194.048 Data-intensive Computing 2020S, Group Project (Group 4)

HÜBLER MORITZ, 01426077
DARMANOVIC FILIP, 01527089
ARFAOUI GHAITH, 01435404
GANDER ARMIN, 11848230

This report documents the approach of our group to participate at the 2020 Twitter-RecSys challenge. It explains our difficulties using BERT word embeddings, due to which we finally chose a pre-processing pipeline including TfidfVectorizing, StringIndexing and OneHotEncoding. Using this setup we were able to create valid submissions using Logistic Regression and Random Forest models. For the latter we implemented a grid search for hyperparameter optimization to improve the prediction results. The documentation ends with a discussion of the results and a short summary of our findings.

ACM Reference Format:
Hübler Moritz, Darmanovic Filip, Arfaoui Ghaith, and Gander Armin. 2020. 194.048 Data-intensive Computing 2020S, Group Project (Group 4). 1, 1 (July 2020), 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The task of the group project was to participate at the 2020 Twitter-RecSys challenge, submit at least one non-trivial submission and to document our approach. For this we registered on Twitter for an developer account and requested access to the twitter API following the instructions on the RecSys homepage[1].

The goal of the RecSys challenge is to predict different kinds of user engagement (Likes, Replies, Retweets and Retweets with comments) with content (Tweets) on Twitter.

2 THE DATASET

The dataset consists of ca. 200 million public engagements on twitter, collected from a span of 2 weeks. There is a special focus on user privacy, for which the dataset is being updated regularly. Whenever a Tweet or user is deleted on the platform it should not occur in the dataset anymore. For this purpose lists of Tweets and users which should be removed can be downloaded from the RecSys homepage. Additional to the public engagements another 100 million pseudo negative engagements where created. These features represent examples of content with which users did not interact, although it is unknown if a user saw the content and chose to not engage with it or if it was not seen at all.

The specifics of the the Tweet, user & engagement features can be taken from the challenge paper[2]. The Tweet text is tokenized and embedded where each integer corresponds to the index of the token in the embedding.

The dataset consists of 24 attributes that can be summarized in the following way:

- Text attributes: list of tokens representing text data related to the tweet.
- Categorical attributes representing some features of the tweet (e.g language, tweet type ...)
- Boolean attributes representing user accounts features
- Timestamp attributes representing user and tweet related features.

The 4 target features we had to predict were:

- Like engagement timestamp
- Reply engagement timestamp
- Retweet engagement timestamp
- Retweet with comment engagement timestamp

3 PRE-PROCESSING

Several transformation are applied to the raw data in order to make it suitable for training. In the following subsections, the different transformations are defined for each type of attribute.

3.1 Text tokens

According to the challenge paper, the tweet text was turned into BERT tokens in order to further anonymize the dataset. However, it is possible to turn the tokens back into words using Huggingface's pytorch BERT library[3], and extract most of the original tweet content. This means that the challenge organisers probably used the same library, and in turn the same dictionary for encoding. For that reason, this anonymization scheme turned out to be flawed.
Privacy aside, the presence of BERT tokens led us to explore this novel technique in natural language processing as a step in our pre-processing pipeline. Essentially, it is a neural network developed by Google in 2019[4], which takes in words encoded as tokens, as well as some extra inputs like a binary mask that marks padding tokens in the input, and returns for each token a vector, referred to as a vector embedding. This vector maps each word of the input to an N-dim space (default is 768) (Fig. 1), with the idea that semantically similar words will be mapped to vectors close to each other in the feature space. While this is a novel idea by itself, and is possibly regarded as a huge break-trough in NLP research, BERT takes it a

[1] https://recsys-twitter.com/

Authors' addresses: Hübler Moritz, 01426077; Darmanovic Filip, 01527089; Arfaoui Ghaith, 01435404; Gander Armin, 11848230.

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/1122445.1122456

[2] https://arxiv.org/abs/2004.13715
[3] https://huggingface.co/transformers/index.html
[4] https://github.com/google-research/bert

step further by taking context into account, and thereby mapping a whole sentence to a feature space.

