TabTransformer: Tabular Data Modeling Using Contextual Embeddings

Gaspari Michele 0001007249

February 27, 2022

Intro

The aim of this report is to present **TabTransformer** [5] architecture, in order to deal with learning task on tabular data. Boosting tree techniques represent the state-of-the-art in this field, outdoing standard multilayer perceptron architecture not only in performances but also from explainability point of view. In contrast, MLP can learn an internal representation of input data and they are suited for continual learning from streaming data, so neural network approach to tabular domain shall not be completely rejected for GBDT. Moreover, in a scenario wherer there's a lack of labeled data, semi-supervised techniques [6, 2] comes into the picture and both MLP and tree-based methods doesn't provide a satisfying answer to the problem.
TabTransformer exploit recent attention-based architecture [11] in order to cover this gap between neural networks models and tree-based methods in tabular domain, surely to match boosting methods performance but also to show effectiveness in understanding categorical information and robustness against missing or noisy data. Therefore, the purpose of this report is to clarify how TabTransformer can obtain such goals, both by inspecting its article and by showing architecture behaviour on **MovieLens** [4] dataset (dataset construction is presented in appendix after references), used to deploy a film rating system. Finally, it's also interesting to take a look at the approach followed by the authors of TabTransformer in order to perform semi-supervised learning on tabular data.

Software setup

In order to test the architecture on a different dataset respect to ones reported in article [5], a whole pipeline with data acquisition, data preprocessing, data visualization, model definition and performance measure, has been built using *PyCharm* (2021.2.4) with *Python* (version 3.7) and the following framework/libraries:

- *Pandas* (1.3.5), *Numpy* (1.21.5) and *Imbalanced-learn* (0.9) for data preprocessing
- *Matplotlib* (3.5.1) and *Seaborn* (0.11.2) for data visualization
- *Pytorch* (1.8.1) for neural network model definition and training
- *tab-transformer-pytorch* is the package that provide TabTransformer architecture
- *Scikit-learn* (1.0.1) for Support Vector Machine and Logistic Regression models, and also to perform k-fold cross validation
- *LightGBM* (3.3.2) to deploy Gradient Boost Decision Tree (this library has been chosen in order to follow [5] setup)
- *Tensorboard* (2.8.0) for embeddings visualization

Data preprocessing

It's a good idea to start discussing TabTransformer from data preprocessing that requires. In order to assess model performance, it has been tested on 15 different datasets, so discussing whether to normalize or scale (or both) the input data is quite useless since it depends on the particular dataset selected for training. For example, in the specific case of MovieLens, column features doesn't seem to follow any gaussian distribution so min-max scaling has been prefered over a standard scaling (TabTransformer is a neural network based model, so it's a rule of thumb apply preprocessing in order to get data in the range $[0, 1]$).
On the other hand, independently from the dataset, it's important to distinguish between continuous features

and categorical data while working in tabular domain. Actually, the model receives as input two subset of the whole feature columns of the original dataset:

- one subset groups all categorical columns of the dataset
- one subset groups together continuous feature.

So first thing to run TabTransformer is to partition dataset and convert categorical data to appropriate unique ids (for example by using a *LabelEncoder* from *Scikit-Learn*, reason behind unique ids will be clear during *column embedding* discussion). This point is quite important because the inner transformer of the architecture learns contextual embedding of categorical data, so features must be divided in two subsets (also here there's the possibility to try different alternatives like leave categorical data as unique ids or to apply one-hot encoding).
Next step is to deal with continuous feature: standard approach is to try models in parallel with different kinds of normalization and scaling. Moreover, it's interesting to notice that a continuous feature can be discretized and treated like a categorical variable. Obviously this is feasible only if cardinality of the new categorical feature is not too big and here we shall consider a trade-off between loosing information with a coarse discretization and the cardinality of the feature. Since transformer module is applied only on categorical data, one may benefit from the discretization of a continuous feature adding more contextual information to the embeddings produced by transformer. Authors says that they didn't explore the full idea of transforming a continuous feature in a categorical one, but in my opinion we shall introduce another trade-off on the number of continuous feature discretized considering the overall model performance or considering contextual embedding produced by transformer.
At this is point, it's clear that data follows two flows in the TabTransformer architecture, so it's time to start discussing about model architecture.

Model Architecture

As explained above, categorical features of the dataset are feed into ***transformer***, while continuous feature are optionally passed through a ***layer normalization*** [1]. Outputs of the two flows are then concatenated together in order to construct input for the last component of TabTransformer: a ***multilayer perceptron***.

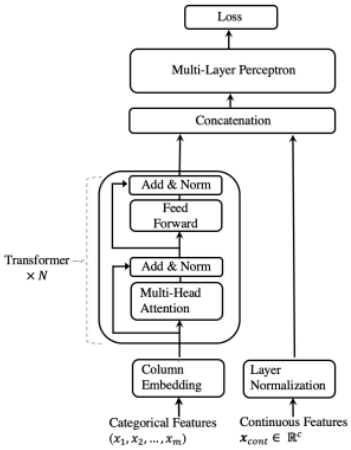


Figure 1: TabTransformer architecture.

