Project Eidolon: Unknown Unknowns Detection for Geolocation Transactions

GeoComply ML/AI

December 20, 2023

Abstract

The present document accompanies Eidolon: a reconstruction-based unspuervised unknown unknowns detection system for geolocation transactions. Our first task is to motivate our work: we begin by setting up the context enveloping our project and clarifying our vision for our system's place within ML/ AI @ Geocomply. We then formalize our hypotheses and objectives and present our early achievements consolidating our confidence in this research direction. Our next task is to take a deep-dive into our design, architecture and implementation. We present and analyze our solution. We provide detailed documentation on the project's API and ellucidate our approach to hyperparameter selection. We illustrate our process on production data collected from the PLC and NDBS solutions. We conclude the report with suggestions for future research.

Contents

1 Introduction: Unknown Unknowns Detection

The present document accompanies Eidolon: a reconstruction-based unspuervised *unknown unknowns detection* system for geolocation transactions.

Unknown unknowns detection refers to the discovery of—previously unknown—strongly predictive signals present in large datasets. The discovery of such signals should be accomplished with minimal feautre engineering and bias. Unknown unknowns detectors are meant to efficiently handle large datasets in relatively raw format so as to discover latent patterns automatically. Furthermore, it's crucial that these systems are able to communicate their discoveries to the user: there is an emphasis on expressive model interpretation.

In the context of fraud detection in geolocation transactions, unknown unknowns detection aims at the discovery of features (or combinations thereof) capturing—previously unknown—spoofing patterns within potentially geo-restricted segments.

1.1 Context & Vision

Prior to the conception of this project, our ML & AI teams had been working on both *supervised* and *unsupervised* learning solutions. We are interested in the development of an unknown unknowns detection system both for the value it would add on its own and through its interaction with our other endeavours.

On its own, a successful unknown unknowns detection system would help us leverage the full power of our historic data. It would allow us to improve the recall of our rules-based engine—without sacrificing its precision. Over the years, we have accumulated enormous transaction archives. Each transaction contains hundreds of features. The spoofers—not having access to this data—are unaware of how to appropriately mask their synthetic signals. The goalposts are not visible to them: they don't know what it means to align with all of the organic transaction patterns represented in our data. As things stand, we can't see the goalposts either. As a result, we don't know when the spoofers' shots miss them. However, this knowledge lives in our data: the onus is on us to unearth it. It's up to us to spot more of the patterns at our disposal so as to come up with ever stricter—yet precise—rules—: rules capable of detecting even the most subtle spoofing instances without increasing false positive counts.

Our supervised learning projects are primarily concerned with the development of binary classifiers for the identication of spoofed transactions. This is an active pursuit across multiple solutions: PLC, iOS, Android, Windows and NDBS. We are advancing on multiple fronts: procurement of suitable training transactions, assignment of accurate training labels, optimization of automatic model retraining and careful feature monitoring for model optimization. Supervised learning models are intended to operate at a very high precision level. Ideally, we'd like to build classifiers able to work alongisde with—or even replace—our engine rules.

Paramount to the success of this work is the quality of our training data. Two factors are important here: the representation of the full extent of fraudulent behaviour and the accuracy of the labels. Recently, we have started building up an archive of manually labeled training transactions: the so called *grountruth* dataset. This is intended to help us improve with respect to the second factor. Naturally, we are interested in discovering ways to help ensure that our training data contain samples that span the full extent of spoofing activity. This is a challenging task: there's no way to really know when our data is really exhaustive. What's more is that the goalposts are constantly shifting: fraudsters are innovating quickly—and we need to keep up with them.

Implementing an effective unknown unknowns detection system would greatly assist us on this front. Such a system should be able to parse production data and identify new kinds of fraud as they emerge. These findings would be communicated to our RiskOps teams who would then manually verify them and include them in the ground-truth dataset. This procedure would strengthen our supervised models: they would thereby gain the ability to identify subtle and novel fraudulent transactions—an outcome we cannot hope for if certain fraud types are absent from our training

data.

An effective unknown unknowns detection system could also assist us in the feature monitoring component of our supervised learning research. Most relevant architectures would leverage a learned transaction embedding. This embedding could be used as a feature in supervised models. The idea here is that the embedding should correlate well with the fraud patterns understood by the unknown unknowns system producing it.

Our unsupervised learning projects have been focusing on the development of narrow-scoped anomaly detectors: systems able to spot irregularities in particular aspects of our transaction data. Examples include the detection of anomalies relating to:

1. location data,
2. altitude data,
3. WiFi access point specs,
4. browser specs,
5. running processes,
6. virtual machine activity.

Restricting the detectors' scope allows us to tune our models to the task at hand, ultimately designing systems able to operate well within their designated domain. These systems are effective in discovering novel signals manifesting within their narrow-scoped inputs. This comes at the cost of their ability to discover signals generated by the interaction of inputs in different categories. Furthermore, all of these systems have been designed with some form of bias in mind: they are created to spot irregularities of a given kind—already known to the system designer.

A successful unknown unknowns detection system would help generate project ideas for targeted unsupervised anomaly detection. The signals discovered by such a system would help us improve our understanding of our own proprietary data. This would guide our bias in the right direction—to subsequently be channeled into the design of new targeted unsupervised learning systems.

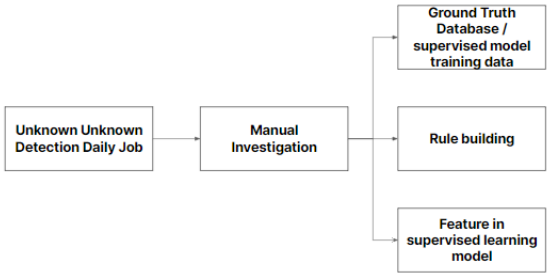

Figure 1: Unknown Unknowns Detection Vision

1.2 Hypotheses

Our work is motivated by two central hypotheses:

1. **Feature Volume Hypothesis:** We hypothesize that spoofers are unable to consistently mask all of the features we record. The odds are stacked against them: a single slip-up will give them away.

   This relates to our earlier discussion on our vision for the impact of unknown unknowns detection on rules optimization. It's the centerpiece of this study and of strong interest to senior leadership.

2. **Spoofing Scarcity Hypothesis:** We hypothesize that spoofed transactions are strongly underrepresented: their impact on transaction statistics is weak.

   The practical implication of this hypothesis is that unsupervised learning (and in particular anomaly outlier detection) can be leveraged to extract spoofing-related unknown unknowns. Had we been interested in discovering signals related to a common transaction class, there would be no hope in this endeavour. It's interesting to note that spoofing scarcity is often brought up in the context of our supervised learning research, where it leads to undesireable class imbalance. In this respect, our research direction rehashes a common liability of fraud detection ML/ AI applications into an asset.

1.3 Objectives

In order to test our hypotheses (and leverage their validity in case they are found to hold), we set out to create an unknown unknowns discovery system based on unsupervised anomaly detection.

Our objective was to design and implement a model capable of identifying and explaining each of the following **three** anomaly types:

1. **Marginal Distribution Anomalies**: out-of-distribution values.

   Marginal distribution anomalies refer to outlier values found within samples from univariate distributions of individual features. They are the simplest of the three anomaly types we aim to detect. To illustrate, an example would be identifying high WiFi altitude values within an Ontario-based dataset (Ontario is flat).

2. **Conditional Distribution Anomalies**: unusual combinations of values in different columns.

   Conditional distribution anomalies refer to feature combinations that violate majority-class patterns. An example would be given by mismatches between highly correlated columns (e.g. the `ip_country_code` and `wifi_country_code`).

3. **Representational Anomalies**: anomalies related to the representation of a record, rather than the value it denotes.

   By *representational anomalies* (a non-standard term) we refer to irregularities in the string representation of numeric data. Our current NDBS rule system conditions on the number of digits in lat/ lon records. We hypothesize that synthetic requests sent by spoofers can result in various unknown representational irregularities in the fields they mask. One of our primary objectives was to design a general algorithm capable of detecting them.

Under our hypotheses, our historic transaction data will be rich in spoofing instances exemplifying these three anomaly types. The validity of the *feature volume hypothesis* will result in the existence of spoofing transactions violating latent patterns. The validity of the *spoofer scarcity hypothesis* will result in such transactions being a small minority. Consequently, when viewed alongside our extensive transaction records, they would constitute anomalies.

The successful implementation of an anomaly detector able to:

1. flag these outlier transactions,
2. attribute its decision to profound novel signals related to spoofing,

would smiultaneously provide positive evidence for our hypotheses and yield a working unknown unknowns detector leveraging their validity for the advancement of the company's interests. Our objective is to develop such a system.

1.4 Early Achievements

Early versions of the model were applied to production data from the PLC and NDBS solutions. To be specific, we studied the PLC US7 complete history from 01/08/23 to 07/08/23 (301,377 transactions, 56 utilized features) and the NDBS BET99 ON & Betano ON complete history from 01/11/23 to 15/11/23 (384,343 transactions, 277 utilized features).

The results quickly demonstrated potential. By inspecting the top transactions flagged by our system, we immediately spotted multiple interpretable anomalies. Identified signals included:

1. Mismatches between the IP and WiFi geolocation sources.

loss_rank	gc_transaction	label	gc_authorized	username	device_uuid	ip_long	wifi_signalStrength_min	ip_region_code	ip_organization	ip_timezone
1	5580a2c06af58267	0	1	115480514	9E7C94EB-BD96478E-4C165E05-80BAF748-357BDBFD-D...	-74.2274	-64.0	NJ	Windstream Communications	America/New_York
2	b7d0339e20b1cd6f	0	1	4D8C472DBBC57D27	97B226B4-61E9FC0A-DB15CC22-92F779E6-23E5765C-2...	-74.2274	-59.0	NJ	Windstream Communications	America/New_York
5	0692aba0ebf4bf5a	0	1	29858895	9E7C94EB-BD96478E-4C165E05-80BAF748-357BDBFD-D...	-74.2274	-64.0	NJ	Windstream Communications	America/New_York
6	ff4eb725ff885916	0	1	4D8C472DBBC57D27	97B226B4-61E9FC0A-DB15CC22-92F779E6-23E5765C-2...	-74.2274	-65.0	NJ	Windstream Communications	America/New_York
7	756f826c36dcda48	0	1	4D8C472DBBC57D27	97B226B4-61E9FC0A-DB15CC22-92F779E6-23E5765C-2...	-74.2274	-63.0	NJ	Windstream Communications	America/New_York
8	ca3ad07f3e965efb	0	1	4D8C472DBBC57D27	97B226B4-61E9FC0A-DB15CC22-92F779E6-23E5765C-2...	-74.2274	-59.0	NJ	Windstream Communications	America/New_York

Figure 2: IP/WiFi Mismatches: Spoofing Colorado Transactions with NJ IPs

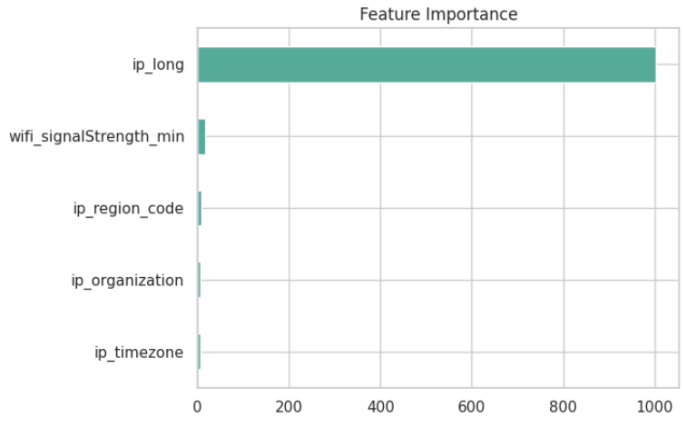

Figure 3: IP/WiFi Mismatches: Interpretation

2. WiFi coordinate representation anomalies. Trailing zeros in the WiFi coordinates of transactions exhibiting IP/ WiFi mismatch. The WiFi was in Miami and the IP in Argnetina. The WiFi coordinates appear to have been masked.

	gc_transaction	label	status	ip_connection_type	ip_lat	wifi_lat	ip_long	wifi_long	ip_wifi_dist_km	ip_accuracy_km	wifi_accuracy_km
4041	f49f9f39fc891d47	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
887	f5bd81964873c8e3	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
4926	e5acdd543f0de77d	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
1357	3ea8388441fef0e0	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
361	33144fbc50f8de21	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	20.0	0.150000
1247	c27362fc4b97cb49	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	20.0	0.150000
2762	83749afc7713fbed	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
1249	bb9b64dbb51879bd	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	20.0	0.150000
6686	842ab9917705ae0d	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
4713	8b2e785938a790b8	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
1831	4e8397e7d8fa9597	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
3497	481e38eb465a15af	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000
5534	0f23dd3a731210e8	0	1	static	-34.4662	25.743000	-58.9791	-80.123000	7056.315173	10.0	0.150000

