NATIVELY MODELING COMPLEX SEQUENTIAL DATA WITH MODULAR FIELD EMBEDDINGS

GRANTHAM TAYLOR

Abstract. Large organizations struggle to model large complex sequential data collected via Event Sourcing. Without the proper solutions to facilitate the use of modern AI techniques, businesses still tend to use traditional machine learning techniques, resulting in significant time and monetary costs. Previous research has developed techniques that can 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 allows for the representation of multiple nullable sequences of any combination of discrete, numerical, and temporal fields as well as anonymous entity representations without any feature engineering. Our findings promise a novel solution that may automate the model development process, reduce model deployment costs, and improve model performance.

0.1. **Problem Statement.**

From fraud detection to predicting each customer's risk of charging off, many organizations choose to model their business problems with tabular machine learning. However, modeling data with tabular machine learning can be surprisingly challenging. Some machine learning projects can take years to develop and deploy to production. The long development timelines for tabular machine models can be attributed to the following four reasons:

1. Preparing production data for tabular machine learning requires manual feature engineering that dilutes the signal 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 complex tabular features within mere milliseconds.
4. Developing models for new business problems requires entirely new features to extract signals from the same data sources, thus limiting an organization's ability to consolidate technological and personnel resources among otherwise identical implementations.

All of these problems originate from one key insight: Many common machine learning problems being solved with tabular machine techniques are not of a truly tabular form. Organizations accrue vast amounts of technical debt to restructure their business problems so that they may be solved with tabular machine learning.

By unraveling this misconception and developing a tailored solution for business data in its native form, we can address these four deficiencies to 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. As a consequence, data scientists must spend enormous amounts of resources to restructure their business data to meet these assumptions to even use tabular machine learning. In other words, data scientists are transforming their data to make it tabular instead of modeling the data in its native form.

Suppose you work at a financial institution and you are tasked with training a model to determine if a customer has misrepresented their identity. This is a standard identity verification business problem. Your model is a small piece of the identity verification pipeline, and your task is to flag 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 is the complete history of every event posted by every customer. This history may fit in a table, but it is not able to be modeled with tabular machine learning models (each observation is not independent of one another).

This data structure is something more universal; a "ledger" of events posted by Jane Doe. Each event arrives asynchronously and is immutable such that once a customer does something, it permanently stays on the ledger. Each event can contain any number of complex fields, and each field can be of any data type (categorical, raw text, graph-like entity identifiers, numeric, boolean, timestamp, geospatial coordinates, and even file paths to images or audio clips).

Additionally, each event may have an associated timestamp for when it was observed. This data architecture pattern is called "Event Sourcing". It is extremely powerful for applying business logic and maintaining a record of every customer's history for the sake of reproducibility. It defines the complete history of everything ever done by every customer. Customers provide enormous amounts of data, almost all of which comes in this format.

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

Modeling the ledger is exceedingly challenging, however. It is extremely common to simplify this complex data schema into a tabular format to utilize traditional machine learning techniques, such Gradient Boosting Machines (GBMs). The process of extracting tabular features from the ledger is called "tabular feature engineering". This process creates a single observation with many hundreds to thousands of tabular features that each represent the composite of previous events. If done correctly, a tabular machine learning model can model the target variable as a function of the tabular features.

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 the customer has verified their most recent phone number
- `email_confirmed`: Whether the customer has verified their most recent email address
- `verified_income`: Whether the customer has verified their most recent reported income
- `account_tenure`: The number of hours since the 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

A GBM will make quick work of such tabular data, likely providing decent initial model results if you have enough labeled training observations and optimize your hyperparameters.

The ledger is much more complicated to model than this tabular view, but the ledger 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 duration of 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 is incapable of modeling all of these potential signals without adding new features.

Of course, you *can* create more tabular features that can collectively approach the signal available within the ledger. For example, you could create more tabular features to, say, count the number of times each customer has changed their email over varying time windows (over the last day, week, and month). You would also need to count the number of times the email was verified among varying time windows (over the last day, week, and month) as well as 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. You would need to create hundreds, if not thousands of tabular features to account for many different possible intersecting events, over many varying time windows, with multiple possible aggregation functions. Meanwhile, the ledger already contains this information in a succinct form.

When you model sequential data with tabular models, you will require such aggregative window functions to capture the complex signal. 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 you should prioritize. Even beyond the costly computational and domain expertise requirements, however, calculating hundreds of arbitrarily unique window functions in real time requires entire teams of dedicated on-call engineers.

Lastly, GBMs, being *greedy* learners, are especially prone to overfitting with many tabular features. You can not simply throw thousands of the infinitely many potential features into the model or its performance will start to degrade. This adds even more constraints to the optimization process.

All of these factors result 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 inference requirements. Because of all of these costs of modeling sequential data with tabular models, many organizations are simply unable to address many problems that would otherwise provide tremendous value-add.

By modeling the data exactly how it already exists in its production source table, as opposed to using a "view" of it, we may be able to skip all of the feature engineering. Feature engineering is the sole reason that 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.

However, by using the raw data to model our problem allows us to instantly eliminate all of these significant costs from the modeling process.

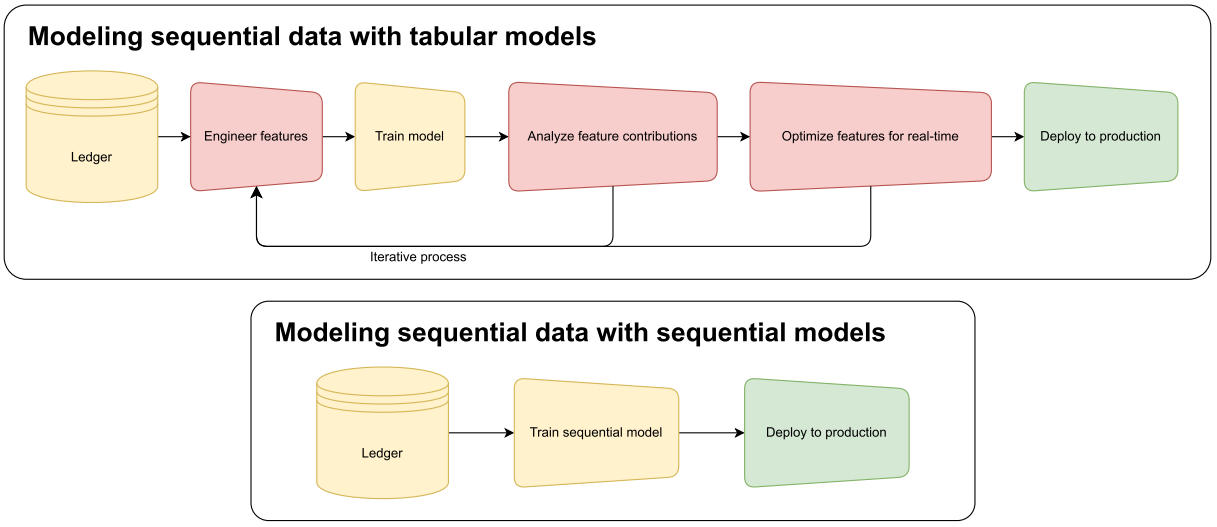

FIGURE 1. Cutting out tabular feature engineering dramatically increases speed-to-market

We should change our model to fit our data instead of changing our data to fit our model. This 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 were manually defining convolution kernels. This is, in essence, how we are currently modeling business problems: with hardcoded tabular features.

