# NATIVELY MODELING COMPLEX SEQUENTIAL DATA WITH MODULAR FIELD EMBEDDINGS

GRANTHAM TAYLOR

Abstract. Large organizations struggle to model complex sequential data collected via Event Sourcing. Without the proper solutions to apply modern deep learning techniques, businesses still use traditional machine learning techniques, which require significant time and monetary costs and yield suboptimal performance. Previous research has developed techniques that narrowly outperform tabular machine learning but only with extensive data preprocessing and domain expertise. This paper describes a model training pipeline that can natively model complex sequential data with a novel modular embedding system paired with a dynamically generated transformer-encoder model architecture. This approach can represent multiple nullable arrays of any combination of discrete, numerical, temporal, and anonymous entity representations without feature engineering or batch preprocessing. This paper also includes novel techniques that enable end-to-end automation and optimize computational performance. Our contributions deliver a novel solution that may automate model development, reduce model deployment costs, and significantly improve model performance.

### 0.1. **Problem Statement.**

From fraud detection to predicting each customer's risk of charging off, many organizations approach their business problems with tabular machine learning. However, modeling data with tabular machine learning can be surprisingly challenging in practice. Some machine learning projects can take years to develop and deploy to production. We aim to solve the following four shortcomings of applying tabular machine learning to problems among a subset of these problems:

1. Preparing production data for tabular machine learning requires manual feature engineering that dilutes the signals present in the raw source data, thus harming model performance.
2. Optimizing tabular models is tedious and time-consuming because the model developers must iteratively engineer tabular features from complex non-tabular data and explore their marginal contributions to the model's performance across various iterations and methods of signal aggregation while ensuring calculated features's computational requirements and complexity are justified.
3. Deploying tabular models in real-time requires extensive data engineering to calculate multiple, arbitrarily complex tabular features within milliseconds.
4. Developing models for emerging business problems requires engineering new, arbitrarily unique features to extract different signals from the same data sources, thus limiting an organization's ability to consolidate technological and personnel resources among otherwise identical implementations.

These problems originate from one critical insight: Many common machine learning problems solved with tabular machine techniques are not of a tabular form. Organizations restructure their business problems so that they may utilize tabular machine learning techniques to model their complex sequential data, accruing vast amounts of technical debt in the process.

By unraveling this misconception and developing a tailored solution for a large subset of business data in its native form, we can address these four deficiencies to significantly improve model performance, increase speed-to-market, automate model deployment, and consolidate the resources among the arbitrarily unique implementations of individual machine learning projects, thus solving all four posited issues simultaneously.

### 0.2. **The Practical Limitations of Tabular Machine Learning.**

Tabular machine learning techniques, specifically GBMs (Gradient Boosted Machines), are powerful when used correctly. However, most "business data" does not conform to the many assumptions that tabular machine learning techniques require. Consequently, model developers must spend significant resources to transform their business data to meet these assumptions required for tabular machine learning. In other words, model developers transform their data to make it tabular instead of modeling it in its native form.

Suppose you work at a financial institution and are training a model to determine if a customer has misrepresented their identity. Your model is a small piece of the identity verification pipeline, and your task is to flag the most risky customers for a manual review by operations. Here is an example of some production data which tells the story of Jane Doe.

| customer | timestamp | event |
|---|---|---|
| Jane | 01-01-2024 15:42:51 | `CreateAccountEvent()` |
| Jane | 01-02-2024 07:44:25 | `AddEmailEvent(email='jame@hotmail.co')` |
| Jane | 01-02-2024 07:45:41 | `AddPhoneEvent(phone='4217838914', is_local=True)` |
| Jane | 01-02-2024 07:48:53 | `ConfirmPhoneEvent(phone='4217838914', is_local=True)` |
| Jane | 01-02-2024 07:51:55 | `AddIncomeEvent(income=90413)` |
| Jane | 01-02-2024 08:05:03 | `ConfirmIncomeEvent(income=90413)` |
| Jane | 01-03-2024 11:18:29 | `FailedEmailAttemptEvent(email='jame@hotmail.co')` |
| Jane | 01-03-2024 16:33:49 | `AddEmailEvent(email='jane@hotmail.com')` |
| Jane | 01-03-2024 16:35:12 | `ConfirmEmailEvent(email='jane@hotmail.com')` |

TABLE 1. How sequential/transactional data might look in a production table

This table contains the complete history of every event posted by every customer, although we only show the history of our customer "Jane" for simplicity. Tabular machine learning techniques cannot model this rich data structure even though the data may fit in a table, as each observation is not independent. In other words, there are inter-record relationships that tabular models cannot understand. This data structure is universal: a "ledger" of events every customer posts. Each event arrives asynchronously and is immutable such that once a customer does something, it permanently stays on the ledger for all time. Each event can contain any number of complex fields, and each field can be of any data type. For example, the type of any field may be categorical, of raw text, having graph-like entity identifiers such as an IP Address or phone number, of arbitrarily complex numeric distributions, a simple boolean value, structured timestamp, structured geospatial coordinates, or even file paths to images, audio clips or other such media.

This data architecture pattern is called "Event Sourcing." Event sourcing can easily apply complex business logic and record every customer's history for complete reproducibility. It is also able to scale with *Big Data*.

**The ledger is the source of truth for everything that has ever happened.**

Modeling the ledger is exceedingly challenging, however. It is standard to simplify such a complex data structure into a tabular format to utilize traditional machine learning techniques, such Gradient Boosting Machines (GBMs), by creating independent observations with hundreds to thousands of tabular features, each representing the composite of previous events. Developing such tabular features from the ledger is called "tabular feature engineering." If done correctly, a tabular machine learning model can model the target variable as a function of the tabular features. However, extensive manual effort and subject matter expertise are required to solve common business problems adequately.

Given the above ledger, as a crude first draft, one might engineer the following features:

- `income`: The customer's most recently reported income
- `phone_confirmed`: Whether each customer has verified their most recent phone number
- `email_confirmed`: Whether each customer has verified their most recent email address
- `verified_income`: Whether each customer has verified their most recent reported income
- `account_tenure`: The number of hours since each customer's account was opened

| customer | income | phone_local | email_confirmed | verified_income | account_tenure |
|---|---|---|---|---|---|
| Jane | 90,413 | True | True | True | 49 |

TABLE 2. An example of tabular features created from sequential data

With the newly transformed data, a model developer may apply a tabular technique, such as a GBM, to predict the probability of whether a customer is lying about their identity. Such a model may provide decent initial model results given enough labeled training observations.

While the ledger is much more complicated to model than this tabular view, it contains significantly more information. For example, the ledger shows three interesting insights that are missing from the tabular view:

1. Jane originally misspelled her email address as "jame@hotmail.co", so she couldn't verify it until she corrected it to "jane@hotmail.com".
2. The time that transpired between each event observed of Jane's activity.
3. Jane has not changed her income since she joined.

All of these signals *could be* vital to modeling an identity verification problem. Tabular machine learning cannot model all these potential signals unless a model developer manually engineers new features. A model developer *can* create more tabular features that can collectively approach the signal available within the ledger.

For example, a model developer could create more tabular features to count the times each customer has changed their email over varying time windows (the last day, week, and month). They would also need to count the number of times the email was verified among varying time windows (over the last day, week, and month) and the number of times the email verification failed. However, consider how many nuanced signals may be lost during this aggregation while operating with more complex data in which customers may post hundreds of unique events, each with dozens of potential fields associated with them. The model developer would need to create hundreds, if not thousands, of tabular features to account for many intersecting events over varying time windows, with multiple possible aggregation functions. Meanwhile, the ledger already contains this information in a concise form.

No matter how skilled or experienced a model developer cannot sufficiently explore all possible tabular features because the set of possible tabular features is of effectively infinite size.

When modeling complex sequential data with tabular models, model developers rely upon such aggregative window functions to capture the complex signal that would otherwise be lost. These window functions are extremely computationally expensive to calculate at scale, and it takes a great deal of time and domain expertise to know which features to prioritize. However, even beyond the costly computational and domain expertise requirements, calculating hundreds of arbitrarily unique window functions in real time requires teams of dedicated on-call engineers.

Lastly, GBMs, being *greedy* learners, are especially prone to overfitting with many tabular features. You cannot simply throw thousands of infinitely many potential features into the model, or its performance will start to degrade. This constraint adds even more complexity to the optimization process, as the model developer now must iteratively prune the tabular features from the model while also considering the model's performance and the computational requirements to deploy the model.

These factors culminate in a painful, manual iterative process of engineering tabular features, training models with them, and then pruning the tabular features for the sake of model complexity and real-time compute requirements. Because of all of these costs of modeling sequential data with tabular models, many organizations cannot address many problems that would otherwise provide tremendous value-add. This unfortunate combination of constraints is why traditional modeling techniques can require years to develop and deploy despite being relatively simple from a machine learning perspective.

By modeling the ledger exactly how it already exists instead of creating a "view" of it, we may be able to skip all of the feature engineering. Feature engineering is the primary reason tabular machine learning is so expensive to develop and deploy for real-world use cases. It is time-consuming, tedious, prone to error, and extremely hard to scale for real-time computation. It requires extensive testing, maintenance, and monitoring. Tabular machine learning fails to address the requirements of businesses' problems.

Using the raw data to model our problem allows us to eliminate these significant costs from the modeling process while also providing a means to introduce the many benefits of more modern machine learning techniques.

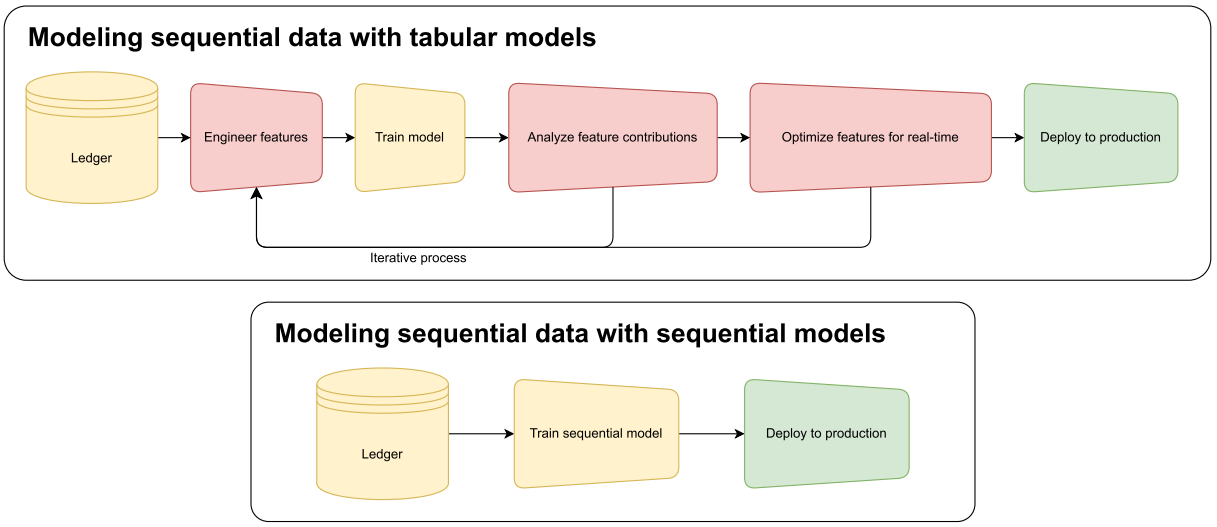


FIGURE 1. Cutting out tabular feature engineering increases the speed-to-market of model development projects

**We should change our model to fit our data instead of changing our data to fit our model.**

The cost of manual transformation has many parallels in the history of machine learning for universal modalities. The modality of computer vision comes to mind. Before the birth of CNNs, data scientists manually defined convolution kernels. This manual transformation is analogous to how organizations currently model their business problems.

Identity verification is hardly the only problem that suffers from the shortcomings of tabular machine learning. This sequential data structure is relevant to many notable business problems that would all benefit from a generalized modeling approach:

1. Given every previous ride observed by an Uber customer, what is the probability that they will be late for the next ride?

