Fall Detection

Rafael A. G. França

Summary

This report was prepared for the Prosigliere ML Challenge. The task was to build and test models for detecting falls from sensor data. The dataset had signals from several body locations, and the goal was to separate normal daily activities, near falls, and actual falls.

We started with some steps to load and prepare the data, followed by exploration to understand the main patterns. After that, we trained two models: XGBoost and MiniRocket. XGBoost required some feature engineering to capture useful information from the signals. MiniRocket, on the other hand, worked directly on the raw time-series data without extra features.

Both models performed very well, but MiniRocket ended up slightly ahead, even without feature engineering. This shows the strength of time-series specific methods for this type of problem.

The report describes the setup, data preparation, models, and their results side by side. Extra plots and details are in the appendix.

Environment

All the work for this challenge was done in Jupyter Notebooks, running locally on a MacBook Pro (16-inch, Nov 2023) with an Apple M3 Pro chip, 18 GB of memory, and macOS 15.7.

The code, notebooks, and results can be found on GitHub:
https://github.com/psyny/FallDetectionChallenge

The environment was Python 3.11 (better ML stability) with the main libraries being pandas, numpy, matplotlib, scikit-learn, and xgboost. MiniRocket was used through the sktime package.

Consolidating the Data

We first pulled every .xlsx inside data/raw and stitched them into one table. For each file we grabbed three bits of metadata from the path: subject (first folder), label (second folder), and experiment (file name without extension). These went in as new columns so we don't lose context later.

That gave us a single DataFrame with 952,165 rows × 67 columns. All sensor columns were cast to float for consistency. The identifier columns stayed as strings.

We ran a full missing-value scan per column and overall. And found no missing values. We also checked and dropped duplicates. None were found.

Finally, we wrote the consolidated dataset to data/consolidated/fall_dataset.parquet using pyarrow with Snappy compression, and did a quick read-back to sanity check. This file is what the notebooks use from this point on so we don't keep re-reading a bunch of spreadsheets.

Data Exploration

Before modeling, we ran some checks to get a sense of the dataset.

First, we looked at the number of samples per experiment and per class. This helped confirm if the dataset was balanced enough. The histograms show that ADLs, Falls, and Near_Falls have fairly consistent sample counts per experiment.

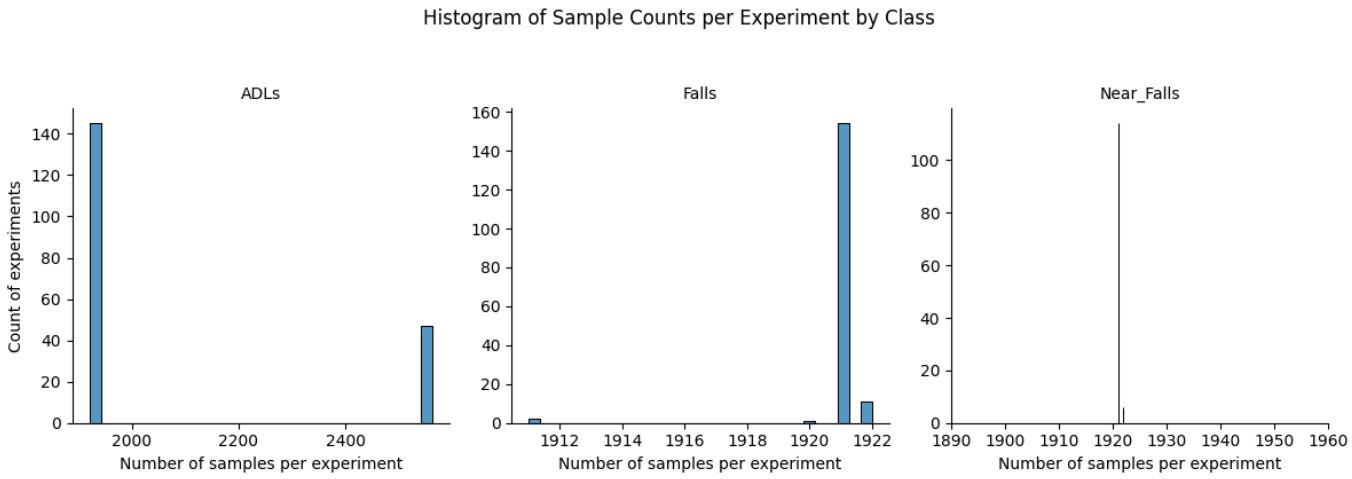


Figure 1 – Histogram of sample counts per experiment by class

We also checked the sampling frequency. All experiments were sampled at ~128 Hz, and this is stable across classes. This means we don't need to do any resampling or corrections.

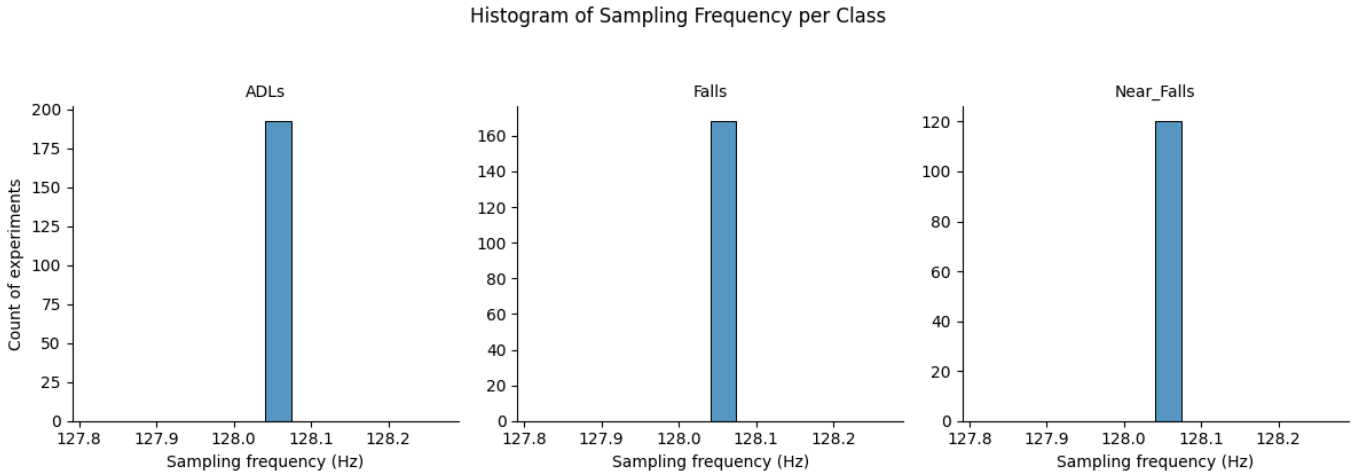


Figure 2 – Histogram of sampling frequency per class

Experiment durations were another point of interest. Most runs lasted around 15 seconds, with some ADL recordings going up to ~20 seconds. This confirmed that experiments were long enough for window-based processing.

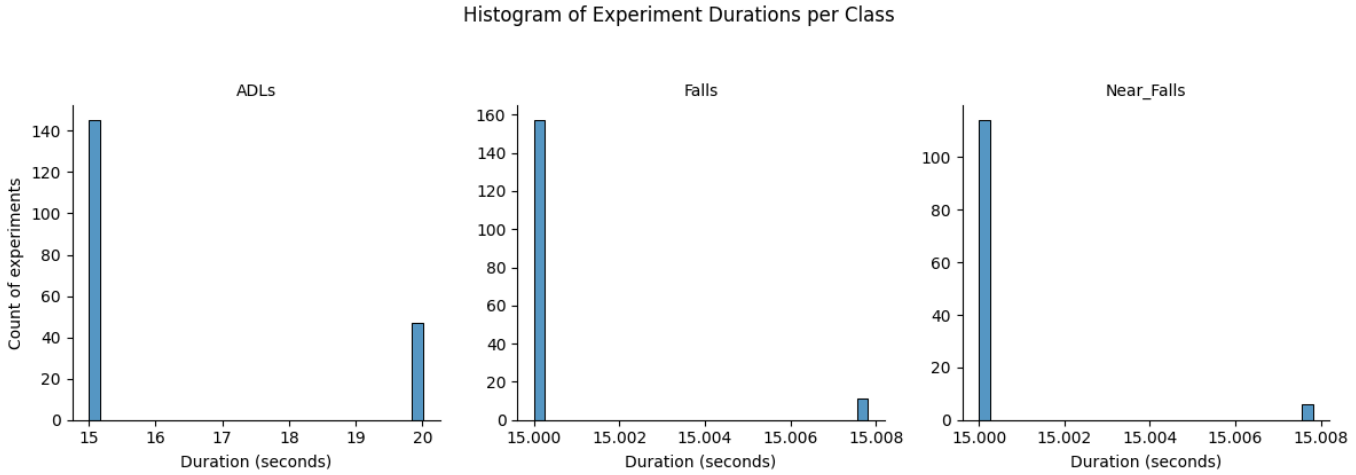


Figure 3 – Histogram of experiment durations per class

Finally, we plotted sensor values over time to see if any visible patterns show up. For instance, the right ankle magnetic field (X axis) shows clear differences during falls compared to near falls and ADLs.

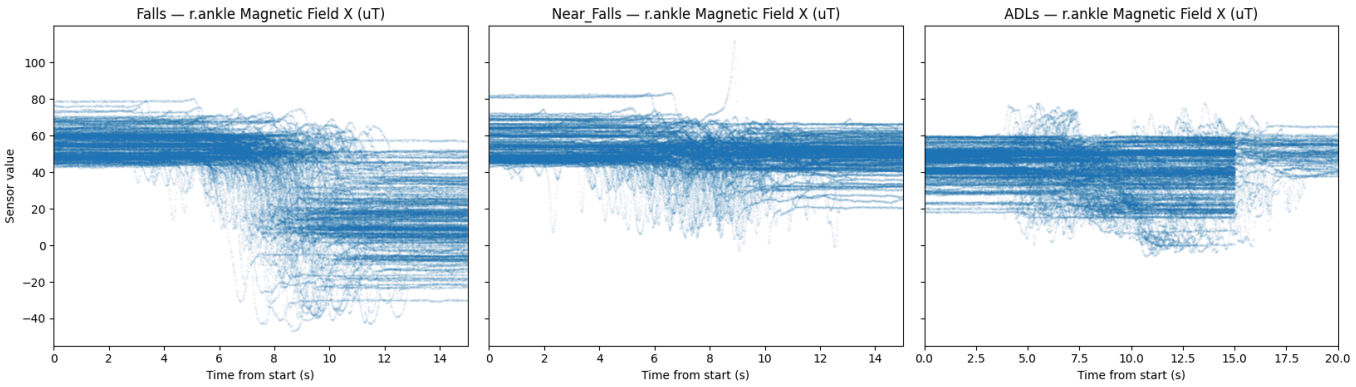


Figure 4 – Right ankle Magnetic Field X over time, separated by class

Plots for the other sensors are available in Appendix A.

XGBoost

Why?

We chose XGBoost as a first model because it's a strong baseline for structured data. It handles large feature sets well, deals with non-linear relations, and usually gives solid results without extreme tuning. Since our plan was to turn the raw signals into tabular features, XGBoost was a natural fit. It also trains fast and makes it easy to check feature importance, which helps in understanding what parts of the signal contribute most.

The limitation is that XGBoost is not time-series specific, so it can't directly model temporal patterns. That's why we later explored MiniRocket, which works directly on the raw sequences.

Preparing the Data

XGBoost works with tabular features, so we had to turn the raw time-series into a structured dataset. The approach was to summarize each sensor signal per experiment into a fixed set of features. That way, the model sees one row per experiment instead of thousands of raw time points.

We engineered both statistical and signal-processing features. The idea was to capture the shape and variability of the signals, as well as some frequency information that might separate falls from daily activities.

The features included:

- Basic stats: mean, std, min, max, median, interquartile range
- Distribution shape: skewness, kurtosis
- Energy measures: RMS, signal energy
- Dynamics: zero crossing rate (captures how often the signal changes direction)
- Entropy: Shannon entropy of the signal histogram (measures unpredictability)
- Spectral features: spectral centroid, and band power in four ranges (0–2 Hz, 2–5 Hz, 5–10 Hz, >10 Hz)

We did this for every sensor channel, prefixing the feature names with the sensor name. Each experiment ended up as a wide vector of these descriptors.

Labels were taken from the folder structure and were consistent within an experiment. We encoded them as integers (ADLs, Falls, Near_Falls).

To avoid data leakage, we split the dataset by subject using grouped shuffles. The split strategy was:

20% test subjects (never seen in training or validation)
From the remaining, 20% validation subjects
The rest used for training

This setup ensures the model learns patterns that generalize across different people, not just repetitions of the same subject.

Training

For XGBoost we started with the most common values for this kind of multiclass setup, and adjusted them a bit based on what made sense for the data. There's always some guesswork here, but the idea was to pick reasonable defaults and let early stopping find the right number of trees.

objective = "multi:softprob" → standard for multiclass with probability outputs.
num_class = 3 → one for ADLs, Falls, Near_Falls.
n_estimators = 2000 → set high, since early stopping decides when to stop anyway.
learning_rate = 0.05 → small enough to keep learning gradual, but not so small that training drags forever.
max_depth = 6 → a middle ground, deep enough to capture interactions but not crazy deep.
min_child_weight = 2.0 → helps avoid splits backed by very few samples.
subsample = 0.9 and **colsample_bytree = 0.9** → add randomness to reduce overfitting.
reg_lambda = 1.0, **reg_alpha = 0.0** → standard L2 regularization; L1 wasn't needed.
tree_method = "hist" → faster training with large data.

random_state = 42 → reproducibility.
n_jobs = -1 → use all cores.
eval_metric = "mlogloss" → good fit when you want calibrated probabilities.
early_stopping_rounds = 100 → gives training a chance to settle, but stops if validation loss stays flat.

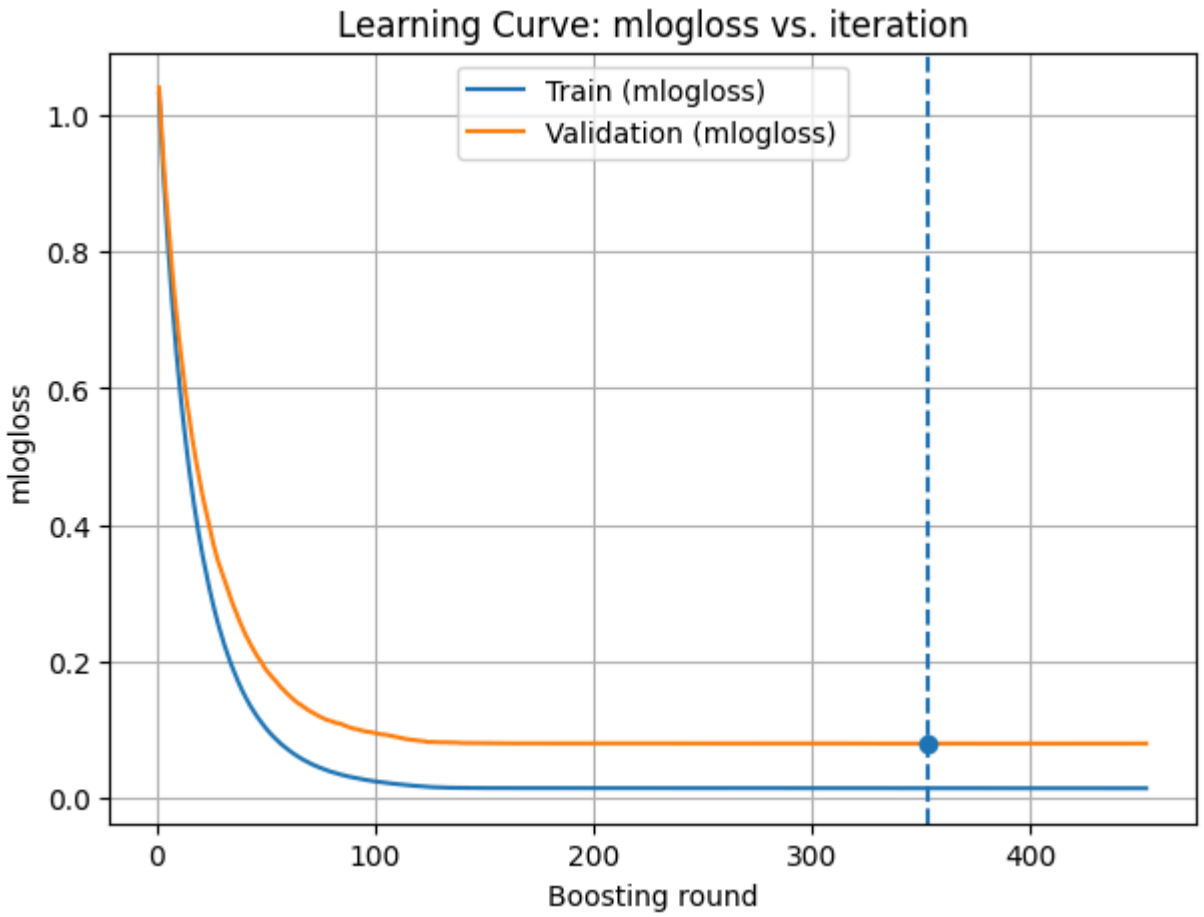


Figure 5 – Learning curve (mlogloss) for training and validation sets, with early stopping point marked

The curve shows loss dropping fast in the first ~100 rounds, then flattening out. The vertical line marks the point where validation stopped improving (around round 350). That's the iteration chosen as best. Training past that wouldn't have helped.

