TABRET: PRE-TRAINING TRANSFORMER-BASED TABULAR MODELS FOR UNSEEN COLUMNS

Soma Onishi*
Meiji University
Kanagawa, Japan
somaonishi4@gmail.com

Kenta Oono & Kohei Hayashi
Preferred Networks, Inc.
Tokyo, Japan
{oono,hayasick}@preferred.jp

ABSTRACT

We present *TabRet*, a pre-trainable Transformer-based model for tabular data. TabRet is designed to work on a downstream task that contains columns not seen in pre-training. Unlike other methods, TabRet has an extra learning step before fine-tuning called *retokenizing*, which calibrates feature embeddings based on the masked autoencoding loss. In experiments, we pre-trained TabRet with a large collection of public health surveys and fine-tuned it on classification tasks in healthcare, and TabRet achieved the best AUC performance on four datasets. In addition, an ablation study shows retokenizing and random shuffle augmentation of columns during pre-training contributed to performance gains. The code is available at https://github.com/pfnet-research/tabret.

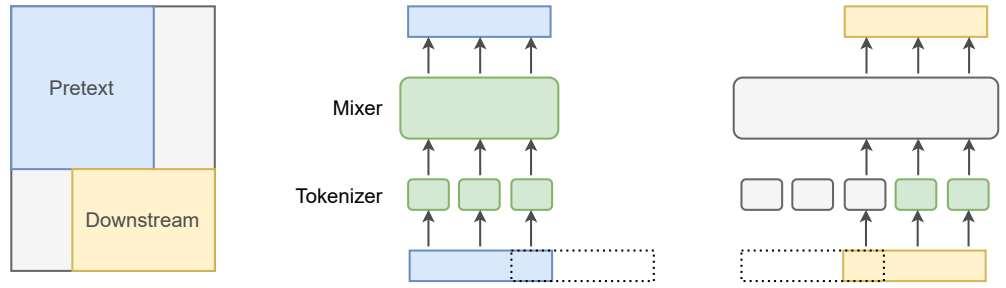
1 INTRODUCTION

Transformer-based pre-trained models have been successfully applied to various domains such as text and images (Bommasani et al., 2021). The Transformer-like architecture consists of two modules: a *tokenizer*, which converts an input feature into a token embedding, and a *mixer*, which repeatedly manipulates the tokens with attention and Feed-Forward Networks (FFN) (Lin et al., 2021; Yu et al., 2022). During pre-training, both modules are trained to learn representations that generalize to downstream tasks.

What has often been overlooked in the literature are scenarios where the input space change between pretext and downstream tasks. A supervised problem on tabular data is a typical example, where rows or records represent data points and columns represent input features. Since the data scale is not as large as text and images, pre-trained models are expected to be beneficial (Borisov et al., 2022). A key challenge is that each table has a different set of columns, and it is difficult to know at the pre-training phase which columns will appear in the downstream task. We need to train the tokenizers from scratch for unseen columns with a small amount of data. Previous studies use text data such as a column name or a description to obtain the embeddings directly from language pre-trained models (Wang & Sun, 2022; Hegselmann et al., 2022), but we cannot do this when there is no such side information.

To address the above issue, we propose *TabRet*, a pre-trainable Transformer network that can adapt to unseen columns in downstream tasks. First, TabRet is pre-trained based on the reconstruction loss with masking augmentation (Devlin et al., 2019; He et al., 2022). Then, when unseen columns appear in a downstream task, their tokenizers are trained through masked modeling while freezing the mixer before fine-tuning, which we call *retokenizing*. In experiments, we pre-trained TabRet with a table having more than two million rows and evaluated the performance on four tables containing $\sim 50\%$ unseen columns. The results show TabRet outperformed baseline methods for all the datasets. Furthermore, the ablation study confirmed that retokenizing and random shuffling of columns within a batch further enhanced the pre-training effect.

*Work done during internship at Preferred Networks, Inc.



(a) Tabular transfer learning (b) Pre-training (c) Retokenizing

Figure 1: **Problem setting and our approach.** (a) We consider transfer learning across tables that have different columns, where the pretext data is not available in the downstream task, and vice versa. (b) Our model consists of two modules: a tokenizer for each column and a mixer. We train the two modules based on the masked autoencoder loss in pre-training. (c) We introduce additional tokenizers for new columns and train them while freezing the old tokenizers and the mixer in a downstream task.

2 Related Work

Deep learning (DL)-based and tree-based methods are the two main streams of supervised learning for tabular data. Which one works better depends on the task, as they capture different aspects of the input characteristics (Grinsztajn et al., 2022; Shwartz-Ziv & Armon, 2022). However, DL-based methods are primarily adapted to transfer learning because they are easier to pre-train than tree-based methods. Among others, self-supervised learning with Transformers is the most common pre-training approach of DL-based methods (see, e.g., Badaro et al. (2023) for the survey).

Despite the popularity, most Transformer-based methods have not adapted to the case where pre-training and fine-tuning tasks have different column sets. To the best of our knowledge, TransTab (Wang & Sun, 2022), TabLLM (Hegselmann et al., 2022), and LIFT (Dinh et al., 2022) are the few exceptions that can train on tabular datasets with different column sets. TransTab creates column-by-column representations of each row from the column name and value, allowing pre-trained models to be applied to unseen columns. TabLLM and LIFT convert a row into a sequence of tokens and feed them to a pre-trained language model (Sanh et al., 2022; Wang & Komatsuzaki, 2021). They assumed there were semantic correspondences of column descriptions between different columns. In practice, however, there are many cases where column descriptions are not informative (e.g., random alphabet) or do not exist. Since TabRet does not explicitly use the column description information, it can transfer to a different column set in such a situation.