Transformer

Categorical data is the input for subsequent *transformer* blocks that are in charge of producing a contextual embedding. In detail, categorical data is first encoded in a **parametric embedding**[1][8] thanks to the column embedding stage, so more precisely transformer layers add contextual information to the output of column embedding. This first step is implemented associating a look up table for each categorical feature, so if variable can take N values then table will have a row for each value plus a row used for missing value. Now we can retrieve an embedding for a particular value of a categorical feature in the corresponding lookup table: that particular embedding can be constructed in different ways, however, the important point is that the embedding has to be built upon a unique identifier that make possible to distinguish the value from values in the other feature columns. This concept recalls *positional encodings* from classic application of *transformers* in Natural Language Processing, but here we don't need to add to the embedding information about it's position in the sentence since we are dealing with tabular data where column order is irrelevant.
Once parametric embedding is obtained, input for transformer is ready so its possible to add contextual information to the embedding. Contextual information is collected by using attention mechanism [11] and therefore calculating a dot product is required in order to tell how much an embedding is related to others. So each embedding represent the query that is multiplied by the attention matrix; this value is finally multiplied for the

[1]parametric embedding simultaneously embeds both objects and their classes in a low-dimensional space, in other words we would like to learn embeddings of samples belonging to the same class that are as near as possible in the embedding space exploiting class posterior vectors for given data points

value matrix and passed through some feedforward layers and that's it, **contextual embeddings** of categorical data is ready to be inputted to MLP. Finally, let's notice that this transformer modules can be replicated N times and also that final output of this architecture block is not a one dimensional vector, but a matrix, since each categorical feature value has been translated by transformer in a contextual embedding (so if our contextual embedding has an associated shape, let's say d, and there are m categorical features, then output of transformer will have dimension $d * m$)

Layer Normalization

This is the block, in TabTransformer, entered by continuous data that can be optional. *Layer normalization* is an alternative to standard *batch normalization* [7]: while in the latter neuron activations are regularized by calculating mean and variance over the mini-batch of data (figure below on the left), in the first one mean and variance are calculated for a single sample across all features. The purpose of both layers is to have neurons activation with a mean and a standard deviation close respectively to 0 and 1.

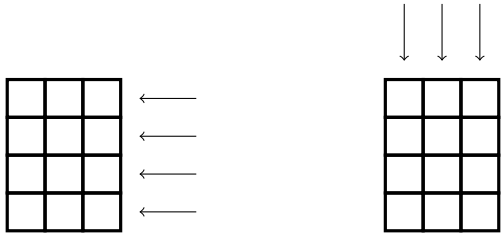


Figure 2: *Left*: Batch normalization. *Right*: Layer normalization.

Multilayer Perceptron

As mentioned at the end of transfromer discussion, contextual embeddings needs to be concatenated in order to form a one-dimensional vector that can be concatenated with continuous feature vector. In this way, input for MLP is ready and tensors can flow across hidden layers. Output layer of MLP has the same size as the number of class of the target feature: all experiments carried out on the 15 datasets, were binary classification problems. This is the reasons why it should be interesting in testing TabTransformer generalization abilities against a multi-class classification problem, like predicting the rating of a film from its characteristics.

Training and Validation

In this section the discussion flows into describing how training of the while model is performed in a supervised scenario. Before discussing results and effectiveness of the model, it would be interesting in also reviewing model training in a semi-supervised fashion.
The main pipeline adopted to train TabTransformer on each of the 15 different dataset, is:

- preprocess data and partition whole dataset in categorical and numerical feature
- split dataset in train and test, using 5-fold cross validation
- split training set in training set and validation set, using 5-fold cross validation again, but this time the purpose is to perform hyperparameters optimization[2]

Since the task is classification, the loss function adopted for MLP is a cross-entropy (let cross-entropy function be H). It's worth to mention that TabTransformer is end-to-end trained, so the loss calculated from the output of MLP is backpropagated also through transformer layer. This is a note to remember and to highlight also with the loss function:

$$L(x, y) = H(g(f(E(x_{\text{cat}})), x_{\text{cont}}), y) \tag{1}$$

In the above equation it's possible to recognize the whole flow of data through the network: first categorical data is passed through column embedding step (function E), then through transformer layer (function f) and finally concatenated to continuous data in order to get final output of MLP (function g).

[2]this way train,validation and test split get a percent of the whole dataset equals to 65/15/20; in the case of MovieLens, after all data preparation and preprocessing stages, number of datapoints is about 10 000 so this partitioning of the dataset is reasonable also for it

Semi-supervised approach

Authors proposes a different approach to classic semi-supervised methodologies when the dataset is composed of a large amount of unlabeled data and few labeled samples are available. For example, **Pseudo Labeling** training procedure is as follows:

- train model on a batch of labeled data
- predict pseudo label for a batch of unlabeled data
- pseudo labels and unlabeled data are stacked together and added to the training set that is shuffled
- train again model on labeled and pseudo-labeled data
- after some epochs of training on labeled/pseudo-labeled data, train on a batch of labeled data (this step acts a correcting stage on training)

Obviously train model on pseudo-labeled data is quite risky, so in general, when training is performed on a mixture of labeled and pseudo labeled data, a hyperparameter α is introduced to weight loss function when training on pseudo-labeled data [3].