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 single transaction posted by a credit card holder of Chase, what is their survival curve (either 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 user of Venmo, what is the probability that the current transaction is an attempt to defraud another user?
7. Given the previous tweets posted by a user of Twitter, what is the probability that this user is a bot?

There are hundreds of other such business problems associated 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 any arbitrarily unique means of extracting signal.

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 each individual may be defined by the composite of their actions. This customer representation may then be used to solve any arbitrary machine learning problem with a single generalized modeling approach which does not require any manual feature engineering, thus accelerating model development timelines, increasing model performance, and decreasing model deployment costs.

0.3. **Terminology.**

It will be helpful to define some terminology before going forward. We will be discussing the architecture of an extensible ***complex sequence model***. The model architecture is extensible in that it can be dynamically instantiated to adapt to the schema of the data with which it will be trained. The complex sequence model can train on ***observations***, each of which is created from a slice of a ***sequence***. A sequence is made of multiple ***events***, in which each event may have any number of ***fields***, which may have multiple possible data types. We may refer to this data as being "sequential" or "transactional". It is, however, a sequence of multivariate events with nullable, heterogeneous fields.

The population of sequences exists in some data table, in which each event is a record. We refer to this source of raw data as a ***ledger***. We must stream through the ledger, identify all of the unique sequence IDs (i.e., customer IDs), and find all of the events that exist within each sequence. Each event includes multiple 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 distinctly represent and learn as distinct and information-bearing values.

We then stream over each sequence, looking at every event available within each sequence. We sample random events from the sequences. Each event has an equal probability of being sampled. After sampling each event, we then take that event, as well as the $L-1$ preceding events to calculate the ***context*** for the 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, 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.

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

We then create many more 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. Observations are used to pre-train and fine-tune the sequence model.

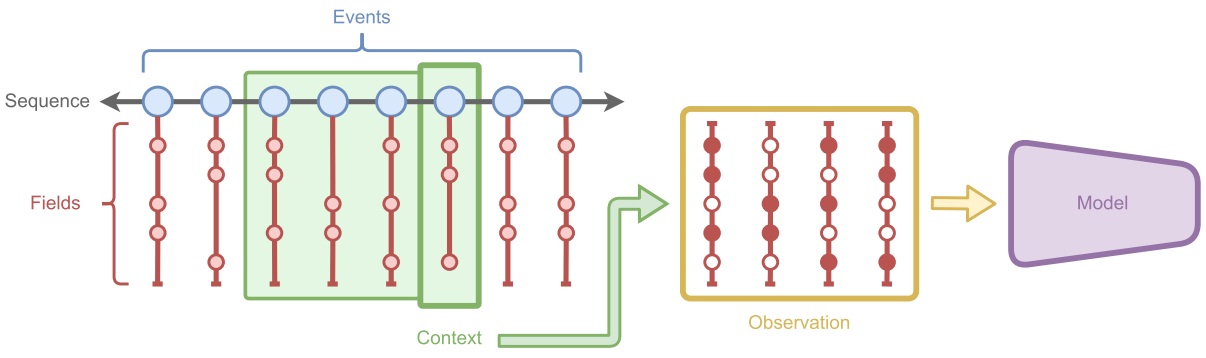

FIGURE 2. An observation being sampled and collated from a sequence

The section *Data Preprocessing for Efficient Streaming Operations* describes how this process may be optimized for streaming arbitrarily large data from disk.

The model architecture, which we discuss in detail in the next section, utilizes transformer-encoder blocks like BERT [1]. The sequential model creates a representation of the events within the context to answer questions about the sequence at a specific point in time. This is in contrast to models like GPT, which utilize transformer-decoder

blocks to predict subsequent tokens. Additionally, the data structure is similar to that 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 being events with a single discrete field, whereas multivariate sequential data may include more than one field. In other words, this model architecture ($\mathbb{R}^{N\times L\times F\times C}$) requires one more data dimension than BERT ($\mathbb{R}^{N\times L\times C}$).
2. BERT samples observations at a sequence level, but we need to sample observations from sequences at an event level. For example, a customer with 10,000 events should result in approximately 10 times more observations than a customer with only 1,000 events because each observation is defined by a randomly sampled event. This enables the model to support fine-tuning with targets uniquely defined for each event.

0.4. **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

0.5. **Background.**

0.5.1. *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 such handcrafted tabular features for a set of supervised problems with sequential data. [2]

In essence, they are framing 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 $\mathbb{R}^{N\times L\times F}$ when collated and batched together to form a training observation. In their implementation, every single field is limited to being exclusively discrete, such that each value is an integer that needs to be embedded via a standard embedding table for each field.

BERT was embedding 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***) lets every field embedding ($\mathbb{R}^{N\times L\times F\times C}$) *attend* to every field embedding within an event. Those contextualized field embeddings are then concatenated 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. The event encoder is constructed similarly to BERT, such that there is an appended class token ($\mathbb{R}^{N\times FC}$) that may be used to represent the sequence as a whole, but one can also use the attended 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 a portion of the events and fields are masked, and the event representations outputted from the event encoder need to be able to reconstruct the masked events and fields. For fine-tuning, the appended class token is passed through an MLP to solve a wide array of supervised learning tasks.

For those unfamiliar with hierarchical transformer architecture, a great analogy comes from the domain of computer vision. If one is building a representation of an image, ideally each pixel should interact with every other pixel. 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 every pixel of every frame. Instead, it would be more appropriate 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 of these frame representations to interact with one another. Similarly, TabBERT decomposes a very large and complex problem into two smaller problems: creating representations of each event and then using those event representations to create a representation of the sequence as a whole.

The hierarchical transformer architecture dramatically reduces the memory complexity of the model. Transformers are notorious for their quadratic memory complexity (both during training and inference) with respect to the context size. By utilizing two transformers to encode the fields and then events separately, we do not require the extensive memory requirements that trouble language models. In effect, we have reduced the memory complexity from a worst-case of $O\left((LF)^2\right)$ into a slightly less scary $O(L^2F + LF^2)$. Within the typical ranges of L (128 to 512) and F (6 to 15), this reduces memory requirements by a bit over an order of magnitude.