3 Method

Problem Formulation. In pre-training, suppose we have a table consisting of a finite set of columns $\mathcal{C}$. Let $\mathcal{X}_c$ be the space where column $c \in \mathcal{C}$ takes its value. A row $\boldsymbol{x}$ is then defined on the product space $\mathcal{X}(\mathcal{C}) = \prod_{c \in \mathcal{C}} \mathcal{X}_c$. Suppose a downstream task is defined as a supervised task on an input-output pair $(\boldsymbol{x}', y)$. Here we consider the case where the input $\boldsymbol{x}' \in \mathcal{X}(\mathcal{C}')$ is a row of another column set $\mathcal{C}' \neq \mathcal{C}$ (Figure 1 a). As an example, consider healthcare records where $\mathcal{C} = \{\texttt{age}, \texttt{gender}, \texttt{weight}\}$ are given as a column set in pre-training and $\mathcal{C}' = \{\texttt{gender}, \texttt{BMI}\}$ are in fine-tuning. In this case, the column `gender` appears in common, but other columns $\{\texttt{age}, \texttt{weight}, \texttt{BMI}\}$ appear only in either pre-training or fine-tuning. We assume that the pre-training table is inaccessible during fine-tuning.

Model Structure. Given a set of columns $\mathcal{C}$, we define a tokenizer $t_c : \mathcal{X}_c \to \mathcal{E}$ for each column $c \in \mathcal{C}$ as a function that converts the column value $x_c \in \mathcal{X}_c$ into an embedding vector $e_c \in \mathcal{E}$. The tokenizers then map an entire row $\boldsymbol{x}$ to a set of embeddings $\{e_c \mid c \in \mathcal{C}\}$, which are passed to a transformer-based encoder $h : \mathcal{E}^{\mathcal{C}} \to \mathcal{Z}$ to produce a latent representation $z \in \mathcal{Z}$. We also use a

decoder $d : \mathcal{Z} \to \mathcal{X}(\mathcal{C})$ to reconstruct the input in pre-training/retokenizing and a head $p : \mathcal{Z} \to \mathcal{Y}$ to predict the target variable in fine-tuning. Note that part of the decoder and the head have the same network structure, and their parameters are partially shared. More details about the network architecture are described in Appendix A.

Pre-training with Shuffle Augmentation. We train the tokenizers of the columns $\mathcal{C}$, the encoder, and the decoder by following the approach of the masked autoencoder (He et al., 2022); that is, we randomly select the columns with a masking ratio and replace the corresponding embeddings with a special embedding called a mask token. We then reconstruct the values of the masked columns and compute the loss to update the parameters. We empirically find that the masking ratio = 0.7 works well for pre-training and 0.5 for retokenizing. We use these values throughout the experiments unless otherwise mentioned.

During pre-training, we apply the shuffle augmentation as follows. Let $\boldsymbol{x}_c = (x_{1c}, \ldots, x_{nc}) \in \mathcal{X}_c^n$ be a batch of column c of size n and $\mathrm{perm}(\cdot)$ be a random permutation. Suppose $\tilde{\mathcal{C}} \subseteq \mathcal{C}$ is a set of columns chosen uniformly randomly based on a shuffle ratio. Then the shuffle augmentation replaces $\boldsymbol{x}_c$ with $\mathrm{perm}(\boldsymbol{x}_c)$ for $c \in \tilde{\mathcal{C}}$. We set the shuffle ratio to 0.1.

Retokenizing and Fine-tuning. In a downstream task, we first add initialized tokenizers for the newly appearing columns. Part of the decoder is also initialized to match the fine-tuning table. Then we train them by masked autoencoding. During this time, we do not update the parameters of the old tokenizers, encoder, and decoder (Figure 1 c). Afterward, the head is added to the backbone network (tokenizer + encoder) and fine-tuned to predict the target variable.

The primary motivation for retokenizing is to efficiently train the new tokenizers with a relatively small number of data points. Although we can train them in a supervised manner, the model is easily overfitted because, in addition to the tokenizers, the head must be trained with a single target signal. In contrast, the masked autoencoding loss used in retokenizing provides more signals to reconstruct than the target variable, which is expected to induce better token representations.

4 Experiments

Datasets. As pre-training data, we preprocessed behavioral risk factor surveillance system (BRFSS), a collection of public health surveys in the US, and created a single table consisting of 2.03 million rows and 74 columns. As downstream tasks, we selected four classification datasets in the healthcare domain from Kaggle: Diabetes, HDHI, PKIHD, and Stroke. Each dataset has about 50% overlap with BRFSS columns. However, some of these overlapping columns have different representations of their values. For example, the age column in BRFSS is categorical, but is represented as continuous in the downstream datasets, such as Stroke. We pre-processed the column representations of the downstream datasets to adjust BRFSS columns. We used 20% of the data as a test set, 100 data points from 80% for fine-tuning (and retokenizing), and the remaining data as a validation set. The dataset specifications are described in Appendix B.1.

Baselines. We used two groups of baselines: supervised methods trained only on the downstream tasks and self-supervised methods pre-trained on BRFSS. As supervised methods, we compared logistic regression (LR), XGBoost, CatBoost, MLP, and Feature Tokenizer Transformer (Gorishniy et al., 2021) (FTTrans). Self-supervised methods were SCARF (Bahri et al., 2022) and TransTab (Wang & Sun, 2022). The hyperparameters, such as learning rate, were optimized by Optuna (Akiba et al., 2019) with the validation set for each method. Details of the baseline methods are described in Appendix B.2.

Results. Table 1 shows that TabRet outperformed the baselines for all the datasets. Among the pre-trained models, TabRet consistently achieved the best. In addition to absolute performance, TabRet had relatively little variability (i.e., small variance) in the results, which is one of the desirable properties of pre-training.

We also ablated TabRet. Since the default masking ratios mentioned in Section 3 were determined based on the performance with all the learning options (pre-training, shuffle augmentation, retokenizing), we also optimized them as hyperparameters here for a fair comparison. The results

Table 1: **Test AUC performance.** The methods from LR to FTTrans were trained only on fine-tuning data (i.e., no pre-training). Each cell reports the mean and standard deviation over 20 random seeds. The best scores are underlined, and those with statistical significance at a significance level of 0.05 (Welch's t-test) are in bold.

