University of Ljubljana, Faculty of Computer and Information Science

Transformers for tabular data and time series

Prof Dr Marko Robnik-Šikonja

Intelligent Systems, Edition 2025

Contents

- Transformers for tabular data
- Foundation models for tabular data
- Transformers for time series (TS) data
- Foundation models for TS data

Traditional tabular data approaches vs transformers

- Traditional tabular-data models (like XGBoost, Random Forests, SVM) often outperform deep neural networks because:
  - Tabular features can be heterogeneous (numeric, categorical, ordinal)
  - Feature interactions are complex and often non-spatial (unlike images or text)
  - There is no inherent ordering between columns
  - Dataset sizes are often small or medium, not millions of samples
- Transformers succeed in NLP and vision partly because inputs have natural sequential or spatial structure, which tabular data lacks
- How do we adapt Transformers—originally for sequences—to unordered sets of features?

Adapt transformers to tabular data

- Treat each input feature as a “token”
- Use attention to automatically learn feature interactions
- Treat features as a set, not a sequence

Treat each input feature as a “token”

- In NLP, tokens = words.
  In tabular transformers, tokens = columns/feature embeddings
- Each original feature becomes a vector embedding:
  - numeric feature → learned numerical embedding
  - categorical feature → learned category embedding (similar to word embeddings)
  - missing-value tokens are possible
- The final input is a set of feature embeddings, e.g.:

$$x \rightarrow \{e_1, e_2, \ldots, e_d\}, \text{one per feature}$$

Automatically learn feature interactions using attention

- Transformers compute pairwise interactions:

$$\text{Attention}(Q, K, V) = softmax\left(QK^{\top}/\sqrt{d}\right)V$$

- This is **useful for tabular data**, because many tasks depend on feature interactions such as:
  - Age × Income
  - Temperature × Humidity
  - Previous purchase × Current product
- Traditional models like XGBoost capture interactions through tree splitting.
- Transformers capture them via **attention weights** between features.

Treat features as a set, not a sequence

- Unlike text, tabular features don’t have left/right neighbors.
- So we use:
  - Set embeddings (no positional encoding) OR
  - Learned positional embeddings representing feature identity (not dependent on order, only on feature name)
- Some models use:
  - Feature tokens
  - Column embeddings
  - Field embeddings (groups of features)
- This gives the model awareness of *which* feature it is attending to.

Token mixing as feature engineering

- Token mixing gives expressive non-linear feature engineering
- Transformers can learn high-order feature interactions implicitly.
  Example:
  - A purely linear model cannot model things like “high income but low age → risky borrower.”
  - XGBoost learns this by splitting; Transformers learn it through multi-head attention layers.
- Multi-head attention allows different heads to learn different types of interactions. E.g.,
  - Head 1: correlations between financial variables
  - Head 2: pattern of missing values
  - Head 3: categorical → numeric interactions
  - etc.
- This is useful for tabular tasks.

Joint handling of categorical and numerical features

- Transformers handle categorical and numerical features jointly
- Advantage: instead of one-hot encoding or target encoding, categorical columns become embedding vectors, just like words in NLP.
- Numeric features can be processed by:
  - binning + embedding
  - normalization + small feedforward projection
  - continuous-value embeddings (similar to positional encodings)
- This eliminates the need for feature engineering.

Example of encoding

- Suppose a dataset contains these features:

Feature	Type	Example Value
Age	Numeric	45
Income	Numeric	60,000
Job Type	Categorical	"Engineer"
City	Categorical	"Paris"

- We want to encode this row:

x_row = { Age=45, Income=60000, Job="Engineer", City="Paris“}

Embedding numeric features

- For **numeric** features, we usually apply:
  - normalization (optional)
  - a small feedforward projection (linear layer)
- Example:

AgeEmbedding = LinearNum_Age(45) # vector in R^d_model

IncomeEmbedding = LinearNum_Income(60000) # vector in R^d_model

Assuming d_model = 32 we get

AgeEmbedding → e_age (32-dim vector)

