Key-Points Matching in NLP

Natural Language Processing
Project

Alex Costanzino
alex.costanzino@studio.unibo.it
GitHub

Marco Costante
marco.costante@studio.unibo.it
GitHub

Alessandra Stramiglio
alessandr.stramiglio@studio.unibo.it
GitHub

Xiaowei Wen
xiaowei.wen@studio.unibo.it
GitHub

Academic year 2021-2022

Abstract

Argument mining is a recent and developing field of research in natural language processing. Essentially, argument mining involves automatically identifying argument structures in free text, such as the argument's conclusion, premises, and reasoning scheme, as well as their interrelationships and counter-considerations.

One of the task that composes argument mining's pipeline is key-point matching: given a sentence (typically an argument) and a key-point, determine whether the latter matches the proposed sentence.

The classification task can be also enriched with other features that may help to tell apart matching and non-matching pairs. In particular, in our scenario, also the topic and the stance of the argument have been integrated to provide context to the models.

Both unsupervised and supervised methods are experimented. In particular, clustering with GloVe embeddings, BERT vector representations and tf-idf have been endeavoured, as unsupervised method, in order to provide a baseline for more advanced methods. Subsequently, also two neural architectures have been assessed in order to exploit the training data available.

At the end, a comparison with a particular focus on the type of error made by each model has been performed.

Contents

Problem presentation

Given a debatable topic, a set of key points per stance, and a set of crowd arguments supporting or contesting the topic, report for each argument its match score for each of the key points under the same stance towards the topic. The task is essentially a binary classification problem.

Setup

The project is developed on Colaboratory Pro, from Google Research, which provides the following hardware:

- Intel® Xeon™ CPU @ 2.30GHz;
- NVIDIA® Tesla P100-PCIE™ 16GB;
- DDR5 32 GB RAM.

The main libraries employed to tackle the problem are:

- *NumPy 1.22.0*, a package for scientific computing with Python;
- *Pandas 1.4.0*, a data analysis and manipulation tool;
- *TensorFlow 2.7.0*, an end-to-end open source platform for machine learning;
- *TensorFlow Text*, a library to work with input in text form such as raw text strings or documents;
- *TensorFlow Hub*, a repository of trained machine learning models ready for fine-tuning;
- *Matplotlib 3.5.1*, a comprehensive library for creating static, animated, and interactive visualizations in Python;
- *tqdm 4.62.3*, a fast, extensible progress bar for Python;
- *contractions 0.1.66*, a comprehensive set of the most common contractions in English language;
- *Focal Loss 0.0.8*, a tiny library that implements binary focal loss for unbalanced datasets;
- *SentenceTransformers 2.1.0*, a Python framework for state-of-the-art sentence, text and image embeddings;
- *NLTK 3.6.7*, an open source Python library for Natural Language Processing;
- *scikit-learn 1.0.2*, an open source machine learning library for the Python programming language.

Data

The dataset employed is the ArgKP dataset by IBM [13]. It comes already divided into train, development and test splits. This dataset contains around 24K argument and key-point pairs, for 28 controversial topics. Each pairs is labeled as matching or non-matching, as well as assigned a stance towards the topic.

Given a set of key points for a topic, an argument could be matched to one or more key points, or to none of them.

Dataset inspection

Each split is divided into three different files, in **.csv* format:

- Arguments file, that contains the columns of `arg_id`, `argument`, `topic` and `stance`;
- Key-points file, that contains the columns of `key_point_id`, `keypoint`, `topic` and `stance`;
- Labels file, that contains the columns of `arg_id`, `key_point_id` and `label`.

The dataset is directly loaded via *url* from its GitHub repository [11]. After the loading a quantitaive analysis of the dataset is performed.

