Meghna BHAUGEERUTTY, Gabrielle MORIN et Jean Baptiste DEL RIO

Université Paris Dauphine | 2023-2024 Academic year

Inference Relation between two short texts

Natural Language Processing

**Introduction**

For this project, we decided to analyze the inference relationship between short sentences. Our objective is to develop a model capable of accurately classifying text pairs into three relational categories : contradiction, neutral or entailment. To do this, we implement a distilled version of the BERT base model which is used to categorize manually written sentence pairs.

To evaluate the performance of our model, we compare its accuracy to the one of several typical NLP models such as LSTM and GloVe, that we trained on our same dataset.

**1. Data Preparation**

The first step of this project is to load the data from the Stanford Natural Language Inference (SNLI) corpus. This database is composed of sentence pairs annotated with labels indicating the relationship between the two sentences: contradiction, neutral or entailment. It contains 570k human-written English sentence pairs, manually labelled and then separated into a train (549 367 pairs), validation (9 842 pairs) and test (9 824 pairs) sets.

To better understand the data structure and characteristics, we extracted an example for each type of label in the dataset.

|  |  |  |
| --- | --- | --- |
| Premise | Judgments | Hypothesis |
| A man inspects the uniform of a figure in some East Asian Country. | Contradiction | The man is sleeping. |
| An older and younger man smiling. | Neutral | Two men are smiling and laughing at the cats playing on the floor. |
| A soccer game with multiple males playing. | Entailment | Some men are playing a sport. |

Finally, we check if there is a data imbalance in the train set with one label appearing more often than the other two. It is not the case, and each label represents approximately a third of the dataset.

Because our data is so voluminous, we use a tokenizer. That way the machine can process and understand vast amounts of text.

**2. Modelling**

First, we initialize the model using *DistilBertForSequenceClassification* from the Hugging Face library, specifying the pre-trained *distilbert-base-uncased* model and tailoring it for a three-class classification task. (We also check if GPU is available, otherwise the model remains on the CPU.)

The optimizer used is *AdamW*, initialized with the model's parameters and a learning rate of 5e-5, which is commonly effective for fine-tuning BERT-based models.

Training spans over three epochs. Within each epoch, the model is set to training mode using *model.train()*. For each batch of data from the *train\_loader*, the inputs are moved to the device where the model resides. The forward pass is executed to obtain outputs and compute the loss. This loss is accumulated to monitor the training progress. Backpropagation is performed via *loss.backward(),* and the optimizer updates the model's parameters with *optimizer.step().* To prevent gradient accumulation, *optimizer.zero\_grad()* is called after each update. The average loss per epoch is printed to track the training process.

In the evaluation phase, the model is switched to evaluation mode using *model.eval().* Each batch from the *val\_loader* is processed similarly, but with *torch.no\_grad()* to disable gradient calculation, which reduces memory usage and speeds up computations. The logits are obtained and converted to predictions by taking the index of the highest logit value. The validation accuracy is computed by comparing the predictions to the true labels.

Finally, the validation accuracy is printed, providing an indication of the model's performance on the validation set. We have a validation accuracy of 87%, which is quite interesting.

It is however quite important to note that for the model to train on each epoch, so 68671 observations, it requires 3 minutes for every 40 observations. This is the equivalent of 85 hours for one epoch, so 340 hours. This is thus a very time-intensive project/code.

**3. Baseline Comparison**

**3.1 Preparing Data for Classical Models**

To train classical machine learning models, we need to prepare the data in a suitable format. We combine each premise and hypothesis into a single text string using the *combine\_texts()* function. This transformation is applied to the entire dataset, effectively creating a new column with the combined text. We then split the data into training and validation sets using the *train\_test\_split()* function, ensuring that we have separate sets for model training and evaluation. This step is critical for preventing overfitting and assessing the generalization performance of our models.