Evaluation

The model showed strong results on both validation and test sets. Accuracy was ~97% on validation and ~98% on test, with balanced precision and recall across the three classes. This indicates the model is not only accurate but also consistent in how it handles the different event types.

```
Validation classification report
              precision    recall  f1-score   support

        ADLs      0.958     0.958     0.958        48
       Falls      0.955     1.000     0.977        42
  Near_Falls      1.000     0.933     0.966        30

    accuracy                          0.967       120
   macro avg      0.971     0.964     0.967       120
weighted avg      0.967     0.967     0.967       120
```

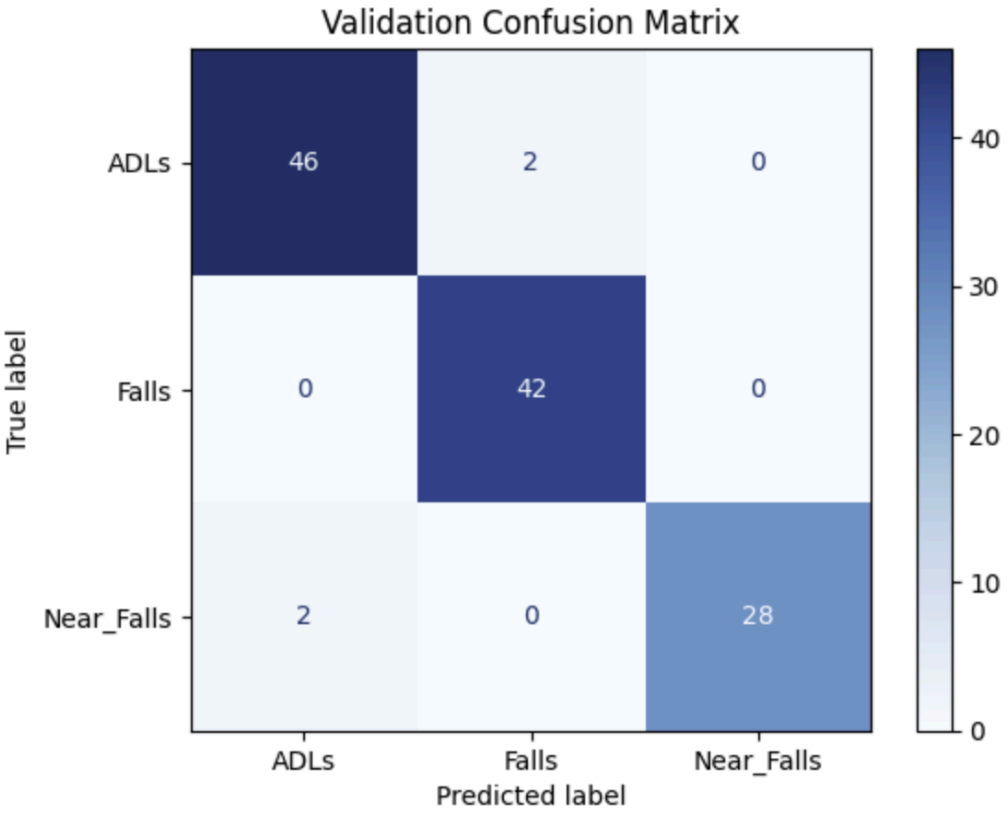


Figure 6 – Validation classification report and confusion matrix

The validation set shows high performance across the board. The confusion matrix highlights that most errors were between ADLs and Near_Falls, which is expected since these two can look similar in the signals.

```
Test classification report
              precision    recall  f1-score   support

        ADLs      0.941     1.000     0.970        48
       Falls      1.000     0.976     0.988        42
  Near_Falls      1.000     0.933     0.966        30

    accuracy                          0.975       120
   macro avg      0.980     0.970     0.974       120
weighted avg      0.976     0.975     0.975       120
```

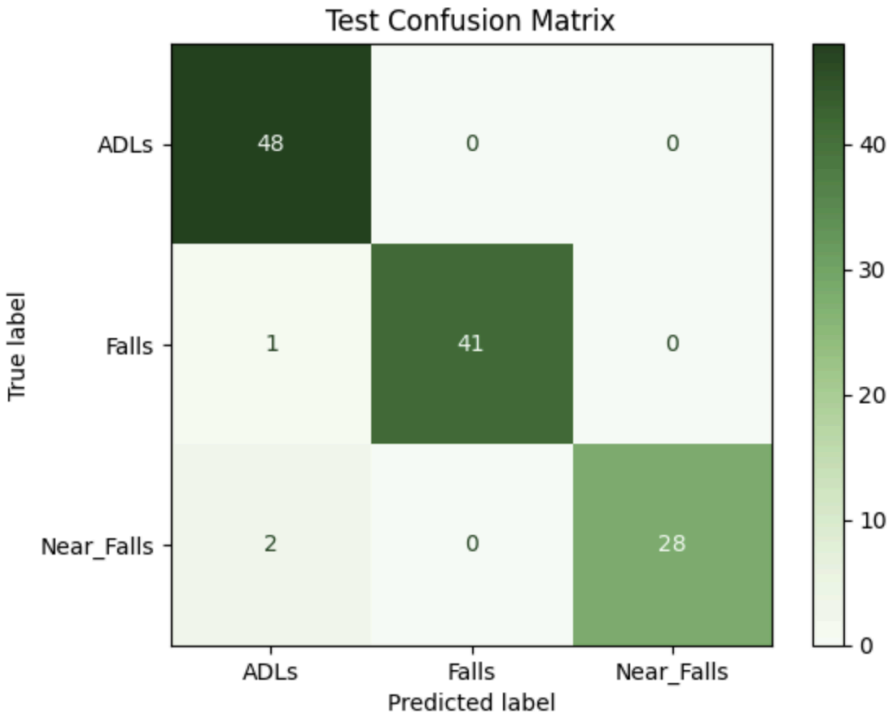


Figure 7 – Test classification report and confusion matrix

The test set confirms that performance holds on unseen subjects. Here, too, the few mistakes came from confusing Near_Falls with ADLs or Falls, which is consistent with how subtle near-fall patterns can be.

To better understand the model, we checked the top features ranked by importance.

```
Top 20 features:
 r.ankle Acceleration X (m/s^2)__mean          0.059940
sternum Acceleration Y (m/s^2)__min           0.050203
sternum Angular Velocity Z (rad/s)__rms       0.041925
sternum Angular Velocity Z (rad/s)__max       0.039138
l.thigh Acceleration X (m/s^2)__kurtosis      0.033555
waist Acceleration X (m/s^2)__std             0.024586
r.ankle Acceleration X (m/s^2)__iqr           0.023898
r.thigh Acceleration Z (m/s^2)__bp_2_5        0.021388
waist Angular Velocity Z (rad/s)__kurtosis    0.020540
head Acceleration Y (m/s^2)__bp_5_10          0.019758
sternum Acceleration Y (m/s^2)__bp_2_5        0.018539
sternum Acceleration Y (m/s^2)__std           0.017711
sternum Acceleration Y (m/s^2)__bp_10_64.0    0.014299
l.ankle Angular Velocity Y (rad/s)__min       0.014201
r.ankle Acceleration X (m/s^2)__kurtosis      0.012264
r.ankle Magnetic Field X (uT)__std            0.011787
r.thigh Acceleration X (m/s^2)__kurtosis      0.011743
l.ankle Magnetic Field X (uT)__mean           0.011666
r.ankle Acceleration X (m/s^2)__rms           0.011610
r.thigh Angular Velocity X (rad/s)__min       0.011551
dtype: float32
```

Figure 8 – Top 20 most important features according to XGBoost

The most influential features came from accelerations and angular velocities, especially on the ankles, thighs, and sternum. This makes sense: falls and near falls should create distinct acceleration bursts and rotational movements in these areas. Seeing these features at the top helps confirm that the model is focusing on physically meaningful signals, not random noise.

Overall, the evaluation suggests the model generalizes well, is not overfitting, and is explainable to a degree through its feature importances.

Stressing The Model

To check how robust the model was, we ran a few stress tests.

Feature pruning
We removed features in order of importance, starting from the least relevant according to XGBoost. As shown below, accuracy, precision, recall, and F1 stayed very high even when we cut down the feature set a lot.

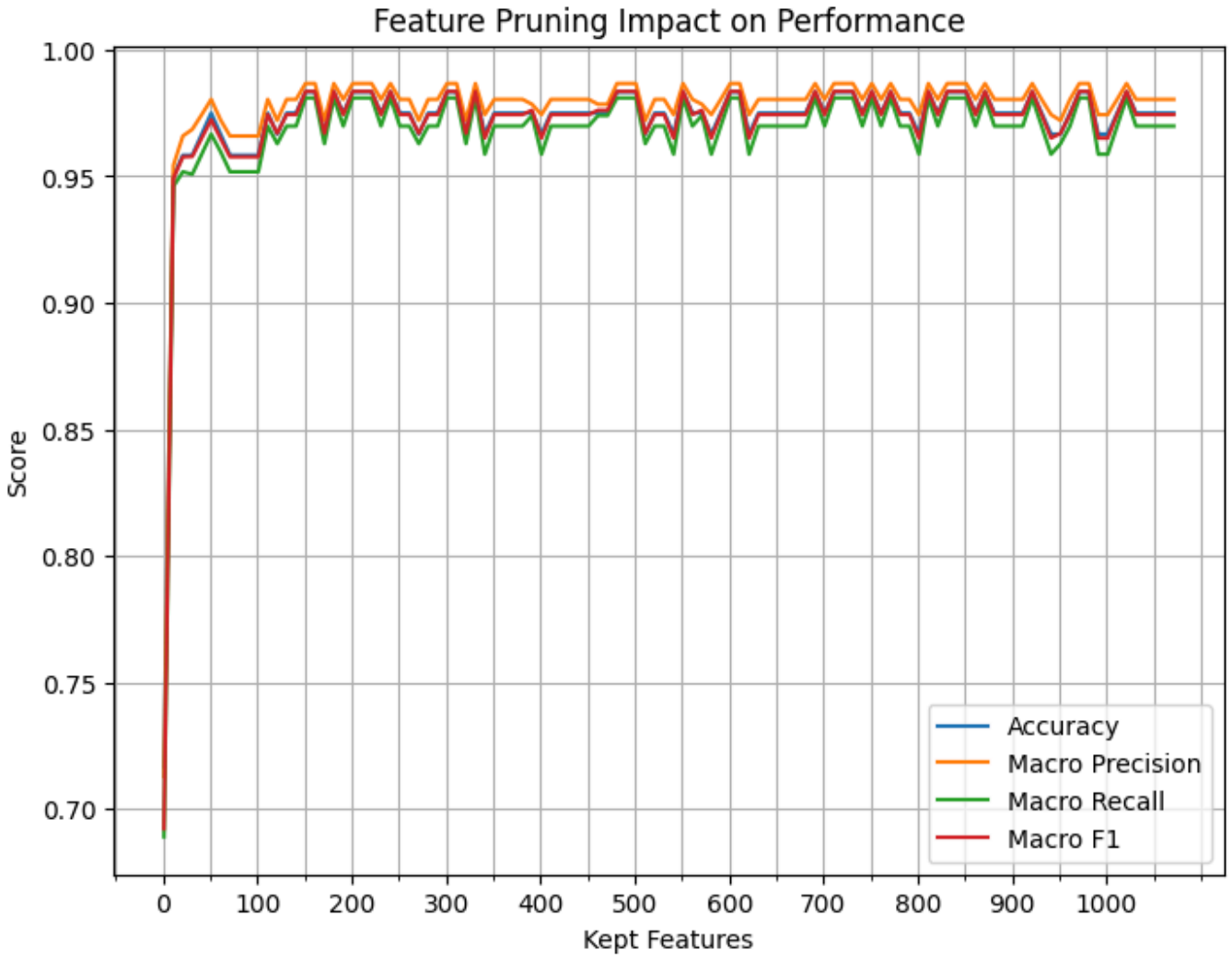


Figure 9 – Performance as features are pruned using feature importance

This suggests that many of the ~1000 engineered features were redundant. The model can work almost as well with a much smaller set, which means we could train and run it faster without losing much accuracy.

Noise injection

We also tested how the model behaves when sensor data is noisy. We added Gaussian noise proportional to the standard deviation of each channel, up to 50% of the original signal spread.

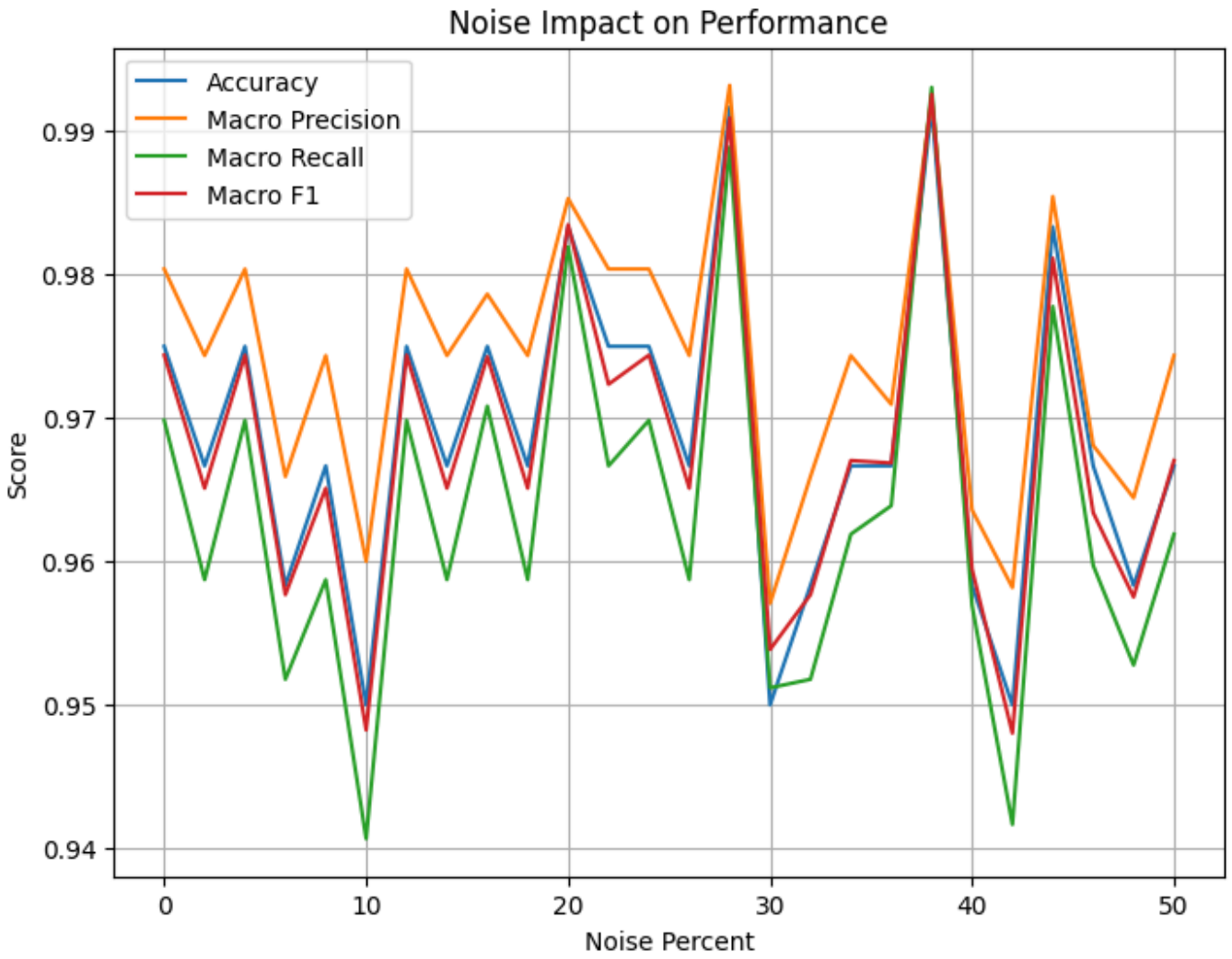


Figure 10 – Performance as Gaussian noise is added to test data

The scores dropped slightly as noise increased, but the model stayed above 94% accuracy even at the highest noise levels. This shows it is fairly resilient to realistic measurement noise.

Leave-One-Subject-Out (LOSO) validation
Finally, we tested generalization by holding out each subject in turn and training on the rest.

```
LOSO-CV summary over 8 folds
  held_out_subject  n_test  accuracy
0             sub1      60  0.933333
1             sub2      60  0.966667
2             sub3      60  0.983333
3             sub4      60  0.983333
4             sub5      60  0.983333
5             sub6      60  1.000000
6             sub7      60  1.000000
7             sub8      60  0.916667

Means ± std:
accuracy: 0.971 ± 0.031
```

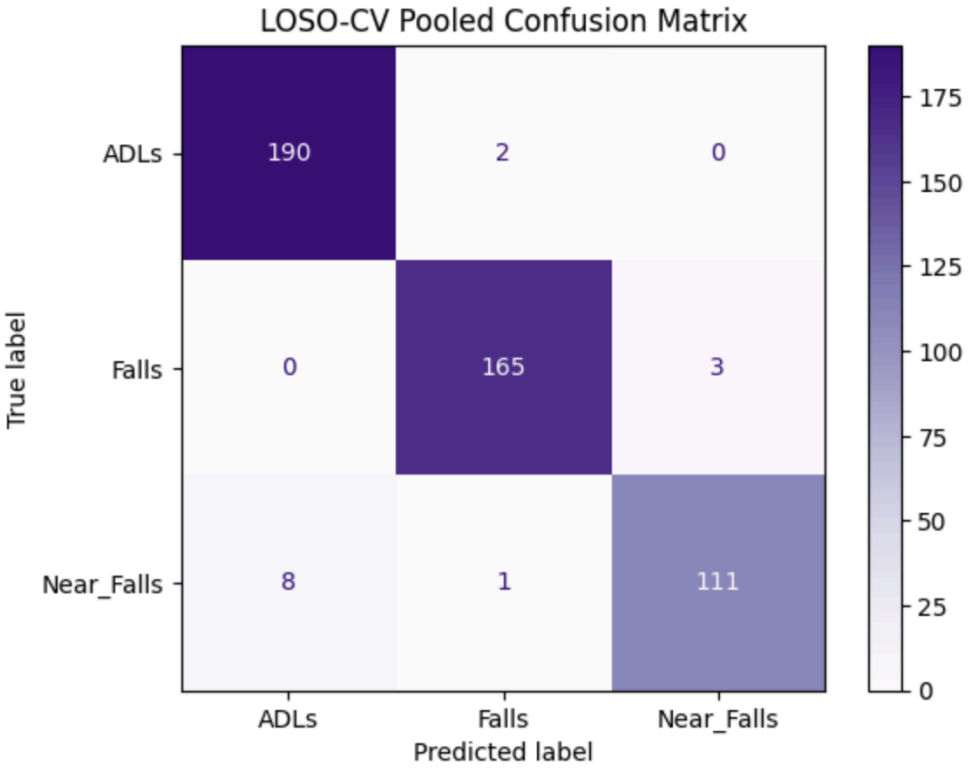


Figure 11 – LOSO-CV confusion matrix and per-subject accuracies

Accuracy varied a bit depending on the subject (lowest ~91%, highest 100%), but the average was 97% with small variance. Errors were mainly in Near_Falls, which are naturally harder to separate. Overall, the model proved robust across different people.

Takeaway
These checks show that the model is leaner than it first looked (thanks to feature pruning), stable under noisy conditions, and generalizes well across subjects. LOSO results also suggest that using the full dataset for training would have been valid, but to keep comparisons consistent we stayed with the train/val/test split strategy.

MiniRocket

Why?

