2022-09

APPLYING MACHINE LEARNING FOR COP/CTP DATA FILTERING

Goh, Wei Ting

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLYING MACHINE LEARNING FOR COP/CTP DATA FILTERING

by

Wei Ting Goh

September 2022

Thesis Advisor:	Victor R. Garza
Co-Advisor:	Curtis L. Blais
Second Readers:	Brian P. Wood
	Christian R. Fitzpatrick

Research for this thesis was performed at the MOVES Institute.

Approved for public release. Distribution is unlimited.

This project was funded in part by the NPS Naval Research Program.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
APPLYING MACHINE LEARNING FOR COP/CTP DATA FILTERING

5. FUNDING NUMBERS

6. AUTHOR(S) Wei Ting Goh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This project was funded in part by the NPS Naval Research Program.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Accurate tracks and targeting are key to providing decision-makers with the confidence to execute their missions. Increasingly, multiple intelligence, surveillance, and reconnaissance (ISR) assets across different intelligence sources are being used to increase the accuracy of track location, resulting in the need to develop methods to exploit heterogeneous sensor data streams for better target state estimation. One of the algorithms commonly used for target state estimation is the Kalman Filter (KF) algorithm. This algorithm performs well if its covariance matrices are accurate approximations of the uncertainty in sensor measurements. Our research complements the artificial intelligence/machine learning (AI/ML) efforts the U.S. Navy is conducting by quantitatively assessing the potential of using an ML model to predict sensor measurement noise for KF state estimation. We used a computer simulation to generate sensor tracks of a single target and trained a neural network to predict sensor error. The hybrid model (ML-KF) was able to outperform our baseline KF model that uses normalized sensor errors by approximately 20% in target position estimation. Further research in enhancing the ML model with external environment variables as inputs could potentially create an adaptive state estimation system that is capable of operating in varied environment settings.

14. SUBJECT TERMS machine learning, Kalman filter, state estimation, data fusion	**15. NUMBER OF PAGES** 175
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

APPLYING MACHINE LEARNING FOR COP/CTP DATA FILTERING

Wei Ting Goh
Captain, Singapore Army
BSCS, University of Edinburgh, UK, 2018

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
September 2022

Approved by: Victor R. Garza
Advisor

Curtis L. Blais
Co-Advisor

Brian P. Wood
Second Reader

Christian R. Fitzpatrick
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Accurate tracks and targeting are key to providing decision-makers with the confidence to execute their missions. Increasingly, multiple intelligence, surveillance, and reconnaissance (ISR) assets across different intelligence sources are being used to increase the accuracy of track location, resulting in the need to develop methods to exploit heterogeneous sensor data streams for better target state estimation. One of the algorithms commonly used for target state estimation is the Kalman Filter (KF) algorithm. This algorithm performs well if its covariance matrices are accurate approximations of the uncertainty in sensor measurements. Our research complements the artificial intelligence/machine learning (AI/ML) efforts the U.S. Navy is conducting by quantitatively assessing the potential of using an ML model to predict sensor measurement noise for KF state estimation. We used a computer simulation to generate sensor tracks of a single target and trained a neural network to predict sensor error. The hybrid model (ML-KF) was able to outperform our baseline KF model that uses normalized sensor errors by approximately 20% in target position estimation. Further research in enhancing the ML model with external environment variables as inputs could potentially create an adaptive state estimation system that is capable of operating in varied environment settings.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. OVERVIEW....1
B. MOTIVATION....1
C. OBJECTIVES....2
D. ASSUMPTIONS....2
E. APPROACH....2
F. BENEFITS OF RESEARCH....4
G. STRUCTURE OF PAPER....4

II. LITERATURE REVIEW....5
A. BACKGROUND OF DATA FUSION....5
1. Data Fusion Architectures and Models....5
2. Processes and Methods Used in Object Refinement (Level 1)....8
3. Challenges and Limitations in Data Fusion Models....10
4. Data Fusion Applications....10
B. EVALUATING THE PERFORMANCE OF DATA FUSION SYSTEMS....12
1. Challenges in Evaluating Fusion System....12
2. Evaluation of Object Refinement Fusion Process....12
C. AI/ML FOR DATA FILTERING....13
1. Motivation: AI/ML as a DOD Capability....13
2. Defining AI and ML....14
3. Machine Learning Operations and Frameworks....15
4. Using AI/ML to Predict Measurement Noise for KF....18

III. SIMULATING SENSORS DATA FOR DATA FILTERING....21
A. PROPERTIES OF SIMULATIONS....21
B. OVERVIEW OF PROCESS WORKFLOW....22
C. SCENARIO DESIGN....22
1. Overview of CMO....22
2. Physics and Stochastic Modeling in CMO....23
3. Considerations and Constraints in the Design of the Scenario....24
4. Scenario Design—Sensors....24
5. Scenario Design—Target....26

6. Scenario Design—Weather Conditions....27
7. Simulation Runs....28
D. DATASET....29
1. Overview of Dataset....29
2. Data Analysis—Target Position....31
3. Data Analysis—Sensor Detection....32
E. LIMITATIONS OF DATASET....35
1. Insignificant Improvement between Simulation Settings....35
2. Absence of Measurement Errors....36

IV. MODEL GENERATION METHODOLOGY....37
A. KALMAN FILTERS....37
1. Predict Step....38
2. Update Step....40
B. CREATING KF BASELINE MODEL USING FILTERPY....43
C. MACHINE LEARNING MODELS....44
1. Formulation of ML Problem—Modeling Uncertainty....44
2. ML Experiment Framework....47
3. Phase 1: Generation of Dataset....47
4. Phase 2: Build ML Model—Creating a Neural Network....48
5. Phase 3: Publish the ML Model....53
D. STATE ESTIMATION BY THE ML-KF MODEL....53

V. ANALYSIS OF RESULTS....57
A. OVERALL PERFORMANCE....57
1. Comparison of Performance between Weather Datasets....58
2. Performance in Prediction of Longitude and latitude....59
B. PREDICTION ERROR DURING KEY PHASES OF TARGET MOVEMENT....61
1. Constant Heading....62
2. Changing Heading....64

VI. CONCLUSIONS....67
A. SUMMARY OF RESEARCH....67
B. LIMITATIONS AND FUTURE WORK....68
1. Simulated Dataset....68
2. Model Limitations....69
C. CONCLUSION....72

APPENDIX A. SCRIPTS FOR CMO SIMULATION 75
A. LUA SCRIPTING FOR RANDOMLY GENERATING A TARGET'S POSITION 75
B. POWERSHELL SCRIPT TO RUN CMO FROM COMMAND LINE INTERFACE 76

APPENDIX B. DATA DICTIONARY 77
A. UNIT POSITION TABLE 77
B. SENSOR DETECTION ATTEMPT TABLE 78

APPENDIX C. SOFTWARE PACKAGES USED 81

APPENDIX D. PYTHON NOTEBOOK–EXPLORATORY DATA ANALYSIS 83
A. INSPECTING HEADERS OF DATASET 83
B. COMPARISON OF COORDINATE SYSTEMS (GEODESIC AND ENU REPRESENTATION) 83
1. Sample Dataset 84
C. EXPLORATORY DATA ANALYSIS OF TARGET UNIT POSITION DATASET 88
1. General Statistics of Dataset 88
2. Visualization of Target Movement in Simulation 90
D. EXPLORATORY DATA ANALYSIS OF SENSOR DATASET 93
1. Periodicity of Data, Number of Detection, Failure Rate in Dataset 93
2. Detection Success / Failure Rate across Each Scenario 105
E. CALCULATE THE SENSOR ERRORS 110

APPENDIX E. PYTHON NOTEBOOK — BASELINE MODEL WITH KF 113
A. PARAMETERS IN THE KALMAN FILTER ALGORITHM 113
1. Initialization 113
2. Predict Step 113
3. Update Step 113
B. SET UP KALMAN FILTER FUNCTIONS 114
C. CREATING AN INTERFACE WITH DATASET 116
D. RUNNING A SIMULATION WITH KF 119
E. EVALUATION OF RESULTS 128

APPENDIX F. PYTHON SCRIPTS–ML MODEL TUNING 133

LIST OF REFERENCES **141**

INITIAL DISTRIBUTION LIST **149**

LIST OF FIGURES

Figure 1. Methodology for Assessing Track Filtering Algorithms. 3

Figure 2. JDL Fusion Processing Model. Source: Steinberg et al. (2017). 7

Figure 3. Automated ML Workflow Pipeline. Adapted from Kreuzberger et al. (2022b). 16

Figure 4. Commonly Used Software in the Various Workflows of MLOps. Source: Karayev et al. (2022). 17

Figure 5. Screenshot of the CMO Database Editor of a Built-in Radar sensor 23

Figure 6. Scenario with Target and Four Sensor Platforms. 25

Figure 7. Weather Settings for Ideal Weather Conditions. 28

Figure 8. Weather Settings for Sub-optimal Weather Conditions for EO/ IR sensor. 28

Figure 9. Simulation Runtime Setting 29

Figure 10. Visualization of Target's Movement during the Simulation. 31

Figure 11. ML Experiment Framework 47

Figure 12. Architecture of Dual-Head Neural Network 49

Figure 13. Process Flow for Predicting Target State by ML-KF Model 55

Figure 14. KF Model RMSE Distribution 60

Figure 15. ML-KF Model RMSE Distribution 61

Figure 16. Model's State Estimation Error during a Single Simulation Run 62

Figure 17. Target Moving between RP-43 to RP-41 (Constant Heading) 63

Figure 18. Target Changing its Course around RP-43 65

Figure 19. Target Changing its Course around RP-42 66

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Description of Simulated Sensors in Scenario 26

Table 2. Sensor Detection Period and Start Time 32

Table 3. Sensor Detection Success 33

Table 4. Average Sensor Detection Error in Different Weather Settings 35

Table 5. Symbols Used in the Predict Step 38

Table 6. Symbols Used in the Update Step 40

Table 7. Sensor Measurement Noise Derived from Dataset 42

Table 8. Optimization Algorithm Parameters 50

Table 9. Hyperparameter Search Space 52

Table 10. Selected Hyperparameters of DHNN Models 52

Table 11. Model Performance and Description 53

Table 12. Performance of Models across 100 Simulation Runs 58

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3D	Three Dimensional
AFSIM	Advanced Framework for Simulation, Integration and Modeling
AGL	Above Ground Level
AI	Artificial Intelligence
Alt	Altitude
AOU	Area of Uncertainty
ASL	Above Sea Level
ATA	Automatic Tracking Aid
ATR	Automatic Target Recognition
C2	Command and Control
CCD-TV	Charge-Coupled Device Television
CLI	Command-line Interface
CMO	Command: Modern Operations
CO_2	Carbon Dioxide
COP	Common Operating Picture
COVID	Coronavirus Disease
CPU	Computer Processing Unit
CSV	Comma-separated Value
CTP	Common Tactical Picture
DBID	Database Identification
DDG	Guided Missile Destroyer
DHNN	Dual-head Neural Network
DL	Deep Learning
DNN	Deep Neural Networks
DoD	Department of Defense
E	East
ENU	East-North-Up
EO	Electro-Optical
ELINT	Electronic Intelligence
ESM	Electronic Support Measures

EW	Electronic Warfare
F	State Transition Matrix (variable)
FLIR	Forward-Looking Infrared
GUI	Graphical User Interface
GPU	Graphics Processing Units
H	Measurement Function (variable)
HH	Hours
HP	Hewlett-Packard
ID	Identification
IMINT	Imagery Intelligence
IR	Infrared Radiation
ISR	Intelligence, Surveillance, Reconnaissance
IW	Information Warfare
IWC	Information Warfare Community
JDL	Joint Directors of Laboratories
JP	Joint Publication
K	Kalman Gain (variable)
KF	Kalman Filter
kt	Knot
Lat	Latitude
Lon	Longitude
LSTM	Long-Short Term Memory
m	Meter
MATLAB	MATrix LABoratory
max	Maximum
min	Minimum
ML	Machine Learning
MLOps	Machine Learning Operations
MM	Minutes
MOE	Measure of Effectiveness
MOP	Measure of Performance
N	North

NATO	North Atlantic Treaty Organization
NAVIFOR	Naval Information Forces
nm	Nautical Mile
NN	Neural Network
P	State Covariance Matrix (variable)
Q	Process Noise Covariance (variable)
R	Measurement Noise Covariance (variable)
RADAR	Radio Detection and Ranging
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSE	Root Mean Square Error
RP	Reference Point
s	Second
SAF	Singapore Armed Forces
SIGINT	Signals Intelligence
SMA	Simple Moving Average
SS	Seconds
std	Standard Deviation
w	Process Noise (variable)
X	State of System (variable)
y	Residual (variable)
z	Measurement (variable)

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The ability to process and exploit multiple intelligence data streams is essential to achieving superior battlespace awareness. The U.S. Navy, and specifically Naval Information Forces (NAVIFOR), is exploring the effectiveness of Artificial Intelligence (AI)/Machine Learning (ML) technology to assist with data fusion and provide quick and timely analysis of the Common Operating Picture (COP)/Common Tactical Picture (CTP). One area of focus is the filtering of data from different sensor systems to provide improved state estimation of targets in the battlespace. This is a critical task as accurate tracks and targeting are key to providing decision makers with the confidence to execute their mission.

This thesis aims to assess the feasibility of integrating AI/ ML algorithms and techniques to filter heterogenous datasets to increase the accuracy of track estimation in developing COP/CTP. The Kalman Filter (KF) and its variants are often used to estimate the position of targets in the battlespace. Estimation accuracy, however, is greatly affected by changes in external conditions and by violations to the assumptions made about the target.

Research conducted by Gao et al. (2020), Jouaber et al. (2021), and Ullah et al. (2019 and 2020) has shown the potential to integrate a learning module within a standard KF to improve the accuracy of state estimation. This research used a neural network (NN) to learn the variability in measurement uncertainties associated with sensor measurements. These variabilities exist because of changes in external factors such as weather conditions that are not directly modeled as the state of the KF algorithm. This has the potential to enhance our COP/CTP, especially when external factors affect our sensor fusion systems dynamically.

We use a quantitative approach to assess the accuracy of selected AI/ML algorithms in filtering datasets of target positions. We hypothesize that inclusion of a learning module within a KF model will outperform a standard KF model and provide a

better estimate of the target position. To that end, we designed a three-phase data pipeline (Figure 1).

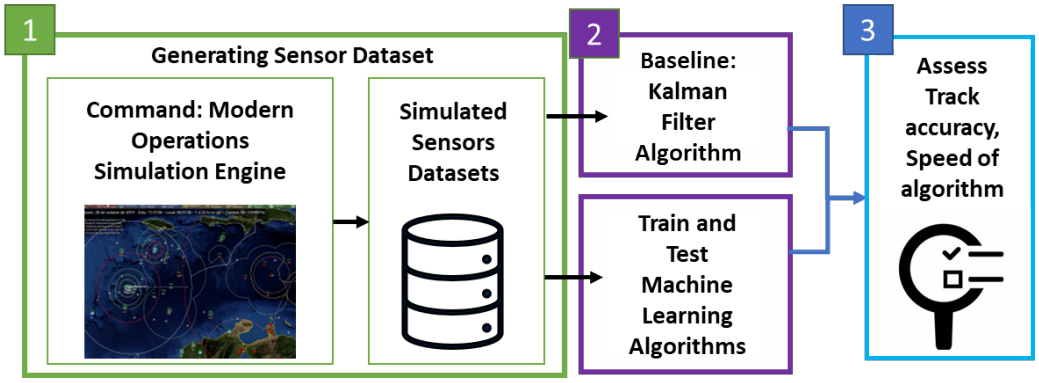


Figure 1. Methodology for Assessing Track Filtering Algorithms.

First, sensor data is generated using simulation software—Command: Modern Operations (CMO) developed by Matrix Games (Matrix Games, 2022b). A scenario consisting of multiple stand-off sensors from different intelligence domains and a single target was used. Second, two sets of models were developed—a standard baseline model using the KF algorithm and another using a neural network embedded in the KF algorithm (we call this the ML-KF model). This neural network is a learning module that was trained on the training dataset and was used to estimate the sensor measurement noise of the KF. We conduct a hyper parameter search across the different hyperparameters possible to improve the performance of each sensor's ML model. In the final phase, the performance of the two models was assessed for accuracy in estimating target state position.

Our findings showed that the integration of ML models to estimate the sensor measurement error matrix for the standard KF algorithm can significantly improve the accuracy of target state estimation by approximately 20% at a 5% confidence level. In summary, our contributions are the following:

1. We have developed an ML operations pipeline that ingests data from a simulation to train, validate, and test machine learning modules for

subsequent deployment in a KF system. The method, dataset, and models generated is reproducible and replicable, as the code base and frameworks used for this development are fully open source.

2. We have shown that a learning module embedded in a standard KF algorithm can improve state estimation over a standard KF model. The ML-KF model was able to generate a sensor measurement error matrix to update the KF algorithm's probabilistic belief of the sensor measurements, thereby improving the KF's estimation.

3. We were able to train the learning module used in the KF model only because our simulation system provides a ground truth target state which live ranges may not be able to provide. This proves the potential for using simulation to develop ML models and of subsequently deploy them in the field.

Our research used ML models to predict sensor measurement errors for a standard KF algorithm. Our ML-KF model was able to significantly outperform our baseline model at 5% confidence level, showing that using an ML-KF model would improve the performance of target position state estimations, alleviating the performance issue when uncertainty of sensor measurement is absent from heterogenous sensor data streams. In other words, in the absence of uncertainty measurements of sensor data, the ML embedded in the KF was able to predict the uncertainty and dynamically updating the parameters of the KF algorithm.

This proof-of-concept has the potential to be further extended using more sophisticated methods. We have proposed three key areas for future research: 1) To improve the generalizability of the ML-KF model by including other parameters (such as weather conditions in the battlespace) that are not directly modeled or used in KF. 2) Using time-series methods to model temporal movement of a target, thereby increasing the predictive power of the learning module embedded in the KF. 3) Using ML models to conduct multiple target data filtering (JDL Level 2), by including a classification task to categorize the track data.

As the DoD increasingly shifts its focus to the application of ML, we believe that such an application in data filtering would be able to augment existing data filtering methods and eliminate the expense of replacing them. For instance, by enhancing existing COP/CTP data filtering algorithms, we would be able to have better accurate state estimation of the target, thereby providing a higher confidence of the target's position in the COP/CTP. The ability for such an ML-KF model to ingest heterogenous data stream is also a powerful tool to automate the work of intelligence analysts who would frequently need to cross-reference their sources across different intelligence domains. By improving the suite of tools available to our warfighters, they will be more lethal in their response to any adversary.

References

Gao, X., Luo, H., Ning, B., Zhao, F., Bao, L., Gong, Y., Xiao, Y., & Jiang, J. (2020). RL-AKF: An adaptive kalman filter navigation algorithm based on reinforcement learning for ground vehicles. *Remote Sensing*, *12*(11), 1704. https://doi.org/10.3390/rs12111704

Jouaber, S., Bonnabel, S., Velasco-Forero, S., & Pilté, M. (2021). NNAKF: A neural network adapted Kalman filter for target tracking. *ICASSP 2021 – 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4075–4079. https://doi.org/10.1109/ICASSP39728.2021.9414681

Matrix Games. "Command: Modern operations." Windows. UK: Matrix Games, 2022. *https://www.matrixgames.com/game/command-modern-operations*.

Ullah, I., Fayaz, M., & Kim, D. (2019). Improving accuracy of the Kalman filter algorithm in dynamic conditions using ANN-based learning module. *Symmetry*, *11*(1), 94. https://doi.org/10.3390/sym11010094

Ullah, I., Fayaz, M., Naveed, N., & Kim, D. (2020). ANN based learning to Kalman filter algorithm for indoor environment prediction in smart greenhouse. *IEEE Access*, *8*, 159371–159388. https://doi.org/10.1109/ACCESS.2020.3016277

ACKNOWLEDGMENTS

This journey was not possible without the unwavering commitment and dedication of the team: Mr. Fitzpatrick who jumped through the hurdles for simulation software, Mr. Wood who kept me on track and framing the thesis report, Dr. Blais for the technical guidance and scoping of the problem, and Mr. Garza for coordinating with our sponsors and steering the direction of the thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

This thesis aims to assess the feasibility of integrating Artificial Intelligence (AI)/ Machine Learning (ML) algorithms and techniques to filter heterogenous datasets to increase the speed and accuracy of tracks in developing a Common Operating Picture (COP)/Common Tactical Picture (CTP) for battlefield awareness. The Kalman Filter (KF) and its variants are often used to estimate the position of targets in the battlespace. Yet, estimation accuracy is greatly affected by changes in external conditions and by violations to the assumptions made about the target.

To improve state estimation from a KF, this research adopts a quantitative approach to assess traditional KF models against a hybrid ML-KF model, whereby a learning module is embedded as part of the KF to improve its adaptability. Using simulation software to generate sensors and track datasets, we assess the improvement in accuracy of estimating the state of a target by these models.

B. MOTIVATION

The ability to process and exploit multiple intelligence data streams is essential to achieving superior battlespace awareness. The U.S. Navy, and specifically Naval Information Forces (NAVIFOR), is exploring the effectiveness of AI/ML technology to assist with data fusion and provide quick and timely analysis of the COP/CTP. The current situation is exacerbated by the proposed increase in the number of Intelligence, Surveillance, and Reconnaissance (ISR) assets within the battlefield, which may potentially overwhelm human operators and intelligence analysts with high *volume* and *velocity* of data, leading to human errors and replicative efforts in going through similar effects. In addition, potential counter-intelligence tactics by the adversary may also affect the *veracity* of data received, disrupting our battlespace situation awareness.

While the Department of Defense (DoD) has been investing heavily in data filtering systems, often embedded as part of the data fusion systems of systems, advances in sensor technology and AI/ML–aided by the availability of big data–provide an avenue

for NAVIFOR to adopt AI/ML technologies from industries such as robotics, autonomous vehicles, and recommender systems. By studying the AI/ML technologies and techniques used to automate the filtering of multiple data streams, we can draw parallels and quantitatively assess the effectiveness of such technologies in data filtering for target state estimation in the development of COP/CTP.

C. OBJECTIVES

Our research objective is to determine the effectiveness of AI/ML algorithms in multi-source data filtering to provide the warfighter with accurate target position estimation.

D. ASSUMPTIONS

In this research, we assume that the sensor data generated from the simulation software is sufficiently representative of the capability of sensors in the fleet. In addition, we assume that all data generated are agnostic to the idiosyncratic data formats required by Command and Control (C2) systems.

E. APPROACH

We use a quantitative approach to assess the accuracy of a selected AI/ML algorithm in filtering datasets of target positions. We hypothesize that the inclusion of a learning module within a KF model will out-perform a standard KF model and provide an improved estimate of the target position. Figure 1 illustrates the three phases of our approach.

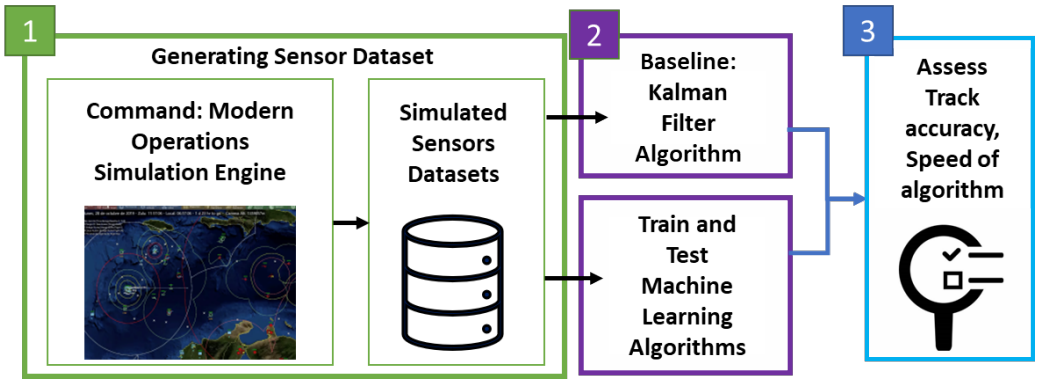


Figure 1. Methodology for Assessing Track Filtering Algorithms.

First, a simulation software—Command: Modern Operations (CMO) developed by Matrix Games (Matrix Games, 2022b) will be used to generate sensor dataset. A suitable scenario that consists of multiple stand-off sensors from different intelligence domains and a single target could be used to model the detection of a single target that is conducting a patrol in a defined area. Two scenarios will be generated to simulate ideal weather conditions and moderately adverse weather conditions, to assess the effectiveness of the algorithm in different weather conditions. In total, 100 iterations of each scenario should be ran with random perturbations to the initial position of the target so that randomness is introduced to the dataset. Each iteration provides us with a set of sensor readings of the target moving within the scenario and the ground truth of the target's position.

Second, two sets of models will be developed—a standard baseline model using the standard KF algorithm and another using a selected ML model embedded in the KF algorithm (which we call this the ML-KF model). The ML model is generated from a learning phase whereby its parameters are updated after each batch of the dataset is presented to it. After learning is completed, the model is evaluated against a dataset that was not seen during this phase to assess the performance of the ML model. The best ML-KF model would be used for evaluation to pit against the KF baseline model.

Third, we assess the performance of the ML-KF model against the baseline KF model. We assess the algorithms on the accuracy of the target's estimated position across each weather datasets.

F. BENEFITS OF RESEARCH

Two potential benefits from this study include the following:

1. It provides a survey on existing AI/ML techniques for automated data filtering to improve the accuracy of track prediction that can inform the wider Information Warfare Community (IWC) and coalition partners on the effectiveness, limitations, and constraints of the technology.
2. It demonstrates a prototype that provides a study on an AI/ML technique used to enhance the KF algorithm. This study informs the design of future AI/ML systems.

G. STRUCTURE OF PAPER

Chapter II sets the foundation of our research, providing the history of data fusion and its relevance to the DoD, AI/ML techniques used in data fusion and track management, and the evaluation of data fusion systems. Chapter III introduces a data pipeline used to generate datasets for the evaluation of algorithms and building machine learning for model evaluation. The scenario design principles and an exploratory data analysis on the datasets generated using CMO are presented. Chapter IV introduces the KF algorithm, formulating data filtering as a predictive task for the KF's parameter and approach to generating ML models. Chapter V informs the performance of each model. Finally, Chapter VI concludes with the limitations of the research, lessons learned, recommendations, and future areas of study.

II. LITERATURE REVIEW

This chapter outlines a summary of the literature and sources used to improve our understanding of AI/ML techniques applied as a data fusion technique. The review aims to provide 1) an overview of data fusion systems and their relevance to the DoD, 2) background on existing AI/ML technologies applied in data fusion and track management, and 3) methods for evaluating data fusion systems.

A. BACKGROUND OF DATA FUSION

Catalyzed by the need to enhance multi-domain battlespace awareness and reduce decision time to respond to threats, and thereby gain a superior advantage against near-peer threats, data fusion technology remains one of the key areas of investment for the DoD (Hoehn, 2022). The conduct of data fusion aims to aid decision making, answer commanders' questions, and reduce the uncertainty of the battlespace. The Data Fusion Group formed by the Joint Directors of Laboratories (JDL), proposed the following definition of Data Fusion: "Data fusion is the process of combining data to refine state estimates and prediction" (Steinberg et al., 2017). The DoD has made significant investments in data fusion technology, cutting across a wide range of mission sets and fusion capabilities (Nicholas, 2008).

1. Data Fusion Architectures and Models

The JDL Data Fusion Model identifies key functions related to data fusion by providing common systems engineering standards and vocabulary for developers. To date, it is the most widely used framework for the development of data fusion functions in military applications (Steinberg et al., 2017). While there are contending fusion models and architectures available, the JDL Data Fusion Model serves as a useful reference for this thesis due to our use case on track management and data filtering.

The model consists of five levels of processing where data fusion would occur (Figure 2). Depending on the application, a data fusion system may hierarchically implement these levels of fusion. We expand on the levels in the following paragraphs:

- Source Pre-processing. This pre-processing step is often conducted at the sensor node. For example, a radar may utilize the moving target indicator technique to confirm the presence of an object at a given time and space. It can change its pulse width and increase its range resolution to improve the signal-to-noise ratio in the presence of a potential target and thus reduce environmental clutter noise. This process results in the detection of an object when the raw signature (signal or pixel) received by the sensor is above the signal-to-noise ratio threshold. This initial "contact" with the target results in a track initialization with the sensor continuously measuring and estimating the state of the object (Dietrich, 2001).

- Level 1 Processing—Object Refinement. The object refinement process provides estimates and predictions of an entity's physical states (position, velocity, attributes such as size and signatures, and identity) in the battlespace. For example, a radar tracking a threat in the battlespace may cross-cue an imaging sensor to zoom in on the area where a target is estimated to be. If the imaging sensor was able to detect a target within its field of view, positive contact and track would be formed. The fusion of both sensors' tracks would result in a fused track. The output from level 1 processing is predictive, as the fused track not only provides the estimated current position, but also the projected location of the object based on existing sensor measurements. This thesis focuses on the refinement of fused track data from multiple sensors.

- Level 2 Processing—Situation Refinement. This level allows the user to draw inferences about the relationship between objects. The fused product allows the user to infer the force structure and command relationship between entities, providing context to their estimated state. For example, in a hierarchical data fusion system, the analysis of velocity histories of level 1 tracks affords further classification of the movement formation

from commonly used doctrines (e.g., the advancement of a carrier strike group is distinctly different from the movement of a littoral combat team).

- Level 3 Processing—Threat Refinement. This process aims to estimate and predict the outcome of a course of action, or the potential impact of the entities' actions. Continuing the example from level 2, having classified the tracks based on their velocity profile and inferring the size and disposition of the threat, the fusion system would further increase the threat level of the targets. This refinement and change in threat classification would result in a strong warning and alert to the user.

- Level 4 Processing—Process Refinement. While source preprocessing to level 3 preprocessing manipulates the sensor data or its derivative directly, level 4 processing is concern with answering the question of "does the products from source pre-processing to level 3 processing fulfills the mission?". To that end, level 4 processing observes the performance of the fusion process occurring at levels 1 to 3 and aims to optimize resource allocation (e.g., sensors and deployment time) to meet the goals of the mission.

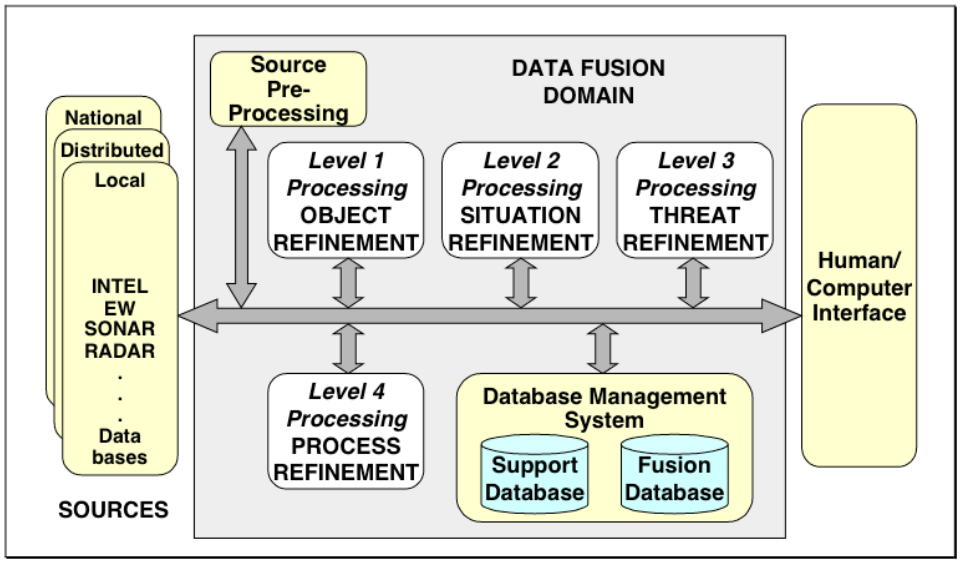


Figure 2. JDL Fusion Processing Model. Source: Steinberg et al. (2017).

2. Processes and Methods Used in Object Refinement (Level 1)

This section details the processes that occur within JDL level 1 object refinement fusion processing. It aims to provide common algorithms and methods used for fusing multiple state estimations from various sensors. Smith and Singh (2006) outlined four key processes in JDL level 1 fusion processing of process refinement as data registration, data association, state estimation, and identification.

a. Data Registration

Data registration is the process of aligning data from various sources into the same frame of reference. This process is often required when the sensors are spatially distributed and have different fields of view. Converting all sensor readings to a common reference frame prevents confusion by the algorithm in subsequent stages. Common methods include the conversion of an estimated target position to latitude or longitude by referencing the target range to the sensor and standardizing the estimated target temperature to the same reference units such as Fahrenheit or Kelvin.

b. Data Association

The data association process can be sub-categorized into measurement-to-track and track-to-track, depending on the data type that is being provided to the association algorithm, and it aims to correlate measurement or tracks from each sensor to the identical real-world object the sensor is measuring. The algorithm, also known as a correlator, can be complex as it manages the entire track cycle. A track cycle comprises three sequential stages: 1) track initialization, 2) track maintenance, and 3) track deletion. The *MATLAB & Simulink Sensor Fusion and Tracking Toolbox* (MathWorks, 2022) describes the tracker's operating logic when a new detection has been made:

(1) The tracker tries to assign a detection to an existing track.

(2) The tracker creates a track for each detection it cannot assign. When starting the tracker, all detections are used to create tracks.

(3) The tracker evaluates the status of each track. For new tracks, the status is tentative until enough detections are made to confirm the track. For

existing tracks, newly assigned detections are used by the filter to update the track state. When a track has no new added detections, the track is predicted until new detections are assigned to it. If no new detections are added after a specified number of updates, the track is deleted.

Common methods include joint probabilistic data association, nearest neighbor clustering, and fuzzy logic systems. Ideally, the output from this step provides a fused track for each real-world object. Sub-optimally, the algorithm may result in a track breaking or a redundant track.

c. State Estimation

Once the tracks are clustered, attributes related to the specific fused track can be estimated, such as location, velocity, heading, and altitude. The most common method used is the KF, a recursive Bayesian estimator for attributes of interests based on sensor measurements in the presence of uncertainties. To utilize a KF, the sensor noise and model of the object to be detected must be known and hence most state estimators are often hand-crafted by subject matter experts for the data fusion system based on the set of targets the system intends to track. Many variants of KF have since been developed (Akca & Efe, 2019). For example, Extended KF, Unscented KF, Particle Filter, or Multiple Models Filters (an ensemble of filters modeling different possible system dynamics) are used to estimate the state of a non-linear dynamic system.

d. Identification

Finally, the object is given a combat identification based on its state and attributes. Common methods include Bayesian inference, expert system with hand-crafted rulesets developed by subject matter experts, and ensemble models that weigh the confidence level for each independent classifier to determine the final classification of the target. Like state estimation, identification often requires expert knowledge on the type of target the fusion system will be tracking for the target to be identified successfully.

3. Challenges and Limitations in Data Fusion Models

Esteban et al. (2005) surveyed widely used data fusion architectures and models from a systems engineering perspective. They found that different use cases and applications have resulted in a wide range of architectures and models and having a common unified architecture for data fusion is challenging. For instance, a fusion process model of a multi-sensor data fusion system may differ from another one due to the sensors' configurations (sensors may be set up in parallel or serial configuration), the level(s) of information desired from the fusion system, and location of fusion algorithm (on level 1 fusion conducted at each sensor in a decentralized model, or a computationally intensive centralized fusion based on all sensors' raw data). Due to the different mission sets and profiles, nuances within each data fusion system exist and there is no all-encompassing data fusion model to holistically describe all existing and potential systems.

In addition, there is no commonly agreed-upon data governance framework to control data input into a fusion system. Research by Watson (2021), building on the dissertation work by Rothenhaus (2008), proposed and showed that a data governance framework for a multi-source data fusion system to remove poor quality sources would significantly improve the performance of track correlators while reducing analysts' time to rectify incorrect position reports. As the adage "garbage in garbage out" goes, the data fusion process is not a panacea when sensor readings are biased or flawed.

4. Data Fusion Applications

a. Intelligence Operations

The joint targeting cycle presents a useful case study for fusion in JDL level 3 threat refinement, often involving intelligence analysts in the loop due to the high cost of erroneous analysis. The DoD *Joint Publication on Joint Targeting* (JP-3-60) (Joint Chief of Staff, 2018) and *Joint Publication on Joint Tactics, Techniques, and Procedures for Intelligence Support to Targeting* (JP-2-01.1) (Joint Chief of Staff, 2013) outline the importance of all-source intelligence fusion in the success of a targeting mission. It is time-critical to inform commanders on the effectiveness of munitions in the conduct of

combat assessment and whether the objective of the mission has been achieved. The ability to achieve quality damage assessment requires intelligence analysts to exploit all-source data through the fusion process. The fused report would provide reattack recommendations for the decision maker, signifying the importance of the report and the iterative nature of intelligence operations.

b. Surveillance

Other military-related data fusion applications include target tracking (Koch, 2014); multi-sensors automated target recognition (ATR) (Schachter, 2020); aerial surveillance (Maltese & Lucas, 1998); maritime surveillance (Guerriero et al., 2008); space-based ISR over an area of interest and threat monitoring of Earth (Crothers et al., 2009); and deep-space surveillance of military and commercial satellites orbiting Earth (Sharma & Stokes, 2002).

Building on the physical space, military data fusion applications increasingly augment data sources from the physical space with sources from cyberspace, such as wireless and computer networks, and social space in online social media (Wang et al., 2019). The counterinsurgency campaigns in Iraq provide a case study where data from social networks, communications networks, and geospatial data were combined to conduct a cultural analysis of the adversary (Merten, 2014). The shift to increasing the volume and variety of data sources for data fusion is a testament to the increasingly digitalized world that we live in, one where data will be exploited for data-driven decision making.

c. Other Applications

Outside of the military, data fusion is widely used in criminal investigations, medical diagnosis, system fault diagnosis, weather forecasting, and economic analysis (Blasch et al., 2014; Li et al., 2018; Murashov, 2021). In recent years, advancements in machine learning, artificial intelligence, and the Internet of Things have resulted in applications in 1) autonomous vehicle navigation and control (Yeong et al., 2021); 2) urban planning decisions, infrastructure management, environment, and waste monitoring, and mobility management (Lau et al., 2019); 3) precision manufacturing

(Kong et al., 2020); and 4) management of pandemics, such as the recent COVID-19 pandemic (Singapore Armed Forces, 2022).

B. EVALUATING THE PERFORMANCE OF DATA FUSION SYSTEMS

Evaluation of a data fusion system is critical to understanding its effectiveness and performance.

1. Challenges in Evaluating Fusion System

In a NATO Science and Technology Organization lecture series on data fusion, Koch noted that a comprehensive evaluation of sensor fusion performance is only possible in highly controlled real-life laboratory testing (Koch, 2015). This is because, for level 1 fusion, it is often impossible to measure the ground truth of an actual target to a high degree of accuracy in real-life scenarios due to the presence of systemic and random errors, making a performance evaluation of a live system futile. One possible way of evaluating sensor fusion algorithms is through simulation-based experiments, where fusion algorithms ingest data from simulated sensors and randomness is introduced to probabilistically simulate the detection of a target based on each sensor's expected performance (Miller et al., 2020).