Methods	Diabetes	HDHI	PKIHD	Stroke
LR	75.10 ± 3.55	75.55 ± 3.43	76.91 ± 2.52	74.29 ± 6.08
XGBoost	79.52 ± 0.79	80.29 ± 1.25	79.74 ± 0.93	69.02 ± 9.63
CatBoost	77.83 ± 1.40	77.65 ± 1.82	76.50 ± 1.79	76.14 ± 3.46
MLP	78.20 ± 1.02	79.39 ± 1.09	77.51 ± 1.68	76.27 ± 5.92
FTTrans	79.11 ± 1.07	78.96 ± 1.30	76.45 ± 2.38	76.48 ± 4.92
SCARF	78.43 ± 1.35	80.36 ± 1.26	81.01 ± 0.94	76.74 ± 5.04
TransTab	78.30 ± 1.18	78.77 ± 1.34	78.56 ± 1.58	75.00 ± 4.80
TabRet	$\underline{79.94 \pm 1.03}$	$\mathbf{81.65 \pm 1.60}$	$\mathbf{82.70 \pm 0.79}$	$\mathbf{80.73 \pm 3.83}$

Table 2: **Ablation study.** *Supervised* indicates TabRet that was trained only with fine-tuning data. *Supervised* row reports AUC scores and the remaining rows report the performance gains averaged over 10 random seeds.

	Diabetes	HDHI	PKIHD	Stroke	Ave. Gain
Supervised	78.71	78.42	75.47	74.29	N/A
+ Pre-training	$+0.12$	$+2.06$	$+4.98$	$+6.12$	$+3.07$
+ Shuffle aug.	$+0.90$	$+2.86$	$+4.99$	$+5.16$	$+3.21$
+ Retokenizing	$+0.64$	$+2.65$	$+6.89$	$+5.30$	$+3.87$

(Table 2) show a positive trend of pre-training. Although shuffle augmentation and retokenizing sometimes worked negatively, they provide additional gains on average.

To make a fair comparison, we evaluated if the shuffle augmentation is effective for TransTab. However, the performance did not consistently improve (refer to AppendixC.1).

5 Discussion

Tabular data have been notorious for transfer learning due to the difference in column sets. We addressed the problem and presented the transformer network with two additional steps — random shuffling in pre-training and retokenizing before fine-tuning — excelled in its potential as a pre-trained model. We hope our results will open a new research direction of pre-trained models on tabular data or other data domains where the input space can be changed.

The current limitation is that pre-training does not always provide performance gain, especially when the domain of a downstream task is irrelevant to the pretext data. We evaluated the pre-trained model used in the experiments on several datasets having no column overlap. The preliminary result was that the performance was worse than the supervised methods. It may be because the generative data process is entirely different, and there is no transferable knowledge in the task pair. Another possibility is that there was transferable knowledge, but our framework failed to capture it. Further investigation of this topic would be promising as future work.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pp. 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330701. URL https://doi.org/10.1145/3292500.3330701.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. Transformers for Tabular Data Representation: A Survey of Models and Applications. *Transactions of the Association for Computational Linguistics*, January 2023. URL https://hal.science/hal-03877085.

Dara Bahri, Heinrich Jiang, Yi Tay, and Donald Metzler. SCARF: Self-supervised contrastive learning using random feature corruption. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=CuV_qYkmKb3.

Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. Deep neural networks and tabular data: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2022. doi: 10.1109/TNNLS.2022.3229161.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 785–794, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL http://doi.acm.org/10.1145/2939672.2939785.

Jacob Devlin, Ming-Wei Chang, and Kristina Lee, Kenton andToutanova. 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)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Tuan Dinh, Yuchen Zeng, Ruisu Zhang, Ziqian Lin, Michael Gira, Shashank Rajput, Jy yong Sohn, Dimitris Papailiopoulos, and Kangwook Lee. LIFT: Language-interfaced fine-tuning for non-language machine learning tasks. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=s_PJMEGIUfa.

Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. Revisiting deep learning models for tabular data. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 18932–18943. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/9d86d83f925f2149e9edb0ac3b49229c-Paper.pdf.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017. URL https://arxiv.org/abs/1706.02677.

Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on tabular data? *arXiv preprint arXiv:2207.08815*, 2022.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 16000–16009, June 2022.

Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. Tabllm: Few-shot classification of tabular data with large language models. *arXiv preprint arXiv:2210.10723*, 2022.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *arXiv preprint arXiv:2106.04554*, 2021.

Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Skq89Scxx.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85): 2825–2830, 2011. URL http://jmlr.org/papers/v12/pedregosa11a.html.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/14491b756b3a51daac41c24863285549-Paper.pdf.

Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9Vrb9D0WI4.

Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *Information Fusion*, 81:84–90, 2022. ISSN 1566-2535. doi: https://doi.org/10.1016/j.inffus.2021.11.011. URL https://www.sciencedirect.com/science/article/pii/S1566253521002360.

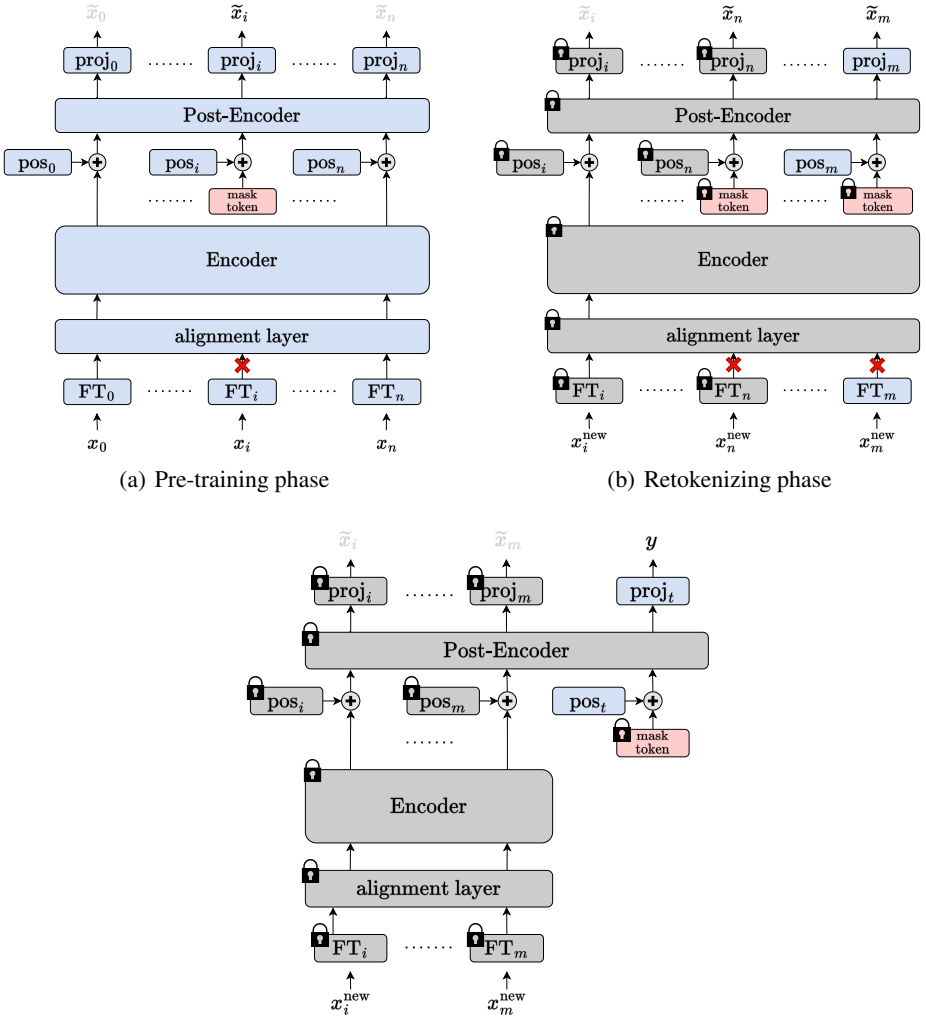
Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1810–1822, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1176. URL https://aclanthology.org/P19-1176.