After testing XGBoost with engineered features, we also wanted to try a method designed for time-series data directly. MiniRocket was a good choice here because it skips the need for heavy feature engineering and works very fast even on large datasets.

It transforms the raw sequences into a set of convolutional features, capturing temporal patterns that tree-based models like XGBoost can't see. This makes it well-suited for fall detection, where the shape and timing of signals matter.

So the idea was: XGBoost gave us a strong baseline with handcrafted features, and MiniRocket could show whether a sequence-based method could do even better without that manual step.

Preparing the Data

For MiniRocket the data preparation was simple compared to XGBoost. We used the same subject-grouped splits (train, validation, test) to keep comparisons fair. The raw sensor readings were reshaped into nested panel format, which is how sktime expects multivariate time-series input.

No extra feature engineering was done, MiniRocket builds its own convolutional features from the sequences. This allowed us to go straight from the cleaned dataset into model training.

Training

For this model we used MiniRocket to generate features from the raw sequences, and a RidgeClassifier to do the actual classification. This combination works well because MiniRocket quickly creates a large set of convolutional features that capture temporal patterns, and RidgeClassifier is a simple but strong linear model that handles high-dimensional data. Together, they're lightweight, fast, and effective.

MiniRocket itself doesn't need many parameters, we just fixed the random seed for reproducibility. After fitting on the training set, it transformed the data into about **10,000** features per sample, which is typical for this method.

For classification, we used RidgeClassifierCV. The parameters were pretty standard defaults for this kind of problem:

alphas = logspace(-3, 3, 13) → a broad range of regularization strengths to cross-validate over.
class_weight = "balanced" → accounts for the smaller Near_Falls class.
store_cv_values = False → left at default since we didn't need to inspect all CV scores.

To be honest, these were not heavily tuned. The reasoning was: stick with defaults that are known to work well on time-series tasks, rely on the regularization to stabilize the large feature set, and see how it performs. Training was fast, which made it easy to iterate.

Evaluation

MiniRocket with RidgeClassifier performed very strongly on both validation and test sets.

```
[VAL] acc=0.9833  f1_macro=0.9835  f1_weighted=0.9833
              precision    recall  f1-score   support

        ADLs       1.00      0.96      0.98        48
       Falls       0.98      1.00      0.99        42
  Near_Falls       0.97      1.00      0.98        30

    accuracy                           0.98       120
   macro avg       0.98      0.99      0.98       120
weighted avg       0.98      0.98      0.98       120

[TEST] acc=0.9917  f1_macro=0.9909  f1_weighted=0.9916
              precision    recall  f1-score   support

        ADLs       0.98      1.00      0.99        48
       Falls       1.00      1.00      1.00        42
  Near_Falls       1.00      0.97      0.98        30

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120
```

Figure 12 – Validation and test classification metrics

On validation, the model reached 98.3% accuracy with macro F1 of 0.98. All three classes were well captured, with only minor confusion between ADLs and Near_Falls. On the test set, performance was even higher, with 99.2% accuracy and nearly perfect precision and recall for Falls. This shows the model generalized very well.

Given the scope of the challenge and how strong the results already were, we did not push further tuning. Still, if we wanted to squeeze more performance, next steps could include:

- Adjusting the number of features MiniRocket generates (currently ~10k) to test if a denser or leaner representation helps.
- Trying different regularization grids for RidgeClassifier beyond the log-spaced defaults.
- Adding simple ensembles, like averaging predictions from multiple MiniRocket seeds, to reduce variance.

Even without these, the model already proved accurate and stable for the task.

Putting models side by side

We compared XGBoost and MiniRocket on validation and test sets, looking at both overall metrics and per-class results.

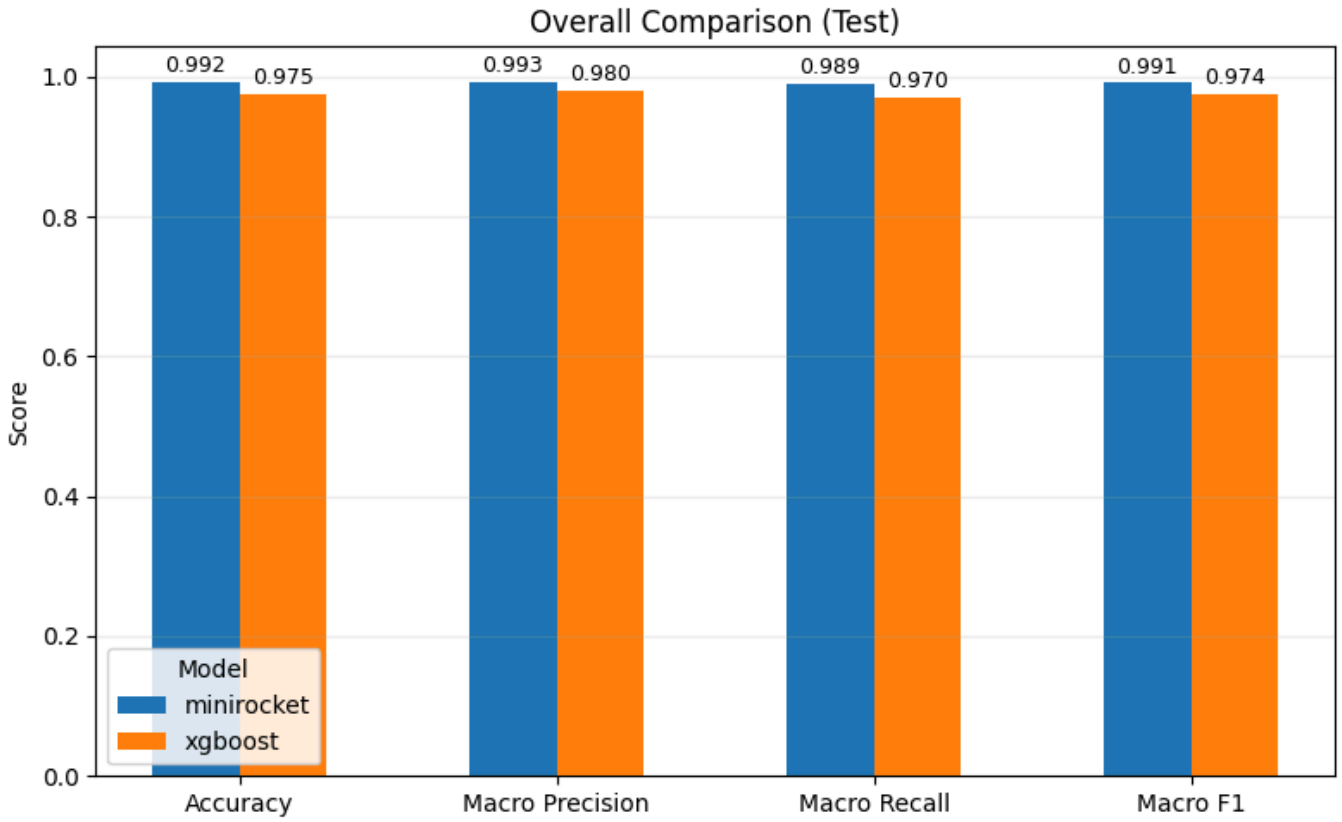


Figure 13 – Overall comparison on test set (accuracy, macro precision, recall, F1)

MiniRocket outperformed XGBoost across all metrics, with the largest gap in recall and F1.

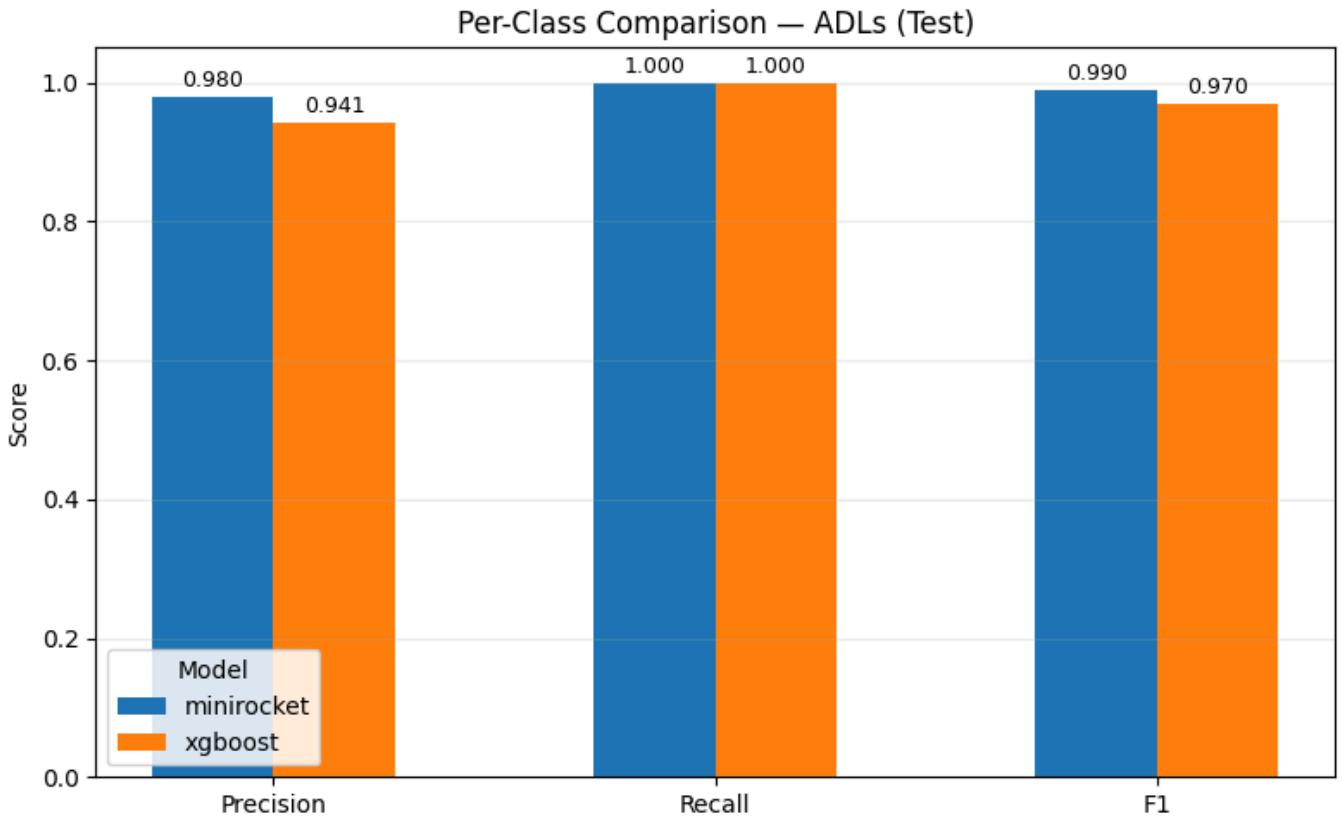


Figure 14 – Per-class comparison on test set (ADLs)

Both models recalled ADLs perfectly, but MiniRocket had higher precision and F1.

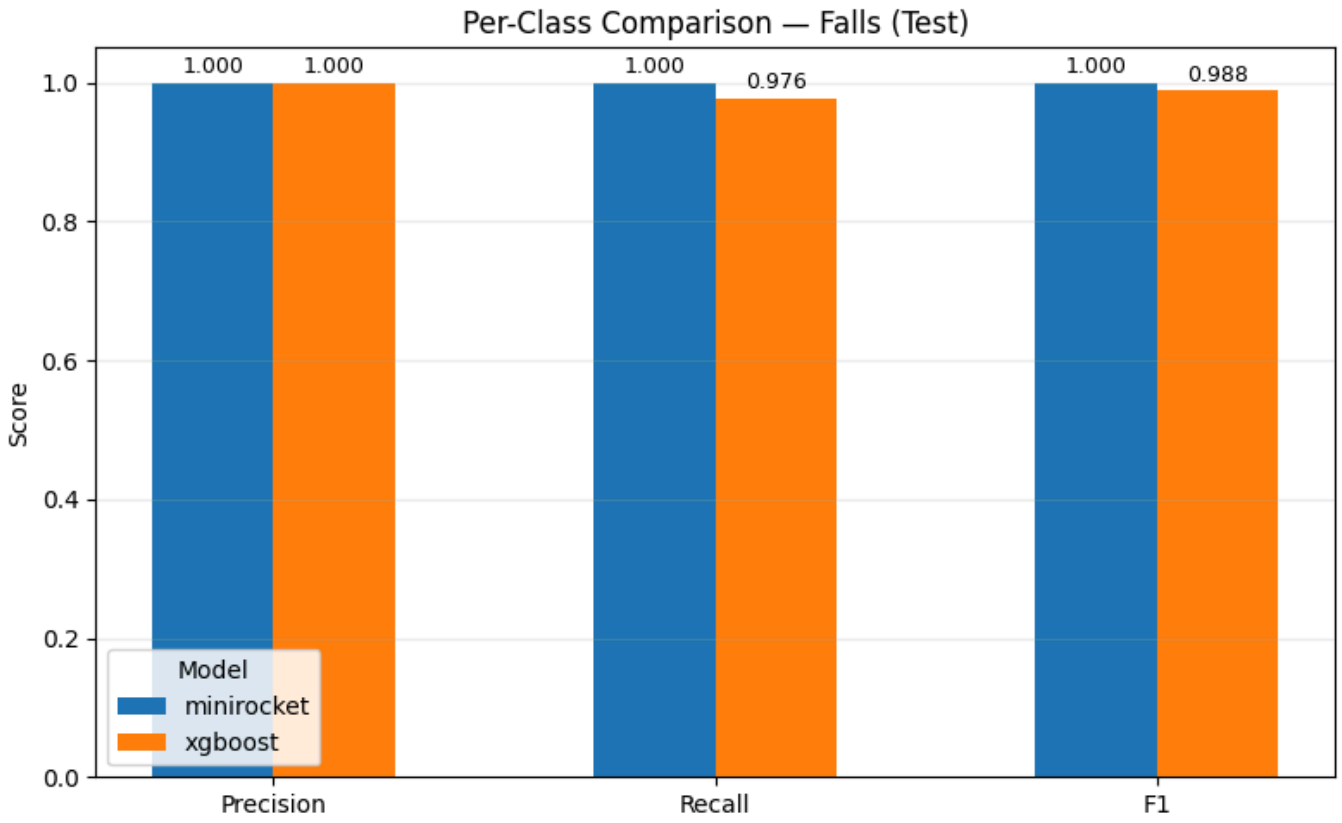


Figure 15 – Per-class comparison on test set (Falls)

Performance was near perfect for both models, though MiniRocket again showed slightly stronger recall.

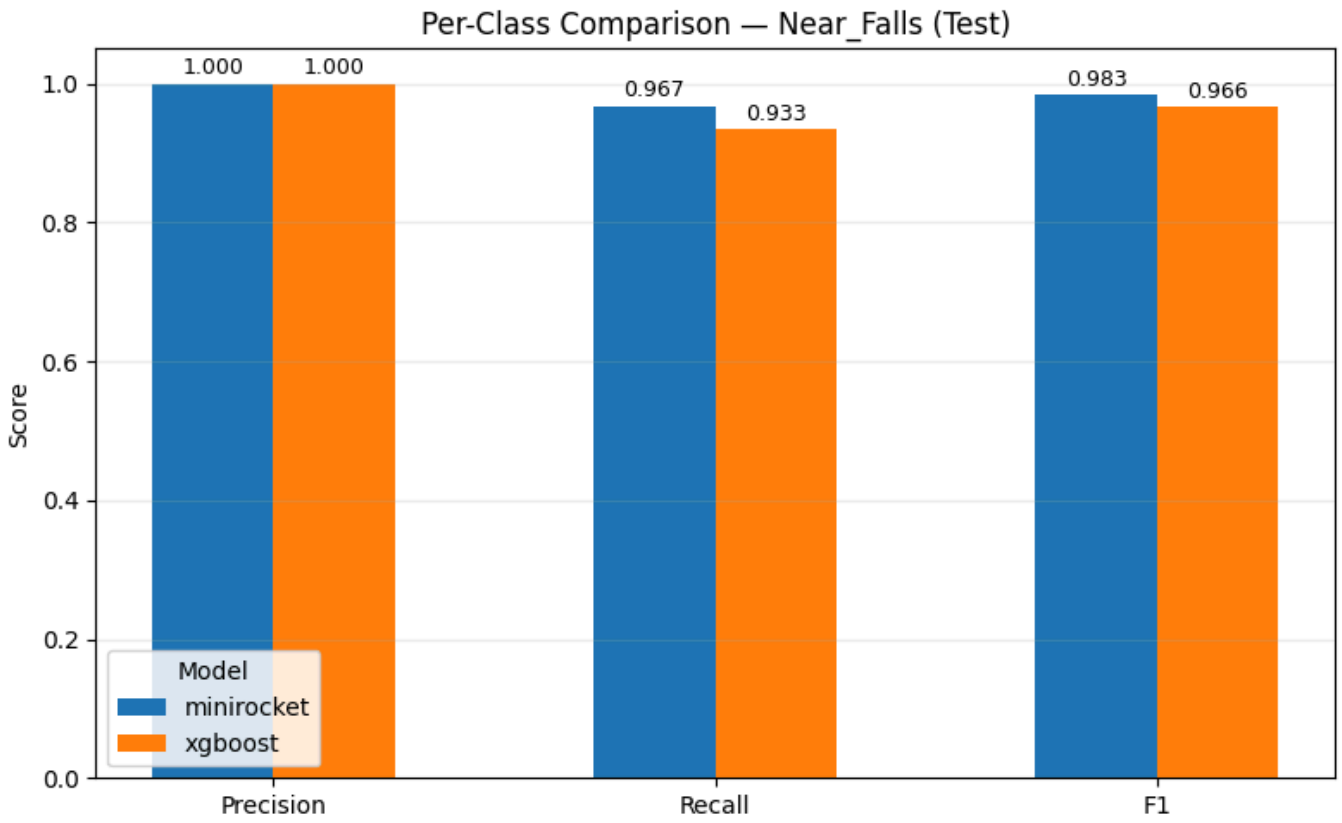


Figure 16 – Per-class comparison on test set (Near_Falls)

This was the hardest class for both models, but MiniRocket kept a higher recall and F1, showing better sensitivity to near-fall events.

