and the available data has been seemingly cleaned by applying various techniques, such as converting all text to **lowercase** and, removing **non-alphanumeric characters**, to avoid noise when introducing the data into the model. With that said, we can move on to the core of our problem, which is **tokenization**.

1.1.1 Tokenization

Tokenization is the process of breaking down raw text into small chunks or sense units called **"tokens"**. These tokens aid in understanding the context of the text. There are various tokenization rules, each adapted to a specific situation, with its own set of flaws. For instance, consider the sentence *"X's cat was left is the U.S."* A simple tokenization process would be splitting on punctuation, resulting in tokens such as {*"X," "'", "s," "cat," "was,"*} and so on. On the other hand, tokenizing this sentence using the **Penn TreeBank rules** would result in tokens such as {*"X's," "cat," "was,"*} and so on. While both tokenization rules result in similar tokens, the second method links the "'s" to "X" to understand the meaning of "that which belongs to X," making it a more appropriate approach.
In this first example, the splitting operation occurs upon each white space character, " " making it known as **"white space tokenization."**
Once you have a tokenized sentence, and by extension a tokenized corpus, further simplifications are to be so as to better shape the data for the upcoming processing. To further simplify this corpus, a token processing layer, either **stemming** or **lemmatization**, can be added. This layer's purpose is to standardize a common form shared between a set of words. Although this step has been overlooked in the present lab section, in real-world scenarios, this step is crucial for obtaining accurate and reliable results.

1.1.2 Stemming—Lemmatization

1.1.2.1 Stemming

Stemming is a technique used in natural language processing to standardize words by preserving only their stem and removing any affixes or suffixes. The resulting stem can then be used as a synonym for all the words that share the same root. For example, the words *"eats"* and *"eating"* both stem to *"eat"*, which can be used as a general term for both words. However, stemming can sometimes lead to confusion as different words with different meanings can share the same root. For instance, the words *"universal"*, *"university"*, and *"universe"* all share the root *"univers"*, but have different meanings. As such, stemming may not always be an optimal choice for text normalization.

1.1.2.2 Lemmatization

A better solution would be grouping together the different forms of a word so they can be analyzed as a single item. It is ultimately the same except that, here, the context is brought to

the words.

$$Likes \Rightarrow like,\ better \Rightarrow good$$

1.1.3 Vocabulary Creation

Now that we have all the components at hand, we can finally create our vocabulary dictionary. This task is carried by **get_word2idx**. Let's call this Vocabulary **V**.

2 Main Principle

2.1 Embeddings

One straightforward way to do so is the one-hot-encoding. Say you have a sentence $S = (w_i)_{i<k}$, w_i being the words of the aforementioned and k its length. Consider now the vector V_i, that would have zeros everywhere except for the index representing the corresponding word in the vocabulary.

	W_0	W_1	W_2	W_3	W_4
V_0	**1**	0	0	0	0
V_1	0	**1**	0	0	0
V_2	0	0	**1**	0	0
V_3	0	0	0	**1**	0
V_4	0	0	0	0	**1**

Table 1: One-Hot-Encoding of S

Now consider the vectorial space generated by the $(V_i)_{i<k,k\in N}$.These vectors, being linearly independent makes it so as no vectors admits a projections following any other axis. As such, *"good"* and *"bad"* for example are completely different words within this scope. However, our objective is to have words with similar context occupy close spatial positions *i.e*: minimize the distance between them. **Embeddings** were created is order to solve this problem.

2.1.1 How Word Embeddings work

The following applies to a supervised embedding algorithm. Consider the vocabulary V = $(v_i)(i < \#V)$ of a corpus, where v_i denotes the word at the i^{th} index. Let $S_i = (s_{i,n})$ be the feature vector of size *n* corresponding to the i^{th} word in V. The value of **n** is fixed by the user and is typically chosen to be large (around 300) for good results. The goal is to find a good set of hyperparameters. To achieve this, we randomly set the features and group them into an **embedding matrix** $M \in \mathbb{R}^{V\times i}$, where M is defined as $M = (S_i)$.
Let **E** be the **one-hot encoding** (OHE) matrix of a sentence **J**. E_m denotes the m^{th} column of the matrix, which represents the **one-hot encoded vector** ($\overrightarrow{OHE}$) of the m^{th} word in the sentence. For each word in **J**, we calculate $M \times E_m$. This operation returns the column of the index equal to the index of the **1** in the $\overrightarrow{OHE}$ vector. We then flatten the resulting vectors by concatenating them vertically. This produces a vertical vector of size $n \cdot length(J)$.
Next, we pass this vector through a neuron and compare the output to the real value. We then backpropagate the found loss and tune the parameters of the model. We continue training the

model until the loss is acceptable. At the end of this process, we will have a tuned embedding matrix. Note that the last column vector is of size equal to the maximum length of sentences in the corpus. If the tested sentence is shorter than the maximum length, we pad the remaining empty word slots with 0.

2.2 Train and Validation Sets

A new item introduced in this lab is the **validation set**. In machine learning, it is widely recognized that the goal is to train a model on a dataset so that it can generalize well to unseen data. However, training and test sets alone may not be sufficient to prevent overfitting of the model. This is where the validation set comes into play.
After the model has been trained on the training set (during the learning phase), it is evaluated on the validation set to assess its performance and fine-tune the hyperparameters. It is generally recommended to have multiple random validation sets, rather than just one, to get a better idea of the model's performance. This technique is known as **cross-validation**. However, this technique is not used in this lab and will not be further developed here.