The split proportion are:

	Training	Development	Test
Arguments	5583	932	723
Key-points	207	36	33
Labels	20635	3458	3426

The labels are highly unbalanced:

	Training	Development	Test
Positive	4260	738	552
Negative	16375	2720	2874
Total	20635	3458	3426

Due to this unbalance some precautions regarding the loss function, the evaluation metrics and the classification threshold are taken into account.

Dataset processing

The dataset already comes pretty clean. Nevertheless some further cleaning is done in order to better exploit pre-trained models. In particular:

- The text is lower-cased since we employed uncased pre-trained models;
- The contractions are expanded since they may have results as out-of-vocabulary words;
- Special characters and uncommon symbols are filtered out.

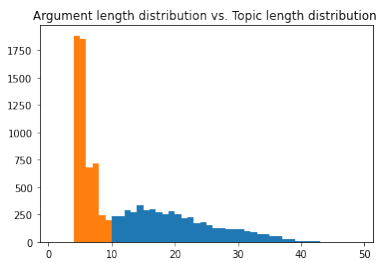
The stopwords are maintained since they are useful to tell apart different stances toward the topic.

Subsequently, the text is tokenized with `wordpunct` by *NLTK*.

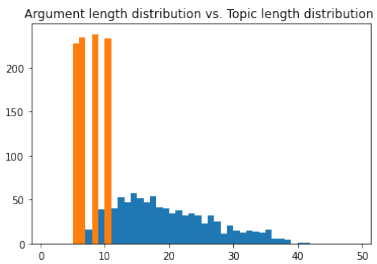
Padding is not needed for processor and the pre-trained models since they automatically takes into account different lengths, but is needed for the GloVe embeddings.

In any case, length distribution studies are useful to decide if it makes sense to employ attention mechanisms. Key-points lengths are not considered since their maximum length is fixed at most ten tokens.

For the training and development set the maximum sequence length is of 44 tokens, so attention mechanisms may not be very useful.



(a) Training set distribution



(b) Development set distribution

1 Unsupervised methods

At first, some clustering methods based on pre-trained models are experimented to create a baseline [8]. Rough evaluation methods and metrics are employed for the moment, while more refined methods are used later for supervised methods.

1.1 GloVe embeddings

GloVe (global vectors for word representation) is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space [1].

The training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence. Owing to the fact that the logarithm of a ratio equals the difference of logarithms, this objective associates (the logarithm of) ratios of co-occurrence probabilities with vector differences in the word vector space. Because these ratios can encode some form of meaning, this information gets encoded as vector differences.

In our case, after the previous tokenization of the test dataset, the latter is encoded through a dictionary, built taking into account also the out-of-vocabulary words of training, development and test set. Such dictionary is initialized starting by the GloVe one, out-of-vocabulary words are taken as random vectors, sampled by a uniform distribution.

After the encoding, a 300 dimensional GloVe embedding is applied (from a matrix).

Then, cosine similarity between the embedded pairs of argument and key-point is applied and results are rounded with a $\frac{1}{2}$ threshold. The same process is repeated concatenating also the topic at the end of each sentence, to verify if it can provides more context to the model.

	GloVe without topic	**GloVe with topic**
Precision	0.73	0.74
Recall	0.83	0.36
F1-score	0.77	0.40
Balanced accuracy	0.50	0.51

The general performances are worsened if the topic is taken into account, the cause can be the redundancy of such feature.

1.2 BERT vector representations

BERT (bidirectional encoder representations from transformers) provides dense vector representations for natural language by using a deep, pre-trained neural network with the transformer architecture [4]. While GloVe is a context independent model that computes a single embedding for each word, BERT is a contextualized embedding model that takes the entire sentence into account.

BERT provides three different kinds of output [12]:

- `pooled_output`: pooled output of the entire sequence with shape `[batch_size, hidden_size]`. It can be used as sentence representation;

- `sequence_output`: representations of every token in the input sequence with shape `[batch_size, max_sequence_length, hidden_size]`;
- `encoder_outputs`: a list of n tensors of shapes `[batch_size, sequence_length, hidden_size]` with the outputs of the n-th transformer block.