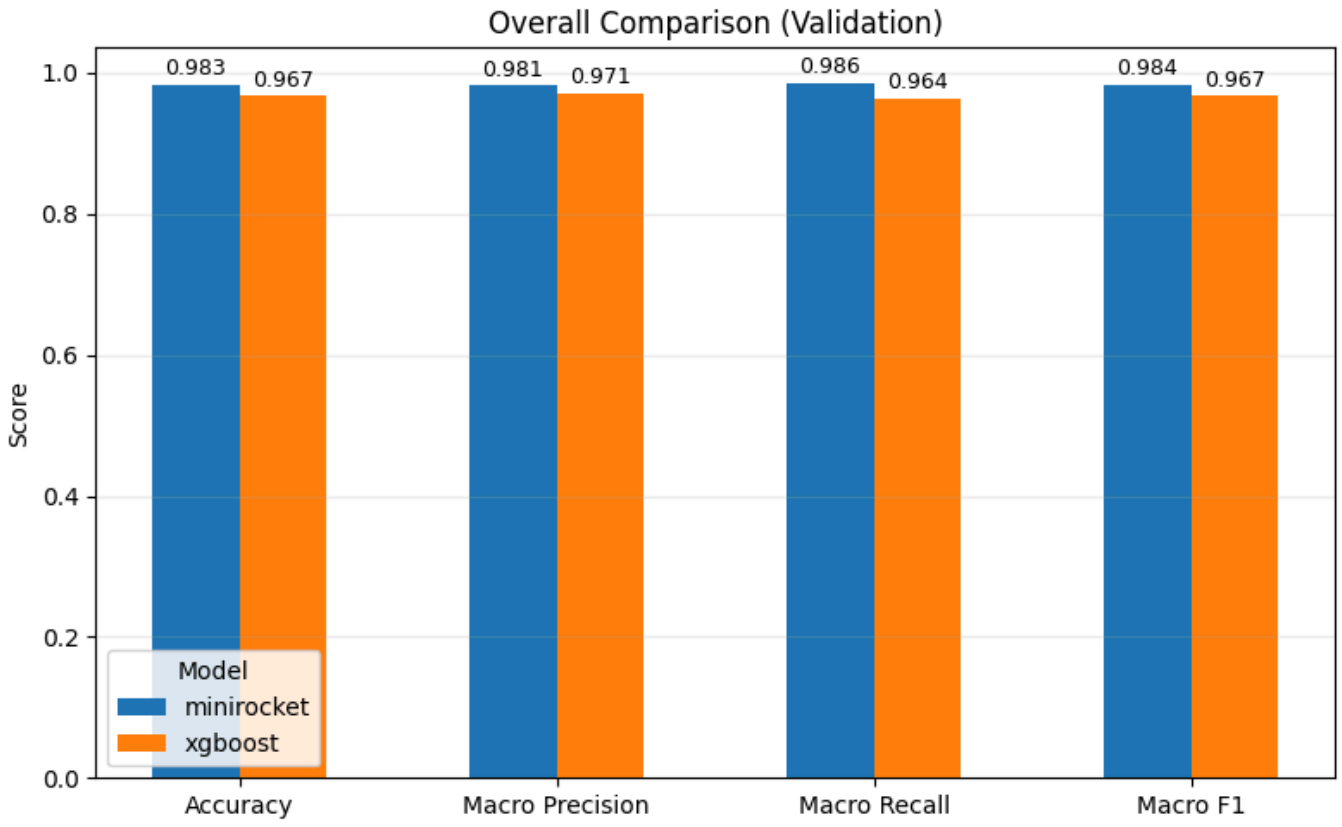


Figure 17 – Overall comparison on validation set (accuracy, macro precision, recall, F1)

The validation results followed the same trend as the test set: MiniRocket consistently a bit stronger.

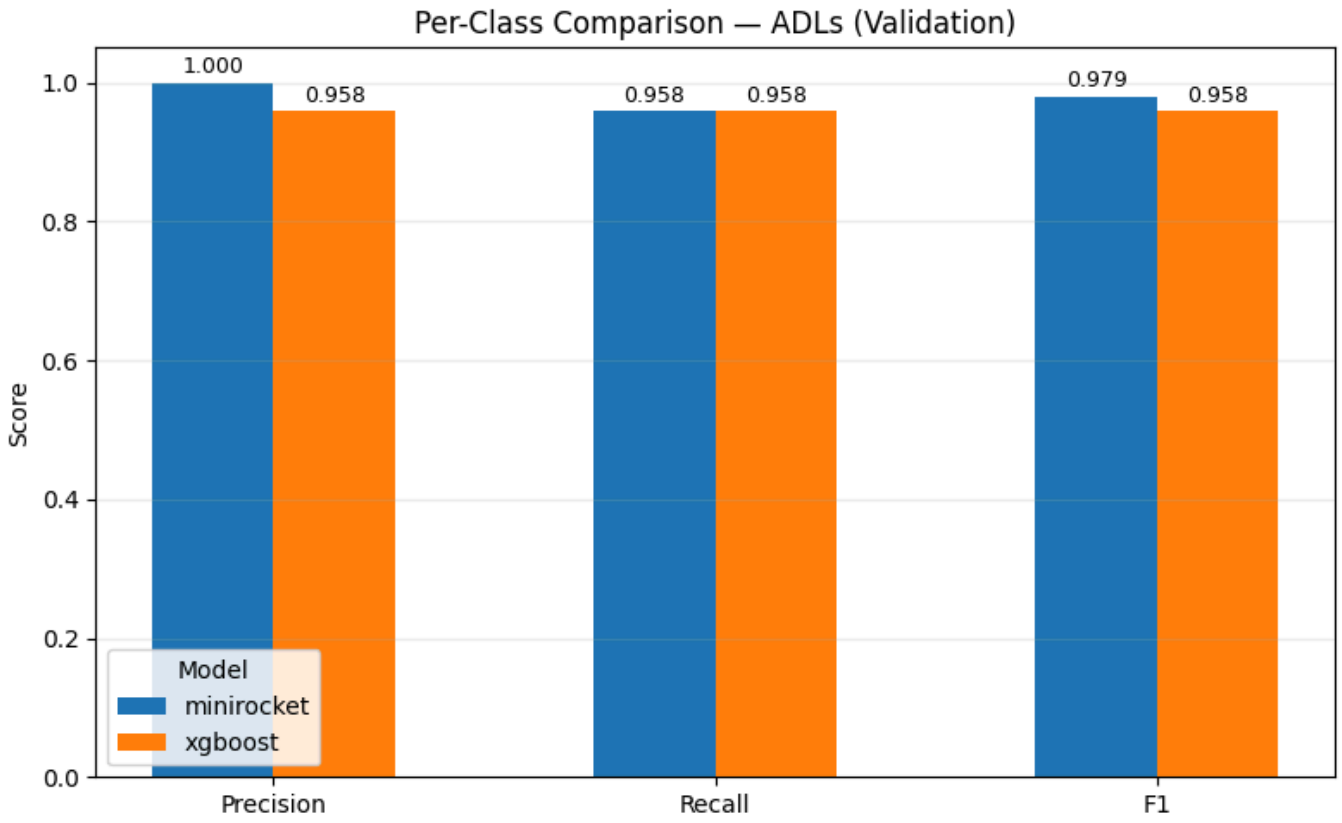


Figure 18 – Per-class comparison on validation set (ADLs)

Precision was noticeably better with MiniRocket, recall stayed the same.

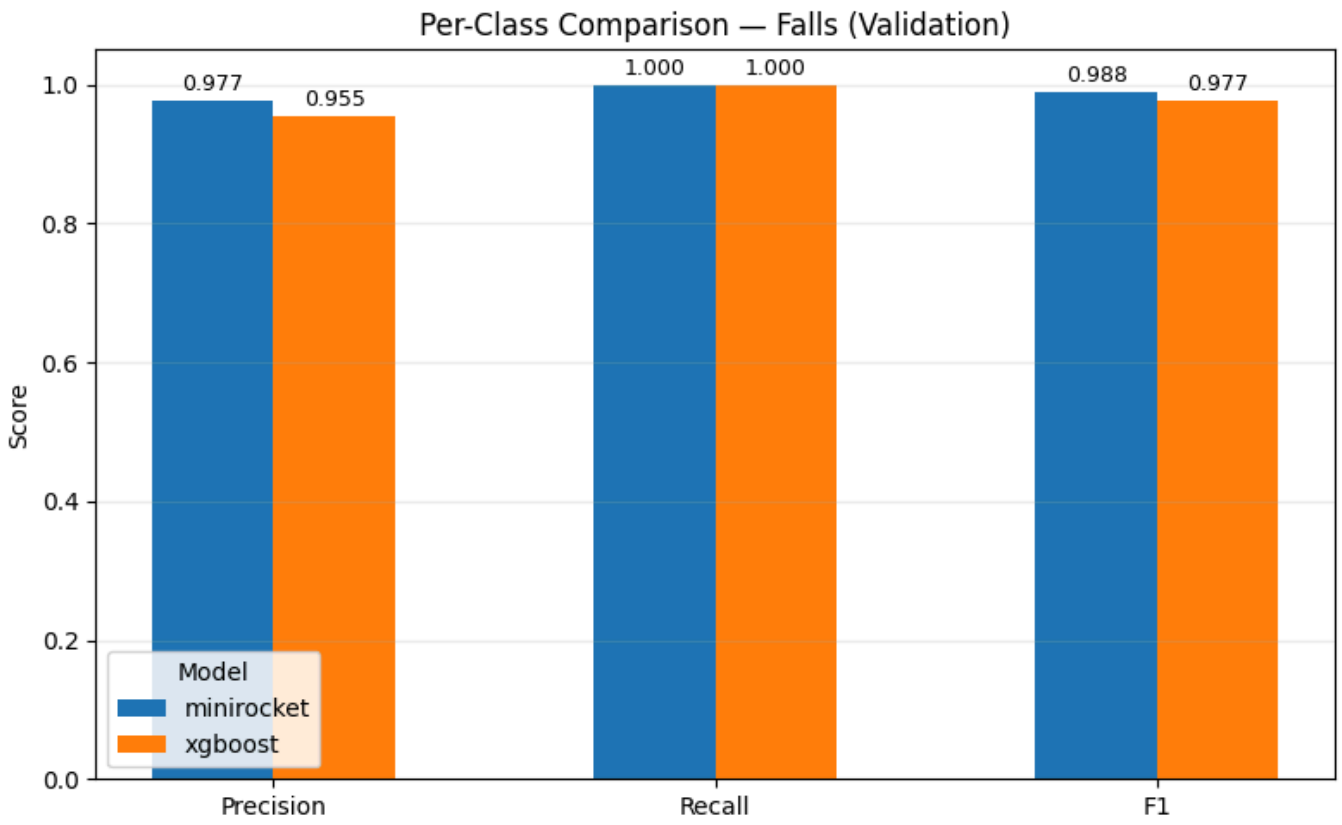


Figure 19 – Per-class comparison on validation set (Falls)

Both models performed very well, but MiniRocket again edged ahead slightly.

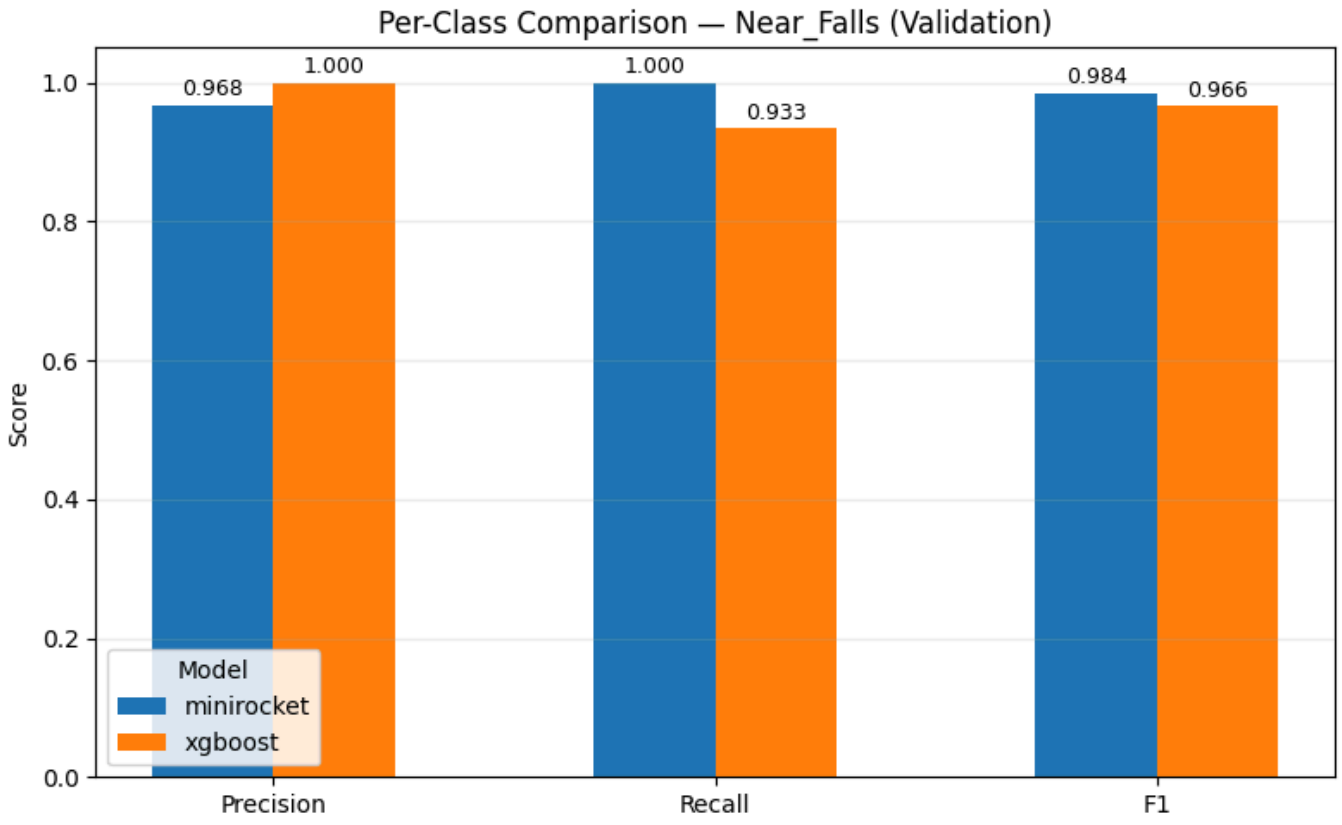


Figure 20 – Per-class comparison on validation set (Near_Falls)

This was again the toughest class. XGBoost dropped some recall here, while MiniRocket stayed higher.

Takeaway
Both models worked very well, but MiniRocket was consistently stronger across metrics and classes. The key difference is that XGBoost needed a lot of feature engineering, while MiniRocket reached better performance straight from the raw sequences.

End Notes

In this challenge we tested two different approaches for fall detection: XGBoost with engineered features and MiniRocket with RidgeClassifier.

Overall, MiniRocket performed better, reaching higher accuracy, precision, recall, and F1 on both validation and test sets. It also required less manual effort, since it worked directly on the raw time-series without feature engineering. On the other hand, XGBoost still showed strong results and gave us better explainability through feature importance, which is a useful advantage when interpretability matters.

Pros and cons

- XGBoost: pros are interpretability and solid performance; cons are the need for heavy feature engineering and bigger preprocessing effort.
- MiniRocket: pros are stronger metrics and simpler pipeline; cons are lower interpretability and less control over the generated features.

Next steps if we wanted to go further

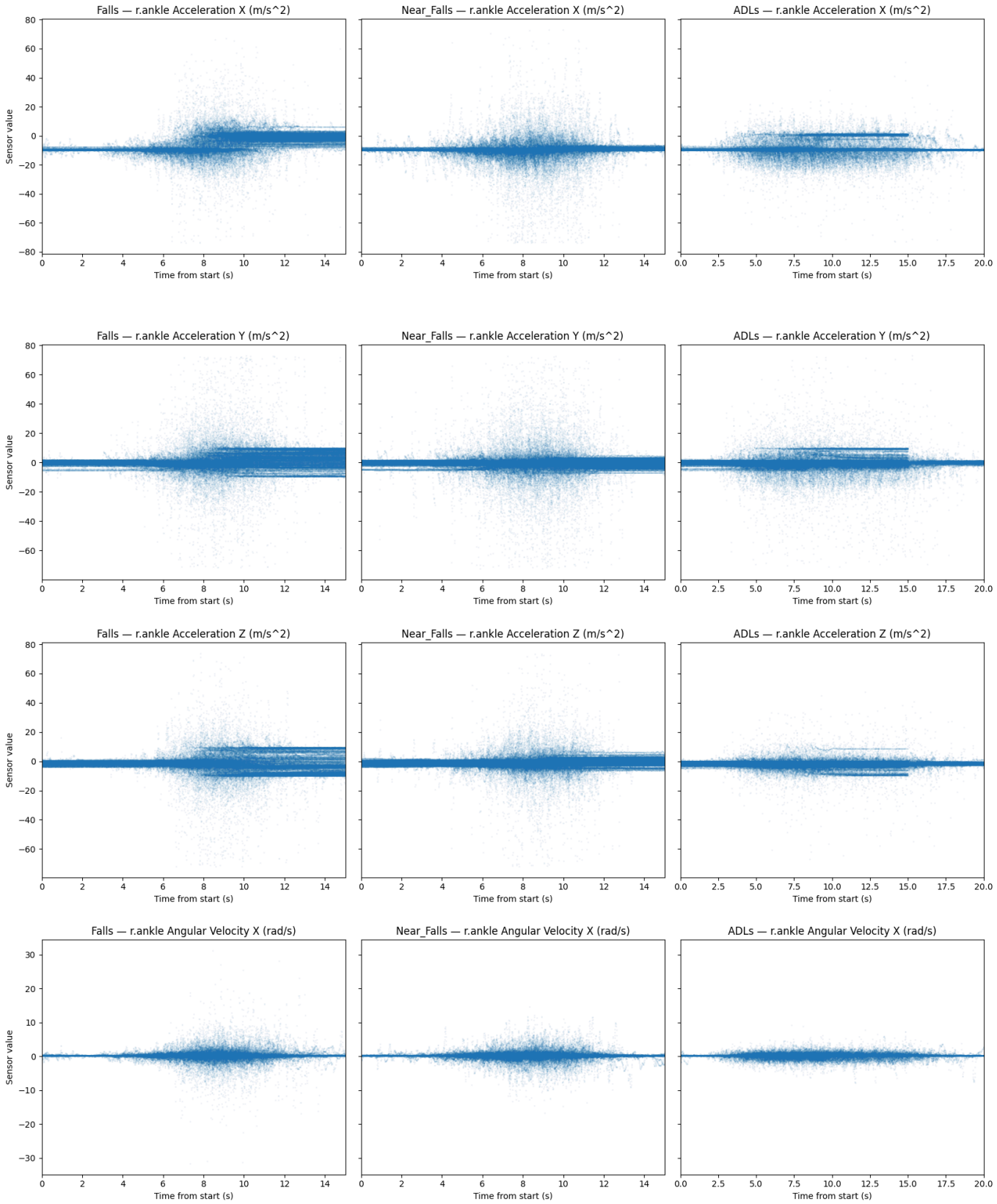
- Stress testing more systematically (e.g., noise, missing data, different window lengths).
- Fine-tuning MiniRocket parameters (number of features, seeds) and Ridge regularization.
- Trying simple ensembles of MiniRocket runs to reduce variance.
- Exploring other sequence models like InceptionTime, ROCKET variants, or lightweight CNNs.