TabBERT, while elegant, did come with some major drawbacks. The largest of which is that TabBERT requires that every field be coerced into a categorical data type ($\mathbb{W}^{N\times L\times F}$). For example, dollar amounts cannot be fully represented, but you can 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 some continuous dollar amounts might be discretized to categorical values

Discretizing every field comes with 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 is free to chase the residuals around these exact cutoffs in the continuous distributions. For some tasks, arbitrary precision of continuous fields could be extremely important such that discretizing the continuous fields irrevocably muting crucial signals.

Additionally, the self-supervised pre-training task suffers from the masked categorical targets if you are working with continuous values that were discretized. Neither the model nor the loss function are aware of 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"`.

0.5.2. *UniTTab.* In 2023, a small Italian consulting firm Prometeia published *One Transformer for All Time Series: Representing and Training with Time-Dependent Heterogeneous Tabular Data*, in which they introduced the UniTTab architecture. UniTTab made use of some very interesting ideas to represent continuous field values. Their novelties included the use of Fourier feature encodings to "embed" floating point field values without having to bin the values. 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 in which they describe how this technique works. [5]

UniTTab also adapted TabBERT's custom self-supervised learning task to work with continuous fields. Instead of trying to reconstruct the raw scalar values of each field via regression, UniTTab simplified the problem to become 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 its source code. Additionally, they were quite vague in the implementation of 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.

0.5.3. *FATA-Trans.* In November of 2023, the Visa Research Lab published *FATA-Trans: FieldRequest And Time-Aware Transformer for Sequential Tabular Data*. FATA-Trans was built on top of TabBERT, and did not provide any support for continuous fields. They included two major contributions: the ability to encode timestamps, and the ability to use tabular features alongside sequential data. Theoretically, one could already input tabular features by making the values constant among every event in the sequence, but as Visa stated, this approach is computationally inefficient. They concluded that their architecture had marginal improvements over TabBERT. [6]

0.6. **Consolidating Complex Sequence Modeling with Modular Fields.**

TabBERT, UniTTab, and FATA-Trans have all contributed a great deal to 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. FATA-Trans introduced encoding timestamps and tabular features. [3, 2, 6]

In this next section, we will be discussing 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 three aforementioned papers (supporting discrete, continuous, and temporal 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 categorical input with far too many levels to be learned with standard learned embeddings (i.e., a device ID, a login session, a phone number, or a merchant name).
3. Introduce the idea of including "embedded media", such as text, images, video, and audio by leveraging foundation models to embed field values for each universal modality as required.

to support the wide variety of these fields, we have designed a custom implementation architecture that can adapt its components to support any combination of field requirements. The implementation manages a set of unique field embedders for each supported field type with extensibility in mind, such that embedders for new field types may be independently managed as a "plugin" system. We refer to the system that manages the instances of the field embedders the "Modular Field Embedding System" (MFES).

0.6.1. *Modular Field Embedding System (MFES).* 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, in each tuple is a row. Alternatively, you could instead view a data frame as a dictionary of lists, in which each list is a column. In this analogy, UniTTab elected to focus on the row-major structure of lists of tuples to avoid having to represent null values. We take the second alternative, in which each column is represented by a list within a dictionary.

Instead of creating "row types" in the style of UniTTab, we define "field types". Each field type is completely modular in that its contents are defined within a `tensorclass` (a `dataclass` of tensors). Each field type requires several components:

1. A type-specific field embedding module (to create the field embeddings for a specific field type)
2. A type-specific field masking function (to mask the field values during pre-training)
3. A (optional) type-specific field decoder (to turn a contextualized event representation back into field-level predictions for the self-supervised learning task). If one isn't specified, the field types will not be used in the self-supervised learning task.

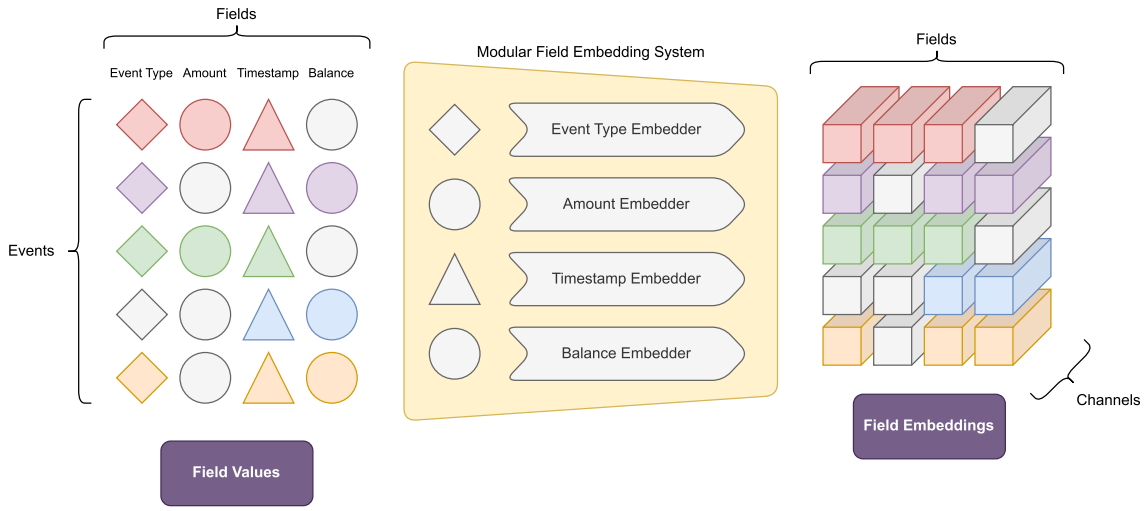

FIGURE 3. The MFES can embed fields of arbitrary types

It is extremely important to note that this approach is only valid with the assumption that each field's non-valued states (being masked, padded, or otherwise null) can be uniquely represented without overlapping with the valued states. This is the cost of viewing the data from a column-major perspective.

To embed each field, we need to ensure that each field can be fully represented regardless of its state. There are four unique possibilities for any given field:

1. There is a field value present (the field is "valued").
2. There is no field value present because the field's event was padded (the field is "padded")
3. There is no field value 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")

0.6.2. *Discrete Field Embeddings.* As a simple example, suppose you intend to initiate a simple model architecture that needs to support four discrete fields per event: `is_foreign`, `is_online`, `merchant_category`, and `transaction_type`.

We use torch's `TensorDict` and `tensorclass` implementations to bundle the heterogeneous fields. For those unfamiliar with these constructs, they are almost exactly how they sound. A `TensorDict` is a torch 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 container for each field type.

