Using Autoencoders to Detect Errors in Telemetry Events

Andrew Younger

April 19, 2022

1 Introduction

Telemetry events are essential for accurate reporting on player activity and progress for modern games. They allow the analytics team to dive deeply into topics and analyze important questions from stakeholders in the project. These events can be quite expansive, detailing anything from how a player kills an enemy to what cosmetics a player applies to their character to how quickly players are able to complete the game.

During the development cycle of a game there are frequent code changes which can affect telemetry frequently both accidentally and by design. QC teams work diligently in order to keep up with testing events as they are created but changes that affect older events often produce bugs which impact reporting done by the analytics team for playtests. In addition, telemetry events can be quite large and have many attributes with many different possibilities. A player.kill event, for example, where there are 50 weapons and 50 enemy types tracked yiels 2500 possibilities in just two columns. This further exacerbates the issues that the QC teams face when trying to catch every erroneous scenario that occurs inside telemetry events leading to the questions of: "How do we determine when an event has bugs?" and "How do we know when QC teams should retest old events?".

An immediate proposition to answer these questions is to use some form of automated bug detection. There are two methods of bug detection which could be helpful for QC teams to track down and deal with bugs: checking for some sort of spike in event frequency (if events start transmitting too often or stop transmitting then there may be an issue with the event's trigger mechanism) and statistsically determining if event attribute A is supposed to match with event attribute B. This report outlines a process for the second method by using an autoencoder to learn the possibility space of an event.

1.1 Autoencoders

Autoencoders are type of neural network that fall into the *encoder-decoder* family of neural networks. The general process for an autoencoder is shown in Figure 1. The encoding section of an autoencoder learns to "compress" the data down into a smaller dimension so that the decoding section of the autoencoder can attempt to recreate the original data from this low-level representation of the data. Autoencoders are used for several purposes including: image denoising, facial recognition, and, importantly for finding bugs, anomaly detection.

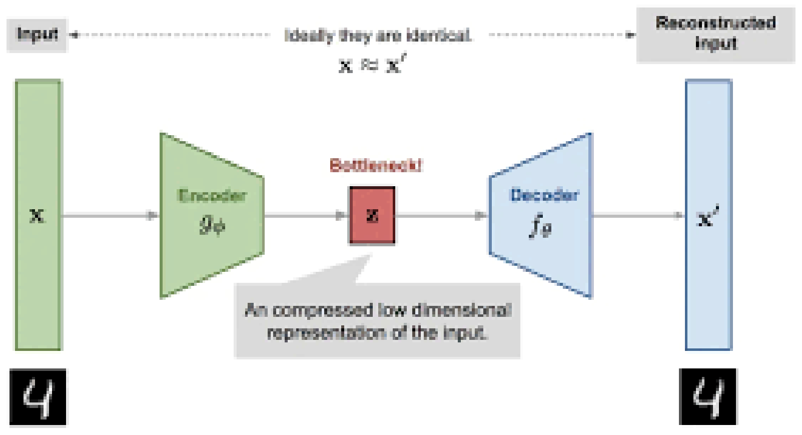
Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 1: The general layout of an autoencoder network. The network works by learning efficient compression to a low dimension and then reconstructing the data as closely as possible.

Using an autoencoder to detect anomalies in data is done by looking for data entries that have too high of an error in reconstruction. The error function is generally mean squared error (MSE) or mean average error (MAE). A threshold can be set, either manually or using some statistical methods, and any entry that gives a higher reconstruction error than the threshold is flagged as an anomalous event. For this particular scenario autoencoders are a prudent choice because they learn the specific data they were trained on quite well which leads to a much larger reconstruction error for anomalous entries. Telemetry events correspond to specific actions in-game and as such the majority of events should not be anomalies which gives autoencoders a larger advantage - especially when combined with semi-supervised learning methods.