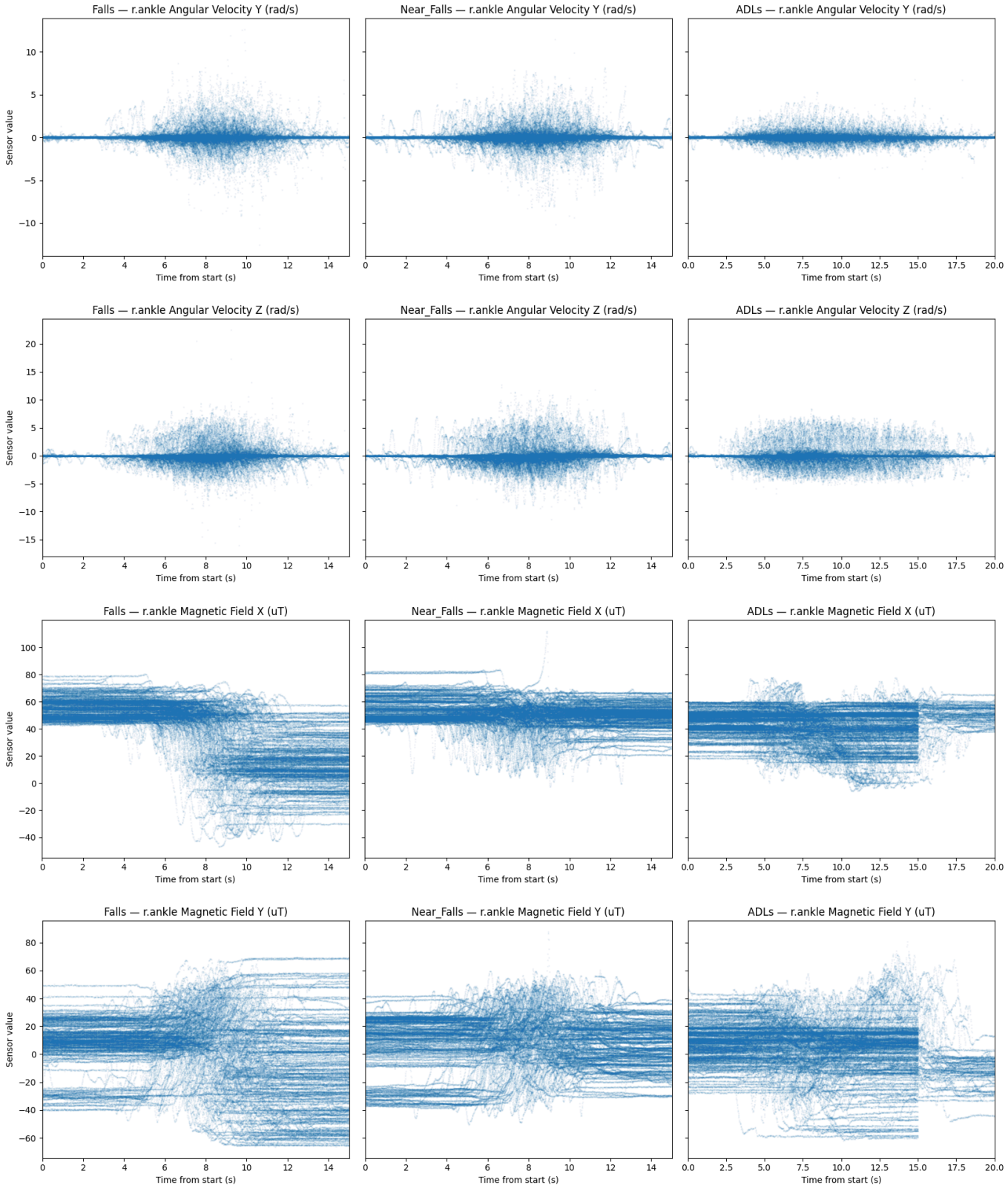
We also want to note that ChatGPT was used as a coding assistant and writing partner throughout this project. It helped draft parts of the preprocessing and training code, supported the writing of this report, and served as a sounding board when deciding which models to try.

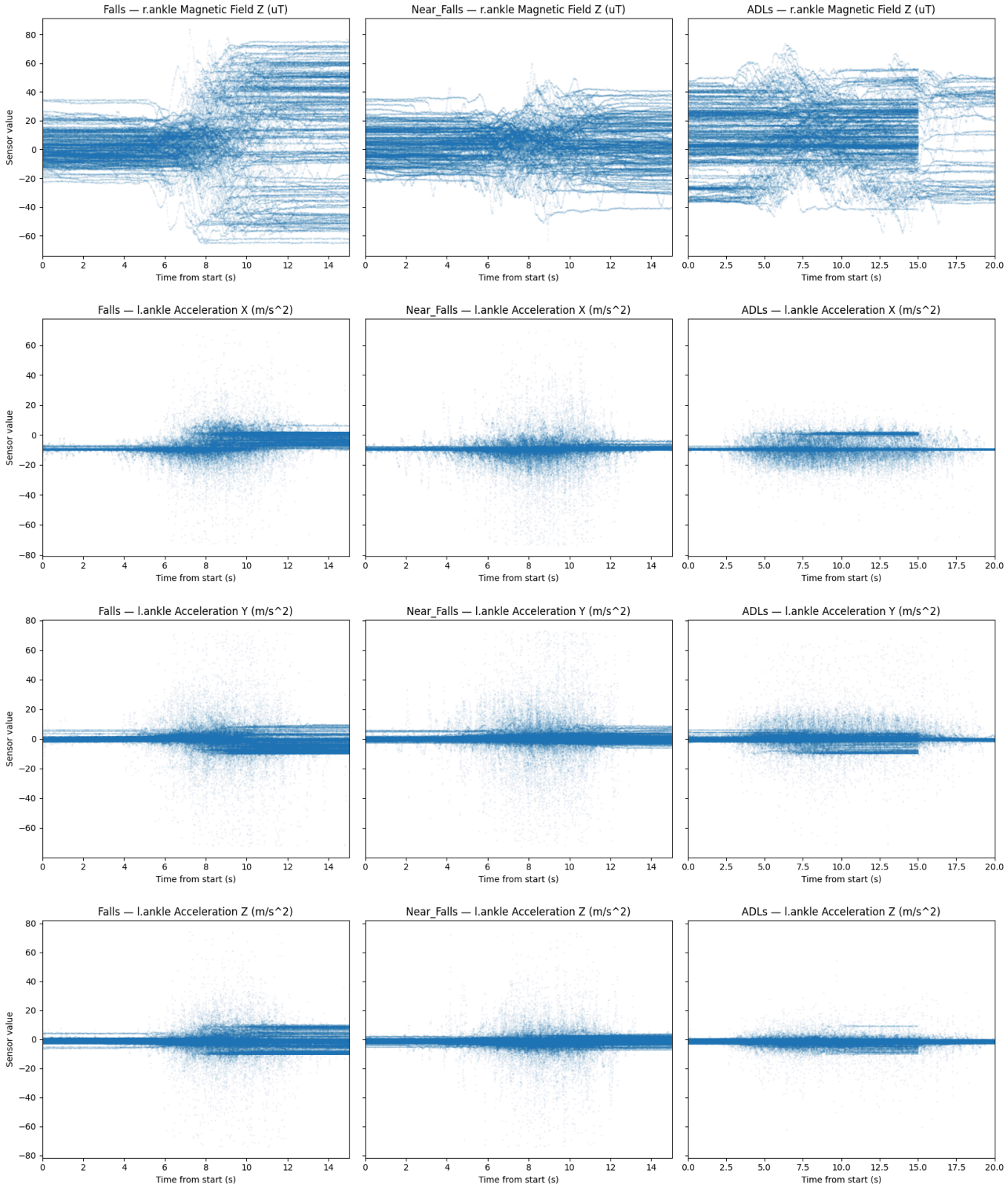
Both models proved robust and accurate, but MiniRocket stood out as the best balance of simplicity and performance. Given its resilience and minimal preprocessing needs, MiniRocket looks like a strong candidate for real-world fall detection systems, while XGBoost remains a valuable option when model interpretability is key.

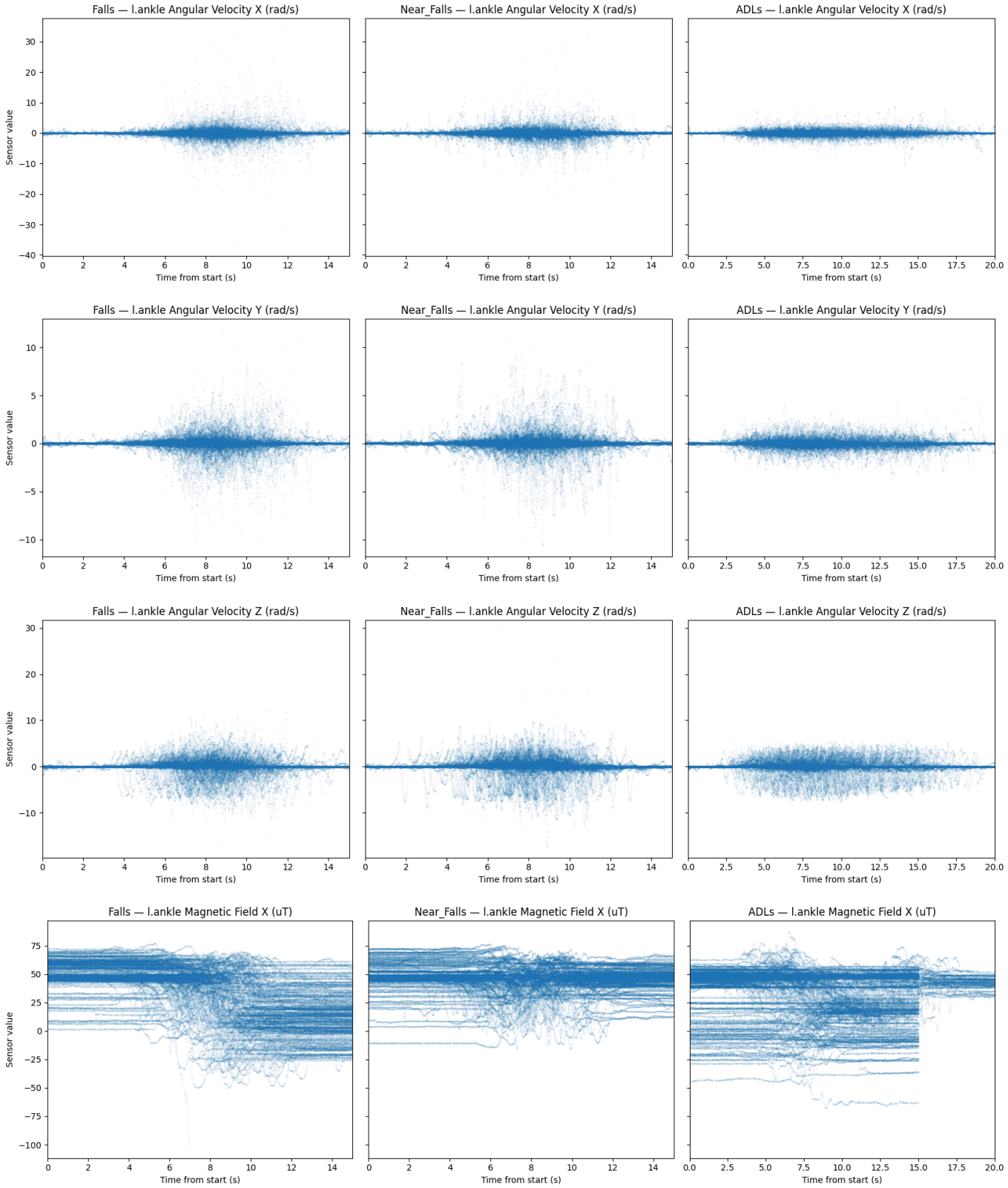
Appendix A

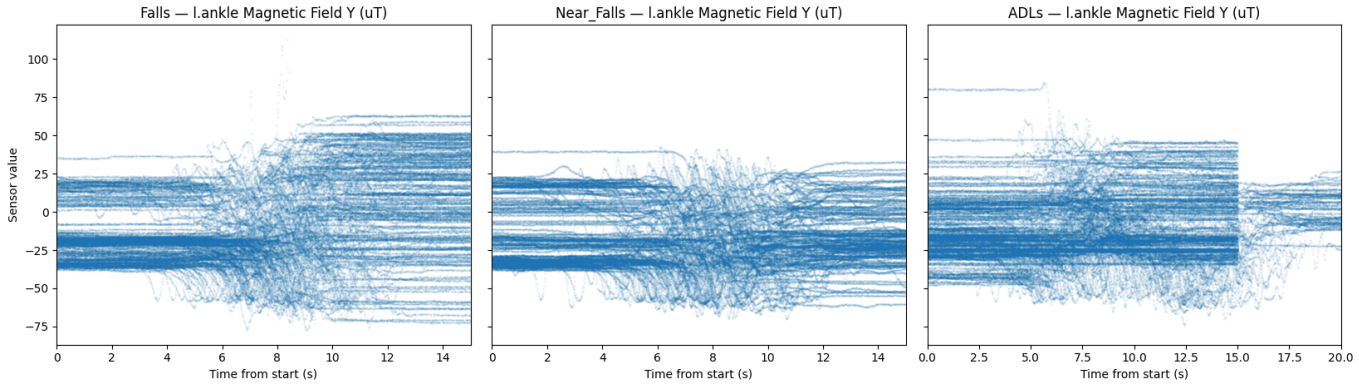
Sensor channels value over time from experiment start.

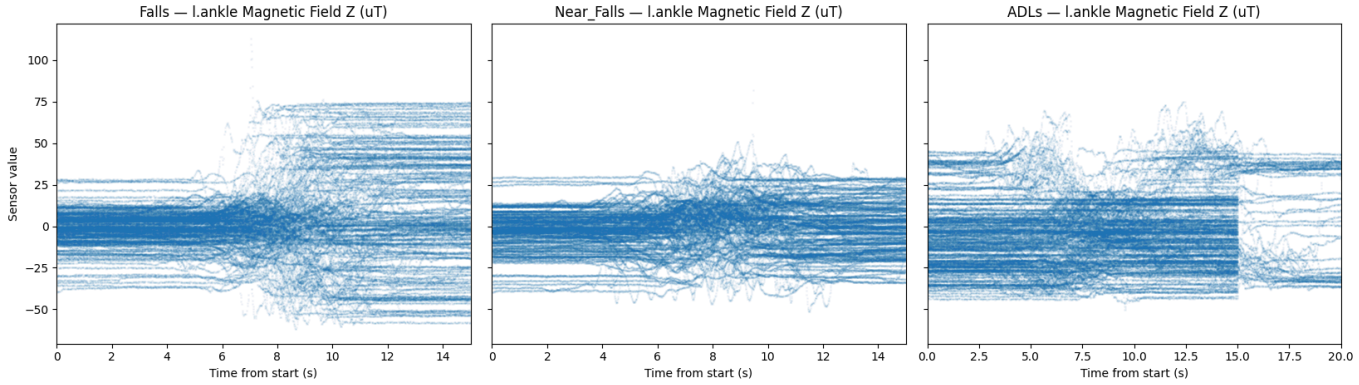


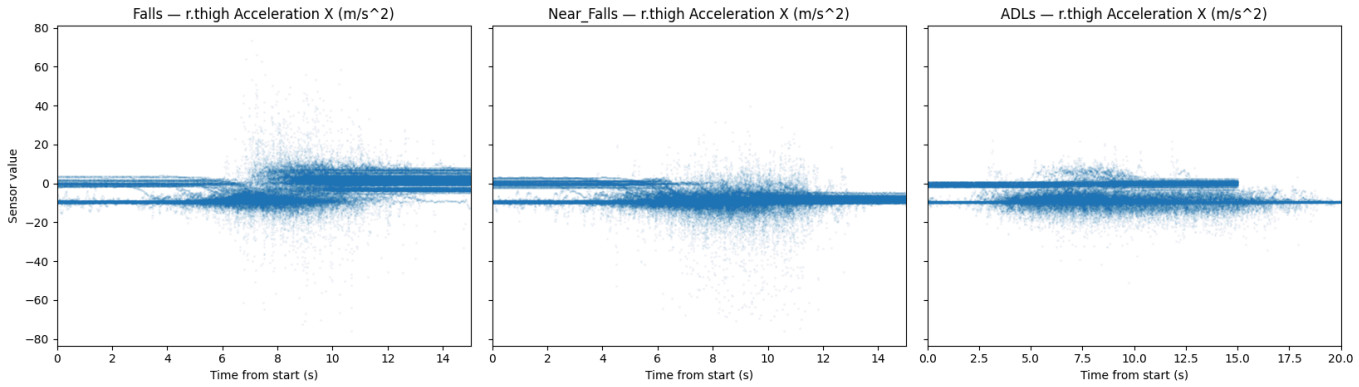

Falls — r.ankle Angular Velocity Y (rad/s)
Near_Falls — r.ankle Angular Velocity Y (rad/s)
ADLs — r.ankle Angular Velocity Y (rad/s)
Falls — r.ankle Angular Velocity Z (rad/s)
Near_Falls — r.ankle Angular Velocity Z (rad/s)
ADLs — r.ankle Angular Velocity Z (rad/s)
Falls — r.ankle Magnetic Field X (uT)
Near_Falls — r.ankle Magnetic Field X (uT)
ADLs — r.ankle Magnetic Field X (uT)
Falls — r.ankle Magnetic Field Y (uT)
Near_Falls — r.ankle Magnetic Field Y (uT)
ADLs — r.ankle Magnetic Field Y (uT)
Sensor value
Time from start (s)