Figure 4: WiFi Coordinate Representational Anomalies: Transactions

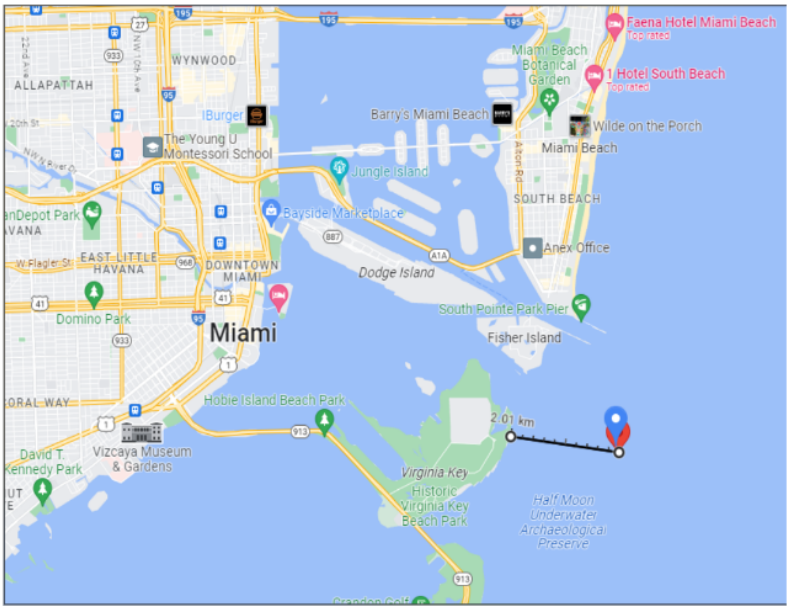

Figure 5: WiFi Coordinate Representational Anomalies: Map

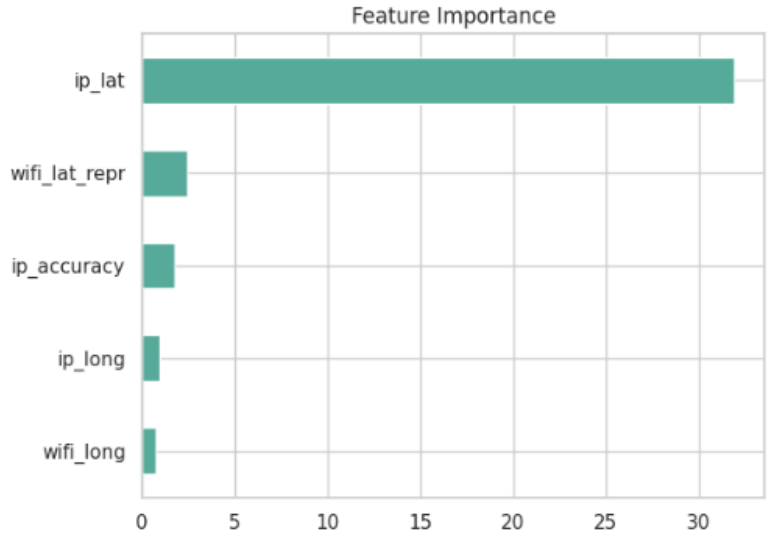

Figure 6: WiFi Coordinate Representational Anomalies: Interpretation

3. WiFi access point irregularities: Multiple access points with identical signal properties.

Time (@timestam...	environment	device_uuid	gc_transaction	operator_name	username	wifi.aps_count	wifi.signalStrength	ip.connection_t...	wifi.channel
2023-08-05 18:38:00	us7	C02D3MVWML7H	7afac8363957dd15	FanDuel CO	29714705	5	-92, -92, -92, -92, -92	static	157, 157, 157, 157, 157
2023-08-05 16:24:45	us7	C02D3MVWML7H	7ab74cb9cd3b4ed1	FanDuel CO	29714705	5	-92, -92, -92, -92, -92	static	157, 157, 157, 157, 157
2023-08-05 16:06:41	us7	C02D3MVWML7H	c1d898fa31468f37	FanDuel CO	29714705	5	-92, -92, -92, -92, -92	static	157, 157, 157, 157, 157

Figure 7: WiFi Access Point Spec Anomalies

4. WiFi altitude irregularities: `wifi_altitude` > 3000 m in Ontario.

loss_rank	gc_transaction	device_uuid	gc_authorized	gc_error	loss	ip_long	wifi_long	ip_lat	wifi_lat	wifi_altitude
56	7c6e0f0c016a4c1d	6C0DD157-0703-4879-B4E4-5B46B1E1E214	1	[]	13.798963	-79.771184	-79.771184	43.955734	43.955734	3268.500000
135	6b179ca52603cb54	668A6626-9109-4815-89C4-C300B48963E9	1	[]	5.940133	-79.720703	-79.720703	43.697929	43.697929	2180.000000
151	af42c5fafee2c823	668A6626-9109-4815-89C4-C300B48963E9	1	[]	5.344795	-79.659282	-79.659282	43.793266	43.793266	2070.000000
152	546d57fcc65f018f	668A6626-9109-4815-89C4-C300B48963E9	1	[]	5.218702	-79.659390	-79.659390	43.793150	43.793150	2010.000000
174	96833661b8f1ebdb	668A6626-9109-4815-89C4-C300B48963E9	1	[]	3.899921	-79.645257	-79.645257	43.707356	43.707356	1780.000000
184	7979acb8be55428d	AD267701-C3D5-4AEC-8A3A-CD78E5D55BBC	1	[]	3.520644	-79.321100	-79.328509	43.852800	43.855892	1623.025600
190	770973bb2e08a117	CD38350B-B315-4634-941C-EB13E1A21C58	1	[]	3.476787	-79.444971	-79.444971	43.717252	43.717252	1661.550000
193	79ca38013609ec59	D166D279-9E06-4AA4-8485-5043AC8EC41C	1	[]	3.457202	-79.606660	-79.606660	43.735960	43.735960	1650.000000
194	10096ae8bfe43bfb	46868744-D097-444B-8C5C-6B1B05840C7A	1	[]	3.352741	-79.442000	-79.328556	43.867800	43.855926	1623.460224

Figure 8: Altitude Anomalies

Our findings instigated manual review aimed at determining whether we'd correctly identified instances of spoofing. Most of the flagged transactions were confirmed as such.

Notably, we identified twenty-four (24) high-risk transactions exhibitng significant mismatches between their IP and WiFi coordinates. Fifteen (15) of these were confirmed to be spoofing cases. This verdict sparked interest in further investigatation into the signal. The findings led to the discovery—and eventual blocking—of 4 distinct spoofer-operated devices. The transactions were added to the groundtruth dataset. This ultimately resulted in reevaluating a recent amendment to our rules system related to the treatment of IPs outside of the US/ Canada. The aforementioned developments are all stronlgy encouraging indicators that our proposed research direction is in alignment with our objectives:

1. Our system was already able to identify a signal responsible for the actionable (blocking) identification of spoofers' devices.

2. Our system was already able to contribute to the breadth of spoofing activity represented in the groundtruth dataset.

3. Our system was already able to contribute to rules optimization.

These achievements are in direct alignment with the objectives set out in the previous sections.

Among the spoofing cases, we also discovered anomalous transactions that did not strictly adhere to our current spoofing criteria. All of them were peculiar in immediatley identifiable ways. The reviewers' verdict was that they were the result of data collection error. This is still an encouraging development in two ways:

1. It demonstrates the capacity of our system to flag interpretable signals not confined within our current understanding of the spoofing landscape.

2. Unearthing unfamiliar data collection errors can help us identify liabilities in our data. By addressing these liabilities, we can improve the performance of our data science efforts at large.

2 Solution

Our anomaly detector is based on the autoencoder architecture. We detect represetnational anomalies by leveraging stacked attention encoders (BERT) and having the autoencoder reconstruct the resulting embedding means in a simple transfer learning procedure.

2.0.1 `TextEncoder`

Our `TextEncoder` network is designed to encode text data. It's an implementation of BERT [3]. Even though—in principle—it could be used to extract meaningful patterns from natural language excerpts, the application we have in mind relates to the identification of irregularities in the string representation of numerical records. Our objective is to produce an embedding that clusters the records into classes so that all regular records group together and all irregularity types correspond to unique and well-separated clusters. We train small (3000 parameters) character-level models

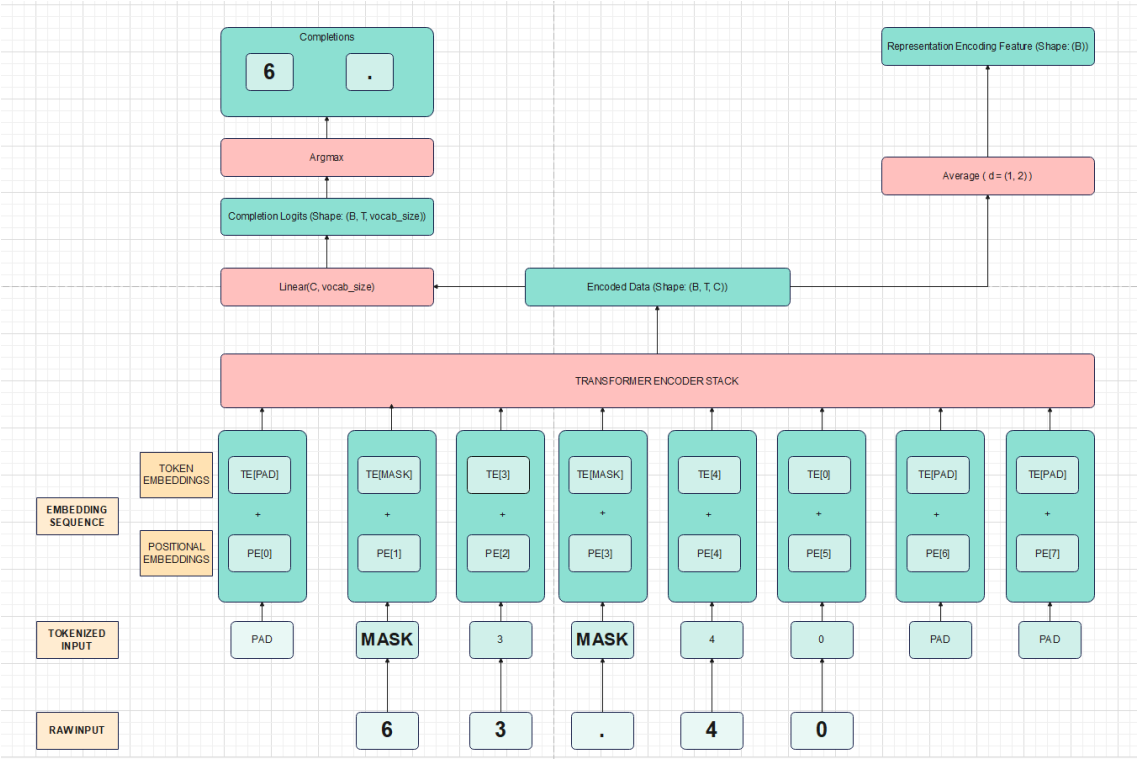

Figure 9: `TextEncoder` Architecture

for each numeric column of interest. This allows us to map entries of numeric columns to tensors of shape:

$$\texttt{shape} = (T, C). \tag{2.1}$$

Here, T is the max string size accepted by the BERT model and C is the token embedding dimension. Averaging the entries of these tensors, we are able to express the representation of any given numeric column through a single derived feature. These features are added to the dataset that will be eventually parsed by the autoencoder.

The network is not directly trained with anomaly detection in mind. Instead, we apply the MLM pretraining procedure from [3]. During training, fifteen percent of incoming tokens are selected—at random—for modification. Ten percent of these are randomly resampled. Eighty percent are replaced with a designated `mask_token` concealing their content. The remaining ten eprcent are left unaltered. The model is trained to unmask the masked tokens (complete the text). In doing so it's forced to seek and leverage latent patterns in the token sequenes. This results in the discovery of an effective represetnation reflecting these patterns. We are not interested in token completions. We merely use this process to gain access to the encoding that facilitates them.

The process is illustrated in Figure 9. The figure depcits the process of extracting the representation encoding of the number 63.40. The first and third characters are masked and the resulting token sequence is passed through the network (stacked bidirectional multi-head attention). The resulting encoding is leveraged for text completion. The encoding is then averaged and the representation feature is extracted.

2.0.2 `TransactionAnomalyDetector`

Our `TransactionAnomalyDetector` network is the centerpiece of this project. It's a feedforward autoencoder designed to support both continuous and categorical features. Figure 10 illustrates the architecture.