IncomeEmbedding → e_income (32-dim vector)

Embedding categorical features

- For categorical columns, we use learned embedding tables, analogous to NLP

Engineer → Embedding(JobType)[Engineer] → e_job (32 dims)

Paris → Embedding(City)[Paris] → e_city (32 dims)

- All categorical embeddings are learned from data, just like word embeddings.
- A learnable embedding matrix maps each category to a dense vector.
- This embedding is updated via gradient descent during training (e.g., if being an engineer increases the probability of prediction, the embedding will be shifted up by the gradient)
- These embeddings replace one-hot encoding with a compact, expressive representation.
- During training the embeddings capture statistical patterns, feature interactions and semantic similarity (unintentional but common)

Column embeddings (feature identifiers)

- Each feature has a learned **column embedding:**

Feature	Column Embedding
Age	col_age
Income	col_income
Job	col_job
City	col_city

- These tell the model *which feature this token corresponds to*, since tabular features have no natural order.

Merge features and their names

Token_Age = e_age + col_age
Token_Income = e_income + col_income
Token_Job = e_job + col_job
Token_City = e_city + col_city

- Now each token knows both:
  - the value (through numerical or categorical embedding)
  - the identity of the feature (column embedding)

Forming an instance

- Optional: prepend a [CLS] token
  A learnable CLS token collects global information (like in BERT):
  Token_CLS = cls_vector (32 dims)
- Final sequence fed into Transformer
- [Token_CLS, Token_Age, Token_Income, Token_Job, Token_City]
- Each token is a vector of length 32, so input shape is 5 tokens × 32 dimensions

Together

Age=45 → Linear → e_age + col_age = Token_Age
Income=60000 → Linear → e_income + col_income = Token_Income
Job="Engineer“ → Embed → e_job + col_job = Token_Job
City="Paris" → Embed → e_city + col_city = Token_City

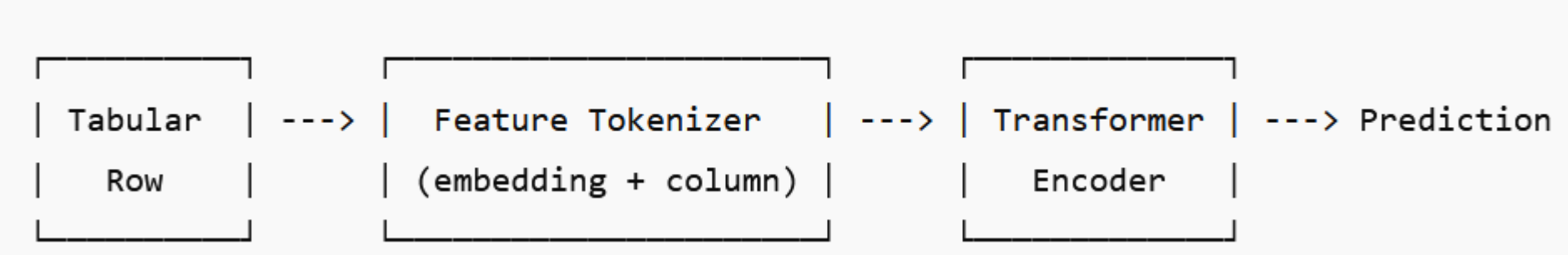
Final sequence:[CLS, Token_Age, Token_Income, Token_Job, Token_City]

- This sequence is then passed through multi-head self-attention, feed-forward layers, pooling, and final prediction head

TabTransformer

- 2020
- Categorical columns are embedded
- A Transformer stack learns interactions *between categorical features*
- Outputs feed into an MLP for final prediction
- Shown to outperform one-hot + deep networks.

FT-Transformer



- Feature Tokenizer (FT) Transformer (Gorishniy et al., 2021)
- Every feature = token
- Add learnable column embeddings
- Apply transformer blocks directly
- Pool the output for prediction
- Often matches or slightly beats XGBoost on some datasets.