Below is an example of the input to a complex sequence model with four discrete fields. The input for each discrete field is contained within a `tensorclass` called `DiscreteField`.

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

# 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)),
})
```

This may seem unnecessarily complex at first, but it is quite necessary to support multiple fields of varying field types. Additionally, methods may be attached to each `tensorclass` to support field-type specific operations, such as masking, preprocessing, and defining the targets for the self-supervised learning task.

The modular field embedding system for this field schema looks like so:

```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 $\mathbb{R}^{N\times L\times C}$. After embedding each discrete field, the `ModularFieldEmbeddingSystem` will then concatenate the embeddings into a single tensor of dimensionality $\mathbb{R}^{N\times L\times F\times C}$, which is then passed into the field encoder.

Representing the non-valued states (padded, null, or masked) is trivial in this case. There are additional tokens available within the `DiscreteFieldEmbedder`'s embedding table. This is almost exactly like language models with three exceptions:

1. In this case each unique field embedding module has an embedding table for these non-valued states. The special token embeddings are not shared among the modules.
2. There is no `[CLS]` token present in the field embedding module. The equivalent of a `[CLS]` token (a `[CLS]` event) is appended to the event representations right before the event encoder module. The field encoder would not add any value to a `[CLS]` token.
3. Positional embeddings are required for both the field encoder and the event encoder. The field encoder's positional embeddings will learn to uniquely represent each field (I.E., what *is* a POS transaction dollar amount, or what is an MCC code). The event encoder's positional embeddings will learn to represent the order of the events within the observation. Both positional embeddings are learned tensors and are of dimensionality $\mathbb{R}^{F\times C}$ and $\mathbb{R}^{L\times FC}$, respectively.

0.6.3. *Continuous Field Embeddings.*

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

```rust
enum ContinuousFieldValue {
    Valued(f32),
    Unknown,
    Padded,
    Masked,
}
```

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

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

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

This guarantees that $\overline{x} \in [0, 1)$. Wherever a field is non-valued (either null, padded, or masked), we impute `value` to be equal to `0.0`. By using the CDF of continuous field values, as opposed to the raw scalar values, we do not require any data standardization/normalization operations to represent values of arbitrary distributions. The CDF of the `value` 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 used as the source of truth for every field. It is important to note that the `[VAL]` token is mapped to 0, which allows for a beautiful mathematical optimization trick described later in this section.

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

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

to solidify this concept, let's walk through some examples to show how every possible state of our `ContinuousFieldValue` enum can be mapped to an instance of `ContinuousField`:

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

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

This mapping provides a guarantee that every possible input has a unique tensor representation. Similar to UniTTab and NeRF, we then embed the `value` 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 `value` with a sequence of bands, `B`, defined as $2^{[-8..3]}$, and then multiplying that again with `pi`. We then compute both `sin(value * B * pi)` and `cos(value * B * pi)`, concatenating these two sequences together to create `gamma`. This tensor is then passed through a simple MLP to represent the tensor `value` to the neural network.

However, there is a small problem. Because we have imputed the non-valued fields to `0.0`, the model is unable to differentiate a value that is masked, padded, null, or is actually a value but just happens to be equal to `0.0`. The solution is surprisingly simple. We subtract the `lookup` tensor from the `value` tensor and instead embed the resulting tensor with FourierEncoding. The model will learn that values exactly equal to `-1`, `-2`, and `-3` 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 continuous inputs can be uniquely mapped to real values

This is only possible for the following three reasons:

1. A field's non-valued states are all effectively mutually exclusive. A field value can not be both padded and null at the same time. While a field can be both masked and null or masked and padded during pre-training, the model should only ever see that it is masked.
2. The `[VAL]` token is represent by 0.
3. The `values` tensor has a limited range of $[0, 1)$.

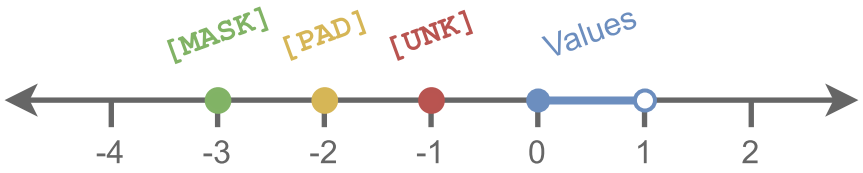

FIGURE 4. How the discrete parts of continuous fields may be uniquely represented alongside the values

This method also comes with a very beautiful runtime assertion to rigorously validate the two tensors `value` and `lookup` for each continuous field: Given that every non-valued input should be imputed to `0.0`, and that the `[VAL]` token itself is equal to `0`, we can *guarantee* that the element-wise product of `lookup` and `value` will always equal zero. 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"
```

There is one last caveat to this approach regarding the precision of the floating point `value` 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, we need to classify which quantile bin a masked value should fall into. If we chose a total of 10 quantile bins then we would need to determine to which decile each of the masked values belong. The choice of quantile bins is a pre-training hyperparameter itself, q, so it is much better to calculate the CDF of the original values for the pre-training task, and then when we choose a value for q we can simply calculate the appropriate bucket index by calculating $\lfloor q * \text{CDF}(x) * 0.\overline{9} \rfloor$. That means that we can use the CDFs for the inputs of the model, and then modify the CDFs for the targets too. The scaling by $0.\overline{9}$ guarantees that the flooring operation will always round down.

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

[1]

This novel strategy provides the ability to embed a nullable continuous field of an arbitrarily complex numeric distribution. Now, let's look at how we may represent multiple, heterogeneous fields within a `TensorDict` input for the MFES.

```python
@tensorclass
class ContinuousField:
    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),
        value=Tensor(torch.Size([2, 8]), torch.float),
    ),
    balance: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        value=Tensor(torch.Size([2, 8]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

The `ModulerFieldEmbeddingSystem` can utilize instances of `ContinuousFieldEmbedder` for each of the continuous fields, each of which is 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 of the other implementations with which we experimented 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 $F = 3$.

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

Timestamps are parsed 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 are used to capture the signal regarding event seasonality. Unlike continuous fields, each of these three inputs fits a known 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 is transformed 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.

Python

```python
@tensorclass
class TemporalField:
    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),
        value=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, this task is smoothened via neighborhood loss smoothing.

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 event and the previous event. This could prevent the model from overfitting on irregular activity that is limited to a small window of time which is not expected to generalize to the population of training data.

0.6.5. *Geospatial Field Embeddings.* Geospatial data also seems rather tricky. However, we can once again use a technique similar to the continuous field embedding mechanism to accomplish this. Unlike the continuous or temporal fields, geospatial coordinates are defined by two numbers: longitude and latitude. Both longitude and latitude require their own `TDigest` model.

