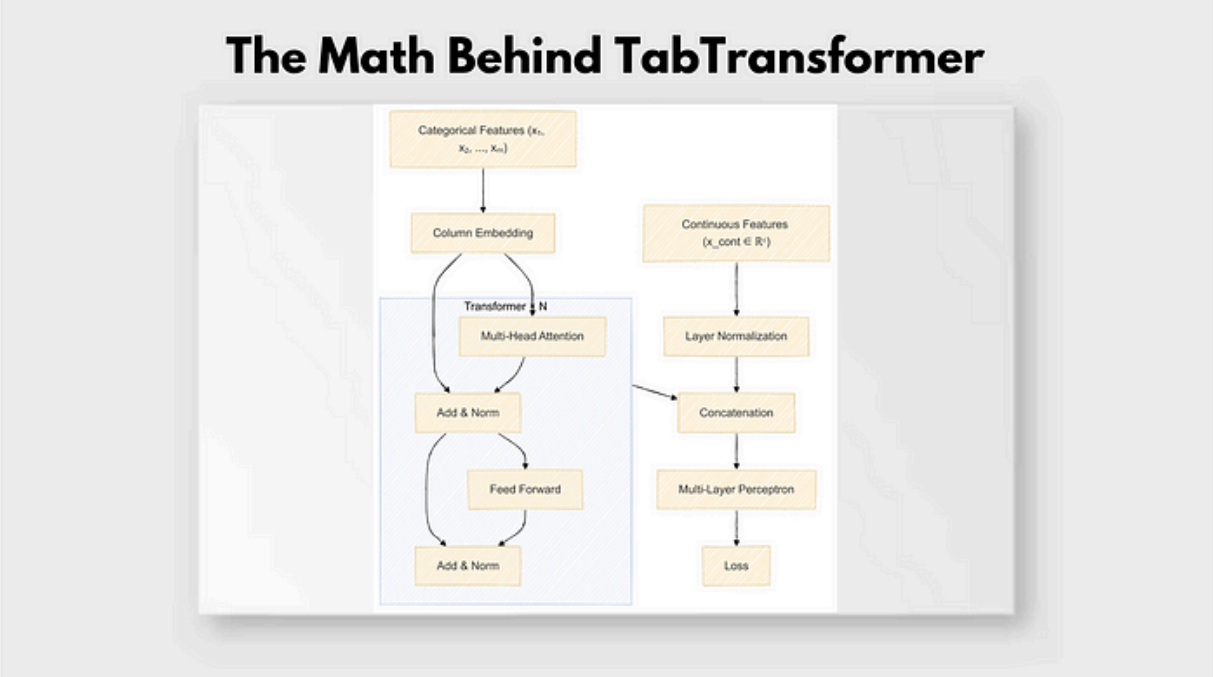


The Math Behind TabTransformer

The king of tabular data: TabTransformer. Here's how the transformer architecture wins over classical Machine Learning

Cristian Leo

Follow

androidstudio · September 10, 2024 (Updated: September 11, 2024) · Free: No

Imagine you're trying to understand a complex story. You wouldn't just read the individual sentences in isolation, right? You'd pay attention to the connections between sentences, the flow of events. That's essentially what TabTransformer does for data.

TabTransformer is a deep learning architecture specifically designed for tabular data, proposed by Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar Karnin in their 2020 research paper,

It revolutionizes tabular data modeling by introducing a new way of understanding data: **contextual embeddings**. A neural network architecture generates these contextual embeddings called Transformers, which borrow inspiration from how humans process language.

This model architecture surpasses traditional accuracy, robustness, and interpretability methods. It unlocks the potential of tabular data, allowing us to make more accurate predictions, build more resilient models, and gain deeper insights from the data we collect. Let's delve into the math behind this revolutionary approach.