The number of hidden states and transformer layers depend on the particular implementation of BERT (or other variations).

For this task a lighter implementation provided by *Sentence Transformers*, that returns only the pooled output, is exploited [6]. This implementation takes strings as input and automatically tokenizes, encodes and transforms the text.

Then, once again, cosine similarity between the transformed pairs of argument and key-point is applied and results are rounded with a $\frac{1}{2}$ threshold. The same process is repeated concatenating also the topic at the end of each sentence, to verify if it can provide more context to the model.

	BERT without topic	**BERT with topic**
Precision	0.81	0.58
Recall	0.55	0.50
F1-score	0.60	0.14
Balanced accuracy	0.63	0.50

Again, the general performances are worsened if the topic is taken into account.

1.3 Tf-idf

Tf–idf (frequency–inverse document frequency) is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It can be used as a weighting factor in text mining and language modeling [5]. Such value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

In this task, tf-idf is computed with `TfidfVectorizer` by *scikit-learn* for each pair of argument and key-point. Once again, also the topic concatenation is endeavoured, and evaluation is performed with cosine similarity.

	Tf-idf without topic	**Tf-idf with topic**
Precision	0.87	0.73
Recall	0.84	0.68
F1-score	0.77	0.70
Balanced accuracy	0.50	0.51

As usual for unsupervised methods topic context integration worsens the performances.

2 Supervised methods

Supervised methods are the core of this task, since neural architectures can exploit more efficiently context features such as topic and stance. Also in this case, an architecture with no

context integration is tried, along with another one where topic and stance are used to enhance the classification.

2.1 Data generators

For supervised methods data generators are implemented in order to avoid high RAM occupancy. Two different data generators are implemented for the two different architectures. Since the second one is a reduced version of the first, only the latter is presented.

The generator is initialized with:

- A `shuffle` parameter to add more robustness to the batches of data;
- A `batch_size` parameter to decide the size of the aforementioned batches;
- The `dataset` parameters;
- The `embedder` parameter to pass the chosen transformer;
- A `no_last_state` parameter to decide how many states of the transformer's output concatenate to obtain the representation.

The data is handled in order to parse and align the pairs in the labels dataframes. Then, topics, arguments and keypoints are encoded through BERT, and averaged over the last `no_last_state` states.

A single datapoint is output in the form of: `([topic, argument, keypoint, stance], label)`.

These generators inherits from `Sequence` *Keras* class, a safer way to do multiprocessing. Such structure guarantees that the network will only train once on each sample per epoch, compared to simple generators. Nevertheless, several bugs of memory leaks have been reported.

In the composition of the batch the `__getitem__()` magic method is defined. This method at first define the indexes to fetch the data, according to the chosen batch size.

Then, `arg_id` and `key_point_id` are extracted from the aforementioned indexes, through the labels dataframe. Subsequently, also topics, arguments, key-points, stances and labels are extracted, locking their respective columns, through their dataframes.

Moreover, topics, argument and key-points are encoded through a pre-trained processor layer and BERT vector representations are extracted with another pre-trained transformer. Such layers are both loaded with *TensorFlow Hub*.

Finally, the chosen last states are concatenated, flattened and then output.