Other main approach to semi-supervised learning is **Entropy Regularization**, that instead consists in adding to normal supervised model loss function an entropy component: in this way loss function is optimized respect to two contributes, one affected by labeled data and other one affected by unlabeled data. With entropy regularization the idea is that model performance can improve in a semi-supervised scenario when target classes doesn't overlap in the manifold (i.e. decision boundary can separate quite well classes apart). This is equivalent to say that disorder, or better entropy, between classes is as small as possible. Behind this intuition, an entropy component is added to the standard loss function of a classifier.
TabTransformer authors show a different approach to semi-supervised learning from Pseudo-Labeling and Entropy-Regularization that is based on a preliminary phase of pre-training on unlabeled data, followed by a fine-tuning phase on labeled data. To achieve this goal, for each split in which each of the 15 dataset is divided p observation are marked as unlabeled (this is required since all datasets are full labeled). *TabTransformer-PL* is the model trained following pseudo labeling approach, while *TabTransformer-RTD* and *TabTransformer-MLM* are trained using a pre-training phase followed by a fine-tuning phase. In particular the two approaches differs in how they do pre-training on TabTransformer. More specifically, just the transformer components of the architecture is pre-trained as follows:

- **Masked-Language-Model**: this is a standard in Natural Language Processing field, idea is to mask a feature among categorical data and then train the model to guess a possible value for the masked feature
- **Replaced-Token-detection**: this approach instead exploits *Electra* [3] pre-trainer, which, starting from a sample, randomly replace a value for a feature with another value for the same categorical feature; also here a discriminator is trained for detecting which feature among the categorical data of the sample has been replaced

Both MLM and RTD can mask/replace k feature in the input sample, so discriminator task will be more difficult if a lot of features have been replaced. In the field of Natural Language Processing:

- MLM discriminator needs to guess a value in a sentence
- RTD discriminator needs to detect with word in the sentence has been replaced.

Both MLM and RTD pre-traing can be modified considering the particular domain of tabular data. In particular:

- MLM: a feature is masked and model predicts a value among the unique ids associated to a categorical variable
- RTD: a categorical feature value is replaced with a random value from its associated unique ids and discriminator guesses with feature has been replaced.

[3] idea is that on early stages of training pseudo-labels leads to very poor results, so alpha value changes during epochs, that's why normally in adition to α there are also two other parameters $T1$ and $T2$: α is 0 when training on labeled data, is very small on the first epochs (till $T1$) and then increases ($T1 \leq epoch \leq T2$) and then remain constant ($epoch > T2$)

The point here is that cardinality of a categorical variable is in general very smaller then cardinality of words in a language. In RTD approach, detecting replaced token may be quite easy in natural language scenario, so authors suggests to use suitable generators for the replaced word in the sentence in order to make the task more difficult. However, in tabular domain this is not so true, in the sense that it's not easy to understand with one of the categorical variable has been replaced due to the fact that columns order is irrelevant and doesn't bring some information. Moreover, in RTD, each categorical variable has its corresponding discriminator associated since each variable has its own set of unique ids so the task of detecting the replaced value should become much more precise rather than a single discriminator, improving robustness of internal representation of transformer. Lastly, in RTD we may avoid using a generator for pre-trainer *Electra* since we can easily exploit a Pytorch *Dataset class*, and do the replacement in the $\_\_getitem\_\_$ method. All these consideration about RTD are confirmed by better results respect to MLM approach in a semi-supervised scenario for tabular data.
After pre-training phase, discriminator is thrown away and transformer should have learned some contextual information about input data. This transformer skill is further improved with the fine-tuning phase on labeled data.

Hyperparameters

Before moving on discussing result and performance of TabTransformer on different datasets and w.r.t other models like GBDT, let's first take a look a how parameters of the model are selected in the article, and then for MovieLens case.
TabTransformer hyperparameters for transformer blocks were chosen after hyperparameter optimization, not on all 15 datasets, but on 5 of them, using as search space:

- number of attention heads: {2,4,8}
- hidden dimension: {32,64,128,256}
- number of transformer layers: {1,2,3,6,12}

From this first hyperparameter optimization, number of attention head, hidden dimension and number of transformer layers were fixed to 8, 32, 6 (this is the default configuration of transformer also followed for MovieLens). Next step is to optimize hyperparameters of *multilayer perceptron* component:

- size of the first hidden layer: let l be the input size, then sizes tried for the first hidden layer are $m * l$, with $\{1 \leq m \leq 8\}$
- size of the second hidden layer: let l be the input size, then sizes tried for the second hidden layer are $m * l$, with $\{1 \leq m \leq 3\}$
- activation function: *SELU* [9]
- learning rate: $\{x = 10^{\mathrm{u}}, u \in \mathbb{U} | -6 \leq u \leq -3\}$, optimizer is *AdamW* [10]
- weight decay: $\{x = 10^{\mathrm{u}}, u \in \mathbb{U} | -6 \leq u \leq -1\}$
- dropout probability: {0.1, 0.2, ..., 0.5}