Before jumping into the article, here's a few articles which will help you with mastering today's concepts. They are not required for the understanding of these articles, so feel free to go check them out afterwards:

The Math Behind Neural Networks

Dive into Neural Networks, the backbone of modern AI, understand its mathematics, implement it from scratch, and...

towardsdatascience.com

The Math Behind Multi-Head Attention in Transformers

Deep Dive into Multi-Head Attention, the secret element in Transformers and LLMs. Let's explore its math, and build it...

towardsdatascience.com

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its math, and build it from scratch...

medium.com

Today's agenda includes:

1: TabTransformer: A Novel Approach

2: The Math Behind TabTransformer

3: Pre-Training for Semi-Supervised Learning

4: TabTransformer in Python ◦ 1. Dataset Preparation ◦ 2. Configuring TabTransformer ◦ 3. Model Training ◦ 4. Model Evaluation

5: Advantages of TabTransformer

6: Conclusion

References

1: TabTransformer: A Novel Approach

Imagine Transformers as a group of highly specialized detectives, each with their area of expertise. They can look at a single piece of information, like a customer's age, and then analyze how that age relates to other factors, like their income, subscription plan, and past activity. This interconnected analysis allows them to build a

TabTransformer uses these **Transformers** to create "*contextual embeddings*" for each categorical feature in your dataset. These embeddings are not just simple representations of the feature; **they capture the nuanced relationship of that feature with others within the data.**

For example, let's say one of your features is a "*Subscription Plan*" with options like "*Basic*," "*Premium*," and "*Enterprise*." A traditional approach might simply assign each plan a numerical value. TabTransformer, however, takes into account how "Subscription Plan" relates to other features, like "Income," "Age," and "Past Activity," creating a much richer representation of the plan. It's like understanding that a "Premium" subscriber is more likely to be older and have a higher income than a "Basic" subscriber.

These contextual embeddings are the foundation for TabTransformer's success. They provide a much more comprehensive understanding of the data, allowing the model to make more accurate predictions and gain deeper insights into the relationships within your data.

2: The Math Behind TabTransformer

Let's get a little more technical and delve into the mathematical heart of TabTransformer. Think of this as looking under the hood of the tool to understand how it works. We will cover both math and Python examples in this section. To help you keep track of all the code, I compiled this notebook for you:

Repo where I recreate some popular machine learning models from scratch in Python ...

github.com

Imagine our data puzzle is represented by a table, where each row is a customer and each column represents a feature like age, income, or purchase history. We can represent this table mathematically using a **vector x**. This vector holds all the information about a single customer.

$$x = [x_1, x_2, \ldots, x_m]$$

Input Vector — Image by Author

Each element **x_i** in this vector represents a feature of the customer. We can represent that in Python using:

Copy

```
import numpy as np

# customer data
customer_age = 35
customer_income = 60000
customer_last_purchase = '2023-10-27'

# input vector x
x = np.array([customer_age, customer_income, customer_last_purchas
```

At first glance, this seems like an ordinary collection of customer details. But what if I told you that we can mold this data, transforming it step by step, with each layer altering the way we

In our case, we are particularly interested in how we can **dynamically adjust features** such as age, income, or the last purchase date. These features won't remain static — they will evolve as we pass them through different layers of transformation.

But before we get too far ahead of ourselves, let's talk about how to handle non-numeric data like dates. Since machine learning models don't understand dates directly, we need to convert them into numbers. We can represent the last purchase date by converting it into the number of days since a reference date (for instance, January 1st, 2000):

Copy

```
from datetime import datetime

# function to convert date string to a numerical value (e.g., days
def encode_date(date_str):
    reference_date = datetime(2000, 1, 1)
    date_obj = datetime.strptime(date_str, '%Y-%m-%d')
    return (date_obj - reference_date).days
```

Now, we can finally create the input vector x, this time fully numeric. Notice how the date is transformed into a number:

Copy

```
customer_last_purchase = encode_date('2023-10-27')  # Convert date
x = np.array([customer_age, customer_income, customer_last_purchas
print(x)

# Output
# [35 60000  8700] # 8700 is our encoded date
```

Let's focus on a single layer, represented by the number q. Within this layer, each edge connecting two features has a **learnable univariate function**. This function represents a transformation applied to a specific feature x_i. These functions are **parameterized as splines** — piecewise polynomial functions that allow for dynamic adjustments based on the data.