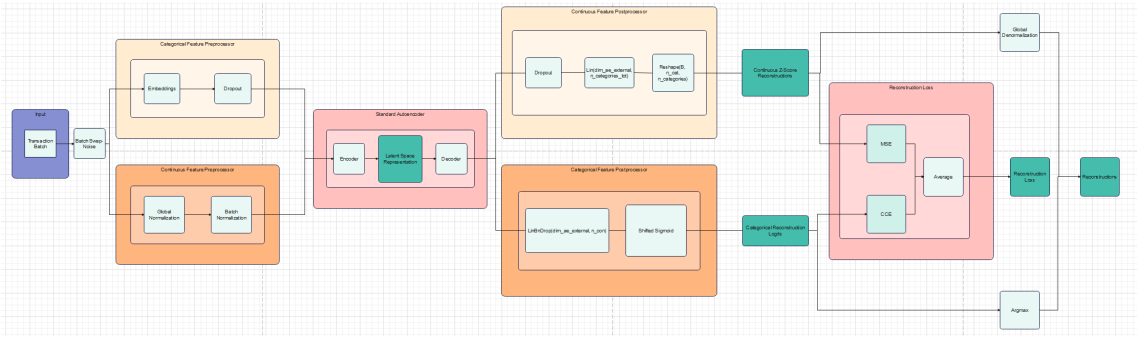

Figure 10: `TransactionAnomalyDetector` Architecture

The model works with tabular data. It expects a certain set of continuous and categorical features. Input records are expected to offer all of these.

The first layer in the network is an implementation of `swap_noise` operating at the batch-level. Like dropout [7], this only operates during training. Its function is to resample a fixed proportion (`swap_rate`) of batch entries from their respective columns. This is intended to force the autoencoder to identify latent intra-feature relationships. The addition of this layer upgrades our model from a simple autoencoder to a *denoising autoencoder*. Without the added noise, we run the risk of having individual features play dominant roles in their own reconstructions. This would go against our intention of capturing hidden unknown patterns.

Continuous features and categorical features are preprocessed differently. At the start of the training process, the model collects global population statistics (min, max, mean and standard deviation) for each continuous feature. It uses these statistics to map incoming values to z-score space. We refer to the corresponding normalization procedure as *global normalization*. Responsible for it is the first component of the *continuous feature preprocessor* (Figure 10). The autoencoder is trained to receive and reconstruct z-scores. Once this is achieved, global normalization can be undone using the recorded global population statistics once more. The second component of the continuous feature preprocessor is a batch normalization layer [4]. This is intended to scale and center the itra-batch z-score values so as to expidate network convergence. Categorical features are passed through learned embedding layers and the resulting embeddings are filtered through a dropout layer whose rate is hardcoded to:

$$\texttt{\_dropout\_rate\_cat\_decoder} = 0.1.$$

This is independent of the `dropout_rate` used in the rest of the network.

The preprocessed inputs are then passed through a standard feedforward autoencoder. This consists of an encoder and a decoder. The encoder compresses the original inputs. The decoder reconstructs the original inputs from their compressed form. To achieve this, the standard autoencoder has to learn latent patterns in the data. When these patterns are disturbed, reconstructions fail. Reconstruction failure can be leveraged for anomaly detection. To encode and decode the data, the standard autoencoder uses stacked linear layers with dropout and batch normalization

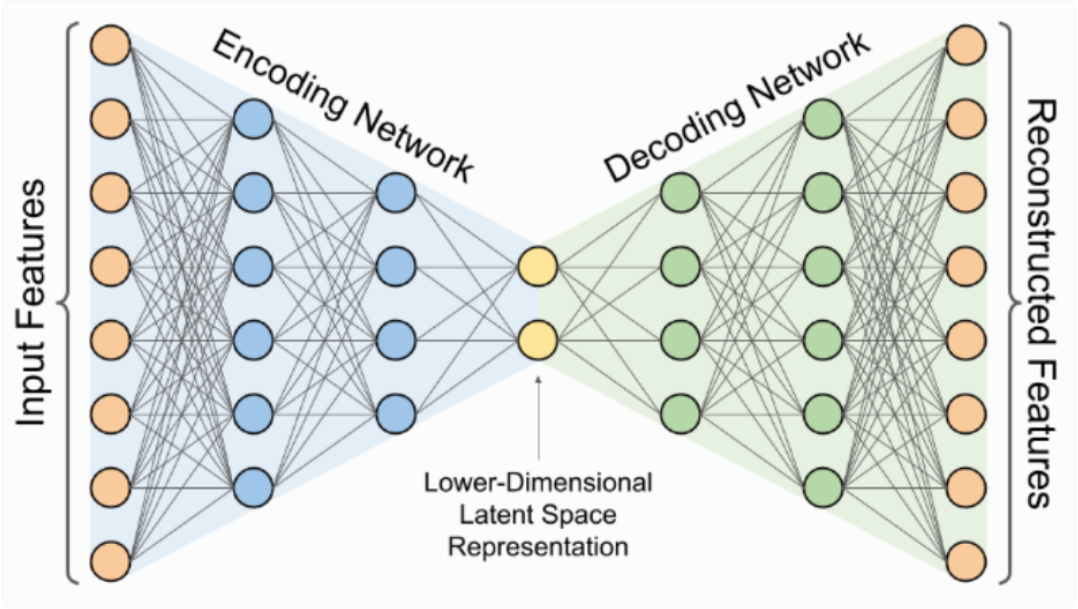

Figure 11: Standard Feedforward Autoencoder

(`LinBnDrop`). The standard autoencoder receives an input signal of higher dimension than the concatenated continuous features and categorical token embeddings (*preprocessed input*). To achieve this, the preprocessed input is passed through a linear layer of configurable shape whose output matches the size of the first encoder layer. The layer sizes of the encoder and the decoder `MLPs` are mirrored (*hourglass architecture*) as in Figure 11.

Once the decoded output has been obtained, it needs to be parsed to form the feature reconstructions. This is the job of the *continuous feature postprocessor* and the *categorical feature postprocessor* modules. It's crucial to note that the decoder's output size does not directly match the number of underlying features (sum of continuous features and categorical embedding dimensions). Instead, it constitutes a reconstruction internal to the model: one that mixes signals from various features together. Both the continuous and the categorical postprocessors cast this signal to the respective necessary dimension. To do so, they each leverage a linear layer. The former filters the output of this layer through parallel sigmoid functions (one for each feature) with horizontal asymptotes determined using the recorded global population statistics:

$$\texttt{sigmoid\_min} = \frac{\texttt{min} - \mu}{\sigma},$$
$$\texttt{sigmoid\_max} = \frac{\texttt{max} - \mu}{\sigma}.$$

Here, μ and σ denote the respective continuous feature's mean and standard deviation, whereas `sigmoid_min` and `sigmoid_max` denote the corresponding sigmoid function's asymptotes:

$$\texttt{shifted\_sigmoid} = \texttt{sigmoid\_min} + (\texttt{sigmoid\_max} - \texttt{sigmoid\_min}) \cdot \texttt{standard\_sigmoid}.$$

The categorical feature preprocessor casts the decoder's output to a vector with length equal to the sum of the embedding dimensions. This tensor is then reshaped to obtain the logits encoding the model's estimates of the respective multinoulli distributions.

Once the reconstructed z-scores and logits have been obtained, their deviation from the original record is measured by the *reconstruction loss* module. This applies MSE to the z-scores and CCE to the logits. The record's reconstruction loss is obtained by averaging the individual feature losses with equal weights.

Anomalies are flagged by learning a reconstruction loss threshold using a configurable quantile.

The latter is determined by passing a `contamination` parameter:

$$\texttt{reconstruction\_loss\_quantile} = 1 - \texttt{contamination}.$$

2.0.3 Anomaly Interpretation & Signal Identification

Once the `TransactionAnomalyDetector` has flagged an anomalous record, it's required to communicate its decision process. This is a critical component of our system: it constitutes the process by which novel spoofing signals are to be discovered. This is achieved by applying the method in [1].

The method is based on Kernel SHAP[5] feature attributions. Shapely values are estimated using local linear regression on a dataset of perturbations of the anomaly (transaction) to be explained. This produces feature attribution scores explaining the reconstruction logic for the features that most dominantly contributed to the total reconstruction loss. In layman's terms, we ask the model to explain its logic where it failed to produce faithful reconstructions. The idea is simple: reconstruction failure correlates to violation of latent patterns discovered during training. A successful expression of the erroenous reconstruction logic should capture these patterns and the way in which they were violated. This is precisely what's required to identify the relevant spoofing signal.

The output of the process can be communicated using plots such as the one in figure Figure 12. The whole plot is dedicated to explain a single anomalous transaction. Rows correspond to the most dominant contributors to the total reconstruction loss. Columns correspond to all available features. Color intensity encodes absolute feature attribution scores (Kernel SHAP coefficients). Green indicates that the feature corresponding to the relevant column helped offset the erroenous reconstruction of the feature corresponding to the relevant row. Red indicates the opposite: a contribution to the anomaly.

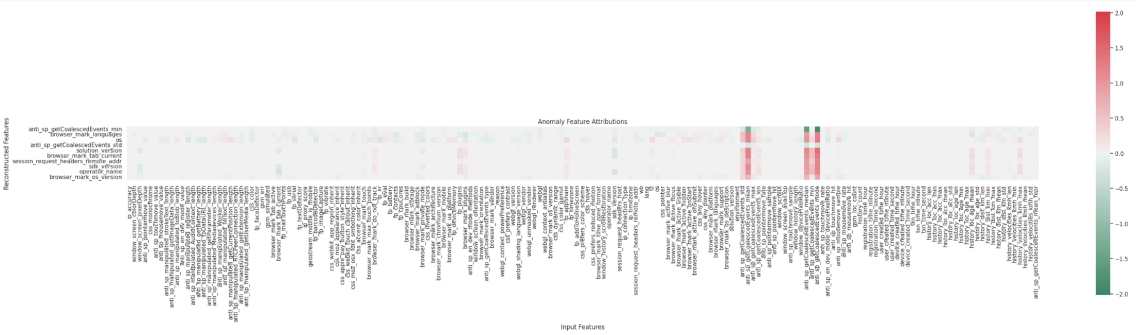

Figure 12: Anomaly Interpretation Plot

2.0.4 Integration: Anomaly Detection Pipeline

We now provide an overview of all the components comprising our anomaly detection process and illustrate how it works to identify the three anomaly types we set out to discover. Note that this does not cover the full extent of our process: it omits our work on data cleaning/ preprocessing and geographical segmentation. These are the subject of section 4.2. Our objective here is merely to illustrate the functionality of our model. In the context of the diagrams that follow, anomalies are indicated using red frames. Normal instances are indicated using green frames.

Overall, our process can be summarized by the following flowchart:

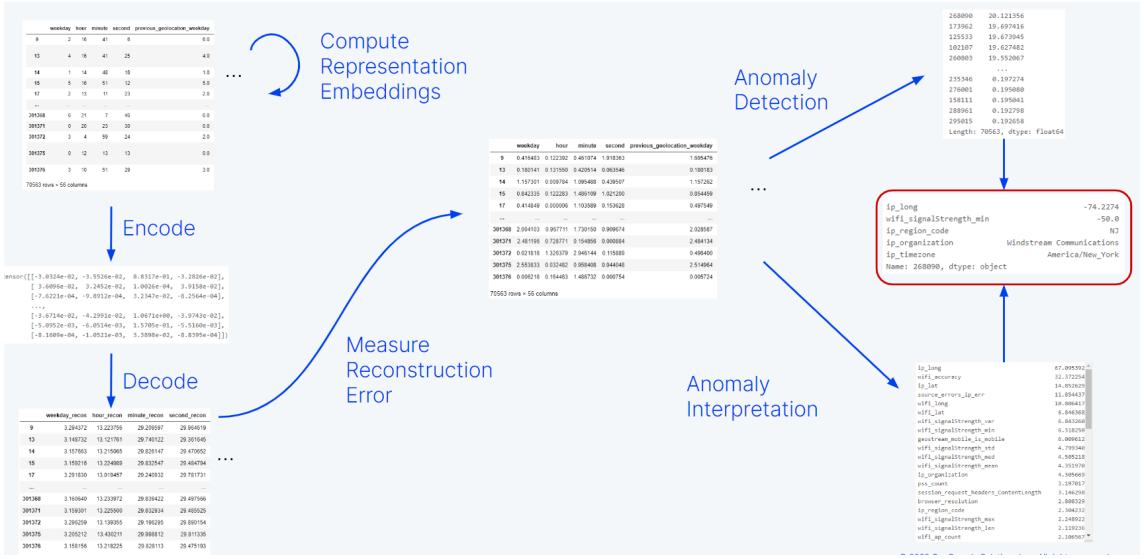

Figure 13: Anomaly Detection Process Flowchart

The model accepts a dataframe of 70000 transactions. These are immediately enriched with representation encodings. The enriched transactions are compressed (encoded) and the compressions are fed into the decoder module. The reconstruction loss is measured and both transactions and features are ranked by it. This leads to the isolation of a single record and ten features exhibitng the WiFi/ IP mismatch signal discussed in section 1.4.

