Assignment 5: RNNs and BERT-based Models for Review Classification

Vilém Zouhar
s5000076
vilem.zouhar@gmail.com

Edu Vallejo Arguinzoniz
s5016894
vallezoniz@gmail.com

Abstract

We explore the task of 6-fold review classification using fine-tuned BERT and training RNNs with static word embeddings. We investigate their architectures and hyperparameters on a small e-commerce dataset and follow up with an inspection of the model stability via random perturbations in the text input space. We also use the sentence embeddings from the various models in kNN classification which do not yield the best results but provide a reasonable baseline.

Even though fine-tuning BERT yields better results than training an RNN from scratch, the difference is not large and can be offset by the longer training and higher memory footprint.

1 Introduction

Topic classification is an important part of many modern commercial systems. While usually there are numerous topics to classify, we treat this as a 6-way classification task.

With BERT-based models with use the following instances: BERT (Devlin et al., 2019), Sentence-BERT (Reimers and Gurevych, 2019) and DPR (Karpukhin et al., 2020). We compare those to the following RNNs:[1] LSTM (Hochreiter and Schmidhuber, 1997), GRU (Cho et al., 2014) and vanilla RNN (Rumelhart et al., 1986). And consider not only their individual architecture/hyperparameter space but also their combination in the form of using BERT-derived embeddings for the RNNs as opposed to using GloVe (Pennington et al., 2014). Apart from comparing the model performances we also investigate how much the models are sensitive to perturbations in the data. We also craft a kNN classification model based on the embeddings from BERT-based models.

In Section 2 we discuss the data and the evaluation methodology. The description and results of various experiments are presented in Section 3, with the main comparison of models found in Table 12. We conclude in Section 4.

1.1 Related Work

Because RNN models are now the standard for text processing, they have been extensively studied for text classification (Zhou et al., 2016; Yogatama et al., 2017; Wang et al., 2019). Similarly, fine-tuning BERT for text classification has been thoroughly explored by Sun et al. (2019). Finally, we compare there results to models of Logistic Regression (Zouhar and Arguinzoniz, 2021a) and SVM (Zouhar and Arguinzoniz, 2021b) which we re-run on the same data sets.

2 Methods

We use 5k reviews for training and 500 for the development and test set each. More detailed analysis can be found in Zouhar and Arguinzoniz (2021a). The classes are somewhat balanced and a blind-baseline hence is $16.4\% - 17.1\%$ accuracy.

2.1 BERT-like pre-trained language models

To further boost the performance of the algorithm we make use of pre-trained language models in the form of BERT and BERT-like pre-trained transformers.

Pretrained language models are a natural evolution of pre-trained static embeddings. Instead of learning word representations, entire language models are learnt that better harness language understanding. This understanding goes beyond

[1] In this report we refer to simple recurent networks as *vanilla RNNs* and the whole family as *RNNs* (LSTM, GRU, vanilla RNN).

just lexical understanding, contrary to word embeddings, it has been shown that language models learn about higher-level linguistic phenomena such as syntax (Htut et al., 2019) and semantics.

2.1.1 Using language models

Pretrained language models can be exploited in different ways, in the case of BERT-like models, there are two main approaches: (offline) using BERT as a static text-encoder and (online) fine-tuning BERT. We will use both approaches for this work.

To use BERT as a text-encoder we feed BERT with our input text, BERT will output a rich vector representation for the text as well as for each token (these outputs are sometimes called contextual-embeddings because they change based on the context of the token). We can then use these representations of the text as a whole and of each token to train other models.

Fine-tunning, on the other hand, consists in adding additional neural layers on top of BERT (sometimes called heads) and training the whole model (BERT and the head) end-to-end. BERT will start from its pretrained weights which already encode plenty of knowledge about language, which helps by substantially cutting down training time and increasing performance.

Using language models especially helps when the training data is not very large (Brown et al., 2020). In such cases, language models can boost performance as they introduce a lot of information (encoded in the weights) that was learnt during the pretraining phase. The pretraining is always a self-supervised learning scheme that allows unannotated text to be used, and thus massive amounts of data can be used (e.g. the whole Wikipedia).