Before looking at the formula of our **univariate function**, let's clarify what the previous blurb of text is trying to say. Imagine you're building a model to predict how much a customer will spend on your website. You have information like their age, income, and past purchase history.

- **Layer**: Think of each layer in TabTransformer as a different stage in this prediction process. We'll focus on a single stage, labeled "q."
- **Edges**: Within this stage, each edge connects two pieces of information, like age and income.
- **Univariate Function**: Each edge has a special rule called a "univariate function." This rule takes a specific piece of information (like age) and changes it based on its relationship with the other connected piece (income).
- **Splines**: These functions are special — they're like flexible rulers. They can bend and adapt to the unique patterns in the data. Imagine you have a ruler that can change its shape to perfectly fit the curve of a piece of wood. Splines are similar — they can adjust their shape to best capture the relationship between features, allowing the model to be more accurate.

customers with higher incomes tend to spend more. This dynamic adjustment is what makes TabTransformer so powerful, allowing it to adapt to the nuances of the data and make more accurate predictions.

Let's now see what this function looks like:

$$\phi_{q,p}(x_i)$$

Univariate Function — Image by Author

Where:

- **x_i** is the specific piece of information (e.g., the customer's age) that goes into the box.
- **q** represents the layer number (think of it as the step in a recipe).
- **p** represents the specific edge within that layer (which edge the box is on).

This function takes the customer's information **x_i** and modifies it based on the rules defined by that specific box. In Python, we can implement this transformation like this:

Copy

```
def univariate_function(x, control_points, degree):
    num_control_points = len(control_points)
    if num_control_points < degree + 1:
        raise ValueError(f"Need at least {degree + 1} control poin

    knot_vector = np.concatenate(([0] * degree, np.arange(num_cont
```

Let's break it down step by step:

```
Copy

num_control_points = len(control_points)
```

This line simply counts the number of control points provided. Control points are like anchor points that define the shape of the spline.

```
Copy

if num_control_points < degree + 1
```

This check ensures you have enough control points for the specified degree of the spline. The degree determines the smoothness of the spline. A higher degree means a smoother curve. To create a valid spline, you need at least **degree + 1** control points.

```
Copy

knot_vector = np.concatenate(([0] * degree, np.arange(num_control_
```

Here, we define the knot vector, which is crucial for B-splines. The knot vector determines where the polynomial pieces of the spline connect. The specific structure of this knot vector is common for creating a clamped B-spline, which has fixed endpoints.

```
Copy

spline = BSpline(knot_vector, control_points, degree)
```

```
Copy

return spline(x)
```

Lastly, we evaluate the spline at the given input **x**. It essentially calculates the value of the B-spline function at point **x**.

So, these B-splines are piecewise polynomial functions that can approximate any continuous function. Imagine a line drawn on a graph, but instead of being straight, it's made up of several smooth curves connected together. This is similar to a spline. For example, let's try to use the function above on random data and plot it:

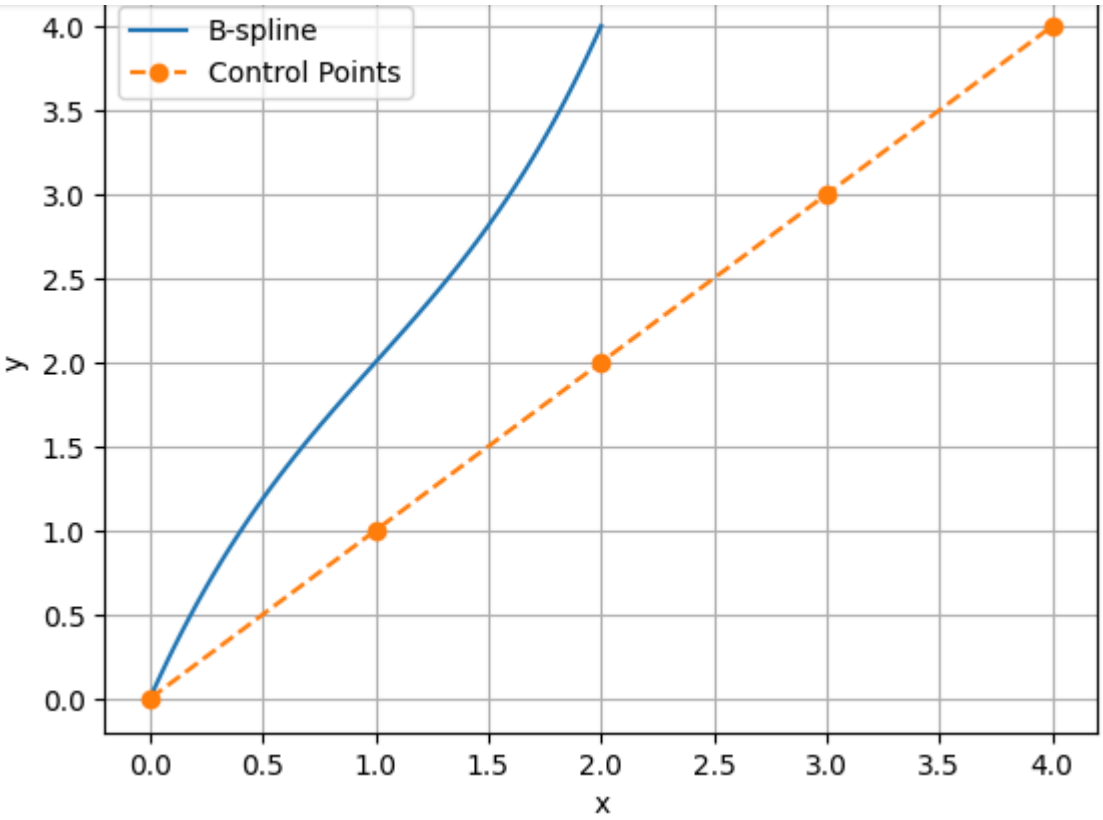
```
Copy

control_points = np.array([0, 1, 2, 3, 4])
degree = 3  # degree of the spline

# Generate a range of parameter values
x_values = np.linspace(0, len(control_points) - degree, 100)

# Evaluate the B-spline at these parameter values
y_values = univariate_function(x_values, control_points, degree)

# Plot the B-spline
plt.plot(x_values, y_values, label='B-spline')
plt.plot(np.arange(len(control_points)), control_points, 'o--', la
plt.legend()
plt.title('B-spline Curve')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.show()
```



B-Spline Plot — Image by Author

A B-spline function is defined by a set of control points and a degree. The control points determine the shape of the spline, and the degree determines the smoothness of the curve.

Now, imagine a series of these boxes arranged in a row. This formula represents a single layer of boxes, which takes the original customer information **x** and modifies each piece of it (**x_i**) using the corresponding function. The transformation of the input vector **x** at layer *q* is then calculated as a composition of these univariate functions:

$$x(q) = \phi_q(x) = [\phi_{q,1}(x_1), \phi_{q,2}(x_2), ..., \phi_{q,m}(x_m)]$$

Layer Transformation — Image by Author

- **Φ_q(x)** is the entire process of applying the boxes in a specific layer **q** to the customer information **x**.

This means that each feature in the input vector is transformed by its corresponding function in the layer.

Until now, we've explored how individual features are transformed, but what about learning the relationships **between** features? This is where **multi-head attention** comes in.