No learning rate scheduling policy applied and early stopping is used.
Baseline to compare TabTransformer is simply a multilayer perceptron, so the hyperparameters are the one listed here above. Instead, GBDT has been built using *LightGBM* library with the following hyperparameters search space:

- number of leaves in the tree: $\{x \in \mathbb{Z} | 5 \leq x \leq 50\}$
- number of datapoints required to split a leaf in the tree: $\{x \in \mathbb{Z} | 1 \leq x \leq 100\}$
- learning rate: $\{x = 5 * 10^{\mathrm{u}}, u \in \mathbb{U} | -3 \leq u \leq -1\}$
- number of estimators: $\{x \in \mathbb{Z} | 10 \leq x \leq 1000\}$

Semi-Supervised Hyperparameters

In the case of *Pseudo-Labeling* MLP, GBDT and TabTransformer models doesn't change, but what change is the training workflow, so hyperparameters are the same listed above plus $\alpha = 3$, $T1 = 30$, $T2 = 70$.

Result Analysis

Hyperparameters optimization is performed for each model following a **grid search cross validation** approach, in order to get the best model in terms of **ROC AUC score**.
First comparison is made between baseline MLP and TabTransformer: contextual embeddings are the reason why TabTransfromer outperforms MLP. This observation is strengthened by the fact that TabTransformer and MLP architectures differs only in the categorical data treatment. To give more evidence of this point, authors conducted the following experiment:

- consider contextual embeddings at different transformer layer (e.g. embedding produced after 1 transfomer layer, then embedding produced after the second layer, etc.; at the end embedding produced at the last layer should be the best one)
- instead of feeding the contextual embeddings to a MLP, concatenate embedding to continuous features and feed them as they were input for another machine learning model as Logistic Regression

From the following graph, we can clearly see that both MLP and Logistic Regression achieve better accuracies with the contextual embeddings produced at the last transformer layer, independently from the machine learning model.

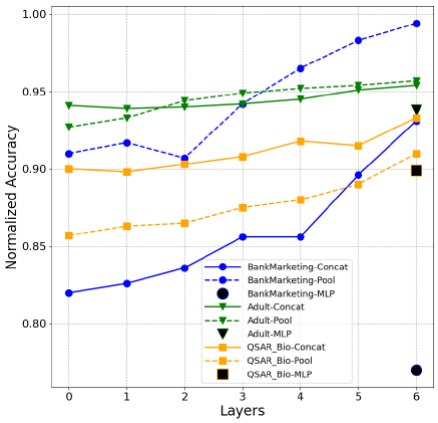


Figure 3: Effectiveness of contextual embeddings, on 3 different datasets.

Moreover, last thing confirming that transformer is gathering contextual information, is TSNE plot of both embedding before entering transformer layers and embedding after N transformer layers. We can see that transformer has added contextual information because it projects similar categories near to each other in the embedding space.

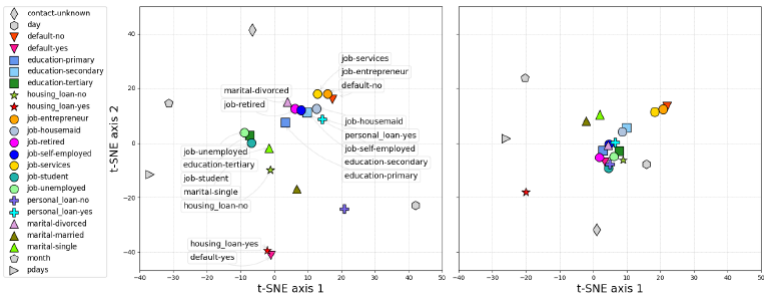


Figure 4: *Left*: embeddings from the last transformer layer. *Right*: embeddings before passing through transformer.

We shall exploit visualization tool as Tensorboard embedding projector with *1995 income* and *Bank marketing* datasets, in order to appreciate transformer contribute (live with demo).
Last thing before moving on the supervised training result, authors also show that TabTransformer is more robust against both missing and noisy data with respect to baseline MLP. To asses this, they incrementally replace either input categorical data with a random value from the column feature or mark data as missing.
Below some tables taken from original article, containing **supervised training result**, are reported in order to make some discussion (in order to see full results please look at the original article). In particular GBDT and TabTransformer represent the best models over the 15 datasets as expected.