2.1.2 Language models for topic classification

For sentence classification, we need to obtain a representation of the sentence to then classify (using some model). We can get a representation of the sentence in different ways using a BERT-like language model. For our experiments we consider three cases: (1) using the embedding of the special `[CLS]` token (this token is intended to encode a sentence representation), (2) averaging the contextual embeddings of all tokens or (3) aggregating the contextual embeddings of all tokens using another model. These three methods can be performed both in an online or offline strategy.

Architecture:
BiLSTM(128, 0.1% dropout) ∘
BiLSTM(128, 0.1% dropout) ∘
Dense(128, 0.1% dropout) ∘ ReLU∘
Dense(64) ∘ ReLU∘
Dense(6) ∘ Softmax
Optimization:
AdamW(10^{-3} lr, 10^{-4} weight-decay)
Batch size 16
No label smoothing
Categorical cross-entropy
Early stopping (max dev acc)
Input:
GloVe (trainable) ∘ Dense(300)
Padding to 300 tokens

Table 1: Description of the RNN (BiLSTM) baseline used for most of the experiments.

3 Experiments

We first investigate the recurrent neural networks, then using BERT-like models, follow with an analysis of sensitivity to perturbations and conclude with a comparison of the various model families.

3.1 RNNs

There are three components to the LSTM unit: input gate, forget gate and output gate. Together they aim to prevent overrepresentation of the last step and vanishing alleviate gradients. The GRU cell is similar in design but with smaller complexity for faster training. The LSTM computation, given the previous hidden state h_{t-1} and the current input x_t is as follows:

$$
\begin{aligned}
i_t &= \sigma(w_i[h_{t-1}, x_t] + b_i) && \text{(input gate)} \\
f_t &= \sigma(w_f[h_{t-1}, x_t] + b_f) && \text{(forget gate)} \\
o_t &= \sigma(w_o[h_{t-1}, x_t] + b_o) && \text{(output gate)} \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tanh(w_c[h_{t-1}, x_t] + b_c) \\
& && \text{(cell state update)} \\
h_t &= o_t \cdot \tanh(c_t) && \text{(output/hidden state)}
\end{aligned}
$$

Apart from the specific model part which we explore in the following sections, we keep the following configuration (shown in Table 1) constant. We intentionally start with an otherwise strong model and not the instructed baseline because we are interested in the influence of design choices on models close to the peak performance.

3.1.1 Pretrained Embeddings

In this part, we explore the effects of using different embedding strategies: using pre-trained word embeddings, fine-tuning the embeddings, adding a dense layer and regularization. The results for various configuration are shown in Table 2. The dense layer had 300 hidden units, bias and no activation. Regularization of the embedding vectors was given by $10^{-5} \cdot L_1 + 10^{-4} \cdot L_2$.

Embeddings	Train	Dev
GloVe	89.6%	91.8%
GloVe + dense	95.2%	93.4% ⋆
GloVe train.	94.5%	92.6%
GloVe train. + dense	97.4%	92.0%
GloVe train. + reg.	91.6%	89.2%
GloVe train. + dense + reg.	96.1%	90.1%
Random train.	99.3%	87.2%
Random train. + dense	97.6%	87.0%
Random train. + reg.	97.1%	88.4%
Random train. + dense + reg.	97.9%	89.6%

Table 2: Accuracies of various embedding configurations with RNN-based model.

Adding a dense layer on top of the embeddings had a very positive effect on the model performance. The reason for overfitting when embeddings are fine-tuned is that it allows the model to memorize individual reviews more. Regularization of the embedding vectors was not sufficient to resolve this. The reason this issue does not arise with the dense layer is that it is shared among all embedding vectors.

The dev accuracy was lower when starting with random embeddings. Naturally, the training takes longer time which also results in severe overfitting. Unfortunately, the regularization fails here as well. For this reason, adding a dense layer only made it worse because the model could train ever for a shorter amount of time before overfitting.