2. Evaluation of Object Refinement Fusion Process

Given that the end goal of the level 1 object refinement fusion process is to improve the measured state of a target, the following measures of performance (MOP) were proposed: number of valid tracks or number of false tracks to measure the performance of data association and identification (Koch, 2015), accuracy of a track to measure the state estimation performance (Llinas, 2008), and time to track to measure timelines of information (Dietrich, 2001).

a. Number of Valid Tracks, Number of False Tracks

Ideally, the fusion system should assign one track ID per target. Additional tracks of the same target are potentially confusing to the warfighter, and misleading to the

analysts. False tracks may occur due to objects that are irrelevant but detected by the sensors, such as clutter, countermeasures, and environmental noise (Koch, 2015).

b. Track Accuracy (State Estimation Metric)

Naturally, the fused track should improve the estimation of the object's state, and hence an effective fusion system should have a small difference between the ground truth of the target state and the estimated state (Llinas, 2008).

c. Time to Track

Due to communications and computation processes, the time taken to fuse tracks within a sensor and across multiple sensors may induce an "extraction delay" between the first detection by the sensor and the confirmation of a track. The timeliness of information is a measure of effectiveness (MOE) of the overall fusion system to determine whether the system was able to assist the warfighter in its operational mission (Dietrich, 2001).

C. AI/ML FOR DATA FILTERING

1. Motivation: AI/ML as a DOD Capability

The 2019 DoD Digital Modernization Strategy and 2018 DoD Artificial Intelligence Strategy highlighted Artificial Intelligence and Big Data Analytics as key technology areas of interest for the DoD. Specifically, the DoD is committed to developing and using AI technologies and systems to augment duty personnel by reducing their cognitive workload for dull and repetitive tasks where machines excel. Supporting these strategies, the 2020 DoD Data Strategy aims to leverage data as a warfighting asset. The Data Strategy sets the direction for the development of data-driven operations wherein operators and decision makers exploit data for enhanced battlespace awareness to outsmart adversaries in multiple operating domains and across levels of operations. AI and ML are key enabling technologies to realize this goal. Our sponsor, NAVIFOR, is interested in exploiting readily available AI and ML technology and techniques to enhance both situational awareness and C2 of our force's operations through the improvement of our COP/CTP.

2. Defining AI and ML

This section aims to provide background on AI and ML techniques that are used for data fusion. In the authoritative and most-used artificial intelligence textbook *Artificial Intelligence: A Modern Approach*, Russell et al. (2010) defines an artificial intelligence agent as a rational agent that "*takes the best possible action in a situation*." An AI perceives and interacts with its task environment and its purpose is to maximize its performance as defined by the programmer. To improve its performance, an AI goes through a learning process to adapt to novel and unseen circumstances. ML is a subfield of AI in which an agent can use the techniques to learn from data to make an informed decision. Through learning, the agent develops models that maps the agent's perceptual inputs from the environment to actions that interact with the environment.

The ML field can be classified into three core paradigms—supervised learning, unsupervised learning, and reinforcement learning—often related to the task that the user aims to solve and the data that is available for learning (Goodfellow et al., 2016). Another buzzword "deep learning" (DL) would be used along with ML because DL is a subfield of ML and focuses on methods that utilize deep neural networks (DNN) as part of the learning algorithm, allowing simpler concepts to be combined into a more complex and nuanced one (Goodfellow et al., 2016).

a. Supervised Learning

Supervised learning requires a labeled dataset, and the objective of the learning algorithm is to build a statistical model of each input-output pair that optimizes a metric that measures the performance of the model (Goodfellow et al., 2016). The ML task is often a classification or regression task that predicts a class label or numerical value. Metrics used to assess the performance of a supervised learning method would include classification error, accuracy, or mean squared error.

b. Unsupervised Learning

Unsupervised learning does not require a labeled dataset and focuses on descriptive tasks such as understanding the structure of data, reducing the feature

dimension of data, and generative modeling (Goodfellow et al., 2016). We may use unsupervised learning in data fusion to answer the question of "Is a given fused track from a sensor associated with that of another?"

c. Reinforcement Learning

Reinforcement learning is fundamentally different from supervised and unsupervised learning methods because the data used for learning is obtained by the AI agent sampling an environment, such as a simulation engine (Goodfellow et al., 2016). The AI agent would perceive and act on the environment to understand the potential reward or penalty it will receive from the environment. If the agent is rational, then the learning algorithm will update the parameters of the agent such that it maximizes its long-term reward (Russell et al., 2010). Recent advances in reinforcement learning include AI achieving grandmaster status in the game of Go without learning from a dataset of games from professional human players (AlphaGoZero), and predicting 3D protein structure from an amino acid sequence (AlphaFold) by DeepMind (Jumper et al., 2021; Silver et al., 2017).

3. Machine Learning Operations and Frameworks

In this section, we introduce Machine Learning Operations (MLOps) and its common ML frameworks. In their survey on MLOps, Kreuzberger et al., (2022) define MLOps as an engineering practice focused on developing ML products and bringing them into operations. It is concerned with the "end-to-end conceptualization, implementation, monitoring, deployment, and scalability of machine learning product(s)," supported by a multi-disciplinary team comprising the business owners, data scientists, data engineers, software engineers, and ML engineers.

Based on their research, an end-to-end MLOps architecture and workflow (depicted in Figure 3) that comprises four key stages:

a. MLOps project initialization aims to establish the ML problem from its business goal.

b. A feature engineering pipeline is set up by establishing rules for extracting features from the dataset. This feature engineering pipeline would be an iterative process with the subsequent experimentation stage until a model that performs well for the task has been developed.

c. In the experimentation stage, the data scientists lead the team through an iterative model training process to test different algorithms and associated hyperparameters. The intended outcome of the experimentation stage is to inform the best-performing model for the task.

d. Finally, the automated ML workflow pipeline is triggered. This workflow pipeline includes similar steps in the experimentation stages but is automated for continuous build, test, and deployment (Figure 3). This allows the model deployed to be periodically updated as new versions of data features are extracted and updated by the model. The automated ML workflow interfaces with the operations through a continuous deployment process. Feedback from the consumer would be monitored so that timely updates to the model can be integrated effectively.

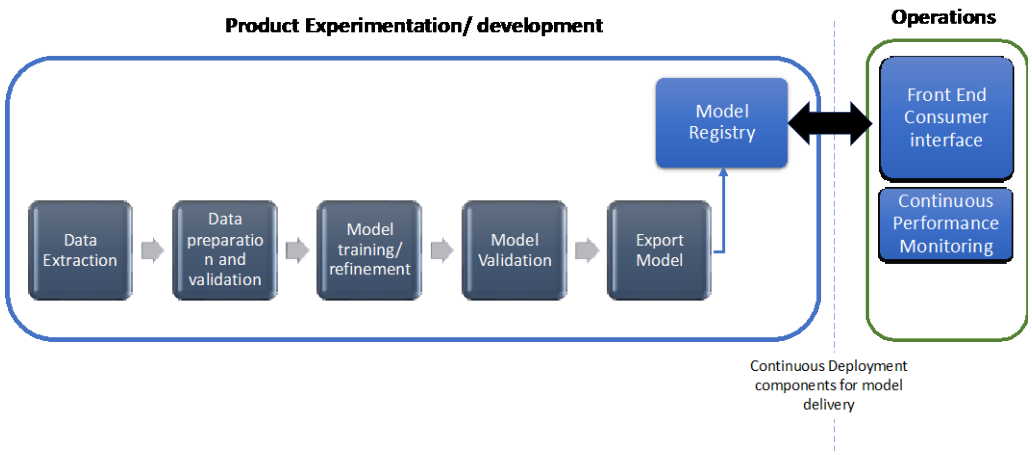


Figure 3. Automated ML Workflow Pipeline. Adapted from Kreuzberger et al. (2022b).

The review of MLOps inform us that integrating an ML model as part of existing operations requires a separate developmental workflow and automated processes to be constructed and integrated into the existing operations. From a systems engineering perspective, some of the key factors influencing the decision on a framework include: 1) ease of exporting trained models for deployment, 2) interoperability with existing systems and between frameworks, 3) licensing cost, and 4) ease of development and maintenance (usability, speed, and programming language). Figure 4 illustrates the multitude of software and hardware options available for each part of the workflow pipeline, as well as platforms that provide a full suite of services, such as Amazon's SageMaker, Microsoft's Azure ML, and Google's Vertex AI platform.

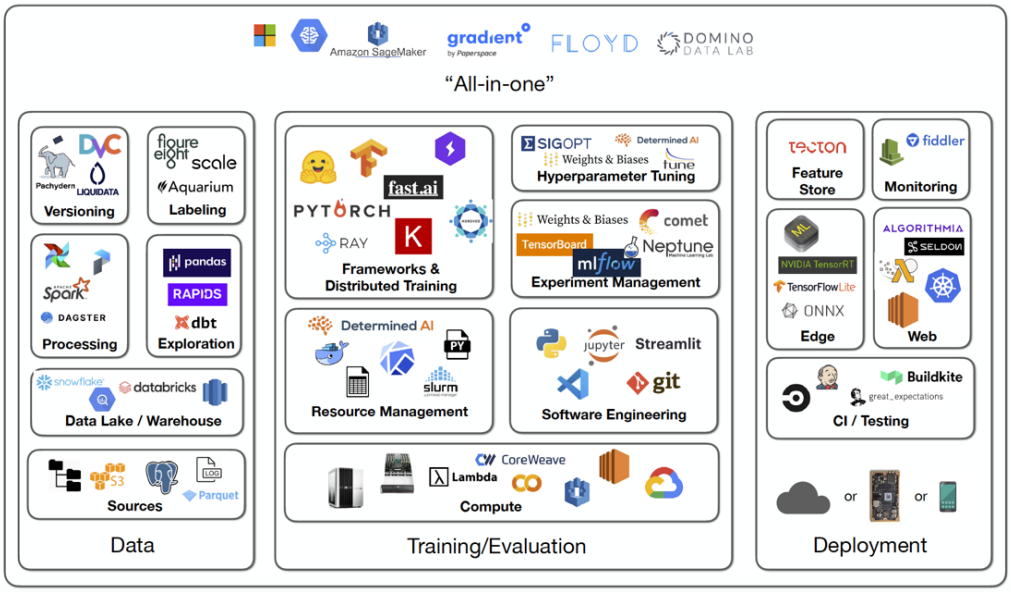


Figure 4. Commonly Used Software in the Various Workflows of MLOps. Source: Karayev et al. (2022).

ML software frameworks provide the necessary abstraction from the low-level implementation, freeing up software developers and programmers to focus on designing, training, and validating models. These ML frameworks are an integral part of MLOps, and hence, careful consideration of the framework is necessary before the commencement of the MLOps workflow. Multiple reviews and surveys have been

conducted to provide a broad overview of existing frameworks, comparative analysis, and benchmarking frameworks for specific ML tasks, such as computer vision and language modeling (Cardoso Silva et al., 2020; Liermann, 2021; Nguyen et al., 2019; Winder, 2019). The most used frameworks for machine learning include PyTorch developed and maintained by Facebook's AI Research Lab (Paszke et al., 2019); TensorFlow and Keras developed by Google's Brain Team (Abadi et al., 2015; Chollet, 2015); MXNet developed by Apache Foundation (Chen et al., 2015); and beginner friendly framework Scikit-Learn (Pedregosa et al., 2011). No one framework provides all the required tools and algorithms; often ML engineers use a combination of tools for the entire MLOps workflow.

4. Using AI/ML to Predict Measurement Noise for KF

In a review on KF with AI techniques by Kim et al. (2022), there are four approaches to integrating AI/ML techniques with a KF:

a. Techniques to tune the parameters of a KF, such as the process noise covariance matrix and measurement noise covariance matrix. This method is mostly used to improve the state estimation in a dynamic environment where external factors that may affect the state of the system are not directly measured by the system. Hence, by using ML techniques to predict the parameters of a KF, the KF can dynamically adapt and adjust its uncertainty matrices, thereby improving its accuracy when the external environment changes.

b. AI/ML techniques predict the errors between the state are estimated by the KF and the ground truth data, and subsequently compensate for error in the KF estimation to provide the final state estimated. During state estimation, the AI predicts the error of the state estimated by the KF and compensates for this error by adding it to the output of the KF.

c. Compared to the previous approach, instead of predicting the error in the KF's estimation, the AI/ML technique predicts the error in the state

estimation first. This state estimation error is subsequently added to the sensor measurements before the new updated KF estimate is produced.

d. Techniques also exist to provide additional measurements for KF, which improves the state estimation. This aims to overcome an imperfect prior mathematical model of state estimation. The AI learns the mathematical model of the state and is subsequently used to generate measurements to be used by a KF. Thus, instead of using the sensor measurements directly, the KF algorithm uses the measurements generated by the AI. This technique is relatively new compared to the previously described techniques.

In this thesis, we explore the first technique—using the AI/ML technique to predict the parameters of the KF. Some notable research and related work done in this area include the following:

The KF algorithm has been integrated with a neural network (NN) which enables it to adapt its parameters when the assumptions built into the KF model are invalid. Bekhtaoui et al. (2017) put forth a Q-learning KF[1] for tracking maneuvering targets. In their research, a reinforcement learning regime is used to learn a policy for deciding which noise matrices are to be used by the KF depending on the sensor measurements of the target. Using Monte-Carlo simulation, the authors found that their proposed methods can provide faster filtering, compared to an Interacting Multiple Models KF algorithm, while preserving the tracking accuracy. Jouaber et al. (2021) solve a similar problem by training a recurrent neural network embedded in the KF to predict additive process noise, thereby modifying the process covariance matrix Q parameter in the KF.

Another parameter of interest is sensor measurement noise represented as covariance matrix R, in the KF algorithm. Like the process covariance matrix, adjustments to matrix R allow the KF to update its probabilistic belief of the

[1] Q-learning is a procedure used to generate a table of state-action pairs with the expected rewards achievable when an action is taken at a given state. This table is called the Q-table. The estimates are updated during the learning by maximizing a given reward function that characterize the cost and benefit of being in a particular state given the history of states visited by an agent.

measurements compared to the estimated state. In 2019, Ullah et al. embedded an NN in a KF to estimate the error in sensor readings. They demonstrated that such a learning module improves the estimation of temperatures by about 10% in varying humidity conditions by taking into account humidity readings during the training of the NN (2019). They furthered their work in 2020 using additional external atmospheric data (solar radiation, wind speed, external CO_2) and internal operational conditions of various actuators in the NN model to filter temperature, CO_2, and humidity readings of a greenhouse's indoor environment.

III. SIMULATING SENSORS DATA FOR DATA FILTERING

This chapter introduces the methodology for simulating sensor data using CMO simulation software and analyzes the dataset generated. Before we delve into the simulation process, we put forth the value of using simulations to generate sensor data and describe a three-phase process workflow adopted for this thesis.

A. PROPERTIES OF SIMULATIONS

The advantage of using computer simulation includes convenience, flexibility, and reproducibility.

(1) Convenience

A simulated dataset enables greater control of the data format output while reducing operational and technology security concerns when exporting operational data. In addition, ML training requires a voluminous dataset for learning; simulation software is best suited for replicating scenarios in quick succession in a fraction of the time required.

(2) Flexibility

Computer simulations afford the ease of changing scenario parameters when and where required. For instance, for our investigation of different weather conditions, using an operational dataset would have limited our investigation to the available weather conditions when the sensors were operating.

(3) Reproducibility

A reproducible experiment provides reassurance to researchers and users on the effectiveness of the methods used. By using a computer simulation, we can replicate the dataset generated and methods used for our experiments, thereby enabling further study into the topic or affirming the results generated from the study. Simulation software allows the ground truth of a target to be recorded, as opposed to a real-life target, whose real position can be difficult to measure accurately.

B. OVERVIEW OF PROCESS WORKFLOW

As illustrated in Figure 1, a three-phase process workflow has been adopted for this research. The three phases are:

1. Generation of sensor dataset using CMO
2. Create data filtering models (KF model and ML-KF model)
3. Assess accuracy of data filtering models

In Phase 1, we use the simulation software CMO to generate a sensor dataset, whereby multiple sensors sense and track a target moving along a designated pathway defined by waypoints in the scenario. The sensor dataset is stored, and the scenario is repeated with slight random variations to the starting position of the target to introduce randomness between each iteration of the simulation run. We use the dataset to build models using AI/ML techniques (Phase 2) and assess the data filtering algorithms (Phase 3).

In Phase 2, an ML-KF model is generated from the training dataset. A train-evaluate-test approach was adopted to determine the best hyperparameters of the machine learning model. This phase is unique as our baseline models using KF do not require any learning.

In Phase 3, the baseline models and the ML-KF model from Phase 2 are evaluated against the desired MOPs, using the test dataset.

The subsequent sections in this chapter elaborate on Phase 1 in detail.

C. SCENARIO DESIGN

This section provides an overview of the simulation software, the parameters and considerations adopted for creating the scenarios for generating the dataset.

1. Overview of CMO

CMO was used to simulate and generate sensor data and the target position in the simulated scenario. The data is generated and stored as comma-separated value (CSV) files. CMO is “Matrix Games’ flagship commercial wargame of modern cross-domain

military operations" (Matrix Games, 2022b). While there are various modes of play in CMO[2], we use CMO as a computer simulation application through the *Scenario Design* mode because it provides the flexibility for designers to create scenarios for experimentation or analysis and simulate the interactions of agents using physics-based models. In addition, CMO's built-in database of capabilities ranges from post-World War II 1940s to modern-day 2020s to a hypothetical next generation. Thus, using CMO as simulation software reduces the complexity required to design sensors for our data fusion experiments. Figure 5 illustrates a slice of the extensive parameters required to define a sensor in CMO, showing the depth of modeling capabilities in CMO. For this thesis, we utilized Command Professional Edition v2.0—a professional-oriented superset of CMO for data generation. To prevent confusion, we continue to use CMO throughout this paper to refer to the simulation software.

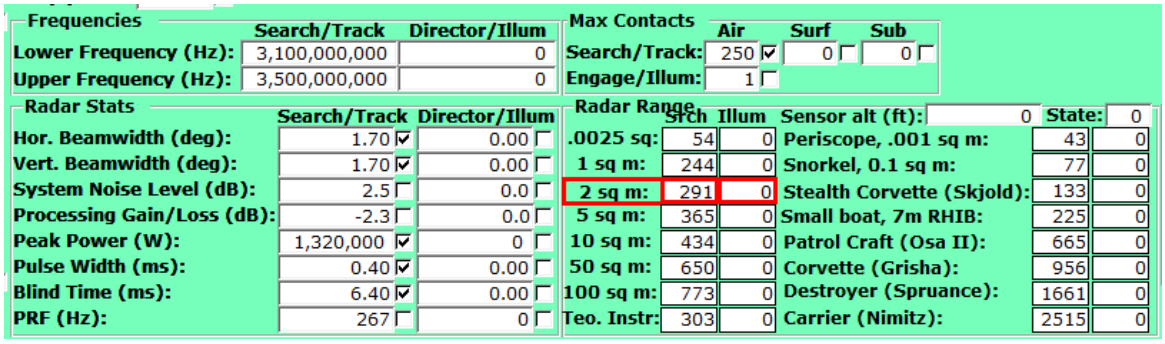


Figure 5. Screenshot of the CMO Database Editor of a Built-in Radar sensor

2. Physics and Stochastic Modeling in CMO

Under the hood, a user designs the following: 1) parameters of the mission and environment; 2) purpose, attributes, and disposition of agents within the environment;

[2] Based on the user manual for CMO, there are four game modes available (Matrix Games, 2022b): *Campaign* allows users to play multi-mission campaigns; *Quick Battle* allows users to enter a pre-defined scenario and role play the game to execute the mission; *Normal Play* is like *Quick Play* but provides flexibility for players to select a side to execute the mission; *Scenario Design* allows user to define their own scenario.

and 3) the interaction behaviors of agents (e.g., to determine if an agent should engage a certain category of agents).

During each simulation run, CMO uses its built-in, physics-based model and game mechanics to simulate the movement and interaction between agents and the environment (e.g., poor visibility due to a heavy rainstorm). While the algorithms and mathematical equations used are not made publicly available, *Command Professional Edition User Manual Version 2.0* (Matrix Games, 2022c) provides insight into the factors taken into consideration for evaluating sensor detections. For instance, to determine if a surface ship can be detected by ground-based radar, the simulation engine is said to account for terrain and sea clutter, presence and geometry of jamming sources, weather effects, properties of the radar (such as pulse width, beamwidth, power output, and operating frequency), and location of the target relative to the sensor.

3. Considerations and Constraints in the Design of the Scenario

The following considerations and constraints were defined so that the effectiveness of the data filtering algorithm can be assessed and variations in the scenario do not influence the performance of the algorithm:

a. At least two simulated sensors should be included, each from different sensor domains; for example, radar and Electro-Optical (EO) camera covering electronic and imagery intelligence, respectively.

b. All sensors should be able to track the target simultaneously.

c. Targets should move between pre-defined waypoints.

d. Variations to the target's start point should be made to provide slight differences between each sample in the dataset.

4. Scenario Design—Sensors

Figure 6 illustrates the global view of the scenario created, and Table 1 defines the sensors used and their technology domain. The scenario consists of the target (yellow icon) and four sensor platforms (green icons). To mimic the data fusion among

heterogeneous data sources and entities dispersed across the battlespace, we placed four sensors at different locations on the map while allowing the sensor detection ranges to be sufficiently overlapped to maintain track of the target. This design decision allows the target to be detected by all the sensors as it carries out its planned mission. In essence, this creates a synthetic test range for conducting measurement and analysis.

Figure 6. Scenario with Target and Four Sensor Platforms.

Table 1. Description of Simulated Sensors in Scenario

Sensor Name	Sensor Domain	Technology
AN/KAX-2 SeaFLIR II [EO]	IMINT	The SeaFLIR II is developed by FLIR Systems and is equipped with a color Charge-Coupled Device-Television (CCD-TV) camera.
AN/ KAX-2 SeaFLIR II [IR]	IMINT	Like the SeaFLIR II [EO] above, the IR version consists of a mid-wave IR thermal imager for imagery of targets.
Bridgemaster E ATA	ELINT	The Bridgemaster E series radar developed by Northrop Grumman Sperry Marine B.V. (2005) operates in the S and X bands and is equipped with automatic tracking aid to track up to 60 surface objects moving up to 150 knots.
AN/ SLQ-32(V2) [ESM]	Passive ELINT	Developed by Raytheon Technologies, the AN/SLQ-32 (Variant 2) electronic support measures (ESM) systems are passive shipboard electronic warfare (EW) systems for early warning against, identification of, and direction finding of targets.

Initially, we attempted to group all the sensor platforms into a single player. However, after observing the initial data throughput, we observed that CMO definitively shares detection information between sensor platforms. This is a natural design for a typical mission, as the sharing of intelligence between sensors is expected for making decisions and taking actions. For our purpose, however, this results in each sensor having the same estimated target location instead of the sensor's independent estimate. To mitigate this effect, we define each sensor platform as an independent player in the simulation.

5. Scenario Design—Target

A single target is given a *Sea Control Mission* to patrol between the three reference points (RP-41, RP-42, and RP-43 in Figure 6) in a repeatable loop within the timeframe of the simulation. The reference points were defined to satisfy the constraint of maintaining the target's detectability by the sensors. The expected movement of the target is as follows:

a. When the scenario is initialized, a target is randomly generated within the boundaries of a navigation area—a triangle defined by the reference points. To accomplish this, a Lua programming language script was used to set the start point of the target. Refer to Appendix A for the script used.

b. The target is expected to move from the start point to RP-41, RP-42, and then RP-43.

c. Upon reaching RP-43, the target moves towards RP-41, RP-42, and then RP-43 again.

6. Scenario Design—Weather Conditions

Sensor performance in CMO is affected by both terrain and weather—average temperature, rainfall rate, visibility, and wind/sea state. Based on CMO's user manual (Matrix Games, 2022c), we learn that high temperature decreases IR sensor range more than it does for EO; rainfall rate degrades the performance of visual, IR, and laser sensors; visibility—due to cloud cover—affects line-of-sight sensors such as visual and IR sensors; and sea states affect the performance of radar.

Hence, by changing the weather conditions in CMO, we would expect the EO and IR FLIR sensors to be most affected and thus least effective in detecting the target's position when the temperature is warm, rainfall is high, and sky is overcast. To that end, we designed two sets of weather condition settings. Figures 7 and 8 illustrate the nominal and sub-optimal weather conditions for sensors, respectively.

Figure 7. Weather Settings for Ideal Weather Conditions.

Figure 8. Weather Settings for Sub-optimal Weather Conditions for EO/ IR sensor.

7. Simulation Runs

For each scenario setting, 100 iterations were run using PowerShell script (see Appendix A for PowerShell script used), and the entities' positions and sensor detections were logged into CSV files. Each iteration of the simulation ends when the simulation time exceeds two hours. Figure 9 illustrates the scenario time settings used in CMO. As the scenario contains only four agents and a limited number of mathematical calculations, each iteration of simulation runs took an average of five to seven minutes to complete on a laptop with Graphics Processing Unit (GPU).

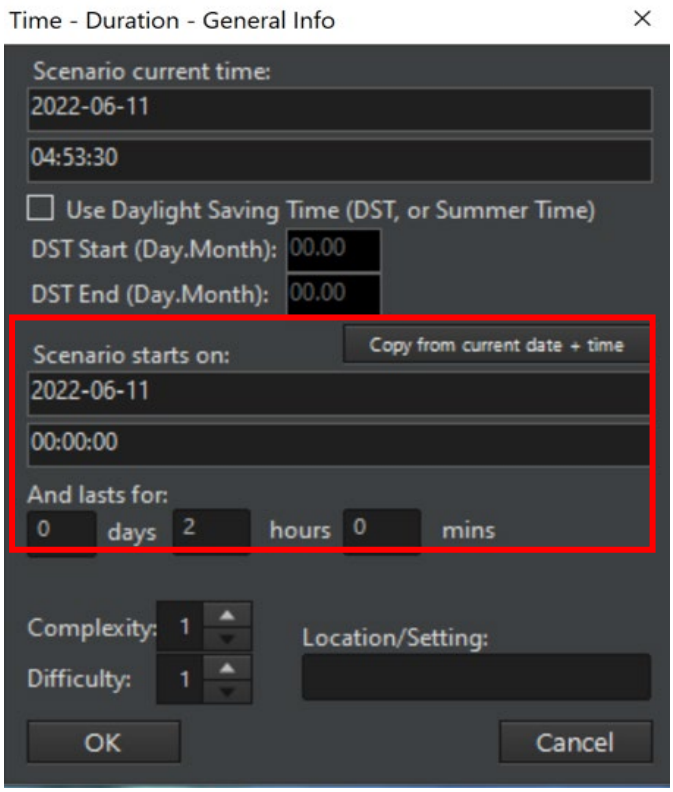


Figure 9. Simulation Runtime Setting

D. DATASET

In this section, we describe the dataset obtained from a simulation execution. Each iteration generates two sets of files—the unit position of the target and each sensor's detection attempt (i.e., the sensor's success in detecting other objects in the scenario). A data dictionary of the fields, description, data type, and sample value of each field is expanded in Appendix B.

1. Overview of Dataset

A dataset is generated for each weather condition settings; for each set, five CSV files were generated per iteration of the simulation run, giving a total of 100 CSVs for the target unit positions and 400 for sensor detection. The *TimelineID* associates the simulation runs with files so that sensor readings and target positions are aligned to the same run. Following, we describe the pre-processing required for each type of file, which

is achieved using the Pandas Data Analysis Library written in the Python programming language (2020).

a. Pre-processing Carried Out for All Files

For all data files generated, the following data pre-processing steps were taken:

(1) Data headers and their values that are deemed unnecessary for the subsequent phases of the experiment were removed.

(2) All *Time* data are represented in the number of seconds elapsed in the scenario instead of the default time format (*HH:MM:SS*).

(3) A single copy of the duplicated data entries is kept, thereby reducing the size of the overall dataset.

(4) All geodesic longitude and latitude were converted from degrees to East, North, Up (ENU) representation using *Python 3-D Coordinate Conversions [Computer Software]* (2022). An ENU representation takes reference from a longitude and latitude (longitude = -118.992 degrees and latitude = 33.660 degrees) and each geodesic point is calculated for this reference point. The projection of geodesic points into Cartesian coordinates allow us to calculate distance between points.

b. Pre-processing Specific for Sensor Detection Files

In addition to the pre-processing steps just listed, the following steps were carried out for sensor detection files. Sensor data contains attempted detection of all players (simulated sensors), except itself, in the simulation. This means that there are additional detection attempts that are not related to the target. Hence, we removed all sensor data where the object detected did not correspond to the target of interest. In addition, we conducted checks on the dataset to make sure that any duplicate entries corresponding to the same time are also removed and that the frequency between each sensor detection attempt is constant.

2. Data Analysis—Target Position

The simulation logged the target's position per simulated second, giving a total of 7,202 entries.[3] The data fields stored are *TimelineID*, *Time*, *UnitID*, *UnitName*, *UnitClass*, *UnitLongitude, UnitLatitude*, and *UnitCourse*. Figure 10 uses the *Time*, *UnitLongitude, UnitLatitude,* and *UnitCourse* fields to visualize the movement of the target. The green dot represents the randomly generated initial position of the target while the black dot represents the final position of the target. Graphing the trajectory of the target is useful for visualizing the position prediction by a data filtering algorithm. This is integrated with the corresponding prediction errors against the ground truth position in the Chapter V.

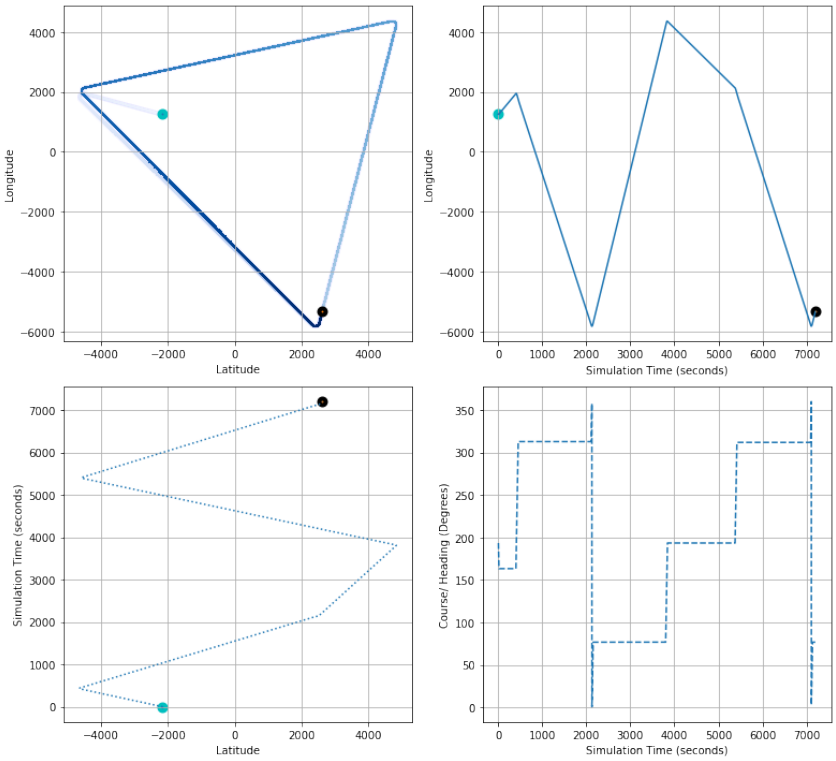


Figure 10. Visualization of Target's Movement during the Simulation.

[3] Although the maximum simulation run-time is two hours (7,200 seconds), two additional seconds were logged, one prior to the start of the simulation (zeroth second) and another after the simulation has ended. These additional position and sensor detection data are logged by CMO directly.

3. Data Analysis—Sensor Detection

The data field in the sensor detection dataset consists of four parts:

- General simulation parameters: TimelineID, Time
- Information about the sensor: SensorID, SensorName, SensorParentLongitude, SensorParentLatitude, SensorParentAltitude_AGL
- Information about the target the sensor is attempting to detect: TargetID, 'TargetName, TargetLongitude, TargetLatitude, TargetAltitude_AGL_m, TargetRangeSlant_nm.
- Outcome of detection: DetectionResult, DetectionAOU.

a. Sensor Detection Period

The sensor detection period is the time between each sensor detection attempt. It is a characteristic of each sensor and is independent of the weather setting. Table 2 presents the sensor detection period (Detection Period) for each sensor used in the simulation. The ESM passive sensor has the least number of detections, as it takes a longer time for the passive sensor to complete a detection cycle, while the radar—a high-frequency active sensor—has the most detection attempts. We also note that the arrival of the first sensor detections differs between sensors (First Detection Time). The different arrival times and detection periods mean that the algorithms need to handle the different arrival periods of the data stream.

Table 2. Sensor Detection Period and Start Time

Sensor	Detection Period (seconds)	First Detection Time	Number of Detections per Simulation Run
Radar	2	0	3601
EO	10	2	720
IR	10	2	720
ESM	20	13	360

b. Detection Success

The *DetectionResult* allows us to understand if the detection attempt by the sensor is a success or a failure. A failed detection attempt means that the sensor did not successfully detect the target, and the target's position is estimated based on its last successful detection attempt. Table 3 presents the average proportion of successful detection across all runs in the different weather conditions for each sensor (as defined in Figures 7 and 8). In normal weather conditions (15 deg C, no rain, sky is clear, calm seas), all the sensors have a 100% detection success rate, which is expected since our scenario is designed to have the target maneuver within the sensor detection ranges. In contrast, with extreme weather conditions (33 deg C, heavy storms, sky is full with thick clouds, seas 4/10 on Beaufort scale), the IR sensor proportion of successful detection is almost halved, indicating that the impact on the IR sensor is the most significant. While we would have expected the impact on sensor detection to be broad-based and affect the radar and ESM sensors due to the impact of weather on electromagnetic wave propagation and the impact of low visibility on the performance of EO and IR visual sensors, it is not evident in the proportion of detection success in the extreme weather scenario. This may be due to the relatively close distances between the target and the sensors which make the extreme weather effects negligible, or the sensors are well-equipped to adapt their characteristics and parameters (such as power and pulse width of the radar to improve range resolution) based on the weather.

Table 3. Sensor Detection Success

Sensor	**Detection Success (%)**	
	Normal Weather	**Extreme Weather**
Radar	100 +/- 0.0	99 +/- 0.0001
EO	100 +/- 0.0	100 +/- 0.0
IR	100 +/- 0.0	53.62 +/- 3.86
ESM	100 +/- 0.0	100 +/- 0.0

c. Average Residual from Target's Position

The accuracy of a sensor is an MOP and can be calculated using the root mean squared error (RMSE). A higher RMSE corresponds to poorer performance while a lower RMSE suggests that the sensor's estimation of the target's state is closer to the ground truth. Before we use the dataset for data fusion, it is necessary to analyze the performance of each sensor so that subsequent analysis allows us to ascertain the improvement in state estimation accuracy by each data fusion algorithm.

To calculate the RMSE, we compare the longitude and latitude of the target detected by the sensor and the target's actual location at the corresponding timestep. The following equations were used to calculate the RMSE.

$$error_i, Error\ for\ timestep\ i = \sqrt{\left(lon_{sesnor} - lon_{target}\right)^2 + \left(lat_{sensor} - lat_{target}\right)^2}$$

$$RMSE\ for\ each\ iteration\ with\ i = 1\ to\ N\ timesteps = \sqrt{\frac{1}{N}\sum_{i=1}^{N} error_i^2}$$

$$Average\ Sensor\ Performance\ across\ 100\ iterations = \frac{1}{M}\sum_{j=1}^{M=100} RMSE_i$$

Subsequently, the performance of the sensor across all simulation scenarios is averaged to provide the mean error and standard deviation. Table 4 presents the average performance of each sensor in different weather conditions.

Table 4. Average Sensor Detection Error in Different Weather Settings

Sensor	Average RMSE (meters)	
	Normal Weather	Extreme Weather
Radar	6.1725 +/- 0.001865	6.1710 +/- 0.0019342
EO	6.1635 +/- 0.001850	6.1640 +/- 0.0018425
IR	6.1656 +/- 0.001895	6.1637 +/- 0.001844
ESM	0.4215 +/- 0.001416	0.4220+/- 0.001413

It is interesting to note that the average performance of the sensor did not worsen as much as expected, and the sensors have a relatively low sensor detection error (a maximum of 6.17 meters or 0.0033 nm). Comparing the sensors, Radar, EO, and IR have similar performance in both weather conditions, while ESM has the best performance (lowest error) among the sensors (0.42 meters or 0.00023 nm). This is a deviation from real-world performance as ESM sensors often provide an area of uncertainty about the target's position, translating to higher error. As CMO's sensors models are proprietary knowledge, we are limited by the sensor dataset provided from the simulation.

It is also interesting to study the IR performance in detail since it has a high detection failure rate in extreme weather conditions, with marginal changes in sensor detection performance. In CMO, a detection is considered a failure when the sensor made a contact with the target but was not successful in classifying it. Further analysis of the average detection error based on the sensor attempt shows that the target's error when the detection is a failure is statistically significantly higher at a 95% confidence level than that when the detection is successful. Thus, the extreme weather effect does have a significant impact on the sensor detection error for IR sensor.

E. LIMITATIONS OF DATASET

1. Insignificant Improvement between Simulation Settings

The exploratory data analysis in Section C illustrates that the dataset has a fairly accurate independent sensor estimation of the target's position, even in the case of extreme weather scenario settings. In addition, the effects of extreme weather affected

only the IR sensor, when the performance of EO, IR, and radar sensors was expected to be affected as well. Thus, to reduce the complexity and further focus our efforts on this research, in subsequent chapters, we only consider the dataset in the normal environment to illustrate the improvement in target position estimation. This means that the AI/ML algorithms utilize only the normal weather dataset for training and evaluating the accuracy of the model.

2. Absence of Measurement Errors

The family of KF algorithms requires covariance of the sensor measurement to represent measurement uncertainty. In our dataset, only the ESM sensor provided an Area of Uncertainty (AOU), while the rest of the sensors did not. Despite our effort to utilize AOU information, we decided that all sensor datasets should provide the same type of information to reduce the complexity and confounding factors in our experiments. Specifically, when using the dataset with the KF algorithm, we would be required to estimate the sensor noise present in the measurements. The methodology to estimate sensor noise is elaborated upon in Chapter IV.

IV. MODEL GENERATION METHODOLOGY

In this chapter, we describe the methodology to create models that take in sensor estimates of the target position in the battlespace and output a refined estimate of the target position. First, we describe our baseline model using the KF algorithm. Second, we describe an ML model that is used to estimate the error from the KF algorithm to improve the performance of the KF algorithm. Appendix C list all Python packages used in the development of our source code, and Appendices D, E, and F are the scripts and output from running the scripts using a Jupyter Python Notebook.

A. KALMAN FILTERS

The KF algorithm is a recursive estimator that predicts the future state of a system based on its previous state (Faragher, 2012). It operates on the current sensor measurements and previously filtered measurement data; thus, requiring a reduced amount of memory and is a fast and efficient real-time estimator. Generally, KF algorithms operate in a two-step procedure—the *predict* step and the *update* step (Labbe, 2022). This two steps are executed sequentially in a single iteration of the algorithm. In our case, we execute the algorithm at every time step of the simulation to estimate the target's state in the subsequent time step.

The predict step uses the system process model to predict the state at the next time step and to adjust its probabilistic belief to account for uncertainty and prediction errors. The update step uses the sensor measurement to update its estimation of the system state. While a sensor model is used in the update step to account for uncertainty and errors in the measurements. Following, we elaborate on the mathematical symbols and equations used for each step, and the parameters used for our model.

1. Predict Step

Table 5 presents the mathematical symbols used in the predict step.