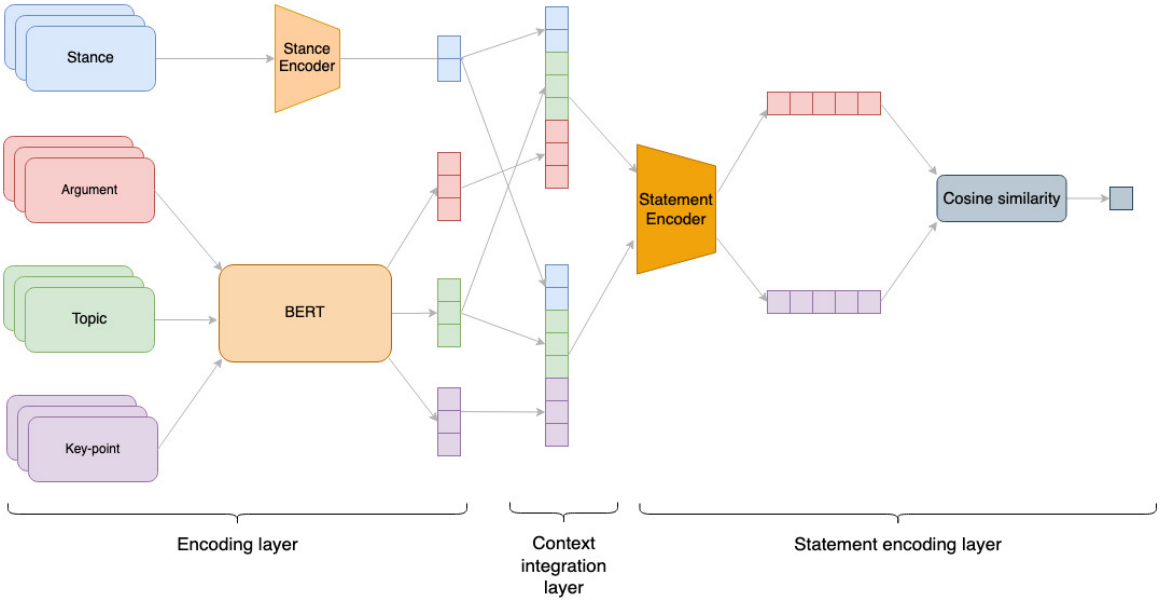
2.2 Neural architecture no. 1

2.2.1 General scheme

The experimented neural architecture is composed by [10]:

- *Stance encoder*: encodes the stance value (1 or -1) employing a fully-connected network with no activation function to map the scalar input to a N-dimensional vector space;

- *BERT*: extracts the contextualized representation for textual inputs. In our use-case it is already integrated in the data generator and no fine-tuning is performed. This particular approach is known as zero-shot learning, a type of machine learning technique where the model is used without fine-tuning on a particular task. The canonical method [7] to achieve the final embedding of a given input is adopted, which is concatenating the last four hidden states of the [CLS] token. These embeddings are fed into the context integration layer as an aggregate representation for topics, arguments and key points;
- A context integration layer is done by stacking stance encoding, topic encoding and statements encoding;
- *Statement encoder*: another fully-connected network on top of the context integration layer to get the final D-dimensional embeddings for key points or arguments.



2.2.2 Implementation

The presented architecture is implemented with *Keras* library, based on *TensorFlow* backend.

For the stance encoder, a simple dense layer with 8 output nodes and no activation function is employed, in order to project the stance into an 8-dimensional array. Adding dimensionality to the stance helps further encodings to better separate the sentiment of the various stances.

For the statement encoder, a double stack of dense layers is employed. Between these layers, also strong 50% dropout layers, along with Lasso and Ridge regularizers, are interleaved in order to avoid overfitting. The first layer has half of the nodes of the second, mimiking an encoder structure. Several number of output nodes are tried to trade-off good representations and overfitting, and the best value found is of 32 output nodes. Several activation functions are also endeavoured, ending up with the choice of `selu` activation function.

The layers are shared. Shared layers are often used to encode inputs from similar spaces (for example two different pieces of text that feature similar vocabulary). They enable sharing of information across these different inputs, and they make it possible to train such a model on less data [14]. If a given word is seen in one of the inputs, that will benefit the processing of all inputs that pass through the shared layer.

After the statement encoder, the resulting vectors are used to compute cosine similarity, and then the result is rescaled in range $[0, 1]$ for compatibility reason with the binary focal loss function.