Zifeng Wang and Jimeng Sun. TransTab: Learning transferable tabular transformers across tables. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=A1yGs_SWiIi.

Weihao Yu, Mi Luo, Pan Zhou, Chenyang Si, Yichen Zhou, Xinchao Wang, Jiashi Feng, and Shuicheng Yan. Metaformer is actually what you need for vision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10819–10829, June 2022.

A MODEL



(a) Pre-training phase

(b) Retokenizing phase

(c) Fine-tuning phase

Figure 2: Schematic view of our model and training phases. (a): Pre-training phase trains all modules by reconstructing masked features. (b) Retokenizing phase trains the modules corresponding to newly added columns. In this figure, the 0-th to $(i-1)$-th columns are removed, and the $(n+1)$-th to m-th columns are added. Feature Tokenizer FT_j, Positional Embedding pos_j, and Projector proj_j are trained for $j = n+1, \ldots, m$. (c): Fine-tuning phase freezes all parameters trained in the pre-training and retokenizing phases and trains the positional embedding and Projector for the target variable (pos_t and proj_t in the figure, respectively).

A.1 ARCHITECTURE

A.1.1 OVERVIEW

This section describes the architecture of our proposed model TabRet. TabRet consists of Feature Tokenizer, Alignment Layer, Random Masking, Encoder, Post-Encoder, and Projector. Feature Tokenizer converts each feature into a token. Alignment Layer normalizes the set of tokens and adjusts the tokens' dimensionality. Random Masking masks some tokens and feeds the remaining tokens to Encoder. Masked tokens are replaced with the mask token [`mask`] before Post-Encoder. Finally, Projector sends back tokens to the input domains. Figure 2 shows the schematic view of our model.

A.1.2 FEATURE TOKENIZER

Feature Tokenizer module converts an input $\boldsymbol{x}$ to an embedding T^{FT}, whose dimensions will be determined later. We assume that each column is either numerical or categorical. Since the order of the feature does not matter, as we see in this section, we write the input as $\boldsymbol{x} = (x_1^{\mathrm{num}}, \ldots, x_{k^{\mathrm{num}}}^{\mathrm{num}}, x_1^{\mathrm{cat}}, \ldots, x_{k^{\mathrm{cat}}}^{\mathrm{cat}})$. Here, k^{num} and k^{cat} are the number of numerical and categorical features, respectively. $x_j^{\mathrm{num}} \in \mathbb{R}$ is the j-th numerical feature and $x_j^{\mathrm{cat}} \in [C_j]$ is the j-th categorical feature, where C_j is the number of categories of j-th categorical feature.

Feature Tokenizer embeds numerical features using the weight matrix $W^{\mathrm{num}} \in \mathbb{R}^{k \times d_{\mathrm{FT}}}$ and the bias vector $b^{\mathrm{num}} \in \mathbb{R}^d$ as follows:

$$T_j^{\mathrm{num}} = x_j^{\mathrm{num}} W_j^{\mathrm{num}} + b_j^{\mathrm{num}} \in \mathbb{R}^{d_{\mathrm{FT}}}.$$

For categorical features, Feature Tokenizer converts the input x_j^{cat} to a one-hot vector $e_j^{\mathrm{cat}} \in \{0,1\}^{C_j}$ and embeds it using the lookup table $W^{\mathrm{cat}} \in \mathbb{R}^{C_j \times d_{\mathrm{FT}}}$ and bias $b^{\mathrm{cat}} \in \mathbb{R}^{d_{\mathrm{FT}}}$:

$$T_j^{\mathrm{cat}} = e_j^\top W_j^{\mathrm{cat}} + b_j^{\mathrm{cat}} \in \mathbb{R}^{d_{\mathrm{FT}}}.$$

W^{num}, b^{num}, W^{cat}, b^{cat} are learnable parameters. Then, embeddings are concatenated for further processing:

$$T^{\mathrm{FT}} = \mathrm{Concat}(T_1^{\mathrm{num}}, \ldots, T_{k^{\mathrm{num}}}^{\mathrm{num}}, T_1^{\mathrm{cat}}, \ldots, T_{k^{\mathrm{cat}}}^{\mathrm{cat}}) \in \mathbb{R}^{k \times d_{\mathrm{FT}}},$$

where $k = k^{\mathrm{num}} + k^{\mathrm{cat}}$.

A.1.3 ALIGNMENT LAYER

We employed different dimensions for Feature Tokenizer and Encoder for the model's flexibility. Alignment Layer changes the token dimensions to adjust Encoder using a linear layer. Alignment Layer also normalizes the scale of tokens using Layer Normalization Ba et al. (2016):

$$T^{\mathrm{AL}} = \mathrm{Linear}(\mathrm{LayerNorm}(T^{\mathrm{FT}})) \in \mathbb{R}^{k \times d_{\mathrm{AL}}},$$

where d_{AL} is the dimension of tokens that Alignment Layer outputs.