Table 5. Symbols Used in the Predict Step

Symbol	Variable	Definition
X	State of system	A vector of state parameters representing the position (latitude/longitude), velocity, and acceleration of the target.
P	State covariance matrix	The state covariance matrix represents the uncertainty of the corresponding state parameters. The diagonal terms of P are the variances associated with the state parameters, while the off-diagonal terms are the covariances between terms in the state vector, which informs us of how much the state vector terms vary from each other.
F	State transition matrix	The state transition matrix applies the effect of each state parameter at time $k-1$ on parameters at time k.
w	Process noise	The process noise are random errors associated with noisy control inputs to the system. It is a vector of random errors assumed to be drawn from a zero mean multivariate normal distribution with covariance Q.
Q	Process noise covariance	The process covariance represents the uncertainty (process noise) in the transition from the current state to the subsequent state.

We are interested in predicting the position and velocity of the target, and hence, X_k is the state vector representing the latitude x_k, longitude y_k, and the respective velocities $\dot{x}_k$ and $\dot{y}_k$ in m/s and acceleration $\ddot{x}_k$ and $\ddot{y}_k$ in m/s^2 of the target at time $k = 0,1,2,3,$ .... (seconds). The system dynamics model from time $k-1$ to time k in a finite timestep Δt uses a combination of linear equations with Gaussian noise with mean zero and covariances defined by matrix P:

$$X_k = [x \quad y \quad \dot{x} \quad \dot{y} \quad \ddot{x} \quad \ddot{y}\,]^T$$

$$X_k = F_{k-1}X_{k-1} + w_{k-1}$$

where $[.]^T$ is the transpose operation.

The state covariance matrix P, represents the uncertainty of our state variables, and will be updated in the update step when measurement inputs are considered. Since the target has a cruising speed of approximately $7\,m/s$ (15 kts), the uncertainty in the measurement of position data in one second would be approximately 65 m (0.035 nm). We further assume that the uncertainty in velocity measurement would be $0.5\ m/s$ (1 kt) and the uncertainty in acceleration $0.5^2\ m/s^2$. The resulting initial covariance matrix, P_0 is

$$P_0 = \begin{bmatrix} 65^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 65^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5^4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5^4 \end{bmatrix}$$

Newton's Equations of Motion provide us with the state transition equations. While the target in the scenario moves at a constant speed for most of the scenario, the change in direction implies that the target is moving at varying velocity during the simulation, but the rate of change of velocity is constant. Hence, a constant acceleration model is used to model the system dynamics.

With constant acceleration, the kinematic equations describing the change in position and velocity of the target at time k in a timestep Δt are given by

$$x_k = x_{k-1} + \dot{x_{k-1}}\Delta t + \frac{1}{2}\ddot{x}\Delta t^2$$

$$y_k = y_{k-1} + \dot{y_{k-1}}\Delta t + \frac{1}{2}\ddot{y}\Delta t^2$$

$$\dot{x_k} = \dot{x_{k-1}} + \ddot{x}\Delta t$$

$$\dot{y_k} = \dot{y_{k-1}} + \ddot{y}\Delta t$$

$$\ddot{x_k} = \ddot{x_{k-1}}$$

$$\ddot{y_k} = \ddot{y_{k-1}}$$

The resulting state transition matrix F is

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{\Delta t^2}{2} \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The prediction of state parameters at time k is given by

$$\hat{X}_k = F_{k-1}\,\hat{X}_{k-1}$$

Considering the Gaussian process noise $w_{k-1} \sim N(0, Q_{k-1})$, the updated state covariance matrix P_k is

$$P_k = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1}$$

2. Update Step

Using the defined system dynamic model, the KF algorithm uses the previous measurements to predict the system state at the next time step k. This state estimation $\hat{X}_k$ is then further refined in the subsequent update step by considering the measurements. The update step refines two key components of the prediction equation: 1) the prior state estimation is refined using the Kalman gain and the residual between state estimates and measurements; 2) the uncertainty associated with the refined system state estimate. Table 6 presents the mathematical symbols used in the update step.

Table 6. Symbols Used in the Update Step

Symbol	Variable	Definition
z	Measurement	Sensor's estimation of target's longitude and latitude in ENU representation
y	Residual	The difference between measurement z and the predicted state from the prediction step $\hat{X}_k$
K	Kalman Gain	The amount of correction applied by the KF algorithm to measurements to make the measurements less noisy
R	Measurement noise covariance	The uncertainty associated with the measurements for each state parameters

Symbol	Variable	Definition
		measured. The diagonal of R represents the variance in the respective sensor measurements, while the off-diagonal elements represent the variances in measurements between different sensors
H	Measurement function	Projects the state parameters X into the measurement space. Since the sensors are not measuring accelerations or velocities directly, we convert the state space to measurement space by removing those terms that are not measured

a. Sensor Measurements and Noise

Since the sensor measurement arrives at different periods and starts at different time interval, we update the KF when a sensor measurement has arrived at the filter. This is possible since we assume that all sensor measurements are independent measurements of the target. The measurements from our sensors are the longitude and latitude of the target, and z_k is a column vector representing the measurements from the sensor $i \in [Radar, EO, IR, ESM]$ at a specific timestep $k = 0,1,2,\ldots$

$$z_i = [x_i,\ y_i]^T$$

The associated uncertainty in the sensor measurements is defined in the covariance matrix R_i. However, as noted in Chapter III, the uncertainty in the sensor measurement is not an output in the sensor data in CMO. Thus, to estimate the uncertainty in the sensor measurements, we derive the variance in the sensor measurements of the longitude and latitude from the dataset. We further assume independence between the sensors' detections and between the longitude and latitude variables. Hence, R_i is a 2×2 matrix for sensor i with the diagonals representing the variances in the uncertainty measurement of longitude and latitude, and the off diagonals are zeros. Mathematically, standard deviation, σ, is given by $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$.

The standard deviation in sensor measurement error across all detection is presented in Table 7.

Table 7. Sensor Measurement Noise Derived from Dataset

Sensor	Sensor Noise (Standard Deviation in Error, meters)	
	Longitude	**Latitude**
Radar	1.7091	1.9157
EO	1.7113	1.9194
IR	1.7113	1.9193
ESM	0.2095	0.2104

b. Measurement Function

To convert the state space X to measurement space z, we define measurement function H as:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

c. Kalman Gain

In KF, the sensor measurements are used to inform the algorithm of the difference in the estimated prediction and the measured position of the target, informing it of the amount required to correct its prediction in the subsequent timestep k. The Kalman Gain K_k is calculated as

$$K_k = P_k(H_k P_k H_K^T + R_k)^{-1}$$

d. Update State Estimation

The residual y_k tells us of how far the estimated state in the predict state is from the measurement. The updated state estimation is given by

$$y_k = z_k - H_k \hat{X}_k$$

$$\hat{X}_k = \hat{X}_k + K_k y_k$$

e. Update State Covariance Matrix

The state covariance matrix P is updated to reflect the changes in uncertainty with the Kalman Gain, K

$$P_k = (I - K_k H_k) P_k$$

B. CREATING KF BASELINE MODEL USING FILTERPY

We use the FilterPy python package to implement the KF. The standard KF function was used to generate the KF's estimate of the longitude and latitude at every timestep of simulation. The KF is updated when a sensor reading(s) has arrived at the timestep before a state estimation is made (prediction step). We use the same formula defined in Chapter III to calculate the squared error between the target and KF's estimate, and subsequently calculate the RMSE and standard deviation of the error of each simulation run. The average error across the 100 simulation runs is used to compare the performance of the model.

The algorithm used in the experimentation set up is as follows:

1. Define a simulation iteration and retrieve the corresponding dataset for sensor detection and unit position.
2. Initialize the KF with measurement function, state transition matrix, and process covariance matrix.
3. Initialize KF's target initial position with the first sensor reading.
4. For each timestep k subsequently:
    - Get a prediction of the target's state estimate from KF and save the estimate.
    - Retrieve sensor detection at that timestep k, and extract the longitude and latitude of the sensor detection: $z_{i,k} = \{x_i, y_i\}_k$.
    - Update KF measurement error matrix of sensor r_i.
    - Update KF with a measurement $z_{i,k}$ from sensor i.

- Repeat until there is no additional sensor detection at timestep k.
- Advance to the next timestep.

Chapter V presents the results of the KF model.

C. MACHINE LEARNING MODELS

In this section, we describe the methods used to create an ML model that is used to estimate the parameters of a KF. Specifically, we take inspiration from the work done by Ullah et al. in 2019 and 2020, whereby a neural network was used to estimate the error matrix to be used in the update step of the KF and thus improve the estimated state of the object. In the subsequent section, we formulate our desire to create an NN using the ML approach and expand in detail the procedures used in the creation and tuning of an ML model.

1. Formulation of ML Problem—Modeling Uncertainty

We desire to use a NN to estimate each sensor error matrix R_i instead of using the average error estimated from the dataset. This is beneficial for the KF algorithm as the error matrix informs the KF of the uncertainty in the sensor's measurement, and hence, a representative error matrix would shift the KF's probabilistic belief of the sensor's measurement accordingly. Therefore, the goal of the NN is to approximate a function to predict the sensor's measurement uncertainty when given the sensor's measurement of the target's longitude and latitude.

A model's uncertainty can broadly be classified with both *epistemic* uncertainty and *aleatoric* uncertainty (Bishop, 2006; Kendall & Gal, 2017; Seitzer et al., 2022). Epistemic uncertainty can be reduced with increasing data points, thus improving the probabilistic belief of the model. In comparison, aleatoric uncertainty is embedded as part of the information which our data is unable to explain. Aleatoric uncertainty can be independent of the input space and can be constant (*homoscedastic*) or vary with the input (*heteroscedastic*). In our ML task, we are predicting the sensor's variance, which is a natural physical characteristic of the sensor.

Next, we wish to derive the loss function for optimization of the NN. Goodfellow et al. (2016) illustrated that the cost function of a neural network is to minimize the cross-entropy loss between the probability distribution inferred from the training data and the model's distribution. The negative log-likelihood of a model $p(y|x;\ \theta)$ determines the model's cost function i.e., $J(\theta) = -\log\ p(y\ |x;\theta)$.

a. Heteroscedastic Interpretation

Adapting from Nix & Weigend (1994), the sensor i's measurement function f is

$$f(x) = \ x + \ \epsilon(x)$$

where $\epsilon \sim N\big(0, \sigma^2(x)\big)$ is the additive Gaussian noise with zero-mean and variance, and x is the ground truth location of the target. The input-dependent variance suggests sensor noise is dependent on the actual location of the target. This is a suitable assumption since a sensor measurement may vary as the distance between the target and the sensor varies.

Given training data of sensor i, $\{\big(x^{(1)}, y^{(1)}\big), \big(x^{(2)}, y^{(2)}\big), \dots, \big(x^{(n)}, y^{(n)}\big)\}$, consisting of: x, location of the target, and y, the sensor measurements at some time $t\ \in \{1, .., n\}$ the maximum likelihood estimate (MLE) of the variance of a dataset can be derived as follows:

- The probability distribution of sensor measurement y given x is normally distributed

  $$p(y|x) = N\big(x, \sigma^2(x)\big) = \frac{1}{\sqrt{2\pi\sigma^2(x)}} exp\left(-\frac{(y-x)^2}{2\sigma^2(x)}\right)$$

  In other words, the uncertainty of locating a target is a Gaussian distribution around its actual location and it is assumed that the uncertainty varies with x.

- Assuming that each example in the training dataset is independent and identically distributed, then the negative log-likelihood of $p(y|x)$ is

$$
\begin{aligned}
&- log\ p(y\,|x;\theta) \\
&= -\sum_{i=1}^{n} log\, p\left(y^{(i)}\middle|x^{(i)};\theta\right) \\
&= -\frac{n}{2} log\, \sigma_\theta^2(x) - \frac{n}{2} log(2\pi) + \sum_{i=1}^{n} \frac{\left(y^{(i)} - x^{(i)}\right)^2}{2\sigma_\theta^2(x)} \\
&= -\frac{n}{2} log\, \sigma_\theta^2(x) + \sum_{i=1}^{n} \frac{\left(y^{(i)} - x^{(i)}\right)^2}{2\sigma_\theta^2(x)} + constant
\end{aligned}
$$

- Ignoring the constant terms, the cost function J to optimize the model parameters θ is

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(\frac{\left(y^{(i)} - x^{(i)}\right)^2}{2\sigma_\theta^2(x)} + log\, \sigma_\theta^2(x)\right)$$

  where $(y^{(i)} - x^{(i)})^2$ is the squared error between the sensor's measurement, $y^{(i)}$ and the target's actual location, $x^{(i)}$, and $\sigma_\theta^2(x)$ is the predicted variance from our NN with parameters θ.[4]

b. Homoscedastic Interpretation

If we relax the assumption that sensor measurement noise is parameterized based on the actual location of the target x, then σ_θ^2 is a constant, and the MLE of the variance is $\sigma^2 = \frac{1}{n}\sum_{t=1}^{m}\left(y^{(t)} - x^{(t)}\right)^2$.

Hence, we use the following cost function–the difference between the underlying variance σ^2, and the model's prediction σ_θ^2–for the training of the model θ:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\|\sigma^2 - \sigma_\theta^2\right\|$$

[4] The derivation of our negative log-likelihood cost function is different from that in Kendall & Gal (2017) and Seitzer et al. (2022), as we do not use the NN model to predict the ground truth location of the target.

$$= \frac{1}{n}\sum_{i=1}^{n} \left\| \left(y^{(i)} - x^{(i)}\right)^2 - \sigma_\theta^2 \right\|$$

where ||.|| is the L1 norm.

We use the cost function resulting from homoscedastic interpretation in the subsequent experiment set up and posit the heteroscedastic cost function for future work.

2. ML Experiment Framework

An ML Experiment Framework describes the operation phases involved in creation of an ML model. Figure 11 illustrates the three-phase framework. It closely mimics an MLOps workflow pipeline described in Kreuzberger et al. (2022b).

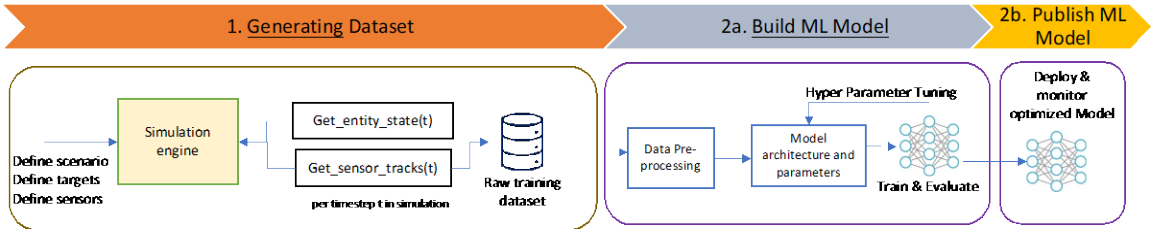


Figure 11. ML Experiment Framework

The following subsections describe each phase in detail.

3. Phase 1: Generation of Dataset

A sensor dataset, one for each sensor, was generated using CMO simulation software as described in Chapter III. The dataset was subsequently split into input variables ($\boldsymbol{x}$) and the output variable (y). Only the normal weather dataset was used.

We used the longitude, x_1, and latitude, x_2, as the input variables to the NN model; hence, at time t, the measurement by sensor i is $\boldsymbol{x_{i,t}} = \left[x_1^{(t)}, x_2^{(t)}\right]$.

The variable to be predicted is the sensor measurement variance for each physical dimension (longitude and latitude). This is given by the squared error between the sensor

measurement and the ground truth target location ($\boldsymbol{x}_t^*$) along each of the physical dimensions at time t: $y_{1,i,t} = \left(x_{1,t} - x_{1,t}^*\right)^2$ and $y_{2,i,t} = \left(x_{2,t} - x_{2,t}^*\right)^2$.

4. Phase 2: Build ML Model—Creating a Neural Network

At the heart of the framework is the development of ML models to meet the objectives of the task. The key activity in this phase includes the following steps.

a. Data Preparation for ML Experimentation

In this phase, the data is further split into three subsets for training, validation, and testing of the model. Subsequently, we normalize the training set, so that each input variable to the model is within the range of zero and one. We use the pre-processing package in Scikit-Learn (Pedregosa et al., 2011) to accomplish the data preparation.

The training set allows the model to update the values of its parameters while the validation set is used to score the model so that the learning algorithm can determine if the model is overfitted to the training set. In other words, we aim to determine if the model has learned (or memorized) the training set to the extent that it was unable to perform when given a dataset that it has not seen during training. In addition, the validation set is used to compare models trained with different hyperparameters, thereby determining the best hyperparameter for each model. Since the model is trained on the training set and tuned using the validation set, the test set provides an unbiased evaluation of the model. To that end, the training set is 70% of the entire dataset while the validation and test sets are 15% each. A random number generating seed is set so that each model is trained on the same subset of the dataset.

We scale the raw dataset so that each input variable is within the range of 0 to 1 according to the following formula

$$x_{scaled} = \frac{x - min(x)}{max(x) - min(x)}$$

The minimum and maximum range of the variable is sampled from the training set, instead of the entire dataset, to prevent data leakage during the training process. By

scaling the dataset, each variable is in the same range and each variable receives equal weighting by the neural network.

b. NN Architecture and Parameters

A dual-head neural network (DHNN) is designed to estimate the sensor error for longitude and latitude (Figure 12). The NN is a fully connected NN with two hidden layers, consisting of an input layer with two neurons for the longitude and latitude of the sensor measurement, and an output layer with two neurons for the estimated error in sensor readings for longitude and latitude, respectively. The h_1 hidden layer is a shared layer between all neurons, while the h_2 hidden layer is disconnected from the other output nodes. This architecture design decision was adopted so that the output nodes can learn the nuances in the data specific to longitude and latitude independently, while not completely foregoing the interdependence that may exist between the dimensions. All neurons in the NN use rectified linear units (ReLU) as their activation function. The number of neurons in each hidden layer is a hyperparameter to be tuned independently, as elaborated in Step 4 below.

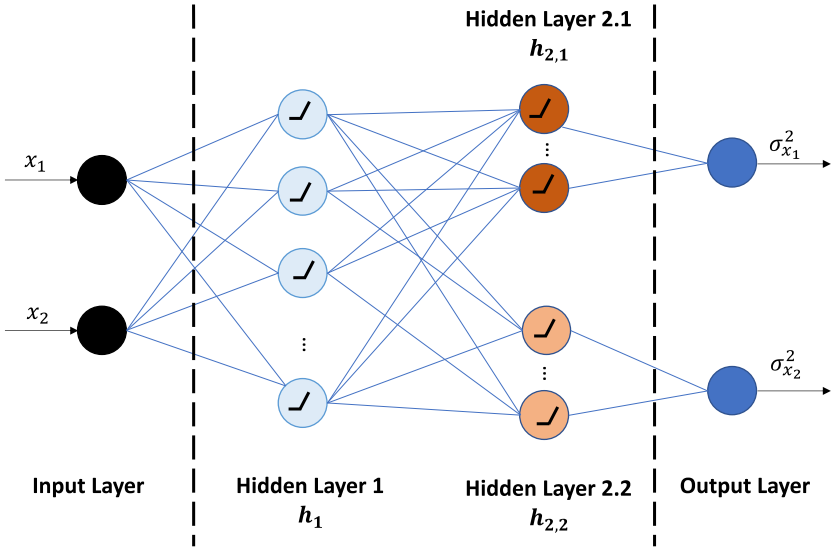


Figure 12. Architecture of Dual-Head Neural Network

c. Training the Neural Network

Training an NN is an iterative process and the NN aims to minimize the loss function by updating its parameters (weights and biases). Each iteration of the training procedure consists of a full pass through the training set whereby the training set is randomly sampled to produce batches of training data and the derivative of the loss function is backpropagated through the network to update the parameters accordingly. After each complete pass of the training set, the validation set is used to assess the performance of the DHNN. When the DHNN validation error stops improving, training is terminated.

We use the adaptive moment estimation (Adam) optimizer algorithm—"an algorithm for first-order gradient-based optimization of stochastic objective functions" (Kingma & Ba, 2017)—to optimize the weights and biases of the neural network. In addition, we bound the error derivate that is being backpropagated to update the parameters of the network (commonly known as gradient clipping), stabilize the weight updates across iteration using weight decay factor (L2 Regularization), and conduct learning rate annealing to prevent the optimization procedure from being stuck in local minima and saddle points and converging towards the global optima point. Table 8 summarizes the parameter values.

Table 8. Optimization Algorithm Parameters

Parameter	**Values**	**Remarks**
Optimization Algorithm		
Optimization algorithm	Adaptive Moment Estimation (Adam)	
Gradient clipping	1.0	Maximum value for the norm of gradients
Weight decay	0.0001	
Decay rate of gradient moving average (β_1)	0.9	Default value in PyTorch
Decay rate of squared gradient moving average (β_2)	0.999	

Parameter	Values	Remarks
Initial learning rate	(Refer to Table 9)	
Learning Rate Scheduler		
Scheduler type	Stepwise decay	
Step size	(Refer to Table 9)	Period of learning rate decay
Gamma	(Refer to Table 9)	Multiplicative factor of learning rate decay
Training Procedure		
Maximum Number of Iterations (Epoch)	500	
Batch size	(Refer to Table 9)	The size of the subset of the training dataset used to evaluate gradient of the loss function and update NN weights
Validation patience	20	The number of times the validation loss can be larger than or equal to the previously smallest loss before terminating training

The training of the DHNN is accomplished using *PyTorch: An Imperative Style, High-Performance Deep Learning Library* written in the Python Programming Language. The pre-processing of data, training, and tuning of the DHNN was carried out on an HP Workstation running 24 x64-based Intel Core i9-7920X CPUs and an NVIDIA GeForce GTX 1080 Ti GPU.

d. Model Tuning

The goal of model tuning is to discover the set of hyperparameters that yield the best performance by the DHNN on the validation set. Table 9 presents the hyperparameters search space.

Table 9. Hyperparameter Search Space

Hyperparameter Category	Hyperparameter	Possible Values
NN architecture	Number of neurons in h_1	8, 16
	Number of neurons in h_2	2, 4
Optimization algorithm	Initial learning rate	0.1, 0.2, 0.3, 0.5
	Learning rate step size	10, 15, 20
	Learning rate gamma	0.1, 0.5, 0.8, 0.9
	Batch size	16, 32, 64

In total, there are 576 potential combinations of hyperparameters. To create models and sample the hyperparameter search space, we use Python library Tune (Liaw et al., 2018). Tune integrates with PyTorch by providing a wrapper function around the training and validation procedures and executes the hyperparameter tuning in parallel. In total, we sampled approximately 69% of the hyperparameter search space using a maximum sample of 500 to derive the best performing hyperparameters for DHNN model for each sensor. The test set is then used to approximate the generalization error of the best model.

The selected hyperparameters and performance of each sensor's DHNN ML model are presented in Table 10 and Table 11 respectively.

Table 10. Selected Hyperparameters of DHNN Models

Parameter	Radar	EO	IR	ESM
Number of neurons in h_1	16	16	16	16
Number of neurons in h_2	4	4	4	2
Initial Learning rate	0.1	0.1	0.2	0.1
Learning rate step size	10	10	15	20
Learning rate gamma	0.5	0.1	0.1	0.5
Batch size	64	64	32	32

Table 11. Model Performance and Description

	Radar	**EO**	**IR**	**ESM**
Total number of weights	152	424	424	356
Total number of biases	26	42	42	38
Score on validation set	0.1046	0.0798	0.0897	0.0099
Score on test set	0.1061	0.0837	0.0937	0.0071

5. Phase 3: Publish the ML Model

In the final phase of the ML Experiment Framework, the best model derived from Phase 2 is stored in a compatible format for subsequent deployment in the operating system. For our purpose, the Tune library used in Phase 2d (Model Tuning), automatically save a copy of the state of the model after each iteration of the training step and the validation score that the model achieves. Hence, the best DHNN model for each sensor can be retrieved using PyTorch.

D. STATE ESTIMATION BY THE ML-KF MODEL

An ML-KF model consists of the trained sensor's DHNN and a KF. The input and output of the ML-KF algorithm are like the algorithm described in Section A2 of this chapter, with the following modifications:

1. The scalers derived from the training dataset are used to scale the sensor measurement inputs to the ML-KF model.
2. The scaled measurements are fed into the DHNN, which would provide the variance of the longitude and latitude.
3. The variances form the sensor measurement error matrix R of the ML-KF.

Figure 13 is a graphical representation of the variables and process flow used by the ML-KF to estimate the target state. The prediction algorithm, with **bold-face fonts** emphasis on the modifications made to the KF model, is described in the following and highlighted in yellow boxes in Figure 13.

1. Define a simulation iteration and retrieve the corresponding dataset for sensor detection and unit position.
2. **Load the DHNN models for each sensor and the respective sensor scalers.**
3. Initialize the KF with measurement function, state transition matrix, and process covariance matrix.
4. Initialize KF's target initial position with the first sensor reading.
5. For each timestep k subsequently:
    - Get a prediction of the target's state estimate from KF and save the estimate.
    - Retrieve sensor detection at that timestep k, and extract the longitude and latitude of the sensor detection: $z_{i,k} = \{x_i, y_i\}_k$.
    - **Update the KF measurement error matrix to that of sensor r_i by using the DHNN to predict the longitude and latitude variance.**
    - Update KF with a measurement$z_{i,k}$ from sensor i.
    - Repeat until there is no additional sensor detection at timestep k.
    - Advance to the next time step.

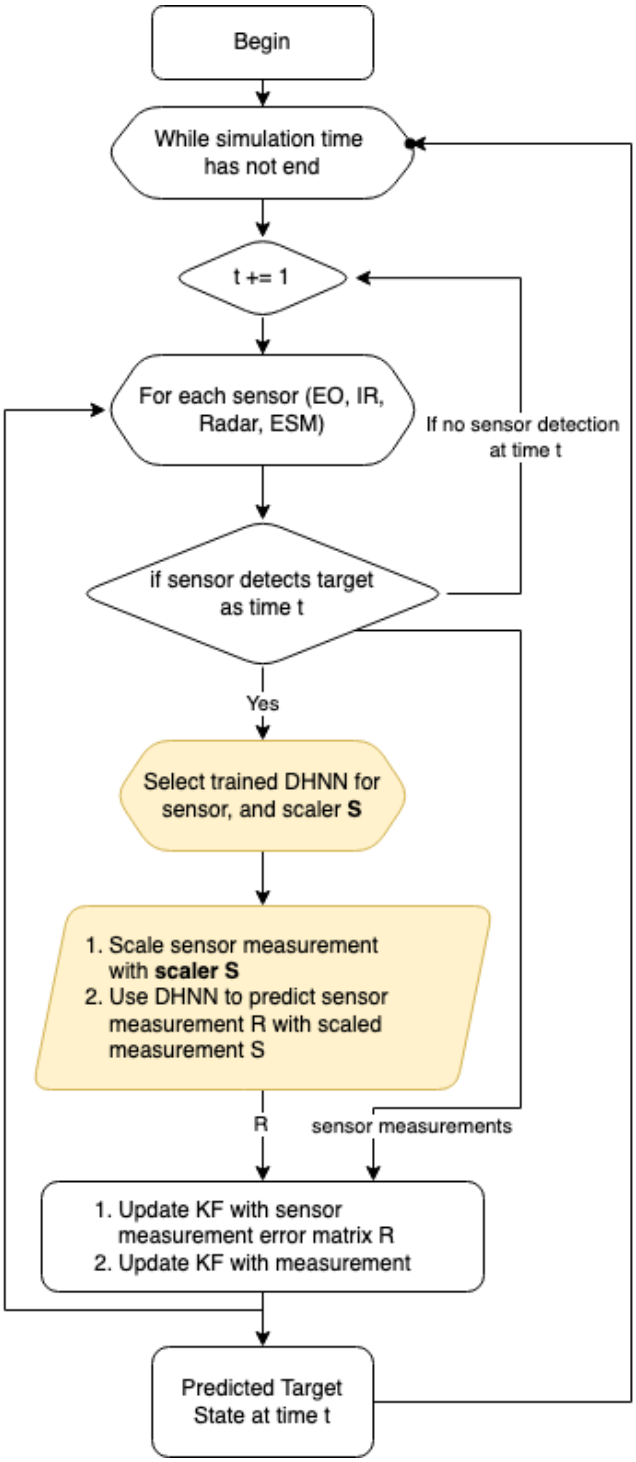


Figure 13. Process Flow for Predicting Target State by ML-KF Model

THIS PAGE INTENTIONALLY LEFT BLANK

V. ANALYSIS OF RESULTS

In this chapter, we present and analyze the results obtained from each model generated. We shed light on the performance of each model in predicting the state of a target using sensor measurements from a wide range of sensors.

We evaluate each model in two ways: 1) using the average RMSE in estimating the target's state across all simulation runs, and 2) evaluating the model's residual error during the prediction of a single run. The average RMSE in estimating the target's state across all simulation runs is useful as a broad and holistic performance metric because each simulation run has subtle differences. However, taking the RMSE of the entire simulation run does not provide us with a means to quantify the performance of each model during specific instances in each scenario, such as when the target is changing its course rapidly as opposed to while traveling on a constant course. This motivates the investigation of the models' performance at different phases of the scenario.

A. OVERALL PERFORMANCE

Table 12 presents the average estimation error for both models across the 100 simulation runs from each environment settings. Recall that the Normal weather environment settings represents an ideal condition in the simulated battlespace while the Extreme weather environment settings simulates a battlespace with poor weather conditions that cause the sensors to perform poorer than the ideal condition. The motivation to compare the two different weather settings is to highlight the improvement in performance even in degraded weather condition.

Overall, the ML-KF model outperforms the KF model and achieves a lower estimation error across the 100 simulation runs for both environment settings (results are highlighted in green in Table 12). Under normal weather conditions, the KF model has an average RMSE of 9.324 ± 0.073 meters (30.59 ± 0.24 feet). In comparison, the ML-KF model has an average RMSE of 7.462 ± 0.043 meters (24.48 ± 0.014 feet),

significantly outperforming the KF model in estimating the target's state by approximately 20% using a 5% significance level[5].

Table 12. Performance of Models across 100 Simulation Runs

Environment	Model		RMSE		
			Longitude	Latitude	Overall
Normal	KF	mean	7.541179	5.478568	9.323550
		std	0.181759	0.131409	0.073194
	ML-KF	mean	5.725176	4.783331	7.462227
		std	0.129570	0.111124	0.043657
Extreme	KF	mean	7.559086	5.467258	9.331756
		std	0.195241	0.140200	0.079345
	ML-KF	mean	5.733017	4.773831	7.462399
		std	0.135887	0.118889	0.043015

1. Comparison of Performance between Weather Datasets

The performance of the KF model is poorer in extreme weather conditions with a result of 9.332 ± 0.079 meters. Once again, the ML-KF model outperforms the KF model with a result of 7.4623 ± 0.043 meters (green background in Table 12). When comparing the performance of the model in the extreme weather dataset against the normal weather dataset, we observe that both models did not perform significantly better or worse at a 5% confidence level.

[5] We conducted a hypothesis testing to determine if the ML-KF model outperforms the KF model. The null hypothesis is that there is no difference in the average RMSE across 100 simulation runs, while the alternate hypothesis is that the ML-KF model have a lower average RMSE. We used a significance level at 5%. The paired t-test across the simulation runs yield a p-value of 3.16×10^{-23}, which is less than the significance level of 0.05. Thus, we strongly reject the null hypothesis that the performance of ML-KF and KF model is similar.

We would have expected the performance of the ML-KF model to perform worse than in the normal weather settings because the ML-KF DHNN models were not trained on the extreme weather dataset and the weather dataset was expected to contain noise that is not like that observed in the normal weather dataset. However, despite the strong performance of the model in extreme weather settings, this comparison is not a strong testament to the performance of the ML-KF model. This is because, as noted in the limitations of the dataset in Chapter III, the extreme dataset only has slight variability in sensor errors when compared to the normal weather dataset. Thus, we are unable to conclude that the ML-KF model outperforms the KF model when sensor measurements are corrupted due to external factors (such as weather) and if the DHNN embedded in the ML-KF model was effective in adapting the measurement noise matrix when sensor measurements are noisy.

2. Performance in Prediction of Longitude and latitude

The distribution of errors across the 100 simulation runs of each environment set is presented in Figure 14 and Figure 15 for the KF and ML-KF models respectively. Evaluating the performance for each physical dimension (longitude and latitude) independently, we observe the following:

a. The error in longitude was much greater than that in the latitude, contributing the most to the overall RMSE. A potential reason is that the range of possible values for longitude is larger than that of latitude ($\sigma_{latitude} = 2876.88 > \sigma_{longitude} = 2680.81$), resulting in a larger margin for error in state estimation.

b. The distribution of error in latitude is a heavy-right tail, while the distribution of longitude is a heavy-left tail, like the distribution of the overall RMSE. This suggests that the overall RMSE is strongly influenced by the errors in the longitude since the range of RMSE(longitude) is larger than that of RMSE(latitude), so the overall RMSE would be skewed by RMSE(longitude).

c. Comparing the distribution of RMSEs between the KF and ML-KF models, no significant differences were observed; both models have RMSEs that are heavy-tailed as described previously. A notable difference is the range of RMSE values, suggesting that the RMSE for the ML-KF model is smaller (lower mean than the KF model) and the spread of the RMSE values is smaller (lower standard deviation than KF Model) (Table 12).

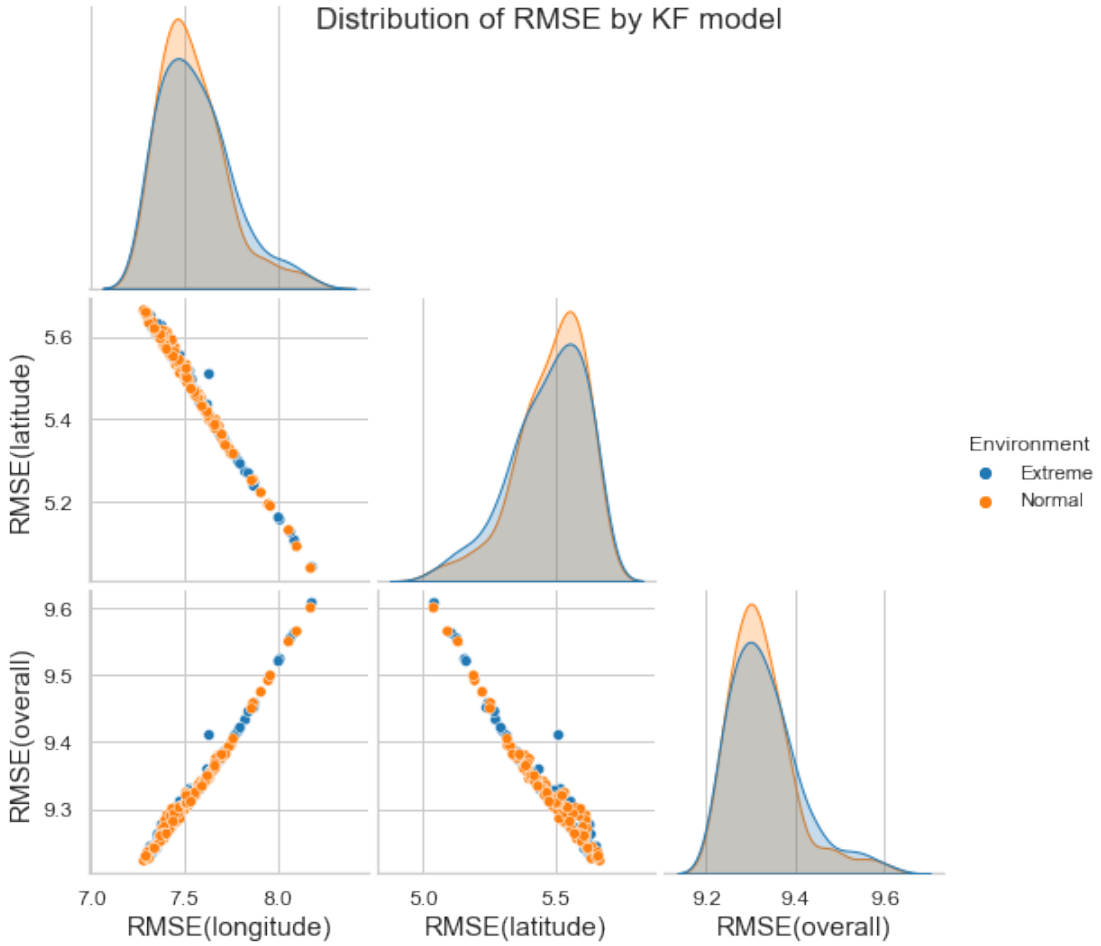


Figure 14. KF Model RMSE Distribution

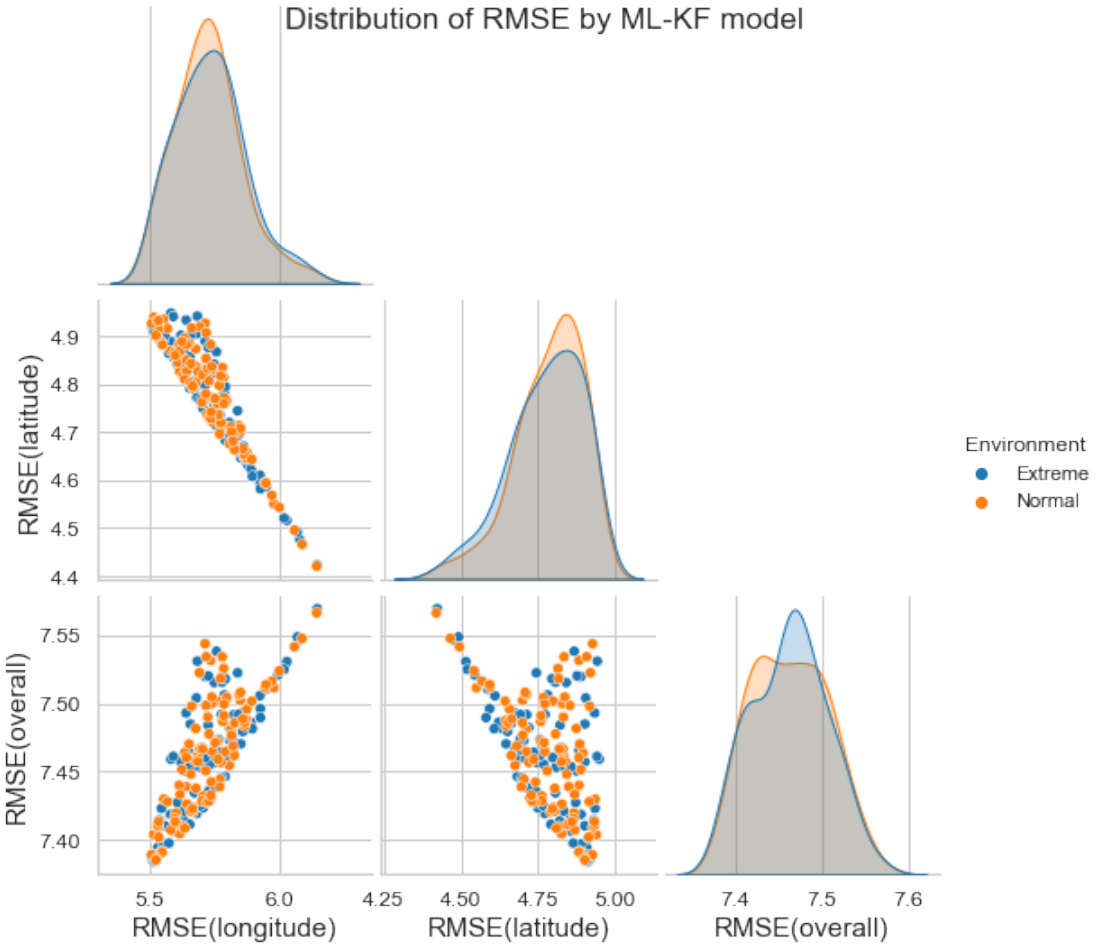


Figure 15. ML-KF Model RMSE Distribution

B. PREDICTION ERROR DURING KEY PHASES OF TARGET MOVEMENT

Investigating the performance of our models during the simulation run offers additional insights, we randomly selected a simulation run–simulation run number 5–from the normal dataset and visualized the RMSE for the prediction of the target's longitude and latitude. Figure 16 illustrates the model's RMSE for longitude and latitude respectively. We use a simple moving average (SMA) function to smooth the RMSE calculated.[6] The SMA(100) plot for each model suggests that the RMSE changes drastically whenever the target changes its heading. The target is said to be changing its

[6]The SMA function averages the RMSE from the previous 100 timesteps in the simulation.

heading when the gradient of the longitude and latitude plots changes, highlighted by the green vertical bands in Figure 16. The target's heading changes whenever the target has arrived at an RP (reference point) in the simulation and is heading towards another RP.