Copy

```
class MultiHeadAttention:
    def __init__(self, num_hiddens, num_heads, bias=False):
        self.num_heads = num_heads
        self.num_hiddens = num_hiddens
        self.d_k = self.d_v = num_hiddens // num_heads

        # Weights for query, key, value, and output projections
        self.W_q = np.random.rand(num_hiddens, num_hiddens)
        self.W_k = np.random.rand(num_hiddens, num_hiddens)
        self.W_v = np.random.rand(num_hiddens, num_hiddens)
        self.W_o = np.random.rand(num_hiddens, num_hiddens)

        if bias:
            self.b_q = np.random.rand(num_hiddens)
            self.b_k = np.random.rand(num_hiddens)
            self.b_v = np.random.rand(num_hiddens)
            self.b_o = np.random.rand(num_hiddens)
        else:
            self.b_q = self.b_k = self.b_v = self.b_o = np.zeros(n

    def __call__(self, queries, keys, values, valid_lens=None):
        return self.forward(queries, keys, values, valid_lens)

    def transpose_qkv(self, X):
        """
        Transposition for batch processing.
        Transpose the Q, K, V matrices for multi-head attention.
        """
        X = X.reshape(X.shape[0], X.shape[1], self.num_heads, -1)
        X = X.transpose(0, 2, 1, 3)
        return X.reshape(-1, X.shape[2], X.shape[3])
```

```
        Transposition for output.
        Combines the multiple heads back into the original format.
        """
        X = X.reshape(-1, self.num_heads, X.shape[1], X.shape[2])
        X = X.transpose(0, 2, 1, 3)
        return X.reshape(X.shape[0], X.shape[1], -1)

    def scaled_dot_product_attention(self, Q, K, V, valid_lens):
        """
        Scaled dot product attention mechanism.
        """
        d_k = Q.shape[-1]
        # Calculate the attention scores
        scores = np.matmul(Q, K.transpose(0, 2, 1)) / np.sqrt(d_k)
        if valid_lens is not None:
            mask = np.arange(scores.shape[-1]) < valid_lens[:, Non
            scores = np.where(mask[:, None, :], scores, -np.inf)
        # Softmax to get attention weights
        attention_weights = np.exp(scores - np.max(scores, axis=-1
        attention_weights /= attention_weights.sum(axis=-1, keepdi
        return np.matmul(attention_weights, V)

    def forward(self, queries, keys, values, valid_lens):
        # Transform the queries, keys, and values into multiple he
        queries = self.transpose_qkv(np.dot(queries, self.W_q) + s
        keys = self.transpose_qkv(np.dot(keys, self.W_k) + self.b_
        values = self.transpose_qkv(np.dot(values, self.W_v) + sel

        # If valid lengths are provided, repeat them across heads
        if valid_lens is not None:
            valid_lens = np.repeat(valid_lens, self.num_heads, axi

        # Apply scaled dot product attention
        output = self.scaled_dot_product_attention(queries, keys,

        # Concatenate the outputs from all heads
        output_concat = self.transpose_output(output)
        return np.dot(output_concat, self.W_o) + self.b_o
```

In this article, I won't go over the explanation of the MultiHead Attention class, as it deserves one article itself. Indeed, I wrote just

The Math Behind Multi-Head Attention in Transformers

Deep Dive into Multi-Head Attention, the secret element in Transformers and LLMs. Let's explore its math, and build it...

towardsdatascience.com

Now, let's apply **multi-head attention** to our customer data:

Copy

```
# Example usage for a transformer layer (categorical features hand
num_features = len(x)
num_layers = 3  # Number of layers in the model
embedding_dim = 16  # Increased dimension for embedding
num_heads = 4  # Number of attention heads

categorical_features = [2]  # Assume the third feature is categori

# Layer transformation function
def layer_transform(x):
    x_transformed = np.zeros_like(x)

    for p in range(num_features):
        if p in categorical_features:
            # Create a tensor of the correct shape for the attenti
            cat_feature_tensor = np.random.randn(1, 1, embedding_d
            mha = MultiHeadAttention(embedding_dim, num_heads)
            transformed = mha(cat_feature_tensor, cat_feature_tens
            x_transformed[p] = transformed.mean()  # Simplify by a

        else:
            # Ensure control points match required size
            control_points = np.linspace(0, 1, degree + 2)  # Gene
            x_transformed[p] = univariate_function(x[p], control_p

    return x_transformed

# Example layer transformation
x_transformed = layer_transform(x)
```

```
print("Transformed input:", x_transformed)
```

Here:

- **num_features = len(x):** This is just counting how many features (or columns of data) are in x. Think of x as a list of data points, and this tells us how many there are.
- **num_layers = 3:** This sets the number of layers in the model to 3. You can think of this as the number of times the data will be processed in a sequence of steps.
- **embedding_dim = 16:** This increases the dimension of the data when working with categorical (non-numeric) features. If a categorical feature is represented by a single number, it will be expanded into 16 numbers, capturing more information.
- **num_heads = 4:** This sets the number of "*attention heads*" used when processing categorical features. Each head will focus on different parts of the data.
- **categorical_features = [2]:** This assumes that the third feature (remember, in programming we start counting from 0) is categorical. In this example, it's treating a date that has been encoded as a number as a categorical feature.

The function **layer_transform(x)** takes the input x and transforms it based on whether a feature is categorical or not. If a feature is in the list **categorical_features**, it creates a random tensor (a grid of numbers) with a shape that is necessary for using attention. It then applies **Multi-Head Attention**to this tensor. After applying the attention mechanism, the result is averaged (simplified) and used to replace the original feature value.

feature. The **univariate_function** is applied to the feature, transforming it in a way that fits the data better.

Think of attention as a way for the model to decide which features are important and how they relate to each other. In our case, we can use multi-head attention to learn complex dependencies between customer features like age, income, and purchase history.

The final output of the TabTransformer is a composition of all these layer transformations. It represents the final understanding of the data, capturing complex relationships between features.

$$y = g_\psi(x(Q))$$

Output — Image by Author

Here:

- **x(Q)** represents the output of the final layer, which has undergone all the transformations through each layer.
- **g_ψ** is the final function that uses the modified customer information **x(Q)** to make a prediction, like whether the customer is likely to churn or not.

TabTransformer's core power lies in its ability to learn these dynamic functions. Each $\phi q,p(xi)$ (our univariate function) is adjusted during training based on the data, allowing the model to adapt to the specific patterns and relationships present in the dataset. This flexibility is what sets TabTransformer apart from traditional models like MLPs, which rely on fixed activation functions. We'll explore these advantages in more detail later.

3: Pre-Training for Semi-Supervised Learning

Let's say you're trying to build a model to predict which customers are likely to churn (stop using your service). You have a lot of historical data, but only a small portion of that data has actual churn labels. How can you use this vast amount of unlabeled data to help your model learn better?

This is where **pre-training** comes in. It's like giving your model a head start, letting it learn the fundamental structure of the data before it sees any labels.

TabTransformer uses a two-phase approach:

1. **Pre-training:** The model is trained on a massive dataset of unlabeled data. Think of this as letting your model explore a huge library, understanding its organization and the relationships between different books before it starts looking for specific titles.
2. **Fine-tuning:** Once the model has a good understanding of the data's structure, you feed it labeled data and fine-tune its predictions. Now, it's like having a librarian who knows the library's layout and can quickly find the specific books you need.

TabTransformer uses two main pre-training methods:

1. **Masked Language Modeling (MLM):** Imagine someone covering up some of the words in a sentence and asking you to guess what they are based on the surrounding context. MLM is like this for TabTransformer — it masks some features in the unlabeled data and trains the model to predict those missing values. This helps the model learn to recognize patterns and dependencies across features.

from the same column. It then trains the model to detect whether a feature has been replaced. This further strengthens the model's understanding of feature relationships and context.