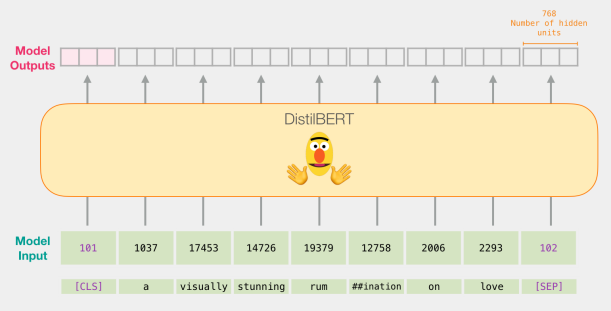

Fig. 1. The simplified IO model of the (Distil)BERT neural network. The corresponding output to the [CLS] token contains the unified embedding of the whole text input

One further aspect of BERT that makes it the prefect candidate for our use case is its multilingual dictionary support. The challenge paper points out how many users interact with tweets in multiple languages, even inside a single tweet. In theory, BERT will embed the same word across different languages to the same, or very similar vector. This functionality should enable seamless semantic feature extraction for multi-lingual inputs.

Seeing how BERT is a recent development, integrating it into our spark pipeline proved to be a big challenge. We had the choice of either using the aforementioned huggingface pytorch library, which is incompatible out-of-the-box with spark, but has a mature feature set, or John Snow Labs' spark-nlp library[5], which is spark-native, but is very limited. We tried the former first, without success, and in fact, at the time of writing this paper (30.06.2020), it was confirmed to us by a John Snow Labs' rep on their official slack channel, that currently custom dictionaries and and pre-processed tokens are not supported. This would mean that we would have to use the pytorch library to turn the tokens back into text, clean up the output, and then feed it back into spark-nlp. For this reason, we tried using only the pytorch library, but that came with its own set of challenges.

In order to connect a third-party python library to spark, we have several choices:

- UDF
- .map()
- Custom Transformer.

According to an online article that benchmarked each strategy[6], the .map() way is the fastest, but it too should be avoided, as it's basically a black box. What we soon realised however, is that python itself is a hurdle, because running such an external library requires the native scala code to run a python interpreter inside the JVM's heap, which consumes a lot of memory and creates overhead. So much so in fact, that we immediately got out-of-memory errors when trying to run the pipeline. There were a few other UDF-implemented steps for creating the padding and the attention mask, but those were simple and were probably translated into native scala code, so they ran fine.

While the leaderboard was still open, we decided to scrap BERT and focus on other strategies, but during the presentation session, it was brought to our attention that one group in the first slot actually managed to use BERT in spark. Their slides didn't go too in-depth on the implementation, but they said they used the sparktorch[7] library to connect pytorch and spark. This is an open-source library maintained by one person and it seems to be fairly limited. After trying to use it, we still got OOM errors, and the group members haven't responded to our questions about their implementation at the time of writing.

To summarize, we were unsuccessful in using BERT in our pipeline, but expect it to be a major component in future NLP tasks, as soon as the tooling becomes more mature.

In addition to BERT embeddings, the following methods are also implemented:

- TfidfVectorizer
- CountVectorizer
- Word2Vec

While both CountVectorizer and Word2Vec were succesfully applied to a subset of the data, it was not possible to apply them on the full dataset due to memory issues. TfidfVectorizer is therefore used to transform both *text tokens* and *hashtags*. The number of features is selected to be as close as possible to the vocabulary size.

For the attribute *Present links*, instead of vectorizing the tokens representing links, a new attribute that contains the count of links in each tweet is created.

3.2 Categorical attributes

Transformations of categorical attributes depends mainly on the number of categories.

For *tweet type* and *language*, having 3 and 66 unique values respectively, label encoding is performed using the pyspark implementation StringIndexer.

The attribute *Present media* contains a list of labels representing the media that are present in the tweet. The allowed values are: GIF, Photo and Video. For every instance, multiple media of the same type can occur. Therefore, three columns representing the meida types are created where in each column the count of how many times the specific media label appears in the list is saved.

3.3 Boolean attributes

As described in section 2, boolean attributes provide information about the users accounts and the relation between them.

For *Engaged with user is verified*, *Engaging user is verified* and *Engagee follows engager* onehot encoding is used by applying OneHotEncoder from pyspark.