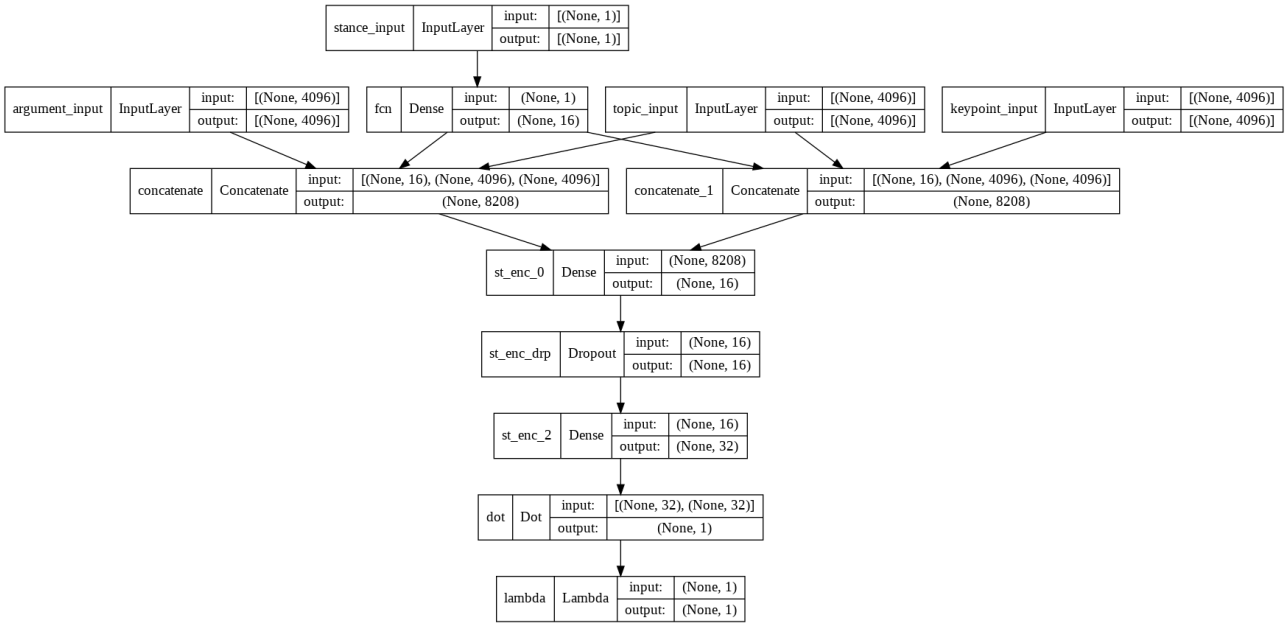
The binary focal loss function with $\gamma = 2$ is used since the dataset is very unbalanced.

For the training, several optimizer were tested, but at the end Adamax is used since it has a particular synergy with embedded data.

The learning rate is handled with a cosine decay scheduler with warm restarts. Such learning rate schedule has two parts:

- Cosine annealing, that has been shown to perform better than alternatives like simple linear annealing;
- Warm restarts, that means that every so often, the learning rate is raised back up.

This strategy has as core idea having periods with high learning rates and periods with lower ones [3]. Periods of high learning rates are useful to prevent the learner from getting stuck in a local minima. Periods of lower learning rates allows the learner to converge to a near-true-optimal point within the global minima it finds.