A.1.4 RANDOM MASKING

Random Masking behaves differently depending on training phases (see Appendices A.2.1 and A.2.2 for the definition of the phases.) In the pre-training and retokenizing phases, Random Masking masks some tokens randomly. More specifically, we set the mask ratio α, chose $m' = \lfloor \alpha k \rfloor$ tokens uniformly randomly from the set of tokens for each data point and dropped the chosen tokens. If $m' = 0$, we overrode the value of m' by 1. That is, Random Masking removes one token uniformly randomly. Consequently, the feature size becomes from $T^{\mathrm{AL}} \in \mathbb{R}^{k \times d_{\mathrm{AL}}}$ to $T^{\mathrm{RM}} \in \mathbb{R}^{m \times d_{\mathrm{AL}}}$, where $m = k - m'$ is the number of tokens that are not dropped. In the fine-tuning phase, Random Masking does nothing, that is, $T^{\mathrm{RM}} = T^{\mathrm{AL}}$.

A.1.5 ENCODER

Encoder is an N-layer Transformer. We use the Pre-Norm variant, which is reportedly easy for optimization Wang et al. (2019). In addition, we add one Layer Normalization after the final Trans-

former block. Mathematically, the computation of Encoder can be described as follows:

$$\begin{aligned} T_0 &= T^{\mathrm{RM}}, \\ T_i &= F_i(T_{i-1}) \quad i = 1, \dots N, \\ T^{\mathrm{enc}} &= \mathrm{LayerNorm}(T_N), \end{aligned}$$

where F_i is the i-th Transformer block. The output T^{enc} of Post-Encoder is the m-tuple of d_{enc}-dimensional tokens: $T^{\mathrm{enc}} = (T_1^{\mathrm{enc}}, \dots, T_m^{\mathrm{enc}}) \in \mathbb{R}^{m \times d_{\mathrm{enc}}}$.

A.1.6 POST-ENCODER

Post-Encoder first applies a linear layer to project embeddings to the Post-Encoder's input dimension d_{PE}. Then, a mask token $[\texttt{mask}]$, which is a $\mathbb{R}^{d_{\mathrm{PE}}}$-dimensional learnable vector, is inserted into each position where Random Masking removes the encoder token. Additionally, Post-Encoder adds learnable positional embedding $[\mathrm{pos}] = ([\mathrm{pos}]_1, \dots, [\mathrm{pos}]_k) \in \mathbb{R}^{k \times d_{\mathrm{PE}}}$ to the embeddings, which are expected to learn column-specific information. Finally, the embeddings are transformed by a one-layer Transformer block F. In summary, the architecture of Post-Encoder is as follows:

$$\begin{aligned} T_1 &= \mathrm{Linear}(T^{\mathrm{enc}}), \\ T_2 &= \mathrm{AddMaskToken}(T_1, [\texttt{mask}]), \\ T_3 &= T_2 + [\mathrm{pos}], \\ T_4 &= F(T_3), \\ T^{\mathrm{PE}} &= \mathrm{LayerNorm}(T_4). \end{aligned}$$

Similarly to Encoder, the output T^{PE} of Post-Encoder is the k-tuple of d_{PE}-dimensional tokens: $T^{\mathrm{PE}} \in \mathbb{R}^{k \times d_{\mathrm{PE}}}$.

A.1.7 PROJECTOR

Projector sends back the tokens to column feature spaces using linear layers. We prepare one projector per column since each column has different scales and dimension. Specifically, let $T^{\mathrm{dec}} = (T_1^{\mathrm{num}}, \dots, T_{k_{\mathrm{num}}}^{\mathrm{num}}, T_1^{\mathrm{cat}}, \dots, T_{k_{\mathrm{cat}}}^{\mathrm{cat}}) \in \mathbb{R}^{k \times d_{\mathrm{PE}}}$ be the Post-Encoder's output. Then, the output of the j-th projector is as follows:

$$\hat{\boldsymbol{x}}_j^* = \mathrm{Linear}(T_j^*),$$

where $* \in \{\mathrm{num}, \mathrm{cat}\}$. The dimensionality of $\hat{x}_j^{\mathrm{num}}$ is 1 for numerical features for all $j = 1, \dots, k^{\mathrm{num}}$, and $\hat{x}_j^{\mathrm{cat}}$ is C_j for the j-th categorical feature for $j = 1, \dots, k^{\mathrm{cat}}$.

We can think of our model as an encoder-decoder model by interpreting the pair of Post-Encoder and Projector as a decoder. While the original masked autoencoder in He et al. (2022) uses its decoder only in the pretext task and removes it in the downstream task, our model uses the decoder trained on the pretext task in the downstream tasks. This architectural difference comes from the modality of data. In He et al. (2022), whose modality of interest is images, models have to solve qualitatively different problems as the pretext and downstream tasks – for example, image inpainting for the pretext task object classification for the downstream task. On the other hand, in tabular learning settings, both pretext and downstream tasks are supervised learning tasks on columns. We expect the decoder is more likely to learn the knowledge beneficial for the downstream task in the fine-tuning phase. Therefore, we design our model to reuse the same decoder in the fine-tuning phases.

A.2 TRAINING

A.2.1 PRETEXT TASK

The pretext task consists of a single phase, namely, the pre-training phase (Figure 2(a)).

Shuffle Augmentation In the pre-training phase, we apply shuffle augmentation (Figure 3) to the input minibatches. We set the shuffle ratio $\beta \in (0, 1)$. Given a minibatch $\boldsymbol{x}$, we choose $\ell = \lfloor \beta k \rfloor$ features uniformly randomly. For each chosen column, we permute the features in the minibatch as described in Section 3.

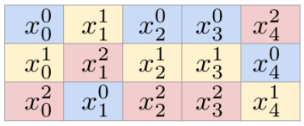


Figure 3: Schematic view of shuffle augmentation. Shuffle augmentation applies column-wise permutation within a minibatch to a randomly chosen subset of columns. This figure shows the shuffle ratio $\beta = 0.4$, and the first and fourth columns out of five are shuffled.