To illustrate how the model detects each of the three anomaly types introduced in section 1.3, we work with synthetic data. Our synthetic dataset contains four features:

1. `wifi_lat` (continuous),
2. `wifi_lon` (continuous),
3. `n_access_points` (categorical),
4. `triangulation_accuracy` (categorical).

The coordinates were randomly sampled around New York City. The number of access points was sampled uniformly randomly:

$$\texttt{n\_access\_points} \sim \mathcal{U}\{2,3,4,5,6\}.$$

The triangulation accuracy was determined from the number of access points uisng the map:

$$\texttt{triangulation\_accuracy} = \mathit{accuracy\_map}(\texttt{n\_access\_points}),$$

where:

$$\mathit{accuracy\_map} = \begin{cases} 2 & \mapsto \texttt{very low} \\ 3 & \mapsto \texttt{low} \\ 4 & \mapsto \texttt{medium} \\ 5 & \mapsto \texttt{high} \\ 6 & \mapsto \texttt{very high}. \end{cases}$$

The flowchart in Figure 14 exhibits the detection of a marginal distribution anomaly: two records have unusual coordinates.

The flowchart in Figure 15 exhibits the detection of a conditional distribution anomaly: two records violate the relationship between the `n_access_points` and `triangulation_accuracy` columns.

The flowchart in Figure 16 exhibits the detection of a representational anomaly: multiple records have trailing zeros in their coordinate fields.

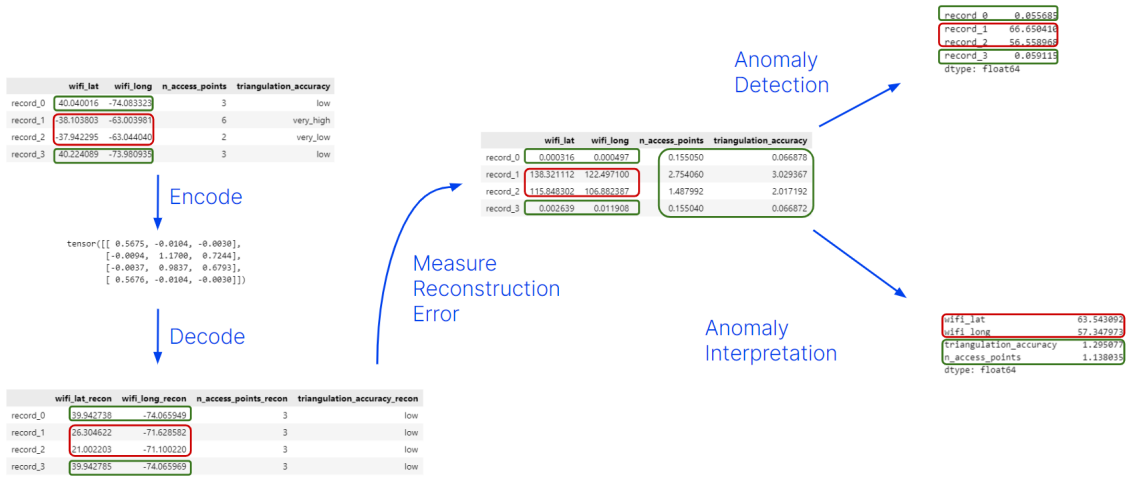

Figure 14: Marginal Anomaly Detection Flowchart

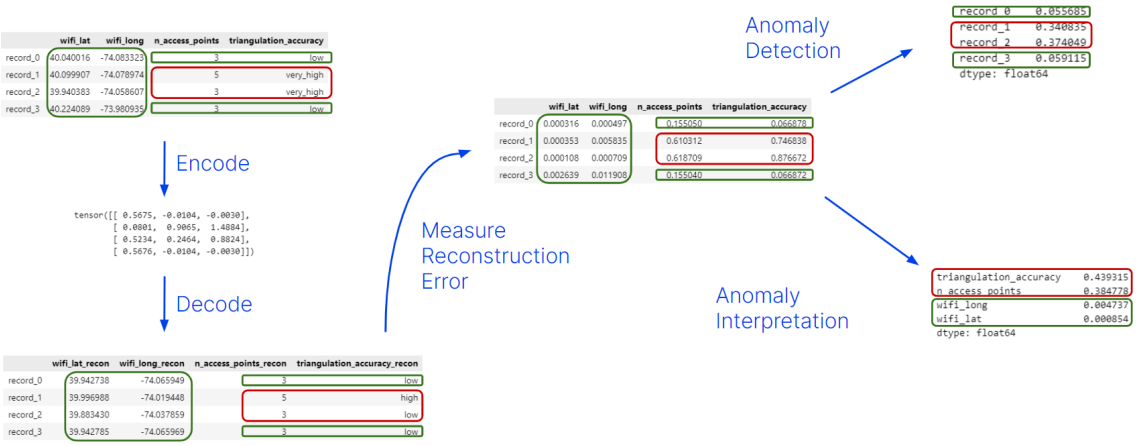

Figure 15: Conditional Anomaly Detection Flowchart

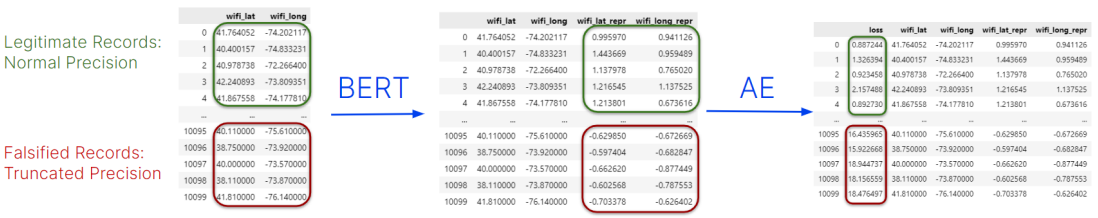

Figure 16: Representational Anomaly Detection Flowchart

3 API Reference

3.1 `clustering`

This directory contains tools to cluster transactions based on their location.

The classes provided are:

1. `clustering`:
   (a) `ClusterIdentifier`.

3.1.0.1 Class `ClusterIdentifier`

Description: An interface for grouping labeled geolocation transactions in nearest neighbour (NN) clusters. The `ClusterIdentifier` is trained on a fixed collection of transactions. It partitions this collection into NN clusters and stores the results internally. It exposes cluster composition statistics to the user, who can then retrieve the desired cluster using an internal indexing system. Input transactions ought to be stored in a geopandas geodataframe including a `label` column. In practice this should either be set to `NOT gc_authorized`, pseudolabels derived from `gc_error` criteria or the manual review labels on `groundtruth`. Class balance statistics derived from the labels are meant to be used as a criterion for cluster selection.

Constructor Parameters: `None`.

Attributes: `df_cluster_stats (pd.DataFrame)`: A dataframe exposing cluster composition statistics (cluster index, distance to centerpoint statistics, class balance).

Method `get_cluster()`

Description: Retrive the desired cluster by providing its index. The latter can be found by inspecting `df_cluster_stats`.

Parameters: `cluster_idx (int)`: Index of desired cluster (found by inspecting `df_cluster_stats`).

Returns: `(gpd.GeoDataFrame)` The desired cluster represented as a geodataframe.

Method `fit()`

Description: Partition a collection of geolocation transactions in NN clusters and store results in internal state.

Parameters:

1. `gdf_transactions (gpd.GeoDataFrame)`: Transaction collection. This is required to have a binary `label` column as discussed in the class description.
2. `n_nbrs (int)`: Number of nearest neighbours.
3. `crs (str)`: Local projected coordinate system for geodesic distance computation.
4. `gdf_centers (Optional[gpd.GeoDataFrame])`: Transactions to use as cluster centers. Defaults to transactions in the positive class.

Returns: `None`

```
# Initialize the cluster identifier.

cluster_identifier = ClusterIdentifier()

# Train it on a gdf (gdf_transactions) containing a collection of geolocation
transactions. Center the clusters at transactions carrying positive
pseudolabels.

cluster_identifier.fit(
    gdf_transactions=gdf_transactions,
    n_nbrs=100,
    crs='EPSG:4326',
    gdf_centers=gdf_transactions[gdf_transactions['label'] == 1]
    )

# Display the identified clusters so as to decide which cluster to work with.

display(cluster_identifier.df_cluster_stats)

# Determine the relevant cluster index by inspecting df_cluster_stats.

cluster_idx = 10

# Access the desired cluster.

gdf_cluster = cluster_identifier.get_cluster(cluster_idx=cluster_idx)
```

Listing 1: `ClusterIdentifier` Demo

3.2 models

This is the core directory of the project. It contains all the functionality related to our models.

The classes provided are:

1. `tools`: auxiliary tools used to train and operate our NN models:
   (a) `tokenization`:
      i. `Tokenizer`,
   (b) Training
      i. `early_stopping`:
         A. `EarlyStopperBase`,
         B. `StandardStopper`,
         C. `MonotoneStopper`,
      ii. `TrainerBase`,
2. `layers`:
   (a) `BatchSwapNoise`,
   (b) `ContinuousFeatureNormalizer`,
   (c) `LinBnDrop`,
   (d) `MaskedCCELoss`,
   (e) `ReconstructionLoss`,
   (f) `ShiftedSigmoid`,
   (g) `TransformerEncoder`,
3. `text_encoder`:
   (a) `TextEncoder`,
4. `autoencoder`:
   (a) `TransactionAnomalyDetector`,

3.2.1 tools

Within transaction_anomaly_detection.models.tools live auxiliary tools supporting the operation and training of our neural network models. These include a Tokenizer class and tools required for neural network training: the TrainerBase abstract base class and different kinds of early stopping tools for regularization.

3.2.1.1 Class Tokenizer

Description: The Tokenizer offers a simple API to encode a given collection of tokens—or sequences thereof—into integers. Once token sequences have been encoded, they can be decoded back to their original form. By fixing a body of text and letting the tokens be the alphabet, the Tokenizer acts as a character-level tokenization tool for simple NLP tasks.

Constructor Parameters:

1. ls_tokens (List[str]): The set of regular tokens handled by the Tokenizer (e.g. the alphabet—iuncluding punctuation characters—in a character-level NLP application). This does not include special tokens for padding, handling unknowns and masking.
2. pad_token (Optional[str]): Special token used for padding. This is needed to increase the size of short sequences to appropriate lengths.
3. unk_token (Optional[str]): Special token representing an element absent from the ls_tokens.
4. mask_token (Optional[str]): Special token used for masking. Regular tokens are replaced with the mask_token to signify that their value has been hidden.

Attributes:

1. vocabulary (Set[str]): The set of all legitimate tokens: both regular and special. All sequences handled by the Tokenizer are valued in this set.
2. pad_token (Optional[str]): See the corresponding constructor parameter.
3. pad_token_encoding (Optional[int]): The integer encoding of the pad_token.
4. unk_token (Optional[str]): See the corresponding constructor parameter.
5. unk_token_encoding (Optional[int]): The integer encoding of the unk_token.
6. mask_token (Optional[str]): See the corresponding constructor parameter.
7. mask_token_encoding (Optional[int]): The integer encoding of the mask_token,
8. regular_token_encodings (Set[int]): The set of integer encodings of all regular tokens in the vocabulary.

Method pad()

Description: Insert a prescribed amount of pad_tokens on the left/ right of the input sequence.

Parameters:

1. sequence (List[str]): The token sequence to be padded.
2. pad_left (Optional[int] = 0): Number of pad_tokens to insert on the left.
3. pad_right (Optional[int] = 0): Number of pad_tokens to insert on the right.

Returns: (List[str]) The padded token sequence.

Method `encode()` Map a token/ sequence of tokens to an integer/ sequence of integers.

Description:

Parameters: `token_or_ls_tokens (str or List[str])`: The token/ token sequence to be encoded.

Returns: `int or List[int]` The integer/ sequence of integers encoding th einput.

Method `decode()`

Description: Map an encoded token/ sequence of encoded tokens to the original token/ token sequence that the input encodes.

Parameters:

1. `encoded_token_or_ls_encoded_tokens (int or List[int])`: The integer sequence to be decoded.

Returns: `str or List[str]`: The decoded token/ token sequence represented by the input.

Method `str_to_ls_tokens()`

Description: Character level string tokenization.

Parameters: `str_input: str`: The string to be tokenized.

Returns: `(List[str])` The corresponding token sequence. This agrees with the list of characters comprising the input, excpet when they are not part of the `Tokenizer's` vocabulary.

Method `ls_tokens_to_str()`

Description: Join a sequence of tokens into a string, separating adjacent tokens by space characters.