```python
@tensorclass
class GeospatialField:
    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),
        value=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 course of action. If one's data is too small it may lead to overfitting. If one is trying to create representations of ride-share drivers based on their previous trips, they might include both the pickup and dropout geospatial coordinates for each trip. Alternatively, one could simply include the distance and zip code of the trip.

Additionally, the lines of longitude overflow at the prime meridian. This means that a coordinate with a longitude value of 179 is as close to the prime meridian as a coordinate value of -179. This is fundamentally different from how our numeral system works, and so it requires a slightly different strategy to decode during the self-supervised learning task.

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

0.6.6. *Entity Field Embeddings.* Entities are extremely common in production datasets. By an entity, we refer to one discrete item in an infinitely large set of items. The set of items, while similar to categorical data, is so large or otherwise ephemeral that each unique level in the category cannot possibly be explained by a unique embedding.

For example, whenever a customer logs into their account, a server might create an ephemeral login session token that is unique to that login session. This login session token is stored as a string alongside all of the events that happen during the same login session. The complex sequence model could then use this login session ID to see which events happened during each login 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. This is different than "merchant name", which may include a rigorously cleaned string like `"McDonald's"`. A "merchant" may also include the address or store number, allowing for an even greater level of detail.

Another common 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 are defining a customer-level model (such that each sequence is defined by all of the events posted by a customer), but customers may have more than one account, you could represent each of the accounts available to a customer with an entity. The opposite is also true; if you are designing an account-level model (such that each sequence is defined by all of the events posted to an account), but each account may be accessed by more than one customer, you 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 of the 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.

Needless to say, effectively tracking the locations, devices, and login sessions of customers 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 use of entities is very common in recommendation systems, which typically utilize multiple large learned embedding tables. To prevent hash collisions, these tables are indexed with multiple lookup values created by many different hash functions. The collection of hashed embeddings is aggregated together to form a single 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 hash embeddings commonly associated with recommendation systems to uniquely represent every possible entity, we could instead elect to simply try to differentiate between the entities available within an observation's context. In other words, we don't necessarily need to know which device a customer is currently using, but we just need to know whether any device is also used for other events within the context, and, if so, which events use the same device. This can be done by assigning a random integer to each unique device in the context. We can guarantee that there will be, at most, the same number of unique devices equal to the size of the context window. The random lookup value assignment can change between each instantiated training observation to prevent overfitting. This introduces an element of randomness into the model during inference, however, the model is trained to expect this randomness.

The special tokens (`[MASK]`, `[UNK]`, and `[PAD]`) are always fixed. In other words, these values are never hashed.

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.