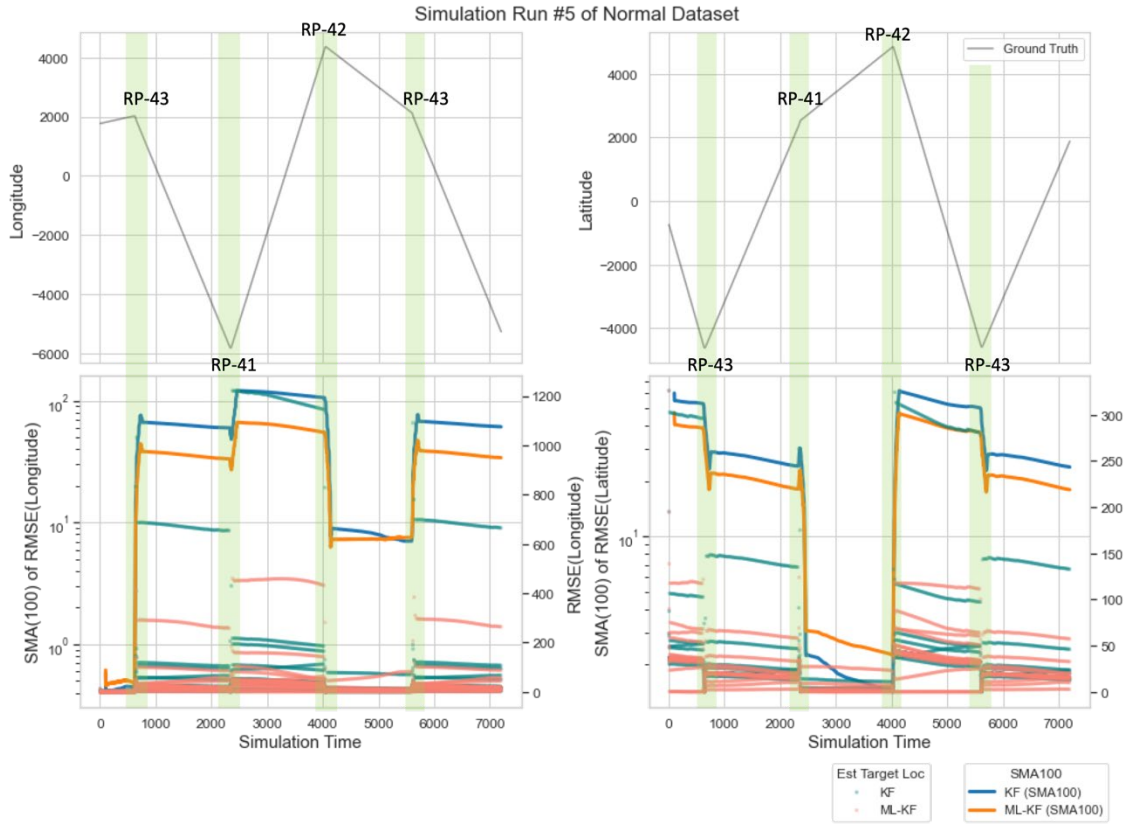


Figure 16. Model's State Estimation Error during a Single Simulation Run

Now, we investigate the model's error when moving between reference points (target has a constant heading) and at a reference point (target is changing its course).

1. Constant Heading

Figure 17 illustrates the RMSEs when the target is moving between RP-43 and RP-41 for simulation run 5 in normal weather conditions. During this movement, the target is expected to traverse at a constant speed (hence constant heading). The following observation and analysis can be made regarding the RMSE values:

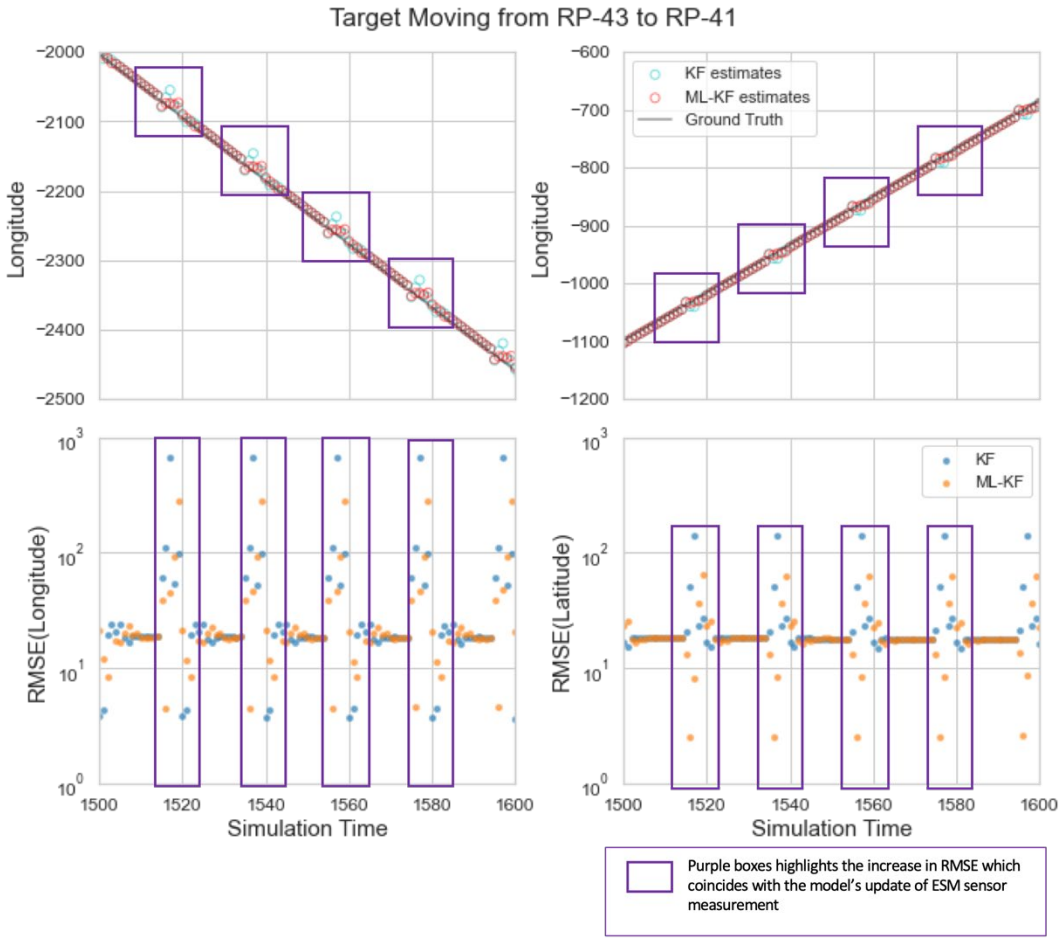


Figure 17. Target Moving between RP-43 to RP-41 (Constant Heading)

The sensor measurements arrive at the KF whenever a detection was made; the pattern in RMSE correlates with the measurement arrival period of each sensor, thereby suggesting that the sensor measurements may disrupt the KF estimation when the sensor measurement matrix does not accurately reflect the uncertainty in sensor measurement. For example, we note that the KF model RMSE increases drastically every 20 timesteps of the simulation, corresponding to the arrival of ESM sensor detection. The purple boxes in Figure 17 highlights this insight.

One reason for the sharp changes in prediction may be due to the low uncertainty in ESM sensor measurements,[7] causing the KF to "increase" its probabilistic belief in the ESM sensor measurements. Then, in subsequent timesteps when the other sensor measurements are different from the KF's prediction, the KF over (or under) compensates for the difference in its prediction and the sensor's measurement, thereby resulting in a sudden jerk in the state estimated by the KF.

Since the ML-KF model uses the KF algorithm for prediction, similar jerks in prediction error are observed as well. In this vein of analysis, the ML-KF model was able to reduce the error (the ML-KF model reports a lower RMSE when the jerks occur) by improving the sensor measurement error to be used by the KF, but not reducing the poor estimation by the KF algorithm.

2. Changing Heading

Figures 18 and 19 illustrate the models' estimates against the ground truth and RMSE error in measurement of longitude and latitude when the target is approaching an RP and subsequently moving away from it towards another RP. In general, both figures show us that 1) the models' estimates at the turning point fluctuate, and 2) the KF estimate is often further away from that of the ML-KF estimate compared to the ground truth. These observations are expected because the KF algorithm assumes a constant acceleration state process model, which would not perform well when a target changes its acceleration, such as a change in the target's heading. In other words, the prediction by the models that uses a constant acceleration KF model is unable to generalize to the case with a target varying its acceleration.

Ideally, the ML-KF model should be able to reduce estimation errors when the target's heading changes, by dynamically adjusting the probabilistic belief of the sensor measurement. A well-performing ML-KF model would result in a consistently low RMSE before and after the target has changed its heading. However, observing that the

[7] The ESM sensor had the lowest sensor detection error compared to the EO, IR and Radar sensors (Table 4, Chapter III).

trend in RMSE by the ML-KF model is strongly correlated to that of the KF model, it was inconclusive that the ML-KF model was adjusting the prediction dynamically when the target is changing its course.

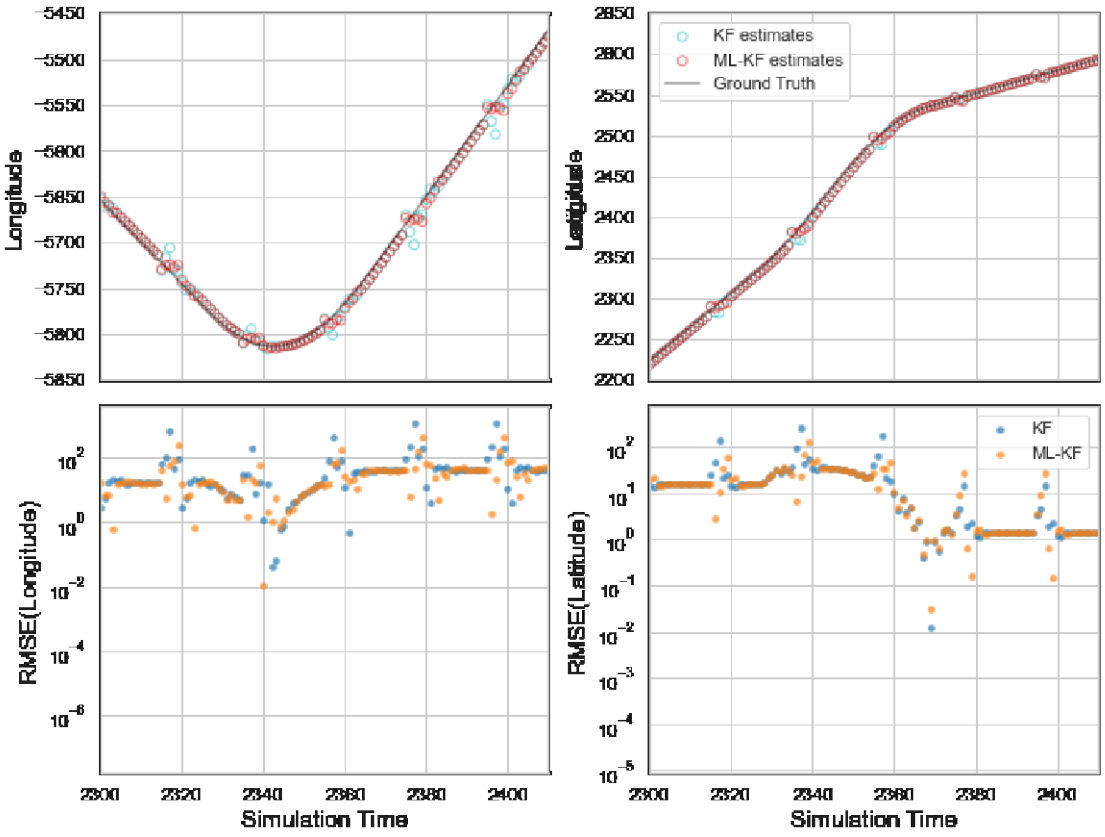


Figure 18. Target Changing its Course around RP-43

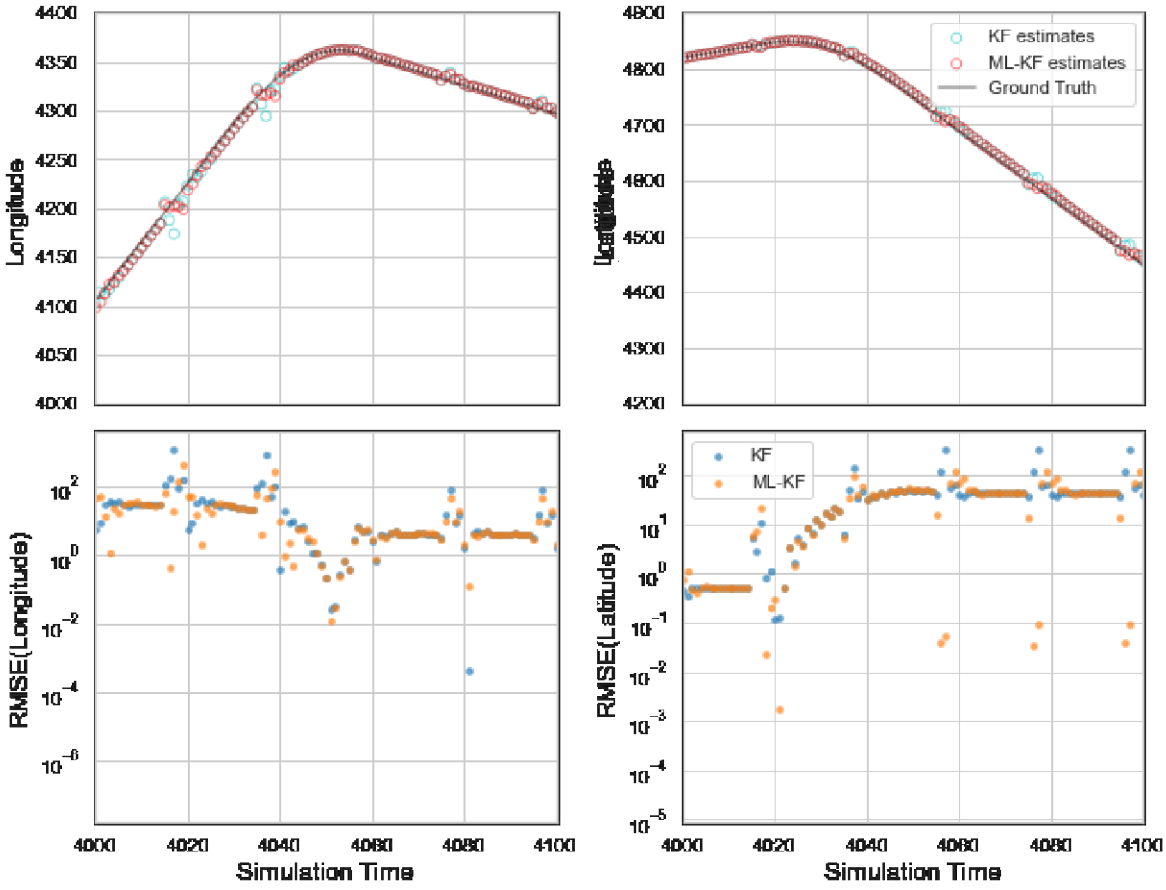


Figure 19. Target Changing its Course around RP-42

From our simulation runs and performance evaluation, we observe that the KF algorithm performance is dependent on the sensor's measurement and can be abruptly changed by the measurements. This is exacerbated by the fact that the sensor's measurement noise has a smaller confidence interval, resulting in varying changes in state estimation, instead of a consistent improvement in state estimation across all time steps. The ML-KF model improves the estimation errors by providing the KF algorithm with improved estimation of the uncertainty of the sensor measurements. However, both the KF and ML-KF models were unable to dynamically adjust when the target changed its heading as the prediction is still strongly predicated on the assumptions embedded in the KF itself.

VI. CONCLUSIONS

A. SUMMARY OF RESEARCH

We set out to investigate whether ML models are able to improve the accuracy of state estimation in a COP/CTP. We adopted a quantitative approach and our findings have shown that integration of ML models to estimate the sensor measurement error matrix for the standard KF algorithm can significantly improve the accuracy of target state estimation by approximately 20%. In summary, our contributions are:

1. We have developed an ML operations pipeline that ingests data from a simulation to train, validate, and test machine learning modules for subsequent deployment in a KF system. The methodology, dataset, and models generated are reproducible and replicable, as the code base and frameworks used for this development are fully open source.
2. We have demonstrated that a learning module embedded in a standard KF algorithm can improve state estimation over a standard KF model. The ML-KF model was able to generate a sensor measurement error matrix to update the KF algorithm's probabilistic belief of the sensor measurements, thereby improving the KF's estimation.
3. We were only able to train the learning module used in the KF model only because our simulation system provides a ground truth target state that live ranges may not be able to provide. This demonstrates the potential of using simulation to develop ML models and of subsequently deploying them in the field.

The following sections present the limitations of our research, challenges faced when developing and executing the ML operations pipeline, and areas for future work.

B. LIMITATIONS AND FUTURE WORK

1. Simulated Dataset

a. Sensor Measurement Uncertainty

As highlighted in Chapter III, we are limited by the dataset generated using the CMO simulation software to generate sensor measurements. As a commercial product, it did not allow the researcher to obtain detailed information about the models used in the simulation and the software could not be modified directly to pursue particular research goals. Critically, the absence of sensor measurement uncertainty output for each of the simulated sensors resulted in the use average error in sensor measurement, against the ground truth target position, as the sensor measurement error matrix in our baseline model. In addition, we found that some of the sensors have measurements that do not meet our realistic expectations for the sensor. Specifically, the ESM sensor measurements had very high accuracy (small error). This may not be the case, since the ESM sensor provides an area of uncertainty where the target may be located, instead of a spot measurement.

For future research, it may be better to employ a more open simulation environment, such as the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) (West & Birkmire, 2019). Modeling processes of interest in AFSIM may require more detailed design and implementation effort, accompanied by necessary verification and validation procedures, which can be more demanding than directly employing a simulation "as-is." However, there are numerous organizations using AFSIM, so there is greater opportunity for scenario and software reuse than in other environments. Of note, there are distribution restrictions on AFSIM that prevent its use in an academic environment by foreign students and make studies using AFSIM less available to the research community at-large. Even so, in an environment such as the Naval Postgraduate School and recognizing the inherent military sensitivity of such topics as data fusion, investigation into the use of AFSIM to support future research is warranted.

b. Inclusion of External Factors in ML models

We started the work intending to discover if variance in external factors affecting sensor measurements would be mitigated by ML models, as described in Ullah et al., (2019 and 2020). Our desire, however, was not satisfied, as the output from the simulation does not provide a dataset suggestive of large variation when the weather conditions have changed. We propose for future researchers to include noise in the sensor dataset to investigate ML-KF models. The additive noise can be modeled as a parameter of water conditions, such as higher noise in sensor measurement corresponding to the degradation of sensor measurements in the presence of heavy rainfall.

2. Model Limitations

We now present some of the limitations of our models. While the ML-KF model has improved performance over the standard KF model, we think that the following areas must be studied further to provide a more conclusive evaluation of the embedding the ML model in a KF to predict parameters for the KF:

a. How Well Does the AI/ML Model Generalize?

This thesis has provided a proof-of-concept on using ML as part of the data filtering process in state estimation but is limited in showing that the ML models can generalize well to different use cases or scenarios. Since the ML models are trained on a fixed set of datasets, the models are not expected to perform well when the simulation settings changes. For example, if the simulation longitude and latitude coordinates were to be set in a different geographic area, and the sensors were set up differently, the AI/ ML models trained in one setting may not be useful in another. Another case where the models may not adapt well is when there are external factors influencing the sensor measurements, such as weather conditions.

While we have attempted to generate a set of datasets based on the different weather conditions, there was not much notable difference in the estimation error across all sensors, and hence, we decided to limit our training dataset to the normal weather

dataset and restrict the number of inputs to the DHNN to only the sensor measurements. Following, we propose different ways to improve the generalizability of the model.

(1) Randomized Reference Points in Simulation

In our dataset, the target moves in a fixed set of pre-defined reference points. While we have randomized the initial target position for each of the simulation run, there exists significant correlations exist between each run, such as the position where a target would be turning. A simulation set up and sensor dataset to train the ML models would enhance the models' adaptiveness for different types of maneuvers.

(2) Inclusion of External Factors for the ML Model Input

Should we have a dataset that is significantly different from nominal conditions, we could include these inputs to the ML model so that the KF parameter to be predicted would dynamically adapt, thereby increasing the accuracy in managing a wide range of potential scenarios.

Another line of effort would be to embed trained ML model(s) within other variants of KF models, such as Interacting Multiple Models KF and Extended KF. In our KF algorithm, we made a bold assumption that the target will be moving in constant acceleration. This assumption was useful for us to assess if the ML-KF model were to be able to improve the performance during a change in acceleration, since a standard KF would perform poorly on it. Since we do not see an improvement in state prediction when the target is accelerating or decelerating, we think that it would be useful to conduct an ablation study, to assess the key contributing factors or changes that brought to bear the improvement in state estimation by the KF (and its variants) or the learning module embed in an ML-KF model.

b. Formulating a Time-series Problem

In our approach, we have assumed that the samples in the dataset are independent and identically distributed. This means each sensor detection provides nothing new about the history of detections or subsequent detections made by the sensor. We assumed *time* to be an independent random variable in our dataset, as it simplifies the ML problem

formulation. We thus encourage future research to consider the case where the sensor measurements are temporally dependent. To that end, we put forth two suggestions that are potentially mutually reinforcing:

(1) Using LSTM modules in NN

Jung et al. (2020) proposed a Long Short-Term Memory (LSTM) KF that integrates LSTM modules to output a target state estimation. A recurrent NN with an LSTM module differs from the standard feedforward NN in that LSTM has feedback connections designed to process sequences of data and only important information from the sequence is kept to aid the processing of subsequent data points (Graves, 2012). Hence, using a recurrent NN with LSTM modules in a KF could improve the confidence of the model in predicting target state estimates and alleviate the Markov property of the KF.

(2) Reinforcement Learning (RL)

Gao et al. (2020) showed that ML models learned using reinforcement learning were effective in predicting process noise covariance matrix for the KF. The ML model was trained using the deep deterministic policy gradient algorithm with the target location error as the penalty to the AI agent. Hence, a potential area for exploring the use of AI/ ML in improving data filtering would be to use reinforcement learning agents as the ML model. This is useful because the reinforcement learning agent will be able to search the action space (the possible outputs of the noise covariance matrices) by interacting with the simulation engine directly. This would be especially beneficial for an ML model where the AI-controlled targets in the simulation moves randomly based on different mission sets. Hence, by learning the target's movement online, there is no need to generate and store datasets for the ML phase, and the agent would be generalized.

These two suggestions are potentially mutually reinforcing. This is evidenced by the successes by Google's DeepMind in creating an AI to achieve Grandmaster status in StarCraft (Vinyals et al., 2019), which uses deep LSTM NN and an RL training regime to create the AI. The AI was able to improve its score through multiple hours of game plays in the simulated environment to improve its decision-making abilities.

c. Multiple Target Data Filtering

In our scenario, only one target is used for this proof-of-concept. It would be of interest to our sponsor and the larger IW community to be able to filter multiple targets. In this requirement, the ML problem would therefore consist of two tasks: classification and regression. While the regression task–to predict sensor measurement errors–remains unchanged, the additional classification task is to classify the target's importance to the analysts–such as if the target is a friend or foe. A more sophisticated multi-class classification problem would be to classify the different types of targets (e.g., different classes of naval ships). In addition, we have not field tested the improved system, and would encourage future researchers to implement similar methods to assess the efficacy, latency, and system overhead incurred to provide performance measures of such a system.

C. CONCLUSION

Our research used ML models to predict sensor measurement errors for a standard KF algorithm. Our ML-KF model was able to significantly outperform our baseline model at 5% confidence level, showing that using an ML-KF model would improve the performance of target position state estimations, alleviating the performance issue when uncertainty of sensor measurement is absent from heterogenous sensor data streams. In other words, in the absence of uncertainty measurements of sensor data, the ML embedded in the KF was able to predict the uncertainty and dynamically updating the parameters of the KF algorithm.

This proof-of-concept has the potential to be further extended using more sophisticated methods. We have proposed three key areas for future research:

1. To improve the generalizability of the ML-KF model by including other parameters (such as weather conditions in the battlespace) that are not directly modeled or used in KF.
2. Using time-series methods to model temporal movement of a target, thereby increasing the predictive power of the learning module embedded in the KF.

3. Using ML models to conduct multiple target data filtering (JDL Level 2), by including a classification task to categorize the track data.

As the DoD increasingly shifts its focus to the application of ML, we believe that such an application in data filtering would be able to augment existing data filtering methods and eliminate the expense of replacing them. For instance, by enhancing existing COP/CTP data filtering algorithms, we would be able to have a more accurate state estimation of the target, thereby providing a higher confidence of the target's position in the COP/CTP. The ability for such an ML-KF model to ingest heterogenous data stream is also a powerful tool to automate the work of intelligence analysts who would frequently need to cross-reference their sources across different intelligence domains. By improving the suite of tools available to our warfighters, they will be more lethal in their response to any adversary.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SCRIPTS FOR CMO SIMULATION

A. LUA SCRIPTING FOR RANDOMLY GENERATING A TARGET'S POSITION

Lua is a programming language used by CMO to provide advanced users with the ability to "implement virtually any desired behavior" (Matrix Games, 2022a). For instance, players can use the Lua command to extend existing AI behaviors and spawn units within the game based on the status of the scenario. For this thesis, we endeavor to have our target to be randomly generated at a random location within the designated patrol area defined by the reference points RP-41, RP-42, and RP-43. Since the three points describe a triangle, vector arithmetic is used to randomly generate a point within the triangle to initialize the target's position. The Lua script used is reproduced below:

```
local a = ScenEdit_GetReferencePoints({
    side = "Target",
    area = {"RP-43", "RP-41", "RP-42"}
})
local vec_a_lat = a[2]["latitude"] - a[1]["latitude"]
local vec_a_lon = a[2]["longitude"] - a[1]["longitude"]

local vec_b_lat = a[3]["latitude"] - a[1]["latitude"]
local vec_b_lon = a[3]["longitude"] - a[1]["longitude"]

local u1 = math.random()
local u2 = math.random()
if u1 + u2 >= 1 then
    u1 = 1 - u1
    u2 = 1 - u2
end
local w_lat = u1 * vec_a_lat + u2 * vec_b_lat + a[1]["latitude"]
local w_lon = u2 * vec_a_lon + u2 * vec_b_lon + a[1]["longitude"]

-- print(w_lat)
-- print(w_lon)

local target_unit = {
    name = 'DDG 72 Mahan [Arleigh Burke Flight II]',
    guid = 'KGQ0E2-0HMIBR4PMU7G2'
}
target_unit['lat'] = w_lat
target_unit['lon'] = w_lon
ScenEdit_SetUnit(target_unit)
```

B. POWERSHELL SCRIPT TO RUN CMO FROM COMMAND LINE INTERFACE

Only available in the premium edition of CMO, a command-line interface (CLI) is available for analysts to run their simulation without the graphical user interface (GUI) hence reducing the graphics processing overhead and allowing the simulation to run faster and more efficiently. Since we must run each scenario 100 times, we leverage the CLI CMO provides to automate the process. The PowerShell script used to run CMO from the CLI is reproduced below.

```
1. $currdir = Get-Location
2. $scenFile = "`"C:\ProgramData\Command Professional Edition
   2\Scenarios\FixedSensors.scen`""
3. for ($i = 0; $i -lt 100; $i++) {
4.     $savedir = Join-Path -Path ${currdir} -ChildPath $i
5.     $savedir = "`"$savedir`""
6.     Write-Host "run iteation $i : writing into $savedir"
7.     Start-Process -Filepath "C:\Program Files (x86)\Command Professional Edition
   2\CommandCLI.exe" -argumentlist "-mode mc -scenfile ${scenFile} -outputfolder
   ${savedir} -autoexit" -Wait
8. }
```

Upon completion of the PowerShell command, there is a folder with 100 sub-folders, each consisting of the dataset generated for its respective scenario.

APPENDIX B. DATA DICTIONARY

A. UNIT POSITION TABLE

Field	Description	Data Type	Sample Value/ Remarks
TimelineID	The unique ID of the simulation run under which the event occurred	String	5c945cd6-18ba-4bf4-9622-bc26d6c33932
Time	The scenario time at which the event occurred	String	The string is formatted as hh:mm:ss
UnitID	The unique ID of the unit	String	KGQ0E2-0HMIBR4PMU7G2
UnitDBID	The database ID of the unit	Integer	2868
UnitName	The actual name of the unit	String	DDG 72 Mahan [Arleigh Burke Flight II]
UnitType	The type of unit	String	Ship
UnitClass	The class of unit	String	DDG 72 Mahan [Arleigh Burke Flight II]
UnitSide	The name of the side to which the unit belongs	String	Target or Sensors
UnitLongitude	The longitude of the unit	Float	-119.0180886
UnitLatitude	The latitude of the unit	Float	33.69657902
UnitCourse	The heading of the unit in degrees	Float	60.61721
UnitSpeed_kts	The speed of the unit in knots	Float	12
UnitAltitude_m	The altitude of the unit in meters	Float	*Not applicable for this project*
UnitAttitude_Pitch	The pitch of the unit	Float	*Not applicable for this project*
UnitAttitude_Roll	The roll of the unit	Float	*Not applicable for this project*
Status	The status of the unit	String	*Not applicable for this project*
Condition_AirOps	The condition of air operations	String	*Not applicable for this project*
Condition_DockingOps	The condition of docking operations	String	*Not applicable for this project*
AssignedMission	The name of the mission	String	Mission: Patrol

	assigned to the unit		
DamagePercent	The percentage of the unit that is damaged	Integer	*Not applicable for this project*
Fire	The unit is on fire	Boolean	*Not applicable for this project*
Flood	The unit is flooded	Boolean	*Not applicable for this project*
ComponentStatus	The status of the component on the unit	String	*Not applicable for this project*

B. SENSOR DETECTION ATTEMPT TABLE

Field	**Description**	**Data Type**	**Sample Value/ Remarks**
TimelineID	The unique ID of the simulation run under which the event occurred	String	5c945cd6-18ba-4bf4-9622-bc26d6c33932
Time	The scenario time at which the event occurred	String	00:00:10
SensorID	The unique ID of the sensor	Integer	KGQ0E2-0HMJ3OD15T23H
SensorName	The name of the sensor's parent	String	AN/ SLQ-32(V)2 [ESM]
SensorParentID	The unique ID of the sensor's parent	Integer	KGQ0E2-0HMJ3OD15T230
SensorParentName	The name of the sensor's parent	String	esm
SensorParentLongitude	The sensor's parent longitude	Float	-119.06086
SensorParentLatitude	The sensor's parent latitude	Float	33.7624658
SensorParentAltitude_ASL	The sensor's parent altitude above sea level in m	Float	*Not applicable for this project*
SensorParentAltitude_AGL	The sensor's parent altitude above ground level in m	Float	*Not applicable for this project*
SensorParentSide	The side of the sensor's parent	String	Sensor(ESM)
DetectionMode	The mode of detection	String	Search
TargetID	The target's unique ID	String	KGQ0E2-0HMIBR4PMU7G2
TargetName	The actual name of the target	String	DDG 72 Mahan [Arleigh Burke Flight II]
TargetSide	The side of the target	String	Target
TargetLongitude	The estimated longitude of the	Float	-119.01751

	target		
TargetLatitude	The estimated latitude of the target	Float	33.6967081
TargetAltitude_ASL_m	The estimated target altitude above sea level in m	Float	*Not applicable for this project*
TargetAltitude_AGL_m	The estimated target altitude above ground level in m	Float	*Not applicable for this project*
TargetRangeHoriz_nm	The horizon range from the sensor to the target in nm	Float	4.49954
TargetRangeSlant_nm	The slant range from the sensor to the target in nm	Float	4.49954
DetectionResult	The outcome of the detection	String	SUCCESS
DetectionAOU	The area of the uncertainty of the detection is defined by six sets of longitude and latitude coordinates	Array of floats	{Lon:-119.06086272032 - Lat:33.7624658026677}{Lon:-119.06086272032 - Lat:33.7624658026677}{Lon:-119.06086272032 - Lat:33.7624658026677}{Lon:-118.978922492301 - Lat:33.5904293782114}{Lon:-118.960190968491 - Lat:33.5974559553062}{Lon:-118.942315320068 - Lat:33.6058911098875}

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SOFTWARE PACKAGES USED

This appendix lists the software packages used in this thesis.

Name of Software	Version	Purpose
Simulation Software		
Command: Modern Operations	Professional Edition V2.0	Simulation software that creates a scenario to generate the sensor detection dataset for our subsequent use
Data Analysis		
Pandas Python Library	1.4.2	Software that enables the manipulation of data, such as converting the longitude and latitude into ENU format
Matplotlib	3.5.1	Data visualization
Seaborn	0.11.2	Data visualization
PyMap3D	2.9.1	Package to convert geodesic coordinates to ENU format.
Kalman Filter		
FilterPy	1.4.5	Python library that implements Kalman filters
Machine Learning		
PyTorch	1.11.0	For training neural network
Scikit-Learn	1.0.2	For the creation of and datasets partitioning
Tune (by Ray)	1.13.0	For the efficiently conduct of hyperparameter search

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. PYTHON NOTEBOOK–EXPLORATORY DATA ANALYSIS

```
import os
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import time
import itertools
from helper import *

# defines the path for the datasetss
path_normalweather = '../data/normal_weather'
path_xtremeweather = '../data/extreme_weather'

SAVE_DF = False
# Load the DF:
if not SAVE_DF:
  dataset = load_all_dfs()  # dataset[weather][sensor] = get_sensor_df(sensor, weather)
```

A. INSPECTING HEADERS OF DATASET

```
sensor_detect_filename = '../data/extreme_weather/0/Sensor(Radar)_SensorDetectionAttempt.csv'
target_pos_filename = '../data/extreme_weather/0/Target_UnitPositions.csv'
df_pos = pd.read_csv(target_pos_filename, nrows=3)
print(list(df_pos.columns))
```

```
['TimelineID', 'Time', 'UnitID', 'UnitDBID', 'UnitName', 'UnitType', 'UnitClass', 'UnitSide',
'UnitLongitude', 'UnitLatitude', 'UnitCourse', 'UnitSpeed_kts', 'UnitAltitude_m',
'UnitAttitude_Pitch', 'UnitAttitude_Roll', 'Status', 'Condition_AirOps', 'Condition_DockingOps',
'AssignedMission', 'DamagePercent', 'Fire', 'Flood', 'ComponentStatus']
```

```
df_sense = pd.read_csv(sensor_detect_filename, nrows=3)
print(list(df_sense.columns))
```

```
['TimelineID', 'Time', 'SensorID', 'SensorName', 'SensorParentID', 'SensorParentName',
'SensorParentLongitude', 'SensorParentLatitude', 'SensorParentAltitude_ASL',
'SensorParentAltitude_AGL', 'SensorParentSide', 'DetectionMode', 'TargetID', 'TargetName',
'TargetSide', 'TargetLongitude', 'TargetLatitude', 'TargetAltitude_ASL_m', 'TargetAltitude_AGL_m',
'TargetRangeHoriz_nm', 'TargetRangeSlant_nm', 'DetectionResult', 'DetectionAOU']
```

B. COMPARISON OF COORDINATE SYSTEMS (GEODESIC AND ENU REPRESENTATION)

1. We are given the following geodesic coordinates:

- actual target location (ground truth),
- sensor location (sensor parent lat lon alt), and
- estimated target location (target lat lon...)

2. We want to calculate the error between estimated target location and the actual target location.

1. Sample Dataset

```
# Sample a dataset
dataset = load_all_dfs()
df_pos = dataset['normal']['pos']
df_radar = dataset['normal']['Radar']

timeline_sample = df_pos.TimelineID.iloc[0]
df_pos = df_pos[df_pos.TimelineID == timeline_sample]
df_radar = df_radar[df_radar.TimelineID == timeline_sample]
df_merged = pd.merge(df_pos, df_radar, left_on=['TimelineID', 'Time'], right_on=['TimelineID',
'Time'], how='left')
df_merged.dropna(axis=0, inplace=True)
```

a. Conversion to ENU representation

(1) Converts the Lon Lat from database into East-North-Up representation. This requires a reference point to be defined and is abstracted in the get_ENU() helper function. We use a common point (middle of all 3 reference points in the simulation as the reference point for projection). ENU is less accurate due to the projection from the reference point.