2. Given every activity log observed by a customer's Apple Watch, what is the probability that they will meet their workout goal this week?
3. Given every transaction a Chase credit card holder posted, what is their survival curve (due to attrition or default) over the next five years?
4. Given previous energy usage events observed by a customer of Duke Energy, what is their expected energy usage for the next week?
5. Given the previous purchases observed by a vendor on eBay, what is the probability that they will be reported as a "scammer" in the next week?
6. Given the previous transactions posted by a Venmo user, what is the probability that the current transaction is an attempt to defraud another user?
7. Given the previous tweets posted by a Twitter user, what is the probability that this user is a bot?

We expect hundreds of other such business problems with dozens of different industries. By developing a consolidated modeling framework that can model all of them with a single strategy that does not require arbitrarily unique means of extracting signals, we may commoditize the machine learning requirements that underlay their business operations.

In summary, instead of manually aggregating the events through miscellaneous window functions to represent their current state, we may, with the use of deep learning, create a representation of each customer at any point in time, given all of the events that they have posted up until that point in time. We assume, from the perspective of our model, that the composite of their actions may define each individual. We may then use these customer representations to solve any arbitrary machine learning problem with a single generalized modeling approach that does not require manual feature engineering, thus accelerating model development timelines, increasing model performance, and decreasing model deployment costs.

### 0.3. **Contributions.**

We contribute the following novelties:

1. "Modular Field Embedding System": A unique modular deep learning architecture capable of representing and pre-training on arbitrarily complex "field types", even with nullable field values.
2. "Tensor Fields": Prebuilt modules that can represent the following field types:
    - Categorical
    - Numeric (of any complex distribution without requiring any preprocessing such as normalization or standardization)
    - Timestamps (their seasonality, such as day of week, week of year, time of day)
    - Entities (such as graph-like string identifiers, such as "IP Addresses", "Login Session IDs", or "Merchant Names")
3. "Random Event-Level Sampling": A novel observation sampling technique capable of generating significantly more unique observations than previous methods with minimal batch preprocessing and an associated proprietary data streaming library capable of streaming larger-than-memory, arbitrarily complex fields.
4. "Partial Field Pruning": A novel technique to provide enhanced model explainability with efficient field pruning.

Lastly, we apply these novel techniques within the internally codenamed "Windmark" machine-learning pipeline, which constructs, pre-trains, and fine-tunes custom complex sequence models directly from a ledger to yield state-of-the-art results.

### 0.4. **Terminology.**

It will be helpful to define some terminology before going forward. We will discuss the architecture of a *complex sequence model*. A complex sequence model trains on *observations*, each created from a slice of a *sequence*. A sequence contains multiple *events*, in which each event may have any number of *fields*. Each field has an associated *field type*. We may refer to this data as "sequential" or "transactional." It is, however, a sequence of multivariate events with nullable, heterogeneous fields that exists within a *ledger*.

The population of sequences exists in one or more database tables, in which each event is a record. We refer to this source of raw data as a ledger. We stream through the ledger, identify the unique sequence IDs (i.e., customer IDs), and find all the events within each sequence. Each event may include one or more unique fields. Every field

exists as a column in this table. The values of each field are nullable; any field value can be empty, which the model must be able to represent and learn as distinct and information-bearing values.

We then stream over each sequence, sampling events from each sequence. Each event has an equal probability of being sampled, as determined by a hyperparameter. After sampling an event (the *anchor*), we take that event and the $L-1$ preceding events to form the *context* for a training sample. If there are not $L-1$ many preceding events before the sampled event, then the context is equal to however many events are available. We then pad and collate the context by collecting every field value within the context to create an *observation* of length $L$. As an analogy to the domain of language modeling, $L$ is similar to max_seq_length.

We take the associated *target* value from the sampled event for any supervised modeling task. In other words, using this methodology, we may support event-level targets for either classification or regression.

We create $N$ many observations from other sampled events, some of which may be (but need not be) from the same sequence, to create a training batch, which we then pass into the sequence encoder. We then pre-train and fine-tune the sequence model with the observations.

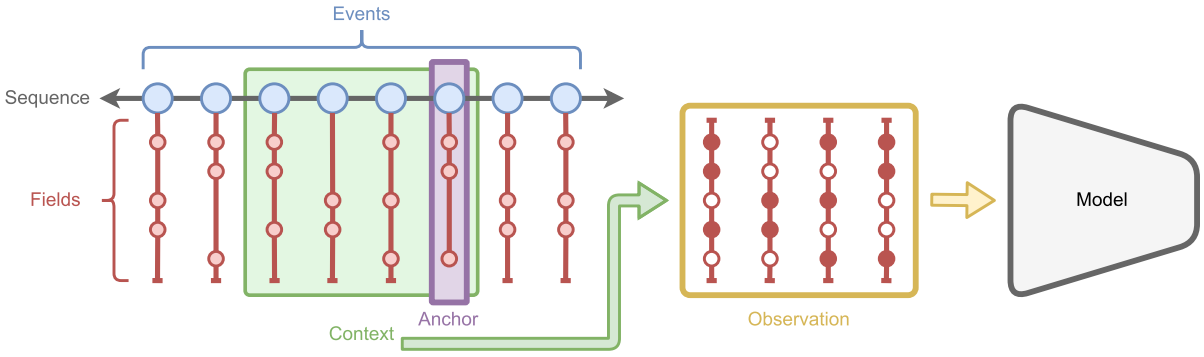


FIGURE 2. An observation being sampled and collated from a sequence

The model architecture, which we later discuss in detail, utilizes transformer-encoder blocks like BERT [1]. We fine-tune the complex sequence model to create a representation of the events within the context for modeling arbitrary supervised learning tasks. This approach differs from models like GPT, which utilize transformer-decoder blocks to predict subsequent tokens. Additionally, the data structure is similar to a sequence of word-piece tokens used to train BERT but with two significant differences:

1. You may conceptualize a list of word-piece tokens as events with a single discrete field, whereas multivariate sequential data may include more than one field. In other words, this model architecture encodes embeddings of shape $(\mathbb{R}^{N\times L\times F\times C})$, whereas BERT encodes embeddings of shape $(\mathbb{R}^{N\times L\times C})$.
2. BERT samples observations at a sequence level, but we sample observations from sequences at an event level. For example, a customer with 10,000 events should create approximately ten times more observations than a customer with only 1,000 events because a randomly sampled event defines each observation. This technique enables the model to support fine-tuning with targets uniquely defined for each event.

0.5. **Tensor Notation.** Given that we are talking about a rather complicated model structure, here is a reference to the common dimensions used for the tensors:

| Tensor Notation | Hyperparameter Value Name | Definition |
|---|---|---|
| $N$ | `batch_size` | Batch Size |
| $L$ | `n_context` | Maximum Context Size of each Observation |
| $F$ | `n_fields` | Number of Fields |
| $C$ | `d_field` | Hidden Dimension for each Field |
| $FC$ | `d_field * n_fields` | Hidden Dimension for each Event |

TABLE 3. Tensor notation and definitions for reference

0.6. **Background.**

0.6.1. *TabBERT.* In early 2021, developers at IBM published *Tabular Transformers for Modeling Multivariate Time Series* (Padhi 2021), in which they described a novel model architecture (TabBERT) that could outperform GBMs trained on handcrafted tabular features for a set of supervised problems with sequential data. [2]

In essence, they frame each sequence as a two-dimensional array. One dimension of the array represents each event in the sequence, and the second dimension represents each field in each event, thus having a dimensionality of $\mathbb{W}^{N\times L\times F}$ when collated and batched together to form a training observation. In their implementation, every single field is limited to discrete tokens. They embed the discrete field values like BERT.

BERT embeds a 1D discrete input of tokens (a sentence of words) into 2D space (an array of embeddings):

```python
def embed_1d(tokens: Int[torch.Tensor, "N L"]) -> Float[torch.Tensor, "N L C"]:
    ...
```

Whereas TabBERT embeds a 2D discrete input (a sliced sequence of events) into 3D space:

```python
def embed_2d(tokens: Int[torch.Tensor, "N L F"]) -> Float[torch.Tensor, "N L F C"]:
    ...
```

TabBERT utilizes a hierarchical transformer-encoder architecture with two encoders. The first encoder (in our terminology, the *field encoder*) allows every field embedding ($\mathbb{R}^{N\times L\times F\times C}$) to *attend* to every field embedding within an event. They then concatenate the contextualized field embeddings to represent the event as a whole ($\mathbb{R}^{N\times L\times FC}$). The second encoder (in our terminology, the *event encoder*) then attends every event representation to one another. They also use the contextualized event representations for a self-supervised learning task for model pre-training similar to BERT's masked language modeling (MLM) [1]. The authors defined a custom pre-training task in which they (1) mask a portion of the events and fields (2) utilize the event representations outputted from the event encoder to reconstruct the masked events and fields. During fine-tuning, they pass the contextualized event representation through an LSTM network to solve arbitrary supervised learning tasks.

A great analogy for hierarchical transformer architecture comes from computer vision. Ideally, if one is building an image representation, each pixel should interact with the other. However, if one is building a representation of a video, it would be virtually impossible to allow every pixel of every frame to interact with one another. Instead, it is significantly more efficient to first build a representation of each frame by letting each of its pixels interact with one another and then build a representation of the video by allowing each frame representation to interact. Similarly, TabBERT decomposes a large and complex operation into two smaller operations: contextualizing representations of each event and then using those contextualized event representations to represent the sequence as a whole.

The hierarchical transformer architecture dramatically reduces the memory complexity of the model. Self-attention is notorious for its quadratic memory complexity (both during training and inference) with respect to context size. By utilizing two transformer-encoders to attend the fields and events separately, we do not require extensive memory requirements for sufficiently long contexts. In effect, we have reduced the memory complexity from a worst-case of $O\left((LF)^2\right)$ into a less prohibitive $O(L^2F + LF^2)$. Within the typical ranges of $L$ (128 to 512) and $F$ (5 to 20), this reduces memory requirements by a bit over an order of magnitude.

TabBERT, while elegant, did come with some major drawbacks. The most significant limitation is that TabBERT coerces continuous fields ($\mathbb{R}^{N\times L\times F}$) into a categorical data type ($\mathbb{W}^{N\times L\times F}$). For example, the model cannot represent the dollar amounts associated with a transaction unless you bin them into categorical levels:

| Continuous Input | Discrete Output |
|---|---|
| \$4.95 | `"$x<$5"` |
| \$8.50 | `"$5<x<$10"` |
| \$19.95 | `"$10<x<$20"` |
| \$103.32 | `"$100<x<$200"` |

TABLE 4. How one might discretize continuous dollar amounts to categorical values

Discretizing every field comes with significant model performance costs. Much of gradient boosting machines' success comes from their ability to define arbitrary cutoff points for continuous features. For example, in some fraud patterns, fraudsters might prefer gift cards, which commonly come in denominations of exactly \$25, \$50, or \$100.

In such a fraud pattern, transactions with dollar amounts equal to exactly those values may be far more likely to be associated with known fraud patterns than transactions with dollar amounts with, say, values around $25.08, $49.12, or $99.53. A GBM may chase the residuals around these exact cutoffs in the continuous distributions. For some tasks, discretizing the raw continuous fields could worsen model performance because the high-frequency nature of continuous fields could be essential.

Additionally, neither the model nor the loss function knows the ordinal nature of the binned field values. For example, if a masked dollar amount was originally $9.95 but the model predicts that the value is in the bucket `"$10<x<$20"`, the loss function harshly penalizes the model even though the model was almost correct. In other words, the loss function is unaware that the model's prediction of `"$10<x<$20"` is not as incorrect as a prediction of `"$100<x<$200"`.

Lastly, LSTM networks are prone to exploding gradients and require linear time complexity with respect to context size, such that training them can be slow.

0.6.2. *UniTTab.* In 2023, Prometeia, a small Italian consulting firm, published *One Transformer for All-Time Series: Representing and Training with Time-Dependent Heterogeneous Tabular Data*, in which they introduced the UniTTab architecture. Their novelties included using Fourier feature encodings to represent floating point field values without discretizing them. They referenced the NeRF paper (Neural Radiance Fields), which highlighted how Fourier feature encodings provide neural networks with a better representation of floating point values than passing in raw scalar values. [3–5]

Fourier feature encodings effectively embed the true value of continuous inputs. They are quite similar to the sinusoidal positional embeddings used by the original transformer-based models [1]. The authors of *Fourier Features Let Networks Learn High-Frequency Functions in Low Dimensional Domains* discussed this approach at a NeurIPS 2020 spotlight. [5]

UniTTab also adapted TabBERT's custom self-supervised learning task to work with continuous fields. Instead of reconstructing the raw scalar values of each field via regression, UniTTab simplified the problem into one of ordinal classification, in which the model attempts to determine the discretized bin of masked continuous field values. They modified the loss function to "smoothen" the loss if the predicted bin is "closer" to the actual value, a technique that they described as "neighborhood loss smoothing."