3.1.2 Recurrent Units

We examine the effect of three standard recurrent units on the model performance: LSTM, GRU and vanilla RNN. The rest of the configuration is left intact and as such, these units are used bidirectionally. Results are shown in Table 3. GRU appears to be the best choice because of its high accuracy and training speed. It also has a higher training accuracy which suggests that further regularization techniques could improve the dev accuracy as well. We stick to LSTM because that seems to be the requirement of the assignment.

Embeddings	Train	Dev
LSTM	94.5%	92.6%
GRU	97.9%	93.2% ⋆
vanilla RNN	84.5%	72.6%

Table 3: Accuracies of various recurrent units.

3.1.3 Layers

We explore various different architectures, namely the number of recurrent layers, bi-directionality and aggregation. For multiple recurrent layers, it is necessary in code to set `return_sequence = True` so that the intermediate hidden states are made accessible to the next RNN layer. Results for different number of recurrent layers (with LSTM) are shown in Table 4. For more than two hidden layers, the performance drops, presumably because of the vanishing gradients and losing signal through backpropagation. This was confirmed also for the higher number of layers (not shown for clarity). A possible solution to this would be to add residual connections Making the computation bidirectional helps over the single pass in all cases. Importantly, processing the text in reverse leads to even stronger improvements. A common trick for machine translation systems with a one-directional encoder would be to go in reverse because then the first tokens, which are the most important words in the sentence, would be better represented in the single vector. A similar rationale can be also applied for review topic classification.

Layers	Train	Dev
1	96.6%	90.0%
2	98.7%	88.4%
3	30.5%	31.8%
1 + rev.	99.1%	93.6% ⋆
2 + rev.	98.8%	93.2%
3 + rev.	98.1%	91.2%
1 + bi	99.5%	92.4%
2 + bi	94.5%	92.6%
3 + bi	93.9%	92.0%

Table 4: Accuracies of different number of recurrent layers.

After the embeddings/previous hidden states are processed by the recurrent units from both directions, the resulting two vectors need to be merged in some way. We list the different options in Table 5. The *concat* operation gains an unfair advantage in that the output is a vector twice as long which also means more parameters when it is being processed by the dense layer. Despite that, most of the operations are balanced and with only minor differences in dev accuracies.

Operation	Train	Dev
Sum	97.1%	91.8%
Mul.	96.6%	92.2%
Concat	94.5%	92.6% ⋆
Avg.	98.4%	92.4%

Table 5: Accuracies of different merge operations for bidirectional recurrent layers.

3.1.4 Dropout

In the current architecture, there are three places for dropout which we consider (see Table 1 for reference): (1) after the first dense layer after the recurrent layer, (2) dropout of the inputs to the recurrent layers and (3) dropout of the hidden state transformation (recurrent dropout). We check each of the settings with 0.0 and 0.1 dropout rate and present the results in Table 6.

Dense	RNN	RNN rec.	Train	Dev
0	0	0	98.7%	93.6% ⋆
0	0	0.1	93.0%	91.8%
0	0.1	0	94.2%	91.2%
0	0.1	0.1	95.1%	91.0%
0.1	0	0	95.5%	92.2%
0.1	0	0.1	94.7%	90.6%
0.1	0.1	0	99.1%	92.4%
0.1	0.1	0.1	94.7%	91.6.%

Table 6: Accuracies of different dropout settings for RNN.

At the first glance, dropout usage does not seem to systematically help with overfitting, at least when early stopping is used. Other regularization techniques, such as weight penalties or parameter tying could be more beneficial in this case. Similar to results of Melis et al. (2017), the recurrent (hidden state) dropout does not help in any situation. Furthermore, it is not implemented by CudNN (as of version `8.2`) and because of that, the computation is much slower. It is important to note that the differences between different configurations could be the results of noise and that cross-validation would yield more trustworthy results. We did not engage in it because of the computational resources limitations.

3.1.5 Optimizers

Note that we keep the batch size constant (to 16) and this may not be the optimum for all of the optimizers. An exhaustive grid-search would be beneficial here as well though we are limited by compute time availability. The results for various optimizers and learning rates (close to optimal for each optimizer) are shown in Table 7. We consider the standard stochastic gradient descent (SGD), RMSProp (Tieleman et al., 2012), Adam (Kingma and Ba, 2017) and Adam with weight decay. The results demonstrate the importance of selecting a good learning rate for specific optimizers. SGD, although with worse performance than Adam, suffers less from overfitting. Overall Adam without weight decay performs best but is severely overfitted.