(2) Subsequently calculate the cartesian distance between a set of ENU points (i.e., the L2-Norm)

(3) In ENU (xyz), we can consider the height dimension easily, it performs better than geodesic distance in this aspect. The latter only considers surface distance (i.e., a walk along earth's surface)

```
import pymap3d as pm
# from geopy.units import nautical

# Using the get_ENU function.
lon = df_pos.iloc[0]['UnitLongitude']
lat = df_pos.iloc[0]['UnitLatitude']
alt = df_pos.iloc[0]['UnitAltitude_m']
get_ENU(lon, lat, alt)

(-554.4601823446596, 2023.0458401609612, -100.34608059636821)

# Unit
df_merged_pm = pd.concat(
    [df_merged, df_merged.apply(lambda r: get_ENU(r.UnitLongitude, r.UnitLatitude,
r.UnitAltitude_m), axis=1, result_type='expand')], axis=1)
df_merged_pm.rename({0: 'Unit_E', 1: 'Unit_N', 2: 'Unit_U'}, axis=1, inplace=True)
# Target
df_merged_pm = pd.concat(
    [df_merged_pm,
     df_merged_pm.apply(lambda r: get_ENU(r.TargetLongitude, r.TargetLatitude,
```

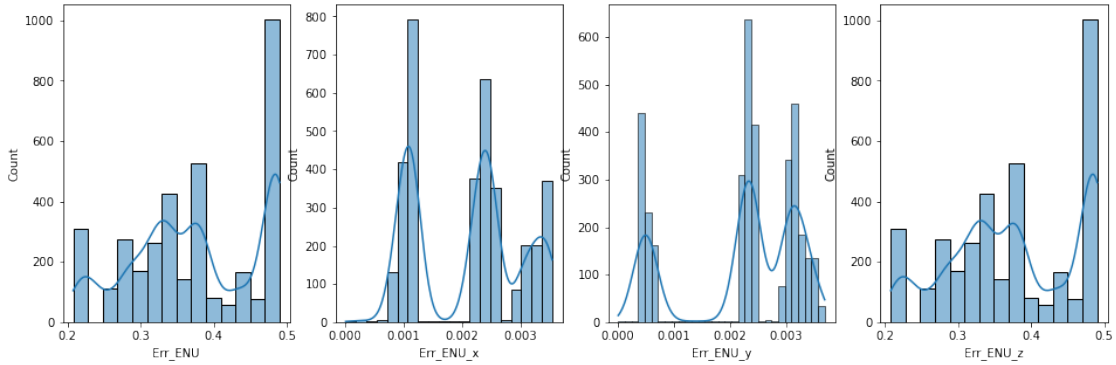
```
r.TargetAltitude_AGL_m), axis=1, result_type='expand')],
    axis=1)
df_merged_pm.rename({0: 'Target_E', 1: 'Target_N', 2: 'Target_U'}, axis=1, inplace=True)
# Sensor
df_merged_pm = pd.concat([
    df_merged_pm,
    df_merged_pm.apply(lambda r: get_ENU(r.SensorParentLongitude, r.SensorParentLatitude,
r.SensorParentAltitude_AGL), axis=1, result_type='expand')
],
                         axis=1)
df_merged_pm.rename({0: 'Sensor_E', 1: 'Sensor_N', 2: 'Sensor_U'}, axis=1, inplace=True)

df_merged['Err_ENU'] = df_merged.apply(lambda r: l2_norm_3d(r.Target_E, r.Target_N, r.Target_U,
r.Unit_E, r.Unit_N, r.Unit_U), axis=1)
df_merged['SlantRange_ENU'] = df_merged.apply(
    lambda r: l2_norm_2d(r.Target_E, r.Target_N, r.Target_U, r.Sensor_E, r.Sensor_N, r.Sensor_U),
axis=1)
df_merged['Err_ENU_x'] = df_merged.apply(lambda r: sq_err_1d(r.Target_E, r.Unit_E), axis=1)
df_merged['Err_ENU_y'] = df_merged.apply(lambda r: sq_err_1d(r.Target_N, r.Unit_N), axis=1)
df_merged['Err_ENU_z'] = df_merged.apply(lambda r: sq_err_1d(r.Target_U, r.Unit_U), axis=1)
df_merged['Err_ENU_2d'] = df_merged.apply(lambda r: l2_norm_2d(r.Target_E, r.Target_N, r.Unit_E,
r.Unit_N), axis=1)

# Check how far off we are with respect to the target slant range calculated in CMO.
df_merged.loc[:,['TargetRangeSlant_nm','SlantRange_ENU']].describe()
```

	TargetRangeSlant_nm	SlantRange_ENU
count	3601.000000	3601.000000
mean	8.542361	8.545815
std	1.403697	1.403840
min	7.059873	7.065747
25%	7.282954	7.288187
50%	8.114071	8.114512
75%	9.505038	9.508726
max	11.821140	11.826809

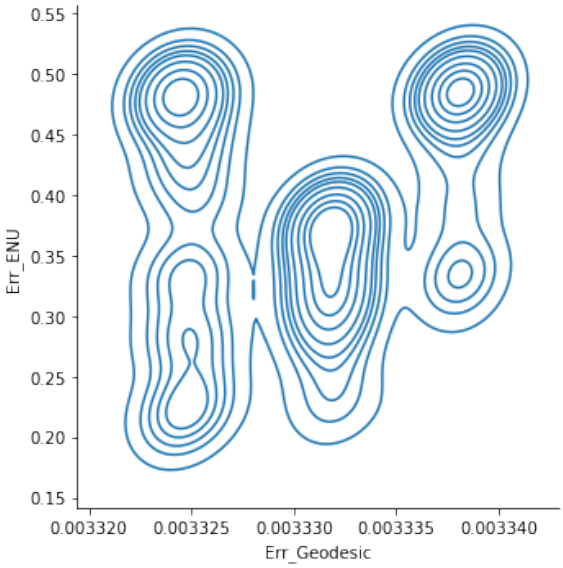
```
# the error distribution of sensor radar in this scenario.
df_merged.loc[:,['Err_ENU','Err_ENU_2d','Err_ENU_x','Err_ENU_y','Err_ENU_z']].describe()
```

	Err_ENU	Err_ENU_2d	Err_ENU_x	Err_ENU_y	Err_ENU_z
count	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000
mean	0.376391	0.003337	0.002088	0.002221	0.376376
std	0.086746	0.000163	0.000891	0.001038	0.086750
min	0.207910	0.002956	0.000005	0.000002	0.207885
25%	0.313193	0.003219	0.001129	0.002164	0.313175
50%	0.374205	0.003334	0.002321	0.002376	0.374192
75%	0.477333	0.003461	0.002548	0.003101	0.477321
max	0.490292	0.003760	0.003533	0.003684	0.490281

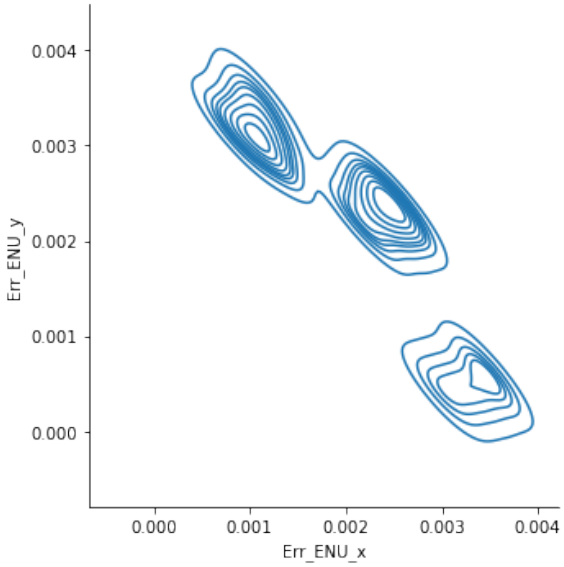
```
f, axes = plt.subplots(1, 4, figsize=(16, 5))
sns.histplot(data=df_merged, x='Err_ENU', kde=True, ax=axes[0])
for i, t in enumerate(['x', 'y', 'z']):
  sns.histplot(data=df_merged, x=f'Err_ENU_{t}', kde=True, ax=axes[i + 1])
```




```
# Visualization of distribution plot in the two different coordinate systems
sns.displot(data=df_merged, x='Err_Geodesic', y='Err_ENU', kind='kde')
```



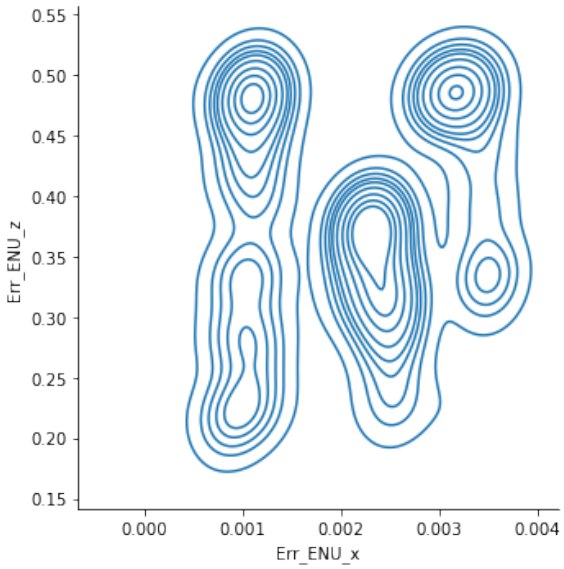

```
# Visualization of the ENU correlation error between longitude and latitude
sns.displot(data=df_merged,x='Err_ENU_x',y='Err_ENU_y', kind='kde')
# >> Seems to suggest some negative correlation in the error.
```




```
np.corrcoef(df_merged.Err_ENU_x, df_merged.Err_ENU_y)
```

```
array([[ 1.        , -0.91733101],
       [-0.91733101,  1.        ]])
```

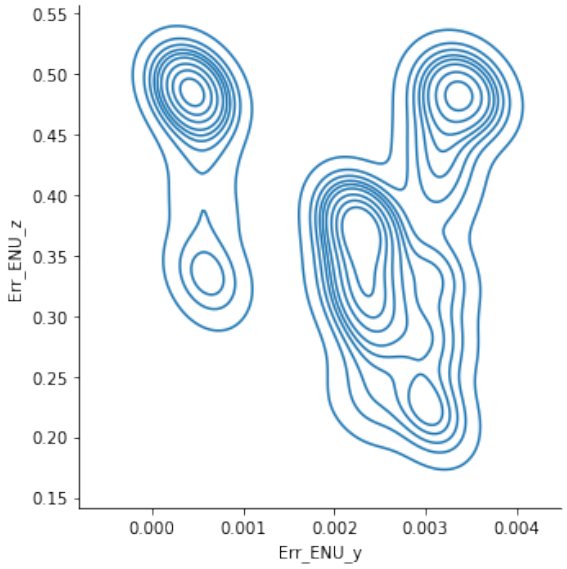
```
# Visualization of the ENU correlation error between longitude and altitude
sns.displot(data=df_merged, x='Err_ENU_x', y='Err_ENU_z', kind='kde')
# >> fairly random
```




```
np.corrcoef(df_merged.Err_ENU_x, df_merged.Err_ENU_z)
```

```
array([[1.       , 0.1382019],
       [0.1382019, 1.       ]])
```

```
# Visualization of the ENU correlation error between latitude and altitude
sns.displot(data=df_merged, x='Err_ENU_y', y='Err_ENU_z', kind='kde')
# >> fairly random
```




```
np.corrcoef(df_merged.Err_ENU_y, df_merged.Err_ENU_z)

array([[ 1.        , -0.26009732],
       [-0.26009732,  1.        ]])
```

C. EXPLORATORY DATA ANALYSIS OF TARGET UNIT POSITION DATASET

1. General Statistics of Dataset

The following tasks were carried out:

(1) Count number of entries per simulation run (each unique TimelineID) in each dataset

(2) Confirm that each simulation run yield the same number of data fields per run

(3) Check the frequency of data yield by CMO in the dataset.

```
df_pos_normal = make_targetPosition_dataset(path_normalweather)
df_pos_normal.describe()

completed

                Time  UnitLongitude   UnitLatitude     UnitCourse  \
count  720200.000000  720200.000000  720200.000000  720200.000000
mean     3601.500000    -118.992098      33.660832     209.886546
std      2079.039743       0.030194       0.025971      94.379690
min         1.000000    -119.054763      33.617448       0.276733
25%      1801.000000    -119.015539      33.637804     151.529900
50%      3601.500000    -118.987425      33.659155     193.527200
75%      5402.000000    -118.967407      33.685672     311.924800
max      7202.000000    -118.944904      33.703720     359.889900

       UnitSpeed_kts  UnitAltitude_m         Unit_E         Unit_N  \
```

```
count       720200.0        720200.0  720200.000000  720200.000000
mean            12.0             0.0      -9.027049      92.659519
std              0.0             0.0    2800.117175    2880.817604
min             12.0             0.0   -5819.901243   -4719.496264
25%             12.0             0.0   -2183.034084   -2461.850713
50%             12.0             0.0     424.335504     -93.413210
75%             12.0             0.0    2281.806939    2848.451202
max             12.0             0.0    4366.065613    4850.241856

             Unit_U  TimeDelta
count  720200.000000   720200.0
mean     -101.267654        1.0
std         0.761930        0.0
min      -103.294023        1.0
25%      -101.732566        1.0
50%      -101.153923        1.0
75%      -100.671927        1.0
max      -100.000002        1.0
```

```
# Confirm that all the value counts are equal:
all(df_pos_normal.TimelineID.value_counts().values ==
np.mean(df_pos_normal.TimelineID.value_counts().values))
```

```
True
```

```
df_pos_extreme = make_targetPosition_dataset(path_xtremeweather)
df_pos_extreme.describe()
```

```
completed
```

```
               Time  UnitLongitude   UnitLatitude     UnitCourse  \
count  720200.000000  720200.000000  720200.000000  720200.000000
mean     3601.500000    -118.992354      33.660935     210.573347
std      2079.039743       0.030425       0.025963      94.640156
min         1.000000    -119.054762      33.617485       0.279358
25%      1801.000000    -119.016254      33.637868     151.905600
50%      3601.500000    -118.987607      33.659434     193.526100
75%      5402.000000    -118.967409      33.685694     311.925800
max      7202.000000    -118.944905      33.703720     359.889400

       UnitSpeed_kts  UnitAltitude_m         Unit_E         Unit_N  \
count       720200.0        720200.0  720200.000000  720200.000000
mean            12.0             0.0     -32.693080     104.092812
std              0.0             0.0    2821.531144    2879.873852
min             12.0             0.0   -5819.821307   -4715.385638
25%             12.0             0.0   -2249.440267   -2454.791947
50%             12.0             0.0     407.492540     -62.372272
75%             12.0             0.0    2281.350168    2850.786373
max             12.0             0.0    4365.971307    4850.248208

             Unit_U  TimeDelta
count  720200.000000   720200.0
mean     -101.276908        1.0
std         0.761801        0.0
min      -103.294010        1.0
25%      -101.742425        1.0
50%      -101.159624        1.0
75%      -100.686284        1.0
max      -100.000089        1.0
```

```
# Confirm that all the value counts are equal:
all(df_pos_extreme.TimelineID.value_counts().values ==
np.mean(df_pos_extreme.TimelineID.value_counts().values))
```

```
True
```

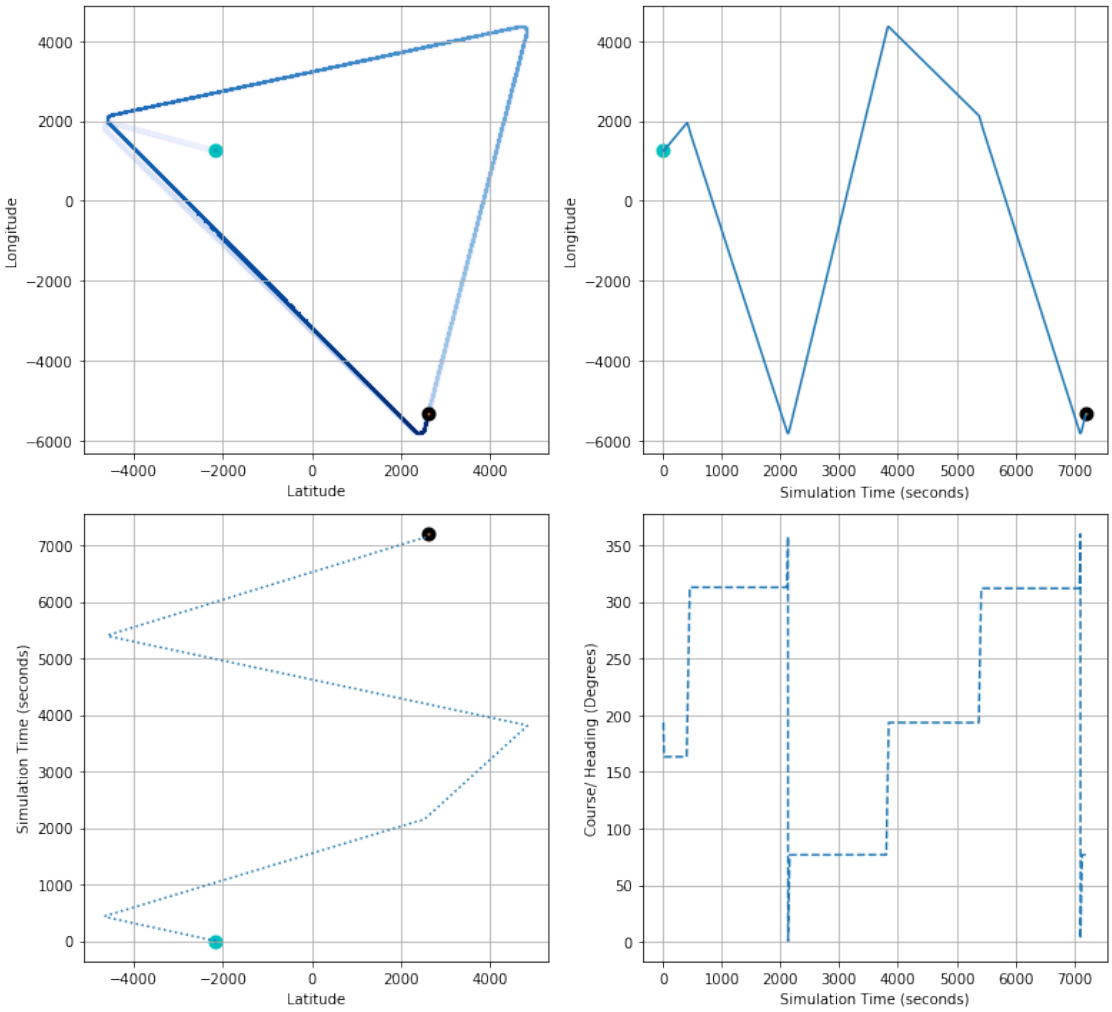
```
if SAVE_DF:
  # save target DF!
  df_pos_normal.reset_index(drop=True, inplace=True)
  df_pos_normal.to_feather('../data/df_pos_normal.ftr')
  df_pos_extreme.reset_index(drop=True, inplace=True)
  df_pos_extreme.to_feather('../data/df_pos_extreme.ftr')
```

Conclusion from Exploratory Data Analysis:

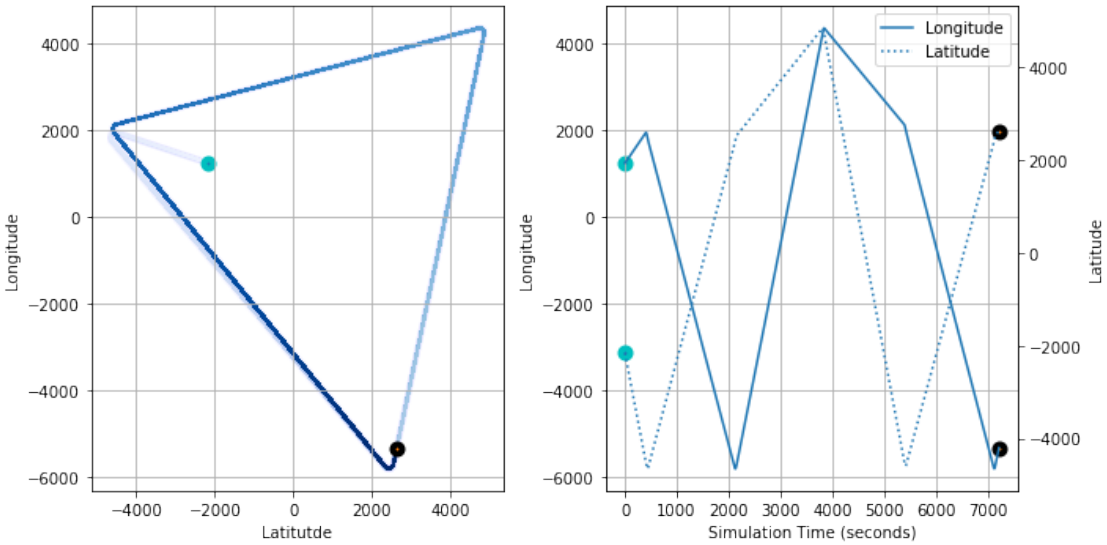
- Both normal and extreme weather datasets have the same number of target unit position data (up to 7202 timesteps, equivalent to 2 hours and 2 seconds of simulation time)
- TimelineID is unique, total 100 unique TimelineID in each dataset
- Each “simulation pulse” is equivalent to 1 second in real time. each entry represents the state of the target for that second.

2. Visualization of Target Movement in Simulation

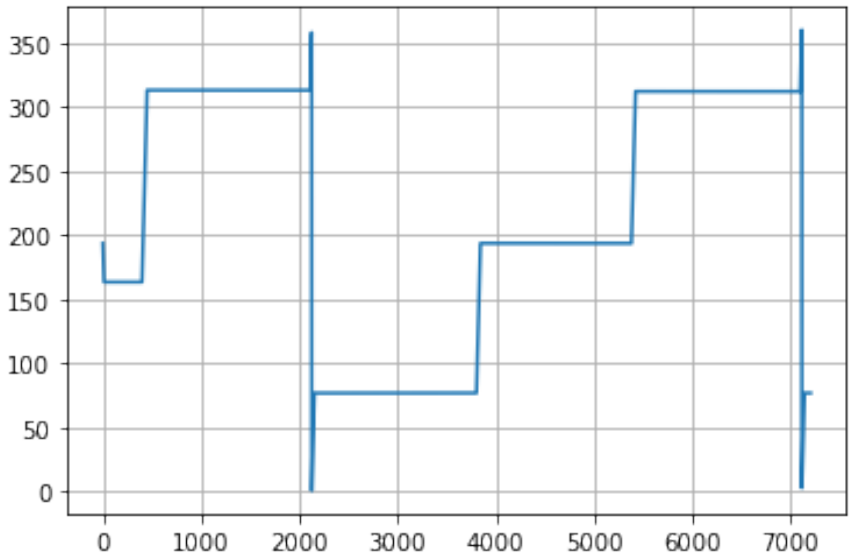
```
df_pos_sampled = extract_target_unitpos(os.path.join(path_xtremeweather, '11',
'Target_UnitPositions.csv'))
plot_tgt_latlon_time_grid(df_pos_sampled)
```



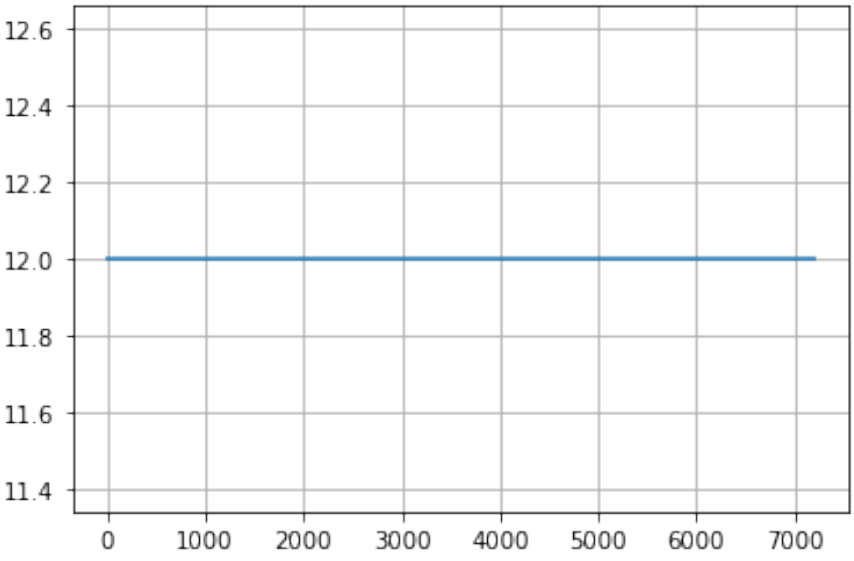

```
plot_tgt_latlon_time(df_pos_sampled)
```



```
ax = df_pos_sampled.UnitCourse.plot()
ax.grid()
```




```
# Constant Speed (magnitude, does not consider the direction i.e., the course)
ax = df_pos_sampled.UnitSpeed_kts.plot()
ax.grid()
```



D. EXPLORATORY DATA ANALYSIS OF SENSOR DATASET

We aim to answer the following questions:

- What is the difference in sensor measurements in different weather conditions? Are there notable differences for each sensors?
- What is the time between detection for each of the sensor? Are they all equal?
- What is the average number of detections for each sensor in a given simulation run? What is the periodicity per sensor?
- For each of the sensors, what is its performance with respect to the actual target position?

1. Periodicity of Data, Number of Detection, Failure Rate in Dataset

- ESM Sensor

```
# create ESM dataset
df_esm_normal = make_sensoDetection_dataset(path_normalweather, sensor_name='ESM')
df_esm_normal.describe()

completed
```

```
              Time  SensorParentLongitude  SensorParentLatitude  \
count  36000.000000           3.600000e+04          36000.000000
mean    3604.000000          -1.190609e+02             33.762466
std     2078.481818           2.842210e-14              0.000000
min       14.000000          -1.190609e+02             33.762466
25%     1809.000000          -1.190609e+02             33.762466
50%     3604.000000          -1.190609e+02             33.762466
75%     5399.000000          -1.190609e+02             33.762466
max     7194.000000          -1.190609e+02             33.762466

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   36000.0     36000.000000    36000.000000
mean                      888.0      -118.992107       33.660829
std                         0.0         0.030203        0.025972
min                       888.0      -119.054747       33.617455
25%                       888.0      -119.015619       33.637815
50%                       888.0      -118.987425       33.659152
75%                       888.0      -118.967405       33.685674
max                       888.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          36000.000000         36000.000000  36000.000000  36000.000000
mean             693.702000             7.164574     -9.822697     92.384244
std              159.271411             1.527780   2801.229704   2881.228633
min              385.000000             4.155985  -5818.960479  -4719.046097
```

```
25%              580.000000             5.813999  -2191.126705  -2460.934385
50%              687.000000             7.228280    424.538082    -93.738987
75%              884.000000             8.467664   2282.264226   2848.953632
max              919.000000             9.754641   4366.579380   4850.744997

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  36000.000000  3.600000e+04  3.600000e+04  3.600000e+04  36000.000000
mean     592.433848 -6.380407e+03  1.136893e+04  7.746446e+02     19.983333
std      159.204976  1.819015e-12  1.819015e-12  2.273768e-13      0.315793
min      283.533681 -6.380407e+03  1.136893e+04  7.746446e+02     14.000000
25%      477.006498 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
50%      586.643116 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
75%      782.790066 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
max      818.391497 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
```

```
# Check the TimeDelta, if there are any deviations in the periodicity of the sensor data.
df_esm_normal.TimeDelta.value_counts()
# >> There are 14s, 100 in total, because 100 simulation, TimeDelta at start is 14-0=14
```

```
20.0    35900
14.0      100
Name: TimeDelta, dtype: int64
```

```
# Check the DetectionResult:
df_esm_normal.DetectionResult.value_counts()
```

```
SUCCESS    36000
Name: DetectionResult, dtype: int64
```

```
# create ESM dataset for extreme_weather
df_esm_extreme = make_sensoDetection_dataset(path_xtremeweather, sensor_name='ESM')
df_esm_normal.describe()
```

```
completed

              Time  SensorParentLongitude  SensorParentLatitude  \
count  36000.000000           3.600000e+04          36000.000000
mean    3604.000000          -1.190609e+02             33.762466
std     2078.481818           2.842210e-14              0.000000
min       14.000000          -1.190609e+02             33.762466
25%     1809.000000          -1.190609e+02             33.762466
50%     3604.000000          -1.190609e+02             33.762466
75%     5399.000000          -1.190609e+02             33.762466
max     7194.000000          -1.190609e+02             33.762466

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   36000.0     36000.000000    36000.000000
mean                      888.0      -118.992107       33.660829
std                         0.0         0.030203        0.025972
min                       888.0      -119.054747       33.617455
25%                       888.0      -119.015619       33.637815
50%                       888.0      -118.987425       33.659152
75%                       888.0      -118.967405       33.685674
max                       888.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          36000.000000         36000.000000  36000.000000  36000.000000
mean             693.702000             7.164574     -9.822697     92.384244
std              159.271411             1.527780   2801.229704   2881.228633
min              385.000000             4.155985  -5818.960479  -4719.046097
25%              580.000000             5.813999  -2191.126705  -2460.934385
50%              687.000000             7.228280    424.538082    -93.738987
75%              884.000000             8.467664   2282.264226   2848.953632
max              919.000000             9.754641   4366.579380   4850.744997

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  36000.000000  3.600000e+04  3.600000e+04  3.600000e+04  36000.000000
```

```
mean     592.433848 -6.380407e+03  1.136893e+04  7.746446e+02     19.983333
std      159.204976  1.819015e-12  1.819015e-12  2.273768e-13      0.315793
min      283.533681 -6.380407e+03  1.136893e+04  7.746446e+02     14.000000
25%      477.006498 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
50%      586.643116 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
75%      782.790066 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
max      818.391497 -6.380407e+03  1.136893e+04  7.746446e+02     20.000000
```

```
df_esm_extreme.TimeDelta.value_counts()
```

```
20.0    35900
14.0      100
Name: TimeDelta, dtype: int64
```

```
df_esm_extreme.DetectionResult.value_counts()
```

```
SUCCESS    36000
Name: DetectionResult, dtype: int64
```

Frequency of ESM sensor is every 20 seconds

- EO Sensor

```
# create EO dataset
df_eo_normal = make_sensoDetection_dataset(path_normalweather, sensor_name='EO')
df_eo_normal.describe()
```

```
completed
```

```
              Time  SensorParentLongitude  SensorParentLatitude  \
count  72006.000000           7.200600e+04          72006.000000
mean    3597.909035          -1.188638e+02             33.754513
std     2078.510558           7.105477e-14              0.000000
min        3.000000          -1.188638e+02             33.754513
25%     1793.000000          -1.188638e+02             33.754513
50%     3598.000000          -1.188638e+02             33.754513
75%     5393.000000          -1.188638e+02             33.754513
max     7193.000000          -1.188638e+02             33.754513

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72006.0     72006.000000    72006.000000
mean                      903.0      -118.992067       33.660833
std                         0.0         0.030179        0.025976
min                       903.0      -119.054756       33.617448
25%                       903.0      -119.015471       33.637794
50%                       903.0      -118.987380       33.659148
75%                       903.0      -118.967404       33.685679
max                       903.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72006.000000         72006.000000  72006.000000  72006.000000
mean             693.810432             8.664566     -6.101467     92.842988
std              159.377596             1.485807   2799.021389   2881.652354
min              385.000000             5.092077  -5819.775043  -4719.805203
25%              580.000000             7.707906  -2177.360386  -2463.121653
50%              687.000000             9.384436    428.445750    -94.139975
75%              884.000000             9.744898   2282.353082   2849.409924
max              919.000000            10.487470   4366.533488   4850.804139

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72006.000000  7.200600e+04  7.200600e+04  72006.000000  72006.000000
mean     592.543037  1.187992e+04  1.049199e+04    783.289474      9.989445
std      159.311443  1.819002e-12  5.457006e-12      0.000000      0.276174
min      283.533591  1.187992e+04  1.049199e+04    783.289474      0.000000
25%      477.002299  1.187992e+04  1.049199e+04    783.289474     10.000000
```

```
50%      586.644457  1.187992e+04  1.049199e+04    783.289474     10.000000
75%      782.792665  1.187992e+04  1.049199e+04    783.289474     10.000000
max      818.407433  1.187992e+04  1.049199e+04    783.289474     10.000000
```

```
df_eo_normal.TimeDelta.value_counts()
# Starts at time
```

```
10.0    71900
3.0       100
0.0         6
Name: TimeDelta, dtype: int64
```

```
# Remove those rows that have timedelta = 0.0
df_eo_normal = df_eo_normal[df_eo_normal.TimeDelta != 0.0]
# Confirm that timedelta = 0. are removed. See the last column, std = 0.
df_eo_normal.describe()
```

```
               Time  SensorParentLongitude  SensorParentLatitude  \
count  72000.000000           7.200000e+04          72000.000000
mean    3598.000000          -1.188638e+02             33.754513
std     2078.473398           8.526572e-14              0.000000
min        3.000000          -1.188638e+02             33.754513
25%     1800.500000          -1.188638e+02             33.754513
50%     3598.000000          -1.188638e+02             33.754513
75%     5395.500000          -1.188638e+02             33.754513
max     7193.000000          -1.188638e+02             33.754513

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72000.0     72000.000000    72000.000000
mean                      903.0      -118.992069       33.660833
std                         0.0         0.030179        0.025976
min                       903.0      -119.054756       33.617448
25%                       903.0      -119.015474       33.637794
50%                       903.0      -118.987390       33.659149
75%                       903.0      -118.967404       33.685679
max                       903.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72000.000000         72000.000000  72000.000000  72000.000000
mean             693.808403             8.664635     -6.258486     92.838562
std              159.378240             1.485811   2799.059415   2881.679140
min              385.000000             5.092077  -5819.775043  -4719.805203
25%              580.000000             7.707944  -2177.436968  -2463.144540
50%              687.000000             9.384846    427.795137    -93.971853
75%              884.000000             9.744924   2282.351654   2849.314603
max              919.000000            10.487470   4366.533488   4850.804139

          Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72000.000000  72000.000000  7.200000e+04  72000.000000  72000.000000
mean     592.540978  11879.917919  1.049199e+04    783.289474      9.990278
std      159.312117      0.000000  5.457006e-12      0.000000      0.260695
min      283.533591  11879.917919  1.049199e+04    783.289474      3.000000
25%      477.002268  11879.917919  1.049199e+04    783.289474     10.000000
50%      586.644457  11879.917919  1.049199e+04    783.289474     10.000000
75%      782.792572  11879.917919  1.049199e+04    783.289474     10.000000
max      818.407433  11879.917919  1.049199e+04    783.289474     10.000000
```

```
df_eo_normal.DetectionResult.value_counts()
```

```
SUCCESS    72000
Name: DetectionResult, dtype: int64
```

```
# create EO dataset (for extreme weather condition)
df_eo_extreme = make_sensoDetection_dataset(path_xtremeweather, sensor_name='EO')
df_eo_extreme.describe()
```

```
completed
```

```
              Time  SensorParentLongitude  SensorParentLatitude  \
count  72004.000000           7.200400e+04          72004.000000
mean    3597.979168          -1.188638e+02             33.754513
std     2078.458621           7.105477e-14              0.000000
min        3.000000          -1.188638e+02             33.754513
25%     1800.500000          -1.188638e+02             33.754513
50%     3598.000000          -1.188638e+02             33.754513
75%     5393.000000          -1.188638e+02             33.754513
max     7193.000000          -1.188638e+02             33.754513

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72004.0     72004.000000    72004.000000
mean                      903.0      -118.992321       33.660935
std                         0.0         0.030410        0.025967
min                       903.0      -119.054761       33.617493
25%                       903.0      -119.016182       33.637856
50%                       903.0      -118.987563       33.659429
75%                       903.0      -118.967400       33.685701
max                       903.0      -118.944907       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72004.000000         72004.000000  72004.000000  72004.000000
mean             693.394784             8.671238    -29.627579    104.185974
std              159.363265             1.489474   2820.421546   2880.725599
min              385.000000             5.092161  -5820.208383  -4714.832246
25%              580.000000             7.703418  -2243.371949  -2456.290195
50%              687.000000             9.415813    411.456345    -62.767558
75%              884.000000             9.747221   2282.375877   2851.928632
max              919.000000            10.487660   4366.390027   4850.753869

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72004.000000  72004.000000  7.200400e+04  72004.000000  72004.000000
mean     592.118150  11879.917919  1.049199e+04    783.289474      9.989723
std      159.298687      0.000000  5.457006e-12      0.000000      0.271113
min      283.533859  11879.917919  1.049199e+04    783.289474      0.000000
25%      477.007620  11879.917919  1.049199e+04    783.289474     10.000000
50%      586.641710  11879.917919  1.049199e+04    783.289474     10.000000
75%      782.796552  11879.917919  1.049199e+04    783.289474     10.000000
max      818.410904  11879.917919  1.049199e+04    783.289474     10.000000
```

```
df_eo_extreme.TimeDelta.value_counts()
```

```
10.0    71900
3.0       100
0.0         4
Name: TimeDelta, dtype: int64
```

```
df_eo_extreme = df_eo_extreme[df_eo_extreme.TimeDelta != 0.]
df_eo_extreme.describe()
```

```
              Time  SensorParentLongitude  SensorParentLatitude  \
count  72000.000000           7.200000e+04          72000.000000
mean    3598.000000          -1.188638e+02             33.754513
std     2078.473398           8.526572e-14              0.000000
min        3.000000          -1.188638e+02             33.754513
25%     1800.500000          -1.188638e+02             33.754513
50%     3598.000000          -1.188638e+02             33.754513
75%     5395.500000          -1.188638e+02             33.754513
max     7193.000000          -1.188638e+02             33.754513

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72000.0     72000.000000    72000.000000
mean                      903.0      -118.992322       33.660934
std                         0.0         0.030410        0.025967
min                       903.0      -119.054761       33.617493
25%                       903.0      -119.016187       33.637855
50%                       903.0      -118.987568       33.659428
```

```
75%                     903.0      -118.967403       33.685701
max                     903.0      -118.944907       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72000.000000         72000.000000  72000.000000  72000.000000
mean             693.384306             8.671354    -29.777571    104.046248
std              159.361327             1.489421   2820.420844   2880.709446
min              385.000000             5.092161  -5820.208383  -4714.832246
25%              580.000000             7.704518  -2243.737269  -2456.344871
50%              687.000000             9.416157    411.164526    -63.099273
75%              884.000000             9.747223   2282.345251   2851.889394
max              919.000000            10.487660   4366.390027   4850.753869

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72000.000000  72000.000000  7.200000e+04  72000.000000  72000.000000
mean     592.107681  11879.917919  1.049199e+04    783.289474      9.990278
std      159.296763      0.000000  5.457006e-12      0.000000      0.260695
min      283.533859  11879.917919  1.049199e+04    783.289474      3.000000
25%      477.007476  11879.917919  1.049199e+04    783.289474     10.000000
50%      586.641540  11879.917919  1.049199e+04    783.289474     10.000000
75%      782.796391  11879.917919  1.049199e+04    783.289474     10.000000
max      818.410904  11879.917919  1.049199e+04    783.289474     10.000000

df_eo_extreme.DetectionResult.value_counts()

SUCCESS    72000
Name: DetectionResult, dtype: int64
```