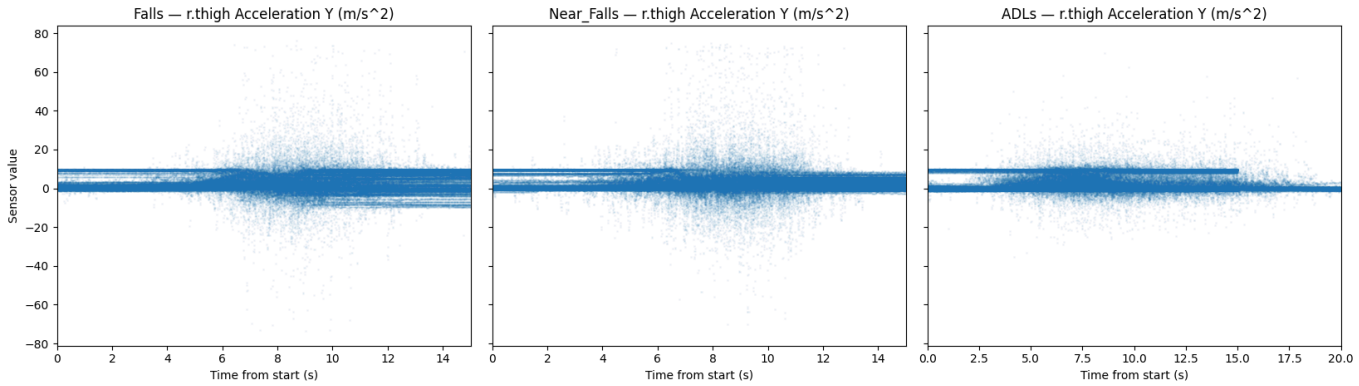

Falls — r.ankle Magnetic Field Z (uT)
Near_Falls — r.ankle Magnetic Field Z (uT)
ADLs — r.ankle Magnetic Field Z (uT)
Falls — l.ankle Acceleration X (m/s^2)
Near_Falls — l.ankle Acceleration X (m/s^2)
ADLs — l.ankle Acceleration X (m/s^2)
Falls — l.ankle Acceleration Y (m/s^2)
Near_Falls — l.ankle Acceleration Y (m/s^2)
ADLs — l.ankle Acceleration Y (m/s^2)
Falls — l.ankle Acceleration Z (m/s^2)
Near_Falls — l.ankle Acceleration Z (m/s^2)
ADLs — l.ankle Acceleration Z (m/s^2)
Sensor value
Time from start (s)

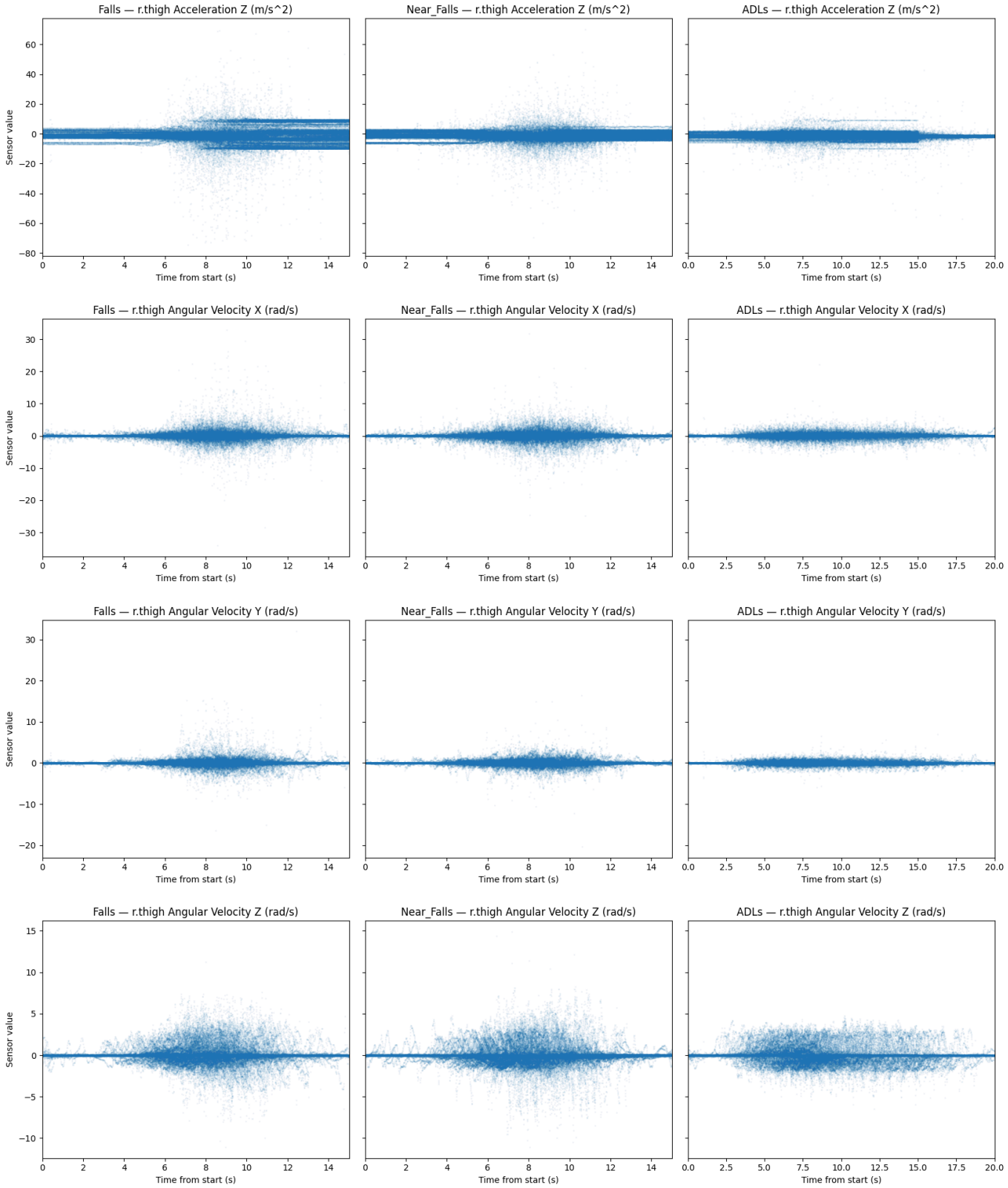

Falls — l.ankle Angular Velocity X (rad/s)
Near_Falls — l.ankle Angular Velocity X (rad/s)
ADLs — l.ankle Angular Velocity X (rad/s)
Falls — l.ankle Angular Velocity Y (rad/s)
Near_Falls — l.ankle Angular Velocity Y (rad/s)
ADLs — l.ankle Angular Velocity Y (rad/s)
Falls — l.ankle Angular Velocity Z (rad/s)
Near_Falls — l.ankle Angular Velocity Z (rad/s)
ADLs — l.ankle Angular Velocity Z (rad/s)
Falls — l.ankle Magnetic Field X (uT)
Near_Falls — l.ankle Magnetic Field X (uT)
ADLs — l.ankle Magnetic Field X (uT)
Sensor value
Time from start (s)

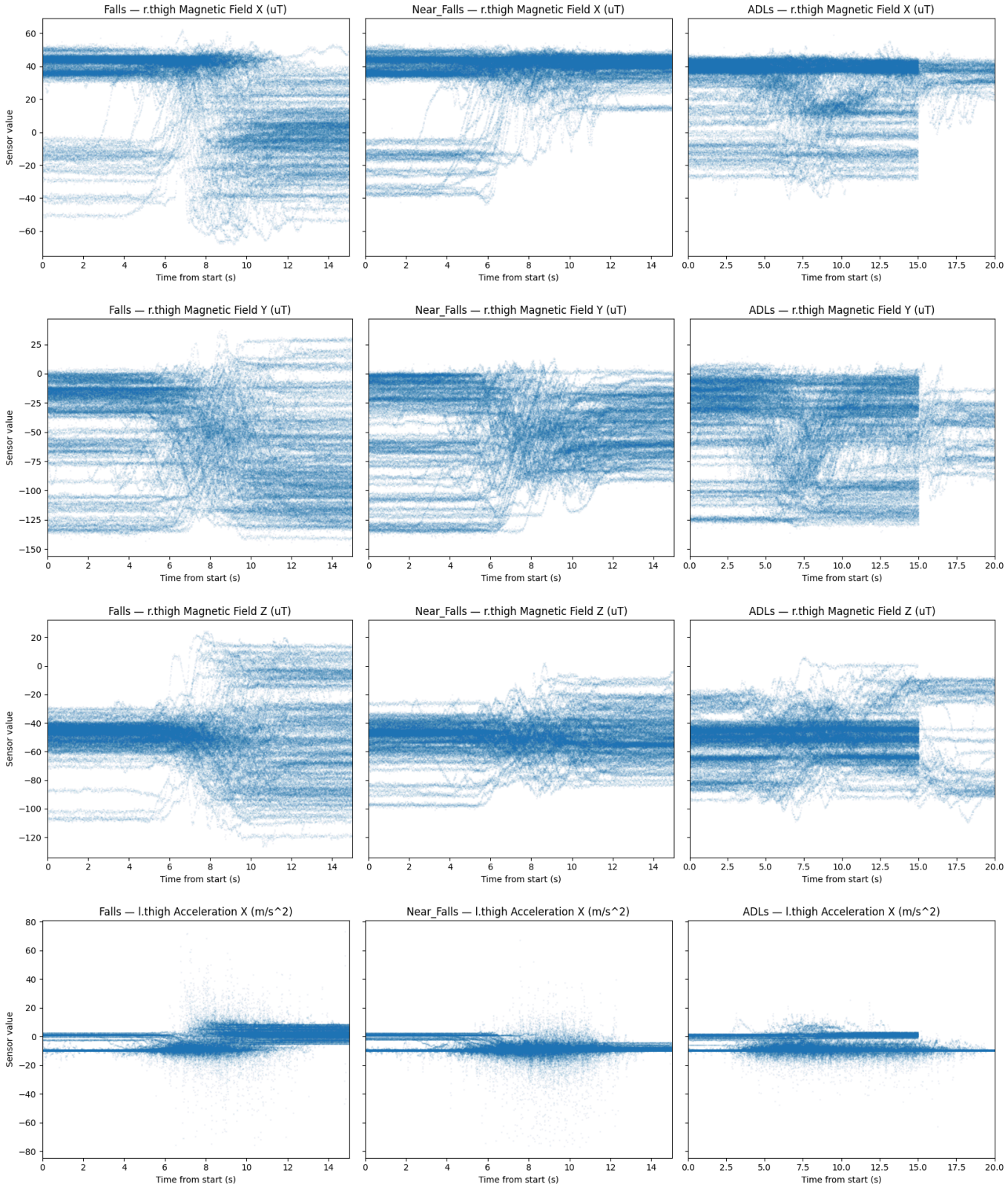

Falls — l.ankle Magnetic Field Y (uT)
Near_Falls — l.ankle Magnetic Field Y (uT)
ADLs — l.ankle Magnetic Field Y (uT)
Sensor value
Time from start (s)

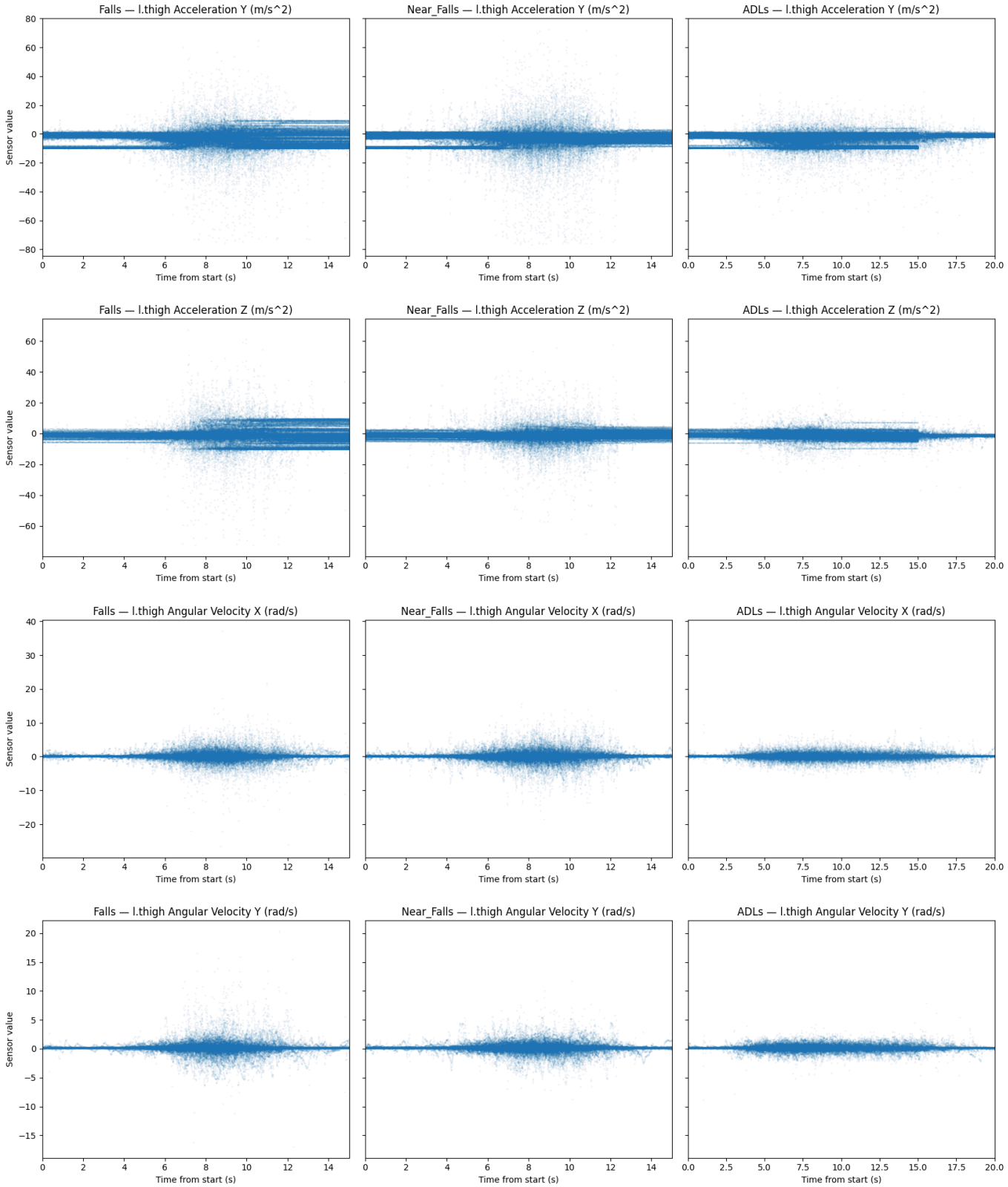

Falls — l.ankle Magnetic Field Z (uT)
Near_Falls — l.ankle Magnetic Field Z (uT)
ADLs — l.ankle Magnetic Field Z (uT)
Sensor value
Time from start (s)

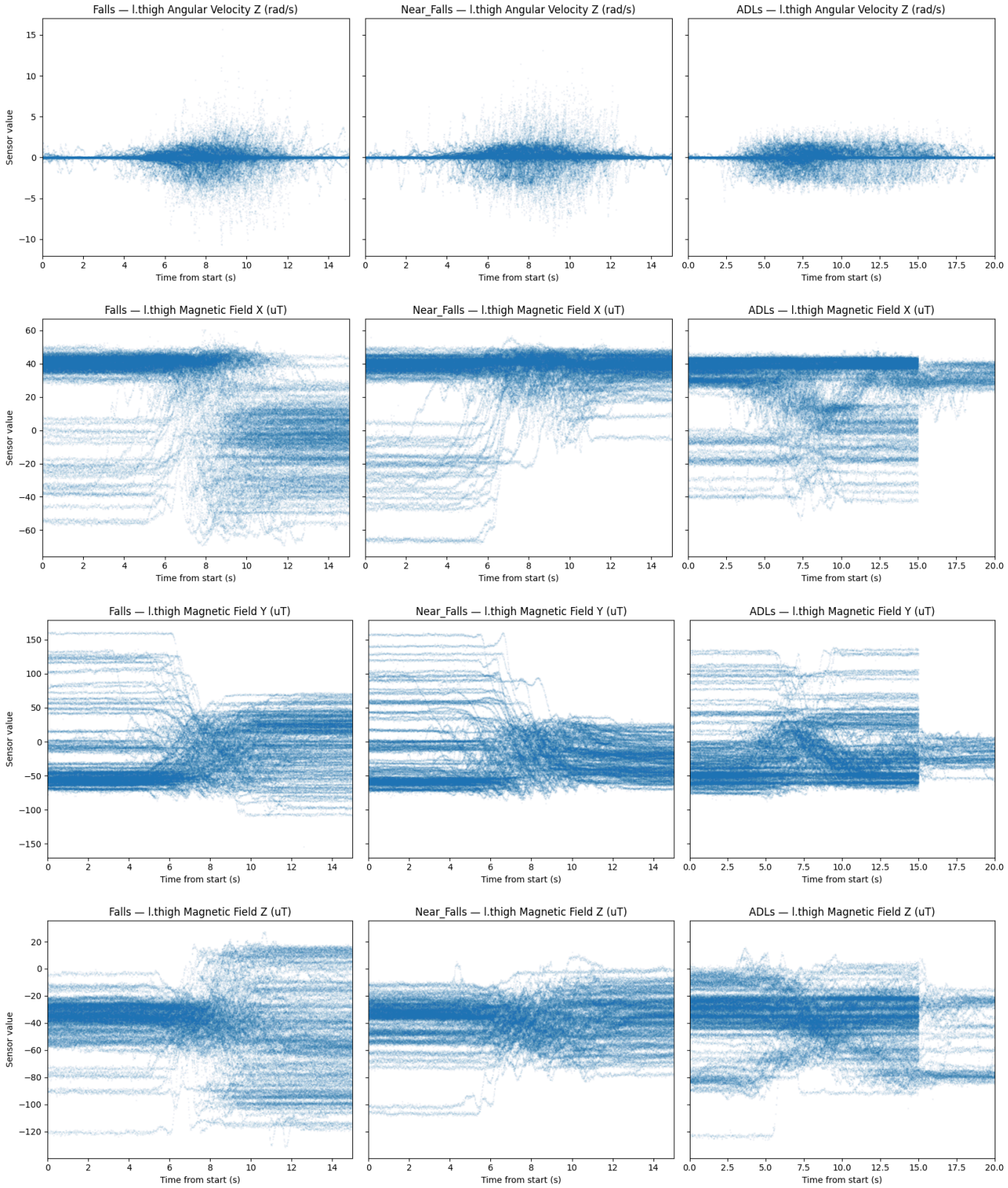

Falls — r.thigh Acceleration X (m/s^2)
Near_Falls — r.thigh Acceleration X (m/s^2)
ADLs — r.thigh Acceleration X (m/s^2)
Sensor value
Time from start (s)