3.4 Target Attributes

A binary encoding is applied to all target attributes. Since these attributes taken timestamps as values, 1 is assigned when such a value is present otherwise 0 is assigned.

[5]https://nlp.johnsnowlabs.com

[6]https://medium.com/@fqaiser94/udfs-vs-map-vs-custom-spark-native-functions-91ab2c154b44

[7]https://github.com/dmmiller612/sparktorch

3.5 Scaling

MinMaxScaler from spark is used to scale numerical , timestamp and some of the transformed attributes. The scaled features are: *Media GIF*, *Media photo*, *Media video* (*attributes created out of Present media*), *tweet timestamp*, *Engaged with user follower count*, *Engaged with user following count*, *Engaged with user account creation*, *Engaging user follower count*, *Engaging user following count* and *Engaging user account creation*

4 TRAINING

4.1 Collaborative filtering using matrix factorization

4.1.1 General model description. Collaborative filtering is a powerful way to generate recommendations for users. With collaborative filtering, recommendations are given exclusively based on observed user behavior. No access to profile data or content is required. This technique is based on the assumptions that users interact with elements in a similar way and users with shared preferences are likely to respond in the same way to the same items. Technically, collaborative filtering can be solved by matrix factorization.

The matrix factorization method assumes that there are a number of attributes that are common to all elements. In addition, the method assumes that each user has a different level of exposure for each of these attributes, regardless of the elements. In this way, the rating of a user can be approximated/estimated. To do this, the user's levels of engagement are summed for each item and weighted by the degree to which the item has the attribute in question. These attributes are called latent factors.

The latent factors are two sets of values (one set for the users and one set for the item) that describe the user and the item.

4.1.2 ALS method - Method of alternating least squares. The alternating least squares method of matrix factorization is an iterative method for determining the optimal factors/matrices X and Y that most closely approximate the ratings matrix R. During each iteration, one of the row or column factors is kept constant. The other is calculated by minimizing the loss function with respect to the other factor. In this project, we used sparks ALSmodel to implement this method of matrix factorization.

4.1.3 Implementation and outcome. We trained the model with 50 percent of the total training dataset (60 million observations). Unfortunately, the model did not yield good results. In total, it was only able to predict ratings for 25000 out of 12 million entries in the validation dataset. The remaining user-item pair's ratings were predicted with NaN.

The bad performance of the ALS model can be attributed to the great sparsity of the training dataset. Usually, datasets for recommender systems are sparse since they contain a much greater number of users than item. Additionally, users rate only a very subset of the total number of items. This, compared with the fact that we were only able to train the ALS model with 50 percent of the total training dataset due to computational effort restrictions, made this model unusable for our task.

An approach to solve the problem of sparsity would be to make ratings matrix denser. This could be achieved by clustering the users into a fixed number of buckets and then predicting a rating of each bucket. Using this method, the data is made denser, which should improve the ALS models performance enormously.

Alternatively, it would be an option to use newer methods like the B-NMF [8](blocks-coupled non-negative matrix factorization) algorithm. With this algorithm: (1) the reconstruction performance of matrix of extreme sparseness is improved as a result of blocking the matrix and modeling based on full use of the coupling between blocks; (2) the coupling between different blocks is ensured via a coupling mechanism that imposes constraints on consistency as the matrix is decomposed.

Finally, however, we decided to not delve into the issue of resolving the data sparsity problem due to a lack of time. Instead, we continued with another type of recommendation system.

PRAUC Retweet	PRAUC Reply	PRAUC Like	PRAUC RT with comment
0.1759	0.0577	0.5888	0.0116
RCE Retweet	RCE Reply	RCE Like	RCE RT with comment
0.1726	3.8712	0.1542	0.3598

Table 1. Logistic regression submission (PRAUC and RCE)

4.2 Logistic regression

4.2.1 Model setup and outcome. After the ALS model was not able to produce acceptable results, we continued our work on the Recsys challenge 2020 with a logistic regression model. For this, we used the final preprocessing as described in 3. In order to include information about the ID-attributes tweet id, engaging user id and engaged user is as well, we hashed those into 100 different buckets as in the baseline method described in the Recsys challenge 2020 paper's baseline method [9]. Finally, the preprocessing for the logistic regression model was completed after creating the feature vectors using sparks StringIndexer, OneHotEncoderEstimator and VectorAssembler. The resulting features vectors served as the input of the Logistic regression model.