Frequency of EO sensor is every 10 seconds

- IR Sensor

```
# create IR dataset
df_ir_normal = make_sensoDetection_dataset(path_normalweather, sensor_name='IR')
df_ir_normal.describe()

completed

              Time  SensorParentLongitude  SensorParentLatitude  \
count  72005.000000           7.200500e+04          7.200500e+04
mean    3597.844941          -1.188296e+02          3.358266e+01
std     2078.504077           4.263286e-14          7.105477e-15
min        3.000000          -1.188296e+02          3.358266e+01
25%     1793.000000          -1.188296e+02          3.358266e+01
50%     3593.000000          -1.188296e+02          3.358266e+01
75%     5393.000000          -1.188296e+02          3.358266e+01
max     7193.000000          -1.188296e+02          3.358266e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72005.0     72005.000000    72005.000000
mean                      607.0      -118.992069       33.660832
std                         0.0         0.030180        0.025976
min                       607.0      -119.054756       33.617448
25%                       607.0      -119.015475       33.637794
50%                       607.0      -118.987391       33.659149
75%                       607.0      -118.967405       33.685677
max                       607.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72005.000000         72005.000000  72005.000000  72005.000000
mean             693.803861             9.494121     -6.337075     92.806810
std              159.378527             1.557213   2799.094753   2881.659295
min              385.000000             7.302612  -5819.775043  -4719.805203
25%              580.000000             8.112456  -2177.637142  -2463.128034
50%              687.000000             9.296807    427.709268    -93.936991
```

```
75%             884.000000           10.705660   2282.322727   2849.119877
max             919.000000           12.730020   4366.533488   4850.804139

          Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72005.000000  72005.000000  7.200500e+04  7.200500e+04  72005.000000
mean     592.536430  15074.295905 -8.567119e+03  4.834323e+02      9.989584
std      159.312455      0.000000  5.457006e-12  3.410629e-13      0.273656
min      283.533591  15074.295905 -8.567119e+03  4.834323e+02      0.000000
25%      477.002269  15074.295905 -8.567119e+03  4.834323e+02     10.000000
50%      586.644438  15074.295905 -8.567119e+03  4.834323e+02     10.000000
75%      782.792466  15074.295905 -8.567119e+03  4.834323e+02     10.000000
max      818.407433  15074.295905 -8.567119e+03  4.834323e+02     10.000000

df_ir_normal.TimeDelta.value_counts()

10.0    71900
3.0       100
0.0         5
Name: TimeDelta, dtype: int64

df_ir_normal = df_ir_normal[df_ir_normal.TimeDelta != 0]
df_ir_normal.describe()

              Time  SensorParentLongitude  SensorParentLatitude  \
count  72000.000000           7.200000e+04          7.200000e+04
mean    3598.000000          -1.188296e+02          3.358266e+01
std     2078.473398           2.842191e-14          7.105477e-15
min        3.000000          -1.188296e+02          3.358266e+01
25%     1800.500000          -1.188296e+02          3.358266e+01
50%     3598.000000          -1.188296e+02          3.358266e+01
75%     5395.500000          -1.188296e+02          3.358266e+01
max     7193.000000          -1.188296e+02          3.358266e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72000.0     72000.000000    72000.000000
mean                      607.0      -118.992069       33.660833
std                         0.0         0.030179        0.025976
min                       607.0      -119.054756       33.617448
25%                       607.0      -119.015474       33.637794
50%                       607.0      -118.987390       33.659149
75%                       607.0      -118.967404       33.685679
max                       607.0      -118.944905       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72000.000000         72000.000000  72000.000000  72000.000000
mean             693.808403             9.494098     -6.258405     92.839063
std              159.378240             1.557179   2799.059538   2881.679526
min              385.000000             7.302612  -5819.775043  -4719.805203
25%              580.000000             8.112457  -2177.436968  -2463.144540
50%              687.000000             9.296772    427.795137    -93.971853
75%              884.000000            10.705652   2282.351654   2849.314603
max              919.000000            12.730020   4366.533488   4850.804139

          Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72000.000000  7.200000e+04  7.200000e+04  7.200000e+04  72000.000000
mean     592.540978  1.507430e+04 -8.567119e+03  4.834323e+02      9.990278
std      159.312117  1.819002e-12  5.457006e-12  3.410629e-13      0.260695
min      283.533591  1.507430e+04 -8.567119e+03  4.834323e+02      3.000000
25%      477.002268  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
50%      586.644457  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
75%      782.792572  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
max      818.407433  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000

df_ir_normal.DetectionResult.value_counts()

SUCCESS    72000
Name: DetectionResult, dtype: int64
```

```
# create IR dataset
df_ir_extreme = make_sensoDetection_dataset(path_xtremeweather, sensor_name='IR')
df_ir_extreme.describe()

completed

              Time  SensorParentLongitude  SensorParentLatitude  \
count  72001.000000           7.200100e+04          7.200100e+04
mean    3597.960487          -1.188296e+02          3.358266e+01
std     2078.486007           4.263286e-14          7.105477e-15
min        3.000000          -1.188296e+02          3.358266e+01
25%     1793.000000          -1.188296e+02          3.358266e+01
50%     3593.000000          -1.188296e+02          3.358266e+01
75%     5393.000000          -1.188296e+02          3.358266e+01
max     7193.000000          -1.188296e+02          3.358266e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                   72001.0     72001.000000    72001.000000
mean                      607.0      -118.992322       33.660933
std                         0.0         0.030410        0.025967
min                       607.0      -119.054761       33.617493
25%                       607.0      -119.016187       33.637853
50%                       607.0      -118.987567       33.659427
75%                       607.0      -118.967404       33.685701
max                       607.0      -118.944907       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72001.000000         72001.000000  72001.000000  72001.000000
mean             693.381161             9.507952    -29.759436    104.002125
std              159.362454             1.567453   2820.405710   2880.713922
min              385.000000             7.302612  -5820.208383  -4714.832246
25%              580.000000             8.111111  -2243.712115  -2456.355915
50%              687.000000             9.312607    411.219979    -63.113737
75%              884.000000            10.740300   2282.336923   2851.885242
max              919.000000            12.729870   4366.390027   4850.753869

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72001.000000  7.200100e+04  7.200100e+04  7.200100e+04  72001.000000
mean     592.104542  1.507430e+04 -8.567119e+03  4.834323e+02      9.990139
std      159.297883  1.819002e-12  5.457006e-12  2.842191e-13      0.263339
min      283.533859  1.507430e+04 -8.567119e+03  4.834323e+02      0.000000
25%      477.007354  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
50%      586.641513  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
75%      782.796353  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
max      818.410904  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000

df_ir_extreme.TimeDelta.value_counts()

10.0    71900
3.0       100
0.0         1
Name: TimeDelta, dtype: int64

df_ir_extreme = df_ir_extreme[df_ir_extreme.TimeDelta != 0]
df_ir_extreme.describe()

              Time  SensorParentLongitude  SensorParentLatitude  \
count  72000.000000           7.200000e+04          7.200000e+04
mean    3598.000000          -1.188296e+02          3.358266e+01
std     2078.473398           2.842191e-14          7.105477e-15
min        3.000000          -1.188296e+02          3.358266e+01
25%     1800.500000          -1.188296e+02          3.358266e+01
50%     3598.000000          -1.188296e+02          3.358266e+01
75%     5395.500000          -1.188296e+02          3.358266e+01
max     7193.000000          -1.188296e+02          3.358266e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
```

```
count                   72000.0     72000.000000    72000.000000
mean                      607.0      -118.992322       33.660934
std                         0.0         0.030410        0.025967
min                       607.0      -119.054761       33.617493
25%                       607.0      -119.016187       33.637855
50%                       607.0      -118.987568       33.659428
75%                       607.0      -118.967403       33.685701
max                       607.0      -118.944907       33.703719

       TargetAltitude_AGL_m  TargetRangeSlant_nm      Target_E      Target_N  \
count          72000.000000         72000.000000  72000.000000  72000.000000
mean             693.384306             9.507973    -29.777824    104.046549
std              159.361327             1.567454   2820.420980   2880.709263
min              385.000000             7.302612  -5820.208383  -4714.832246
25%              580.000000             8.111196  -2243.737269  -2456.344871
50%              687.000000             9.312612    411.164526    -63.099273
75%              884.000000            10.740325   2282.345251   2851.889394
max              919.000000            12.729870   4366.390027   4850.753869

           Target_U      Sensor_E      Sensor_N      Sensor_U     TimeDelta
count  72000.000000  7.200000e+04  7.200000e+04  7.200000e+04  72000.000000
mean     592.107681  1.507430e+04 -8.567119e+03  4.834323e+02      9.990278
std      159.296763  1.819002e-12  5.457006e-12  3.410629e-13      0.260695
min      283.533859  1.507430e+04 -8.567119e+03  4.834323e+02      3.000000
25%      477.007476  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
50%      586.641540  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
75%      782.796391  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
max      818.410904  1.507430e+04 -8.567119e+03  4.834323e+02     10.000000
```

```
df_ir_extreme.DetectionResult.value_counts()  # NOTE THE LARGE NUMBER OF FAILURE!
```

```
SUCCESS    38604
FAILURE    33396
Name: DetectionResult, dtype: int64
```

Frequency of IR sensor is every 10 seconds

- Radar Sensor

```
# create Radar dataset
df_radar_normal = make_sensoDetection_dataset(path_normalweather, sensor_name='Radar')
df_radar_normal.describe()
```

```
completed
```

```
                Time  SensorParentLongitude  SensorParentLatitude  \
count  360147.000000           3.601470e+05          3.601470e+05
mean     3600.866843          -1.190876e+02          3.354606e+01
std      2079.116050           5.684350e-14          2.842175e-14
min         1.000000          -1.190876e+02          3.354606e+01
25%      1801.000000          -1.190876e+02          3.354606e+01
50%      3601.000000          -1.190876e+02          3.354606e+01
75%      5401.000000          -1.190876e+02          3.354606e+01
max      7201.000000          -1.190876e+02          3.354606e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                  360147.0    360147.000000   360147.000000
mean                      177.0      -118.992089       33.660832
std                         0.0         0.030189        0.025972
min                       177.0      -119.054763       33.617448
25%                       177.0      -119.015520       33.637800
50%                       177.0      -118.987410       33.659155
75%                       177.0      -118.967407       33.685674
max                       177.0      -118.944905       33.703720
```

```
       TargetAltitude_AGL_m  TargetRangeSlant_nm       Target_E  \
count         360147.000000        360147.000000  360147.000000
mean             693.756125             8.541511      -8.177971
std              159.339332             1.405408    2799.914827
min              385.000000             7.030613   -5820.429935
25%              580.000000             7.287161   -2181.938157
50%              687.000000             8.066295     425.734235
75%              884.000000             9.528920    2282.070128
max              919.000000            11.821320    4366.579380

            Target_N       Target_U      Sensor_E      Sensor_N  \
count  360147.000000  360147.000000  3.601470e+05  3.601470e+05
mean       92.756207     592.488525 -8.883363e+03 -1.263428e+04
std      2881.204708     159.273136  5.456976e-12  3.637984e-12
min     -4719.805203     283.533361 -8.883363e+03 -1.263428e+04
25%     -2462.362747     477.004657 -8.883363e+03 -1.263428e+04
50%       -93.434946     586.644092 -8.883363e+03 -1.263428e+04
75%      2848.831384     782.791056 -8.883363e+03 -1.263428e+04
max      4850.929508     818.407433 -8.883363e+03 -1.263428e+04

            Sensor_U      TimeDelta
count  360147.000000  360147.000000
mean       58.261496       1.999461
std         0.000000       0.028273
min        58.261496       0.000000
25%        58.261496       2.000000
50%        58.261496       2.000000
75%        58.261496       2.000000
max        58.261496       2.000000

df_radar_normal.TimeDelta.value_counts()

2.0    360000
1.0       100
0.0        47
Name: TimeDelta, dtype: int64

df_radar_normal = df_radar_normal[df_radar_normal.TimeDelta != 0]
df_radar_normal.describe()

                Time  SensorParentLongitude  SensorParentLatitude  \
count  360100.000000           3.601000e+05          3.601000e+05
mean     3601.000000          -1.190876e+02          3.354606e+01
std      2079.041126           5.684350e-14          2.842175e-14
min         1.000000          -1.190876e+02          3.354606e+01
25%      1801.000000          -1.190876e+02          3.354606e+01
50%      3601.000000          -1.190876e+02          3.354606e+01
75%      5401.000000          -1.190876e+02          3.354606e+01
max      7201.000000          -1.190876e+02          3.354606e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                  360100.0    360100.000000   360100.000000
mean                      177.0      -118.992090       33.660833
std                         0.0         0.030189        0.025972
min                       177.0      -119.054763       33.617448
25%                       177.0      -119.015520       33.637800
50%                       177.0      -118.987412       33.659156
75%                       177.0      -118.967407       33.685674
max                       177.0      -118.944905       33.703720

       TargetAltitude_AGL_m  TargetRangeSlant_nm       Target_E  \
count         360100.000000        360100.000000  360100.000000
mean             693.756934             8.541522      -8.249560
std              159.337394             1.405437    2799.969614
min              385.000000             7.030613   -5820.429935
25%              580.000000             7.287151   -2181.951401
50%              687.000000             8.066305     425.637259
```

```
75%              884.000000             9.529140    2282.069816
max              919.000000            11.821320    4366.579380

            Target_N       Target_U      Sensor_E      Sensor_N  \
count  360100.000000  360100.000000  3.601000e+05  3.601000e+05
mean       92.824525     592.489297 -8.883363e+03 -1.263428e+04
std      2881.231633     159.271177  5.456976e-12  1.818992e-12
min     -4719.805203     283.533361 -8.883363e+03 -1.263428e+04
25%     -2462.360313     477.004735 -8.883363e+03 -1.263428e+04
50%       -93.236604     586.644093 -8.883363e+03 -1.263428e+04
75%      2848.948128     782.791065 -8.883363e+03 -1.263428e+04
max      4850.929508     818.407433 -8.883363e+03 -1.263428e+04

            Sensor_U     TimeDelta
count  360100.000000  360100.000000
mean       58.261496       1.999722
std         0.000000       0.016662
min        58.261496       1.000000
25%        58.261496       2.000000
50%        58.261496       2.000000
75%        58.261496       2.000000
max        58.261496       2.000000

df_radar_normal.DetectionResult.value_counts()

SUCCESS    360100
Name: DetectionResult, dtype: int64

# create Radar dataset
df_radar_extreme = make_sensoDetection_dataset(path_xtremeweather, sensor_name='Radar')
df_radar_extreme.describe()

completed

                Time  SensorParentLongitude  SensorParentLatitude  \
count  360206.000000           3.602060e+05          3.602060e+05
mean     3600.742825          -1.190876e+02          3.354606e+01
std      2079.086320           5.684350e-14          2.842175e-14
min         1.000000          -1.190876e+02          3.354606e+01
25%      1801.000000          -1.190876e+02          3.354606e+01
50%      3601.000000          -1.190876e+02          3.354606e+01
75%      5401.000000          -1.190876e+02          3.354606e+01
max      7201.000000          -1.190876e+02          3.354606e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                  360206.0    360206.000000   360206.000000
mean                      177.0      -118.992345       33.660934
std                         0.0         0.030419        0.025963
min                       177.0      -119.054761       33.617485
25%                       177.0      -119.016219       33.637867
50%                       177.0      -118.987596       33.659432
75%                       177.0      -118.967409       33.685694
max                       177.0      -118.944905       33.703720

       TargetAltitude_AGL_m  TargetRangeSlant_nm       Target_E  \
count         360206.000000        360206.000000  360206.000000
mean             693.323726             8.541366     -31.895451
std              159.322074             1.404933    2821.314200
min                0.000000             7.034362   -5820.208383
25%              580.000000             7.288851   -2246.500214
50%              687.000000             8.062613     408.664318
75%              884.000000             9.528897    2281.445486
max              919.000000            11.821310    4366.557253

            Target_N       Target_U      Sensor_E      Sensor_N      Sensor_U  \
count  360206.000000  360206.000000  3.602060e+05  3.602060e+05  3.602060e+05
mean      104.135351     592.046883 -8.883363e+03 -1.263428e+04  5.826150e+01
```

```
std      2880.262490     159.257441  5.456976e-12  1.818992e-12  7.105437e-15
min     -4715.694308    -100.834577 -8.883363e+03 -1.263428e+04  5.826150e+01
25%     -2455.097115     477.008369 -8.883363e+03 -1.263428e+04  5.826150e+01
50%       -62.485051     586.641057 -8.883363e+03 -1.263428e+04  5.826150e+01
75%      2851.109719     782.795140 -8.883363e+03 -1.263428e+04  5.826150e+01
max      4850.924599     818.426246 -8.883363e+03 -1.263428e+04  5.826150e+01

           TimeDelta
count  360206.000000
mean        1.999134
std         0.038131
min         0.000000
25%         2.000000
50%         2.000000
75%         2.000000
max         2.000000
```

```
df_radar_extreme.TimeDelta.value_counts()
```

```
2.0    360000
0.0       106
1.0       100
Name: TimeDelta, dtype: int64
```

```
df_radar_extreme = df_radar_extreme[df_radar_extreme.TimeDelta != 0.]
df_radar_extreme.describe()
```

```
                Time  SensorParentLongitude  SensorParentLatitude  \
count  360100.000000           3.601000e+05          3.601000e+05
mean     3601.000000          -1.190876e+02          3.354606e+01
std      2079.041126           5.684350e-14          2.842175e-14
min         1.000000          -1.190876e+02          3.354606e+01
25%      1801.000000          -1.190876e+02          3.354606e+01
50%      3601.000000          -1.190876e+02          3.354606e+01
75%      5401.000000          -1.190876e+02          3.354606e+01
max      7201.000000          -1.190876e+02          3.354606e+01

       SensorParentAltitude_AGL  TargetLongitude  TargetLatitude  \
count                  360100.0    360100.000000   360100.000000
mean                      177.0      -118.992345       33.660935
std                         0.0         0.030420        0.025963
min                       177.0      -119.054761       33.617485
25%                       177.0      -119.016219       33.637867
50%                       177.0      -118.987596       33.659433
75%                       177.0      -118.967408       33.685694
max                       177.0      -118.944905       33.703720

       TargetAltitude_AGL_m  TargetRangeSlant_nm       Target_E  \
count         360100.000000        360100.000000  360100.000000
mean            693.322646             8.541402     -31.863930
std             159.325000             1.404936    2821.370839
min               0.000000             7.034362   -5820.208383
25%             580.000000             7.288854   -2246.531834
50%             687.000000             8.062624     408.680739
75%             884.000000             9.529084    2281.542512
max             919.000000            11.821310    4366.557253

            Target_N       Target_U      Sensor_E      Sensor_N  \
count  360100.000000  360100.000000  3.601000e+05  3.601000e+05
mean      104.179312     592.045769 -8.883363e+03 -1.263428e+04
std      2880.282450     159.260372  5.456976e-12  1.818992e-12
min     -4715.694308    -100.834577 -8.883363e+03 -1.263428e+04
25%     -2455.104375     477.008353 -8.883363e+03 -1.263428e+04
50%       -62.471799     586.641054 -8.883363e+03 -1.263428e+04
75%      2851.147074     782.795196 -8.883363e+03 -1.263428e+04
max      4850.924599     818.426246 -8.883363e+03 -1.263428e+04
```

```
            Sensor_U      TimeDelta
count  360100.000000  360100.000000
mean       58.261496       1.999722
std         0.000000       0.016662
min        58.261496       1.000000
25%        58.261496       2.000000
50%        58.261496       2.000000
75%        58.261496       2.000000
max        58.261496       2.000000

df_radar_extreme.DetectionResult.value_counts()

SUCCESS    360098
FAILURE         2
Name: DetectionResult, dtype: int64
```

Frequency of Radar is every 2 seconds

(1) Observation:

Number of entries per scenario run for normal and extreme weather remains unchanged for target position and sensors.

$$ExpectedPeriod = \frac{\text{Number of target positions}}{\text{Number of sensor detections}}$$

Dataset	Normal Weather	Extreme Weather	Expected Sensor Period
Target Position	7202	7202	-
Radar	3601	3601	2
ESM	360	360	20
EO	360	360	10
IR	720	720	10

Presence of data with TimeDelta being 0 days 00:00:00 This suggests that these data are replicated within the dataset,m hence the TimeDelta is different. Modification made to import script is to remove rows with TimeDelta = 0.

2. Detection Success / Failure Rate across Each Scenario

```
def scn_detect_proportion(df):
  avg = []
  for _, _df in df.groupby('TimelineID'):
    val = _df.DetectionResult.value_counts()
    success = val['SUCCESS']
    total = len(df)
    success_rate = success / total * 100
    avg.append(success_rate)
  return avg, np.mean(avg), np.std(avg)


for sensor in SENSORS:
  _, mu, std = scn_detect_proportion(dataset['normal'][sensor])
  print(f'{sensor}: {mu:.4e} +/- {std:.4e}')
```

```
IR: 1.0000e+00 +/- 0.0000e+00
EO: 1.0000e+00 +/- 0.0000e+00
Radar: 1.0000e+00 +/- 0.0000e+00
ESM: 1.0000e+00 +/- 0.0000e+00
```

```
for sensor in SENSORS:
  _, mu, std = scn_detect_proportion(dataset['extreme'][sensor])
  print(f'{sensor}: {mu:.4f} +/- {std:.4f}')
```

```
IR: 0.5362 +/- 0.0386
EO: 1.0000 +/- 0.0000
Radar: 1.0000 +/- 0.0001
ESM: 1.0000 +/- 0.0000
```

Conclusion:

- Detection Rate for NORMAL weather is 100%,
- while that for EXTREME weather is about 50% for IR, while close to perfect for Radar. EO and ESM is unaffected by extreme weather simulated
- Individual Sensor Error with respect to Ground Truth

```
weather = 'normal'
df_pos = dataset[weather]['pos']
for sensor in SENSORS:
  _df_sensor = dataset[weather][sensor].copy(deep=True)
  _df_pos = df_pos.copy(deep=True)
  df_merged = pd.merge(_df_sensor, _df_pos, left_on=['TimelineID', 'Time'],
right_on=['TimelineID', 'Time'], how='left')
  df_merged['Err_ENU_x'] = df_merged.apply(lambda r: sq_err_1d(r.Target_E, r.Unit_E), axis=1)
  df_merged['Err_ENU_y'] = df_merged.apply(lambda r: sq_err_1d(r.Target_N, r.Unit_N), axis=1)
  df_merged['Err_ENU_z'] = df_merged.apply(lambda r: sq_err_1d(r.Target_U, r.Unit_U), axis=1)
  df_merged['Err_ENU_2d'] = df_merged.apply(lambda r: l2_norm_2d(r.Target_E, r.Target_N, r.Unit_E,
r.Unit_N), axis=1)

  del _df_pos, _df_sensor  # memory management
  errs = []
  for _, _df in df_merged.groupby('TimelineID'):
    errs.append(_df.Err_ENU_2d.mean())

  # save file for future reference:
  if SAVE_DF:
    df_merged.to_feather(f'../data/df_merged_normal_pos_{sensor}.ftr')

  # sensor-aggregated performance
  mu = np.mean(errs)
  std = np.std(errs)
  print(f'{sensor}: {mu:.4e} +/- {std:.4e}')
```

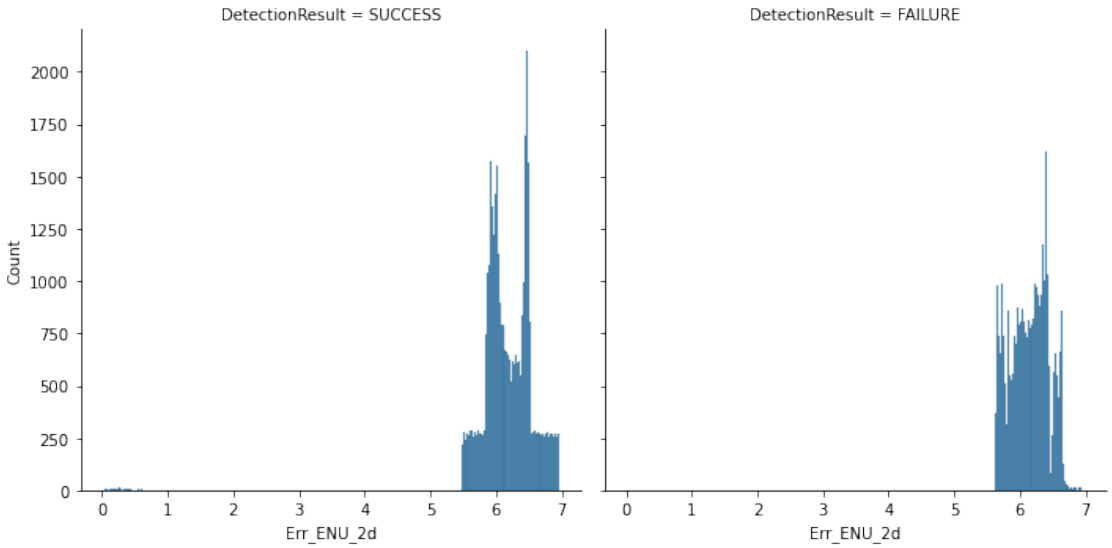
```
IR: 6.1656e+00 +/- 1.8957e-02
EO: 6.1653e+00 +/- 1.8502e-02
Radar: 6.1725e+00 +/- 1.8650e-02
ESM: 4.2150e-01 +/- 1.4163e-02
```

```
dataset['extreme']['IR'].DetectionResult.value_counts()
```

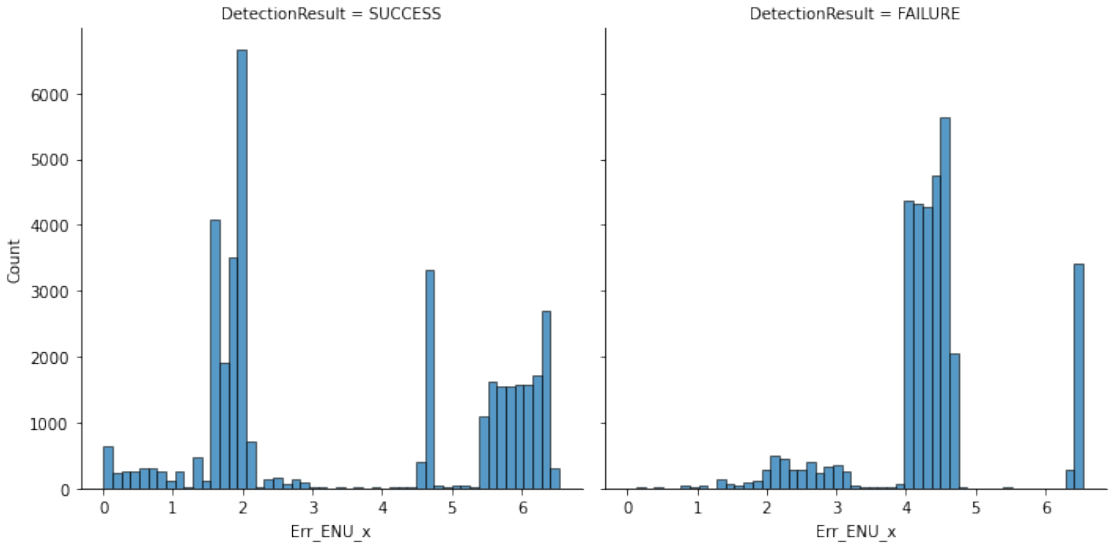
```
SUCCESS    38604
FAILURE    33396
Name: DetectionResult, dtype: int64

weather = 'extreme'
df_pos = dataset[weather]['pos']
for sensor in SENSORS:
  _df_sensor = dataset[weather][sensor].copy(deep=True)
  _df_pos = df_pos.copy(deep=True)
  df_merged = pd.merge(_df_sensor, _df_pos, left_on=['TimelineID', 'Time'],
right_on=['TimelineID', 'Time'], how='left')
  df_merged['Err_ENU_x'] = df_merged.apply(lambda r: sq_err_1d(r.Target_E, r.Unit_E), axis=1)
  df_merged['Err_ENU_y'] = df_merged.apply(lambda r: sq_err_1d(r.Target_N, r.Unit_N), axis=1)
  df_merged['Err_ENU_z'] = df_merged.apply(lambda r: sq_err_1d(r.Target_U, r.Unit_U), axis=1)
  df_merged['Err_ENU_2d'] = df_merged.apply(lambda r: l2_norm_2d(r.Target_E, r.Target_N, r.Unit_E,
r.Unit_N), axis=1)
  df_merged.to_feather(f'../data/df_merged_{weather}_pos_{sensor}.ftr')

  del _df_pos, _df_sensor  # memory management
  errs = []
  for _, _df in df_merged.groupby('TimelineID'):
    errs.append(_df.Err_ENU_2d.mean())

  # save file for future reference:
  if SAVE_DF:
    df_merged.to_feather(f'../data/df_merged_extreme_pos_{sensor}.ftr')

  # sensor-aggregated performance
  mu = np.mean(errs)
  std = np.std(errs)
  print(f'{sensor}: {mu:.4e} +/- {std:.4e}')

IR: 6.1637e+00 +/- 1.8438e-02
EO: 6.1640e+00 +/- 1.8425e-02
Radar: 6.1710e+00 +/- 1.9342e-02
ESM: 4.2200e-01 +/- 1.4134e-02
```

(1) Are there significant differences in DetectionResult = FAILURE vs SUCCESSS for IR in adverse weather?

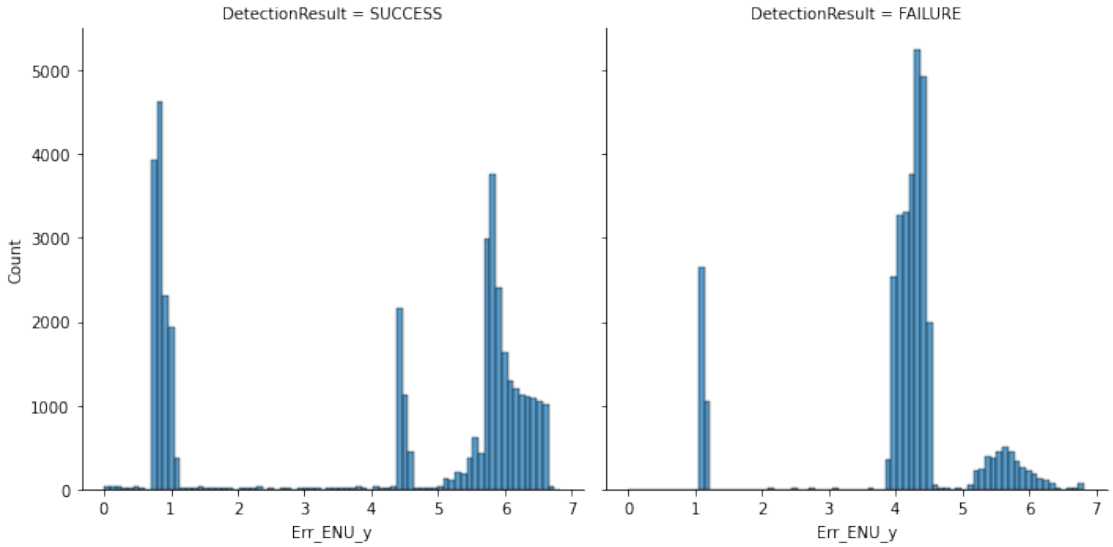
```
df_merged = pd.read_feather('../data/df_merged_extreme_pos_IR.ftr')

# global statistics
sns.displot(df_merged, x='Err_ENU_2d',col='DetectionResult')
```



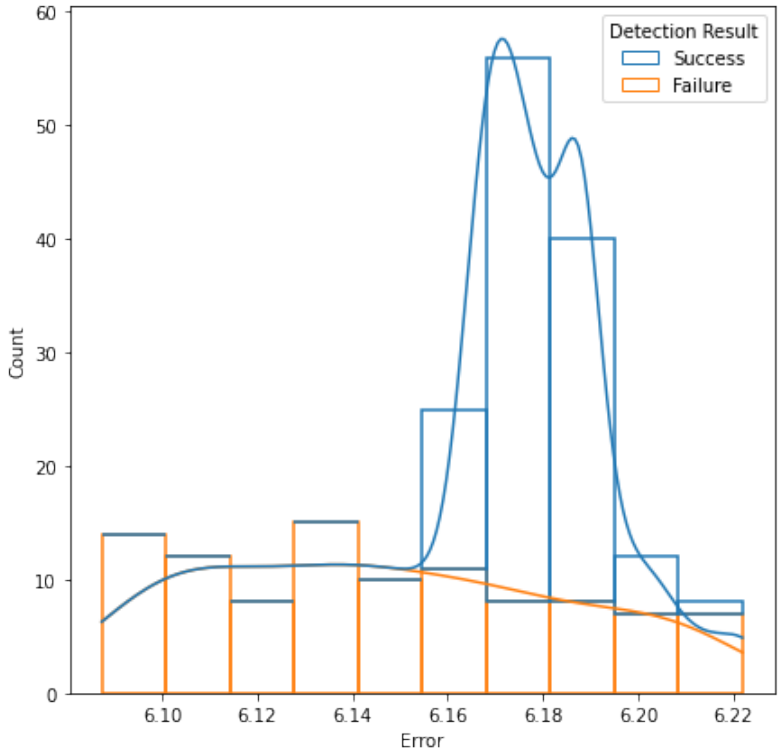

```
sns.displot(df_merged, x='Err_ENU_x',col='DetectionResult')
```



```
sns.displot(df_merged, x='Err_ENU_y',col='DetectionResult')
```




```
# Average across all the scenarios, instead of global statistics.
success_errs = []
failure_errs = []
for _, _df_merged in df_merged.groupby('TimelineID'):
  success = _df_merged[_df_merged.DetectionResult == 'SUCCESS']
  failure = _df_merged[_df_merged.DetectionResult == 'FAILURE']
  success_errs.append(success.Err_ENU_2d.mean())
  failure_errs.append(failure.Err_ENU_2d.mean())
print(f'{sensor}, DetectionResult=Success: {np.mean(success_errs):.4e} +/-
{np.std(success_errs):.4e}')
print(f'{sensor}, DetectionResult=Failure: {np.mean(failure_errs):.4e} +/-
{np.std(failure_errs):.4e}')

ESM, DetectionResult=Success: 6.1788e+00 +/- 1.0368e-02
ESM, DetectionResult=Failure: 6.1471e+00 +/- 3.7479e-02

sx = pd.DataFrame({'Err':success_errs,'Detection Result':'Success'})
fx = pd.DataFrame({'Err':failure_errs, 'Detection Result':'Failure'})
out = pd.concat([sx,fx],ignore_index=True)

f, ax = plt.subplots(1,1,figsize=(7,7))
sns.histplot(out, x='Err',hue='Detection Result', ax=ax,kde=True, fill=False, multiple='stack')
ax.set_xlabel('Error')
```



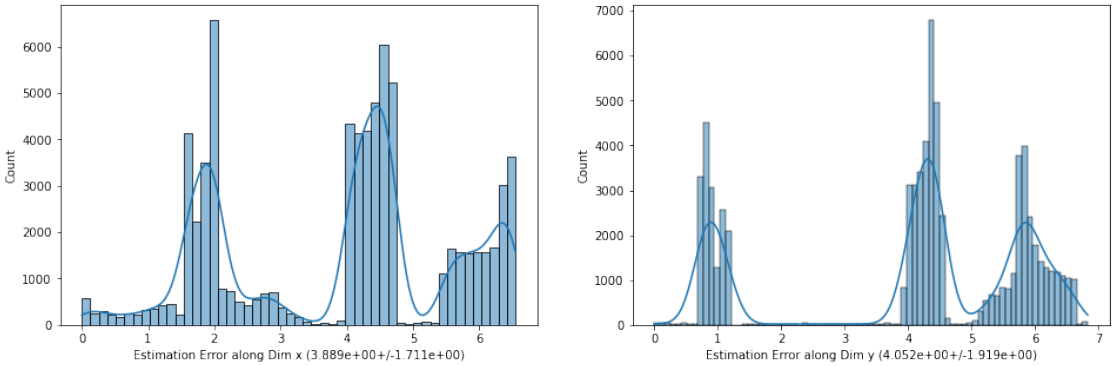
E. CALCULATE THE SENSOR ERRORS

The purpose of calculating each sensor's error is to provide KF with uncertainty of measurement, so that KF will be able to update its probabilistic belief of the target's state. There is an error for each dimension of the state measured - i.e., lon (x), lat (y). Thus, we represent the sensor's uncertainty as an average across all measurements by the sensor vis-a-vis the ground truth dataset.

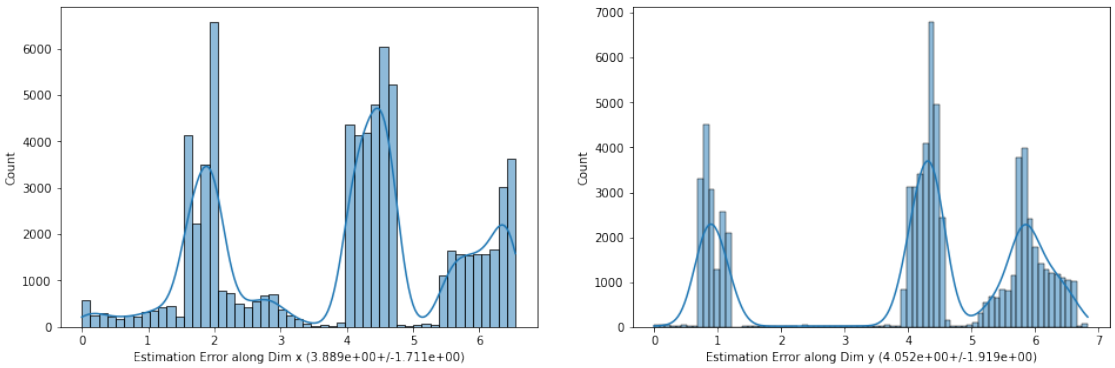
```
weather = 'normal'
stats = {}
for sensor in SENSORS:
    df_merged = pd.read_feather(f'../data/df_merged_{weather}_pos_{sensor}.ftr')
    f, axes = plt.subplots(1, 2, figsize=(16, 5))
    plt.suptitle(f'{sensor}')
    mu_x = df_merged.Err_ENU_x.mean()
    std_x = df_merged.Err_ENU_x.std()
    mu_y = df_merged.Err_ENU_y.mean()
    std_y = df_merged.Err_ENU_y.std()

    for i, t in enumerate(['x', 'y']):
        ax =axes[i]
        sns.histplot(data=df_merged, x=f'Err_ENU_{t}', kde=True, ax=ax)
        if t=='x':
            ax.set_xlabel(f'Estimation Error along Dim {t} ({mu_x:.3e}+/-{std_x:.3e})')
        else:
```

```
        ax.set_xlabel(f'Estimation Error along Dim {t} ({mu_y:.3e}+/-{std_y:.3e})')
stats.update({sensor:{'mu_x':mu_x, 'std_x':std_x,'mu_y':mu_y, 'std_y':std_y}})
```

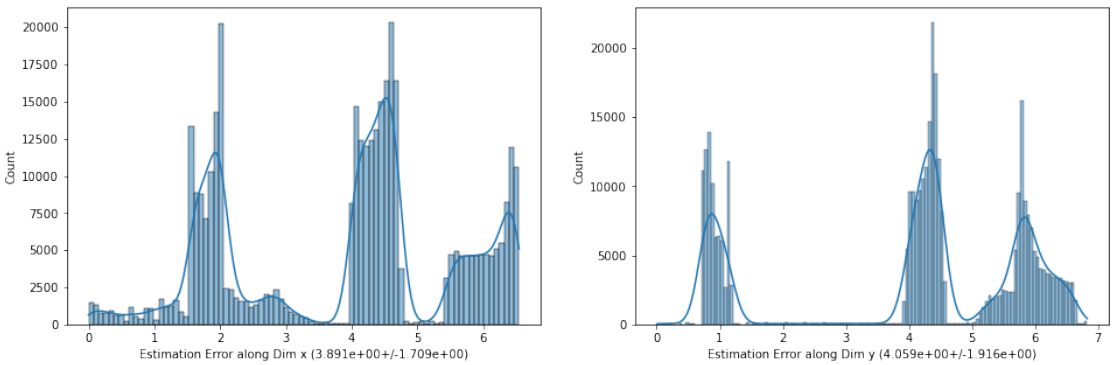
IR



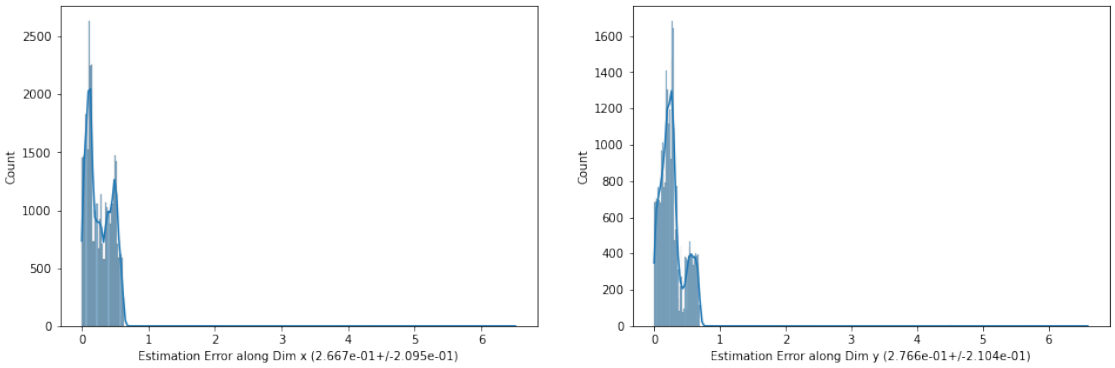
EO



Radar



ESM




```
dat = pd.DataFrame(stats)
dat.to_pickle('../data/sensor_err_stats.pkl')

dat
```

	IR	EO	Radar	ESM
mu_x	3.889024	3.889116	3.890564	0.266667
std_x	1.711266	1.711265	1.709136	0.209541
mu_y	4.052315	4.051945	4.058989	0.276587
std_y	1.919304	1.919407	1.915697	0.210433

APPENDIX E. PYTHON NOTEBOOK — BASELINE MODEL WITH KF

The objective of this notebook is to implement Kalman filter as a predictor-corrector estimator for position prediction.