Comparison to XGBoost

- **Advantages**
  - Learns high-order interactions automatically
  - Works well with raw categorical features
  - End-to-end differentiable
  - Scales well with data
  - Great for multimodal fusion (text + image + tabular)
- **Disadvantages**
  - Needs more data than XGBoost
  - Harder to tune
  - Computationally expensive
  - XGBoost still wins on many small tabular datasets

Foundation models for tabular data

- Foundation models for tabular data are large pretrained Transformer-based models that learn general-purpose representations of structured data across many datasets so they can be fine-tuned for any downstream tabular ML task.
- This is analogous to:
  - BERT → pretrained on text → fine-tuned for classification, QA
  - ViT → pretrained on images → fine-tuned for detection, segmentation
  - Tabular FM → pretrained on millions of tables → fine-tuned for forecasting, classification, regression, anomaly detection, etc.

Goals of foundation models

- A tabular foundation model aims to:
  - Learn universal patterns of tabular data
  - Capture relationships between features, categories, numeric distributions
  - Generalize across diverse datasets and domains
  - Enable training on low-data regimes (few-shot)
  - Replace hand-crafted feature engineering and classical ML baselines
- This is a major shift because historically, tabular models have been task-specific and dataset-specific (e.g., XGBoost, RF, SVM, etc).

Tabular Data Is Hard for Foundation Models

- Tabular data has challenges absent in text/images:
- Mixed feature types
  - numeric
  - categorical
  - ordinal
  - high-cardinality categories
  - missing values
- No natural spatial or sequential structure
  - Transformers succeed in domains with structure (linguistic or spatial). Tables have neither.
- Diverse ranges, distributions, and semantics
  - Income, temperature, stock price, age, ZIP code—vastly different.
- Datasets are smaller
  - Most tabular datasets have thousands (not millions) of rows.

Core approach to Tabular Foundation Models

1. Feature Tokenization
2. Large-Scale Pretraining on Many Tables
3. Universal Embedding Space for Features and Tables
4. Typically, only transformer encoder architecture with classification head

Feature Tokenization

- Each feature (numeric or categorical) is represented as a token, similar to words in NLP.
- For each training instance (row):

  [Token_age, Token_income, Token_job, Token_zipcode, Token_missing_mask, ...]

- This enables Transformers to model feature–feature interactions via self-attention.
- Popular method is FT-Transformer (simple and effective)

Large-Scale Pretraining on Many Tables

- Similar to BERT learning English by reading the entire Internet,
- Tabular foundation models learn from:
  - many tabular datasets across domains
  - synthetic tables
  - database dumps
  - business and scientific datasets
- Pretraining strategies include:
  - Masked-cell modeling: Mask random feature values and predict them (like BERT's MLM):
    Predict $x_{i,j}$ *from the rest of the row and col.*
  - Row contrastive learning: make representations of rows within the same dataset consistent.
  - Causal modeling of numeric distributions: Learn typical shapes of distributions (skewed, heavy-tailed, etc).
  - Schema-aware training: Teach the model that some features belong to the same table schema (e.g., Age, Zip_code, etc.)
- Pretraining creates universal representations.

Universal Embedding Space for Features and Tables

- The model learns:
  - embeddings for numeric values
  - embeddings for categorical values
  - embeddings for feature indices (column embeddings)
  - embeddings for table schemas
- Representation unification is key.
- Examples:
  - ZIP codes in many datasets share distributional patterns
  - Age, salary, population, etc., have a consistent statistical structure
  - Categories such as job types may share semantics across tables
- The foundation model captures these.

Architecture of Tabular Foundation Models

- While several architectures exist, most rely on:
- Transformer encoder handles feature interactions.
- Feature tokenizer encodes numeric + categorical features + metadata into tokens.
- Schema embeddings capture the meaning of table columns.
- Missing value embeddings represent missingness explicitly (often predictive).
- Prediction head is added during fine-tuning for regression/classification.