UniTTab's improvements over TabBERT resulted in significantly better performance; however. The authors did not share their source code. Additionally, they were vague in how they implemented their "row type" embedding system. They require that each event be associated with a registered "row type" and that each "row type" has a required, non-nullable schema, which requires an extraordinarily complex implementation to be able to be trained with accelerated hardware. We were unable to reproduce their results.

### 0.7. **Consolidating Complex Field Types with the Modular Field Embedding System.**

TabBERT and UniTTab both contributed a great deal to complex sequential modeling. TabBERT introduced the idea of hierarchical attention to capture the two-dimensional nature of multivariate sequences. UniTTab improved upon TabBERT to utilize continuous fields. [3, 2]

We introduce a novel approach to consolidate the embedding strategies of the previous papers with a focus on automation and extensibility, providing the following benefits:

1. Combine the abilities of the above papers (supporting discrete, continuous field types) while providing improved support for nullable heterogeneous fields without requiring a set schema for each "row type."
2. Introduce a unique representation of "entities" and geospatial coordinates. We define an "entity" as a discrete input with a finite but impractically large input space, such as a device ID, a login session, a phone number, or a merchant name.
3. Support arbitrarily complex field types, such as text, images, video, and audio, by leveraging foundation models to embed field values for each universal modality as required.

To embed the wide variety of these fields, we have developed the Modular Field Embedding System (MFES) to initiate and route modular field types. The MFES manages a set of unique field embedders for each supported field type with extensibility in mind, so embedders for new field types may be independently managed as a "plugin" system.

As stated above, UniTTab supports discrete and continuous fields through a system the authors call "row types," in which each event must have a specific type with a rigid, non-nullable structure. This system is inflexible and not necessarily intuitive to implement either. We offer an alternative approach that does not require any concept

of "row types" but instead relies upon "field types." Instead of trying to define the unique "type" of each event, we view the data structure from the perspective of its unique fields.

A simple analogy comes from the concept of *dataframes*. You might view a data frame as a list of tuples, each tuple being a row. Alternatively, you could view a data frame as a dictionary of lists, each being a column. In this analogy, UniTTab elected to focus on the row-major structure of lists of tuples to avoid representing null values. We developed a tensor representation around the field types similar to a column-major data frame representation. This approach allows for extensible methodology by which we later define arbitrarily complex field types.

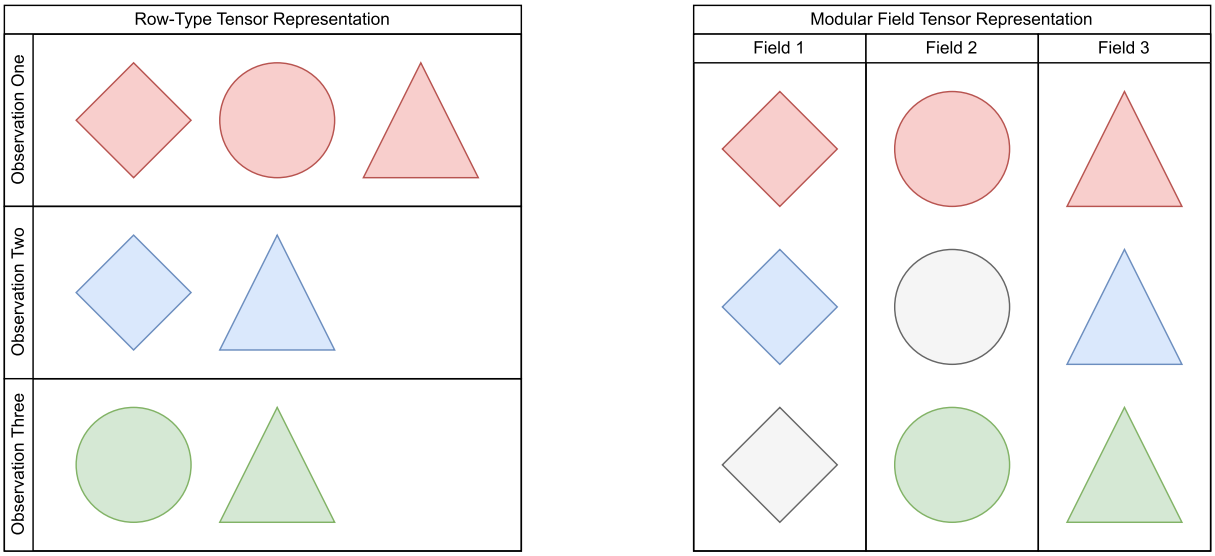


FIGURE 3. A "Row-Type" Tensor Representation compared to a "Field-Type" Tensor Representation

We define "field types" instead of creating "row types" like UniTTab. Each field type is entirely modular. A type-checked `tensorclass` (a `dataclass` of tensors) contains all the necessary information to embed every field value's valued and non-valued states. Each modular field type requires two components:

1. A `TensorField` implementation with the following methods:
    - `TensorField::new`: Pad, pre-process, and collate a streaming array of raw data sampled from the ledger to initiate the `TensorField`.
    - `TensorField::mask`: Mask some portion of events and fields during pre-training.
    - `TensorField::postprocess`: Convert field representation into target predictions during pre-training.
    - `TensorField::mock`: Create random mock data for unit testing.
2. A `FieldEmbedder` module to create the field embeddings from a `TensorField` instance.

After defining a `TensorField` and `FieldEmbedder` class, and registering it to the `FieldInterface` class, one may integrate their own custom field types into the MFES.

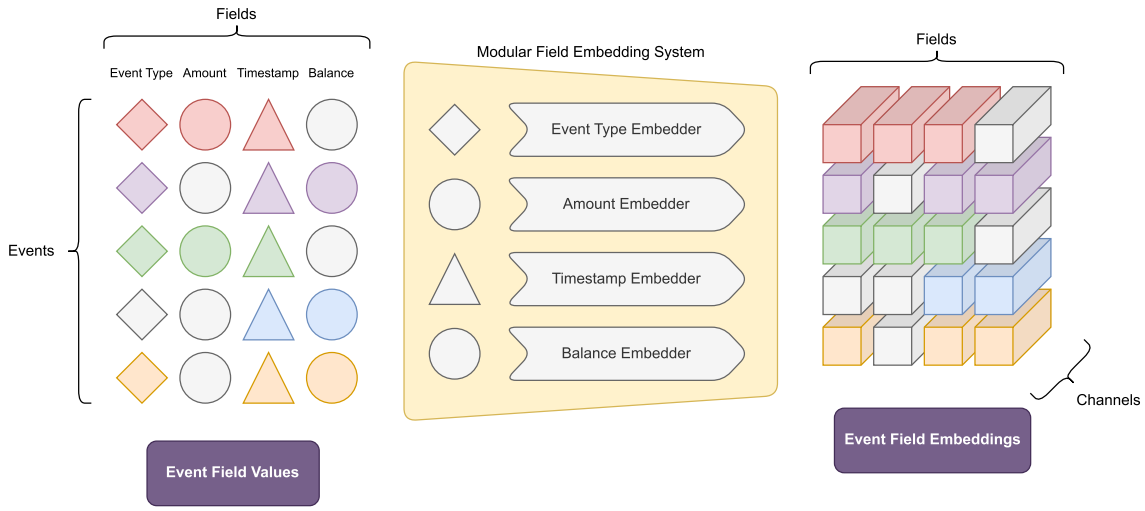


FIGURE 4. The MFES can embed fields of arbitrary types

This approach assumes that each field type uniquely represents its non-valued states (padded, masked, or otherwise unavailable) without overlapping with the valued state. Abiding by this assumption is the cost of viewing the data from a column-major perspective.

We must ensure that each field uniquely represents all possible states. There are four unique possible states for any given field:

1. A field value is present (the field is "valued").
2. No field value is present because the field's event was padded (the field is "padded")
3. No field value is present because there wasn't a field value in the data source (the field is "null")
4. There may or may not be a field value present, but the model is not allowed to see it (the field is "masked")

0.7.1. *Discrete Field Embeddings.* Suppose one intends to initiate a simple model architecture that utilizes four discrete fields per event: `is_foreign`, `is_online`, `merchant_category`, and `transaction_type`.

We use torch's `TensorDict` and `tensorclass` implementations to represent a collection of the fields. For those unfamiliar with these constructs, they are almost exactly how they sound. A `TensorDict` is a pytorch dictionary that can contain instances of `torch.Tensor`. A `tensorclass` may be used similarly to `dataclass`. We use it to create a unique, type-checked instance for each field type.

Below is an example of the input to a complex sequence model with four discrete fields. A `TensorField` called `DiscreteField` contains the input for each discrete field.

```python
@tensorclass
class DiscreteField(TensorField):
    lookup: Int[torch.Tensor, "N L"]

    @classmethod
    def new(cls, ...) -> "DiscreteField":
        ...

    def mask(self, ...):
        ...

# example input with a batch size of 2, sequence length of 8, and 4 discrete fields
TensorDict({
    is_foreign: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    is_online: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    merchant_category: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
```

```
18 })
```

The usage of `TensorDict` and `tensorclass` enable us to input arbitrarily complex fields of heterogeneous types. Methods may be attached to each `tensorclass` to support field-type specific operations, such as preprocessing, masking, and defining the targets for the self-supervised learning task.

The modular field embedding system for this field schema:

```python
ModularFieldEmbeddingSystem(
    (embedders): ModuleDict(
        (transaction_type): DiscreteFieldEmbedder(
            # six unique transaction types plus padded, null, or masked
            (embeddings): Embedding(9, 12)
        ),
        (merchant_category): DiscreteFieldEmbedder(
            # twenty unique transaction types plus padded, null, or masked
            (embeddings): Embedding(23, 12)
        ),
        (is_foreign): DiscreteFieldEmbedder(
            # yes, no, padded, null, or masked
            (embeddings): Embedding(5, 12)
        ),
        (is_online): DiscreteFieldEmbedder(
            # yes, no, padded, null, or masked
            (embeddings): Embedding(5, 12)
        ),
    )
)
```

Upon inputting the above `TensorDict` into the `ModularFieldEmbeddingSystem`, the MFES will iterate over each of the four discrete fields, embedding each of them with their respective `DiscreteFieldEmbedder` to create four tensors, each of dimensionality $\mathbb{R}^{N\times L\times C}$. The `ModularFieldEmbeddingSystem` module concatenates the embeddings into a single tensor of dimensionality $\mathbb{R}^{N\times L\times F\times C}$, which we then pass into the field encoder.

Representing the non-valued states (padded, null, or masked) is trivial for discrete fields. There are additional tokens available within the `DiscreteFieldEmbedder`'s embedding table. With three exceptions, this technique is almost precisely like how one embeds discrete tokens for models like BERT:

1. Each unique field embedding module has an independent embedding table for these non-valued states.
2. No `[CLS]` token exists in the field embedding module. We append the equivalent of a `[CLS]` token to the event representations, which we then input to the event encoder module. The field encoder would not add value to a `[CLS]` token.
3. Positional embeddings are required for both the field and event encoders. The field encoder's positional embeddings will learn to uniquely represent each unique field (I.E., what *is* a POS transaction dollar amount or an MCC code). The event encoder's positional embeddings will learn to represent the order of the events within the observation. The positional embeddings for the field and event encoders are tensors of dimensionality $\mathbb{R}^{F\times C}$ and $\mathbb{R}^{L\times FC}$, respectively.

0.7.2. *Continuous Field Embeddings.*

UniTTab's approach to embeddings avoids the complications of representing unknown, padded, or masked continuous field values by enforcing a rigid structure for each "row type." We found this to be challenging to implement in practice. Instead, our approach fully represents continuous fields in their native form as, effectively, `enum` data structures with non-valued states:

```rust
enum NullableNumber {
    Valued(f32),
    Unknown,
```

```
    Padded,
    Masked,
}
```

In other words, a continuous field value can either contain a real number or be non-valued, in which case it is either padded, masked, or otherwise null. We represent this `enum`-like structure with continuous fields with two tensors: `values` and `lookup`. `values` is of dimensionality $\mathbb{R}^{N\times L}$, and `lookup` is of dimensionality $\mathbb{W}^{N\times L}$.