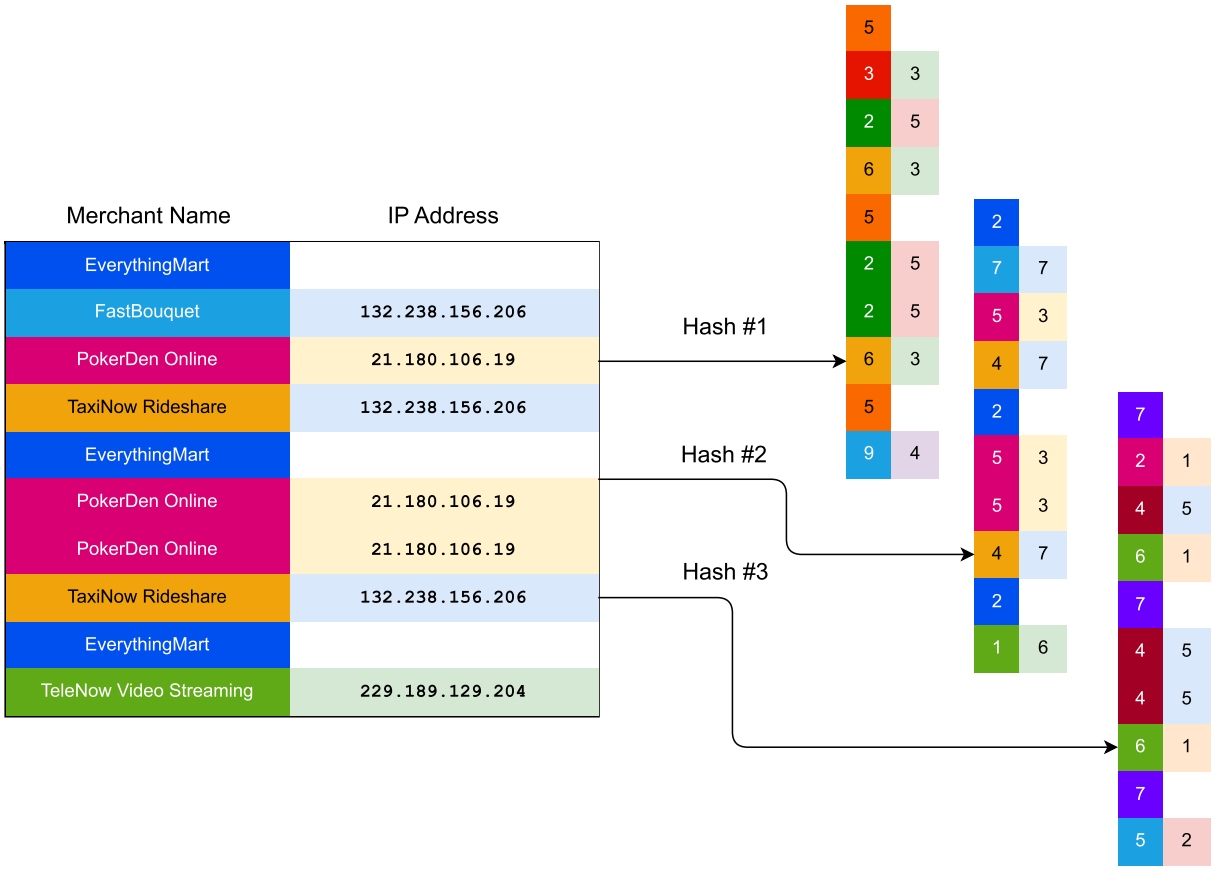

FIGURE 5. How the same observation of two entity fields may result in different lookup tokens during training

```python
def hash(values: list[str]) -> list[int]:
    '''Embeds 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 need to be replaced with the fixed [UNK] token
    mapping.update({"": Tokens.UNK})

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

The usage of these "anonymous" entity embeddings has a few benefits over traditional hash embeddings. For one, they prevent overfitting because the model is not able to 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:
```

```
    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),
        value=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 plus the three non-valued states). We are simply trying to differentiate these unique values from one another. Devices, login sessions, and merchants are ephemeral and infinite, and this novel mechanism allows us to represent them as such without storing many different large embedding tables.

0.6.7. *Putting it all together: MFES and examples.* The MFES can embed discrete, continuous, temporal, geospatial, and entity field types over large sequences of data. That means 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 extremely versatile. It can work with many different data types. Additionally, it can work with your data exactly how it already is. Similar to language and vision, there is effectively zero batch preprocessing.

0.7. **Model Encoder Architecture.** Upon the creation of the field embeddings with the Modular Field Embedding System, TabBERT's hierarchical transformer-encoder architecture can learn sequence and event representations as before. 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. The contextualized field representations are then concatenated to form event representations for every event. Each event representation then attends to every other event representation in the event encoder, which produces contextualized event representations, as well as a sequence representation via a special `[CLS]` token, which is appended to the inputted event representations. The event representations are used to pre-train the model, and the sequence representation can be used for arbitrary downstream supervised learning problems.

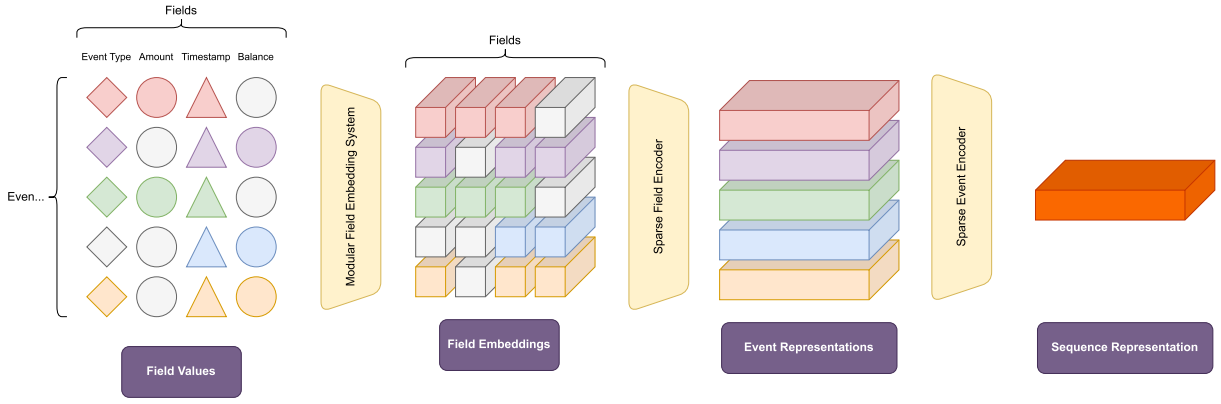

FIGURE 6. The consolidated model architecture outputting a sequence representation

0.8. **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.

Both TabBERT and UniTTab describe their observation sampling strategies as being fixed and occurring at an event level. Their observations are "fixed" in that they are formed as strided slices over all of the events of each sequence. Both TabBERT and UniTTab elected to utilize strided slices of events to mitigate potential data leakage, as the observations created from any sequence, after batch preprocessing, could be used in the train, validation, or test strata. In other words, one slice from a sequence could be used to train a model, and then a neighboring slice could be used to validate it. In order to mitigate the risk of data leakage, their strategy requires that the observations never overlap.

This event-sampling technique is unique to this problem. In computer vision, you can uniformly sample from the population of images. In language modeling, you can uniformly sample from the population of documents from a corpus. However, for complex sequence modeling, sampling observations at a sequence level does not necessarily

work because some sequences may have very few events. For example, you may be running a financial institution with 10 million customers, each with a credit card. However, a large portion of your 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 having sampled any notable portion of the events belonging to more active customers. This results in a long time to train and poor performance because the model has over-fit on the less active customers and under-fit on the more active customers.

We elected to pursue a different approach whereby each sequence in its entirety 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 that this will improve the generalizability of the model, as this strategy disallows the model from ever training on any data available anywhere within the sequences that are used for out-of-sample analyses.

By being able to generate 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. This means that significantly less work is required before the model may begin training. Additionally, the data preprocessing does not need to be executed multiple times among consecutive experiments even if the hyperparameters change.

Additionally, by generating observations randomly we can generate significantly more unique observations. 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 unique events in the entire ledger, our methodology can randomly sample c_e many unique observations during both pre-training and fine-tuning. This effectively provides a means of data augmentation that, in theory, improves both model performance and computation requirements.

However, as both TabBERT and UniTTab highlight, sampling too many overlapping observations from a sequence can quickly result in overfitting. By "overlapping observations", we are referring 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 with which 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 sampling operation comes with replacement, such that during subsequent epochs each event may once again be used to form an observation. In practice, r_s should be equal to or less than $\frac{1}{L}$. This 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. As such, our methodology supports far more robust testing on the out-of-sample test stratum because it will create an observation for every single event in every sequence in the test stratum.

0.9. **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 are not able to utilize. The modality of language has benefitted enormously by 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.

0.9.1. *Masked Event and Masked Field Modeling Tasks.* Applying SSL to our data is tricky, however, because every field can be of a different type. We must ensure that each field type can support non-valued states so that we can mask any field. In addition to being able to mask any field, we also need to be able to decode the contextualized event representations back into their original values. This is very easy for a discrete field, however it is more difficult for the other 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

0.9.2. *Semi-Ordinal Classification of Quantiles.* For the continuous, temporal, and geospatial fields, instead of attempting to reconstruct the original CDFs, we can instead attempt to reconstruct the binned CDFs. This requires that we set a hyperparameter, q, to define the number of bins for the customer SSL task. Going forward, let's assume we want to calculate the number of deciles ($q = 10$).

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 learn that this value belongs in the 6th decile. Every masked CDF should be mapped to the bin $\lfloor \mathrm{CDF}(x) \cdot q \cdot 0.\overline{9} \rfloor$.

However, this alone is imperfect. 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 says that the masked value belongs in the 8th decile, but it belongs in the 10th decile, then it should punished 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 as well.

If a masked field is "padded" but the model guessed that the value is in any decile (or is null), we want to fully penalize the model. Additionally, if a masked field is of any decile, but the model predicted that the masked field is either null or padded, then 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.