Again, we were not able to train the model using the full training dataset due to the great computational effort. Instead, we used only 20 percent of the training data. The logistic regression model was implemented using sparks LogisticRegression. The parameters we chose were the following: aggregationDepth of 2, L2 penalty (elasticNetParam = 0), maximum iterations of 100 and a regularization parameter of 0. Using this configuration, we were able to produce a valid submission for the Recsys challenge 2020. The results can be seen in table 1. At the time of submission, our entry to the Recsys challenge 2020 was at place 89.

4.3 Random forest

4.3.1 Model setup and outcome. For the random forest model we used the same setup and preprocessing as for the logistic regression. We even included the scaling which is not necessary for the random forest model.

For the implementation the RandomForestClassifier from spark was used. For each of the 4 target features one model was generated

[8]https://www.sciencedirect.com/science/article/abs/pii/S0925231217314467
[9]https://arxiv.org/abs/2004.13715

PRAUC Retweet	PRAUC Reply	PRAUC Like	PRAUC RT with comment
0.2396	0.0765	0.5918	0.0153
RCE Retweet	RCE Reply	RCE Like	RCE RT with comment
7.3878	8.2176	7.0574	3.1450

Table 2. Random forest submission (PRAUC and RCE) with default parameters

Response type	maxDepth	numTrees	maxBins
Retweet	20	10	32
Reply	20	10	32
Like	20	10	64
Retweet	20	5	64

Table 3. Grid search results

using only 0.1% of the training dataset. The parameters for the random forest where kept to default values (numTrees: 20, maxDepth: 5, maxBins: 32). We merged the predictions of the 4 models to create a submission for the leaderboard which gave us an overall score of 123. The detailed performance can be taken from table 2.

5 PARAMETER SELECTION

We perform a small grid search to optimize the parameters of the Random forest model. In order to make this feasible in term of computational effort, we use a small subset of the original training dataset of about 80.000 rows. Using sparks ParamGridBuilder and CrossValidator (3 folds), we varied three different parameters: maxDepth (10, 20), numTrees (5, 10) and maxBins (16, 32, 64). The results showed that using a maximum depth of 20 outperformed a maximum depth of 20 in every case. To facilitate the visualization of the grid search results, we therefore only show the results corresponding to a maxDepth of 20.

5.0.1 Grid search results. The results of the grid search corresponding to each response type of the Recsys challenge can be seen in figures 2 to 5.

Table 3 shows the best parameters for each response types' grid search.

The grid search shows that a higher number of trees leads to better results for almost all values for the maximum number of bins. Furthermore, also a higher number of bins tends to improve the average RMSE of the predictions. The model predicting the engagement type Retweet with comment poses the only small exception. The best average RMSE was achieved for a number of trees of 5 and a maximum number of bins of 64. Besides the combinations of parameters showed in 2 to 5, we tested the same combinations of numTrees and maxBins for a maximum tree depth of 10 as well. The results for that, however, were worse than for maxDepth of 20 without exceptions.

6 RESULTS AND DISCUSSION

Table 4 shows a comparison of the performance (PRAUC) of the models tested in the course of this project. The ALS model, as described