The `values` tensor contains the cumulative distribution function (CDF) of the original floating point values scaled by $0.\overline{9}$.

$$\overline{x} = F_X(x) \cdot 0.\overline{9}$$

This guarantees that $\overline{x} \in [0, 1)$. Wherever a field is non-valued (either null, padded, or masked), we impute `values` as `0.0`. Using the CDF of continuous field values instead of the raw scalar values, we do not require data standardization/normalization operations to represent values of arbitrary distributions. The CDF of the `values` tensor also provides additional benefits that we describe later in this section.

The `lookup` tensor contains one of four integers to explain the state of each field (valued, padded, masked, or otherwise null). In practice, we define the definitions of these four unique integers within a Python `IntEnum`:

```python
class SpecialTokens(enum.IntEnum):
    VAL = 0 # the field is "valued"
    UNK = 1 # the field is otherwise "null"
    PAD = 2 # the field is "padded"
    MASK = 3 # the field is "masked" for pre-training
```

This token definition is the source of truth for every field. It is important to note that the `[VAL]` token is mapped to 0, which allows for an elegant optimization trick described later in this section.

Now that we have these two tensors, `values` and `lookup`, we may combine them in a new `tensorclass` to represent any nullable continuous field:

```python
@tensorclass
class ContinuousField(TensorField):
    lookup: Int[torch.Tensor, "N L"]
    value: Float[torch.Tensor, "N L"]
```

Consider how every possible state of `enum NullableNumber` maps to an instance of `ContinuousField`:

| Input Field Value | Output Tensorclass Representation |
|---|---|
| `Padded` | `ContinuousField(lookup=[[2]], values=[[0.00]])` |
| `Valued(0.93)` | `ContinuousField(lookup=[[0]], values=[[0.93]])` |
| `Valued(0.65)` | `ContinuousField(lookup=[[0]], values=[[0.65]])` |
| `Unknown` | `ContinuousField(lookup=[[1]], values=[[0.00]])` |
| `Valued(0.31)` | `ContinuousField(lookup=[[0]], values=[[0.31]])` |
| `Unknown` | `ContinuousField(lookup=[[1]], values=[0.00])` |
| `Valued(0.00)` | `ContinuousField(lookup=[[0]], values=[[0.00]])` |
| `Masked` | `ContinuousField(lookup=[[3]], values=[[0.00]])` |

TABLE 5. How continuous inputs can uniquely map to a pair of tensors

This mapping guarantees that every possible input has a unique tensor representation. Like UniTTab and NeRF, we embed the `values` tensor with Fourier feature encodings. This process looks like the following:

$$B = 2^{[-8..3]}$$

$$\gamma(\overline{x}) = [\cos(\pi B\overline{x}), \sin(\pi B\overline{x})]$$

$$\text{FourierEncoding}(v) = \text{MLP}(\gamma(\overline{x}))$$

In other words, we embed the continuous fields by multiplying the tensor `values` with a sequence of bands, `B`, defined as $2^{[-8..3]}$, and then multiplying that again with `pi`. We then compute `sin(value * B * pi)` and `cos(value * B * pi)` and concatenate these two sequences to create `gamma`. We then pass this tensor through a simple MLP to represent the tensor `values` to the neural network.

However, there is a small problem. Because we have imputed the non-valued fields to `0.0`, the model cannot differentiate a value that is masked, padded, null, or is a value but happens to be equal to `0.0`. The solution is surprisingly simple. We subtract the `lookup` tensor from the `values` tensor and instead embed the resulting tensor with FourierEncoding. The model will learn that values equal to `-1.0`, `-2.0`, and `-3.0` are each of a non-valued representation and that the floating point inputs in the range of $[0, 1)$ are of valued fields.

| Input Field Value | Output TensorRepresentation |
|---|---|
| `Padded` | `torch.Tensor([[-2.0]])` |
| `Valued(0.93)` | `torch.Tensor([[0.93]])` |
| `Valued(0.65)` | `torch.Tensor([[0.65]])` |
| `Unknown` | `torch.Tensor([[-1.0]])` |
| `Valued(0.31)` | `torch.Tensor([[0.31]])` |
| `Unknown` | `torch.Tensor([[-1.0]])` |
| `Valued(0.00)` | `torch.Tensor([[0.00]])` |
| `Masked` | `torch.Tensor([[-3.0]])` |

TABLE 6. How complex continuous input parts can uniquely map to real values

The valued and non-valued states are guaranteed not to collide for the following two reasons:

1. The special `[VAL]` token is represented by a value of 0.
2. The `values` tensor has a limited range of $[0, 1)$.

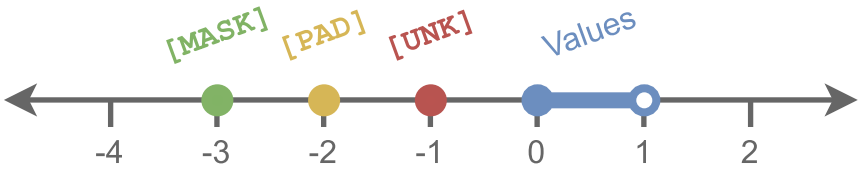


FIGURE 5. Every possible input value to FourierEncoding($v$)

This method also comes with an elegant runtime assertion that validates the two tensors `values` and `lookup` for every continuous field. Because we had set the `[VAL]` token itself equal to `0` and imputed non-valued inputs to `0.0`, we can *guarantee* that the element-wise product of `lookup` and `values` will always equal `0.0`. The following assertions will check if the field's contents are invalid:

```python
assert torch.all(value.mul(lookup).eq(0.0)) ,\
    "Every value should be imputed to 0.0 if not null, padded, or masked"

assert torch.all(value.lt(1.0)) ,\
    "Every value should be strictly less than 1.0"

assert torch.all(value.ge(0.0)) ,\
    "Every value should be greater than or equal to 0.0"
```

In other words, if one were to input a non-representable state to `enum NullableNumber`, this assertion would loudly fail.

There is one last caveat to this approach regarding the precision of the floating point `values` tensor. Fourier feature encodings have a limited range, which is yet another reason we use the CDFs of the original floating point values. As previously mentioned, UniTTab requires an ordinal classification pre-training task. In other words, the model will classify which quantile bin a masked value should fall into. If we chose ten quantile bins, we would need to determine which decile each of the masked values belongs. Because the choice of quantile bins is a pre-training hyperparameter, $q$, it is better to calculate the CDF of the original values for the pre-training task. Therefore, when we choose a value for $q$, we can determine the appropriate bucket index by calculating $\lfloor q * F_X(x) * 0.\overline{9} \rfloor$. Scaling the inputs by $0.\overline{9}$ guarantees that the flooring operation will always round down. Consequently, we can use the CDFs for the continuous field inputs and then modify the CDFs for the targets.

In practice, Ted Dunning's TDigest algorithm can quickly and efficiently calculate the CDFs of the original values as a *streaming* operation during both model training and inference, so we do not have to calculate the CDFs as a batch operation. [6] This operation also significantly reduces storage requirements because we do not have to store the normalized or binned value targets. We only have to store the original values.

[1]

This novel strategy efficiently embeds a nullable continuous field of an arbitrarily complex numeric distribution with negligible marginal computation.

The following `TensorDict` contains multiple heterogeneous fields.

```python
@tensorclass
class ContinuousField(TensorField):
    lookup: Int[torch.Tensor, "N L"]
    value: Float[torch.Tensor, "N L"]

# example input with a batch size of 2, context size of 8, and 2 continuous fields, 1 discrete field
TensorDict({
    amount: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    balance: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

The `ModulerFieldEmbeddingSystem` utilizes multiple instances of `ContinuousFieldEmbedder`, each dedicated to embedding a specific `ContinuousField`.

```python
ModularFieldEmbeddingSystem(
    (embedders): ModuleDict(
        (amount): ContinuousFieldEmbedder(
            (linear): Linear(in_features=16, out_features=12, bias=True)
            # masked, null, padded, or valued
            (positional): Embedding(4, 12)
        ),
        (balance): ContinuousFieldEmbedder(
            (linear): Linear(in_features=16, out_features=12, bias=True)
            # masked, null, padded, or valued
```

[1] We found that the library `pytdigest`, developed by Tomas Protivinsky, provides a blazingly fast implementation of the TDigest algorithm. [7] All other implementations we experimented with are far too slow, such that the streaming CDF calculations bottleneck model training and inference.

```
            (positional): Embedding(4, 12)
        ),
        (transaction_type): DiscreteFieldEmbedder(
            # six unique transaction types plus padded, null, or masked
            (embeddings): Embedding(9, 12)
        ),
    )
)
```

As before, this instance of `ModularFieldEmbeddingSystem` with heterogeneous field types will iterate over each of its three fields, embedding them individually and then concatenating their embeddings to create a single embedding for the entire batch of dimensionality $\mathbb{R}^{N\times L\times F\times C}$ where $F = 3$.

0.7.3. *Temporal Field Embeddings.* Working with dates and timestamps might seem challenging, but we can approach it similarly to how we handled continuous fields.

We parse timestamps into three input parts:

1. The week of the year: An integer between 1 and 53.
2. The day of the week: An integer between 1 and 7.
3. The minute of the day: An integer between $24 \cdot 60$.

These three input parts capture the signal regarding event seasonality. Unlike continuous fields, each of these three inputs fits a known, fixed range such that a calculation of the CDFs is not required.

The discrete input parts (week of the year and day of the week) are embedded via a `TemporalFieldEmbedder` using standard learned embeddings.

However, the minute of the day transforms into a floating point value within the range $[0, 1)$ by dividing it by $24 \cdot 60 + 1$. The minute of the day is then embedded via a FourierEncoding like continuous field values.

The discrete input parts (week of the year and day of the week) each have their non-valued tokens (`[MASK]`, `[UNK]`, and `[PAD]`). The continuous input part (minute of the day) allows for non-valued inputs by utilizing a `lookup` tensor like those used to represent the discrete part of continuous fields. However, this is redundant because the two tensors `week_of_year` and `day_of_week` can each differentiate when a field is non-valued.

Python

```
@tensorclass
class TemporalField(TensorField):
    lookup: Int[torch.Tensor, "N L"]
    week_of_year: Int[torch.Tensor, "N L"]
    day_of_week: Int[torch.Tensor, "N L"]
    minute_of_day: Float[torch.Tensor, "N L"]

# example input with a batch size of 2, context size of 8
# 1 temporal field, 1 continuous field, 1 discrete field
TensorDict({
    timestamp: TemporalField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        week_of_year=Tensor(torch.Size([2, 8]), torch.int),
        day_of_week=Tensor(torch.Size([2, 8]), torch.int),
        minute_of_day=Tensor(torch.Size([2, 8]), torch.float),
    ),
    amount: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

During pre-training, the model will attempt to find the correct hour of the year for each masked temporal field. Because there are up to $366 \cdot 24$ hours per year, there are 8784 possible targets. Similar to continuous fields, we smoothened the loss of this semi-ordinal classification task via neighborhood loss smoothing.

Lastly, this timestamp embedding technique can only capture the seasonality of a timestamp. It cannot capture the original timestamp's absolute value nor the duration from the present. For example, if we pass in a field containing timestamps of the time a transaction occurred, the model will not know exactly when the transaction occurred. The model will only be able to learn the seasonality regarding the day of week, day of year, and time of day. This choice is intentional to help the model generalize to out-of-time observations.

As an alternative to passing in the raw timestamp data, one might instead include a continuous field defined as the number of seconds between the current and previous events. Using such a "duration" instead of a raw timestamp may help prevent the model from overfitting on irregular activity that is limited to a small window of time that we do not expect to generalize to the population of training data.

For example, suppose one is trying to determine whether a transaction is associated with account takeover fraud. The model developers are working with transactions that occurred between 2020 and 2023. However, in July of 2021, there was an enormous spike in fraud cases for one reason or another. Consequently, the model might learn that all transactions in July are more risky than transactions in other months. In practice, however, this pattern will not generalize to out-of-time observations.

To solve the same problem, instead of using a "duration" field, the model developer could include a continuous field representing the minute of the day.