A.2.2 Downstream Task

We considered two types of downstream tasks. The first one was the independently-and-identically-distributed (IID) transfer learning (Appendix C.2), where pretext and downstream tasks have the same column sets. In the IID transfer learning setting, the downstream task has a single phase, namely, the fine-tuning phase (Figure 2(c)). The task is equivalent to the usual supervised learning task in the IID case. The other type was the out-of-distribution (OOD) transfer learning (Section 4), where column sets of pretext and downstream tasks could be different. The downstream task in the OOD transfer learning additionally has the retokenizing phase (Figure 2(b)) before the fine-tuning phase.

Retokenizing Phase TabRet has Feature Tokenizer, Positional Embedding, and Projector for each column, as explained in Appendix A.1. Therefore, we needed to learn these modules for columns unseen in the pretext task. In the retokenizing phase, we freeze all parameters except these modules for columns unseen in the pretext task. We employ the same masked modeling training as the pretext task. We treat the unseen columns as masked and feed the frozen mask tokens [`mask`] as the input to Post-Encoder corresponding to the columns.

Fine-tuning Phase The target value y of the downstream task is treated as a masked entry in the newly added column during the fine-tuning phase. All parameters learned during the pre-training and retokenizing phases are frozen. The Positional Embedding and Projector are then trained for the target value column, similarly to the retokenizing phase. By doing so, this approach significantly reduces the number of learning parameters, which in turn reduces the required sample size of the training dataset for the downstream task. Furthermore, we note that our model does not use the [`cls`] token, unlike the approach by He et al. (2022).

A.2.3 Loss Function

For pre-training and retokenizing phases, we define the loss value $L(\boldsymbol{x})$ of the data point $\boldsymbol{x}$ as the sum of losses for those features that are masked by Random Masking and are not shuffled by the shuffle augmentation:

$$L(\boldsymbol{x}) = \sum_{j=1}^{k^{\mathrm{num}}} \ell^{\mathrm{num}}(x_j^{\mathrm{num}}, \hat{x}_j^{\mathrm{num}}) + \sum_{j=1}^{k^{\mathrm{cat}}} \ell^{\mathrm{cat}}(x_j^{\mathrm{cat}}, \hat{x}_j^{\mathrm{cat}}).$$

For fine-tuning phase, we compute the loss value $L(\boldsymbol{x}, y)$ of a data point $(\boldsymbol{x}, y)$ as the loss for its target value:

$$L(\boldsymbol{x}, y) = \begin{cases} \ell^{\mathrm{num}}(y, \hat{y}) & \text{if } y \text{ is numerical,} \\ \ell^{\mathrm{cat}}(y, \hat{y}) & \text{if } y \text{ is categorical,} \end{cases}$$

where $\hat{y}$ is the output of Projector corresponding to the target feature when the input is $\boldsymbol{x}$. The loss function ℓ^{num} for numerical features is the mean squared loss, and the loss function ℓ^{cat} for categorical features is the cross-entropy loss. The training objective is the sum of the loss values for all instances in the training dataset.

Table 3: Dataset specifications. All datasets are binary classification tasks. Cat: the number of categorical features. Num.: the number of numerical features. Positive: the ratio of data points with the positive target. Overlap: the ratio of columns that exist in the pre-training dataset BRFSS.

Name	Data Points	Cat.	Num.	Positive	Overlap
Diabetes	253,680	20	1	0.139	0.524
HDHI	253,680	20	1	0.094	0.524
PKIHD	319,795	15	2	0.086	0.529
Stroke	4,909	2	8	0.049	0.5

Table 4: Splitting of BRFSS dataset. The numbers represent the sample sizes. The fine-tuning and test data were used in the experiments in Appendix C.2.

All			
2,038,772			
Train			**Test**
1,631,018			497,754
Pre-training	**Validation**	**Fine-tuning**	
1,453,235	163,102	100	

B Details of Experiment Settings

B.1 Datasets

We used 4 datasets in OOD experiment. Table 3 summarizes the specifications of the datasets and Table 18 lists their sources.

B.1.1 Pre-training Datasets

We used telephone survey datasets obtained from Behavioral Risk Factor Surveillance System (BRFSS)[1] as a pre-training dataset. These datasets collected state-specific risk behaviors related to chronic diseases, injuries, and preventable infectious diseases of adults in the United States. We combined the datasets from 2011–2015 and removed missing values by deleting rows and columns by the following steps: 1. deleted columns with more than 10% missing values; 2. deleted rows with missing values. We call the resulting dataset **BRFSS**. We split the BRFSS as shown in Table 4 and used the pre-train and validation datasets.

B.1.2 Fine-tuning Datasets

We prepared four datasets for the OOD Transfer Learning:

- **Diabetes**: The dataset made from the BRFSS dataset of 2015 to predict whether a subject had diabetes.
- **HDHI** (Heart Disease Health Indicator): The dataset made from the BRFSS dataset of 2015 to predict whether the subject has heart disease.
- **PKIHD** (Personal Key Indicator of Heart Disease): Similarly to HDHI, the task is to predict whether the patient has heart disease, but was made from the BRFSS dataset of 2020.
- **Stroke**: The dataset recording clinical events. The task is to predict whether a subject is likely to get a stroke using 10 features: gender, age, hypertension, heart disease, marriage, work type, residence type, glucose, BMI, and smoking status.

These datasets were divided into 80% training dataset and 20% test dataset. Of the training data, 100 samples were separated as a fine-tuning dataset.

In some columns, BRFSS and the above four data are named differently but have the same meaning. Therefore, to create overlap, the column names of the above four data were changed to match the

[1] `https://www.cdc.gov/brfss`

Table 5: Hyperparameters of Transformer layer in TransTab. The default value of n_layer is 2. FFN stands for Feed-Forward Network and is often used in the Transformer layer.

n_layer	2	6
Hidden_size	128	384
Num_attention_head	8	
Hidden_dropout_prob	0.0	
FFN_dim	128	512
Activation	ReLU	

column names in BRFSS. In addition, we performed the same feature engineering performed in BRFSS, such as categorizing age.