Dataset	Datapoints	Number of features	Best Model
albert	425240	79	GBDT
hcdr_main	307511	120	GBDT
dota2games	92650	117	Logistic Regression
bank_marketing	45211	16	TabTransformer
adult	34190	25	GBDT
1995_income	32561	14	TabTransformer
online_shoppers	12330	17	GBDT
shrutime	10000	11	GBDT
blastchar	7043	20	GBDT
philippine	5832	309	TabTransformer
insurance_co	5822	85	TabTransformer
spambase	4601	57	GBDT
jasmine	2984	145	GBDT
seismicbumps	2583	18	GBDT
qsar_bio	1055	41	TabTransformer

Dataset	TabTransformer	GBDT
albert	0.757 ± 0.002	**0.763 ± 0.001**
hcdr_main	0.751 ± 0.004	**0.756 ± 0.004**
dota2games	**0.633 ± 0.002**	0.621 ± 0.004
bank_marketing	**0.934 ± 0.004**	0.933 ± 0.003
adult	0.737 ± 0.009	**0.756 ± 0.011**
1995_income	**0.906 ± 0.003**	**0.906 ± 0.002**
online_shoppers	0.927 ± 0.010	**0.930 ± 0.008**
shrutime	0.856 ± 0.005	**0.859 ± 0.009**
blastchar	0.835 ± 0.014	**0.847 ± 0.016**
philippine	**0.834 ± 0.018**	0.812 ± 0.013
insurance_co	**0.744 ± 0.009**	0.732 ± 0.022
spambase	0.985 ± 0.005	**0.987 ± 0.005**
jasmine	0.853 ± 0.015	**0.862 ± 0.008**
seismicbumps	0.751 ± 0.096	**0.756 ± 0.084**
qsar_bio	**0.918 ± 0.038**	0.913 ± 0.031

We sum up the above table in a smaller one where we can confront the different models trained for comparison:

Model	Mean AUC score
TabTransformer	**82.8 ± 0.4**
MLP	81.8 ± 0.4
GBDT	**82.9 ± 0.4**
Sparse MLP	81.4 ± 0.4
Logistic Regression	80.4 ± 0.4
TabNet	77.1 ± 0.5
VIB	80.5 ± 0.4

As anticipated, TabTransformer performance is really comparable to GBDT, outperforming other classical models for tabular data.
In the following table, instead, semi-supervised result are presented considering datasets with less than 30k datapoints and incremental number of labeled data (recall that for each split in which each of the 15 dataset is divided, first p observations are marked as labeled):

Model	50	200	500
TabTransformer-RTD	78.6 ± 0.6	**81.6 ± 0.5**	**83.4 ± 0.5**
TabTransformer-MLM	78.5 ± 0.6	81.0 ± 0.6	82.4 ± 0.5
MLP(ER)	**79.4 ± 0.6**	81.1 ± 0.6	82.3 ± 0.6
MLP(PL)	79.1 ± 0.6	81.1 ± 0.6	82.0 ± 0.6
TabTransformer(ER)	77.9 ± 0.6	81.2 ± 0.6	82.1 ± 0.6
TabTransformer(PL)	77.8 ± 0.6	81.0 ± 0.6	82.1 ± 0.6
MLP(DAE)	78.5 ± 0.7	80.7 ± 0.6	82.2 ± 0.6
GBDT(PL)	73.4 ± 0.7	78.8 ± 0.6	81.3 ± 0.6

This table is interesting because we can notice that MLM and RTD approaches performs similar, but they get worse results when labeled data are very few (50) respect to ER and PL. Both ER and PL semi-supervised training procedures has a specific term in the loss function dedicated to unlabeled data, so, even if labeled datapoints are very few, they still can benefit form the mix of both labeled and unlabeled data together. Instead, that's not the case for MLM and RTD since pre-training and fine-tuning phases are completely separated.

Conclusion

In this report TabTransformer architecture has been analyzed in order to present a neural network based model that can rival GBDT performance on tabular data. Focus is not only on the novel architecture, but also in the new semi-supervised methodology that may is major achievement presented by the article.
From result reported it's possible to compare GBDT with TabTransformer, but the whole point in TabTransformer architecture is the **preprocessing** of input data: without an appropriate handling of both continuous and categorical data, this TabTransformer may also perform worse than baseline MLP (see appendix results for more). In particular, in absence of categorical information in feature set, continuous data must be discretized in order to benefit from contextual information added by transformer component; moreover, even learned embedding must be handled very carefully to avoid overfitting and get performance comparable to GBDT.
Instead, RTD/MLM are quite effective in a semi-supervised scenario surely representing a valid methodology to be prefered over standard approaches as PL and ER (may some improvement in MLM/RTD procedures should be done to deal with issues highlighted in the semi-supervised results analysis).
Summing up, TabTransformer can be considered an alternative to GBDT at the cost of a more careful handling of continuous and categorical features (actually, TabTransformer should be preferred in place of GBDT or other standard machine learning techniques for tabular data since contextual information can alleviate noisy and missing data effects), while representing a new interesting and effective technique in semi-supervised learning.

References

[1] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

[2] O. Chapelle, B. Scholkopf, and A. Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[3] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020.

[4] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), dec 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

[5] X. Huang, A. Khetan, M. Cvitkovic, and Z. S. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *CoRR*, abs/2012.06678, 2020. URL https://arxiv.org/abs/2012.06678.

[6] D. hyun Lee. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks, 2013.

[7] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In F. R. Bach and D. M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/ioffe15.html.

[8] T. Iwata, K. Saito, N. Ueda, S. Stromsten, T. Griffiths, and J. Tenenbaum. Parametric embedding for class visualization. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2005. URL https://proceedings.neurips.cc/paper/2004/file/1d94108e907bb8311d8802b48fd54b4a-Paper.pdf.

[9] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017. URL http://arxiv.org/abs/1706.02515.