0.7.4. *Geospatial Field Embeddings.* Working with Geospatial data poses a unique challenge. However, we can again use a technique similar to the continuous field embedding mechanism. Geospatial coordinates contain two numbers: longitude and latitude. Both longitude and latitude are of fixed domains ($[-180, 180]$, $[-90, 90]$, respectively), so a `TDigest` model is not required to normalize these values as a streaming operation.

```python
@tensorclass
class GeospatialField(TensorField):
    lookup: Int[torch.Tensor, "N L"]
    longitude: Float[torch.Tensor, "N L"]
    latitude: Float[torch.Tensor, "N L"]

# example input with a batch size of 3, context size of 64
# 1 geospatial field, 1 temporal field, 1 continuous field, 1 discrete field
TensorDict({
    location: GeospatialField(
        lookup=Tensor(torch.Size([3, 64]), torch.int),
        longitude=Tensor(torch.Size([3, 64]), torch.float),
        latitude=Tensor(torch.Size([3, 64]), torch.float),
    ),
    timestamp: TemporalField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        week_of_year=Tensor(torch.Size([2, 8]), torch.int),
        day_of_week=Tensor(torch.Size([2, 8]), torch.int),
        minute_of_day=Tensor(torch.Size([2, 8]), torch.float),
    ),
    amount: ContinuousField(
        lookup=Tensor(torch.Size([3, 64]), torch.int),
        values=Tensor(torch.Size([3, 64]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([3, 64]), torch.int)),
})
```

However, using raw geospatial coordinates is not always the best option. If a model developer would like to create representations of ride-share drivers based on their previous trips, they might include the pickup and dropout geospatial coordinates for each trip. Alternatively, they could utilize the ride distance as a continuous field or the zip code as a discrete field.

Regardless, incorporating raw geospatial coordinates into the model may allow for faster inference and improved model performance depending on each use case.

0.7.5. *Entity Field Embeddings.* Entities are prevalent in production datasets. By an entity, we refer to an item existing in an infinitely large set of items. While similar to categorical data, the set of items is so large or ephemeral that a unique embedding cannot uniquely represent each level in the category.

For example, whenever a customer logs into their account, a server might create a temporary ID unique to that web session. We associate this web session ID with all the events during the same web session. However, because the web session is temporary, the session IDs available in the training data should never exist in production data during inference. However, we want our model to utilize this session ID to see which events happened during each session.

We might also consider each unique "merchant" as an entity. We might want to be able to allow the model to see which purchases happened at each merchant. The "merchant" field differs from "merchant name", which may include a rigorously cleaned string like `"McDonald's"`. A "merchant" may also include the address or store number, allowing for extreme detail. Millions of unique merchants may exist in a large enough point-of-sale (POS) dataset.

Another example of an entity is the "account" of a customer or the "customer" of an account. For some businesses, an "account" and a "customer" are not synonymous. If one is defining a customer-level model (such that the events posted by a customer define a sequence), but customers may have more than one account, they could represent each account available to a customer with an entity. The opposite is true; if one is designing an account-level model (such that an account's posted transactions define a sequence), but more than one customer may access each account, they could represent each customer ID as an entity.

Lastly, we might consider the devices of each customer to be entities. Device identifiers may be tracked and stored among all online events within a sequence (IP address, unique smartphone application tracker, etc). These device identifiers may allow the model to understand which event happened from which of the customer's devices, providing significant value for protecting customers against theft or fraud.

Effectively tracking customers' locations, devices, and web sessions may be considered a breach of privacy. However, in some cases, such as fraud detection, these data points are critical for optimal coverage to deliver the best customer experience.

The concept of entities is common in recommendation systems. For example, a media streaming service may choose to create an embedding of every single piece of content in its collection and every single user. Recommendation systems utilize multiple large learned embedding tables, which they index with numerous lookup values created by many hash functions. [8] They then aggregate multiple embeddings to form a unique embedding per entity. This approach allows for large recommendation systems to memorize all of the entities. However, there are other ways of accomplishing the same goal.

Instead of using the standard hash functions commonly associated with recommendation systems to tokenize every entity, we contribute a custom hash function to enable the model to differentiate among the entities available within an observation's context.

We don't necessarily need to represent every unique device every customer has ever used. We only need to be able to uniquely represent the devices a customer has used within that customer's current context.

We can guarantee that the total count of unique devices in the context will never exceed the context window size. Therefore, we can simply assign a random integer in the range $0..L$ to each unique device in the context such that each unique device maps to a unique integer. We randomly initiate an entity tokenizer for each observation to prevent the model from overfitting. This tokenization process introduces an element of randomness into the model that the model must overcome by learning to discover the frequently used entities among other events in the observation's context and how the current event relates to these events.

The special tokens (`[MASK]`, `[UNK]`, and `[PAD]`) are always fixed. In other words, we never hash these special values such that we tokenize them to static integers.

```python
def hash(values: list[str]) -> list[int]:
    '''embed list of strings into a list of hashed embedding lookup values
```

```
    '''

    offset = len(Tokens)
    unique = set(values)
    integers = random.sample(range(offset, params.n_context + offset), len(unique))
    mapping: dict[str, int] = dict(zip(unique, integers))

    # Unknown values are represented with an empty string and must be replaced with the fixed [UNK] token lookup value
    mapping.update({"": Tokens.UNK})

    return list(map(lambda value: mapping[value], values))
```

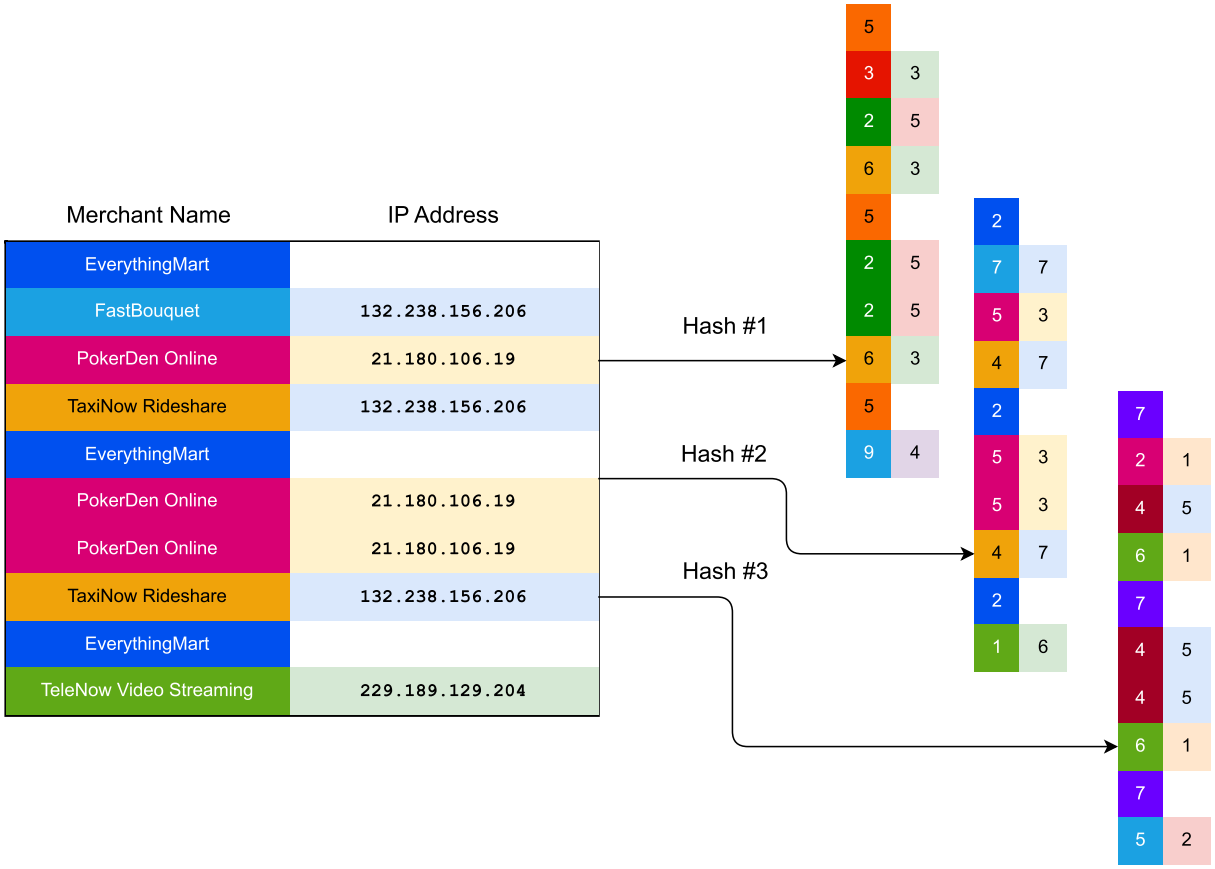


FIGURE 6. How the same observation of two entity fields may result in different lookup tokens during training

During pre-training, the model will learn to utilize the unmasked context to determine the available entities and which available entity is most likely to be associated with the masked entity fields. We prove that self-attention is capable of learning this task.

Using these "anonymous" entity embeddings has a few benefits over the traditional hash embeddings that are commonplace in recommendation systems. For one, they prevent overfitting because the model cannot memorize any entity. Secondly, they prioritize the privacy of customers. Additionally, they do not require large embedding tables, thus dramatically reducing the size of trained models. Lastly, they guarantee that a hash collision will never occur during inference, thus reducing silent model failures.

```python
@tensorclass
class EntityField(TensorField):
```

```
    lookup: Int[torch.Tensor, "N L"]

# example input with a batch size of 3, context size of 64
# 1 entity field, 1 geospatial field, 1 temporal field, 1 continuous field, 1 discrete field
TensorDict({
    device_id: EntityField(lookup=Tensor(torch.Size([3, 64]), torch.int)),
    location: GeospatialField(
        lookup=Tensor(torch.Size([3, 64]), torch.int),
        longitude=Tensor(torch.Size([3, 64]), torch.float),
        latitude=Tensor(torch.Size([3, 64]), torch.float),
    ),
    timestamp: TemporalField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        week_of_year=Tensor(torch.Size([2, 8]), torch.int),
        day_of_week=Tensor(torch.Size([2, 8]), torch.int),
        minute_of_day=Tensor(torch.Size([2, 8]), torch.float),
    ),
    amount: ContinuousField(
        lookup=Tensor(torch.Size([3, 64]), torch.int),
        values=Tensor(torch.Size([3, 64]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([3, 64]), torch.int)),
})
```

This technique is only possible because we are defining a context window with an upper-bound size limit. If we have a context window with a maximum size of 64, then we know there are, at most, 67 maximum possible unique values that we need to represent (64 entities plus the three non-valued states). We are simply trying to differentiate these unique values from one another. Devices, login sessions, and merchants are infinite or ephemeral, and this novel mechanism allows us to represent them as such without storing many different large embedding tables.

Representing "entities" is virtually impossible for tabular machine learning techniques. We posit that entities provide enormous value to practical problems. Many real business problems require tracking complex entities such as IP addresses, phone numbers, email addresses, mailing addresses, device identifiers, login sessions, transaction counter-parties, merchants, and account numbers.

0.7.6. *Putting it all together: MFES and examples.* The MFES can embed discrete, continuous, temporal, geospatial, and entity field types over long sequences. Consequently, we may input sequences that describe all of the transactions ever posted by a customer of a financial institution:

| Device ID | Action | Amount | Location | Timestamp | Comments | Merchant Name |
|---|---|---|---|---|---|---|
| `e8e905fbb00f` | Annual Fee | 395.00 | | 2022-03-19 08:34:02 | Annual Card Fee | BigBank |
| | Purchase | 171.54 | (-74.20453, −158.0078) | 2022-03-21 11:19:53 | Home improvement supplies | HomeCraft |
| | Purchase | 464.69 | (-76.638826, −36.341502) | 2022-04-30 20:43:02 | Charity donation | GiveBack |
| `e8e905fbb00f` | Online Payment | 463.58 | | 2022-09-13 16:21:23 | Hotel booking | StayInn |

| | | | | | | |
|---|---|---|---|---|---|---|
| `f2e31f29343b` | Dispute Transaction | 489.35 | (-34.316692, 170.608224) | 2023-05-01 20:40:15 | Pet supplies | PetWorld |
| `e8e905fbb00f` | Refund | 53.73 | | 2023-06-09 06:55:00 | Refund for returned item | ReturnDesk |
| `e8e905fbb00f` | Balance Inquiry | | | 2023-08-23 01:48:57 | | BigBank |
| | Purchase | 25.00 | (81.202198, 175.041279) | 2023-11-18 19:42:07 | Coffee shop | BrewBeans |
| | Purchase | 251.61 | (-41.539246, 154.148774) | 2023-12-01 22:12:23 | Transfer to friend | PeerPay |
| `e8e905fbb00f` | Balance Inquiry | | | 2024-02-11 09:38:10 | | BigBank |

TABLE 7. An example sequence of mixed field types

This approach is highly versatile. It can work with many different data types. Additionally, it can work with complex sequential data exactly how it already exists. There is effectively zero batch preprocessing other than a simple sorting and sharding operation to facilitate larger-than-memory streaming.