Geospatial fields are more complicated, however, because lines of longitude wrap around at the prime meridian. This requires an adaption such that the 1st decile is as close to the 10th decile as the 9th decile. This is only relevant to the longitude tensor of geospatial fields. However, in practice, this is only applicable to locations surrounding the prime meridian.

0.10. **Fine-tuning Complex Sequence Models.** The sequence representation, generated from the equivalent of a `[CLS]` token, is a tensor of fixed size equal $\mathbb{R}^{F \times C}$. The sequence representation is passed through a small MLP that can be trained to resolve arbitrarily supervised learning tasks, including classification and regression. By fine-tuning the sequence representation, you may also use the sequence representation for similarity search or clustering problems as well. A properly pre-trained complex sequence model can be fine-tuned in a matter of hours on modest hardware.

However, because it is not of a universal modality like language, vision, video, or audio, the pre-trained models are only applicable to problems that share their fields.

0.10.1. *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 event-level supervised tasks: classification and regression. In other words, we support different target labels depending on which event was sampled. This is in contrast to sequence-level tagging, which would restrict the target to be homogeneous for every event in a sequence.

Additionally, we have enforced a mechanism that we call "nullable tagging", such that one can include null values in their targets. Events with null targets can be used in the historical context of future events, but they can not be sampled as events. The purpose of this mechanism is to allow model developers to be able to provide the signal of an event for any training observation without necessarily requiring that it be used to create a training observation itself.

For example, in the use case of 3rd party fraud, a model developer might have a select few events that they strongly believe to be associated with fraudulent activity. However, the events that occurred before the "confirmed fraud" events might be more ambiguous. Therefore, model developers might prefer to allow these ambiguous events to be used in the context to create observations for the confirmed fraud events while also disallowing the ambiguous events to be used to create observations.

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

0.10.2. *Dynamically Managing Class Imbalance.*

In addition to the random observation sampling strategy previously discussed, we introduce a novel strategy 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 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 as well.

0.10.3. *Model Explainability.*

Neural network architecture often comes with a significant cost to model explainability. However, model explainability can be an absolute necessity 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 the aforementioned 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 then iteratively prune each field in the input. By "pruning", we refer to permanently masking every value of a field among all events with a special token: `[PRUNE]`. The model is then fine-tuned using every field that is not pruned without having access to 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 change to model performance. We then run another set of pruning for every field and the running list of the least important fields. With this technique, we define an ordered sequence of the least to most important fields with only $\frac{F\cdot(F+1)}{2}$ fine-tuned pruned model instances. Because much of this computation may be executed in parallel among multiple machines, this may be executed in the same amount of time that 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 choose to permanently prune the least important fields or fields that otherwise pose model governance risks that are not justified given the field's contribution.

On the note of model explainability, the usage of succinct fields has another significant benefit: they are simpler to monitor and simpler to 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 that is used to construct tabular features from the ledger may lead to unexpected behavior.

0.11. 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 that can perform all of these stages automatically. This is possible by engineering a framework that executes the above strategy as a function of the data and hyperparameters.

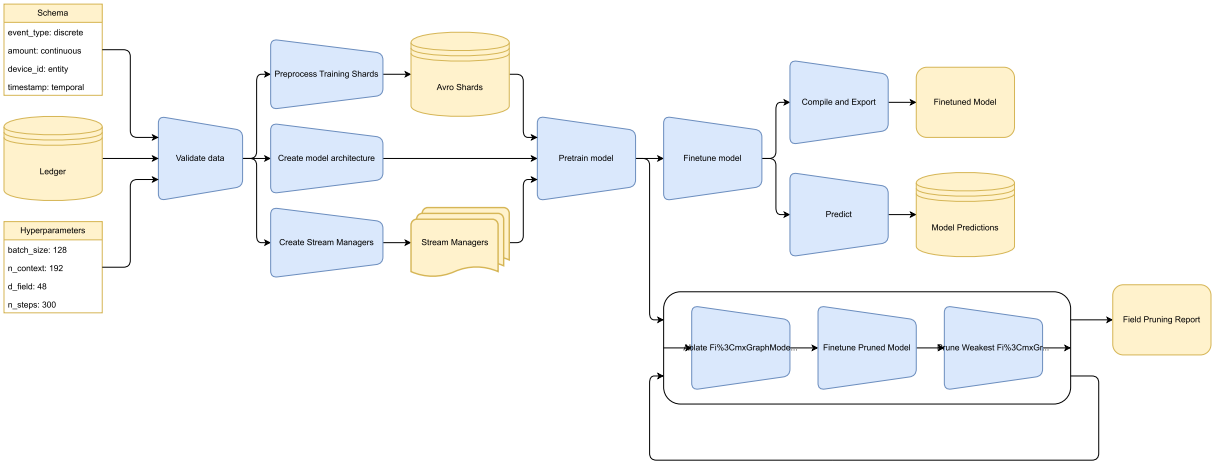

FIGURE 7. The model training pipeline in its entirety

0.11.1. *System State Managers.*

Unlike many other data science pipelines, however, this pipeline requires a wide variety 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 in addition to calculating effective sampling rates per class label for downsampling and the effective loss function weights for upsampling.

Split Manager Contains how to split 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 Calculates and serializes the `TDigest` object for each continuous field, and can reconstruct 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 the initiation of a model and to "tokenize" 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.

0.11.2. *Pipeline Workflow.*

Six major 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 of the necessary system state managers required to solve the task at hand.
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.

0.11.3. *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

0.11.4. *Peripheral Pipeline Steps.* Because the orchestrated pipeline has been developed to 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 quality of the data.
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 report on model performance on out-of-sample data.
8. Generate a whitepaper on model hyperparameters, including information from the aforementioned steps for automated model version documentation.

0.12. **Further Research.**

0.12.1. *Media Fields.* Using standard field types (discrete, continuous, temporal, geospatial, entity field types) solves a great deal of common business problems. However, there are more advanced data types that may also be utilized, such as raw text, images, audio, and video. These are more complicated and rely upon pre-trained models specific to each modality.

0.12.1.1. *Textual Field Embeddings.* We can also *potentially* use raw string text. This one is a little bit tricky, however. In effect, this is akin to running BERT inside of the modular field embedding system before passing the textual representation into two more transformer encoders. As you can imagine, this requires enormous amounts of memory for model training. The computational requirements are especially hard because the effective batch size for this BERT model becomes equal to `N*L*F`. In other words, you need to individually encode every textual field of every event of every observation simultaneously.

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 of the textual fields.

2. Use multiple instances of a smaller pre-trained BERT model and fine-tune them for each of the textual fields.

The BERT model would likely need to be limited to work with an extremely small token context to prevent running out of memory, hindering the quality of its field embeddings.

Additionally, pre-training textual field embedding modules is challenging. Running masked-language modeling *inside* our custom self-supervised learning task is not possible. That being said, if the BERT model is already pre-trained, it might be best to leave it frozen (thus slightly alleviating the memory requirements). At this current point in time, we do not plan on including a custom textual field event decoder to facilitate the self-supervised learning task. Instead, they will be frozen during pre-training and may be unfrozen during fine-tuning.