These pre-training techniques allow TabTransformer to learn powerful contextual embeddings, even without labeled data. This gives it a significant advantage in semi-supervised learning scenarios, where labeled data is scarce, but unlabeled data is plentiful.

4: TabTransformer in Python

Now, let's see **TabTransformer** in action! We will work with a well-known dataset — the **KDD'99 dataset**, which is widely used for anomaly detection tasks. In this example, we will build a **TabTransformer** using the `pytorch-tabular` library, a powerful tool that simplifies the implementation of tabular models.

This dataset contains various network connection records, and our task is to classify whether a connection is normal or represents an attack. The features include both categorical (e.g., `protocol_type`) and continuous variables (e.g., `duration` and `src_bytes`). You can download the dataset from Kaggle:

KDD Cup 1999 Data

Computer network intrusion detection

kaggle.com

models-from-scratch-python/TabTransformer at main · cristianleoo/models-from-scratch-python

Repo where I recreate some popular machine learning models from scratch in Python ...

github.com

1. Dataset Preparation

We'll begin by loading the dataset, encoding the labels, and preprocessing both categorical and continuous features.

Copy

```
import pandas as pd
pd.set_option('display.max_columns', None)

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load the dataset
df = pd.read_csv('data/kddcup.data_10_percent_corrected', header=N

# Define column names based on the KDD'99 dataset
columns = [
    "duration", "protocol_type", "service", "flag", "src_bytes", "
    "land", "wrong_fragment", "urgent", "hot", "num_failed_logins"
    "num_compromised", "root_shell", "su_attempted", "num_root", "
    "num_shells", "num_access_files", "num_outbound_cmds", "is_hos
    "is_guest_login", "count", "srv_count", "serror_rate", "srv_se
    "rerror_rate", "srv_rerror_rate", "same_srv_rate", "diff_srv_r
    "srv_diff_host_rate", "dst_host_count", "dst_host_srv_count",
    "dst_host_diff_srv_rate", "dst_host_same_src_port_rate", "dst_
    "dst_host_serror_rate", "dst_host_srv_serror_rate", "dst_host_
    "dst_host_srv_rerror_rate", "label"
]

df.columns = columns
```

```
df['label'] = LabelEncoder().fit_transform(df['label'])

# Encode categorical columns
categorical_columns = ['protocol_type', 'service', 'flag']

for col in categorical_columns:
    df[col] = LabelEncoder().fit_transform(df[col])

num_cols = [col for col in df.columns if col not in categorical_co

# Define the target and feature columns
target = 'label'
features = df.drop(columns=[target])

# Split the dataset into train and test
train_df, test_df = train_test_split(df, test_size=0.2, random_sta

# Convert to PyTorch Tabular compatible format
train_df.reset_index(drop=True, inplace=True)
test_df.reset_index(drop=True, inplace=True)
```

In this step, we:

- Load the **KD** dataset.
- Encode the target labels (attack vs. normal) using `LabelEncoder`.
- Preprocess the categorical columns (`protocol_type`, `service`, `flag`) to be suitable for our model.
- Split the data into training and testing sets, making sure everything is well-structured and ready for the **TabTransformer** model.

Just normal data preprocessing we will likely do in any ML task.

2. Configuring TabTransformer

Now that our dataset is ready, let's configure our **TabTransformer** model. I will tend to use default settings for this model as much as

follow:

> **Approaching any Tabular Problem using PyTorch Tabular**
>
> Pre-requisites: Basic knowledge of Machine Learning and Tabular Problems like Regression and Classification Level...
>
> pytorch-tabular.readthedocs.io

TabTransformer can process both continuous and categorical data through embeddings and attention mechanisms. We'll configure the data, model, and trainer for this task.