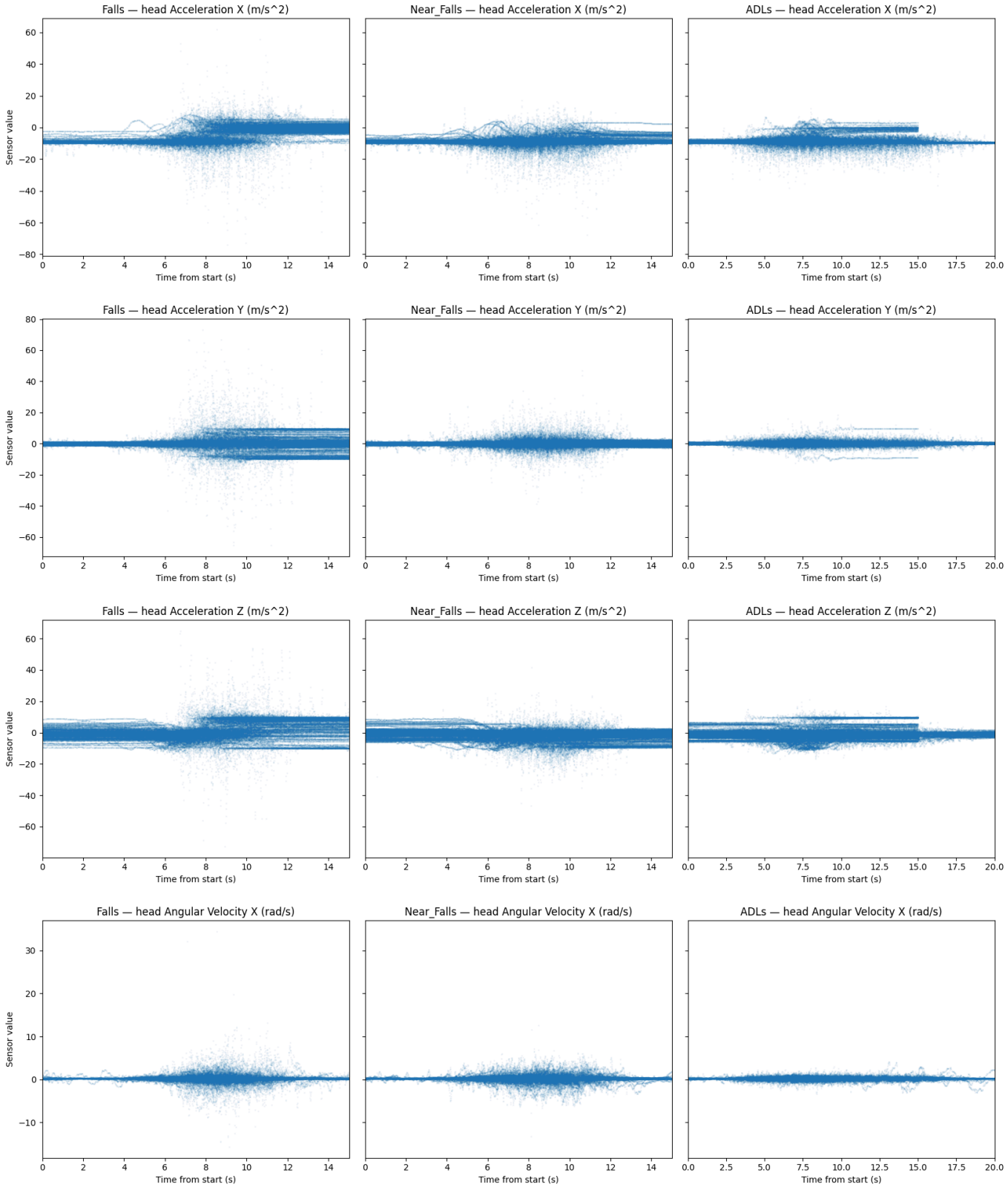

Falls — r.thigh Acceleration Y (m/s^2)
Near_Falls — r.thigh Acceleration Y (m/s^2)
ADLs — r.thigh Acceleration Y (m/s^2)
Sensor value
Time from start (s)

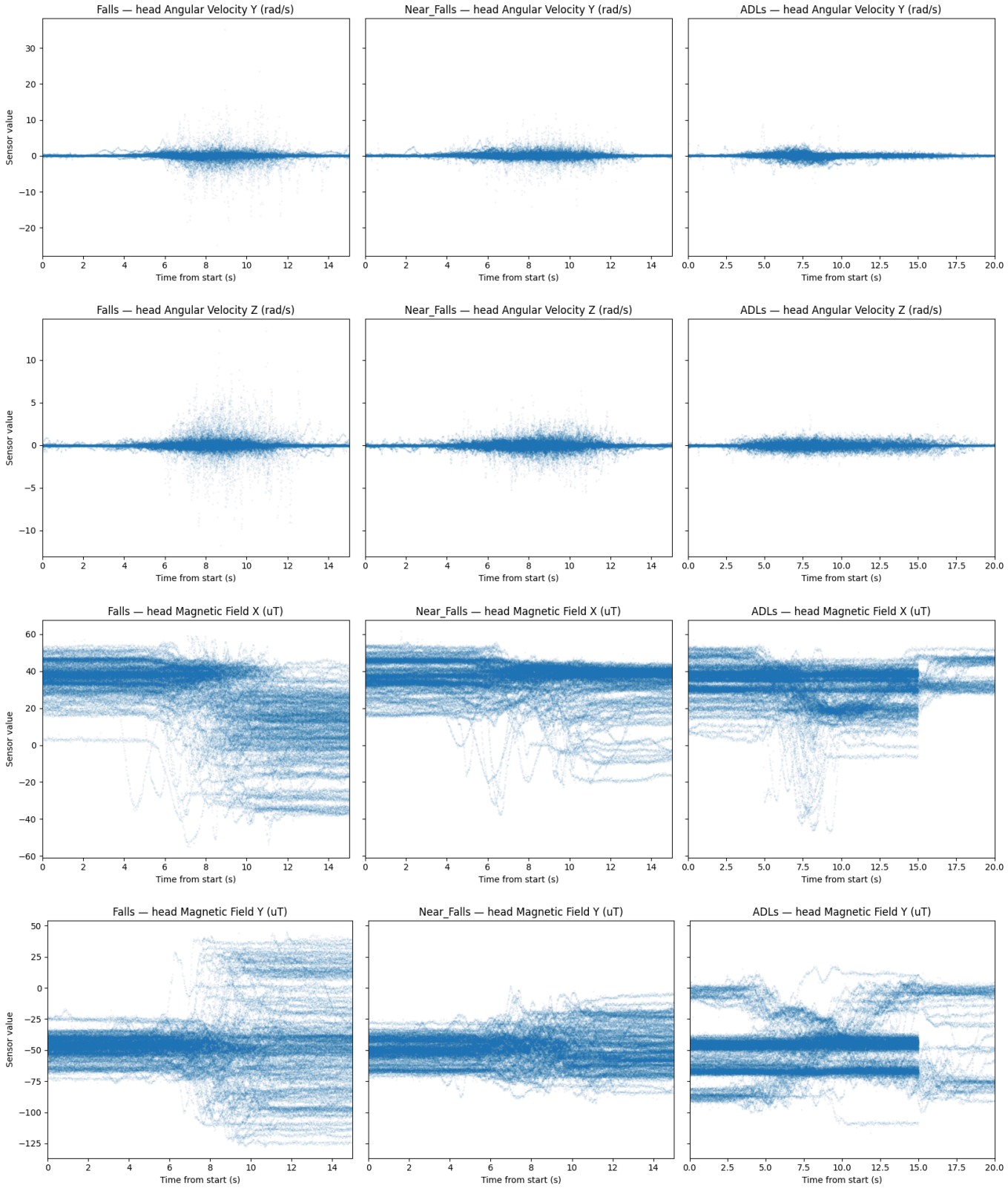

Falls — r.thigh Acceleration Z (m/s^2)
Near_Falls — r.thigh Acceleration Z (m/s^2)
ADLs — r.thigh Acceleration Z (m/s^2)
Falls — r.thigh Angular Velocity X (rad/s)
Near_Falls — r.thigh Angular Velocity X (rad/s)
ADLs — r.thigh Angular Velocity X (rad/s)
Falls — r.thigh Angular Velocity Y (rad/s)
Near_Falls — r.thigh Angular Velocity Y (rad/s)
ADLs — r.thigh Angular Velocity Y (rad/s)
Falls — r.thigh Angular Velocity Z (rad/s)
Near_Falls — r.thigh Angular Velocity Z (rad/s)
ADLs — r.thigh Angular Velocity Z (rad/s)
Sensor value
Time from start (s)

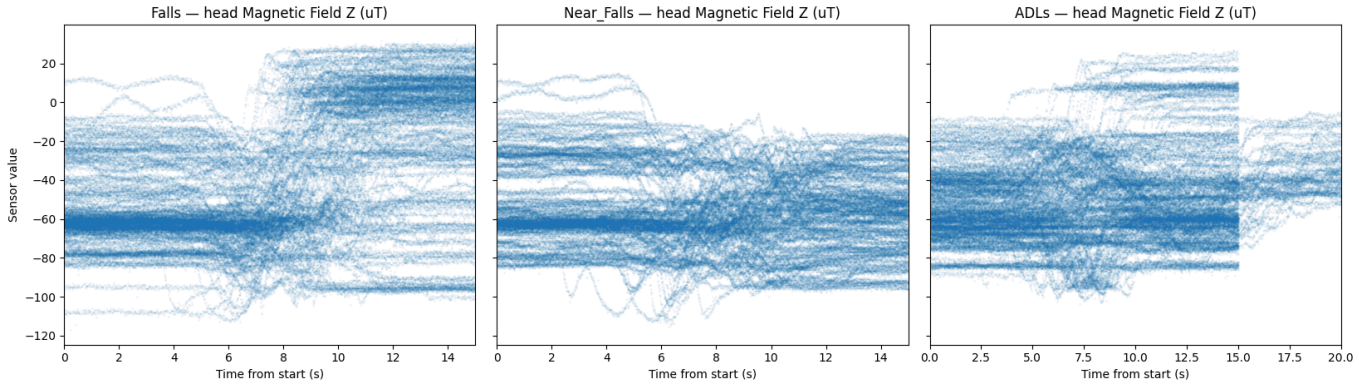

Falls — r.thigh Magnetic Field X (uT)
Near_Falls — r.thigh Magnetic Field X (uT)
ADLs — r.thigh Magnetic Field X (uT)
Falls — r.thigh Magnetic Field Y (uT)
Near_Falls — r.thigh Magnetic Field Y (uT)
ADLs — r.thigh Magnetic Field Y (uT)
Falls — r.thigh Magnetic Field Z (uT)
Near_Falls — r.thigh Magnetic Field Z (uT)
ADLs — r.thigh Magnetic Field Z (uT)
Falls — l.thigh Acceleration X (m/s^2)
Near_Falls — l.thigh Acceleration X (m/s^2)
ADLs — l.thigh Acceleration X (m/s^2)
Sensor value
Time from start (s)

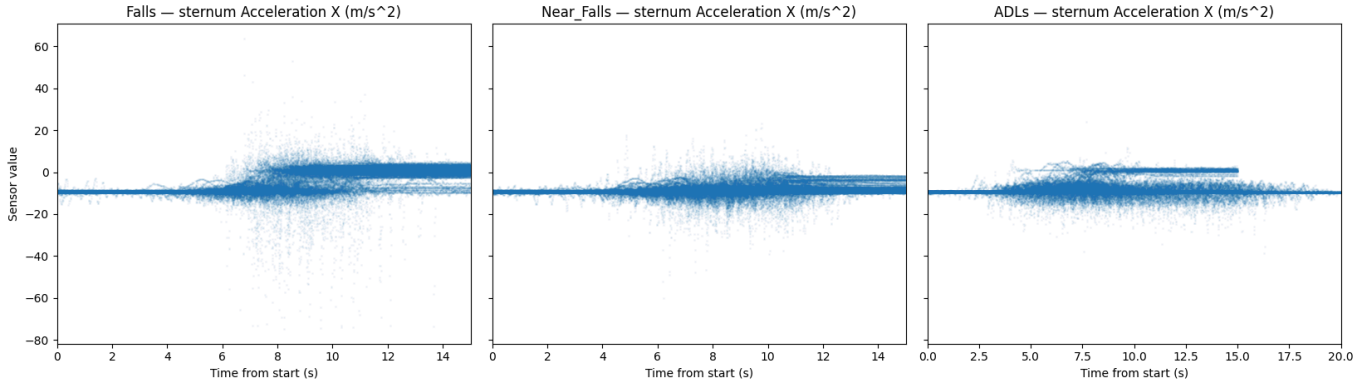

Falls — l.thigh Acceleration Y (m/s^2)
Near_Falls — l.thigh Acceleration Y (m/s^2)
ADLs — l.thigh Acceleration Y (m/s^2)
Falls — l.thigh Acceleration Z (m/s^2)
Near_Falls — l.thigh Acceleration Z (m/s^2)
ADLs — l.thigh Acceleration Z (m/s^2)
Falls — l.thigh Angular Velocity X (rad/s)
Near_Falls — l.thigh Angular Velocity X (rad/s)
ADLs — l.thigh Angular Velocity X (rad/s)
Falls — l.thigh Angular Velocity Y (rad/s)
Near_Falls — l.thigh Angular Velocity Y (rad/s)
ADLs — l.thigh Angular Velocity Y (rad/s)
Sensor value
Time from start (s)

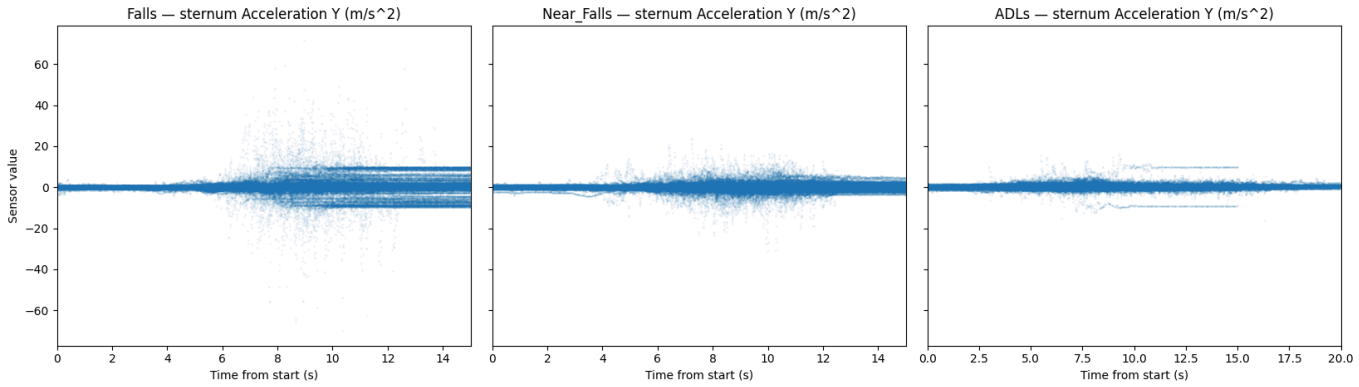

Falls — l.thigh Angular Velocity Z (rad/s)
Near_Falls — l.thigh Angular Velocity Z (rad/s)
ADLs — l.thigh Angular Velocity Z (rad/s)
Falls — l.thigh Magnetic Field X (uT)
Near_Falls — l.thigh Magnetic Field X (uT)
ADLs — l.thigh Magnetic Field X (uT)
Falls — l.thigh Magnetic Field Y (uT)
Near_Falls — l.thigh Magnetic Field Y (uT)
ADLs — l.thigh Magnetic Field Y (uT)
Falls — l.thigh Magnetic Field Z (uT)
Near_Falls — l.thigh Magnetic Field Z (uT)
ADLs — l.thigh Magnetic Field Z (uT)
Sensor value
Time from start (s)

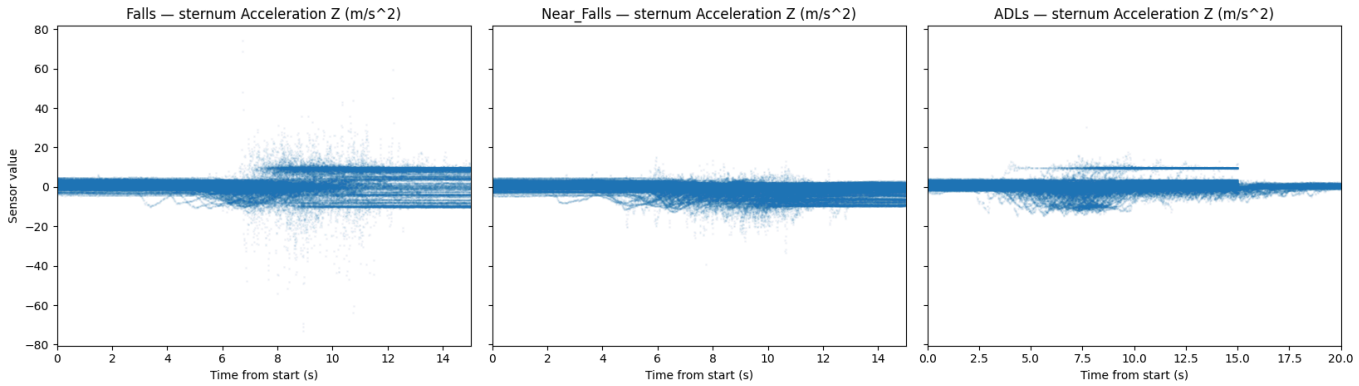

Falls — head Acceleration X (m/s^2)
Near_Falls — head Acceleration X (m/s^2)
ADLs — head Acceleration X (m/s^2)
Falls — head Acceleration Y (m/s^2)
Near_Falls — head Acceleration Y (m/s^2)
ADLs — head Acceleration Y (m/s^2)
Falls — head Acceleration Z (m/s^2)
Near_Falls — head Acceleration Z (m/s^2)
ADLs — head Acceleration Z (m/s^2)
Falls — head Angular Velocity X (rad/s)
Near_Falls — head Angular Velocity X (rad/s)
ADLs — head Angular Velocity X (rad/s)
Sensor value
Time from start (s)