1.2 Semi-Supervised Learning

Machine learning problems are often considered to fall into the either the category of supervised learning or into unsupervised learning. Semi-supervised learning provides a middle ground between the two methods where a model can be sufficiently trained on a dataset where only a small percentage of data points are labeled as anomalies beforehand. Under the assumption that *most* of the data entries are not anomalies a semi-supervised autoencoder can be trained to detect erroneous entries.

2 Methods and Models

In order to determine if a semi-supervised autoencoder is a suitable candidate for detecting bugs in telemetry events multiple experimental steps were taken:

1. Proof of concept model using a specific bug that was found and fixed as well as the fixed events corresponding to the expected behaviour of the event.
2. The same bug and corrected behaviour but testing MSE as the error function instead of MAE.
3. A larger-scale version of the initial proof of concept that has a smaller percentage of known anomalies.
4. A model that is truly semi-supervised and uses a large number of arbitrary events to detect the specific bug.

Many telemetry events are designed to give context on what tools a player is using and how the player uses them and thus many of the most important attributes of a telemetry event are categorical data. To have a model read categorical data it must be “translated” into some numerical form.

2.1 One-Hot Encoding

One-hot encoding is a common method to do this and works by turning each categorical entry in a column of distinct length n into a sparse vector of length n where only the corresponding entry is non-zero. For an example of how this process works consider a simple dataset where each entry is simply an ID column, a colour and a shape:

ID	Colour	Shape
1	Blue	Square
2	Red	Triangle
3	Red	Square
4	Orange	Circle
5	Pink	Circle

To use one-hot encoding and vectorize this dataset first look at the number of unique entries in each categorical field. There are 4 unique colours and 3 unique shapes so each entry in the colour column will be a sparse vector of length 4 and each entry in the shape column will be a sparse vector of length 3. The vectorized entries would look like this:

ID	Colour	Shape
1	[1 0 0 0]	[1 0 0]
2	[0 1 0 0]	[0 1 0]
3	[0 1 0 0]	[1 0 0]
4	[0 0 1 0]	[0 0 1]
5	[0 0 0 1]	[0 0 1]

Each entry in the vector can be thought of as a binary flag for the entry: "Is it blue? Is it red?" etc. All the different models that were trained used one-hot encoding as part of the data preprocessing.

2.2 Proof of Concept Model

The original proof of concept model used a relatively small size of proper telemetry events and a relatively high number of anomalous events to compare against. The dataset that was chosen for the original proof of concept model was from Watch Dogs: Legion and used the player.kill event. This model was trained against a specific bug that was found and fixed; when players destroyed an enemy drone it would occasionally register as a melee kill even when the player had used a gun to destroy the drone. Ten thousand player.kill events where a drone was destroyed and sent a proper event were used and one thousand of the aforementioned bugged events were used.

The training and testing process for this model was replicated for the subsequent models that were created.

1. Split data into train/test sizes. 80% training size was used.
2. Normalize numerical data columns.
3. Vectorize categorical columns using one-hot encoding.
4. Train the autoencoder on the processed data.
5. Determine a threshold for what constitutes an anomalous event.

There are a few options for determining the threshold for anomalous events. The first is to look at the reconstruction errors and manually pick a threshold that separates the proper events from the anomalies. This could lead to a lot of inconsistency in other use-cases so in order to have a consistent threshold it was chosen to be one standard deviation away from the mean reconstruction error. For this first model this was a threshold of 0.708. Any reconstruction error above this was labeled as a bugged event. Figure 2 shows the reconstruction errors for the training of the model.