```python
@tensorclass
class TextualField:
    document: Int[torch.Tensor, "N L T"]
```

0.12.1.2. *Images, Video, and Audio Field Embeddings.* In addition to working with just embedded textual fields, one could *theoretically* include images, audio, and video data as well. In other words, any event of any sequence could include a reference to a file. As a streaming operation, that file is fetched, opened, preprocessed, and embedded.

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 include the following fields: `timestamp`, `content`, `device`, `location`, `photo`, `video`, `is_deleted`. The photo (or video) is then embedded, as are all of the other fields. Additionally, each Twitter user may post a `ProfilePictureUpdateEvent`, which may include the following fields: `timestamp`, `device`, `location`, `video`.

Streaming the operations to read, decode, and collate images will certainly add complications to the data streaming pipeline, however, there are some ways to optimize it. For example, you could store each sequence as a small file, and then store all of its associated media in separate files. All of these files for each sequence are compressed into a single `TAR` file, which could then be streamed through and sampled. If any of the media fields include a file path name, that file will be read from, decoded, and collated.

```python
@tensorclass
class ImageField:
    image: Int[torch.Tensor, "N L X Y"]

@tensorclass
class VideoField:
    video: Int[torch.Tensor, "N L V X Y"]
```

The usage of media is complicated but emphasizes the versatility of approaching your data with sequential modeling. In practice, using asynchronous streaming operations (RayData) is key to scaling these more complex media fields.

0.12.2. *Including Tabular Data and Multiple Sequences of Varying Event Rates.* We have not yet explored integrating static tabular data with sequential data, although this approach has been explored by Visa Research in their FATA-Trans paper. We are also curious about working with multiple sequences of data per account, in which each sequence has different event rates. For example, one sequence of 48 monthly billing statements over the last four years, another sequence for the last 256 customer financial transactions, and a final sequence of the last 2048 online click stream events. This 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 implementing multiple sequences 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? This would require each click stream event to contain an index to its corresponding monthly statement.

Regardless, the utilization of multiple sequences and tabular data is yet another solution to the aforementioned "flushing problem" because the signal may be captured from multiple potential sequences of varying event rates.

0.12.3. *Sub-quadratic Alternatives to Self-Attention.* An interesting 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

0.12.4. *Sparse Attention.*

Currently, the modular field embedding system has a static definition of special tokens, such as `[PAD]` and `[UNK]`. These may be used to facilitate sparse attention for the field encoder and event encoder to improve memory requirements. Within the field encoder, an attention mask may be created that masks out fields that are equal to either `[PAD]` or `[UNK]`. Within the event encoder, an attention mask may be created that masks out events in which all of the fields are equal to `[PAD]`. Given high enough sparsity, this may result in significantly better memory requirements.

0.13. **Appendix.**

0.13.1. *Data Preprocessing for Efficient Streaming Operations.* I found success in storing the data in a row-major format for custom streaming operations. `ndjson` and `avro` are both good options. We found that `parquet` had issues supporting complex, nested fields. We store the data in many small `avro` files, in which each sequence is stored as an object.

Within each sequence object, we store the following items:

- The sequence's unique ID, such as the customer ID (`str`)
- The unique event IDs of each event (`list[str]`)
- event fields (`dict[str, list[int] | list[float] | list[str]]`)
- event targets for regression or classification tasks (`list[int] | list[float]`).

With this format, we may sample any event index number, and then slice the context directly from the event fields (`slice(max(0, idx - L), index)`), which can be trivially padded and collated to a tensor, which may then be stored in a `TensorDict` of tensorclasses.

This data structure ensures that the grouping and filtering operations required to collect all of the events for each sequence will not bottleneck the model training process. It optimizes streaming larger-than-memory sequences from disk to accelerate model training and inference. As a reference to the `pytorch-lifestreams` python library, developed primarily by Dmitri Babaev, we refer to this data structure as a ***lifestream***.

As mentioned above, this preprocessing pipeline requires the calculation of CDFs as a streaming operation. This operation can be extremely expensive. The TDigest implementation specifically from the `pytdigest` package is extremely optimized. According to their benchmarks, they are 10x to 1000x faster than other open-source implementations.

As a side note, performance may be further improved by utilizing another data format, such as `protobuf`. We expect that `protobuf`, as an alternative to `avro`, would be approximately 30% faster to decode at a negligible cost (~10%) to the size of compressed files.

0.13.2. *Sequence Entropy.* A unique characteristic of such multivariate sequences is that they are not nearly as structured as human language. Multivariate sequences can be quite noisy, whereas human language is bound by the rules of grammar. Human language has evolved to be structured, redundant, and fault-tolerant. For example, An adjective must always be followed by another adjective or the noun they describe. If you were to shuffle all of the words in a sentence, the fact that it was changed will likely be obvious. However, if one were to just randomly mask out a random word, you may still have a very good guess as to the general message of the sentence.

On the contrary, complex sequences of transactional data tend to have relatively less structure. If you shuffle all of the events in a sequence, it will likely remain just as "plausible" as before. However, if you were to mask out a random event, it would be much more difficult to reconstruct. 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 right answer. The order is almost completely arbitrary.

Additionally, pre-training a large language model on every human language is extremely complicated 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 there is less structure to be learned, thus requiring significantly fewer resources.

0.13.3. *Arbitrary Dimensionality.* In all of the examples above, we use a specific dimensionality: Each sequence contains events, and each event contains fields. In other words: `Customer > Transaction > Field`. This is extremely simple and 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 multiple 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. The usage of sequential modeling is still applicable. We do not currently intend on developing this 3D implementation, however.

0.13.4. *The Clipping Problem.* I came across an interesting complication while training sequential models on slices of time series. It is intuitive in hindsight, but it is challenging to identify.

It is common to subset slices of time series based on some time window (IE: all events between January 2020 and January 2023). This can lead to very unexpected behavior, however, due to an issue that we refer to as the "clipping problem".

Simply put, the model may sometimes not have enough information to understand if a sequence with an event in the onset of the window is truly the first event in the sequence, or if it just looks like the first event in the sequence because the prior events were filtered out. There is ambiguity in what had happened before the context.

This problem has multiple solutions:

1. Include a "buffer" window in the data, during when the model may sample the events as historical context, but not as the event at the end of an observation
2. Include information regarding the start of the sequence, such as the creation of the account (`AccountOpenEvent`) or a field that provides the time since the sequence started (`account_tenure`)
3. Instead of creating a fixed time window (Jan 2020 to Jan 2023) at an event level, perhaps filter your data to only accounts that were opened between some two dates (depending on your use case)

All three of these solutions prevent the ambiguity between an observation that includes an artificial cutoff versus an observation that includes the start of a sequence.

0.13.5. *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 simply "log in" and "log out" of their account several thousand times to attempt to manipulate the model in some way.

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 point in 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 truly are.

The risk of such an attack may be mitigated by multiple means:

1. Increase the context size.
2. Set up policies to flag high event rates, where needed.
3. Do not include trivial events (log in or log out events) in the context, but rather 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. 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)
7. Protivinsky, T.: PyTDigest, (2023)

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