0.8. **Model Encoder Architecture.** Upon instantiating the Modular Field Embedding System, our hierarchical transformer-encoder architecture can learn sequence and event representations. We include two transformer-encoder modules, the "field encoder" and the "event encoder." The field encoder allows each field to attend to every other field within each event. We concatenate the field representations among each event together to form a representation for each event. These event representations then attend to one another in the event encoder to create contextualized event representations and a sequence representation via a special `[CLS]` token. We pre-train the model with these contextualized event representations. We then fine-tune the model by inputting the sequence representations to a small MLP to solve arbitrary downstream supervised learning problems.

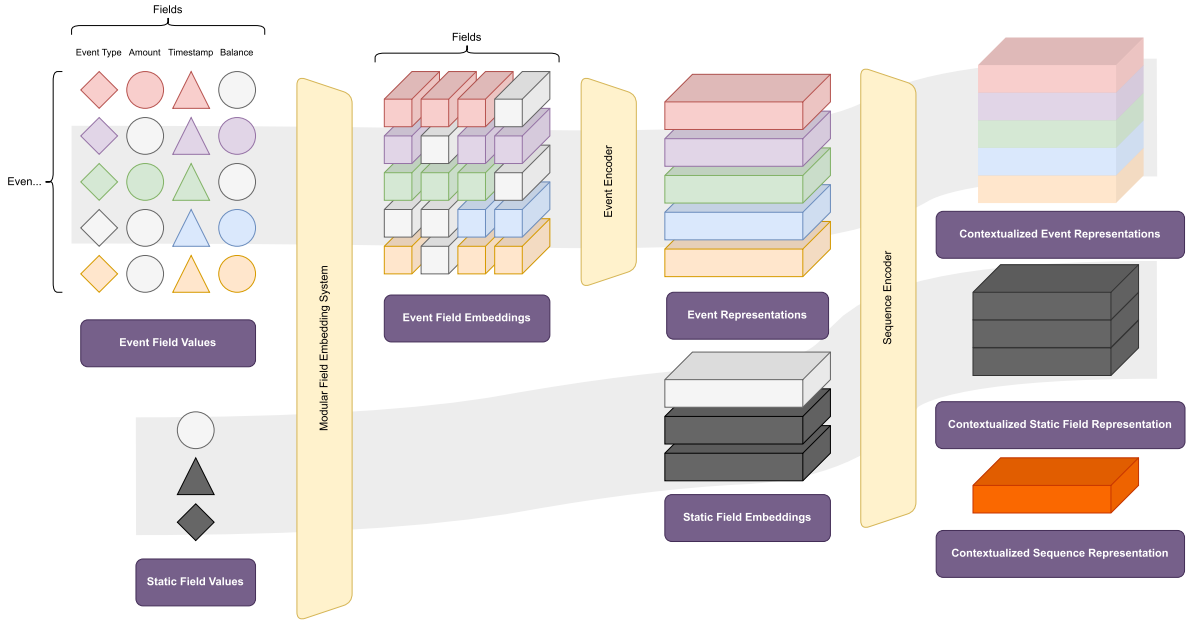


FIGURE 7. The consolidated model architecture outputting a sequence representation

0.9. **Observation Sampling Strategy.**

In the section *Terminology*, we described how we form observations from the ledger. This approach is novel and fundamentally different from published literature.

TabBERT and UniTTab describe their observation sampling strategies as fixed and sampled at an event level. Their observations are "fixed" in that they create the observations as strided slices over each sequence's events during a batch preprocessing stage. Both TabBERT and UniTTab elected to utilize strided slices of events to mitigate potential data leakage, as the observations created from one sequence could end up in the train, validation, or test strata. In other words, they could use one slice from a sequence to train a model, and then a neighboring slice could validate it. Their strategy requires that the observations never overlap to mitigate the risk of data leakage.

This event-sampling technique is unique to the problem of complex sequence modeling. In computer vision, practitioners uniformly sample from the population of images. In language modeling, practitioners uniformly sample from the population of documents in a corpus. However, sampling observations at a sequence level do not necessarily work for complex sequence modeling because some sequences may have very few events.

For example, in the case of a financial institution with 10 million customers, each with a credit card, many credit card users may be "gamers" who only signed up for the card to collect a sign-on bonus. These inactive customers may only have tens of events on their records, whereas other customers may have thousands of events in their records. Sampling uniformly from each customer's sequence will result in sampling the same few transactions from inactive customers a hundred times before sampling any notable portion of the events belonging to more active customers, resulting in a long training time and poor performance because the model over-fits the less active customers and under-fits the more active ones.

We developed a novel approach whereby each sequence is committed to belonging to the train, validation, or test strata. In other words, a sequence can only belong to the training, validation, or testing strata. This intuition was inspired by how traditional machine learning techniques choose to split their observations to prevent data leakage. The benefit of doing so is that observations may be generated randomly from each sequence without concern for data leakage. We believe this improves the model's ability to generalize to out-of-sample sequences, as this strategy disallows the model from ever training on any data available anywhere within the sequences used for out-of-sample analyses.

By generating observations randomly without concern of data leakage, we do not require the creation of observations as a batch operation before model training. In other words, we can sample observations from the sequences as a streaming operation, requiring significantly less work before the model may begin training. Additionally, the data preprocessing does not need to be executed multiple times among consecutive experiments with the proper artifact caching techniques, even if the hyperparameters change.

Additionally, we can generate significantly more unique observations by sampling observations randomly. For example, while a fixed, "strided" observation sampling strategy requires that a model with a context size of $L$ also be "strided" for every $L$ many events, thus creating $c_e \frac{e}{L}$ many unique observations, where $c_e$ is the total count of events in the entire ledger, our methodology can randomly sample $c_e$ many unique observations during both pretraining and fine-tuning. Randomly sampling observations effectively provides a means of data augmentation that theoretically improves model performance and computation requirements.

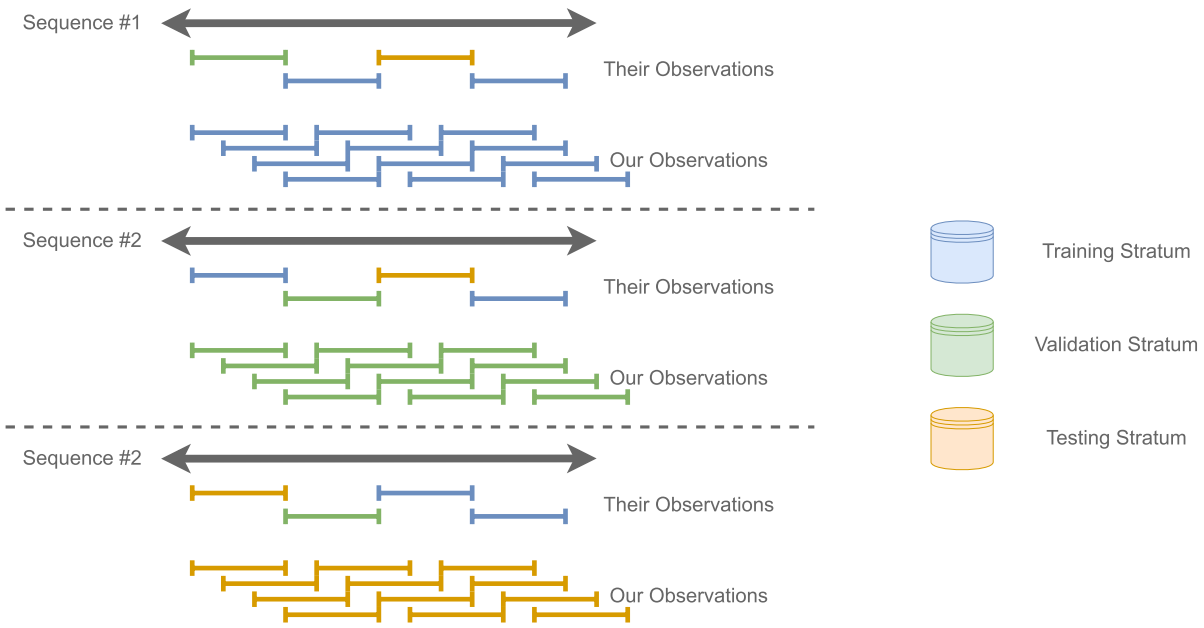


FIGURE 8. Randomly sampling observations from a sequence results in a significantly larger population of unique events

However, as TabBERT and UniTTab highlight, sampling too many overlapping observations from a sequence can quickly result in overfitting. By "overlapping observations," we refer to observations that share many events in the context. As such, during each training epoch, we utilize a "sampling rate" that controls the probability of using any event as the anchor to form an observation. In other words, during each training epoch we will only have $c_e * r_s$ many observations, where $r_s$ is the sampling rate. However, this operation samples with replacement, so each event may form an observation during subsequent epochs. The value of $r_s$ should equal to or less than $\frac{1}{L}$. This hyperparameter provides a convenient means of changing the intended number of training steps per epoch to optimize how frequently one wishes to validate the model. The randomly sampled observations are shuffled with a larger streaming buffer to lessen the risk of potentially overlapping observations from training the model in the same batch.

Lastly, we utilize varying sampling rates for each stratum. In other words, we choose a small sampling rate for training, a medium sampling rate for validation, and a sampling rate equal to 1.0 during testing. Our methodology supports far more robust testing on the out-of-sample test stratum because it will create an observation for every event in every sequence in the test stratum.

0.10. **Self-Supervised Learning Strategy.** One of the many advantages of this network architecture is the opportunity to utilize self-supervised learning (SSL). In many business problems, there is a large amount of unlabeled data that traditional machine learning models cannot use. The modality of language has benefitted enormously from applying masked language modeling (MLM) to pre-train models like BERT. TabBERT, as the name implies, also utilized an adaption of MLM, in which they mask random fields or events.

However, utilizing self-supervised learning to pre-train from a ledger is more complicated because the field types can be heterogeneous. In addition to being able to mask any field, we also need to decode the contextualized event representations back into their original values or a decent proxy of them. Both discrete and entity field types may be pre-trained like the discrete fields of TabBERT because they always contain discrete tokens. Decoding discrete and entity field types is trivial, but we will require complex decoding operations for others.

0.10.1. *Semi-Ordinal Classification of Continuous Fields' Quantiles.* For the continuous fields, instead of attempting to reconstruct the original CDFs, we can reconstruct the binned CDFs. We define a hyperparameter, $q$, that sets the number of quantile bins into which we may split the original continuous fields.

Assuming we want to calculate set $q = 10$, if one of the masked continuous fields has a CDF of `0.05`, the model should learn to predict that this value belongs in the 1st decile. If another masked continuous field has a CDF of 0.51, the model should predict that this value belongs in the 6th decile. Every masked CDF should be mapped to the bin $\lfloor F_X(x) \cdot q \cdot 0.\overline{9} \rfloor$.

However, this approach has an issue. We penalize the model if it guesses the wrong bin, but we also want to encourage the model if it guesses the incorrect bins that are *closer* to the correct bin. In other words, if the model predicts that a masked value belongs in the 8th decile but belongs in the 10th decile, then we should punish the model less than if it had said the masked value belongs in the 1st decile. In other words, we want to run a semi-ordinal classification task. We need to apply the "neighborhood loss smoothing" function from the UniTTab paper; however, we need to modify it to support the non-valued states of null and padded.