Optimizer	WD	LR	Train	Dev
SGD	–	10^{-2}	89.4%	91.4%
SGD	–	10^{-3}	43.9%	50.0%
RMSProp	–	10^{-3}	94.7%	92.4%
RMSProp	–	10^{-4}	97.1%	93.2%
Adam	0	10^{-3}	100%	93.2%
Adam	0	10^{-4}	98.0%	93.8% ⋆
Adam	10^{-4}	10^{-3}	94.5%	92.6%
Adam	10^{-4}	10^{-4}	96.3%	93.4%

Table 7: Accuracies of different optimizer settings for RNN (WD = weight decay).

3.2 Language models

For the experiments with language models, we mostly work with BERT and try different strategies to integrate BERT in the pipeline. We experiment with both online and offline approaches. For online approaches, we experiment with different (neural) heads that make the whole trainable end-to-end. For offline approaches, we employ the same end-to-end approaches but with the weights of BERT frozen and other non-neural approaches that only use BERT as a feature extractor/embedder. We later make use of other more

modern language models for comparison, but we do not experiment with them to the same lengths because of time constraints. In a similar fashion, we restrict much of our exploration with BERT to few runs and little hyperparameter tuning due to the big amount of computational resources that these models require.

3.2.1 Initial hyperparameter exploration

We make an initial hyperparameter exploration using the default method for online sentence classification with BERT, which consists in adding a neural head to the special `[CLS]` token. We first set a baseline with some reasonable hyperparameters which are shown together with the best hyperparameters in Appendix A. We explore along the dimensions of **max sequence length**, **epochs**, **batch-size** and **learning rate**. The number of epochs is not important as long as more than one epoch is used. Adding more epochs increases the train-set performance but has no measurable effect (negative or positive) otherwise in development and test performance. Since more epochs do not result in overfitting, and taking into account the recommendation of the authors of 2-4 epochs we settle for a reasonable epoch count of 3. [vz: What authors?] Figure 1 shows the scores of different runs with respect to their training epoch count.

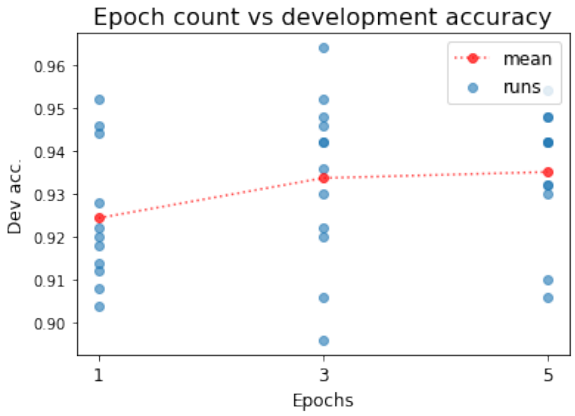

Figure 1: Development performance against training epochs. The variance is due to each run having different hyper-parameters

Performance is most sensitive to batch size and the maximum length of the input sequence. Although the improvement in dev accuracy is small, we can conclude from experiments not presented that bigger batch-size and longer maximum sequence length help with performance. Figure 2 shows how performance scales when increasing both batch-size and the maximum sequence length.

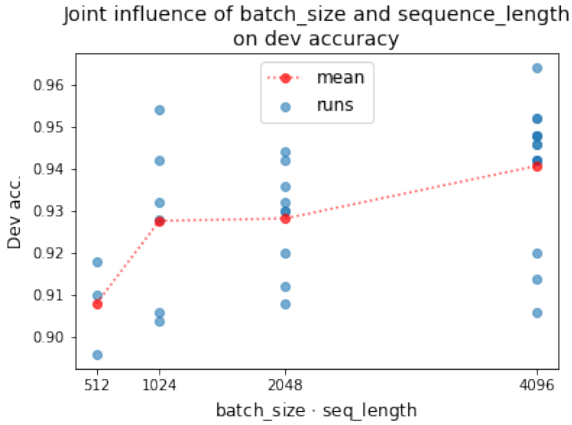

Figure 2: Join influence of maximum sequence length and batch size. The extrapolated mean clearly shows a positive trend for higher batch-size/sequence-length products.