Pretraining Example: Masked Cell Modeling

- Given a row:

  Age=45, Income=60000, Job=Engineer, City=Paris
- Randomly mask:

  Age=[MASK], Income=60000, Job=Engineer, City=Paris
- Goal:

  Predict Age
- Same for categorical and numerical values.
- This forces the model to learn relationships:
  - Income ↔ Job
  - Job ↔ City
  - City ↔ Age
  - Missingness patterns

Benefits of Tabular Foundation Models

- Few-shot learning: Fine-tune on small datasets (hundreds of rows, not thousands).
- Cross-domain generalization: One pretrained model → many downstream tasks.
- No manual feature engineering: Embeddings + self-attention learn feature interactions automatically.
- Powerful transfer learning: Models pretrained on many tables outperform models trained from scratch.
- Interpretability: Attention maps may highlight feature interactions.
- Multimodal fusion capabilities: Foundation models naturally integrate:
  - text columns (via tokenization)
  - image fields (via vision embeddings)
  - time series fields
  - knowledge graph context
- Several pretrained models exist, e.g., TabLLM, TabPFN, etc.

Transformers for time series

Why Use Transformers for Time Series?

- Traditional sequence models have limitations:
  - RNNs/LSTMs struggle with long-range dependencies and parallelization
  - CNNs have fixed receptive fields unless very deep
  - Classical models (ARIMA, ETS) handle only simple patterns
  - Tree-based models do not capture sequential structure
- Transformers solve these problems by:
  - using *self-attention* to capture relationships between *all time steps at once*
  - enabling global context modeling
  - supporting multivariate time series naturally

Core idea: time steps are tokens

- Represent Each Time Step as a Token

  A time series: $x_1, x_2, x_3, \ldots, x_T$

  becomes a sequence of tokens, just like words in a sentence.
- For multivariate time series:

$$x_t = \left[x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(d)}\right]$$

- Each time step gets encoded into a vector using:
  - projection (linear layer) for numeric features
  - convolutional/embedding preprocessing for high-dimensional data
- Result:

  Embedding_1, Embedding_2, ..., Embedding_T

Add Positional Encoding!

- Unlike NLP text with word order, time series are *explicitly ordered*. Transformers need to know time step positions.
- Two approaches:
  - 1. Sinusoidal positional encoding
  - 2. Learnable positional embeddings
- The model learns a vector for each time step index.
- These are added to the step embeddings:

$$z_t = Embed(x_t) + PosEmbed(t)$$

- This gives meaning to the temporal order.

Self-Attention Learns Temporal Dependencies

- Self-attention computes how much each time step should attend to others:

$$\text{Attention}(Q, K, V) = \textit{softmax}\left(\frac{QK^{\top}}{\sqrt{d}}\right)V$$

Where:

  - Q = queries (what each time step wants to know)
  - K = keys (what each time step provides)
  - V = values (information being passed)
- In a time series, each time step can attend to any other time step.
- This lets the model learn:
  - short-term dependencies
  - long-term seasonal patterns
  - periodicity
  - sudden regime shifts
  - interactions between variables
- Unlike RNNs, attention is parallel, not sequential.

Masked Attention for Forecasting

- To prevent “peeking into the future,” transformers use **causal masks**:
  - At time t, the model can attend only to time steps ≤ t.
- This is essential for autoregressive forecasting.

Multi-Step Forecasting

- For multi-step predictions $y_{t+1}, \ldots, y_{t+h}$,the Transformer uses:
  - Encoder → processes historical data
  - Decoder → predicts future points sequentially or all at once
- The decoder inputs:
  [StartToken, y_hat(t+1), y_hat(t+2), ...]
- during inference, cross-attention connects:
  - decoder (future steps)
  - encoder (past steps)

Transformer TS properties

- **Advantages:**
  - Capture long-range dependencies
  - Parallel training
  - Flexible handling of covariates
  - Capture temporal + cross-variable relationships
  - State-of-the-art performance on many benchmarks
- **Weaknesses:**
  - Quadratic attention cost
  - Require large datasets
  - Harder to tune than simpler models

Foundation models for TS

- A time series foundation model is a *large pretrained model*—usually Transformer-based—that learns universal temporal patterns from massive collections of time series across many domains.
- Time series foundation models are pretrained on diverse temporal data so they can be adapted quickly to new time series tasks such as:
  - forecasting
  - anomaly detection
  - imputation
  - classification
  - control
  - simulation
- They aim to extract universal temporal dynamics, such as trends, seasonality, periodicity, noise patterns, cross-channel relationships
- This is highly useful because *most time series datasets are small*, but foundation models leverage large-scale pretraining to overcome this.