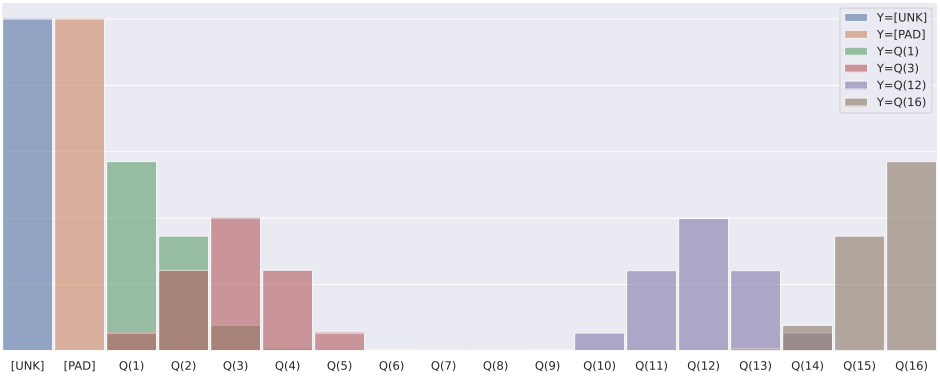


FIGURE 9. How various targets' expected predictions are smoothed

If a masked field is "padded" but the model guessed that the value is in any decile (or is null), we want to penalize the model harshly. Additionally, if a masked field is of any decile, but the model predicted that the masked field is either null or padded, we should fully penalize the model. The "neighborhood loss smoothing" only applies if the masked field is valued, and even then it only applies among valued quantile bins.

0.10.2. *Semi-Ordinal Classification of Temporal and Geospatial Fields.*

Unlike continuous fields, which we reconstruct to $q$ many buckets plus the number of special tokens, temporal and geospatial fields have fixed domains such that we do not calculate the CDFs. Therefore, we utilize their original values to define semi-ordinal targets.

For temporal fields, we attempt to classify the hour of the year present in a timestamp. We normalize latitude and longitude for geospatial fields and create 1,000 buckets for each. We average the losses of these two targets together. In both cases, we smoothen the losses like continuous fields.

| Device ID | Action | Amount | Location | Timestamp | Comments | Merchant Name |
|---|---|---|---|---|---|---|
| e8e905fbb00f | [MASK] | 395.00 | | 2022-03-19 08:34:02 | Annual Card Fee | BigBank |
| | Purchase | 171.54 | (-74.20457, −158.00785) | 2022-03-21 11:19:53 | [MASK] | HomeCraft |
| [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] |
| e8e905fbb00f | Online Payment | 463.58 | | 2022-09-13 16:21:23 | Hotel booking | StayInn |
| [MASK] | Dispute Transaction | 489.35 | (-34.316692, 170.608224) | 2023-05-01 20:40:15 | Pet supplies | PetWorld |
| [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] |
| e8e905fbb00f | Balance Inquiry | | | 2023-08-23 01:48:57 | | BigBank |
| [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] | [MASK] |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Purchase | 251.61 | (-41.539246, 154.148774) | [MASK] | Transfer to friend | PeerPay |
| [MASK] | Balance Inquiry | | [MASK] | 2024-02-11 09:38:10 | | BigBank |

TABLE 8. An example masked sequence

0.11. **Fine-tuning Complex Sequence Models.** Unlike TabBERT or UniTTab, we create a sequence representation from the equivalent of a [CLS] token in BERT. The model learns a tensor of fixed size equal $\mathbb{R}^{N\times 1\times FC}$ and appends it to the event representations, which it inputs into the event encoder. We pass the sequence representation from the sequence encoder to a small decision head (MLP) to train both the sequence encoder and the decision head to solve arbitrary supervised learning problems or improve the quality of the sequence representations for other tasks. In two hours, we were to fine-tune exhaustively pre-trained complex sequence models on modest consumer-grade hardware.

However, because complex sequence encoders utilize case-dependent fields, they cannot produce universal "foundation models." We may only fine-tune our pre-trained complex sequence encoder for supervised problems that use the set of fields with which it was pre-trained or a subset thereof.

0.11.1. *Versatility of Supervised Tasks.* Complex sequence models can support a variety of tasks. In our implementation, we have designed the complex sequence model to support supervised classification and regression. Additionally, our technique supports "event-level supervised labels" such that each event may have an independent target label. In other words, we do not restrict each sequence from having only a single supervised target label.

Additionally, we have developed a mechanism called "nullable tagging." Nullable tagging allows model developers to include null values in their targets. The data streaming operations will not sample events with null targets to create an observation. However, other observations may contain the contents of an event with a null target. This novelty aims to provide model developers the flexibility to include events that, while offering signal to "model-triggering" events, are not "model-triggering" events themselves.

For example, in the use case of 3rd party fraud, a model developer may believe that any event occurring within 48 hours of a confirmed account takeover incident should be associated with fraudulent activity. However, the events before this fraud window are more ambiguous, such that the model developer is not confident whether the events should be labeled as fraudulent activity. Therefore, the model developer might prefer to use these ambiguous events to provide context for the confirmed fraud events while also disallowing the ambiguous events to be used to form labeled observations.

We also found this mechanism resolves an issue which we describe in the section titles The Clipping Problem

0.11.2. *Dynamically Managing Class Imbalance.*

In addition to the random observation sampling strategy previously discussed, we introduce a novel approach to dynamically downsample these observations as a function of the distributions of available class labels during a supervised classification task.

Simply put, the observation sampling rates are modified such that majority class labels are less likely to be sampled than the less populous class labels. This modification to the observation sampling technique provides for faster model training during highly imbalanced classification problems. Because the original class label distributions are known ahead of time, and the newly modified class label distributions are also known, this process will generate the approximate loss function weights for the new distribution, thus managing upsampling.

0.11.3. *Model Explainability.*

Neural network architecture often comes with a significant cost to model explainability. However, model explainability can be necessary for business problems currently being modeled with more transparent tabular machine learning techniques such as fraud detection or default prediction. We propose an additional novel technique that leverages the inner workings of our architecture to discover relative "field importance."

After pre-training a sequence encoder, we can fine-tune the model to any specific task. The fine-tuning process may take as little as an hour on a single GPU. After fine-tuning the model, we iteratively prune each input field. By "pruning," we refer to permanently masking every field value among all events with a special token: [PRUNE]. We then fine-tune the model while permanently hiding the values of the pruned fields. After pruning every field and comparing the pruned model scores, we identify the field that resulted in the smallest drop in model performance. We then run another set of pruning for every field and the running list of the least important fields. This technique

produces an ordered sequence of the least to most important fields with only $\frac{F\cdot(F+1)}{2}$ fine-tuned pruned model instances. Because we may execute this algorithm in parallel among multiple machines, its effective run time equals the time it takes to fine-tune $F$ many encoders.

We refer to this strategy as *partial field pruning*. The field pruning is "partial" in that, while the model was pre-trained with them, they are unavailable to the model during fine-tuning. By pruning the fields after pre-training the model, we can remove over 95% of the computational requirements required for iterative field pruning.

With the relative feature importance, a model developer may permanently prune the least important fields or fields that otherwise pose model governance risks should their potential risk not justify model lift.

Regarding model explainability, using complex sequential fields has another significant benefit over manually engineered tabular features: they are easier to monitor and intuitively comprehend. Some tabular machine learning techniques do provide better explainability in theory. However, in practice, it is very challenging to understand how hundreds of tabular features may interact with one another. Additionally, the logic used to construct tabular features from the ledger may lead to unexpected behavior.

### 0.12. **Automated Model Training Pipeline.**

Having defined the necessary operations to construct, pre-train, and fine-tune the model architecture, we developed a model training pipeline to perform all these stages automatically. This automation is possible by engineering a framework that executes the above strategy as a function of the data and hyperparameters.

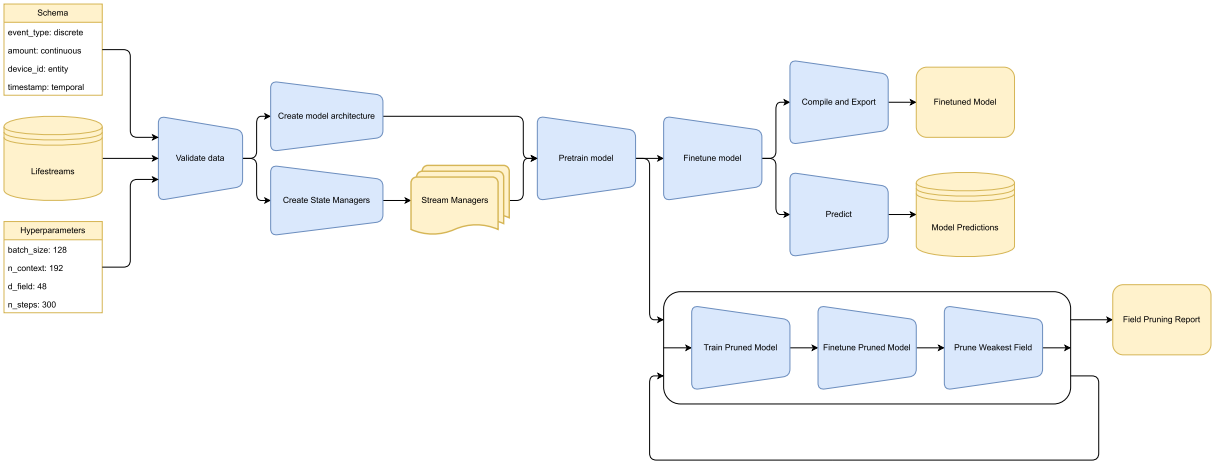


FIGURE 10. The model training pipeline in its entirety

#### 0.12.1. *System State Managers.*

Unlike many other data science pipelines, however, this requires a set of stateful operations to execute as intended. As such, the pipeline will instantiate a collection of system state managers that handle the necessary stateful processes. These system state managers include, but are not limited, to the following:

- Schema Manager: Contains the available fields and their requested field types.
- Balance Manager: In the case of a classification problem, discover the available class labels and their respective label counts, calculate effective sampling rates per class label for downsampling, and calculate the effective loss function weights for upsampling.
- Split Manager: Contains requested strata split sizes and splits sequences among the train, validation, and test strata.
- Sample Manager: Calculates how to effectively sample observations from the available sequences during pre-training and fine-tuning.
- Centroid Manager: Trains and serializes the `TDigest` object for each continuous field and reconstructs the `TDigest` objects during training and inference.
- Level Manager: Discovers each unique available level (category) for all discrete fields to construct the appropriate embeddings table during a model's initiation and "tokenizes" the levels during training and inference.

System state managers are limited to stateful operations. In other words, all that one requires to instantiate a model is the collection of state managers and the hyperparameters. The state managers are, in most cases, calculated as a function of the available data and hyperparameters.

0.12.2. *Pipeline Workflow.*

Six steps are required to complete the model training pipeline:

1. **Data Validation**: Parse the inputted ledger and confirm that the requested field types are available and valid.
2. **Manager Instantiation**: Create all the system state managers required to complete the requested pipeline execution.
3. **Data Preprocessing**: Convert the inputted ledger to a lifestream for optimal streaming operations.
4. **Model Pre-Training**: Execute the self-supervised learning task to pre-train the model.
5. **Model Fine-Tuning**: Execute the supervised learning task to fine-tune the model.
6. **Model Inference**: Predict supervised target labels for ad-hoc analyses.