Both these settings fight for GPU memory, which makes it harder to show that each of these variables contributes on its own. Thus a trade-off must be found that is suitable for the training setup. In our case, we found out the batch size of 16 and maximum sequence length of 256 to be a good compromise as indicated in Figure 3.

Finally, learning-rate (lr) had little effect on the final performance except when the lr was too high (higher than 10^{-4}), in which case the model would fail to train and would have a baseline performance of 17% (which corresponds to the performance of a random classifier in 6-way classification) in both train and development.

Anything lower than 10^{-4} had little effect on performance after the established 3 epochs. We also tried Polynomial weight decay which appears to slightly improve performance, but can not assert that with confidence since we only ran a limited set of experiments with it (too many parameters to tune).

We picked the hyperparameters with the highest development score (which might not have the highest test score) so as to not bias the model selection in favour of test performance.

The hyperparameters for the final model are shown in Appendix A and the performance in Section 4. Figure 4 shows the confusion matrix of the errors the best model made on the development set. The sparsity of the matrix gives a good picture of the performance that the model is able to

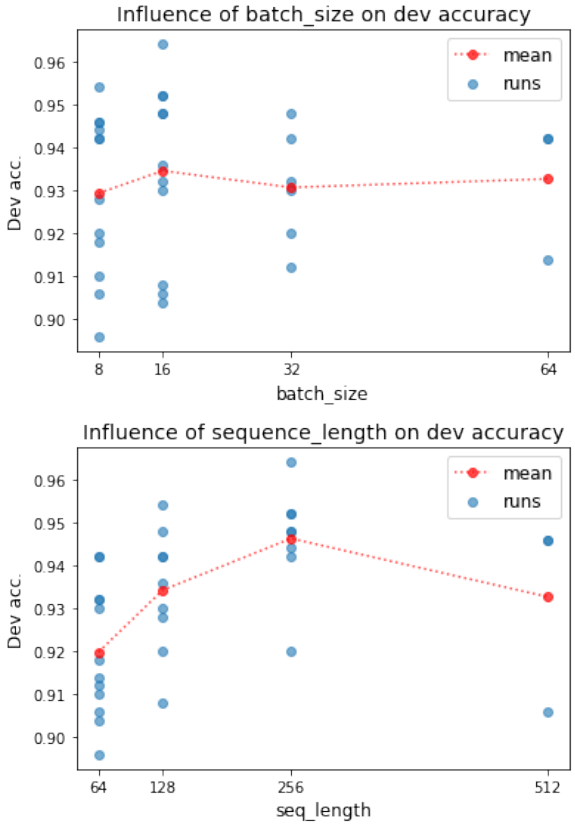

Figure 3: Individual contributions of batch-size and sequence length. Note that higher values of one variable require lower values of the other one to fit in memory. The product of both should not exceed 4096.

achieve.

3.2.2 Different Ways of Using BERT

In contrast to how BERT performs when using the standard fine-tuning, we show how other strategies compare (both online and offline). We compare the three aforementioned (Section 2.1.1) ways of embedding sentences with BERT. For the embedding aggregation strategy, we use an LSTM and a BiLSTM. Table 8 shows performance scores for different sentence embedding strategies with BERT, using the best hyper-parameters found in the previous step (which may not be the best for these alternate approaches). For offline methods a larger learning rate (10^{-3}) and more epochs (9) were needed to reach convergence. We can conclude that online methods have a noticeable lead in performance. However they come at the cost of higher memory consumption and train time requirements.