[10] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017. URL http://arxiv.org/abs/1711.05101.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Appendix

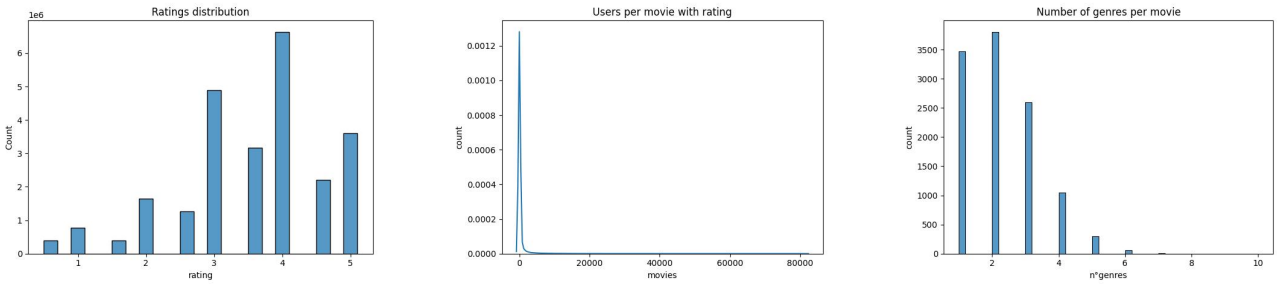
In this appendix is reported the process of data acquisition, data preprocessing, data visualization, model and performance measure applied to MovieLens dataset in order to deploy a film rating system. Models training and testing are performed following paper directions, in order to confirm paper results and also to verify TabTransformer ability to generalize in a multi-class classification task.

Data Acquisition

Exploiting *request* library from Python, MovieLens comma separated values are downloaded and extracted with *ZipFile* from https://files.grouplens.org/datasets/movielens/ml-25m.zip. Among csv files downloaded, MovieLens provide a dataset, *links.csv*, that allows to associate a movie identifier in MovieLens dataset to the corresponding movie in *Imdb* dataset. So, in addition to MovieLens data, also tab separated values files containing information about run time minutes and directors of a movie are downloaded from https://datasets.imdbws.com/ (this time *gzip* library is required since Imdb files are tar.gz files).

Data Preprocessing

After data download and extraction, some preliminary data visualization is performed in order to guide preprocessing. Since we are interested in giving a film a rating, we initially plot rating column distribution from *ratings.csv* file: we can see (figure 5/a) that this distribution is not gaussian, but is a bit skewed (see also *pipeline_output.txt* for more precise statistics one mean, standard deviation and median). Starting from ratings.csv file its also possible to see mean rating per movie, but before grouping movies, its worth to notice that ratings.csv may associate to a movie more ratings reported by different users, so it's reasonable to associate to a movie mean rating given by users. From this point of view, there will be some movies that are more popular than other and so from a statistical point of view they will be more interesting and unbiased with respect to movies rated by a small number of people (we can see a **long tail** behaviour on figure 5/b).
In order to deal with this problem, it's a good idea to first delete movies with small number of ratings, paying attention at not deleting too much films from dataset (a trade-off), and then associate to each movie a mean rating; in particular before averaging rating, we can make a more realistic calculus, by making a **weighted average** with number of voting users per category as weights (for example, there may be 100 users voting 4 to a movie, and 30 users giving 2.5 to the same movie, obviously class 4 should have more weight in the mean calculus). Once we have mean rating for a movie, we can also add some information from other csv/tsv files, like number of tags given by the users, most relevant tag (relevance is provided by a machine learning system), run time minutes, directors, genre(s) and year. Particular attention should be made for **genres**, since each movie has more than one genre associated, but it has 2-3 genres associated (figure 5/c).

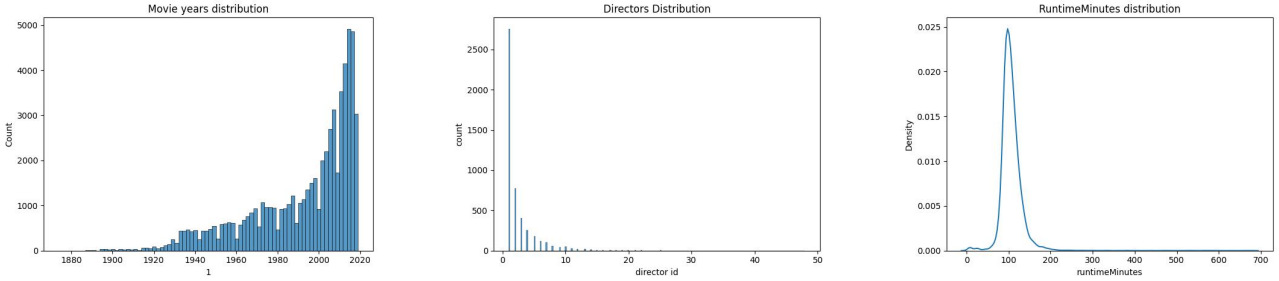


(a) Rating distribution. (b) Long tail of users per movie. (c) Number of genres per movie.

Figure 5: Generic plots of MovieLens csv files.