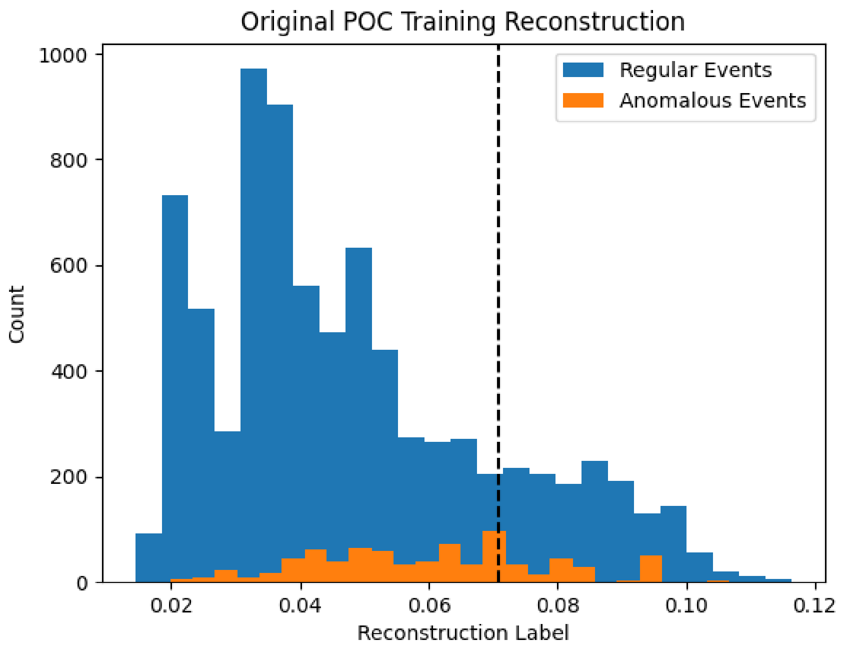

Figure 2: Reconstruction errors for regular events and anomalous events during training. The black line shows the anomaly threshold.

During training it is easy to see that the majority of regular events have a reconstruction error that is less than the calculated threshold. However many of the known anomalous events are also lower than the calculated threshold. There is even a higher percentage of the regular events that are higher than the threshold than there are anomalous events. This ultimately led to a relatively low accuracy score for the original proof of concept. The scores for the original model are shown in the table below. With the exception of the model accuracy the metrics are very promising for a proof of concept. In Figure 3 the testing reconstruction errors can be seen and the reconstruction errors for only anomalous events can be seen in Figure 4.

Metric	Score
Accuracy	78%
Precision	91%
Recall	84%
F1 Score	87%

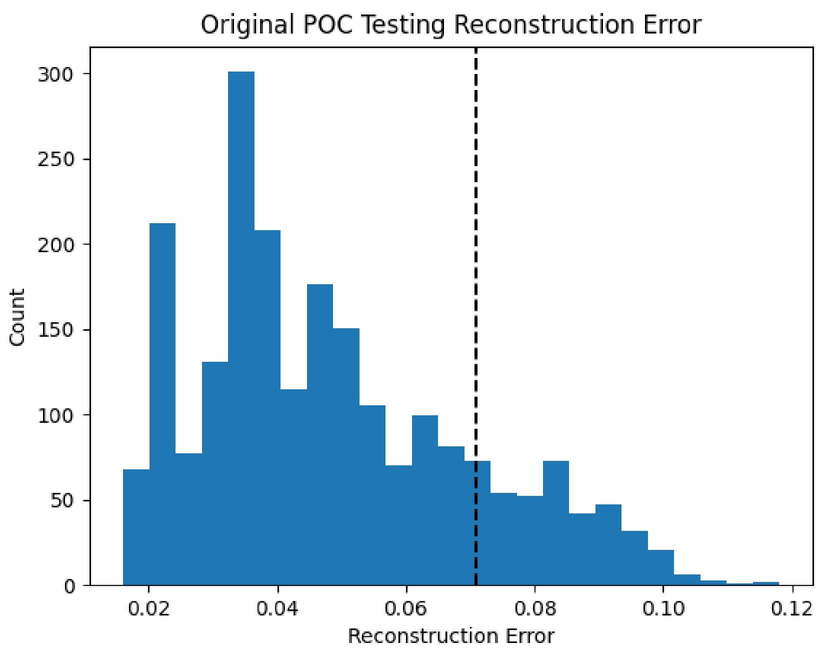

Figure 3: Reconstruction errors for regular events and anomalous events during testing. The black line shows the anomaly threshold.