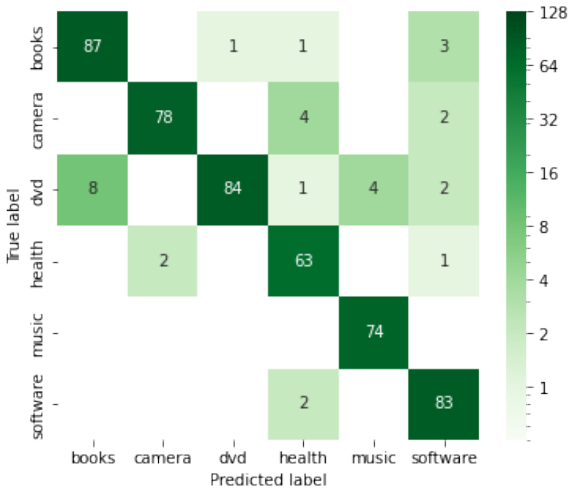

Figure 4: Confusion matrix of errors produced by BERT with the best hyper-parameters.

Approach	Dev acc.	Test acc.
Online		
CLS	96.4%	93.8%
Avg.	96.0%	93.8%
LSTM	94.6%	92.8%
BiLSTM	94.4%	94.4%
Offline		
CLS	84.0%	84.4%
Avg.	92.0%	91.4%
LSTM	93.2%	91.8%
BiLSTM	94.4%	93.6%

Table 8: Accuracy scores of different approaches of using BERT.

3.2.3 Other state-of-the-art language models

We mostly focused on BERT for this assignment but there are other relevant language models. In Table 9 we present some preliminary results from using other language models. Comparatively we spent less time working with these models and as such are not able to pick the best hyper-parameters. The results should thus be taken as a baseline and not indicative of the highest potential performance of these models. The parameters used are the best hyper-parameters for BERT except for the learning rate which was fixed instead of scheduled for simplicity.

From Table 9 we can see that all state-of-the-art models perform very similarly, at least in a baseline configuration. This suggests that it could be

Model	Dev	Test
BERT (baseline)	93.6%	93.4%
BERT (best h.param)	96.4%	93.8%
ROBERTA	95.8%	93.6%
DistilROBERTA	95.4%	92.2%
ALBERT	96.0%	91.8%

Table 9: Accurarcies of different state-of-the-art language models.

the case that we are approaching the limit of the attainable performance on this task given the restrictions of possibly faulty annotations.

3.3 Sensitivity to Perturbations

Creating adversarial examples for complex NLP models is difficult because of the discrete input space. It would be possible to compute gradients with respect to the inputs (embedding vocabulary) and replace the input word with the one with the highest loss. While a similar approach would be possible in computer vision where making small changes to pixels can go undetected to a human eye, one step in computing an adversarial example, in this case, would entail completely changing the whole word which could be easily detected. For simpler models such as SVM, it is possible to compute the mapping between words and their influence on the computation (Zouhar and Arguinzoniz, 2021b). Such a thing is not possible with models that are sensitive to word order because then the presence of a specific word does not affect the computation in a straightforward way. Nevertheless, we wish to create a first step in the exploration of the possible adversarial attacks and do so by perturbing the input data. The data is generated once for the whole training though it would certainly be possible to generate this differently during each epoch. We compare the effect across the baseline RNN and BERT models. The RNN has the advantage that it has no prior in terms of how a well-formed sentence should look like which is not true for the pre-trained BERT. On the other hand, BERT has an attention mechanism that may disregard the positional encoding. Note that more complex methods for attacking BERT-based models for classification exist (Garg and Ramakrishnan, 2020; Li et al., 2020; Sun et al., 2020). Our results are presented in Table 10.

When adding new tokens or changing the words, we chose the new token uniformly from the whole training vocabulary (e.g. `the` has the same probability of being selected than `frustrating`). We list an example of the severity of the perturbations which make it very hard for a human to correctly classify:

Original (health):
`this brush is worth about 10 bucks . anything more and you are throwing money down the drain`
Shuffle:
`drain down bucks about . you anything 10 throwing brush and the more are . is this worth money`
Delete 50%:
`this bucks anything more you are throwing down drain .`
Add 50%:
`this brush is ] worth about ¿ 10 { bucks. anything more 4 and you a 9 are throwing a money down ¿ the drain ⋆.`
Change 50%:
`c brush is + s # bucks . anything · ! ⋆ are throwing money . & drain .`

Changing the tokens seems to be the most detrimental to the model performance while shuffling or adding 25% hurts the accuracy the least. There seem to be no large differences between models when measured by the dev accuracy.