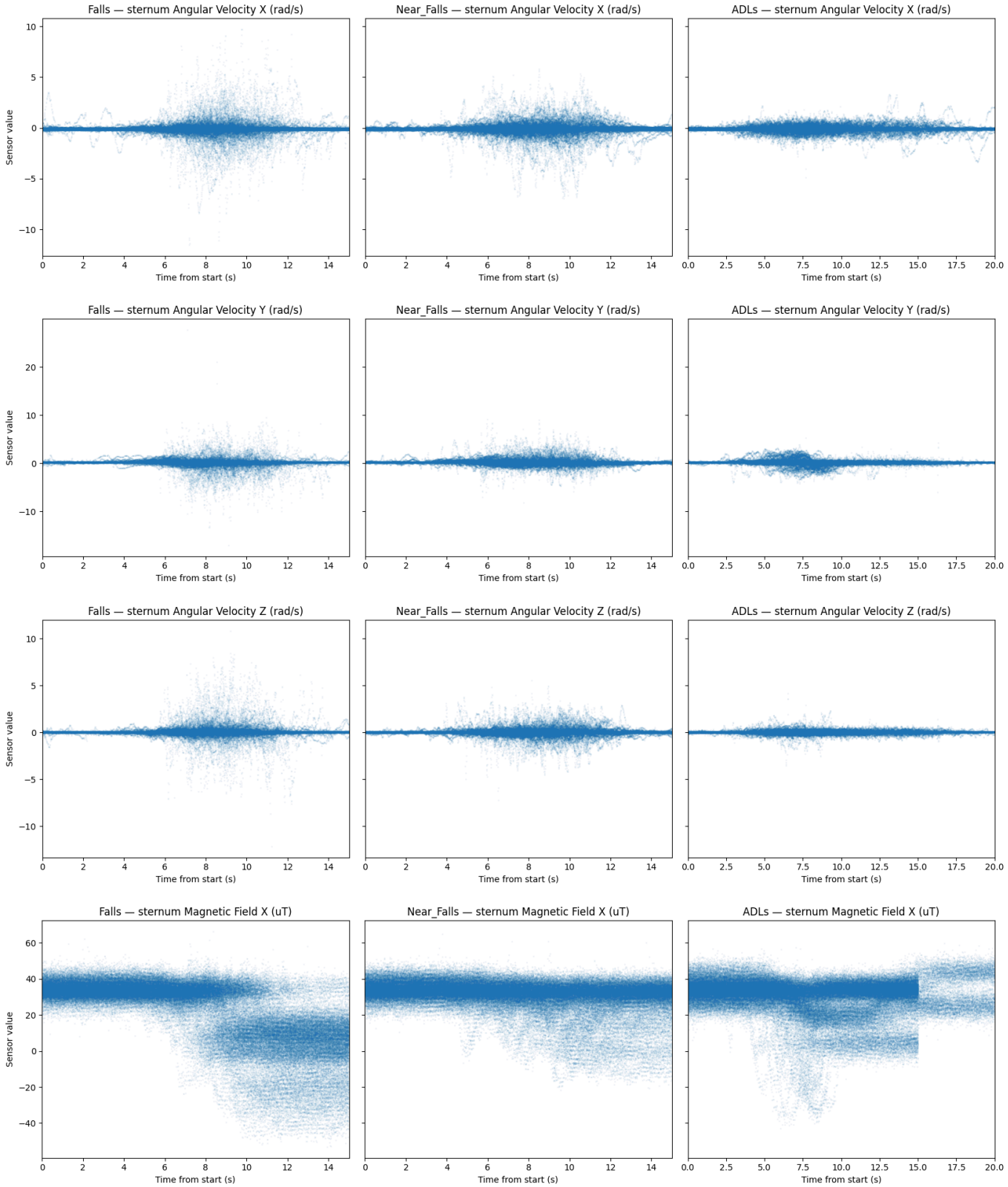

Falls — head Angular Velocity Y (rad/s)
Near_Falls — head Angular Velocity Y (rad/s)
ADLs — head Angular Velocity Y (rad/s)
Falls — head Angular Velocity Z (rad/s)
Near_Falls — head Angular Velocity Z (rad/s)
ADLs — head Angular Velocity Z (rad/s)
Falls — head Magnetic Field X (uT)
Near_Falls — head Magnetic Field X (uT)
ADLs — head Magnetic Field X (uT)
Falls — head Magnetic Field Y (uT)
Near_Falls — head Magnetic Field Y (uT)
ADLs — head Magnetic Field Y (uT)
Sensor value
Time from start (s)

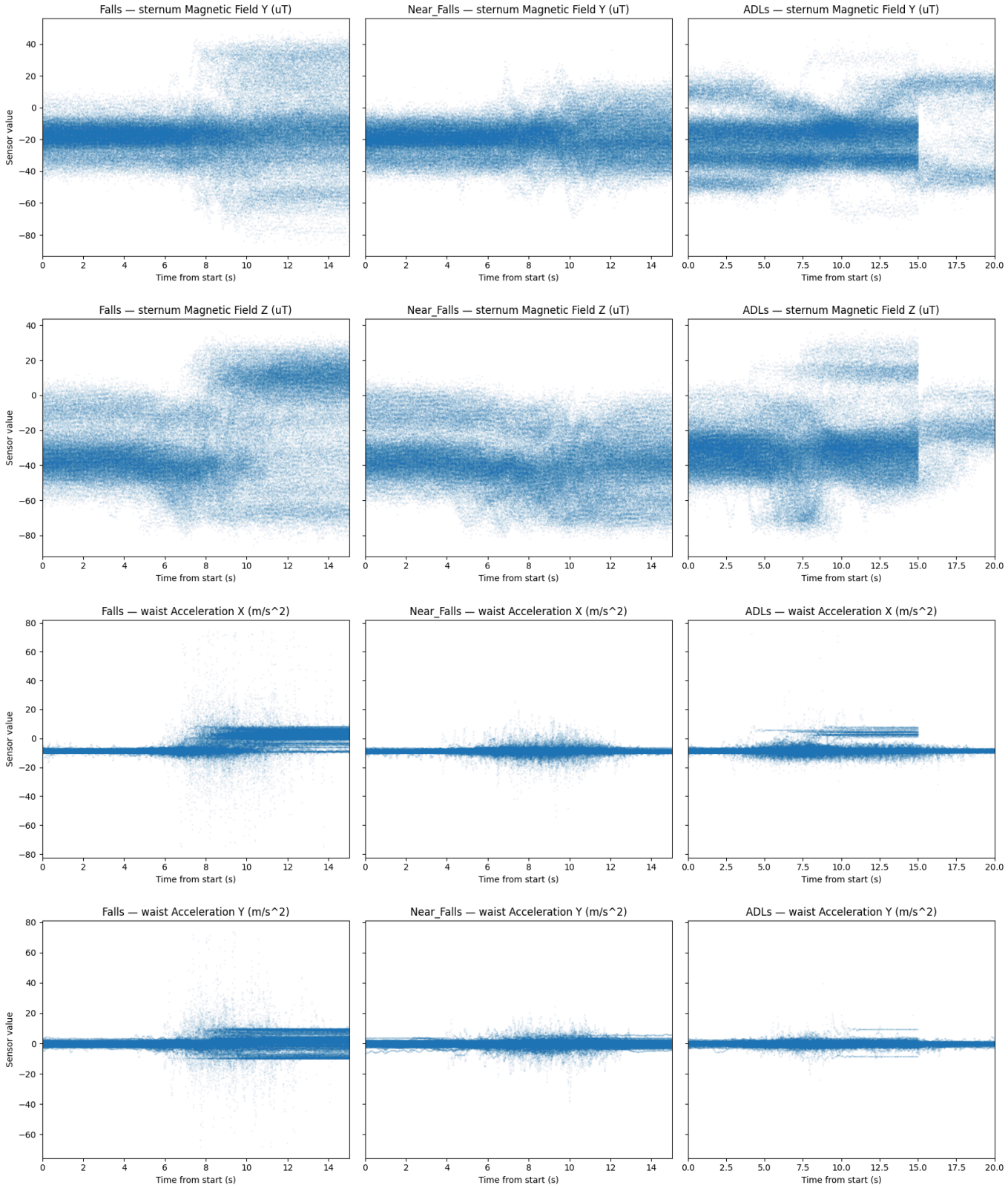

Falls — head Magnetic Field Z (uT)
Near_Falls — head Magnetic Field Z (uT)
ADLs — head Magnetic Field Z (uT)
Sensor value
Time from start (s)

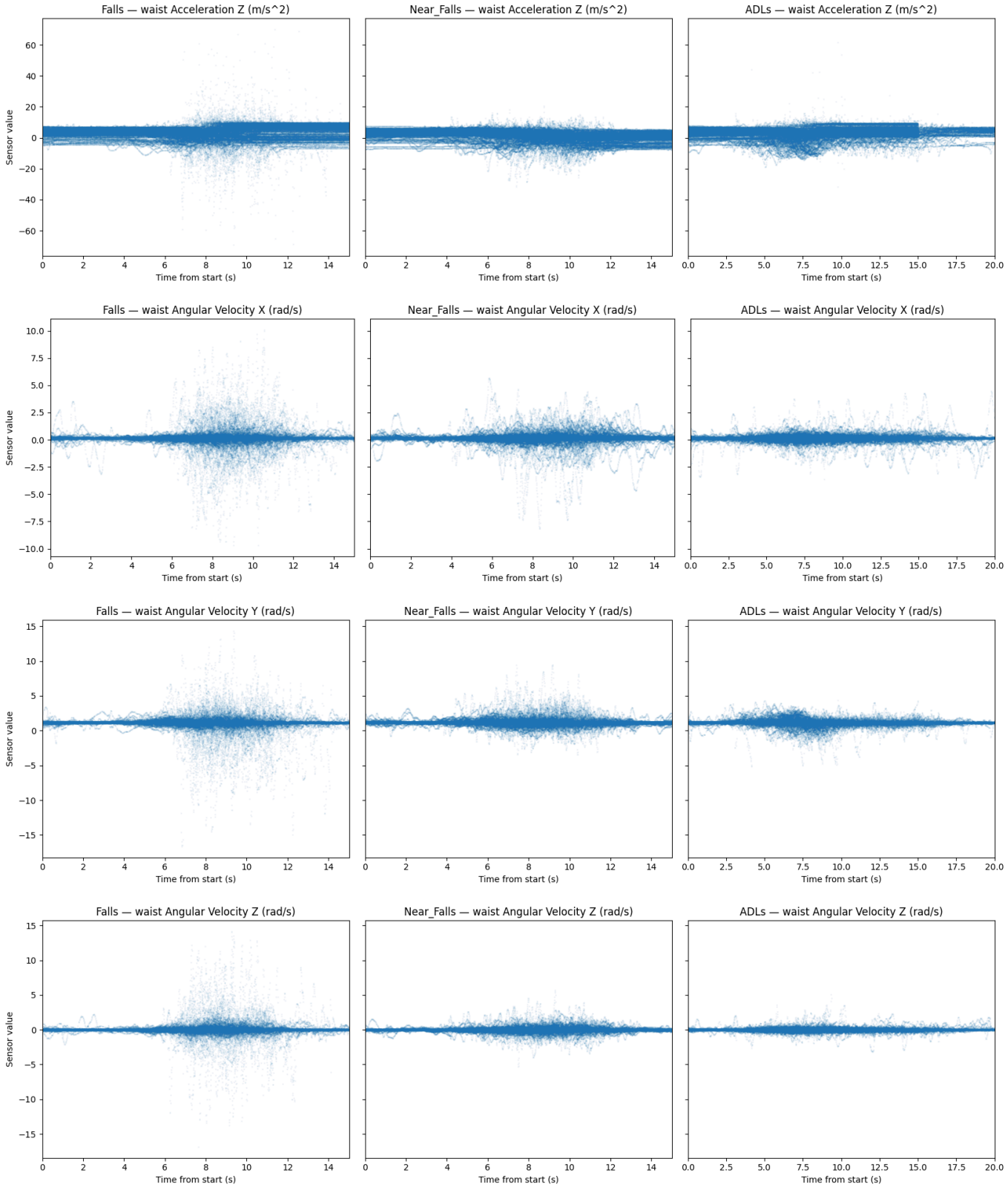

Falls — sternum Acceleration X (m/s^2)
Near_Falls — sternum Acceleration X (m/s^2)
ADLs — sternum Acceleration X (m/s^2)
Sensor value
Time from start (s)

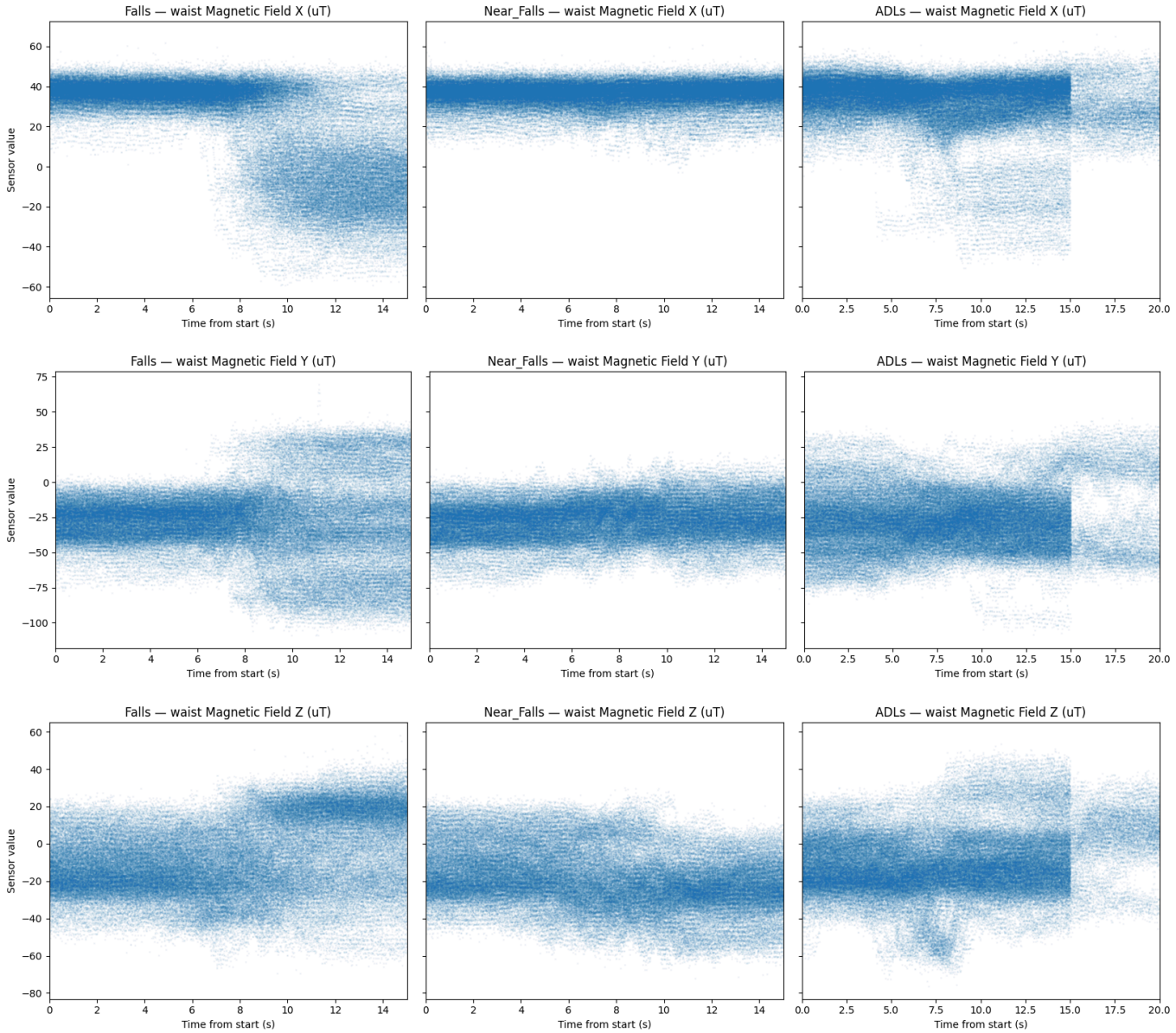

Falls — sternum Acceleration Y (m/s^2)
Near_Falls — sternum Acceleration Y (m/s^2)
ADLs — sternum Acceleration Y (m/s^2)
Sensor value
Time from start (s)


Falls — sternum Acceleration Z (m/s^2)
Near_Falls — sternum Acceleration Z (m/s^2)
ADLs — sternum Acceleration Z (m/s^2)
Sensor value
Time from start (s)


Falls — sternum Angular Velocity X (rad/s)
Near_Falls — sternum Angular Velocity X (rad/s)
ADLs — sternum Angular Velocity X (rad/s)
Falls — sternum Angular Velocity Y (rad/s)
Near_Falls — sternum Angular Velocity Y (rad/s)
ADLs — sternum Angular Velocity Y (rad/s)
Falls — sternum Angular Velocity Z (rad/s)
Near_Falls — sternum Angular Velocity Z (rad/s)
ADLs — sternum Angular Velocity Z (rad/s)
Falls — sternum Magnetic Field X (uT)
Near_Falls — sternum Magnetic Field X (uT)
ADLs — sternum Magnetic Field X (uT)
Sensor value
Time from start (s)


Falls — sternum Magnetic Field Y (uT)
Near_Falls — sternum Magnetic Field Y (uT)
ADLs — sternum Magnetic Field Y (uT)
Falls — sternum Magnetic Field Z (uT)
Near_Falls — sternum Magnetic Field Z (uT)
ADLs — sternum Magnetic Field Z (uT)
Falls — waist Acceleration X (m/s^2)
Near_Falls — waist Acceleration X (m/s^2)
ADLs — waist Acceleration X (m/s^2)
Falls — waist Acceleration Y (m/s^2)
Near_Falls — waist Acceleration Y (m/s^2)
ADLs — waist Acceleration Y (m/s^2)
Sensor value
Time from start (s)


Falls — waist Acceleration Z (m/s^2)
Near_Falls — waist Acceleration Z (m/s^2)
ADLs — waist Acceleration Z (m/s^2)
Falls — waist Angular Velocity X (rad/s)
Near_Falls — waist Angular Velocity X (rad/s)
ADLs — waist Angular Velocity X (rad/s)
Falls — waist Angular Velocity Y (rad/s)
Near_Falls — waist Angular Velocity Y (rad/s)
ADLs — waist Angular Velocity Y (rad/s)
Falls — waist Angular Velocity Z (rad/s)
Near_Falls — waist Angular Velocity Z (rad/s)
ADLs — waist Angular Velocity Z (rad/s)
Sensor value
Time from start (s)


Falls — waist Magnetic Field X (uT)
Near_Falls — waist Magnetic Field X (uT)
ADLs — waist Magnetic Field X (uT)
Falls — waist Magnetic Field Y (uT)
Near_Falls — waist Magnetic Field Y (uT)
ADLs — waist Magnetic Field Y (uT)
Falls — waist Magnetic Field Z (uT)
Near_Falls — waist Magnetic Field Z (uT)
ADLs — waist Magnetic Field Z (uT)
Sensor value
Time from start (s)