Why Time Series Need Foundation Models

- Diversity of shapes
  Financial data ≠ weather ≠ healthcare ≠ industrial sensors.
- Long-range dependencies
  Some effects happen months or years after causes.
- Multivariate interactions
  Many time series involve dozens–thousands of correlated variables.
- Missing values, irregular sampling
  Common in sensor / biomedical / transport data.
- Small labeled datasets
  Most forecasting datasets are tiny compared to NLP corpora.
- Foundation models provide generalizable temporal priors learned from large corpora.

Core Ideas of TS Foundation Models

- Large-Scale Pretraining on Many Time Series
- A Universal Sequence Representation
- Fine-Tuning for Downstream Tasks

Large-Scale Pretraining on Many Time Series

- Models are trained on millions of sequences across domains using self-supervised tasks such as:
- Masked time-step prediction: analogous to BERT's masked token modeling:

  x1, x2, [MASK], x4, x5

  predict x3 from context
- Next-step prediction (autoregressive pretraining)

  Teach the model dynamics by predicting x[t] from previous values
- Contrastive learning: learn representations invariant to noise, scaling, augmentation:

  two augmented versions of the same time series → same representation
- Patch-based objectives
  - Split the time series into patches
  - reconstruct missing patches
  - match patch distributions
  - predict temporal relations across patches
- This teaches models universal time-series structure.

A Universal Sequence Representation

- The dominant architecture: Temporal Transformers, which supports:
- Long-range attention captures patterns over hundreds or thousands of time steps.
- Cross-variable attention for multivariate time series, each variable becomes a token or a patch.
- Learnable positional embeddings encode time ordering.
- Several variants:
  - Informer → sparse attention for long sequences
  - Autoformer / FEDformer → frequency-domain decomposition
  - PatchTST → patch-level tokenization
  - TimesNet → periodic convolutions + attention
  - GPT-style models → pure autoregressive time series modeling

Fine-Tuning for Downstream Tasks

- After pretraining, the model is adapted with *small datasets* for tasks like:
- Forecasting

  Add a decoder head:

  model → historical window → predict future horizon
- Imputation

  Predict missing timestamps.
- Anomaly detection

  Score how “unlikely” a segment is under the learned distribution.
- Classification

  Pool the sequence representation and use an MLP head.
- Representation learning

  Use learned embeddings as features for downstream ML.
- Foundation models allow few-shot learning, where even hundreds of labeled samples are enough.

Step by step TS foundation model

1. Pretraining dataset creation; gather millions of time series from diverse sources like finance, energy, IoT sensors, healthcare, climate, retail, industrial telemetry, etc
2. Tokenization; two common strategies:
   - Time-step tokenization: Each time step is a token.
   - Patch tokenization: split time series into fixed-size patches:
     [x1...x16], [x17...x32], ...
     Each patch becomes a token (more stable and scalable).
3. Positional embeddings: make the model aware of time order.
4. Self-attention learns temporal relationships:
5. Self-supervised training: predict masked segments or next steps.
6. Save model as a foundation model and use it for transfer learning

Summary of Foundation Models for TS

- Performance boost on small datasets
  Few-shot forecasting becomes feasible.
- Capture global temporal patterns
  Trends, cycles, periodic behavior across domains.
- Rich temporal representations
  Useful for anomaly detection, classification, forecasting simultaneously.
- Scalability
  Transformers + patches allow long-context modeling.
- Multivariate compatibility
  Models learn cross-channel dependencies automatically.

Several pretrained models: PatchTST, GPT4TS, Chronos