Parameters: `ls_tokens (List[str])`: The token sequence to be joined

Returns: `(str)` The joined token sequence.

3.2.1.2 Abstract Class `EarlyStopperBase`

Description: Abstract base class for early stopping tools. `EarlyStopperBase` implements only the core functionality required for early stopping, leaving the particular stopping criteria to be implemented in subclasses.

Constructor Parameters: `max_n_epochs (int)`: Maximum number of training epochs.

Attributes:

1. `max_n_epochs (int)`: See the corresponding constructor parameter.
2. `n_epochs_ellapsed (int)`: Number of training epochs completed thus far.
3. `stop (bool)`: `True` iff the stopping condition has been satisfied.
4. `best_epoch (int)`: The number of the epoch when the monitored metric ahchieved its best value thus far.
5. `best_metric (int)`: Best value achieved for the monitored metric.
6. `best_model (nn.Module)`: The model state corresponding to the best metric.

Method `update()`

Description: Update the `EarlyStopper's` internal state after changes to the monitored metric and model state. In practice, this method is to be called once at the end of each epoch.

Parameters:

1. `metric (float)`: The latest value of the monitored metric.
2. `model (nn.Module)`: The latest model state.

Returns: `None`

Abstract Method `update_stop_state()`

Description: Decide if the stopping condition has been met and update internal state accordingly.

Parameters: `None`

Returns: `None`

Abstract Class Method `stopping_condition_met()`

Description: Decide if the stopping condition has been met and return the answer as a `bool`. The core logic is to be implemented in subclasses. The only condition implemented here sets an upper bound on the number of epochs.

Parameters:

1. `n_epochs_ellapsed (int)`: Number of training epochs completed thus far.
2. `max_n_epochs(int)`: Maximum number of training epochs.

Returns: `(bool)`: `True` iff the stopping condition has been satisfied.

3.2.1.3 Class `StandardStopper(EarlyStopperBase)`

Description: Standard early stopper demanding a minimum improvement on the best value of the monitored metric over a bounded number of epochs.

Constructor parameters:

1. `patience (int)`: The maximum number of epochs without sufficient improvement in the monitored metric.
2. `delta_threshold (float)`: The bound controlling when an improvement in the monitored metric is considered sufficient.
3. `max_n_epochs (int)`: Maximum number of training epochs.

Attributes:

1. `patience (int)`: See the corresponding constructor parameter.
2. `delta_threshold (float)`: See the corresponding constructor parameter.

Method `update()`

Description: Update the `StandardStopper's` internal state: the best value of the monitored metric, the corrsponding model state and the number of epochs ellapsed since the most recent sufficient improvement in the monitored metric.

Parameters:

1. `metric (float)`: The latest value of the monitored metric.
2. `model (nn.Module)`: The latest model state.

Returns: `None`

Class Method `stopping_condition_met()`

Description: Decide if the stopping condition has been met and return the answer as a `bool`. The core logic is to be implemented in subclasses. The only condition implemented here sets an upper bound on the number of epochs.

Parameters:

1. `max_n_epochs (int)`: Maximum number of training epochs.
2. `patience (int)`: The aximum number of training epochs without sufficient improvement in the monitored metric.
3. `n_epochs_ellapsed (int)`: Total number of training epochs completed thus far.
4. `n_epochs_without_improvement (int)`: Number of training epochs completed since latest sufficient improvement in the monitored metric.

Returns: `(bool)`: `True` iff the stopping condition has been satisfied.

3.2.1.4 Class `MonotoneStopper(EarlyStopperBase)`

Description: Early stopper conditioning on the sequence of differences (deltas) between successive values of the monitored metric. The stopping condition is satisfied iff a configurable number of successive deltas do not exceed a minimum improvement.

Constructor Parameters:

1. `patience (int)`: The maximum number of successive epochs without any metric delta exhibiting a sufficient improvement.
2. `delta_threshold (float)`: The bound controlling when a metric improvement is considered sufficient.
3. `max_n_epochs (int)`: Maximum number of training epochs.

Attributes:

1. `patience (int)`: The maximum number of successive epochs without any metric delta exhibiting a sufficient improvement.
2. `delta_threshold (float)`: The bound controlling when a metric improvement is considered sufficient.

Method `update()`

Description: Update the `MonotoneStopper's` internal state: the best value of the monitored metric, the corrsponding model state and the stopper's historic metric record.

Parameters:

1. `metric (float)`: The latest value of the monitored metric.
2. `model (nn.Module)`: The latest model state.

Returns: `None`

Class Method `stopping_condition_met()`

Description: Decide if the stopping condition has been met and return the answer as a `bool`. The core logic is to be implemented in subclasses. The only condition implemented here sets an upper bound on the number of epochs.

Parameters:

1. `max_n_epochs (int)`: Maximum number of training epochs.
2. `patience (int)`: The maximum number of successive epochs without any metric delta exhibiting a sufficient improvement.
3. `delta_threshold`: The bound controlling when a metric improvement is considered sufficient.
4. `n_epochs_ellapsed (int)`: Total number of training epochs completed thus far.
5. `ls_historic_metrics (List[int])`: Historic record of metrics at the end of each epoch.

Returns: `(bool)`: `True` iff the stopping condition has been satisfied.

3.2.1.5 Abstract Class `TrainerBase`

Description: Base class for neural network training. Core training functionality like splitting, batching and publishing progress updates is implemented here. The training loop itself is implemented in subclasses.

Abstract Class Method `train()`

Description: Training loop.

Parameters:

1. `model (nn.Module)`: The model to be trained.

Returns: (Tuple[nn.Module, Dict[str, pd.Series]]) Tuple containing the trained model and a dictionary of pd.Series objects representing the train and val loss evolutions respectively.

Static Method `get_progress_bar_desc()`

after each

Format and return the text to be printed on the training progress bar (`tqdm`).

Description Parameters:

1. `current_epoch (int)`: The number of the current epoch.
2. `previous_epoch_val_loss (float)` The mean validation loss achieved at the end of the previous epoch.
3. `min_loss (float)` The minimum validation loss achieved thus far.
4. `best_epoch (float)` The epoch number corresponding to the minimum vlaidation loss.

Returns: `(str)`: The progress bar descriptor.

Static Method `get_loss_evolution_update()`

Description: Format and return a message informing the user about the training and validation loss achieved during a given epoch.

Parameters:

1. `epoch (float)`: The epoch number.
2. `train_loss (float)` The training loss achieved during the relevant epoch.
3. `val_loss (float)` The validation loss achieved during the relevant epoch.

Returns: `(str)`: Message providing an update on the training and validation loss values.

Static Method `get_train_recap()`

Description: Format and return a message recapping the outcome of the training procedure. Information provided includes the best epoch and the corresponding validation loss value.

Parameters:

1. `best_epoch (float)`: The number of the epoch with the minimum validation loss.
2. `min_val_loss (float)`: The minimum validation loss achieved during training.

Returns: `(str)`: Training recap message.

Class Method `get_batch_generator()`

Description: Return a generator object yielding training batches.

Parameters:

1. `t_dataset (torch.tensor)`: The full dataset represented as a tensor. The 0th dimension indexes training examples.
2. `sz_batch (int)`: The size of minibatches partitioning the dataset.

Returns: `(Generator[torch.tensor])` Generator yielding minibatches of training examples.

Static Method `shuffle_dataset()`

Description: Randomly shuffle the examples within a dataset.

Parameters: `t_dataset (torch.tensor)`: The dataset to be shuffled.

Returns: `(torch.tensor)` The suffled dataset.

Static Method `split_dataset()`

Description: Partition a dataset in two pieces using prescribed proportions.

Parameters:

1. `t_dataset (torch.tensor)`: The dataset to be partitioned.
2. `val_ratio (float)`: The proportion of records in the second piece.

Returns: `Tuple[torch.tensor, torch.tensor]` The two pieces partitioning the dataset.

Static Method `get_n_batches()`

Description: Get the number of batches obtained by partitioning a dataset of prescribed size into a minibatches of prescribed size.

Parameters:

1. `n_records (int)`: The dataset size.
2. `sz_batch (int)`: The minibatch size.

Returns: `(int)`: The number of minibatches.

3.2.2 `layers`

Within `transaction_anomaly_detection.models.layers` live all the building blocks for our NN models.

3.2.2.1 Class BatchSwapNoise

Description: Swap noise layer augmenting training batches by resampling along the columns.

Constructor Parameters: `swap_rate (float)`: the probability that a given entry of a training batch is replaced with a random sample from the same column.

Attributes: `swap_rate (float)`: See the corresponding contructor parameter.

Method `forward()`

Description: Forward pass.

Parameters: `x (torch.tensor)`: Input training batch (shape = (B, C)).

Returns: `(torch.tensor)`: Augmented training batch (shape = (B, C)).

3.2.2.2 Class ContinuousFeatureNormalizer

Description: Preprocessing layer for continuous features. Population statistics are passed on initialization. The forward pas normalizes (or denormalizes) its input.

Constructor Parameters:

1. `means (torch.tensor)`: The continuous feature means (shape = `(n_con_features)`).
2. `stds (torch.tensor)`: The continuous feature standard deviations (shape = `(n_con_features)`).

Attributes:

1. `means (nn.Parameter)`: See the corresponding contructor parameter.
2. `stds (nn.Parameter)`: See the corresponding contructor parameter.

Method `forward()`

Description: Forward pass.

Parameters:

1. `x (torch.tensor)` Continuous feature input (shape = (B, C) or (C)).
2. `denormalize (Optional[bool] = False)` If `False`, normalize the input. If `True`, denormalize the input.

Returns: `(torch.tensor)` The normalized (or denomralized) input (shape = (B, C) or (C)).

3.2.2.3 Class `LinBnDrop`

Description: Standard linear layer followed by batch normalization and dropout.

Constructor Parameters:

1. `dim_in (int)`: The dimensionality of the input.
2. `dim_out (int)`: The dimensionality of the output.
3. `activation (nn.Module)`: The nonlinear activation function.
4. `bn (bool)`: Apply batch normalization iff this flag is set to `True`.
5. `dropout_rate (float)`: The dropout rate: probability to kill a given neuron during a training pass.

Attributes: `None`

Method `forward()`

Description: Forward pass.

Parameters: `x (torch.tensor)`: Input tensor (shape = (B, `dim_in`)).

Returns: `(torch.tensor)` Output tensor (shape = (B, `dim_out`)).

3.2.2.4 Class `MaskedCCELoss`

Description: Masked categorical cross entorpy (CCE) loss for masked language modeling (MLM) as in [3]. The layer applies standard CCE to entries selcted using a binary mask.

Constructor Parameters: `None`.

Attributes: `None`.

Method `forward()`

Description: Forward pass.

Parameters:

1. `t_logits (torch.tensor)`: The logits encoding the token distribution at each position (shape = (B, T, `n_tokens`)),
2. `t_targets (torch.tensor)`: The unmasked token sequence—prescribing the target token at each position (shape = (B, T)).
3. `t_mask (torch.tensor)`: Binary mask controlling which positions are ignored (iff $\texttt{t\_mask}_{ij} = 0$) in the CCE loss computation (shape = (B,T)).
4. `reduction (Optional[str] = None)`: Categorical input controlling the reduction process.

   When `reduction = None`, the output is a tensor of shape (B, T): unmasked entries are assigned the corresponding negative log-likelihoods while masked ones are set to 0.

   When the argument is used, the return value is a scalar (tensor of null shape). Possible values include `sum` (sum over all unmasked entries) and `mean` (average over all unmasked entries).

Returns: `(torch.tensor)` The masked CCE loss value (shape = (B, T) if `reduction` = None else ()).

3.2.2.5 Class `ReconstructionLoss`

Description: Equally weighted reconstruction loss implementation for continuous and categorical features.

Constructor Parameters: `None`.

Attributes: `None`.

Method `forward()`

Description: Forward pass.

Parameters:

1. `ls_cat_logits (Optional[List[torch.tensor]])`: List of tensors encoding the multinoulli distribution (in log-probabilities) of each categorical feature. Entries are tensors of shape (B, `n_categories`).

2. `ls_cat_targets (Optional[List[torch.tensor]])`: List of tensors prescribing the target category of each categorical feature. Entries are tensors of shape (B).
3. `t_con_predictions (torch.tensor)`: Tensor encoding the predicted continuous feature values (shape = (B, `n_con_features`))
4. `t_con_targets (torch.tensor)` Tensor prescribing the target value for each continuous feature (shape = (B, `n_con_features`)).
5. `batch_reduction (Optional[str] = None)` Categorical input controlling the batch-reduction process.

   When `batch_reduction = None`, the output is a tensor of shape (B): each entry corresponds to the mean reconstruction loss of a record in the batch. The categorical feature CCE loss vales and the continuous feature MSE loss values are averaged with equal weights.

   When the argument is used, the return value is a scalar (tensor of null shape). Possible values include `sum` (sum over records in the batch) and `mean` (average over records in the batch).