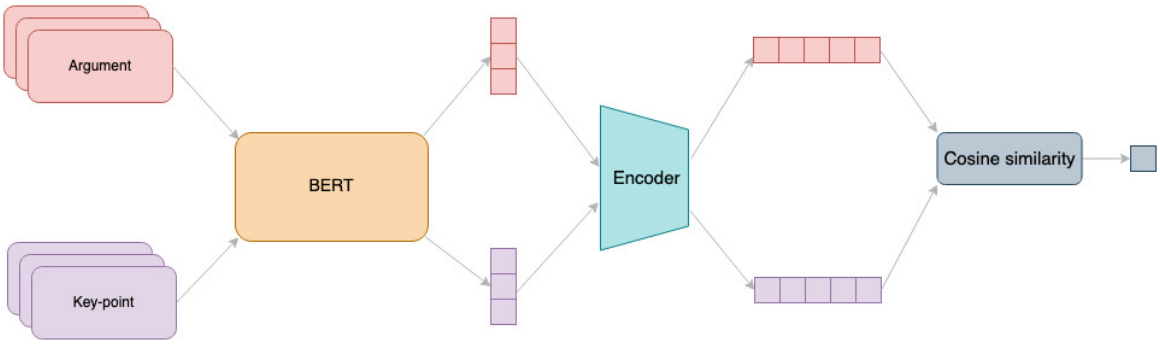


2.3 Neural architecture no. 2

2.3.1 General scheme

The experimented neural architecture is composed by:

- *BERT*: extracts the contextualized representation for textual inputs. In our use-case it is already integrated in the data generator and no fine-tuning is performed. This time to achieve the final embedding of a given input, all hidden states of the [CLS] token are considered;
- *Encoder*: a fully-connected network on top of the extracted vector representation.



2.3.2 Implementation

The presented architecture is implemented with *Keras* library, based on *TensorFlow* backend.

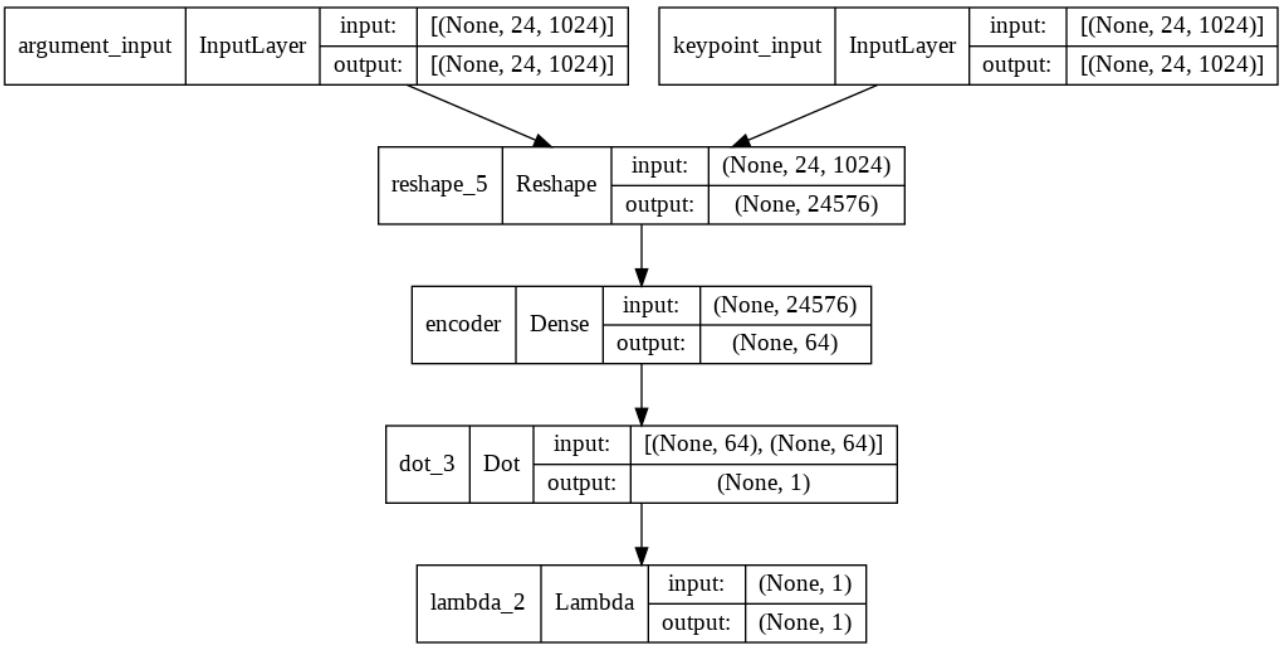
For the encoder, a dense layer is employed. Also strong 50% dropout layer, along with Lasso and Ridge regularizers, is used in order to avoid overfitting. Several number of output nodes are tried to trade-off good representations and overfitting, and the best value found is of 32 output nodes. For dense layers `gelu` activation function are employed.