**3.2 Creating Bag-of-Words Representation**

For classical models, we convert the combined text data into a bag-of-words representation using the *CountVectorizer()*. This method transforms each text into a vector of word counts, where each unique word in the dataset is represented as a feature. The bag-of-words model is a simple yet effective way to represent textual data for machine learning algorithms. It captures the frequency of words but ignores the order and context, making it suitable for models like Naive Bayes and Logistic Regression.

**3.3 Training and Evaluating Classical Models**

We train and evaluate two classical machine learning models: Naive Bayes and Logistic Regression. These models are chosen for their simplicity and efficiency in handling text classification tasks.

Naive Bayes and Logistic Regression are both classical models used for text classification. Naive Bayes is based on Bayes' theorem with a strong assumption of independence between features, making it simple and fast, particularly suited for tasks like spam filtering and sentiment analysis, though its independence assumption is often unrealistic for textual data. Logistic Regression, on the other hand, is a linear classification model that uses a logistic function to estimate the probability of class membership. It is effective for binary and multi-class classification, offers good interpretability, and handles high-dimensional data well, but it may be less effective for non-linear data and requires careful feature selection.

Using the bag-of-words vectors from the training set, we fit each model and then evaluate their performance on the validation set. Predictions are made using the validation data, and accuracy scores are calculated to measure how well each model performs. This step provides a baseline performance metric for classical models, against which we can compare more sophisticated approaches.

**3.4 Training and Evaluating the GloVe Model**

Next, we implement a GloVe (Global Vectors for Word Representation) model to learn word embeddings from the dataset. We create a co-occurrence matrix from the bag-of-words representation and train the GloVe model to generate dense vector representations for each word. These embeddings capture semantic relationships between words, providing a richer representation compared to the bag-of-words model. We then use these embeddings to transform our data and train a Logistic Regression classifier on the resulting vectors. The classifier is evaluated on the validation set, and its accuracy is calculated to assess the effectiveness of the GloVe embeddings.

**3.5 Preparing Data for the LSTM Model**

For the LSTM (Long Short-Term Memory) model, we need to tokenize the text data into sequences of tokens. Using the *DistilBertTokenizer*, we convert the texts into tokenized sequences suitable for input into the LSTM model. We prepare *DataLoader* objects for both the training and validation sets to facilitate efficient batch processing during model training and evaluation. This preprocessing step is essential for handling sequential data and leveraging the capabilities of LSTM networks in capturing temporal dependencies.

**3.6 Training and Evaluating the LSTM Model**

We initialize the LSTM model with parameters such as input dimension, hidden dimension, output dimension, and the number of layers. The model is trained for a specified number of epochs, during which it learns to classify the sentence pairs based on their tokenized representations. For each batch, the inputs and labels are processed by the model, and the loss is computed and minimized using backpropagation. After training, we evaluate the LSTM model on the validation set by making predictions and calculating the accuracy. This step demonstrates the potential of deep learning models in handling complex text classification tasks.

**3.7 Comparing Performances**

Finally, we compare the performance of all the trained models: Naive Bayes, Logistic Regression, GloVe + Logistic Regression, and LSTM. The accuracy scores of these models are printed and compared to determine which model performs best on the given task of determining the relationship between sentence pairs. This comprehensive comparison provides insights into the effectiveness of different modeling approaches and helps identify the most suitable model for the task at hand.

The performance comparison of the models reveals interesting insights into their effectiveness for text classification. Logistic Regression achieves the highest accuracy at 54.28%, indicating its robustness in capturing relevant features using the bag-of-words representation. Naive Bayes follows with an accuracy of 51.73%, performing slightly worse due to its strong independence assumption, which is less suited for natural language data where word dependencies matter. Surprisingly, the combination of GloVe embeddings with Logistic Regression results in a lower accuracy of 44.68%, suggesting that the semantic relationships captured by GloVe embeddings may not be as beneficial in this context as the simpler bag-of-words approach. The LSTM model, despite its potential for handling complex text classification tasks and capturing temporal dependencies, achieves the lowest accuracy of 33.82% and takes significantly longer to train, indicating that it might not be well-suited for this particular dataset and baseline comparison.