Operation	RNN	BERT
Baseline	92.6% (0%)	93.6% (0%)
Shuffle	91.4% (−1.2%)	91.8% (−1.8%)
Delete 25%	89.8% (−2.8%)	90.4% (−3.2%)
Delete 50%	87.2% (−5.4%)	89.4% (−4.2%)
Add 25%	92.0% (−0.6%)	91.6% (−2.0%)
Add 50%	89.6% (−3.0%)	93.6% (−0.0%)
Change 25%	89.0% (−3.6%)	91.6% (−2.0%)
Change 50%	87.8% (−4.8%)	88.4% (−5.2%)

Table 10: Accuracies of models on training and dev data with different perturbations. Value in brackets is the absolute change in percentage compared to dev accuracy on unmodified data.

3.4 k-NN Classification

We embed the articles to get a fixed-vector representation. On this, we construct a simple k-NN classification model. Each model was grid-searched with odd k from 1 to 17, voting strategy (uniform, distance) and whether the dimensions are centred and normalized (center). We

consider TF-IDF features as a baseline though it is important to note that straight comparison is not fair because, without pruning, the resulting number of features is 32214. We, therefore, include versions with 300 and 768 dimensions that replicate the dimensionality of GloVe and BERT-like embeddings. High dimensionality, despite leading to better results, creates issues with the complexity of nearest neighbour search and also the memory requirements. For DPR we use the document (context) embedding and not the query embedding because of marginally higher performance. The results are presented in Table 11 together with the best parameters on the dev set.

Model	Best parameters	Accuracy
TF-IDF 32k	k=11, dist.	80.0%
TF-IDF 768	k=15, dist.	69.6%
TF-IDF 300	k=15, dist.	69.6%
GloVe Max.	k=3, dist.	42.0%
GloVe Avg.	k=5, unif., center	37.6%
BERT CLS	k=11, dist., center	55.8%
BERT Avg.	k=3, dist., center	81.8%
S-BERT CLS	k=3, dist., center	74.8%
S-BERT Avg.	k=5, dist., center	81.8%
DPR CLS	k=15, dist., center	91.8% ⋆
DPR Avg.	k=3, dist., center	87.4%

Table 11: Development accuracy of k-NN classification model based on different article embeddings.

Even though TF-IFD with a comparable number of features provides a strong baseline, the computation is made even slower by having to retrieve a higher number of neighbours to achieve the best results. GloVe performed poorly and we hypothesise that it was caused by the vectors being not distinguishable enough, which is a consequence of using non-contextual word embedding. BERT CLS had unsurprisingly low performance because it is trained to represent the classification in next sentence prediction. SentenceBERT performed better because its training objective is to directly compute better sentence embeddings and is, therefore, better suited for semantic similarity search. Finally, DPR CLS outperforms all other models because it is trained to learn the metric space (with contrastive learning loss) and is directly used for information retrieval applications, such as for question answering.

3.5 Model Comparison

We select the best configuration from every model family (RNN, BERT-like, kNN, Logistic regression, SVM) and present the results in Table 12. For this, we evaluate also on the test set which is the only time we used this set. The specific configurations are detailed in Appendix A. Logistic regression, naïve Bayes and the SVM classifier use the TF-IDF vectorizer with default settings. The fine-tuned BERT models outperform all the others, especially the simpler ones.

Model	Train	Dev	Test
RNN	97.6%	93.6%	91.2%
BERT	97.6%	96.4%	93.8% ⋆
kNN (DPR CLS)	100%	91.8%	88.8%
Logistic regression	97.6%	92.0%	89.0%
Naïve Bayes	97.0%	91.0%	89.0%
SVM Classifier[2]	99.7%	91.0%	89.0%

Table 12: Comparison of best configurations from each presented model family and also our previous work.

4 Conclusion

In this report, we examined two large families of models for review topic classification: RNN-based (GRU, LSTM, vanilla RNN) and BERT-based. We find that both families are quite resilient to even high input space perturbations. We have also demonstrated a large range of model performance depending on the hyperparameters (most importantly optimizer and learning rate).