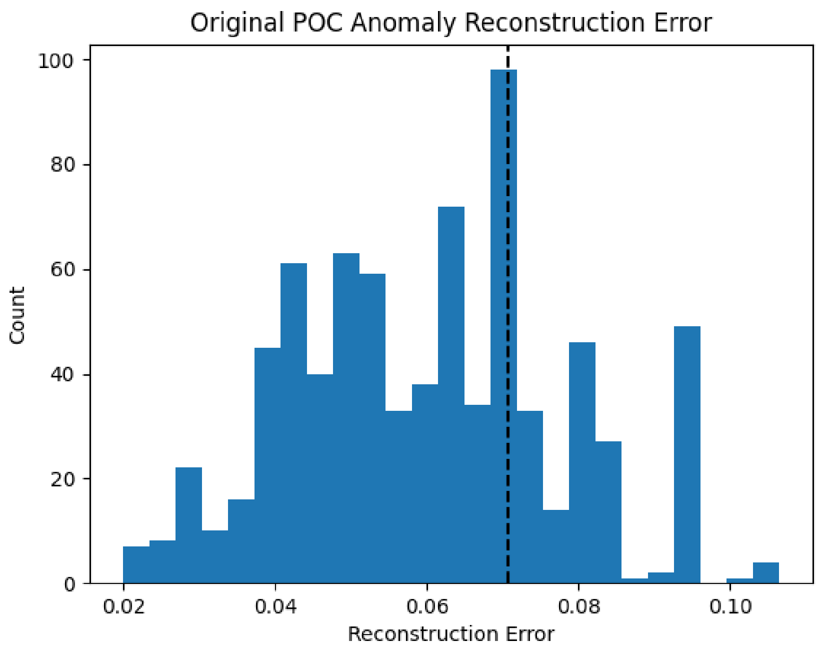

Figure 4: Reconstruction errors for regular events and anomalous events during testing. The black line shows the anomaly threshold.

From these two graphs it is easier to see that the model was very good at identifying anomalous events after training finished. The immediate concern is the how low the accuracy and the recall for the model are. This could be the result of the relatively high number of anomalous events. Almost 10% of the training data was anomalous and in a realistic scenario bugs likely between 2% and 3% of events at most. In a future model this percentage is lowered to a more realistic amount.

2.3 MSE Model

The second model was identical to the first in both data and process with the only exception that the error function was switched from MAE to MSE. Since MSE uses the square of the error instead of the average it should give a higher relative error for the anomalies. The MSE model improved all the metrics but only very slightly - less than 1%. However any improvement is good and so MSE is the chosen error function for futre models.

2.4 Small Sample Model

The small sample model is very similar to the original proof of concept. The main difference is that the ratio of regular events to labeled anomalous events is much smaller. Instead of ten thousand regular events and one thousand anomalies it used thirty thousand regular events and the same one thousand anomalies. Using this much larger number of regular events to train the autoencoder greatly improved the model performance, reducing the threshold for anomalous events to 0.067. In Figures 5 and 6 the reconstruction errors for anomalies and regular events respectively can be seen for the small sample model.