Once again the layers are shared.

After the encoder, the resulting vectors are used to compute cosine similarity, and then the result is rescaled in range $[0, 1]$ for compatibility reason with the binary focal loss function.

The binary focal loss function with $\gamma = 2$ is used since the dataset is very unbalanced.

Once again, for the training Adamax optimizer is used since it has a particular synergy with embedded data, along with cosine decay scheduler with warm restarts.



2.4 Evaluation and threshold tuning

The training has been carried tracking down the the validation loss. Binary validation accuracy, validation recall and validation precision have also been monitored.

The best models provided the following results:

Architecture	**Parameters**	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Threshold**
No. 1	131 920	0.66	0.66	0.61	0.65	0.29
No. 2	786 464	0.66	0.59	0.58	0.66	0.35

These metrics are calculated taking into account a different rounding with respect to the usual $\frac{1}{2}$ threshold, since the dataset is unbalanced.

Threshold tuning is achieved with ROC curves: these curves typically feature true positive rate on the vertical axis, and false positive rate on the horizontal axis. This means that the top left corner of the plot is the ideal point, hence a false positive rate of zero and a true positive rate of one. The steepness of ROC curves is important, since it is ideal to maximize the true positive rate while minimizing the false positive rate [2].

The idea of using the ROC curve for tuning the threshold is to identify that threshold that gives us the top left corner of the curve, analytically speaking, that threshold ξ which satisfies the equation $TPR(\xi) = 1 - FPR(\xi)$.

This is a general way of tuning the threshold, often found in literature.

More generally, the goal is to find the threshold that satisfies the following equation:

$$\xi_* = \arg\min_{\xi} |TPR(\xi) + FPR(\xi) - 1|$$

This approach convert a root finding problem into an optimization problem, calculating all the scores generated by a model, finding the one that minimizes the argument above.

Such procedure is effortlessly implemented with *scikit-learn* methods.

Samples weighting has also been endeavoured, taking into account the relative frequency of the classes, but since it did not improved significantly the results, it is not considered for evaluation purposes.

3 Comparison, error analysis and further improvements

3.0.1 GloVe embeddings

No topic	**Positive**	**Negative**
Positive	2843	31
Negative	545	7

With topic	**Positive**	**Negative**
Positive	810	2064
Negative	142	410

GloVe embeddings approach is good at catching true positive matches, but fails to detect correctly negative samples, misclassifying often positive values. Topic implementation worsen the problem since it exacerbates false detections.

3.0.2 BERT vector representations

BERT vector representations is better at detect true negative matches, but false detections remains a major problem. Topic implementation broke the classification.

No topic	**Positive**	**Negative**
Positive	1453	1421
Negative	131	421

With topic	**Positive**	**Negative**
Positive	4	2870
Negative	0	552

In both cases the reason may be the necessity of fine-tuning on this particular dataset, since it does not contains a lot of words (there are very few OOVs), but we have no information on the spatial distribution between dataset-specific embeddings, with respect to the pre-trained Glove or BERT ones.

Essentially, training could help in better distancing word embeddings of the ArgKP dataset.

3.0.3 Tf-idf

No topic	**Positive**	**Negative**
Positive	2874	0
Negative	550	2

With topic	**Positive**	**Negative**
Positive	2178	696
Negative	410	142

Tf-idf is good at catching true positives and is very good with false positives. Nevertheless, it has a lot of false negative detections and it is bad at classifying true negatives values. In this case topic seems to balance the classification.