2.3 Code Snippet

Having explained the basic principle, an analogy between it and the code is of order in order to better grasp the introduced notions.

2.3.1 Model 1: The Feed-Forward Network

Let us first consider the Class **FFNN** which inherits from nn.Module as clearly explicit in the script. The constructor of the said class takes as parameters: **embedding_dim, hidden_dim, vocab_size, num_classes**.

1. **embedding_dim**: is the dimension of the token embeddings that will be learned by the model. Equivalent to **n** in the aformentioned

2. **hidden_dim**: is the number and size of the hidden layers

3. **vocab_size**: is the size of the vocabulary, *i.e* the number of unique token in the Vocabulary dictionary. Equivalent to **#V** in the aforementioned.

4. **num_classes**: is the number of classes the model will predict. Here it is equal to 2: either 0(bad) or 1(good).

The **nn.Module** method **embedding(vocab_size, embedding_dim, padding_idx=0)** creates an embedding matrix of size $vocab_size \times embedding\_dim$ and automatically handles 0-padding for sentences of length inferior to the maximum length.
The **nn.Linear(embedding_dim, hidden_dim)** method applies a linear transformation to the input data. It transforms the word embeddings into a lower-dimensional space that is more manageable for downstream processing. It is used twice in the FFNN: once to define the hidden layer, which computes the average of the word embeddings, and once to define the output layer, which produces the final class predictions. Adding linear transformations to the model

introduces non-linearity, which allows the model to learn more complex relationships between the input and output.
The **Forward(self,x)** method takes a batch of sentences in tensor form and applies the layers of the FFNN to generate class predictions. It passes the input sequences through an embedding layer to obtain corresponding token embeddings, then computes the average of the embeddings for each sequence. It then passes the average embeddings through a hidden layer with the **ReLU** activation function and finally passes the output of the hidden layer through an output layer with a linear activation function. ReLU is chosen over Sigmoid because it avoids the vanishing gradient problem, which occurs when the gradient becomes too small during backpropagation, causing the weights to change very slowly or not at all.
The accuracy measurement is done by the **accuracy** function, which implements the following formula:

$$Accurary = \frac{TruePositives + TrueNegatives}{TruePositives + TrueNegatives + FalsePositives + FalseNegatives}$$

torch.sigmoid is used to convert the output of the model to a tensor of probabilities. This function is preferred for binary classification tasks.
The details of how to optimize the model, tune hyperparameters, measure loss, etc., are explained in the previous lab.

2.3.2 Model 2: The Convolutional Neural Network

The architecture of the CNN model is quite similar to that of the FFNN model, with the addition of a few key features, namely the **dropout layer**, **convolution layer**, and **max pooling layer**. The max pooling layer has already been covered in the previous lab report and will not be discussed further here.

2.3.2.1 Convolutional Layer

The **convolutional layer** is a set of filters that extract features from the input and produce a feature map. This layer is utilized in a way that allows the model to better detect patterns and extract more meaningful features from the input.

2.3.2.2 Dropout Layer

The **dropout layer** is a technique that randomly drops out a certain fraction of the input to a layer during training. The purpose of this layer is to prevent over-fitting, which occurs when the model becomes too dependent on the output of the previous layer. By forcing the network to learn more robust features, the dropout layer ensures that the model is better equipped to generalize to new, unseen data.

2.3.2.3 Code

The main difference in the class constructor is the attributes **window_size** and **dropout**. **output_dim** is the same as the above-introduced **num_classes**.

- **window_size**: is the size of the sliding window used for the convolution layer. Check attached video link found in the last report to better visualise it.

- **dropout**: is the probability of dropping part of the information in the dropout layers.

The **forward** method takes the input text and passes it through the embedding layer, which produces word embeddings that are expanded in dimension (**unsqueeze(1)**) so that they can be processed by the convolution layer. The convolutional layer is used to compute a features map, with a **ReLU** activation function that introduces non-linearities. This is followed by the **max_pool1d** layer, which pools the resulting tensor to produce a pooled feature map. The dropout layer is then applied to reduce overfitting, before passing the pooled feature map through a final layer to obtain the output scores.

3 FFNN or CNN for Sentiment Analysis/Classification?

Based on the introduction to CNN in the lab text, it seems that CNNs are particularly adept at capturing patterns in data. Applied to the task of sentiment analysis, this could mean that CNNs are better suited for identifying patterns of words that indicate a particular sentiment, regardless of their order or location within a sentence. It's worth noting that CNNs are often used for image classification tasks, which typically involve large datasets. In contrast, FNNs are simpler and may be better suited for smaller datasets and simpler classification tasks.

4 Ressources

- https://towardsdatascience.com/skip-gram-nlp-context-words-prediction-algorithm-5bbf34f84e0c
- https://towardsdatascience.com/tokenization-for-natural-language-processing-a179a891bad4
- https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa
- https://www.v7labs.com/blog/neural-networks-activation-functions#h1
- https://www.youtube.com/watch?v=sZGuyTLjsco
- Attached documents