For BERT-like language models we explored the effect of hyper-parameters and found out about the importance of batch-size and sequence length. We also explored different way in which BERT can be used for a classification task. Finally we looked at other more modern BERT-like language models.

Future work. We did not perform an in-depth analysis on the model confidence and its dependency on the input perturbations. Even with otherwise black-box access, having the information to the posteriors could be useful in determining how susceptible the model is to adversarial noise injections.

References

[Brown et al.2020] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

[Cho et al.2014] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[Devlin et al.2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

[Garg and Ramakrishnan2020] Siddhant Garg and Goutham Ramakrishnan. 2020. Bae: Bert-based adversarial examples for text classification. *arXiv preprint arXiv:2004.01970*.

[Hochreiter and Schmidhuber1997] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Htut et al.2019] Phu Mon Htut, Jason Phang, Shikha Bordia, and Samuel R. Bowman. 2019. Do attention heads in bert track syntactic dependencies?

[Karpukhin et al.2020] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

[Kingma and Ba2017] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A method for stochastic optimization.

[Li et al.2020] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. Bert-attack: Adversarial attack against bert using bert. *arXiv preprint arXiv:2004.09984*.

[Melis et al.2017] Gábor Melis, Chris Dyer, and Phil Blunsom. 2017. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

[Pennington et al.2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Reimers and Gurevych2019] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

[Rumelhart et al.1986] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

[Sun et al.2019] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer.

[Sun et al.2020] Lichao Sun, Kazuma Hashimoto, Wenpeng Yin, Akari Asai, Jia Li, Philip Yu, and Caiming Xiong. 2020. Adv-bert: Bert is not robust on misspellings! generating nature adversarial samples on bert. *arXiv preprint arXiv:2003.04985*.

[Tieleman et al.2012] Tijmen Tieleman, Geoffrey Hinton, et al. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[Wang et al.2019] Dan Wang, Jibing Gong, and Yaxi Song. 2019. W-rnn: News text classification based on a weighted rnn. *arXiv preprint arXiv:1909.13077*.

[Yogatama et al.2017] Dani Yogatama, Chris Dyer, Wang Ling, and Phil Blunsom. 2017. Generative and discriminative text classification with recurrent neural networks. *arXiv preprint arXiv:1703.01898*.

[Zhou et al.2016] Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. 2016. Text classification improved by integrating bidirectional lstm with two-dimensional max pooling. *arXiv preprint arXiv:1611.06639*.

[Zouhar and Arguinzoniz2021a] Vilém Zouhar and Edu Vallejo Arguinzoniz. 2021a. Assignment 1: Comparison of baseline models for sentiment and topic classification. Not published. Learning from Data assignment.

[Zouhar and Arguinzoniz2021b] Vilém Zouhar and Edu Vallejo Arguinzoniz. 2021b. Assignment 3: Analysis and parameter search for sentiment classification with svm. Not published. Learning from Data assignment.

A Best Model Configurations

Architecture:
BiGRU(128, 0.1% dropout) ∘
BiGRU(128, 0.1% dropout) ∘
Dense(128, 0.1% dropout) ∘ ReLU∘
Dense(64) ∘ ReLU∘
Dense(6) ∘ Softmax
Optimization:
Adam(10^{-4} lr)
Batch size 16
No label smoothing
Categorical cross-entropy
Early stopping (max dev acc)
Input:
GloVe (trainable) ∘ Dense(300)
Padding to 300 tokens

Table 13: Description of the best RNN-based model (dev set).

Hyper-parameters	**Baseline**	**Best**
Epochs	3	3
Batch-size	16	16
Max sequence length	128	256
Learning-rate	Fixed	Polyn.
Initial learning-rate	$5 \cdot 10^{-5}$	$5 \cdot 10^{-5}$
Final learning-rate	$5 \cdot 10^{-5}$	$5 \cdot 10^{-7}$
Decay steps	0	1000
Polynomial degree	NA	1

Table 14: Hyper-parameters of the baseline and the best performing BERT model fine-tuning (dev set).