A. PARAMETERS IN THE KALMAN FILTER ALGORITHM

Outline of the KF algorithm:

1. Initialization

- Initialize the state of the filter
- Initialize our probabilistic belief in the state

2. Predict Step

- Use process model to predict state at the next time step
- Adjust probabilistic belief to account for the uncertainty in prediction

3. Update Step

- Get a measurement and associated probabilistic belief about its accuracy
- Compute residual between estimated state and measurement
- Compute scaling factor based on whether the measurement or prediction is more accurate
- Set state between the prediction and measurement based on scaling factor
- Update model's probabilistic belief of the estimated state, using the measurement noise covariance matrix

```
import os
import pandas as pd
import numpy as np
```

```
# import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
sns.set_context('notebook', font_scale = 1)
from pprint import pprint
from helper import *
# Kalman Filter Package
from filterpy.Kalman import KalmanFilter
from filterpy.common import Saver
from filterpy.common import Q_discrete_white_noise

### 1. Import dataset
data = load_all_dfs()
data_merged=load_merged_dfs(weather='normal')

data_merged['EO'].columns

Index(['TimelineID', 'Time', 'SensorID', 'SensorName', 'SensorParentLongitude',
       'SensorParentLatitude', 'SensorParentAltitude_AGL', 'TargetID',
       'TargetName', 'TargetLongitude', 'TargetLatitude',
       'TargetAltitude_AGL_m', 'TargetRangeSlant_nm', 'DetectionResult',
       'DetectionAOU', 'Target_E', 'Target_N', 'Target_U', 'Sensor_E',
       'Sensor_N', 'Sensor_U', 'TimeDelta_x', 'UnitID', 'UnitName', 'UnitType',
       'UnitClass', 'UnitLongitude', 'UnitLatitude', 'UnitCourse',
       'UnitSpeed_kts', 'UnitAltitude_m', 'Unit_E', 'Unit_N', 'Unit_U',
       'TimeDelta_y', 'Err_ENU_x', 'Err_ENU_y', 'Err_ENU_z', 'Err_ENU_2d'],
      dtype='object')
```

B. SET UP KALMAN FILTER FUNCTIONS

This section illustrates how a KF is set up and the internal workings.

The state variable would be $[x, y, dx, dy, d^2x, d^2y]$ where:

- Velocity: dx, dy are the rate of change for x, y and
- Acceleration: d^2x, d^2y are the rate of change for dx, dy
- x = longitude,
- y = latitude

```
# Number of state parameters
dim_x = 6
# size of measurement vector = each sensor provides x and y == 2
dim_z = 2

tracker = KalmanFilter(dim_x = dim_x, dim_z=dim_z)
dt = 1. # we set the timestep = 1 because that is the target position update periodicity.
```

a. Define the State Transition Matrix

(1) Constant Acceleration Model

```
F_accl = np.array([[1, 0, dt, 0, (dt**2) / 2, 0], [0, 1, 0, dt, 0, (dt**2) / 2], [0, 0, 1, 0, dt,
0], [0, 0, 0, 0, 0, dt], [0, 0, 0, 0, 1, 0],
                   [0, 0, 0, 0, 0, 1]])
```

(2) Constant Velocity Model

```
F_vel = np.array([[1, 0, dt, 0, 0, 0], [0, 1, 0, dt, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]])
```

b. Define the Process Noise Matrix

- Assume that the process is a discrete Wiener Process
- This assumes independence between lat, lon (N and E respectively) (the dimensions of each state variables)

```
X=[x y dx dy d2x d2y]

Q = Cov(X)
```

Hence, $Q_{11} = \sigma_x$ and etc

```
q = Q_discrete_white_noise(dim=3, dt=dt, var=1, block_size=2, order_by_dim=False)
tracker.Q = q
print(tracker.Q)
```

```
[[0.25 0.   0.5  0.   0.5  0.  ]
 [0.   0.25 0.   0.5  0.   0.5 ]
 [0.5  0.   1.   0.   1.   0.  ]
 [0.   0.5  0.   1.   0.   1.  ]
 [0.5  0.   1.   0.   1.   0.  ]
 [0.   0.5  0.   1.   0.   1.  ]]
```

c. Define the Measurement Function

```
tracker.H = np.array([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]])
print(tracker.H)
```

```
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]]
```

```
print(tracker.H.shape)
```

```
(2, 6)
```

d. Define the Measurement Noise Matrix

- Assume that lat, lon, lat are independent White Gaussian Process ~ $N(0,5)$ - i.e., a measurement gaussian noise of $5m^2$

- Measurement noise per sensor per dimension of measurement.

```
# or assume that the noise is uniform and constant for all sensors.
q = np.eye(tracker.dim_z, tracker.dim_z)
# assume that the sensor noise is 1e-6...
sensor_noise = 1e-6
tracker.R = q  * sensor_noise # replicate onces for each of the sensors.
```

e. Define Initial Conditions

```
# Define the initial condition
# std for location E, N: 2800.117175     2880.817604
P0 = [2800**2, 2800**2, 1, 1, .5, .5]
P0 = np.eye(6, 6) @ P0

print(P0)

[7.84e+06 7.84e+06 1.00e+00 1.00e+00 5.00e-01 5.00e-01]

# assume that we do not know the initial
x0 = np.zeros([6,1])

# Initialise tracker with our belief and initial uncertainity
tracker.P = P0
tracker.x = x0
```

C. CREATING AN INTERFACE WITH DATASET

The dataset has the following properties that require our attention

1. The sensor data does not arrive at regular timestep; each sensor has its own periodicity
2. Lack of covariances in the measurement function.

```
class SensorDataWrapper:
  """
  Define each sensor as a sensor class itself, and provides the dataset when
  called by the other function.
  """


  def __init__(self, dataset_dict, sensor, weather='normal'):
    self.data = dataset_dict[weather][sensor]
    self.timelineID = None
    self.time = 0.  # maintains a clock within itself to provide error-checking.
    self._data = None
    self.num_entries = None
    self.start_time = self.data.Time.min()
    self.end_time = self.data.Time.max()
    self.sensor = sensor  # name of sensor


  def update_periodicity(self, period):
    self.periodicity = period


  def set_timelineID(self, timelineID):
    self.timelineID = timelineID
```

```
    self.set_time(0)  # reset the clock, since we are interested in the new timeline now
    self._data = self.data[self.data.TimelineID == timelineID]
    self._data.set_index('Time', inplace=True)
    self.num_entries = len(self._data)

  def set_time(self, time):
    self.time = time

  def get_next_detection(self):
    """
    this is the main interface with various functions.
    After calling the set_timelineID, get_next_detection would return parameters of interests
    returns (t,x,y) where t= time of detection, x=latitude, y=longitude
    """
    try:
      x = self._data.loc[self.time]['Target_E']
      y = self._data.loc[self.time]['Target_N']
      return (self.time, x, y)
    except KeyError:
      print(f'{self.time} is not in the index from data')

  def tick(self):
    # Advance the clock
    # if there are no more sensor data, return False
    self.time += 1.
    return self.time <= self.end_time and self.check_alert()

  def check_alert(self):
    ### alert when there is a detection.
    return self.time in self._data.index
```

-

```
def get_sensor_stats():
  sensor_err_stats = pd.read_pickle('../data/sensor_err_stats.pkl')
  mat_R = {}
  for sensor in SENSORS:
    mat_R[sensor] = np.array([[sensor_err_stats.loc['std_x', sensor]**2, 0.], [0.,
sensor_err_stats.loc['std_y', sensor]**2]])
  return mat_R
```

-

```
class KFWrapper:
  """
  Creates a Kalman Filter Wrapper around KalmanFilter from FilterPy to interface with the dataset
  """

  def __init__(self, F_type='acc', dim_x=6, dim_z=2, var_Q=10., sensor_err_stats=None):
    """
    F_type  :   The type of systems dynamic model. Either 'acc' or 'vel' for constant acceleration
or constant velocity model
    dim_x   :   sizer of state parameters
    dim_z   :   size of measurement space per sensor
    """
    dim_x = dim_x
    dim_z = dim_z
    # Set up the KF:
    self.kf = KalmanFilter(dim_x=dim_x, dim_z=dim_z)
    self.dt = 1.
    self.init = False
    self.saver = Saver(self.kf)

    # Process model
```

```
    assert F_type in ['acc', 'vel']
    if F_type == 'acc':
      self.kf.F = np.array([[1, 0, self.dt, 0, (self.dt**2) / 2, 0], [0, 1, 0, self.dt, 0,
(self.dt**2) / 2], [0, 0, 1, 0, self.dt, 0],
                            [0, 0, 0, 0, 0, self.dt], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]])
    elif F_type == 'vel':
      self.kf.F = np.array([[1, 0, self.dt, 0, 0, 0], [0, 1, 0, self.dt, 0, 0], [0, 0, 1, 0, 0,
0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0]])

    # process noise
    # High variance in process ==> KF relies on measurement more.
    self.kf.Q = Q_discrete_white_noise(dim=3, dt=self.dt, var=var_Q, block_size=2,
order_by_dim=False)
    # print(self.kf.Q)

    # measurement function
    # we only have 1 set of x, y measurement from the sensor
    self.kf.H = np.array([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]])

    # measurement noise
    # the measurements is a 2x1 matrix z
    # hence the measurement noise would be 2x2
    # assume each measurement of x and y are independent and constant noise
    # sensor_noise = 1e-3
    # self.kf.R = np.eye(dim_z, dim_z) * sensor_noise

  def set_init_process_noise(self, P0):
    self.kf.P = P0

  def update_observation(self, sensor, z):
    if not self.init:
      self.kf.x[0] = z[0]
      self.kf.x[1] = z[1]
      self.init = True
      print(f'init kf with position {z}')
    else:
      self.kf.R = sensor_err_stats[sensor]
      # print(self.kf.R)
      self.kf.update(z)
    # self.saver.save() # save the state of x after updating with observation.

  def get_prediction(self):
    self.kf.predict()
    self.saver.save()
```

```
# use the mu, std from population statistics...
sensor_err_stats = pd.read_pickle('../data/sensor_err_stats.pkl')
print(sensor_err_stats)
# R is an 2x2 matrix, there will be one matrix for each sensor, based on the std in
sensor_err_stats
# power 2 because cov is sigma^2
def get_sensor_stats():
  sensor_err_stats = pd.read_pickle('../data/sensor_err_stats.pkl')
  mat_R = {}
  for sensor in SENSORS:
    mat_R[sensor] = np.array([[sensor_err_stats.loc['std_x', sensor]**2, 0.], [0.,
sensor_err_stats.loc['std_y', sensor]**2]])
  return mat_R
get_sensor_stats()
```

```
             IR        EO     Radar       ESM
mu_x   3.889024  3.889116  3.890564  0.266667
std_x  1.711266  1.711265  1.709136  0.209541
mu_y   4.052315  4.051945  4.058989  0.276587
std_y  1.919304  1.919407  1.915697  0.210433
```

```
{'IR': array([[2.92843122, 0.        ],
        [0.        , 3.68372751]]),
 'EO': array([[2.92842714, 0.        ],
        [0.        , 3.68412198]]),
 'Radar': array([[2.92114647, 0.        ],
        [0.        , 3.66989405]]),
 'ESM': array([[0.04390724, 0.        ],
        [0.        , 0.04428191]])}

def setup_run(timelineID, data, weather):
  sensor_dat = {}
  for sensor in SENSORS:
    sensor_dat[sensor] = SensorDataWrapper(data, sensor, weather=weather)
    sensor_dat[sensor].set_timelineID(timelineID)
    print(
        f'sensor: {sensor}\tstart: {sensor_dat[sensor].start_time}\t#entries:
{sensor_dat[sensor].num_entries}'
    )
  endtime = data['normal']['pos'].Time.max()
  return sensor_dat, endtime
```

D. RUNNING A SIMULATION WITH KF

```
def run(timelineID, kf, data, weather='normal'):
  """
  Run the Kalman filter on the sensor dataset
  """
  print(f"filtering timelineID: {timelineID}")
  time = 0.
  sensor_dat, endtime = setup_run(timelineID, data, weather=weather)

  while time < endtime:
    time += 1
    # Predict Step:
    # estimate the location of the target, only update the prediction
    # iff there are measurements from the sensors.
    if time > 1:
      kf.get_prediction()
    # update clock of sensors_dat
    # get the detection if the sensor has a detection at this timestep
    # accumulate all the measurements from the sensors
    for sensor in SENSORS:
      if sensor_dat[sensor].tick():
        (t, x, y) = sensor_dat[sensor].get_next_detection()
        assert t == time, f"alert is out of sync (got t={t}, but time is {time})"
        # print(f'{sensor}:{t}')
        kf.update_observation(sensor, np.array([[x, y]]).T)

  return kf

sensor_err_stats = get_sensor_stats()
kf = KFWrapper(sensor_err_stats=sensor_err_stats)
# Define the initial condition
P0 = [65**2, 65**2, .5**2, .5**2, .5**4, .5**4]
P0 = np.eye(6, 6) * P0
kf.set_init_process_noise(P0)
# sample for a timeline ID.
timelineIDs = data['normal']['ESM'].TimelineID.unique()
timelineID = timelineIDs[5] # running on normal weather sample 5
kf = run(timelineID, kf=kf, data=data, weather='normal')

filtering timelineID: 67362da0-b4b9-458e-b14d-e76d72dbbc9d
sensor: IR      start: 3#entries: 720
sensor: EO      start: 3#entries: 720
sensor: Radar   start: 1#entries: 3601
sensor: ESM     start: 14       #entries: 360
```

```
init kf with position [[1774.92667137]
 [-720.46927762]]
```

-

(1) Calculate Estimation Error by KF Algorithm

```
def get_metrics(kf, df_pos, timelineID):
  pos_extract = np.array([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]]).T
  df_pos = df_pos[df_pos.TimelineID == timelineID]
  df_pos = df_pos.loc[:, ['Time', 'Unit_E', 'Unit_N']]
  df_pos.drop(df_pos.index[0], axis=0, inplace=True)
  df_pos.reset_index(inplace=True, drop=True)
  kf_latlon = np.array(kf.saver.x).squeeze() @ pos_extract
  df_kf = pd.DataFrame(kf_latlon, columns=[
      'KF_E',
      'KF_N',
  ])
  df_merged = pd.concat([df_pos, df_kf], axis=1)
  df_merged['residual_E'] = df_merged.apply(lambda r: sq_err_1d(r.KF_E, r.Unit_E), axis=1)
  df_merged['residual_N'] = df_merged.apply(lambda r: sq_err_1d(r.KF_N, r.Unit_N), axis=1)
  df_merged['residual_2d'] = df_merged.apply(lambda r: sq_err_2d(r.KF_N, r.Unit_N, r.KF_E,
r.Unit_E), axis=1)
  return kf_latlon, df_pos, df_merged

df_pos = data['normal']['pos']
df_latlon, df_pos, df_merged = get_metrics(kf, df_pos, timelineID)
```

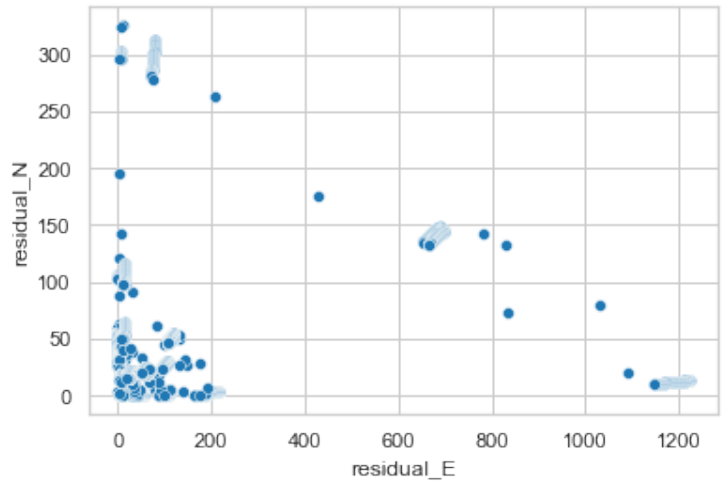
-

(2) Visualization

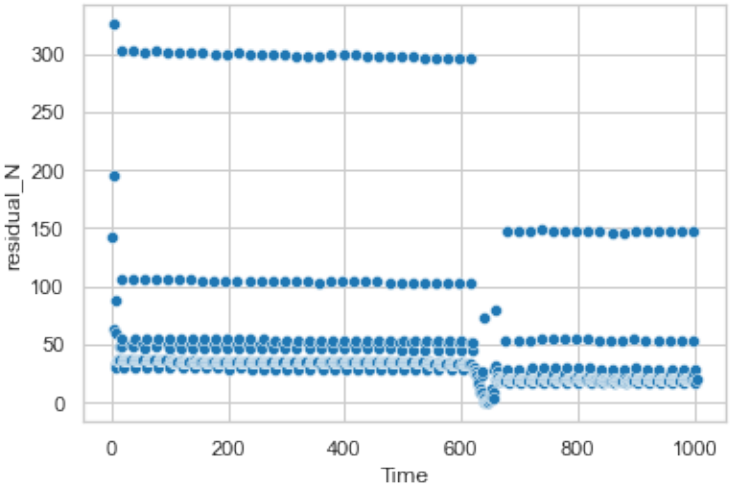
```
df_merged.describe()
```

```
              Time       Unit_E       Unit_N         KF_E         KF_N  \
count  7201.000000  7201.000000  7201.000000  7201.000000  7201.000000
mean   3602.000000  -121.291135    78.954700  -120.139041    78.649491
std    2078.893977  2924.502682  2859.777332  2924.188147  2860.033013
min       2.000000 -5813.182289 -4619.082017 -5816.758788 -4620.465758
25%    1802.000000 -2656.802653 -2435.575792 -2658.526479 -2435.807653
50%    3602.000000   329.895356   -98.149750   329.942908   -99.538963
75%    5402.000000  2280.884665  2820.640969  2279.818083  2820.919260
max    7202.000000  4361.794368  4849.184519  4362.413869  4850.974606

         residual_E   residual_N  residual_2d
count  7.201000e+03  7201.000000  7201.000000
mean   5.807884e+01    29.268166    87.347004
std    1.612977e+02    41.794770   174.577502
min    4.967165e-08     0.001089     0.987838
25%    4.107866e+00    14.620390    35.281986
50%    1.954108e+01    19.202310    39.954806
75%    3.984480e+01    35.008957    50.015390
max    1.221567e+03   326.047007  1235.578773
```

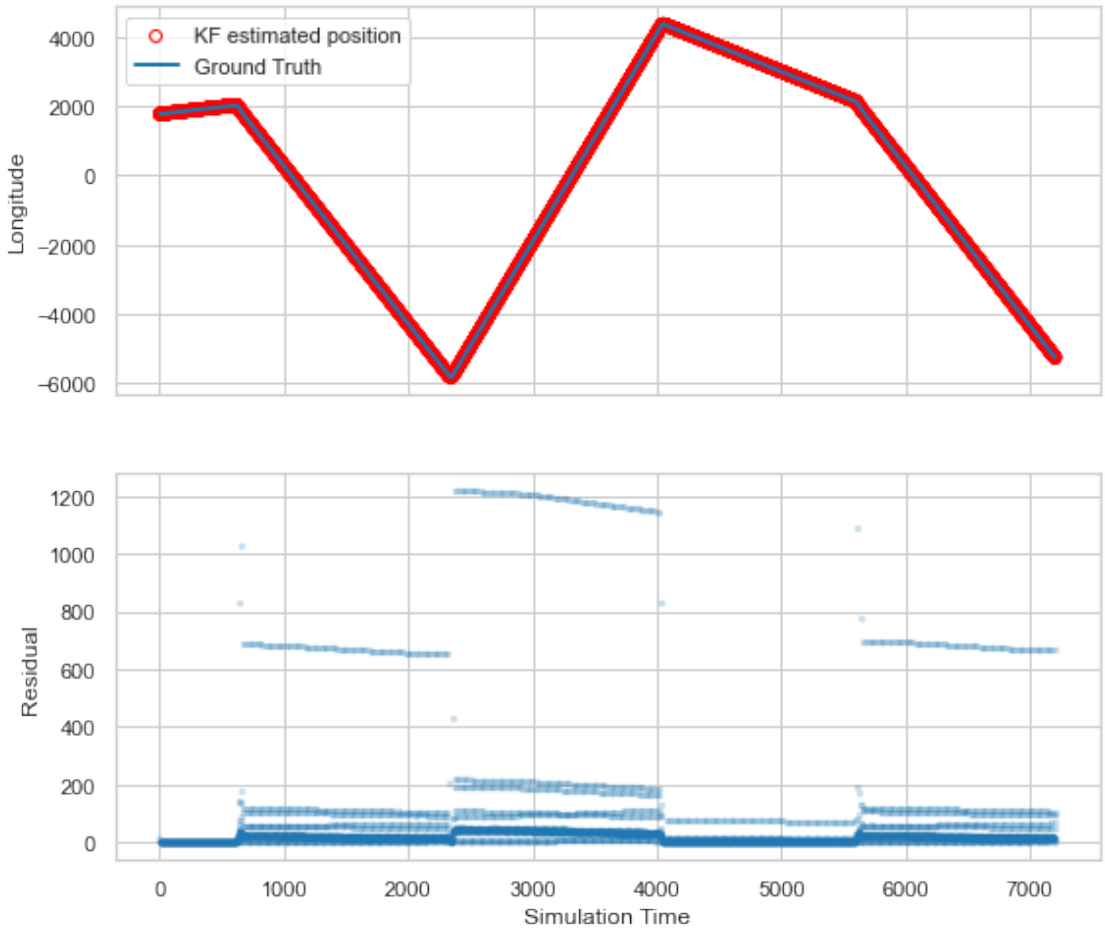
```
sns.scatterplot(df_merged.loc[:, 'residual_E'], df_merged.loc[:, 'residual_N'])
ax = plt.gca()
```



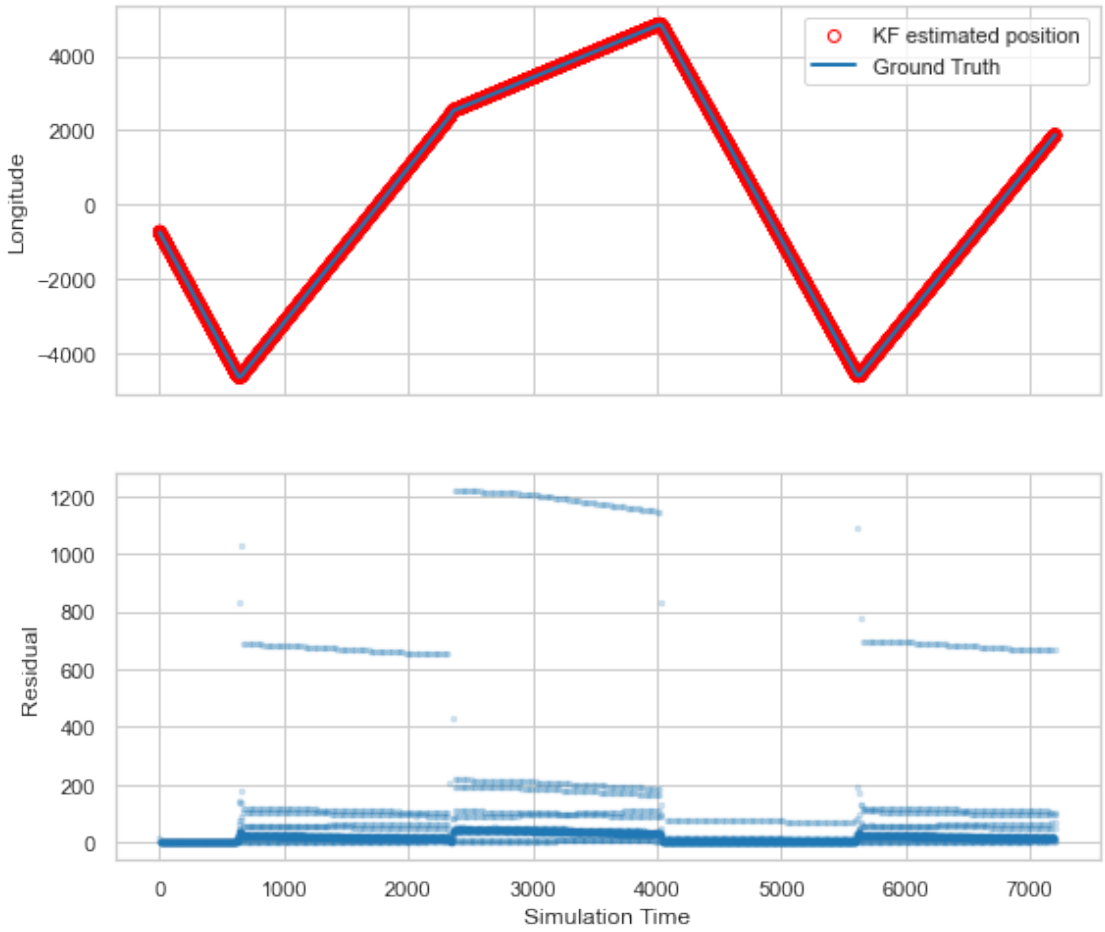

```
sns.scatterplot(df_merged.loc[0:1000, 'Time'], df_merged.loc[0:1000, 'residual_N'])
```



```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(9, 8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated position', facecolor='none',
edgecolors='r', marker='o', linewidths=1)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=2, label='Ground Truth')
ax1.set_ylabel('Longitude')
ax1.legend()
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, marker='.', linewidths=.5, alpha=0.2)
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('Residual')

Text(0, 0.5, 'Residual')
```

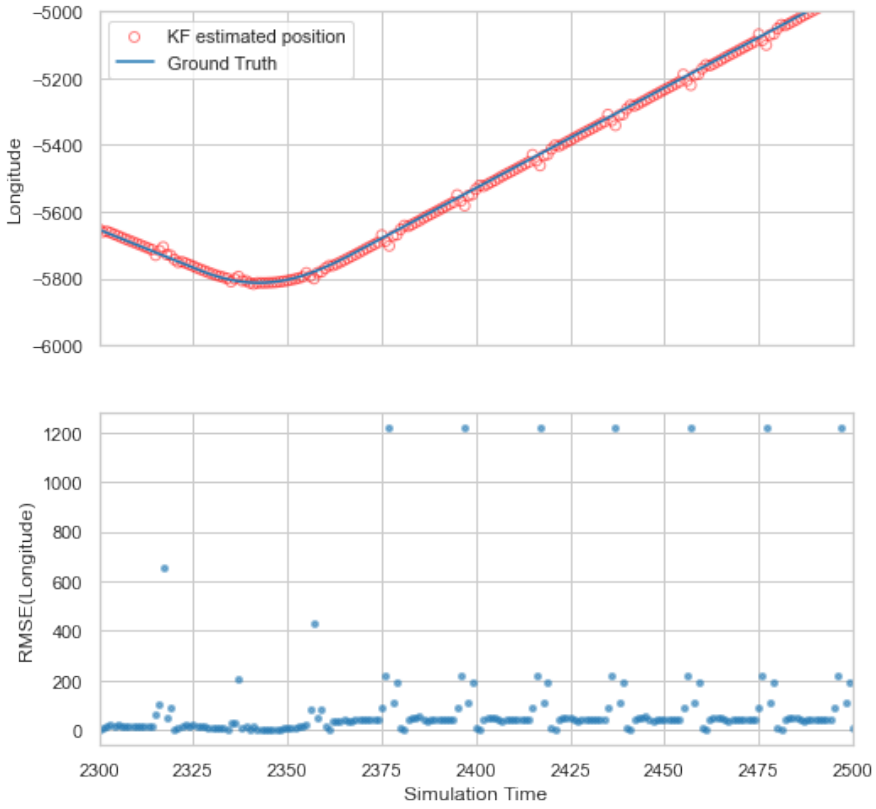


```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(9, 8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_N, label='KF estimated position', facecolor='none',
edgecolors='r', marker='o', linewidths=1)
ax1.plot(_df_merged.Time, _df_merged.Unit_N, linewidth=2, label='Ground Truth')
ax1.set_ylabel('Longitude')
ax1.legend()
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, marker='.', linewidths=.5, alpha=0.2)
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('Residual')

Text(0, 0.5, 'Residual')
```

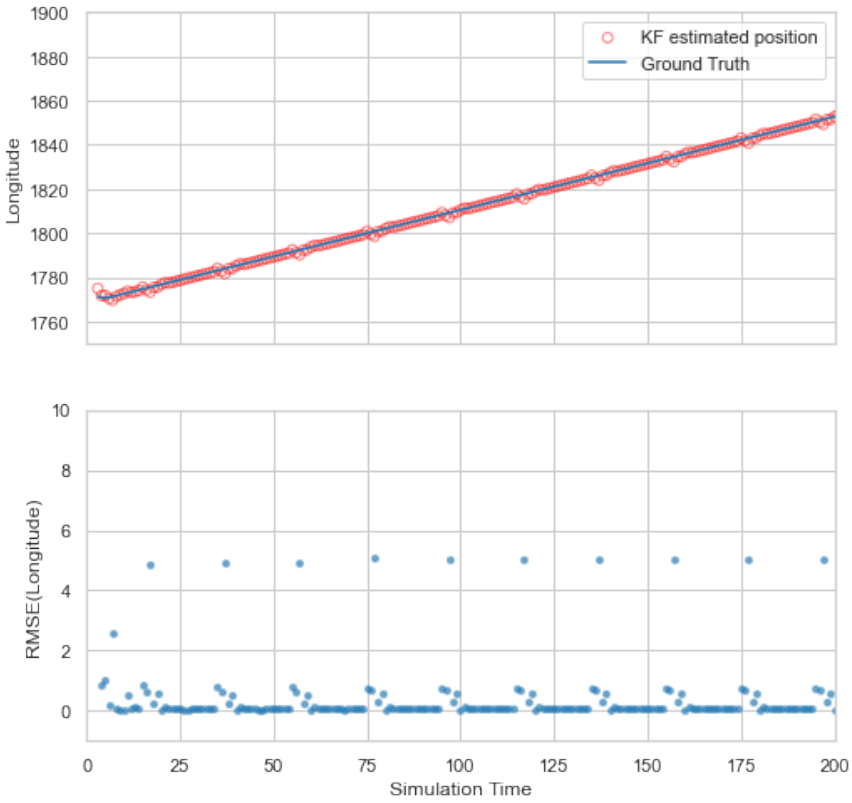



```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated position', facecolor='none',
edgecolors='r', marker='o', linewidths=1, alpha=.5)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=1.5, label='Ground Truth')
ax1.set_ylabel('Longitude')
ax1.legend()
```

```
ax1.set_xlim(2300, 2500)
ax1.set_ylim(-6000, -5000)
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, marker='.', alpha=.5, linewidths=2)
ax2.set_xlim(2300, 2500)
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('RMSE(Longitude)')
```

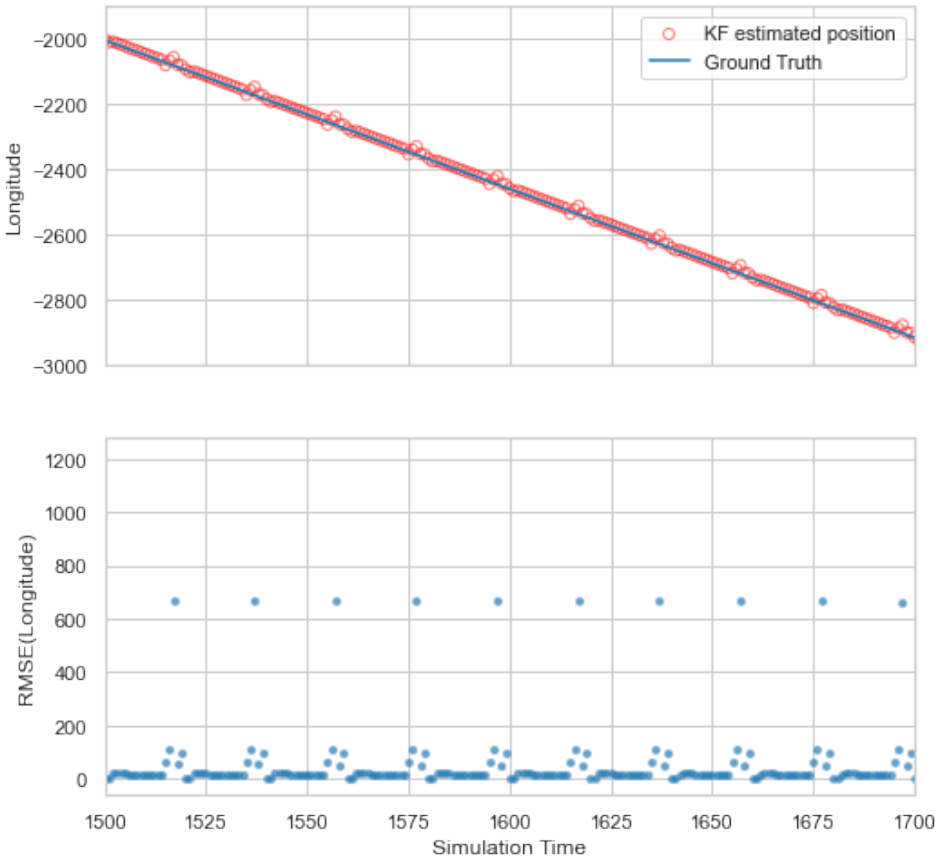


```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated position', facecolor='none',
edgecolors='r', marker='o', linewidths=1, alpha=.5)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=1.5, label='Ground Truth')
ax1.set_ylabel('Longitude')
ax1.legend()
ax1.set_xlim(0, 200)
ax1.set_ylim(1750, 1900)
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, marker='.', alpha=.5, linewidths=2)
ax2.set_xlim(0, 200)
ax2.set_ylim(-1, 10)
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('RMSE(Longitude)')
```

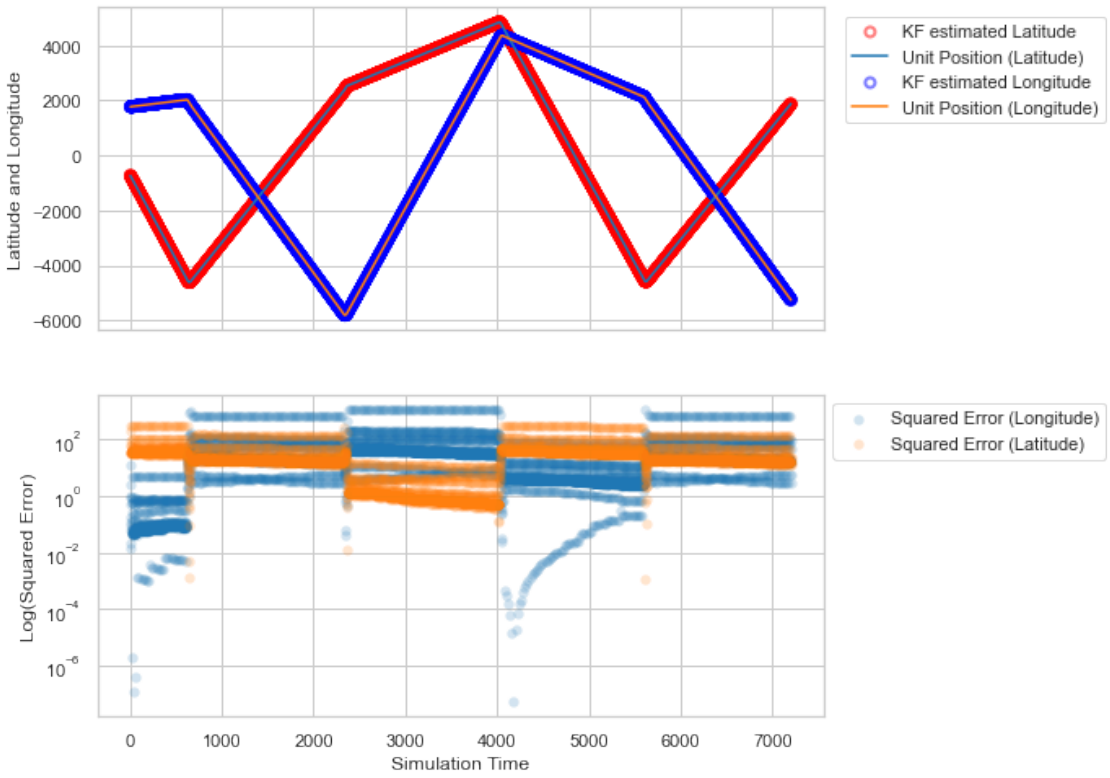



```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated position', facecolor='none',
edgecolors='r', marker='o', linewidths=1, alpha=.5)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=1.5, label='Ground Truth')
ax1.set_ylabel('Longitude')
ax1.legend()
ax1.set_xlim(1500, 1700)
ax1.set_ylim(-3000, -1900)
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, marker='.',alpha=.5, linewidths=2)
```