Copy

```
from pytorch_tabular import TabularModel
from pytorch_tabular.config import DataConfig, TrainerConfig, Opti
from pytorch_tabular.models import TabTransformerConfig

# Define the configurations
data_config = DataConfig(
    target=['label'],
    continuous_cols=num_cols,
    categorical_cols=categorical_columns
)

model_config = TabTransformerConfig(
    task="classification",
    metrics=["accuracy"]
)

trainer_config = TrainerConfig(
    max_epochs=10
)

optimizer_config = OptimizerConfig()
```

- **DataConfig**: We define the continuous columns (`num_cols`), categorical columns (`categorical_columns`), and target (`label`).
- **TabTransformerConfig**: We set the task to `"classification"` and specify `accuracy` as the evaluation metric.
- **TrainerConfig**: We limit training to 10 epochs.
- **OptimizerConfig**: This handles the optimizer configurations for training the model.

3. Model Training

Next, we initialize the model and train it on our dataset.

```
Copy

# Initialize the model
tabular_model = TabularModel(
    data_config=data_config,
    model_config=model_config,
    optimizer_config=optimizer_config,
    trainer_config=trainer_config
)

# Fit the model
tabular_model.fit(train=train_df, validation=test_df)
```

The model is trained using the `train_df` and validated on the `test_df` . The **TabTransformer** architecture uses attention layers to capture relationships between features, both categorical and continuous.

4. Model Evaluation

Once the model is trained, we can evaluate its performance on the test set.

```
# Evaluate the model
test_metrics = tabular_model.evaluate(test_df)
print(test_metrics)
```

The model's accuracy on the test data will be printed. Here's an example of what the output might look like:

```
Copy

[{'test_loss': 2.305927276611328, 'test_accuracy': 0.9893730282783
```

The **TabTransformer** achieved a **98.94% accuracy**, which is impressive for a model trained on this dataset with minimal configuration and tuning. In this example, we used a minimal configuration, and trained just over 10 epochs. Just imagine what this model could do if we let it train for longer and further optimized the hyper-parameters! I will let this task to you.

5: Advantages of TabTransformer

Now that we've seen TabTransformer in action, let's talk about why it's such a game-changer for tabular data.

Remember our detective analogy? TabTransformer is like that super-smart detective who can see the connections that others miss. This ability to understand context leads to several key advantages:

- **Accuracy:** TabTransformer often achieves higher accuracy than traditional methods with fewer parameters. It's like our detective using fewer clues but still reaching more accurate conclusions because of their superior understanding. This makes TabTransformer more efficient and adaptable, especially in scenarios with limited data.

providing valuable insights into the relationships between features. By examining the embeddings, we can understand which features are most important and how they interact with each other. This helps us gain a deeper understanding of the data and make more informed decisions.

- **Robustness:** TabTransformer is more robust to missing and noisy data than traditional methods. Think of it as our detective being less susceptible to misleading clues or missing information. Contextual embeddings help the model "fill in the blanks" when data is incomplete or corrupted, leading to more reliable predictions.
- **Scalability:** TabTransformer exhibits faster scaling laws than MLPs, making it better suited for handling larger and more complex datasets. This is like our detective being able to handle more clues and connect them effectively without getting overwhelmed.

6: Conclusion

TabTransformer represents a powerful leap forward in how we analyze and understand tabular data. It's a versatile tool, able to handle complex relationships within datasets and adapt to real-world challenges. Its ability to learn contextual embeddings unlocks a new level of accuracy, robustness, and interpretability.

References

- Huang, X., Khetan, A., Cvitkovic, M., & Karnin, Z. (2020). *TabTransformer: Tabular Data Modeling Using Contextual Embeddings*. NeurIPS 2020. Retrieved from https://arxiv.org/abs/2012.06678

You Need. NeurIPS 2017. Retrieved from https://arxiv.org/abs/1706.03762

- Cristian Leo (2024). The Math Behind Multi-Head Attention in Transformers". Link

#data-science #machine-learning #artificial-intelligence #python #programming