0.12.3. *Pipeline Hyperparameters.* The pipeline executes as a function of the inputted ledger and the following set of hyperparameters:

1. `batch_size`: How many observations are included per training step.
2. `n_context`: how many events are included per observation.
3. `d_field`: The hidden size of each field embedding (48 to 128).
4. `n_heads_field_encoder`: Number of heads in field encoder (4 to 16).
5. `n_layers_field_encoder`: Number of layers in field encoder (1 to 2).
6. `n_heads_event_encoder`: Number of heads in event encoder (4 to 16).
7. `n_layers_event_encoder`: Number of layers in event encoder (4 to 16).
8. `dropout`: The probability of disregarding certain nodes in a layer at random during training (5% to 25%)
9. `n_bands`: Precision of Fourier feature encoders (8 to 12)
10. `head_shape_log_base`: How quickly to converge sequence representation to final head size (3 to 5)
11. `n_quantiles`: Number of pretraining quantiles for continuous fields (64 to 128)
12. `n_pretrain_steps`: Number of steps to take per epoch during pretraining (512)
13. `n_finetune_steps`: Number of steps to take per epoch during finetuning (64)
14. `quantile_smoothing`: Smoothing kernel of continuous fields' ordinal classification pre-training task (1.0)
15. `p_mask_event`: Probability of masking any event during pretraining (7.5%)
16. `p_mask_field`: Probability of masking any field value during pretraining (7.5%)
17. `n_epochs_frozen`: Number of epochs to freeze sequence encoder while finetuning
18. `interpolation_rate`: Interpolation rate of imbalanced classification labels (10%)
19. `learning_rate`: Learning Rate during Pretraining
20. `learning_rate_dampener`: Learning Rate Modifier during Finetuning
21. `patience`: Number of Epochs Patience for Early Stopping
22. `swa_lr`: Learning rate of averaging multiple points along the trajectory of loss function
23. `gradient_clip_val`: Maximum allowable gradient values (prevents exploding gradients)
24. `max_pretrain_epochs`: Maximum number of epochs for pretraining
25. `max_finetune_epochs`: Maximum number of epochs for finetuning

0.12.4. *Peripheral Pipeline Steps.* Because the orchestrated pipeline can operate automatically, we may incorporate a set of helpful utility components to execute within the pipeline:

1. Run a series of assertions on the ledger to test the data quality.
2. Generate visualizations for each of the input fields within the dataset.
3. Compile and quantize the model for more efficient model inference.
4. Compare model scores between the trained torch model and the compiled model to validate model compilation.
5. Attempt model rollout to managed production environment.
6. Iteratively prune fields from the pre-trained models to cheaply discover relative field importance (partial field pruning).
7. Generate a report on model performance using out-of-sample data.
8. Generate a whitepaper on model hyperparameters, including information from the steps above for automated model version documentation.

0.13. **Further Research.**

0.13.1. *Media Fields.* Standard field types (discrete, continuous, temporal, geospatial, entity field types) solve many common business problems. However, we may utilize more complex field types to represent raw text, images, audio, and video. These are more complicated and rely upon pre-trained models specific to each modality.

0.13.1.1. *Textual Field Embeddings.* We can also *potentially* use raw string text. This one is a little bit tricky, however. This approach resembles running BERT inside the modular field embedding system before passing the textual representation into two more transformer encoders. However, utilizing BERT to embed the textual fields will require enormous amounts of memory for model training. The computational requirements will be very costly because this BERT model's effective batch size equals `N*L*F`. In other words, you must simultaneously encode every textual field of every event and observation.

For each event in each observation, we want to take the textual fields, tokenize their content, and then run them through BERT, collecting the text representation from the class token. There are two ways of doing this, neither of which are especially pleasant:

1. Use a single instance of a pre-trained BERT model and fine-tune it for all textual fields.
2. Use multiple instances of a smaller pre-trained BERT model and fine-tune them for each textual field.

The BERT model would likely need to be limited to work with minimal token context size to prevent running out of memory. However, this small context size will limit the quality of the embeddings it produces.

Additionally, pre-training textual field embedding modules is challenging. Running masked-language modeling *inside* our custom self-supervised learning task would be exceedingly difficult. If the BERT model is already pre-trained, it might be best to leave it frozen (thus slightly alleviating the memory requirements).

```python
@tensorclass
class TextualField(TensorField):
    document: Int[torch.Tensor, "N L S"]
```

0.13.1.2. *Images, Video, and Audio Field Embeddings.* In addition to working with just embedded textual fields, one could *theoretically* include images, audio, and video data. In other words, any event could contain a field value of a file path. We may then fetch, read, decode, and preprocess the necessary files as a streaming operation.

For example, suppose you are trying to create representations of all Twitter users. Each Tweet may include a photo or video, so each Tweet event may optionally include a photo or video. In other words, each `TweetEvent` might consist of the following fields: `timestamp`, `content`, `device`, `location`, `photo`, `video`, `is_deleted`. The photo (or video) is then embedded, as are all other fields. Each Twitter user may also post a `ProfilePictureUpdateEvent` including the following fields: `timestamp`, `device`, `location`, and `video`.

Streaming the operations to read, decode, and collate images will complicate the data streaming pipeline. However, there are some ways to optimize it. For example, we could store each sequence as a small file and all its associated media in separate files. We then compressed each sequence's files into a single `TAR` file. We then stream through the collection of compressed sequences and sampled events and fields as before. If any media fields include a file path name, the necessary decompressed files will be read, decoded, and collated into an observation.

```python
@tensorclass
class ImageField(TensorField):
    image: Float[torch.Tensor, "N L X Y"]

@tensorclass
class VideoField(TensorField):
    video: Float[torch.Tensor, "N L V X Y"]
```

Implementing such media fields is complicated but illustrates the versatility of approaching complex sequential data with modular field embeddings. Asynchronous streaming operations and distributed model training environments are necessary to scale these more complex media fields.

0.13.2. *Including Tabular Data and Multiple Sequences of Varying Event Rates.* We have not yet explored integrating static tabular data with sequential data, although Visa Research has explored this approach in their FATA-Trans paper. [9] We are also curious about working with multiple contexts of event types per sequence where the event types among each context have similar event rates. For example, one context of 48 monthly billing

statements over the last four years, a second context for the last 256 customer financial transactions, and a final context of the previous 2048 online click stream events. This approach would allow the model to learn from multiple sources without low-frequency events being diluted from a single context by high-frequency events.

The most significant obstacle to utilizing contexts of varying event rates is the systematic synchronization of these events as a streaming operation. For example, which monthly statements should be available to the context given which online click stream events? Each click stream event would require an index to its corresponding monthly statement.

Utilizing multiple sequences and tabular data provides another solution to the "flushing problem." Capturing signals from contexts of varying event rates will better utilize high-frequency events, such as clickstream events, without diluting low-frequency events, such as monthly credit bureau snapshots.

0.13.3. *Arbitrary Dimensionality.* In the examples above, we use a specific dimensionality: Each sequence contains events, and each event includes fields. In other words: `Customer > Transaction > Field`. This dimensionality fits the vast majority of use cases that come to mind. However, we realize there are counter-examples to this design.

Walmart may want to model their transactions differently. For example, they might want to include the concept of "shopping trips", in which each customer might make multiple shopping trips, and during each shopping trip, they might purchase numerous items, and each item may have multiple fields. So, instead of `Customer > Transaction > Field`, we might instead model the sequence as `Customer > Trip > Item > Field`.

While the implementation of the model will change completely, the concept is not all that different. Complex sequential modeling is still applicable. However, we do not currently have the development of this 3D implementation in scope.

0.13.4. *Sub-quadratic Alternatives to Self-Attention.* A unique property of sequential data is that events asynchronously arrive in separate streaming messages. Consequently, there is a potential optimization trick for real-time inference. Using emerging alternatives to transformers, such as RWKV or Mamba, one can save enormous amounts of the compute requirements for inference. The memory complexity with respect to the context size decreases from quadratic to constant.

- Store sequence state representations in an in-memory cache like Redis
- Whenever a new event arrives:
    1. Let each field attend to one another within the new event
    2. Concatenate the attended fields to create a contextualized event representation
    3. Retrieve the current contextualized event representation hidden state
    4. Update the hidden state with the new event representation
    5. Pass the new hidden state to a decision head to create and return a new model score

However, that approach will require an effectively infinite context size. An infinite context size may challenge the current means of embedding the `EntityField` field types through random integers. In other words, we will likely need to use larger embedding tables and accept the possibility of hash collisions.

0.13.5. *Sparse Attention.*

The modular field embedding system currently has a static definition of special tokens, such as `[PAD]` and `[UNK]`. These tokens can facilitate sparse attention for the field and event encoders to improve memory efficiency. Within the field encoder, we may create an attention mask to mask out fields equal to `[PAD]` or `[UNK]`. Within the event encoder, we may create an attention mask to mask out events in which all fields are equal to `[PAD]`. Given high enough sparsity, this may result in significantly better memory requirements.

## 0.14. **Appendix.**

0.14.1. *Sequence Entropy.* A unique characteristic of such multivariate sequences is that they are not nearly as structured as human language. Multivariate sequences can be noisy, whereas grammar rules bind human language. Human language has evolved to be structured, redundant, and fault-tolerant. For example, An adjective must precede another adjective or the noun it describes. If one shuffles all of the words in a sentence, the resulting collection of words would likely be nonsensical. However, if one were to mask out a random word, you may still have a decent guess as to the general message of the sentence.

On the contrary, complex sequences of transactional data tend to have relatively less structure. Shuffling every event in a sequence will result in a new sequence almost as "plausible" as before. However, reconstructing after having masked out a random event would be much more challenging. For example, if you had to go grocery shopping

and also top your tank with gas, what portion of the time do you go to the grocery store *before* you go to the gas station? There is no correct answer. The order is almost entirely arbitrary.

Additionally, pre-training a large language model on every human language is challenging due to large vocabulary sizes, disproportionate word usage, quotes, mannerisms, plagiarism, and writing styles. Pre-training complex sequence models with transactional data is trivial in comparison because the signals are not as complex, thus requiring significantly fewer resources.

0.14.2. *The Clipping Problem.* We discovered an unexpected complication while training sequential models on slices of complex sequential data. It is intuitive in hindsight but was initially challenging to identify.

It is commonplace to subset slices of time series based on some time window (IE, all events between January 2020 and January 2023). However, this subsetting strategy can lead to very unexpected behavior due to an issue we call the "clipping problem."

Simply put, the model may sometimes not have enough information to understand if the first event in a ledger's sequence is the first event or if it just looks like the first event in the sequence because we had manually filtered out the previous events. In other words, there is ambiguity in what had happened before the start of the available events in a sequence.

This problem has at least two solutions:

1. Include information regarding the start of the sequence, such as the creation of a field that provides the time since the sequence started (`account_tenure`)
2. Instead of creating a fixed time window (Jan 2020 to Jan 2023) at an event level, filter the ledger to only accounts that opened between some two dates.

These solutions resolved the ambiguity between a sequence with an artificial cutoff and a sequence with an available initial event.

0.14.3. *The Flushing Problem.* A unique adversarial vulnerability arises when modeling sequential data with a fixed context size depending on the types of events allowed to fill the context window. Theoretically, an ill-intentioned individual could "flush" the context by spamming innocuous events. In other words, an attacker may "log in" and "log out" of their account several thousand times to attempt to manipulate the historical context.

A more concrete example: A savvy hacker, having recently hacked into their victim's account at Big Online Store, could choose to reset the user's password, email, phone number, and mailing address before attempting to purchase gift card codes. Such risky actions, which may otherwise result in a security alert, may go unnoticed if the fraudster intentionally spams many login events between each nefarious action, such that the model doesn't see more than one such nefarious action at any time. In other words, if you perform a thousand low-risk events between every high-risk event, the model will not see just how risky your recent actions are.

There are multiple strategies to mitigate the risk of such an attack:

1. Increase the context size.
2. Set up policies to flag high event rates where needed.
3. Do not include trivial events (such as log-in or log-out events) in the context, but instead represent them whenever possible as entity fields (such as a "login session")

## References

1. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, (2019)
2. Padhi, I., Schiff, Y., Melnyk, I., Rigotti, M., Mroueh, Y., Dognin, P., Ross, J., Nair, R., Altman, E.: Tabular Transformers for Modeling Multivariate Time Series, (2021)
3. Luetto, S., Garuti, F., Sangineto, E., Forni, L., Cucchiara, R.: One Transformer for All Time Series: Representing and Training with Time-Dependent Heterogeneous Tabular Data, (2023)
4. Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., Ng, R.: NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis, (2020)
5. Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J. T., Ng, R.: Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains, (2020)
6. Dunning, T., Ertl, O.: Computing Extremely Accurate Quantiles Using t-Digests, (2019)
7. Protivinsky, T.: PyTDigest, (2023)
8. Coleman, B., Kang, W.-C., Fahrbach, M., Wang, R., Hong, L., Chi, E. H., Cheng, D. Z.: Unified Embedding: Battle-Tested Feature Representations for Web-Scale ML Systems, (2023)
9. Zhang, D., Wang, L., Dai, X., Jain, S., Wang, J., Fan, Y., Yeh, C.-C. M., Zheng, Y., Zhuang, Z., Zhang, W.: FATA-Trans: FieldRequest And Time-Aware Transformer for Sequential Tabular Data. In: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management. ACM (2023)

Research Science, Windmark Labs, Arlington, VA 22202
*Email address:* granthamtaylor@icloud.com