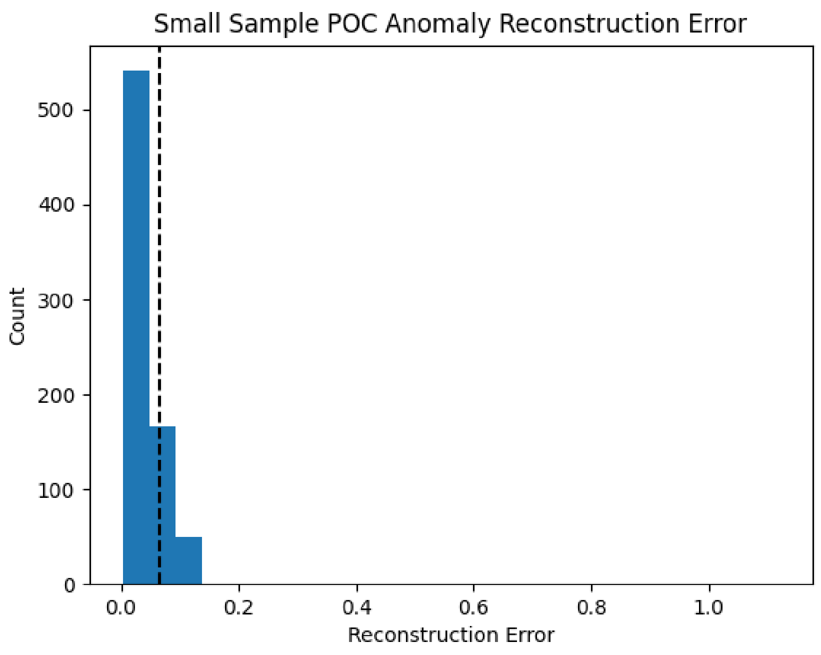

Figure 5: Reconstruction errors for anomalous events. The black line shows the anomaly threshold.

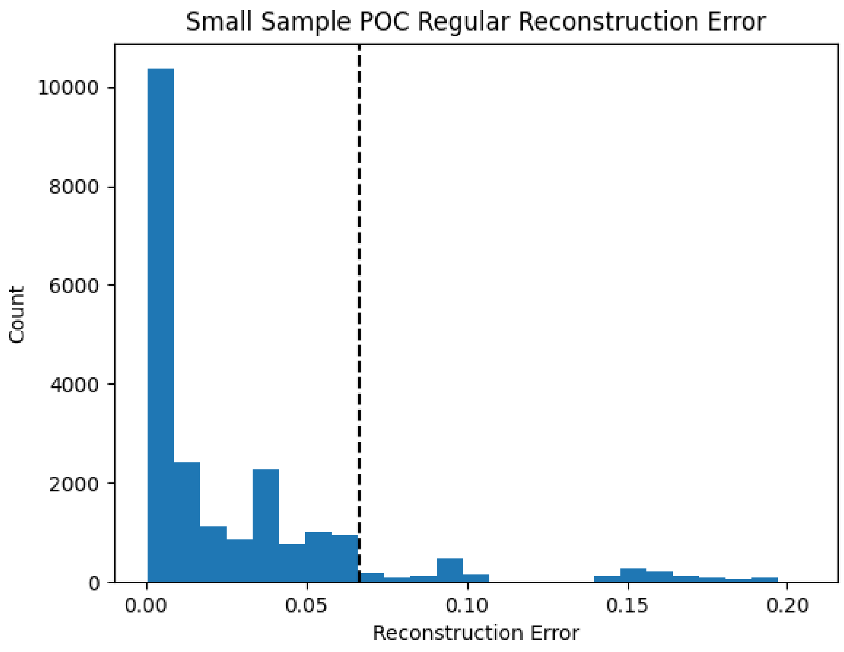

Figure 6: Reconstruction errors for regular events. The black line shows the anomaly threshold.

The small sample model classifies more regular events correctly compared to the original model although still misses several of the anomalous events. The following table shows the increase in model performance metrics between the original model and the small sample model. The small sample model improved over the original model in every metric. This was promising for the final model which increased the number of training samples even further.

Model	Accuracy	Precision	Recall	F1 Score
Original POC	78%	91%	84%	87%
Small Sample	88%	96%	91%	93%

2.5 Large and Arbitrary Model

The fourth and final model that was created used 100k completely arbitrary player.kill events to train the autoencoder. The previous models all used corrected versions of the bug that was attempting to be detected. This means that while it is possible that there are some other bugs in the training data it is the most similar to how an autoencoder would be used in practice to detect bugs in telemetry events. In this model the reconstruction error threshold was reduced to 0.0525. For this model there is no comparison of regular events to anomalous events because we do not know for sure if the events used to train the model are anomalies or not. Figure 7 shows the reconstruction error for the testing sample in this model.