Returns: `(torch.tensor)` The reconstruction loss value (shape = (B) if `batch_reduction` = None else ()).

3.2.2.6 Class `ShiftedSigmoid`

Description: Parallel sigmoid functions with configurable horizontal asymptotes for each component:

$$\texttt{ShiftedSigmoid}\,(x_i)_{i=1}^{n} = \left(\texttt{min\_vals}_i + (\texttt{max\_vals}_i - \texttt{min\_vals}_i)\,\sigma(x_i)\right)_{i=1}^{n},$$

where σ denotes the standard sigmoid (logistic) function.

Attributes:

1. `min_vals (nn.Parameter)`: The lower (`min`) horizontal asymptotes (shape = (`n_con_features`)).
2. `max_vals (nn.Parameter)`: The upper (`max`) horizontal asymptotes (shape = (`n_con_features`)).

Method `forward()`

Description: Forward pass.

Parameters: `x (torch.tensor)` The input to the parallel sigmoid functions (shape = (B, `n_con_features`)).

Returns: `(torch.tensor)` The output of the parallel sigmoid functions (shape = (B, `n_con_features`)).

3.2.2.7 Class `TransformerEncoder`

Description: Standard transformer encoder layer as in [8].

Note: our implementations uses the *pre-norm* convention. We apply layer normalization [2] at the start of each multi-head attention block.

Constructor Parameters:

1. `max_len (int)`: The expected number of tokens (frequently denoted `T`).
2. `d_model (int)`: The channel/ token embedding dimension (frequently denoted `C`).

3. `n_encoder_layers (int)`: The number of stacked multi-head attention blocks comprising the encoder layer.
4. `n_parallel_heads_per_layer (int)`: The number of parallel attention heads inside a multi-head attention block. This needs to divide `d_model`.
5. `dim_feedforward (int)`: The hidden layer size in the `MLP` component at the end of each multi-head attention block.
6. `activation (nn.Module)`: The nonlinear activation function.
7. `layer_norm_eps (Optional[float] = 10^-5)`: The ϵ parameter of layer normalization [2]: a fixed positive constant added to the activation variance.
8. `dropout_rate (Optional[float] = 0)`: The dropout rate [7]: the probability of killing a given neuron during a forward pass.

Attributes: `None`.

Method `forward()`

Description: Forward pass.

Parameters: `t_embedded_tokens (torch.tensor)`: A batch of embedded tokens (shape = (B, T, C)).

Returns: `(torch.tensor)`: The encoded input tokens (shape = (B, T, C)).

3.2.3 Text Encoder

Within `transaction_anomaly_detection.models.text_encoder` lives the implementation of the BERT-based [3] text encoder used for representation encodings.

3.2.3.1 Class `TextEncoder`

Description: BERT implementation designed to encode representations of numeric data.

Constructor Parameters:

1. `ls_standard_tokens (List[str])`: The set of regular tokens understood by the model. This does not include special tokens used for padding, handling unknown tokens and masking.
2. `max_n_standard_tokens (int)`: The expected number of tokens (frequently denoted `T`).
3. `d_model (int)`: The channel/ token embedding dimension (frequently denoted `C`).
4. `n_parallel_heads_per_layer (int)`: The number of parallel attention heads inside a multi-head attention block. This is required to divide `d_model`.
5. `dim_feedforward (int)`: The hidden layer size in the `MLP` component at the end of each multi-head attention block.
6. `n_encoder_layers (int)`: The number of stacked multi-head attention blocks comprising the encoder layer.
7. `activation (nn.Module)`: The nonlinear activation function.
8. `layer_norm_eps (Optional[float] = 10^-5)`: The ϵ parameter of layer normalization [2]: a fixed positive constant added to the activation variance.
9. `dropout_rate (Optional[float] = 0)`: The dropout rate [7]: the probability of killing a given neuron during a forward pass.

Attributes:

1. `vocabulary (List[str])`: The set of regular and special tokens understood by the model. This includes both the regular tokens—passed on initialization—and special tokens used for padding (`pad`), handling unknown tokens (`unk`) and masking (`mask`).
2. `max_n_standard_tokens (int)`: See the corresponding constructor parameter.
3. `d_model (int)`: See the corresponding constructor parameter.
4. `n_encoder_layers (int)`: See the corresponding constructor parameter.
5. `n_parallel_heads_per_layer (int)`: See the corresponding constructor parameter.
6. `dim_feedforward (int)`: See the corresponding constructor parameter.
7. `activation (nn.Module)`: See the corresponding constructor parameter.
8. `layer_norm_eps (float)`: See the corresponding constructor parameter.
9. `dropout_rate (float)`: See the corresponding constructor parameter.
10. `bert_encoder_architecture (str)`: String representation of the BERT model architecture.

Method `get_n_params()`

Description: Count the number of trainable paramters.

Parameters: `None`

Returns: `(int)`: The number of trainable parameters.

Method `fit()`

Description: `Adam`-based implementation of MLM training [3]. Fifteen percent of tokens are chosen at random. Ten percent of these are randomly selected for resampling. Eighty percent are randomly selected for masking. The remaining ten percent (of the original fifteen percent) are left unaltered. The model is trained for sentence completion by unmasking the masked tokens.

Parameters:

1. `corpus (List[str])`: The training data. Each entry of `corpus` is a text chunk to be encoded.
2. `val_ratio (float)`: The proportion of training data randomly split off at the start of each epoch to be used for vlaidation.
3. `sz_batch (int)`: The training minibatch size.
4. `learning_rate (float)`: `Adam` learning rate.
5. `patience (int)`: Early stopping hyperparameter. The maximum number of epochs to tolerate a non-satisfactory reduction in the validation loss.
6. `loss_delta_threshold (float)`: Early stopping hyperparameter. The bound determining when a given reduction in the validation loss is considered satisfactory.
7. `max_n_epochs (Optional[int OR float] = np.nan)`: Early stopping hyperparameter. The absolute upper bound on the number of training epochs.
8. `verbose (Optional[bool] = False)`: If `True`, publish training and validation loss updates to `stdout` at the end of each epoch.

Returns: `(Dict[str, pd.Series])` The training and validation loss evolutions.

Method `encode()`

Description: Encode the input text chunk(s).

Parameters: `input_text (str OR List[str])`: The text to encode.

Returns: `(torch.tensor)`: The encoded text (shape = (B, T, C)).

Method `complete()`

Description: Complete the input list of tokens.

Parameters:

1. `ls_tokens (List[str])`: List of tokens to complete. The positions that will be filled in are precisely those containing the token `mask` (which is always part of the `vocabulary` attribute).

Returns: `(List[str])`: The completed token sequence.

Method `export()`

Description: Export the model to long-term memory.

Parameters:

1. `path_export_dir (Path)`: Path pointing to the model directory (folder).
2. `model_name (str)`: Model name.

Returns: `None`

Method `load_exported_model()`

Description: Load exported model from long-term memory.

Parameters:

1. `path_export_dir (Path)`: Path pointing to the model directory (folder).
2. `model_name (str)`: Model name.

Returns: `(TextEncoder)`

3.2.4 Autoencoder

Within `transaction_anomaly_detection.models.autoencoder` lives the implementation of the centerpiece of this project: the autoencoder model used for anomaly detection.

3.2.4.1 Class `TransactionAnomalyDetector`

Description:

Constructor Parameters:

1. `dict_cat_feature_to_ls_categories_n_embd (Dict[str, Tuple[List[str], int]])`: Dictionary prescribing the unique categories and embedding dimension for each categorical feature.
2. `ls_con_features (List[str])`: The continuous feature names.
3. `encoder_layer_szs (List[int])`: The layer sizes for the standard encoder component of the model.
4. `ae_activation (nn.Module)`: The nonlinear activation function.
5. `dropout_rate (float)`: The dropout rate [7]: the probability of killing a given neuron during a forward pass.
6. `batchswap_noise_rate (float)`: The batchswap noise rate: the probability to resample a given training batch element from within its column.

Attributes:

1. dict_cat_feature_to_ls_categories_n_embd (Dict[str, Tuple[List[str], int]]): See the corresponding constructor parameter.
2. ls_con_features List[str]: See the corresponding constructor parameter.
3. ae_activation (nn.Module): See the corresponding constructor parameter.
4. encoder_layer_szs (List[int]): See the corresponding constructor parameter.
5. dropout_rate (float): See the corresponding constructor parameter.
6. batchswap_noise_rate (float): See the corresponding constructor parameter.
7. has_cat (bool): True iff the model has been trained on data with at least one categorical feature.
8. ls_cat_features List[str]: The categrocial feature names.
9. dict_cat_feature_to_ls_categories (Dict[str, List[str]]): Dictionary mapping categorical features to their unique categories.
10. dict_cat_feature_to_n_embd (Dict[str, int]): Dictionary mapping categorical features to their embedding dimensions.
11. has_con (bool): True iff the model has been trained on data with at least one continuous feature.
12. df_con_stats (pd.DataFrame): Dataframe exhibiting continuous feature summary statistics.
13. autoencoder_architecture (str): String representation of the model architecture.
14. reconstruction_loss_threshold (float): The reconstruction-loss cut-off for anomaly detection.

Method get_n_params()

Description: Count the number of trainable paramters.

Parameters: None

Returns: (int): The number of trainable parameters.

Method fit()

Description: Adam-based training loop.

Parameters:

1. df_dataset (pd.DataFrame): Dataframe containing the training data. Each row represents a transaction. The columns are required to be a (not necessarily strict) superset of the union of ls_cat_features and ls_con_features.
2. contamination (float): Prior controlling the expected proportion of anomalous records. This is used for quantile-based reconstruction-loss threshold tuning.
3. val_ratio (float): The proportion of training data randomly split off at the start of each epoch to be used for vlaidation.
4. sz_batch (int): The training minibatch size.
5. learning_rate (float): Adam learning rate.
6. patience (int): Early stopping hyperparameter. The maximum number of epochs to tolerate a non-satisfactory reduction in the validation loss.
7. loss_delta_threshold (float): Early stopping hyperparameter. The bound determining when a given reduction in the validation loss is considered satisfactory.
8. max_n_epochs (Optional[int OR float] = np.nan): Early stopping hyperparameter. The absolute upper bound on the number of training epochs.
9. verbose (Optional[bool] = False): If True, publish training and validation loss updates to stdout at the end of each epoch.

Returns: (Dict[str, pd.Series]) The training and validation loss evolutions.

Method `fit_reconstruction_loss_threshold()`

Description: Set the `reconstruction_loss_threshold`—used for anomaly detection—by passing the expected `contamination` proportion.

Parameters:

1. `df_dataset (pd.DataFrame)`: Dataframe containing the training data. Each row represents a transaction. The columns are required to be a (not necessarily strict) superset of the union of `ls_cat_features` and `ls_con_features`.
2. `contamination (float)`: Prior controlling the expected proportion of anomalous records. This is used for quantile-based reconstruction-loss threshold tuning.

Returns: `None`

Method `detect_anomalies()`

Description: Select the records with loss exceeding the `reconstruction_loss_threshold`

Parameters: `input_data (pd.Series OR pd.DataFrame)`: The records among which to identify anomalies.

Returns: `(Optional[pd.DataFrame])`: The anomalous records ranked by reconstruction loss. The return typ is `None` if no anomalies are found.

Method `reconstruct()`

Description: Reconstruct the input data by passing it through the network.

Parameters: `input_data (pd.Series OR pd.DataFrame)`: Data to reconstruct.

Returns: `(pd.DataFrame)`: Reconstructed data.

Method `encode()`

Description: Encode the input data by passing it through the network and extracting its latent space representation.

Parameters: `input_data (pd.Series OR pd.DataFrame)`: Data to encode.

Returns: `(torch.tensor)`: Encoded data.

Method `compute_mean_reconstruction_loss()`

Description: Evaluate the reconstruction loss, first averaged over features and subsequently over records.

Parameters: `input_data (pd.Series OR pd.DataFrame)`: The data to evaluate the loss on.

Returns: `(float)`: Mean reconstruction loss.

Method `compute_reconstruction_loss_by_feature()`

Description: Evaluate the reconstruction loss for each feature—averaged over records.

Parameters: `input_data (pd.Series OR pd.DataFrame)`: The data to evaluate the loss on.

Returns: `(pd.Series)` The features and their corresponding loss averages, ranked in descending order.

Method `compute_reconstruction_loss_by_record()`

Description: Evaluate the reconstruction loss for each record.

Parameters: 1. `input_data (pd.Series OR pd.DataFrame)`: The data to evaluate the loss on.
2. `average_over_features (Optional[bool] = False)`: If `True`, average over features—obtaining a scalar for each record. If `False`, obtain the unreduced reconstruction loss matrix—with entries corresponding to record/ feature pairs.