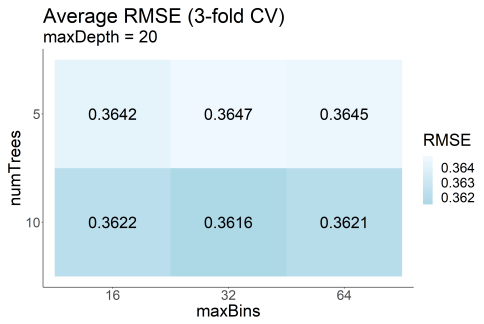

Fig. 2. Grid search retweet

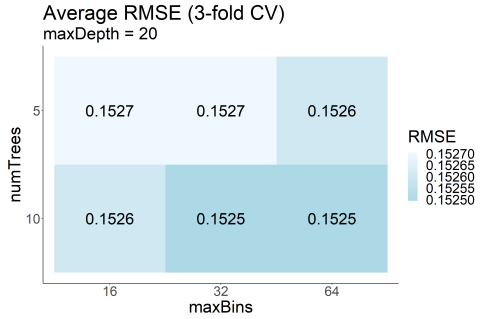

Fig. 3. Grid search reply

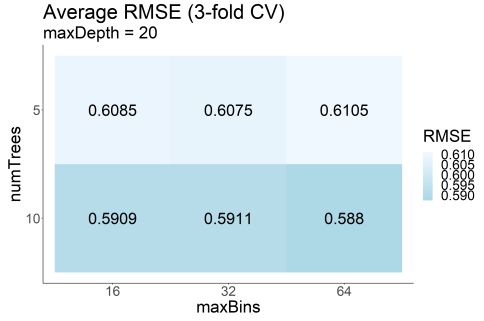

Fig. 4. Grid search like

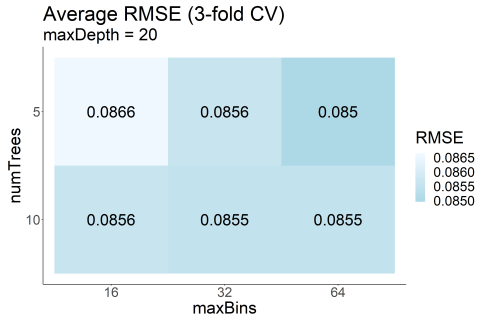

Fig. 5. Grid search retweet with comment

Model	PRAUC Retweet	PRAUC Reply	PRAUC Like	PRAUC RT with comment
Logistic Regression	0.1759	0.0577	0.5888	0.0116
Random forest (RF)	0.2396	0.0765	0.5918	0.0153
RF (optimized)	0.3335	0.0705	0.6673	0.0188

Table 4. Comparison of recommendation models

in 4.1.2, is not included as it was not able to produce acceptable predictions. Furthermore, it should be noted that the models were not trained on the same proportion of the original training dataset.

The Random forest model outperformed the Logistic regression model in every case. Especially of the engagement types Retweet and Like, the resulting PRAUC values are clearly better. This is quite surprising since only 0.1 percent of the data was used to train the RF model compared to 20 percent for the Logistic regression model. The optimized Random forest model yields even better results. Clear improvements in PRAUC can be seen especially for the engagement types Retweet and Like. Those are the engagement types with the highest ratio of positive implicit feedback. We assume that the parameter optimization was more efficient in these two cases since the binary class values (i.e. 0 and 1 for no engagement/engagement) were more balanced.

That the random forest without hyperparameter optimization performed quite well might have to do with the very small set of the trainig data it was trained on (only 0.1%). This might prevent the model from overfitting, resulting in a robust prediction.

7 CONCLUSION

The main challenge of this project work was the very time-consuming pre-processing of the raw data as well as the sheer size of the data we worked with. In order extract the most value of information out of the data we were given, it is important to be knowledgeable about the state-of-the-art methods in the field - natural language processing (NLP) in our case. Even though we came to the conclusion that the usage of BERT tokens was not feasible due to the very early stage of development of BERT word embedding in Spark, it gave us nonetheless a glimpse into the possibilities of current state-of-the-art NLP methods.

Besides NLP, the project also helped us greatly to get an overview over different recommender system models and their application. We conclude that the great sparsity of the training data posed great problems for collaborative filtering methods (ALS matrix factorization in our case). Better results were achieved using Logistic regression and Random Forest models to predict the likelihood of twitter users engaging with a tweet.

We were able to make two valid Recsys challenge submissions using a Logistic regression model and a Random Forest model. Even though we didn't use the full training dataset to train them, both model's submissions ended up around the middle of the leaderboard. After the submission deadline ended, we continued working on the project by optimizing the parameters of the Random Forest model using a small subset of the original training data. Judging from the comparison of results, we can conclude that the optimized Random Forest model performs best in terms of PRAUC for each engagement type but one. It is notable, however, that the original Random Forest model, which was trained using only 0.1 percent of the training data, performed very well in comparison. We suspect that the low amount of training data probably prevents the model from overfitting, which leads to surprisingly good results.