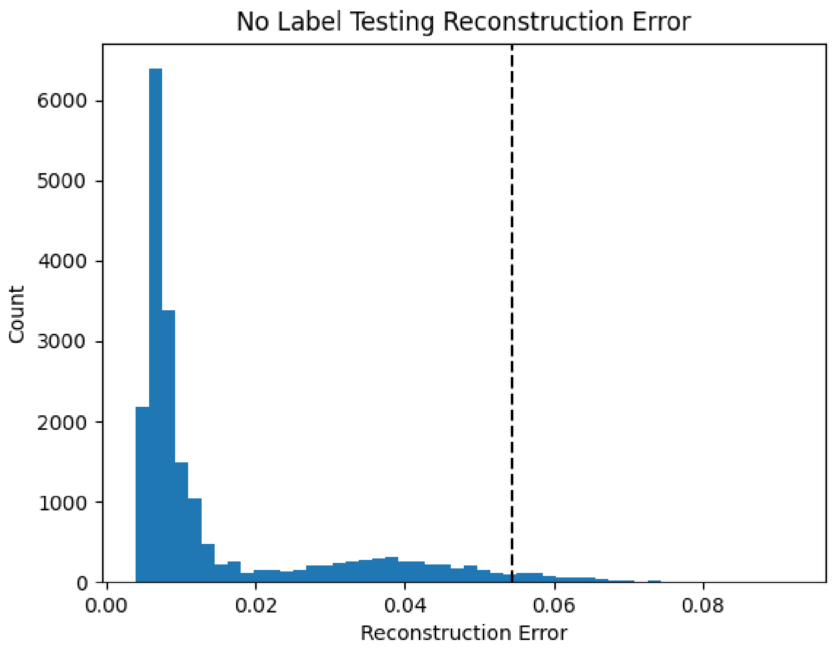

Figure 7: Reconstruction errors during model testing. The black line shows the anomaly threshold.

We can see that perhaps the standard deviation threshold is too high for this model and the threshold should maybe be set closer to 0.04 or even 0.02 to be almost certain of catching every anomaly. Figure 8 shows that the vast majority of the known bug (drones destroyed via melee attacks) are caught and properly labeled as being bugs. This suggests that while the threshold is too high, it is not so high as to be completely useless. The model's metrics were calculated twice, once while using the threshold as one deviation higher than the mean and one with the threshold set to be the mean (0.191). Using the lower threshold of the mean hugely improved the performance of the model. Figure 9 shows the drastic improvement in detecting known bugs.

Model	Accuracy	Precision	Recall	F1 Score
Original POC	78%	91%	84%	87%
Small Sample	88%	96%	91%	93%
No Label + Arbitrary	89%	99%	89%	94%
No Label + Arbitrary (Low Threshold)	98%	99.8%	99.1%	99.4%