This could be trouble in order to considering unique ids or one-hot encoding. One possibility is to consider genres, e.g. "drama" and "drama/horror", as two separate categories. Another possibility is to exploit Pandas

get_ dummies but number of columns become too big when considering also other categorical features like directors or most relevant tag. Finally, other approach is to adopt not a dummy one-hot encoding but a binary encoding, that also leads to a bigger number of features even if smaller than *get_ dummies*.
Solution adopted is to consider "drama" and "drama/horror" as two separate categories, and convert them to unique ids. This is obviously not preferable respect to one-hot encoding since model could learn that "action" $\leq$ "action/comedy".
Below, in figure 6, some distribution of feature columns are reported in order to highlight that **data has no gaussian distribution** and so *MinMaxScaler* form *Scikit-Learn* is applied first to training data and then to validation and test set. Final notes on preprocessing is that target variable is clearly **unblanced** as we can see in figure 7, so a mix of random oversampling on the under represented classes and random under sampling on the over represented classes is applied. In this way, using a stratified k-fold cross validation, every split of the dataset will have same number of samples per class. An alternative that I didn't tried is to random over sample to the $\frac{1}{4}$ of the majority class, random under sample to the $\frac{3}{4}$ of the majority class and then introduce class weights. However, some class will have same amount of samples and it would be more interesting to try a different approach by random over/under sample each class in a customized way and then apply class weight.



(a) Year distribution. (b) Directors distributions. (c) Run time distribution.

Figure 6: Non gaussian distributions in the dataset.

Model

Model definition of MLP, TabTransformer and GBDT have been already discussed (their hyperparameters have been defined in the Hyperparameters section, notice that for the purpose of the project not all the combination of hyperparameters have been tested, but only a subset of it). Other model (from *Scikit-Learn* library) proposed are Support Vector Machine and Logistic Regression. The grid search for their hyperparameters optimization are defined over the following sets:

- SVM: C {0.01, 1, 10, 100}, tol {10, 1, 0.1, 0.01}, kernel {'linear', 'rbf'}, gamma {0.001, 0.1, 1}
- LR: penalty {'l2'}, C {0.001, 0.01, 1, 10, 100}, tol {0.01, 0.1, 0.01, 0.001}, solver {'saga', 'lbfgs'}, multiclass {'multinomial'}

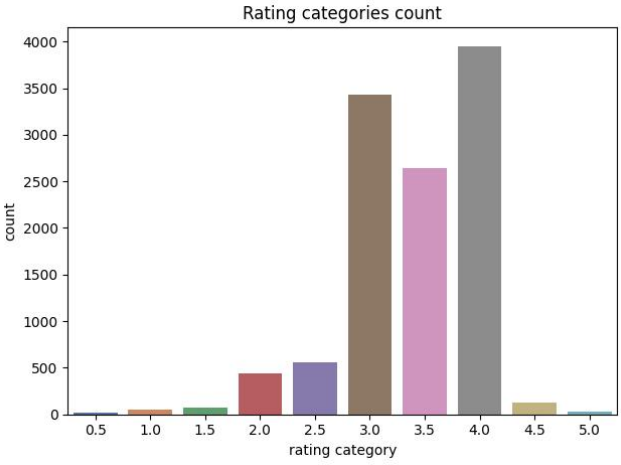


Figure 7: Unbalanced rating target variable.

Training, validation and testing (in a supervised scenario) have been already described (5-fold cross validation is applied, and another 5-fold cross validation is applied to perform grid search). Particular attention should be made for TabTransformer:

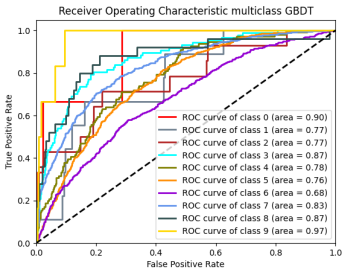
- dataset column features are splitted in continuous and categorical
- number of continuous feature is stored in the pytorch Dataset class (*MovieLensDataset* in the code)
- for every categorical feature, its number of unique values is is stored in the pytorch Dataset class
- also number of output categories (10 for MovieLens) is stored as a class field

The following, is en example of how is possible to initialize TabTransformer[4](example code, this is not initialization required for MovieLens):

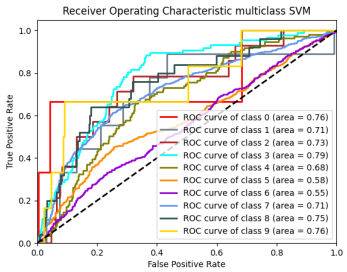
```
model = TabTransformer(
    categories = (10, 5, 6, 5, 8), # unique ids per category
    num_continuous = 10, # number of continuous features
    dim = 32, # transformer hidden layer size
    dim_out = 10, # number of output classes
    depth = 6, # number of trasnformer layers
    heads = 8, # number of attention heads
    attn_dropout = 0.1, # transformer dropout
    ff_dropout = 0.1, # mlp dropout
    mlp_hidden_mults = (4, 2), # mlp hidden size
    mlp_act = nn.SELU(), # activation for mlp
    continuous_mean_std = cont_mean_std # (optional) - layer normalization
)
```

Performance Measure

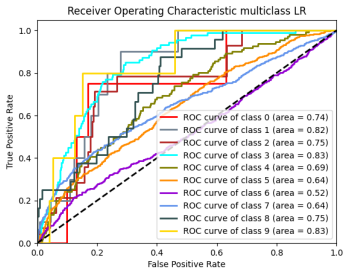
In order to compare different models, AUC score is computed and ROC AUC Curve is plotted for each model (figure 8). Since ROC curves are used for binary classification, it's necessary to extend them at the multi-class case using a "one-versus-rest" approach. Below, ROC Curves, training and test results are reported.