B.1.3 Data Preprocessing

For all methods except the tree-based methods, numerical features were transformed using the Quantile Transformation from the scikit-learn library (Pedregosa et al., 2011), and categorical features were transformed using the Ordinal Encoder. For the tree-based methods, only categorical features were transformed using the Ordinal Encoder.

B.2 Baselines

We compared our model with the following baselines:

- **Logistic Regression** (LR). We implemented it using `LogisticRegression` module of the scikit-learn package
- **Multi-layer Perceptron** (MLP). A simple deep learning model. We implemented using PyTorch (Paszke et al., 2019).
- **Gradient Boosting Decision Tree**: One of the most standard tree-based algorithms. We used two implementations in the experiments, **XGBoost** (Chen & Guestrin, 2016) and **CatBoost** (Prokhorenkova et al., 2018).
- **FT-Transformer** (Gorishniy et al., 2021). One of the standard Transformer models for tabular data, on which our model is based. We implemented using `rtdl` package[2].
- **SCARF** (Bahri et al., 2022) **with FT-Transformer**. Self-supervised learning for tabular data using contrastive learning. Although Bahri et al. (2022) employed a multi-layer perceptron as an encoder, we substituted it with FT-Transformer to remove the effect of the choice of backbones. In the pre-training phase, the class token in the Encoder's output was fed to Projector, and computed the infoNCE loss using the output of Projector. In the fine-tuning phase, the class token of the Encoder's output was fed to the classification head to solve the downstream task. We set the number of Encoder's blocks to 6. Details of Encoder are shown in Table 6.
- **TransTab** (Wang & Sun, 2022). A transformer-based model that supports transfer learning across different column sets. TransTab assumes that column names have semantic meanings, which may not be the case in the BRFSS dataset (see Section 4 and Appendix B.1 for details of dataset characteristics.) For a fair comparison, we used the same Feature Tokenizer as our model for the tokenization of features. We set the number of Encoder's blocks to 6. Details of Encoder are shown in Table 5.

We applied only supervised learning using the fine-tuning dataset to those methods that did not support transfer learning, namely LR, MLP, GBDT, and FT-Transformer. For SCARF and TransTab, we used part of the authors' implementation and modified the model as described above. Even for those models, we implemented the training part of the models by ourselves. For the methods that require pre-training, including our model, we trained the pre-trained models with multi-node distributed learning using PyTorch's `DistributedDataParallel`.

[2] `https://github.com/Yura52/rtdl`

Table 6: Hyperparameters of Transformer layer. FFN stands for Feed-Forward Network and is often used in the Transformer layer.

n_blocks	1	2	3	4	5	6
Token size	96	128	192	256	320	384
Head count	8					
Activation & FFN size factor	(ReGLU, 4/3)					
Attention dropout	0.1					
FFN dropout	0.0	0.05	0.1	0.15	0.2	0.25
Residual dropout	0.0					
Initialization	Kaiming					

B.3 Implementation of TabRet

In all experiments, unless otherwise specified, We set the number of Encoder blocks to 6. Details of each hyperparameter are shown in Table 6. We changed only the FFN dropout of Encoder to 0.1. The output dimension of the Feature Tokenizer and the input/output dimension of the Alignment Layer were aligned with the input dimension of the Encoder: $d_{\mathrm{FT}} = d_{\mathrm{AL}} = d_{\mathrm{enc}}$.

B.4 Training and Hyperparameter Optimization

We performed hyperparameter optimization with Optuna (Akiba et al., 2019) for all methods except Logistic Regression, where the number of optimization trials was 500 trials for the tree-based method and 100 trials for the DL-based method. And we set the early stopping patience to 20 for all methods except Logistic Regression.

B.4.1 Non-DL Methods

Logistic Regression (LR). We used the default settings, except that the maximum number of iterations was set to 1000.

XGBoost. The hyperparameter space for Optuna is shown in Table 7.

CatBoost. The hyperparameter space for Optuna is shown in Table 8.

Table 7: XGBoost hyperparameter space, the same hyperparameter space searched by Grinsztajn et al. (2022). We used defaults for the other hyperparameters.

Parameter	Distribution
max_depth	UniformInt[1, 11]
n_estimators	UniformInt[100, 6000, 200]
min_child_weight	UniformInt[1, 1e2]
subsample	Uniform[0.5, 1.0]
learning_rate	LogUniform[1e-5, 0.7]
colsample_bylevel	Uniform[0.5, 1.0]
colsample_bytree	Uniform[0.5, 1.0]
gamma	LogUniform[1e-8, 7]
lambda	LogUniform[1, 4]
alpha	LogUniform[1e-8, 1e2]

B.4.2 DL Methods

Table 9 shows the details of training for DL methods. For pre-training and fine-tuning, we use the linear lr scaling rule (Goyal et al., 2017) for both pre-training and fine-tuning: $lr = base\_lr \times batchsize/256$.

Table 8: CatBoost hyperparameter space. We used defaults for the other hyperparameters.

Parameter	Distribution
max_depth	UniformInt[3, 10]
learning_rate	LogUniform[1e-5, 1]
bagging_temperature	Uniform[0, 1]
l2_leaf_reg	LogUniform[1, 10]
leaf_estimation_iterations	UniformInt[1, 10]

Table 9: Detail about training for DL methods.

Config	Pre-training	Fine-tuning
optimizer	AdamW	
weight decay	1e-5	
optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.99$	
learning rate schedule	cosine decay (Loshchilov & Hutter, 2017)	
epochs	1000	200
base learning rate	1.5e-5	Searched by Optuna
batch size	4096	32
warmup epochs (Goyal et al., 2017)	40	5

Multi-layer Perceptron (MLP). The hyperparameter space for Optuna is shown in Table 10. Note that the sizes of the first and last layers were tuned and set separately, while the size of the in-between layers is the same for all of them.

FT-Transfomrer. The hyperparameter space for Optuna is shown in Table 11. Other parameters, such as token size, are determined from the corresponding parameters in Table 6 according to the selected n_blocks.