Returns: `(pd.Series OR pd.DataFrame)` The reconstruction loss of each record. The output type depends on `average_over_features`. Series are returned iff the latter is `True`.

Method `export()`

Description: Export the model to long-term memory.

Parameters:

1. `path_export_dir (Path)`: Path pointing to the model directory (folder).
2. `model_name (str)`: Model name.

Returns: `None`

Method `load_exported_model()`

Description: Load exported model from long-term memory.

Parameters:

1. `path_export_dir (Path)`: Path pointing to the model directory (folder).
2. `model_name (str)`: Model name.

Returns: `(TransactionAnomalyDetector)`

...

4 Process Specifications

The present section is intended to provide the necessary guidelines and best practice heuristics for successful use on production data. We begin by providing our view on hyperparameter tuning and subsequently discuss our approach to training and geographical segmentation. We make sure to provide multiple examples to illustrate the impact of our choices. We conclude the section by taking a deep-dive on the runner script for our daily production job.

4.1 Hyperparameters: Meaning & Selection Guidelines

4.1.1 `TextEncoder`

The hyperparameters to be specified when initializing a `TextEncoder` are:

- `ls_standard_tokens (List[str])`
- `max_n_standard_tokens (int)`
- `d_model (int)`
- `n_parallel_heads_per_layer (int)`
- `dim_feedforward (int)`
- `n_encoder_layers (int)`
- `activation (nn.Module)`
- `layer_norm_eps (float)`
- `dropout_rate (float)`

4.1.1.1 `ls_standard_tokens`

This hyperparameter controls the standard tokens understood by the `TextEncoder`. This does not inculde special tokens used for padding, handling unknown tokens and masking.

When the use-case aims at obtaining numerical data representation encodings, we have:

$$\texttt{ls\_standard\_tokens} = [0, 1, ..., 9, .] \,.$$

4.1.1.2 `max_n_standard_tokens`

This hyperparameter controls the upper-bound (inclusive) on the token sequence length that the model can handle.

When the use-case aims at obtaining numerical data representation encodings, we typically select:

$$\texttt{max\_n\_standard\_tokens} = \max\left(10, \max\left\{\texttt{len}(x) \mid x \in \texttt{corpus}\right\}\right).$$

The reason why the inner maximum is insufficient is that we typically pretrain and then evaluate on incoming data for at least a few days. When incoming data contains longer examples than the ones the model was trained on, these examples will be truncated. This inevitably results in some information loss. The outter maximum is intended to minimize this effect.

4.1.1.3 `d_model, n_parallel_heads_per_layer and dim_feedforward`

These hyperparameters control the shape of the multi-head attention layers leveraged by the BERT model [8].

The channel dimension is prescribed by $\texttt{d\_model} \in \mathbb{N}$.

A single multi-head attention layer consists of $\texttt{n\_parallel\_heads\_per\_layer} \in \mathbb{N}$ parallel attention heads, each receiving a tensor of shape:

$$\texttt{head\_input\_shape} = (\texttt{max\_n\_standard\_tokens}, \texttt{d\_model}) \,,$$

and producing a tensor of shape:

$$\texttt{head\_output\_shape} = \left(\texttt{n\_parallel\_heads\_per\_layer}, \frac{\texttt{d\_model}}{\texttt{n\_parallel\_heads\_per\_layer}}\right).$$

Note that `n_parallel_heads_per_layer` ought to divide `d_model`.

The parallel heads' outputs are concatenated and their channel-dimension slices are passed through an MLP with shape:

$$\texttt{MLP\_layer\_sizes} = [\texttt{dim\_feedforward}, \texttt{d\_model}] \,.$$

This results in a final output of shape:

$$\texttt{output\_shape} = (\texttt{n\_parallel\_heads\_per\_layer}, \texttt{d\_model}) \,.$$

Following [8], we set:

$$\texttt{dim\_feedforward} = 4 \cdot \texttt{d\_model}. \tag{4.1}$$

Since the applications we have in mind concern situations with high amounts of random noise (representation encodings of numerical data rather than real NLP), overparameterization can be problematic. We need to avoid overfitting to the noise. We have found success in using small models with:

$$\begin{aligned} \texttt{d\_model} &= 8, \\ \texttt{dim\_feedforward} &= 32, \\ \texttt{n\_parallel\_heads\_per\_layer} &= 2. \end{aligned} \tag{4.2}$$

4.1.1.4 `n_encoder_layers`

This hyperparameter specifies, the number of stacked multi-head attention layers to form the BERT transformer encoder.

Since the applications we have in mind concern situations with high amounts of random noise (representation encodings of numerical data rather than real NLP), overparameterization is problematic. We have found success in using 3 stacked multi-head attention layers:

$$\texttt{n\_encoder\_layers} = 3.$$

4.1.1.5 `activation`

This hyperparameter controls the activation following the hidden MLP layer inside each multi-head attention unit. We have mostly been working with `nn.GELU()`, a customary choice in this context.

4.1.1.6 `layer_norm_eps`

Multi-head attention typically includes layer normalization [2] prior to the scaled-dot product computation. The present hyperparameter controls the normalization unit by adding a fixed positive constant $\epsilon > 0$ to the activation variance [2]. This is done to avoid division by zero. Small values of `layer_norm_eps` result in aggressive normalization, while larger values result relax the normalization procedure. We have stuck to the default of:

$$\texttt{layer\_norm\_eps} = 10^{-5}.$$

4.1.1.7 `dropout_rate`

This hyperparameter controls the degree of dropout regularization [7]: the proportion of random neurons set to 0 in each pass. Since there is already a lot of noise in the data and we are using a low number of parameters, we have found that strong dropout obstructs training. We have had success with a light `dropout_rate` of:

$$\texttt{dropout\_rate} = 0.1.$$

4.1.2 `TransactionAnomalyDetector`

The hyperparameters to be specified when initializing a `TransactionAnomalyDetector` are:

- `dict_cat_feature_to_ls_categories_n_embd ( Dict[str, Tuple[List[str], int]])`
- `ls_con_features (List[str])`
- `encoder_layer_szs (List[int])`
- `ae_activation (nn.Module)`
- `dropout_rate (float)`
- `batchswap_noise_rate (float)`

4.1.2.1 Input Feature Specifiers

The first two hyperparameters (`dict_cat_feature_to_ls_categories_n_embd` and `ls_con_features`) specify the structure of the input received by the model. The former specifies the categorical features—and how to encode their values in a suitable format. The latter specifies the continuous features. The training and evaluation data is passed as pandas objects, possibly including more fields than those leveraged by the model. The selections made here tell the model which columns to look for and how to treat them (continuous/ categorical).

The dictionary `dict_cat_feature_to_ls_categories_n_embd` should be formatted as follows. Keys should correspond to categorical feautre names. Values should be tuples specifying the unique categories and the embedding dimensions—in this order—:

$$\texttt{dict\_cat\_feature\_to\_ls\_categories\_n\_embd} = \Big\{\texttt{cat\_feature:} \quad \big([cat_1, ..., cat_k],\ \mathrm{d}_{\mathrm{embd}}\big)\Big\}.$$

Here, $cat_1, ..., cat_k$ is a list of strings forming the codomain of `cat_feature` and $\mathrm{d}_{\mathrm{embd}} \in \mathbb{N}_{>0}$ is the embedding dimension.

Once `dict_cat_feature_to_ls_categories_n_embd` and `ls_con_features` have been selected, the dimensionality of the input layer is determined:

$$\mathrm{d}_{\mathrm{input}} = \mathrm{len}(\texttt{ls\_con\_features}) + \sum_{f \in \texttt{categorical\_features}} \mathrm{d}^f_{\mathrm{embd}} \tag{4.3}$$

4.1.2.2 `ls_encoder_layer_szs`

This hyperparameter specifies the shape of the standard autoencoder component. Only the encoder layer sizes need to be passed. The decoder layer sizes are detrmined by transversing `ls_encoder_layer_szs` in reverse (hourglass autoencoder architecture).

The user is required to pass in a list of integers:

$$\texttt{encoder\_layer\_szs} = [s_0,\ ...,\ s_n].$$

These have the following significance.

The integer $n \in \mathbb{N}$ specifies the encoder depth: the number of stacked linear layers dedicated to compressing the input.

The first entry s_0 specifies the external dimension of the stadnard autoencoder. Note that this is typically greater than the dimensionality of the raw input (4.3):

$$\mathrm{d}_{\mathrm{ae\_external}} < s_0 \sim 10 \cdot \mathrm{d}_{\mathrm{input}}.$$

The final entry s_n specifies the dimensionality of the latent space: the size of the encoded data. The ratio:

$$r_{\texttt{compression}} = \frac{s_n}{s_0} \in (0, 1)$$

is known as the *compression ratio* of the autoencoder. It quantifies the degree of the bottleneck introduced by the compression. The lower the value the larger the bottleneck. Evidently, lowering the ratio reduces the capacity of the model. Lower capacity results in learning more abstract latent patterns. Tuning the compression ratio can heavily impact performance. Higher values result in high resolution paterns being detected, but this can lead to anomalies being flagged due to unimportant details. Lower values target more meaningful anomaly attributions, but run into the risk of bein unable to capture even the crucial abstract patterns in the data. As a rule of thumb we have typically worked with:

$$\begin{aligned} s_n &\sim \frac{d_{\mathrm{input}}}{2}, \\ s_n &\sim \frac{s_1}{20}. \end{aligned} \tag{4.4}$$

The rest of the `encoder_layer_sizes` are to be selected similarly to tuning the layer sizes of an `MLP`. Deeper architectures have higher capacity but are harder to train. Furthermore, steep jumps in the sizes of adjacent layers result in inefficient information propagation. We have found success in starting from the following heuristic: interpolating between the external and latent dimensions in successive powers of 2.

4.1.2.3 `ae_activation`

This hyperparameter specifies the activation used in the `LinBnDrop` layers comprising the standard autoencoder component of the model.

The idea is to pick a simple nonlinear function that renders the network trainable. We have been using `nn.LeakyReLU(0.01)`. The reason lies in `ReLU`'s ability to avoid gradient saturation for high absolute values (as opposed to `tanh`). `ReLU` itself cannot be used to reconstruct features taking on negative values, motivating us to slightly tilt its tail by choosing `nn.LeakyReLU(0.01)`.

4.1.2.4 `dropout_rate`

This hyperparameter controls the degree of dropout regularization: the proportion of random neurons set to 0 in each pass.

Since working with high compression ratios already preventsw overfitting, we have found that light dropout (`dropout_rate` = 0.1) typically works best.

4.1.2.5 `batchswap_noise_rate`

This hyperparameter tunes the intensity of the noise augmenting the input so as to encourage the model to leverage relationships among the columns. Higher values increasingly diminish the impact of the value of feature f in its own reconstruction.

The idea is to push the value high enough to capture intricate intra-column relationships, but not too high so as to render the network untrainable. Through experimentation, we have found that values between 0.1 and 0.2 work best.

4.2 Geospatial Segmentation

A decision that can significantly impact the performance of our model is that of the geographical segment to operate in. The idea here is that restricting the relevant region should cause distributions to lose entropy, thereby bringing us in a situation where *normal* transactions follow rigid structured patterns. It's in this setting that the reality conjectured in the feature volume hypothesis (section 1.2) would shine the brightest. Spoofers, unaware of these rigid patterns, would be bound to make mistakes when masking their signals. This would result in their eventual identification—using our system.

We have not yet concluded on the optimal way to approach this issue. This section is dedicated to outlining our (limited) current understanding.

Our experiments to date have been completed by segmenting at a:

1. country-wide level,
2. state-wide level,
3. nearest enighbour cluster (`n_nbrs` = 100) level.

   The component of our repository responsible for geographical segmentation is the `ClusterIdentifier` class. It's job is to partition a collection of transactions in nearest neighbour clusters. For details on its funcitonality see the corresponding section in the API reference.

Training on a country-wide level is challenging. The entropy in the distributions makes it difficult to learn a meaningful transaction representation. We were not able to get sensible anomaly interpretations over such large geographical segments. By zooming into the state-wide level, this problem was alleviated. All of the results presented in section 1.4 have been obtained by training and evaluating on state-wide datasets. These observations already partially validate our intuition: the signals clarified by restricting the geographical segment. A very direct instance of this phenomenon is our detection of abnormal altitude records in Ontario. Had we operated at a country-wide level, our system might have been unable to detect this anomaly. Since Ontario is flat, this was achieved with ease. It might be possible for the autoencoder to understand the correlation between coordaintes and altitude, thus learning (at least partial) information on Eart's altitude map. While it would be interesting to test that idea, any signal of this sort would certainly be less clear than in the geo-restricted context.