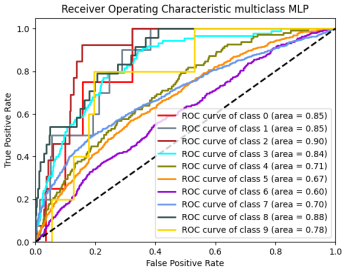
(a) GBDT



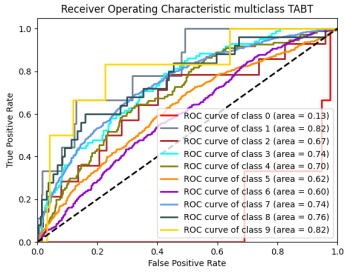
(b) SVM



(c) LR



(d) MLP

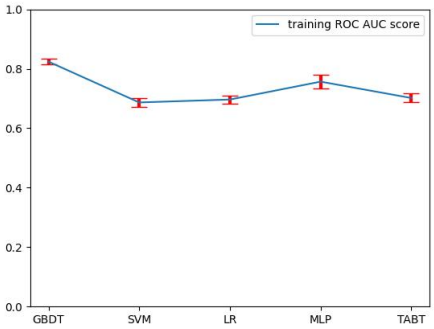


(e) TABT

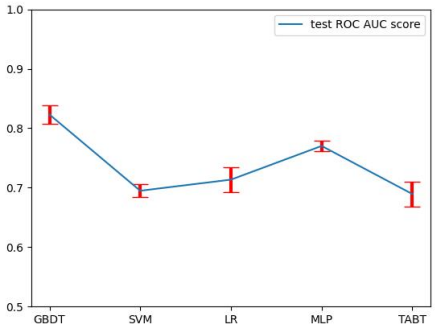
Figure 8: Models ROC AUC curves (test dataset)

As we can see (figure 9), models performances reflects results reported in the article: GBDT is the best model, followed by MLP. Other classic machine learning methods like Support Vector Machine and Logistic

[4] https://github.com/lucidrains/tab-transformer-pytorch



(a) Training results.



(b) Test results.

Figure 9: Training and performance measure results.

Regression behave quite similar.
TabTransfromer results are really poor due to overfitting: **MovieLens dataset is not a suitable dataset on which test this architecture**. The problem stands because learned embeddings are responsible for the model to overfit, since categorical features have a cardinality comparable to the number of datapoints. This is highlighted by the authors in the paragraph *B.2 Feature Engineering* where categorical data preprocessing is discussed. Hereafter, number of datapoints and categorical features cardinality is reported:

- final dataframe shape after preprocessing: 11293 datapoints with 9 feature columns
- *genres* categorical feature number of unique values: 986
- *directors* categorical feature number of unique values: 4610
- *tagId* categorical feature number of unique values: 804

Instead, now let's take a look at *1995 income* and *bank marketing*[5]dataset number of datapoints and categorical features cardinality:

- *1995 income*:
  - dataframe shape: 32561 datapoints with 15 feature columns
  - *workclass* categorical feature number of unique values: 9
  - *education* categorical feature number of unique values: 16
  - *marital-status* categorical feature number of unique values: 7
  - *occupation* categorical feature number of unique values: 15
  - *relationship* categorical feature number of unique values: 6
  - *race* categorical feature number of unique values: 5
  - *sex* categorical feature number of unique values: 2
  - *native-country* categorical feature number of unique values: 42
- *Bank marketing*:
  - dataframe shape: 45211 datapoints with 17 feature columns
  - *job* categorical feature number of unique values: 12
  - *marital* categorical feature number of unique values: 3
  - *education* categorical feature number of unique values: 4
  - *default* categorical feature number of unique values: 2
  - *housing* categorical feature number of unique values: 2
  - *loan* categorical feature number of unique values: 2
  - *contact* categorical feature number of unique values: 3

[5] *1995 income*: https://archive.ics.uci.edu/ml/datasets/bank+marketing
Bank marketing: https://www.kaggle.com/lodetomasi1995/income-classification.

- *month* categorical feature number of unique values: 12
- *poutcome* categorical feature number of unique values: 4

Sure we can see that situation is different in MovieLens respect to both other two datasets. Possible solution to make TabTransformer work properly is to augment number of films in MovieLens, but this can results in a bad performing rating system due to the statistical issue mentioned above (long tail behaviour). Another possible way is to try to cluster unique values of each categorical feature, but this may be done easily for features like *tag* while *directors* cannot be clustered.