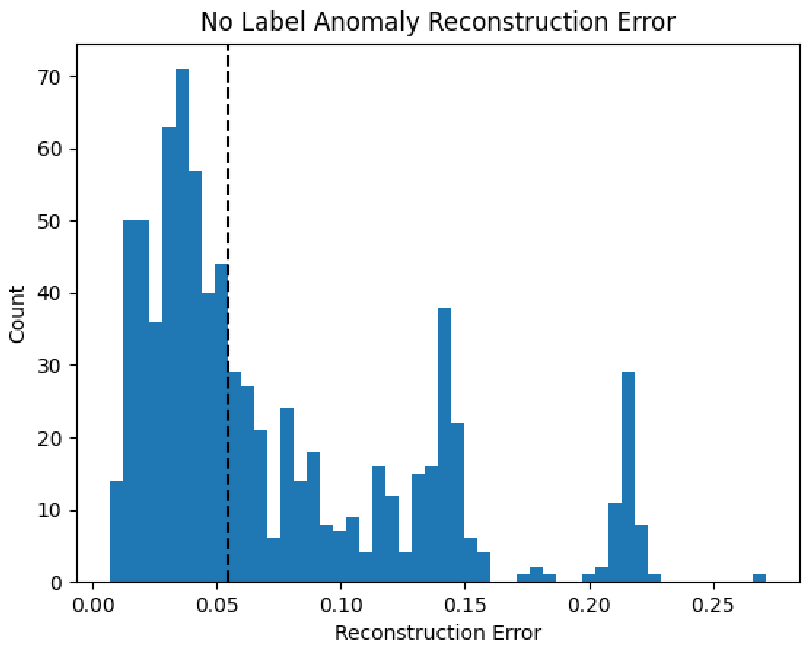

Figure 8: Reconstruction errors for known anomalous events. The black line shows the anomaly threshold - most of the known bugs are properly captured.

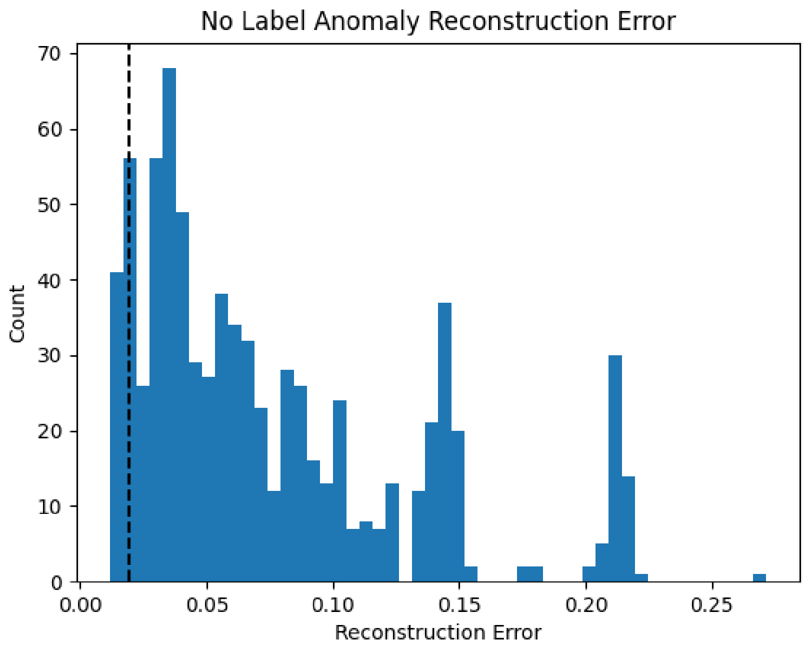

Figure 9: Reconstruction errors for known anomalous events using the lower threshold. The black line shows the anomaly threshold - almost all of the known bugs are properly captured.

3 Conclusions and Future Work

Using autoencoders to detect errors in telemetry events is entirely possible. With a sufficiently large sample size a very precise and accurate model can be trained. The reconstruction error on arbitrary events is closely clustered and using a low threshold like the mean reconstuction error can accurately determine if an event is an anomaly or not. A threshold that is too low could mean falsely labeling many events as bugs when they are not. This could lead to lots of manual work from QC correctly labeling events as bugs or not; with regular retraining this should help improve the model's performance.

Future work for detecting anomalies includes investigating other methods of vectorization for categorical columns as well as other types of models that could be more flexible than an autoencoder. One-hot encoding is a very inefficient method of vectorization and when the dataset size becomes larger can take a lot of time to generate vectors that the neural network can read. Other encoding options are learned embedded encoding and ordinal encoding. Autoencoders are not very flexible in the sense that a different autoencoder has be trained for every event. This can get repetitive and if multiple events need similar but slightly different models it can be expensive to maintain many different ones. There are other options for anomaly detection which may prove to be more flexible and should be investigated including: generative adversarial networks, sequence-to-sequence models, one-class support vector machines and siamese networks for oneshot learning.

The more immediate next step is to establish an automatic process to detect possible anomalies in events. This will include a dashboard system where anomalous events are logged for a specific event as well as the rate and number of anomalies between builds. The model will retrain periodically but will not yet include input from QC marking events as anomalous or not.