SCARF. We corrupted features independently using the Bernoulli distribution. For all datasets, the probability of corruption was fixed at 0.6. The Feature Tokenizer and Encoder parameters learned during pre-training were frozen for all fine-tuning phases, as it was confirmed that this improved the fine-tuning accuracy. The hyperparameter space for Optuna is shown in Table 12.

TransTab. We used Self-VPCL proposed by Wang & Sun (2022) and pre-trained with overlap ratio = 0.1 and num_partition = $\{2, 3, 4\}$. The hyperparameter space for Optuna is shown in Table 13.

TabRet. For all datasets, the mask ratio for pre-training was fixed at 0.7 and the mask ratio for retokenizing was fixed at 0.5.The hyperparameter space for Optuna is shown in Table 14.

Table 10: MLP hyperparameter space.

Parameter	Distribution
layers	UniformInt[1, 8]
layer_size	UniformInt[1, 512]
dropout	Uniform[0, 0.5]
category_embedding_size	UniformInt[64, 512]
base_learning_rate	LogUniform[1e-4, 1e-1]

Table 11: FT-Transformer hyperparameter space.

Parameter	Distribution
n_blocks	UniformInt[1, 6]
base_learning_rate	LogUniform[1e-4, 1e-1]

Table 12: SCARF hyperparameter space.

Parameter	Distribution
base_learning_rate	LogUniform[1e-4, 1e-1]

C Additional Experiments

C.1 TransTab with shuffle augmentation

For a fair comparison, the test AUC performance for fine-tuning in TransTab applied shuffle augmentation during pre-training has been presented in Table 15. It was observed that TransTab applied shuffle augmentation during pre-training did not result in a consistent performance improvement.

C.2 IID Experiment

We investigate how TabRet is competitive with the existing self-supervised tabular models in the same column setting.

C.2.1 Datasets

We used four popular benchmarks listed in Table 16, in addition to BRFSS.

Each of the newly added datasets was split into 4 subset, each of which is for pre-training (`pre`), fine-tuning (`fine`), validation (`val`), and test (`test`). The ratio of these subsets is as follows:

1. $\texttt{all} \rightarrow \texttt{train}_0 : \texttt{test} = 0.8 : 0.2$
2. $\texttt{train}_0 \rightarrow \texttt{train}_1 : \texttt{val} = 0.9 : 0.1$
3. $\texttt{train}_1 \rightarrow \texttt{pre} : \texttt{fine}_0 = 0.8 : 0.2$
4. $\texttt{fine}_0 \rightarrow \texttt{fine} : \texttt{none} = 100 : \text{Remaining}$

The splitting of BRFSS is shown in Table 4.

C.2.2 Baseline

We compared FTTrans, SCARF, and TransTab. Except for BRFSS, we decreased the batch size for pre-training to 512. We used 3 transformer blocks for SCARF and TabRet (see Table 6; the number of Post-Encoder blocks is 1) and 2 for TransTab as the default value of (Wang & Sun, 2022). Other settings were the same as described in Appendix B.4.

C.2.3 Result

Table 17 shows the results. TabRet outperformed the baselines on BRFSS, but the performance was not stable for other datasets. In contrast to the pre-trained models, FTTrans achieved competitive performance for all the datasets. The results suggest the size of the datasets was not large enough to surpass the supervised method.

Table 13: TransTab hyperparameter space.

Parameter	Distribution
num_partitions	UniformInt[2, 4]
base_learning_rate	LogUniform[1e-4, 1e-1]

Table 14: TabRet hyperparameter space.

Parameter	Distribution
base_learning_rate (Retokenizing)	LogUniform[1e-4, 1e-1]
base_learning_rate	LogUniform[1e-4, 1e-1]

Table 15: Test AUC performance. Comparison of Transtab with and without shuffle augmentation during pre-training. We set the shuffle ratio to 0.1, similar to the shuffle ratio in TabRet.

Method	Diabetes	HDHI	PKIHD	Stroke
TransTab	78.30 ± 1.18	78.77 ± 1.34	78.56 ± 1.58	75.00 ± 4.80
TransTab w/ shuffle aug.	77.21 ± 1.30	77.75 ± 1.71	79.36 ± 1.50	76.26 ± 4.82

Table 16: Self-Supervised datasets.

Name	Data Points	Cat.	Num.	Positive
BRFSS (2011–2015)	2,038,772	64	10	0.127
Adult (AD)	32,561	8	6	0.241
Bank Marketing (BM)	45,211	10	6	0.117
HTRU2 (HR)	17,898	0	8	0.092
Online Shoppers (OS)	12,330	7	10	0.155

Table 17: Test AUC performance in IID setting. The number of random seeds is 10.

Method	BRFSS	AD	BM	HR	OS
FTTrans	77.77 ± 1.18	88.11 ± 0.46	$\underline{86.92 \pm 0.70}$	96.82 ± 0.47	$\underline{89.89 \pm 0.68}$
SCARF	75.36 ± 2.27	88.00 ± 0.29	74.92 ± 2.38	96.34 ± 0.41	80.19 ± 3.75
TransTab	77.41 ± 2.11	$\underline{88.32 \pm 0.45}$	85.65 ± 1.35	96.74 ± 0.49	89.85 ± 0.83
TabRet	$\mathbf{79.67 \pm 1.19}$	88.08 ± 0.39	76.93 ± 1.21	$\underline{96.90 \pm 0.39}$	84.96 ± 2.21

Table 18: Dataset source.

Dataset Name	URL
BRFSS	https://www.kaggle.com/datasets/cdc/behavioral-risk-factor-surveillance-system
Diabetes	https://www.kaggle.com/datasets/alexteboul/diabetes-health-indicators-dataset
HDHI	https://www.kaggle.com/datasets/alexteboul/heart-disease-health-indicators-dataset
PKIHD	https://www.kaggle.com/datasets/kamilpytlak/personal-key-indicators-of-heart-disease
Stroke	https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset
Adult	https://archive.ics.uci.edu/ml/machine-learning-databases/adult/
Bank Marketing	https://archive.ics.uci.edu/ml/datasets/bank+marketing
HTRU2	https://archive.ics.uci.edu/ml/datasets/HTRU2
Online Shoppers	https://archive.ics.uci.edu/ml/datasets/Online+Shoppers+Purchasing+Intention+Dataset