Note that the above example—although exceedingly simple—does provide a model for the type of signals we are looking for. The feature volume hypothesis (section 1.2) postulates the existance of rigid patterns restricting (combinations of) the multitude of technical features we record. To illustrate, provided geo-segment restrictions cause the local access point population to be sufficiently

limited, features related to wifi connection specs might become so rigid that their degrees of freedom reduce to a single digit number. This is in direct analogy to the Ontario altitude distribution being highly concentrated near the $\sim$ 100m. mark.

Nevertheless, are yet to find benefit in restricting to smaller clusters. In the datasets we worked with, we found that once restricting the cluster enough to align with our intuition (`n_nbrs` = 100), the number of unique devices dropped dramatically. This resulted in very limited patterns represented in the data. As a result, the transactions flagged did not come with understandable interpretations. Furthermore, we have found the choice of clustering algorithm and the corresponding hyperparameter selection to cause dramatic variations in the anomaly list. This is to be juxtaposed with the hyperaameter tuning involved in our NN models—where results were relatively stable.

Formalizing a practical evaluation procedure designed to select the optimal segment size/ shape is the most pressing research direction to advance our work. This presents an extra dimension obstructing our evaluation process. Evaluating unknown unknowns detectors is already challenging: we don't know what we don't know. To overcome this, we opted to build up a bank of model outputs and have human verifiers manually review it (work in progress). This intorduced a lot of manual labour: there is a real cost to obtaining the relevant evaluation results. We are not prepared to multiply that effort over the space of possible geogrpahical segmentation choices. A more efficient method is required.

4.3 SoFi Daily Job

This section is dedicated to documenting the daily production job designed for our one month engagement program with SoFi. The task at hand was to provide the client with daily anomaly scores on their NDBS transactions. Consequently, the process is designed to work with NDBS data. Nevertheless it's still illustrative: with minor modifications, it can be made to work with data generated by any GeoComply solution.

The job has been implemented directly on databricks and lives in:

`job_root_dir` = `transaction_anomaly_detection/notebooks/apps/sofi_daily_job`

4.3.1 High-Level View: Runner

Our runner script lives in the notebook:

`task_dir` = `job_root_dir/runner`.

This is the entry-point for the SoFi job. The `runner` notebook should be scheduled to run daily. In it, we implement the run schedule and use it to decide which actions to perform.

We have organized actions into tasks and jobs. Tasks are the quanta of our process: they are meant to capture individual actions composing the pipeline. Jobs are formed by combining various tasks. The runner decides which jobs to call depending on the implementation of the run schedule.

The run schedule is determined by implementing an algorithm to select which job to run (based on the current date) and to pass the `run_date` (date on which the run takes palce) and the `training_date` (date on which the relevant models were trained) as configuration paramters.

This architecture leverages luxuries afforded to us by databricks to assist in debugging. When a run fails, the corresponding job notebook can be called by overriding the `run_date` and `training_date` parameters (which are normally passed on by the `runner`). This initiates an interactive run, remembering internal state and exposing it to the user along with all intermediate outputs. Discovering what went wrong is rendered almost trivial.

4.3.2 Tasks

Tasks are the quanta of our process. There are **four** separate tasks living in:

`task_dir` = `job_root_dir/tasks`.

These are:

1. **Training Data Prepreocessing**: Receive incoming training data and parse it in order to identify continuous and categorical features. Enrich the data with derived features and finally impute it. Export the clean dataset.
2. **Representation Encoder Training**: Use the exported clean data to train BERT `TextEncoder` models.
3. **Anomaly Detector Training**: Enrich the exported clean dataset with representation encodings obtained using the pre-trained BERT models. Export the final training dataset. Use this dataset to train the tabular autoencoder `TransactionAnomalyDetector` model.
4. **Evaluation Data Preprocessing**: Import metadata exported during the Training Data Preprocessing task and consume them in order to apply the same cleaning procedure on incoming data. Export the cleaned dataset.
5. **Evaluation**: Use the pre-trained `TransactionAnomalyDetector` model on the clean evaluation data to compute and export anomaly scores and HML anomaly classifications.

4.3.3 Jobs

Jobs are formed by combining tasks. We have designed **two** jobs living in:

$$\texttt{task\_dir} = \texttt{job\_root\_dir/jobs}.$$

These are:

1. **Training & Evaluation**: Apply all the tasks in sequence: receive new training data, clean it, initializae and train new models, evaluate these models on their own training data and finally export anomaly scores and HML classifications.
2. **Evaluation Only**: Apply only the final two tasks (i.e. *Evaluation Data Preprocessing* and *Evaluation*): Load new data, clean it in alignement with the specs of pre-trained models and use those pre-trained models to compute and export anomaly scores and HML classifications.

So far we have opted to run job 1 (Training & Evaluation) once a week and job 2 (Evaluation Only) on all other days. We have been training on one week's worth of data and evaluating on two day's worth of data at a time.

4.3.4 Configuration

Hyperparameter tuning is achieved by editing a central config file living in:

$$\texttt{task\_dir} = \texttt{job\_root\_dir/config.py}.$$

This is partitioned into six parameter groups: one for setting up the export directory structure and one for each task. We outline their significance below:

1. `dict_config_directories`: Hyperparameter group controlling the export directory structure.
   (a) `delta_table_transaction_data`: Name of the delta table—on unity catalog—containing the parsed (tabulated) transaction data.
   (b) `path_run_archive_root_dir`: DBFS path to root directory for run exports. This contains folders corresponding to different runs and indexed by `run_date`.
   (c) `training_data_preprocessing_subdirectory_name`: Name of subfolder containing exports produced by the training data preprocessing task. This will be created within the subdirectory dedicated to each run.
   (d) `schema_json_name`: Filename of exported json file encoding the schema of a dataset,
   (e) `process_specs_json_name`: Filename of exported json file encoding the cleaning procedure specs,

(f) `clean_data_table_name`: Filename of exported csv containing the clean data produced by the training data preprocessing task,

(g) `text_encoder_export_subdirectory_name`: Name of subfolder containing exports produced by the representation encoder training task. This will be created within the subdirectory dedicated to each run.

(h) `repr_features_json_name`: Filename of exported json file containing a list of the features for whch representation encoders were trained.

(i) `anomaly_detector_training_data_subdirectory_name`: Name of subfolder containing the final autoencoder training data produced during the anomaly detector training task. This will be created within the subdirectory dedicated to each run.

(j) `anomaly_detector_training_data_table_name`: Filename of exported json csv containing the autoencoder training data produced by the anomaly detector training task.

(k) `anomaly_detector_export_subdirectory_name`: Name of subfolder containing the exported anomaly detector model. This will be created within the subdirectory dedicated to each run.

(l) `exported_anomaly_detector_name`: Name of the anomaly detector model trained and exported during the anomaly detector training task.

(m) `evaluation_data_subdirectory_name`: Name of subfolder containing exports produced by the evaluation data preprocessing task. This will be created within the subdirectory dedicated to each run.

(n) `evaluation_data_table_name`: Name of exported csv containing the clean evaluation data produced by the evaluation data preprocessing task.

(o) `results_subdirectory_name`: Name of subfolder containing exports produced by the evaluation task. This will be created within the subdirectory dedicated to each run.

(p) `anomaly_score_table_name`: Name of exported csv containing the computed anomaly scores. Exported during the evaluation task.

(q) `feature_ranking_table_name`: Name of exported csv containing the reconstruction loss feature ranking for each record. Exported during the evaluation task.

2. `dict_config_training_data_preprocessing`: Hyperparameter group controlling the training data preprocessing task.

(a) `training_data_operator_name`: Operator name used to filter training transactions (filter discarded if this is set to `None`).

(b) `n_training_days`: Number of days (including `run_date`) for which to pull training data.

(c) `ls_identifiers`: List of special identifier (key) features (e.g. `gc_transaction`) found in the dataset.

(d) `ls_engine_results`: List of special features found in the dataset that relate to the operation of our rules engine (e.g. `gc_authorized`, and `gc_error`).

(e) `ls_ignore_features`: List of features to ignore.

(f) `missing_value_prop_drop_threshold`: Missing value threshold above which to discard a given column.

(g) `unique_count_lower_numeric`: Lower bound on the unique count of a numeric column for that column to be utilized.

(h) `unique_count_upper_numeric`: Upper bound on the unique count of a numeric column for that column to be treated as categorical. Unique greater than this threshold result to the respective feature being treated as a continuous one.

(i) `unique_count_lower_hashable_object`: Lower bound on the unique count of a non-numeric hashable column for that column to be utilized as a categorical feature.

(j) `unique_count_upper_hashable_object`: Upper bound on the unique count of a non-numeric hashable column for that column to be utilized as a categorical feature.

(k) `ls_raw_nonhashables`: List of raw nonhashable (e.g. list-valued) features to consider.

(l) `std_low_con`: Lower bound on the standard deviation of a continuous feature for that feature to be utilized.

(m) `std_high_con`: Upper bound on the standard deviation of a continuous feature for that feature to be utilized.

(n) `missing_categorical_data_token`: Special token used to replace NaN values in categorical columns during imputation.

3. `dict_config_representation_encoder_training`: Hyperparameter group controlling the representation encoder training task.

   (a) `min_unique_count_to_train_repr_encoder`: Lower bound on the unique count of a continuous feature in order to train a dedicated representation encoder. We do not train representation encoders for continuous features with low value counts, as they tend to overfit.

   (b) `bert_hyperparam_config`: Hyperparameter kwargs for `TextEncoder` model. See the API reference for details. Note that not all hyperparameters need to be specified here. Since the relevant text encoders will be used to derive representation encodings of numeric data, some hyperparameters (e.g. the token list) are already determined. Such choices have been hardcoded in the task script.

   (c) `bert_train_config`: Parameter kwargs for the `TextEncoder`'s `fit()` method. See the API reference for details.

   (d) `bert_min_context_length`: Minimum context length to be used by the text encoders. Since we do not train daily, evaluation data might contian records with larger lengths than the max length encountered during training. Records exceeding the model's `max_n_standard_tokens` attribute will be truncated on the right. This will inevitably result in some information loss. The present hyperparameter is intended to minimize this effect.

4. `dict_config_anomaly_detector_training`: Hyperparameter group controlling the anomaly detector training task.

   (a) `min_n_cats_per_cat_feature`: Minimum number of unique categoires for a given categorical feature to be added to the model.

   (b) `max_n_cats_per_cat_feature`: Maximum number of unique categoires for a given categorical feature to be added to the model.

   (c) `embedding_dim_calculator`: Callable receiving the number of unique categories of a given categorical feature and returning the corresponding embedding dimension.

   (d) `ae_hyperparam_config`: Hyperparameter kwargs for `TransactionAnomalyDetector` model. See the API reference for details.

   (e) `ae_train_config`: Parameter kwargs for the `TransactionAnomalyDetector`'s `fit()` method. See the API reference for details.

5. `dict_config_eval_data_preprocessing`: Hyperparameter group controlling the evaluation data preprocessing task.

   (a) `evaluation_data_operator_name`: Operator name used to filter evaluation transactions (filter discarded if this is set to `None`).

   (b) `n_evaluation_days`: Number of days (including `run_date`) for which to pull evaluation data.

6. `dict_config_evaluation`: Hyperparameter group controlling the evaluation task.

   (a) `anomaly_score_quantile_cutoffs`: dictionary prescribing the HML reconstruction loss quantiles. The keys should be given by `low`, `medium` and `high` and the values should be equal to the respective quantiles.

5 Directions for Future Research

This section is intended to provide an account of pending tasks and open research directions—ranked by significance.

1. Design an efficient validation procedure to tackle the geographical segmentation optimization problem.
2. Complete the manual review process for the exported anomaly archive in order to obtain concrete evaluation metrics.
3. Design a procedure (potentially using offline LLMs) to incorporate text data.
4. Explore alternative anomaly interpretation methods (e.g. [6]).

References

[1] L. Antwarg, R. M. Miller, B. Shapira, and L. Rokach. Explaining Anomalies Detected by Autoencoders using Shapley Additive Explanations. *Expert Systems with Applications*, 186, 2021.

[2] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *arXiv:1607.06450v1 [stat.ML]*, 2016.

[3] J. Devlin, M-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT*, 1:4171–4186, 2019.

[4] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning*, 37:448–456, 2015.

[5] S. M. Lundberg and S. Lee. A Unified Approach to Interpreting Model Predictions. *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing*, 31:4768–4777, 2017.

[6] D. F. N. Oliveira, L. F. Vismari, A. M. Nascimento, J. R. de Almeida Jr, P. S. Cugnasca, J. B. Camargo Jr, L. Almeida, R. Gripp, and M. Neves. A New Interpretable Unsupervised Anomaly Detection Method Based on Residual Explanation. *arXiv:2103.07953 [cs.LG]*, 2021.

[7] N. Srivastava, G. Hinton, A.Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Reseach*, 15(56):1929–1958, 2014.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All You Need. *Neural Information Processing Systems*, 30:6000–6010, 2017.