3.0.4 Neural architectures

Arch. #1	**Positive**	**Negative**
Positive	1890	984
Negative	189	363

Arch. #2	**Positive**	**Negative**
Positive	1907	967
Negative	186	366

Both neural architectures reach similar results. The main difference between them is that the first one it is endowed with a context integration layer, where topic and stances of the arguments are also provided. Moreover, the first architecture works with only the last four state of [CLS] token, while the second one uses all the vector. Furthermore, the first architecture is faster to train.

Hence, even if the result are comparable, topic and stance provide useful information when employed as features for the training. A further training may shows a divergence of results, and, once again, fine-tuning should improve classification.

Even if neural architectures are far from being perfect classifiers, they are generally more balanced, with a particular bias in the misclassification of false positives, that is a trend of all the models.

This trend can be due to the fact that there are several systematic patterns of errors in the data.

In most cases, non-matching arguments and key points received a high match score if:

- They share some key phrases;
- They share a large portion of the sentence, but not the main point;

- They are at least partially related, but labeled as non-matching due to a better fitting key point for the same argument.

For arguments and key points that were labeled as matched but received a low match score, the relation was in many cases implied or required some further knowledge.

Such problems have been also identified by other researchers that worked on the same dataset [9]. Hence, the effort to improve this task should be more towards the mitigation of this bias, in contrast to the enhancement of the neural architectures. Moreover, such problems make evident the reasons why unsupervised models tend to fail.

References

[1] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[2] Peter Flach and Meelis Kull. "Precision-Recall-Gain Curves: PR Analysis Done Right". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/33e8075e9970de0cfea955afd4644bb2-Paper.pdf.

[3] Ilya Loshchilov and Frank Hutter. "SGDR: Stochastic Gradient Descent with Restarts". In: *CoRR* abs/1608.03983 (2016). arXiv: 1608.03983. URL: http://arxiv.org/abs/1608.03983.

[4] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[5] Shahzad Qaiser and Ramsha Ali. "Text Mining: Use of TF-IDF to Examine the Relevance of Words to Documents". In: *International Journal of Computer Applications* 181 (July 2018). DOI: 10.5120/ijca2018917395.

[6] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: https://arxiv.org/abs/1908.10084.

[7] Chi Sun et al. "How to Fine-Tune BERT for Text Classification?" In: *CoRR* abs/1905.05583 (2019). arXiv: 1905.05583. URL: http://arxiv.org/abs/1905.05583.

[8] Roy Bar-Haim et al. "From Arguments to Key Points: Towards Automatic Argument Summarization". In: *CoRR* abs/2005.01619 (2020). arXiv: 2005.01619. URL: https://arxiv.org/abs/2005.01619.

[9] Roy Bar-Haim et al. *From Arguments to Key Points: Towards Automatic Argument Summarization*. 2020. arXiv: 2005.01619 [cs.CL].

[10] Hoang Phan et al. "Matching The Statements: A Simple and Accurate Model for Key Point Analysis". In: *Proceedings of the 8th Workshop on Argument Mining*. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 165–174. DOI: 10.18653/v1/2021.argmining-1.17. URL: https://aclanthology.org/2021.argmining-1.17.

[11] GitHub. *Quantitative Summarization – Key Point Analysis Shared Task*. URL: https://github.com/IBM/KPA_2021_shared_task.

[12] Google. *Bidirectional Encoder Representations from Transformers (BERT)*. URL: https://tfhub.dev/google/collections/bert/1.

[13] IBM. *IBM Project Debater - Debater Datasets*. URL: https://research.ibm.com/haifa/dept/vst/debating_data.shtml.

[14] Keras. *Shared layers*. URL: https://keras.io/guides/functional_api/#shared-layers.