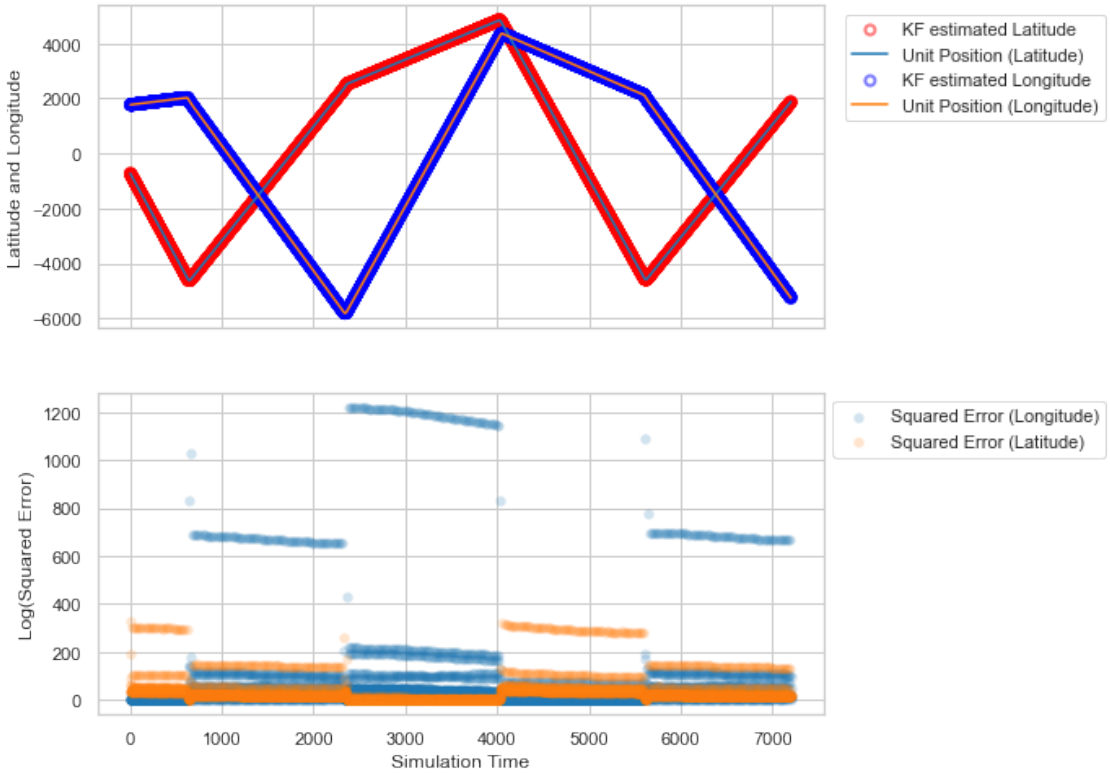
```
ax2.set_xlim(1500, 1700)
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('RMSE(Longitude)')
```




```
f, (ax1, ax2) = plt.subplots(2,1,figsize=(8,8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_N, label='KF estimated Latitude', facecolor='none',
edgecolors='r', marker='o', alpha=.5, linewidths=2)
ax1.plot(_df_merged.Time, _df_merged.Unit_N, linewidth=1.5, label='Unit Position (Latitude)')
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated Longitude', facecolor='none',
edgecolors='b', marker='o', alpha=.5, linewidths=2)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=1.5, label='Unit Position (Longitude)')
ax1.set_ylabel('Longitude and latitude')
ax1.legend(loc=1,bbox_to_anchor=(1.4,1))
ax2.scatter(_df_merged.Time, _df_merged.residual_E, linewidth=.1, label='Squared Error
(Longitude)',alpha=.2)
ax2.scatter(_df_merged.Time, _df_merged.residual_N, linewidth=.1, label='Squared Error
(Latitude)',alpha=.2)
ax2.legend(loc=1,bbox_to_anchor=(1.4,1))
ax2.set_yscale('log')
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('Log(Squared Error)')
```




```
f, (ax1, ax2) = plt.subplots(2,1,figsize=(8,8), sharex=True)

_df_merged = df_merged.iloc[1:]
# Longitude
ax1.scatter(_df_merged.Time, _df_merged.KF_N, label='KF estimated Latitude', facecolor='none',
edgecolors='r', marker='o', alpha=.5, linewidths=2)
ax1.plot(_df_merged.Time, _df_merged.Unit_N, linewidth=1.5, label='Unit Position (Latitude)')
ax1.scatter(_df_merged.Time, _df_merged.KF_E, label='KF estimated Longitude', facecolor='none',
edgecolors='b', marker='o', alpha=.5, linewidths=2)
ax1.plot(_df_merged.Time, _df_merged.Unit_E, linewidth=1.5, label='Unit Position (Longitude)')
ax1.set_ylabel('Longitude and latitude')
ax1.legend(loc=1,bbox_to_anchor=(1.4,1))
# Latitude
ax2.scatter(_df_merged.Time, _df_merged.residual_E, linewidth=.1, label='Squared Error
(Longitude)',alpha=.2)
ax2.scatter(_df_merged.Time, _df_merged.residual_N, linewidth=.1, label='Squared Error
(Latitude)',alpha=.2)
ax2.legend(loc=1,bbox_to_anchor=(1.4,1))
ax2.set_xlabel('Simulation Time')
ax2.set_ylabel('Log(Squared Error)')
```



E. EVALUATION OF RESULTS

```
# NORMAL DATASET
kf_runs = {}
sensor_err_stats = get_sensor_stats()
for timelineID, df_pos in data['normal']['pos'].groupby('TimelineID'):
  kf = KFWrapper(sensor_err_stats=sensor_err_stats)
  # Define the initial condition
  P0 = [65**2, 65**2, .5**2, .5**2, .5**4, .5**4]
  P0 = np.eye(6, 6) * P0
  kf.set_init_process_noise(P0)
  kf = run(timelineID, kf=kf, data=data, weather='normal')
  _, _, df_merged = get_metrics(kf, df_pos, timelineID)
  # save for subsequent processing
  df_merged.to_feather(f'./kf_eval_normal/{timelineID}.ftr')

# EXTREME DATASET
kf_runs = {}
sensor_err_stats = get_sensor_stats()
for timelineID, df_pos in data['extreme']['pos'].groupby('TimelineID'):
  kf = KFWrapper(sensor_err_stats=sensor_err_stats)
  # Define the initial condition
  P0 = [65**2, 65**2, .5**2, .5**2, .5**4, .5**4]
  P0 = np.eye(6, 6) * P0
  kf.set_init_process_noise(P0)
  kf = run(timelineID, kf=kf, data=data, weather='extreme')
  _, _, df_merged = get_metrics(kf, df_pos, timelineID)
  # save for subsequent processing
  df_merged.to_feather(f'./kf_eval_extreme/{timelineID}.ftr')
```

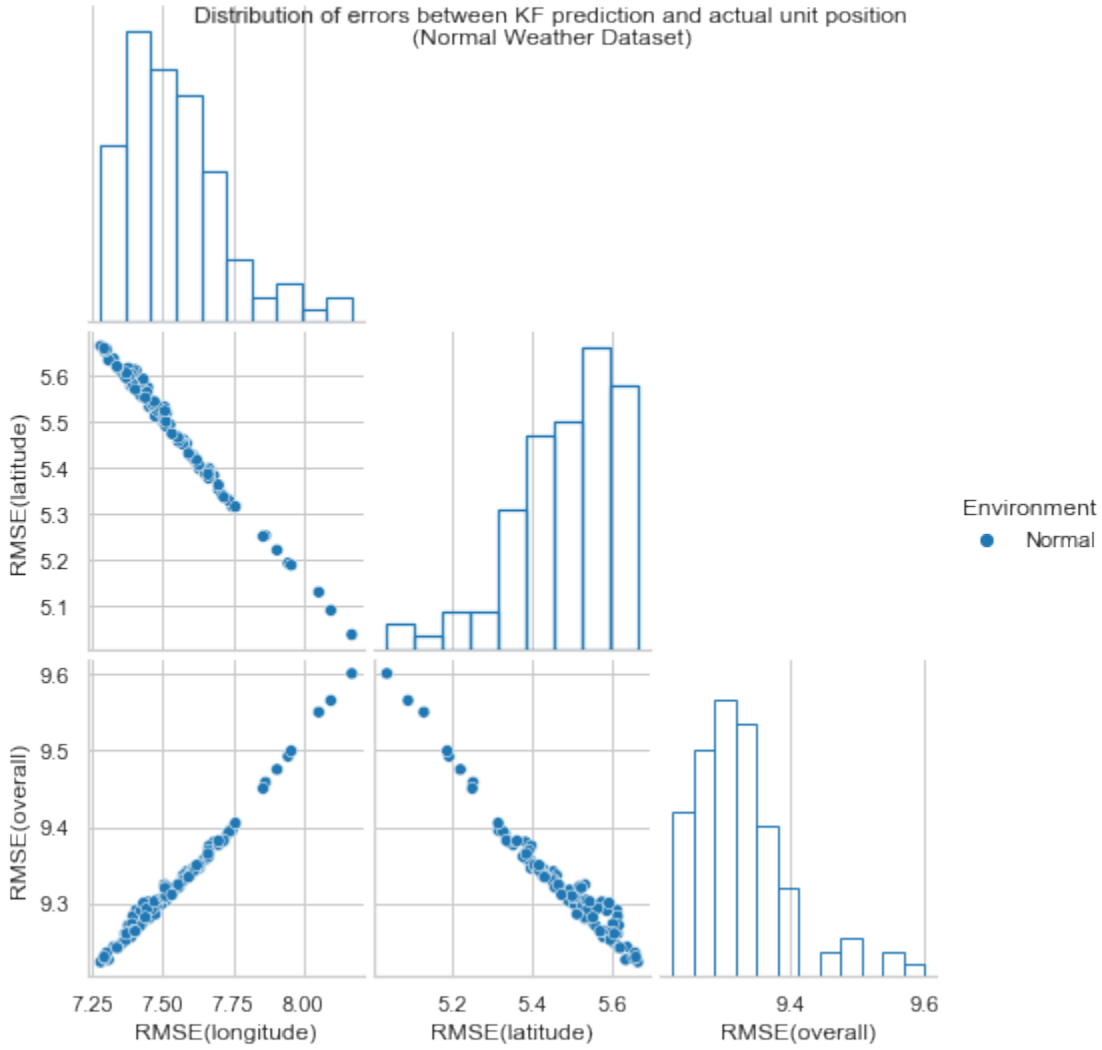
The following analysis is conducted for each scenario.

1. Calculate the average detection error (RMSE) for each simulation run,
2. Calculate the mean and standard deviation of detection error for KF estimation for each type of settings.

```
# Normal environment
dir = './kf_eval_normal/'
res_mu_normal_x = []
res_mu_normal_y = []
res_mu_normal = []
for outfile in os.listdir(dir):
  df = pd.read_feather(os.path.join(dir, outfile))
  res_mu_normal_x.append(np.sqrt(df.residual_E.mean())) # calculates the RMSE of the error
  res_mu_normal_y.append(np.sqrt(df.residual_N.mean()))
  res_mu_normal.append(np.sqrt(df.residual_2d.mean()))

df.describe()
```

```
              Time       Unit_E       Unit_N         KF_E         KF_N  \
count  7201.000000  7201.000000  7201.000000  7201.000000  7201.000000
mean   3602.000000   239.824163   141.206389   240.200543   141.893502
std    2078.893977  2658.804546  2899.506122  2658.940273  2900.052696
min       2.000000 -5816.487395 -4650.551169 -5817.020354 -4652.149989
25%    1802.000000 -1519.905938 -2458.942698 -1522.850643 -2459.945697
50%    3602.000000   686.380796    44.131313   686.463870    44.890885
75%    5402.000000  2282.164401  2893.976985  2282.316028  2895.304009
max    7202.000000  4365.368905  4850.039709  4367.081153  4851.811333

        residual_E   residual_N  residual_2d
count  7201.000000  7201.000000  7201.000000
mean     53.163313    32.026724    85.190037
std     155.677964    44.883423   168.368409
min       0.000002     0.001087     1.475548
25%       3.618628    15.637721    35.834311
50%      18.204557    20.282036    40.453811
75%      38.292947    36.485428    49.754200
max    1245.670857   346.307507  1252.985527
```

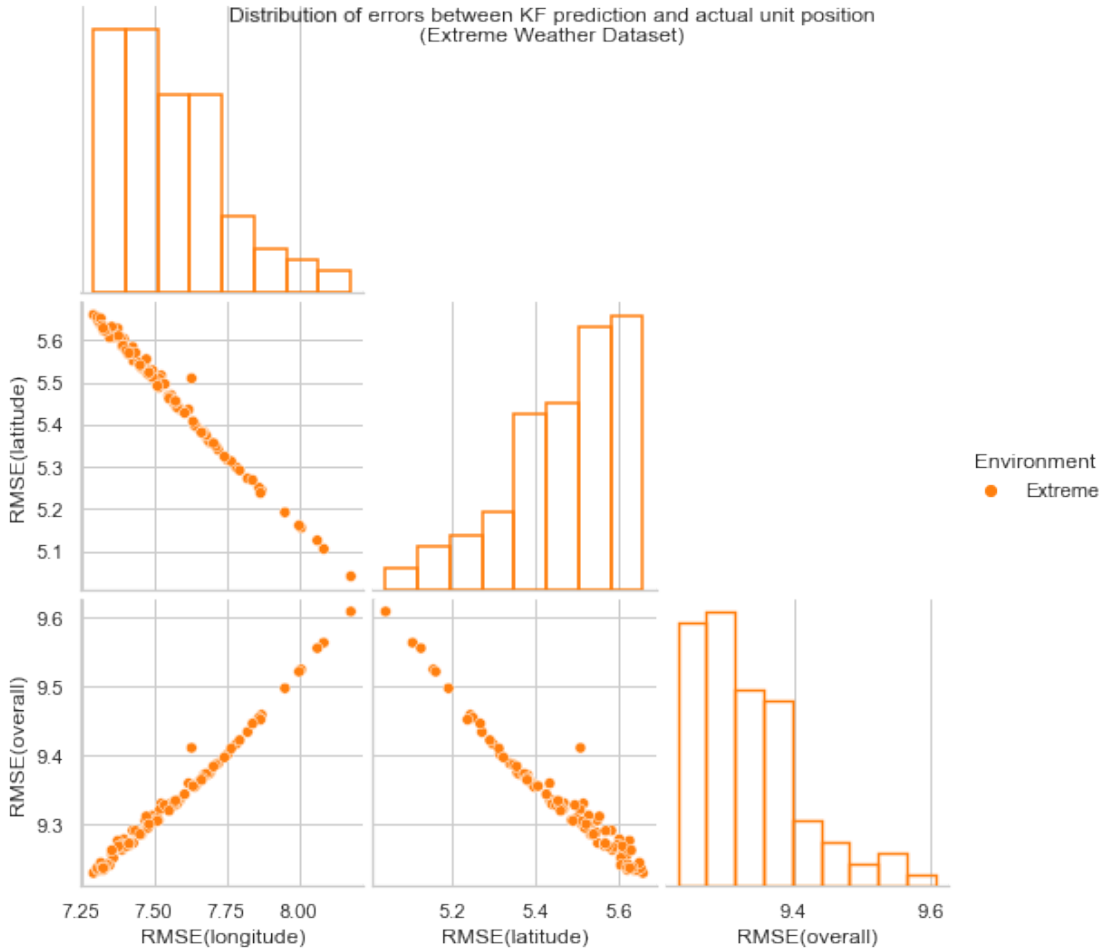
```
res_normal = pd.DataFrame({
    'RMSE(longitude)': res_mu_normal_x,
    'RMSE(latitude)': res_mu_normal_y,
    'RMSE(overall)': res_mu_normal,
    'Environment': 'Normal'
})

g = sns.pairplot(res_normal, corner=True, aspect=.9,
hue='Environment',palette=dict(Normal=sns.color_palette()[0], Extreme=sns.color_palette()[1]),
plot_kws=dict(), diag_kws=dict(fill=False), diag_kind="hist")
f = plt.gcf()
f.suptitle('Distribution of errors between KF prediction and actual unit position\n(Normal Weather
Dataset)')

Text(0.5, 0.98, 'Distribution of errors between KF prediction and actual unit position\n(Normal
Weather Dataset)')
```




```
res_normal.describe()
```

	RMSE(longitude)	RMSE(latitude)	RMSE(overall)
count	100.000000	100.000000	100.000000
mean	7.541179	5.478568	9.323550
std	0.181759	0.131409	0.073194
min	7.279208	5.038347	9.223251
25%	7.404439	5.398342	9.273100
50%	7.505960	5.507928	9.308336
75%	7.633440	5.579259	9.351967
max	8.172307	5.664053	9.600601

```
# Extreme environment
dir = './kf_eval_extreme/'
res_mu_extreme_x = []
res_mu_extreme_y = []
res_mu_extreme = []
for outfile in os.listdir(dir):
  df = pd.read_feather(os.path.join(dir, outfile))
  res_mu_extreme_x.append(np.sqrt(df.residual_E.mean()))
  res_mu_extreme_y.append(np.sqrt(df.residual_N.mean()))
  res_mu_extreme.append(np.sqrt(df.residual_2d.mean()))
res_extreme = pd.DataFrame({
    'RMSE(longitude)': res_mu_extreme_x,
    'RMSE(latitude)': res_mu_extreme_y,
    'RMSE(overall)': res_mu_extreme,
  'Environment': 'Extreme'
})

g = sns.pairplot(res_extreme, corner=True,
hue='Environment',palette=dict(Normal=sns.color_palette()[0], Extreme=sns.color_palette()[1]),
plot_kws=dict(), diag_kws=dict(fill=False), diag_kind="hist")
f = plt.gcf()
f.suptitle('Distribution of errors between KF prediction and actual unit position\n(Extreme
Weather Dataset)')

Text(0.5, 0.98, 'Distribution of errors between KF prediction and actual unit position\n(Extreme
Weather Dataset)')
```

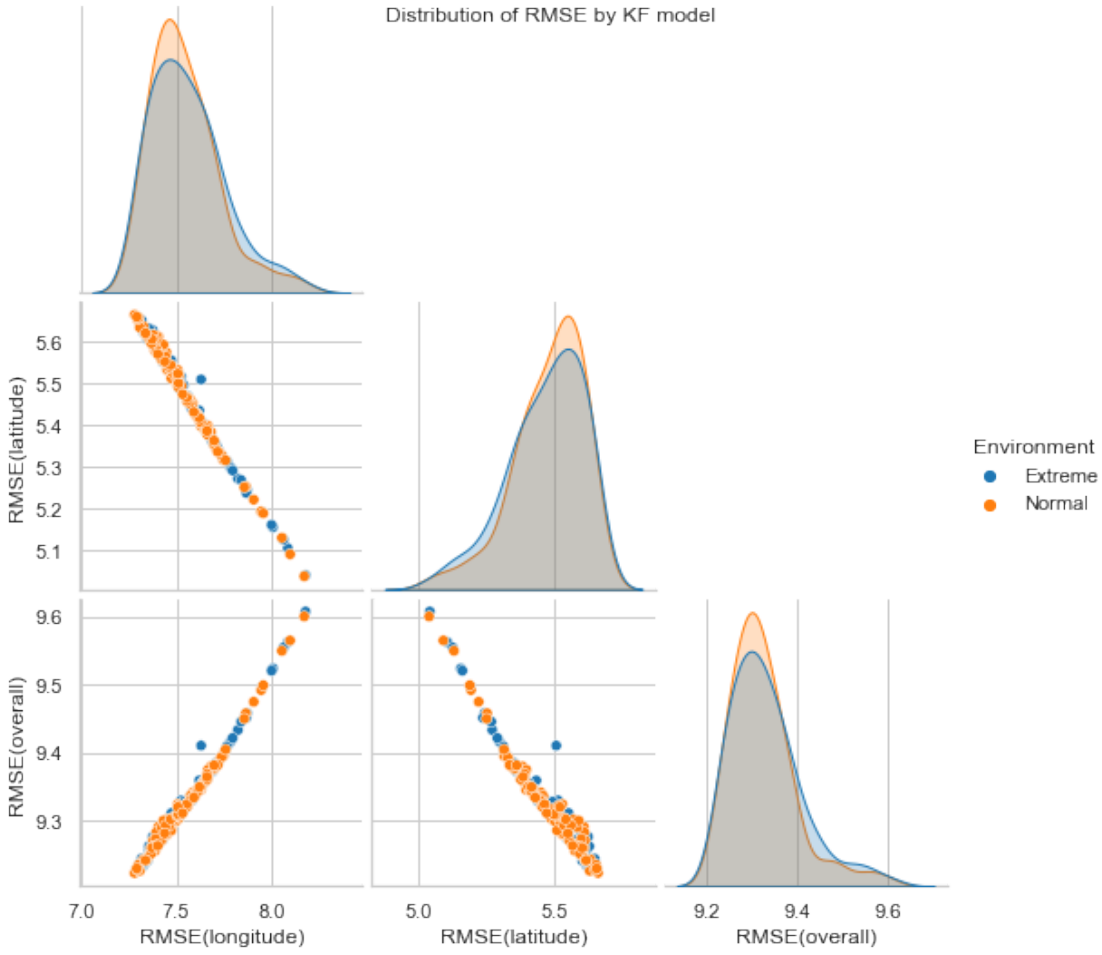


```
res_extreme.describe()
```

	RMSE(longitude)	RMSE(latitude)	RMSE(overall)
count	100.000000	100.000000	100.000000
mean	7.559086	5.467258	9.331756
std	0.195241	0.140200	0.079345
min	7.290751	5.040938	9.229639
25%	7.409365	5.381114	9.272755
50%	7.522444	5.497256	9.320632
75%	7.668317	5.577457	9.370688
max	8.179266	5.659610	9.607884

```
# Put EXTREME and NORMAL together
res_combined = pd.concat([res_extreme, res_normal], axis=0, ignore_index=True)
g = sns.pairplot(res_combined, corner=True, hue='Environment', diag_kind='kde')
f = plt.gcf()
f.suptitle('Distribution of RMSE by KF model')
```

```
Text(0.5, 0.98, 'Distribution of RMSE by KF model')
```



APPENDIX F. PYTHON SCRIPTS–ML MODEL TUNING

File: tune_nn.py

```
1.  import os
2.  import numpy as np
3.  import torch as th
4.  import torch.nn.functional as F
5.  import pandas as pd
6.  import random
7.
8.  from ray import tune
9.  from ray.tune.schedulers import ASHAScheduler
10. from functools import partial
11. from helper import *
12. from sensor import SensorDataWrapper, SensorDataset
13. from collections import OrderedDict
14. from sklearn.model_selection import train_test_split
15. from sklearn.preprocessing import MinMaxScaler
16.
17. # Set the manual seed for reproducibility.
18. seed = 3
19. random.seed(seed)
20. np.random.seed(seed)
21. th.manual_seed(seed)
22.
23.
24. # Create Data Lodaer
25. class CustomDataset(th.utils.data.Dataset):
26.
27.   def __init__(self, x, y):
28.     self.x = th.tensor(x)
29.     self.y = th.tensor(y)
30.     # debug
31.     # print(self.x.shape)
32.     # print(self.y.shape)
33.     assert self.x.shape[0] == self.y.shape[0], "x, y do not agree in length"
34.
35.   def __len__(self):
36.     return self.x.shape[0]
37.
38.   def __getitem__(self, i):
39.     return self.x[i], self.y[i]
40.
41.
42. class FC_NN(th.nn.Module):
43.   ### Creates a simple fc regression neural net
44.
45.   def __init__(self, input_dim, layers_dim, act_fn):
46.     super().__init__()
47.
48.     # input layer
49.     layers = [('input', th.nn.Linear(input_dim, layers_dim[0])), ('act_fn_0',
    act_fn())]
50.
51.     # fc hidden layers
52.     for i in range(1, len(layers_dim)):
53.       layers.append((f'layer_{i}', th.nn.Linear(layers_dim[i - 1],
```

```
    layers_dim[i])))
54.       layers.append((f'act_fn_{i}', act_fn()))
55.
56.     # output layer
57.     layers.append(('output_layer', th.nn.Linear(layers_dim[-1], 2)))
58.
59.     self.model = th.nn.Sequential(OrderedDict(layers))
60.
61.   def forward(self, obs):
62.     o = self.model(obs)
63.     return o
64.
65.
66. class Double_Head_NN(th.nn.Module):
67.   ### Creates a simple fc regression neural net
68.   ### Takes as input the
69.
70.   def __init__(self, input_dim, act_fn=th.nn.ReLU, num_hidden_nodes_1=8,
    num_hidden_nodes_2=2):
71.     super().__init__()
72.
73.     # input layer
74.     layers = [('input', th.nn.Linear(input_dim, num_hidden_nodes_1)),
    ('act_fn_0', act_fn())]
75.     layers.append((f'layer_1', th.nn.Linear(num_hidden_nodes_1,
    num_hidden_nodes_1)))
76.     layers.append((f'act_fn_1', act_fn()))
77.
78.     # fc mixed layers
79.     layers_1 = layers.copy()
80.     layers_1.append((f'layer_2_1', th.nn.Linear(num_hidden_nodes_1,
    num_hidden_nodes_2)))
81.     layers_1.append((f'act_fn_2_1', act_fn()))
82.     layers_1.append((f'output_1', th.nn.Linear(num_hidden_nodes_2, 1)))
83.
84.     layers_2 = layers.copy()
85.     layers_2.append((f'flayer_2_2', th.nn.Linear(num_hidden_nodes_1,
    num_hidden_nodes_2)))
86.     layers_2.append((f'act_fn_2_2', act_fn()))
87.     layers_2.append((f'output_2', th.nn.Linear(num_hidden_nodes_2, 1)))
88.
89.     self.model_1 = th.nn.Sequential(OrderedDict(layers_1))
90.     self.model_2 = th.nn.Sequential(OrderedDict(layers_2))
91.
92.   def forward(self, obs):
93.     x = self.model_1(obs)
94.     y = self.model_2(obs)
95.     # print(o)
96.     return th.cat([x, y], dim=1)
97.
98.
99. def create_dataset(sensor, weather='normal'):
100.         ds = {}
101.         random_state = 29071993
102.         data =
    pd.read_feather(f'C:\\Users\\moves\\Documents\\ml4cop\\data\\df_merged_{weather}
    _pos_{sensor}.ftr')
103.         ds = {}
104.         # remove columns that are not required, save memory
105.         data.drop([
```

```
            'TimelineID', 'Time', 'SensorID', 'SensorName',
    'SensorParentLongitude', 'SensorParentLatitude', 'SensorParentAltitude_AGL',
    'TargetID',
            'TargetName', 'TargetLongitude', 'TargetLatitude',
    'TargetAltitude_AGL_m', 'TargetRangeSlant_nm', 'DetectionResult',
    'DetectionAOU', 'Target_U',
            'Sensor_U', 'TimeDelta_x', 'UnitID', 'UnitName', 'UnitType',
    'UnitClass', 'UnitLongitude', 'UnitLatitude', 'UnitCourse', 'UnitSpeed_kts',
            'UnitAltitude_m', 'Unit_E', 'Unit_N', 'Unit_U', 'TimeDelta_y',
    'Err_ENU_z', 'Err_ENU_2d'
        ],
                    axis=1,
                    inplace=True)
        train_data, test_data = train_test_split(data, test_size=0.3,
    random_state=random_state, shuffle=True)
        val_data, test_data = train_test_split(test_data, test_size=0.5,
    random_state=random_state, shuffle=True)
        ds['train'] = {
            'x': train_data[['Target_E',
    'Target_N']].to_numpy(dtype=np.float32),
            'y': train_data[['Err_ENU_x',
    'Err_ENU_y']].to_numpy(dtype=np.float32)
        }
        ds['val'] = {
            'x': val_data[['Target_E',
    'Target_N']].to_numpy(dtype=np.float32),
            'y': val_data[['Err_ENU_x',
    'Err_ENU_y']].to_numpy(dtype=np.float32)
        }
        ds['test'] = {
            'x': test_data[['Target_E',
    'Target_N']].to_numpy(dtype=np.float32),
            'y': test_data[['Err_ENU_x',
    'Err_ENU_y']].to_numpy(dtype=np.float32)
        }

        print(f'sensor: {sensor}')
        print(f"train: {ds['train']['x'].shape}, {ds['train']['y'].shape}")
        print(f"val: {ds['val']['x'].shape}, {ds['val']['y'].shape}")
        print(f"test: {ds['test']['x'].shape}, {ds['test']['y'].shape}\n")

        # we only fit it to the training dataset, and use the fitted scaler to
    scale the rest of splits.
        scalers = MinMaxScaler()
        scalers.fit(ds['train']['x'])
        # Transform input:
        for _type in ['train', 'test', 'val']:
          ds[_type]['x'] = scalers.transform(ds[_type]['x'])

        return ds


    # Function to train moodel on training dataset.
    def train(model, dataLoader, criterion, optimizer, grad_clip_value,
    device='cpu'):
        # store training statistics
        batch_loss = []
        # Set model to training
        model.train()
        for x, y in dataLoader:
```

```
150.           x = x.to(device)
151.           y = y.to(device)
152.           optimizer.zero_grad()
153.           y_pred = model(x)
154.           loss = criterion(y, y_pred)
155.           # calculate gradient
156.           loss.backward()
157.           # clip the gradients to norm(grad) = 1.0
158.           th.nn.utils.clip_grad_norm_(model.parameters(), grad_clip_value)
159.           optimizer.step()
160.           # calculate training loss across batches.
161.           batch_loss.append(loss.item())
162.         avg_loss = np.mean(batch_loss)
163.         return avg_loss
164.
165.
166.       # Function for validating model on validation dataset.
167.       def val(model, dataLoader, criterion, device='cpu'):
168.         val_loss = []
169.         # set model to evaluation model
170.         model.eval()
171.         for x, y in dataLoader:
172.           x = x.to(device)
173.           y = y.to(device)
174.           y_pred = model(x)
175.           loss = criterion(y, y_pred)
176.           val_loss.append(loss.item())
177.         avg_loss = np.mean(val_loss)
178.         return avg_loss
179.
180.
181.       def train_doublehead_tune(config, checkpoint_dir=None, sensor=None):
182.
183.         # Create model
184.         model = Double_Head_NN(input_dim=config['input_dim'],
185.                                num_hidden_nodes_1=config['num_hidden_nodes_1']
    ,
186.                                num_hidden_nodes_2=config['num_hidden_nodes_2']
    )
187.         optimizer = th.optim.Adam(model.parameters(), lr=config['lr_init'],
    weight_decay=1e-4)
188.         scheduler = th.optim.lr_scheduler.StepLR(optimizer,
    step_size=config['lr_drop_step'], gamma=config['lr_drop_fraction'])  #,
    verbose=True)
189.         criterion = th.nn.L1Loss()
190.
191.         # Check for GPU
192.         device = 'cpu'
193.         # if th.cuda.is_available():
194.         #   device = 'cuda:0'
195.         # print(device)
196.         model = model.to(device)
197.
198.         # Create dataset
199.         data = create_dataset(sensor)
200.         train_dataset = CustomDataset(data['train']['x'], data['train']['y'])
201.         train_dataLoader = th.utils.data.DataLoader(train_dataset,
    batch_size=config['batch_size'])
202.         val_dataset = CustomDataset(data['val']['x'], data['val']['y'])
203.         val_dataLoader = th.utils.data.DataLoader(val_dataset)
```

```
204.         # store variables.
205.         best_epoch = None
206.         best_loss = np.inf
207.         # epoch_loss = {'train': [], 'val': []}
208.         patience_count = 0
209.
210.         # begins training neural network
211.         for epoch in range(config['max_epochs']):
212.
213.           # training step
214.           train_loss = train(model, train_dataLoader, criterion, optimizer,
    config['grad_clip'], device=device)
215.           scheduler.step()  # update the scheduler and change lr when
    required.
216.
217.           # validation step
218.           val_loss = val(model, val_dataLoader, criterion, device=device)
219.
220.           # store loss metrics for graphing:
221.           # epoch_loss['train'].append(train_loss)
222.           # epoch_loss['val'].append(val_loss)
223.           tune.report(train_loss=train_loss, val_loss=val_loss)
224.
225.           if val_loss < best_loss:
226.             best_loss = val_loss
227.             best_epoch = epoch
228.             # print(f'epoch {best_epoch}: best model loss = {best_loss:.4f}')
229.             # reset patience_count:
230.             patience_count = 0
231.           # Early stop training if the validation loss does not improve after
    val_patience epoch.
232.           if patience_count == config['patience']:
233.             break
234.           else:
235.             patience_count += 1
236.
237.           with tune.checkpoint_dir(epoch) as checkpoint_dir:
238.             path = os.path.join(checkpoint_dir, "checkpoint")
239.             th.save((model.state_dict(), optimizer.state_dict()), path)
240.           # print(path)
241.
242.
243.       if __name__ == "__main__":
244.
245.         def main(sensor=None, weather='normal'):
246.           # load dataset
247.           # the dataset consists of target_E,N,U and sensor_E,N,U and the
    measurement error (using RMSE)
248.           max_num_epochs = 500
249.           num_samples = 400
250.
251.           config = {
252.               "patience": 20,
253.               "input_dim": 2,
254.               "max_epochs": 500,
255.               "num_hidden_nodes_1": tune.choice([8, 16]),
256.               "num_hidden_nodes_2": tune.choice([2, 4]),
257.               "lr_init": tune.choice([0.5, 0.3, 0.2, 0.1]),
258.               "lr_drop_step": tune.choice([10, 15, 20]),
259.               "lr_drop_fraction": tune.choice([0.9, 0.8, 0.5, 0.1]),
```

```
260.                 "batch_size": tune.choice([16, 32, 64]),
261.                 "grad_clip": 1,
262.             }
263.
264.             Scheduler = ASHAScheduler(metric="val_loss", mode="min",
    max_t=max_num_epochs, grace_period=10, reduction_factor=2)
265.
266.             reporter = tune.CLIReporter(metric_columns=["train_loss",
    "val_loss", "training_iteration"], print_intermediate_tables=True)
267.             result = tune.run(
268.                 partial(train_doublehead_tune, checkpoint_dir=f'./tune/',
    sensor=sensor),
269.                 config=config,
270.                 num_samples=num_samples,
271.                 scheduler=scheduler,
272.                 local_dir=f'./tune/',
273.                 name=f'{sensor}',
274.                 resources_per_trial={"cpu": 2},
275.                 max_concurrent_trials=10,  # 24 CPU in total, using up to 20
    CPU.
276.                 resume=True,
277.                 progress_reporter=reporter)
278.
279.             best_trial = result.get_best_trial("val_loss", "min", "last")
280.             print("Best trial config: {}".format(best_trial.config))
281.
282.         main(sensor='ESM')
283.         main(sensor='EO')
284.         main(sensor='IR')
285.         main(sensor='Radar')
```

File: Sensor.py

```
1.  """
2.  Everything related to sensors here.
3.  """
4.
5.  from helper import SENSORS
6.  import pandas as pd
7.  import numpy as np
8.
9.  DEBUG_PRINT = True
10.
11.
12. class SensorDataWrapper:
13.   """
14.   Define each sensor as a sensor class itself, and provides the dataset when
15.   called by the other function.
16.   """
17.
18.   def __init__(self, dataset_dict, sensor, weather='normal'):
19.     self.data = dataset_dict[weather][sensor]
20.     self.timelineID = None
21.     self.time = 0.  # maintains a clock within itself to provide error-checking.
22.     self._data = None
23.     self.num_entries = None
24.     self.start_time = self.data.Time.min()
25.     self.end_time = self.data.Time.max()
26.     self.sensor = sensor  # name of sensor
```

```
27.
28.   def update_periodicity(self, period):
29.     self.periodicity = period
30.
31.   def set_timelineID(self, timelineID):
32.     self.timelineID = timelineID
33.     self.set_time(0)  # reset the clock, since we are interested in the new
    timeline now
34.     self._data = self.data[self.data.TimelineID == timelineID]
35.     self._data.set_index('Time', inplace=True)
36.     self.num_entries = len(self._data)
37.
38.   def set_time(self, time):
39.     self.time = time
40.
41.   def get_next_detection(self):
42.     """
43.     this is the main interface with various functions.
44.     After calling the set_timelineID, get_next_detection would return parameters
    of interests
45.     returns (t,x,y) where t= time of detection, x=latitude, y=longitude
46.     """
47.     try:
48.       x = self._data.loc[self.time]['Target_E']
49.       y = self._data.loc[self.time]['Target_N']
50.       return (self.time, x, y)
51.     except KeyError:
52.       print(f'{self.time} is not in the index from data')
53.
54.   def tick(self):
55.     # Advance the clock
56.     # if there are no more sensor data, return False
57.     self.time += 1.
58.     return self.time <= self.end_time and self.check_alert()
59.
60.   def check_alert(self):
61.     ### alert when there is a detection.
62.     return self.time in self._data.index
63.
64.
65. class SensorDataset:
66.   """
67.   SensorDataset is a class comprising of all the sensor dataset for a specific
    run.
68.   """
69.
70.   def __init__(self, data, weather='normal') -> None:
71.     self.sensor_dat = {}
72.     for sensor in SENSORS:
73.       self.sensor_dat[sensor] = SensorDataWrapper(data, sensor, weather=weather)
74.     if DEBUG_PRINT:
75.       print(f'sensor:                                                {sensor}\tstart:
    {self.sensor_dat[sensor].start_time}\t#entries:
    {self.sensor_dat[sensor].num_entries}')
76.     self.timelineIDs = self.sensor_dat[SENSORS[0]].TimleineID.unique()
77.     self.detections = []
78.
79.   def get_timelineIDs(self):
80.     return self.timelineIDs
81.
```

```
82.   def set_timelineID(self, timelineID):
83.     """
84.     The simulation run defined by the timelineID that the dataset should be
    focused on
85.     """
86.     self.time = 0  # reset the clock when setter is called.
87.     self.detections = {}
88.     for sensor in SENSORS:
89.       self.sensor_dat[sensor].set_timelineID(timelineID)
90.     if DEBUG_PRINT:
91.       print(f'sensor dataset: set timeline @ {timelineID}')
92.
93.   def tick(self):
94.     """
95.     provide the next available sensor detection from the dataset.
96.     Some seconds may have more than 1 detection, hence we prepare an array of
    detection from all sensors
97.     and provide the detection to the environment accordingly.
98.     The order in which the detections are presented are in an unspecified order.
99.     """
100.           self.time += 1
101.           self.detections = []
102.           for sensor in SENSORS:
103.             if self.sensor_dat[sensor].tick():
104.               (t, x, y) = self.sensor_dat[sensor].get_next_detection()
105.               assert t == self.time, f"alert is out of sync (got t={t}, but
    time is {self.time})"
106.               if DEBUG_PRINT:
107.                 print(f'{sensor}:{t}')
108.               self.detections.append((sensor, t, x, y))
109.
110.
111.       def get_sensor_stats():
112.         sensor_err_stats = pd.read_pickle('../data/sensor_err_stats.pkl')
113.         mat_R = {}
114.         for sensor in SENSORS:
115.           mat_R[sensor] = np.array([[sensor_err_stats.loc['std_x', sensor]**2,
    0.], [0., sensor_err_stats.loc['std_y', sensor]**2]])
116.         return mat_R
```

LIST OF REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … & Zheng, X. (2015). Tensorflow: Large-scale machine learning on heterogeneous systems [Python]. https://www.tensorflow.org/

Akca, A., & Efe, M. Ö. (2019). Multiple model Kalman and particle filters and applications: A survey. *IFAC-PapersOnLine*, *52*(3), 73–78. https://doi.org/10.1016/j.ifacol.2019.06.013

Bekhtaoui, Z., Meche, A., Dahmani, M., & Meraim, K. A. (2017). Maneuvering target tracking using Q-learning based Kalman filter. *2017 5th International Conference on Electrical Engineering – Boumerdes (ICEE-B)*, 1–5. https://doi.org/10.1109/ICEE-B.2017.8192005

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Blasch, E. P., Rogers, S. K., Holloway, H., Tierno, J., Jones, E. K., & Hammoud, R. I. (2014). QuEST for information fusion in multimedia reports. *International Journal of Monitoring and Surveillance Technologies Research*, *2*(3), 1–30. https://doi.org/10.4018/IJMSTR.2014070101

Cardoso Silva, L., Rezende-Zagatti, F., Silva-Sette, B., Nildaimon dos Santos Silva, L., Lucredio, D., Furtado Silva, D., & de Medeiros Caseli, H. (2020). Benchmarking machine learning solutions in production. *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 626–633. https://doi.org/10.1109/ICMLA51294.2020.00104

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., & Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems (Item No. arXiv:1512.01274). arXiv. http://arxiv.org/abs/1512.01274

Chollet, F. (2015). Keras [Python]. https://keras.io

Crothers, M. B., Lanphear, M. J., & Garino, M. B. (2009). U.S. space-based intelligence, surveillance, and reconnaissance. In *AU-18 space primer*. Air University Press. https://www.airuniversity.af.edu/Portals/10/AUPress/Books/AU-18.PDF

Department of Defense. (2018). Summary of the 2018 Department of Defense artificial intelligence strategy. https://media.defense.gov/2019/Feb/12/2002088963/-1/-1/1/summary-of-dod-ai-strategy.PDF

Department of Defense. (2019). DoD digital modernization strategy 2019. https://media.defense.gov/2019/Jul/12/2002156622/-1/-1/1/DOD-digital-modernization-strategy-2019.PDF

Department of Defense. (2020). DoD data strategy 2020. https://media.defense.gov/2020/Oct/08/2002514180/-1/-1/0/DOD-data-strategy.PDF

Dietrich, N. S. (2001). Performance metrics for correlation and tracking algorithms. [Master's thesis, Naval Postgraduate School]. NPS Archive: Calhoun. https://calhoun.nps.edu/handle/10945/2473

Esteban, J., Starr, A., Willetts, R., Hannah, P., & Bryanston-Cross, P. (2005). A review of data fusion models and architectures: Towards engineering guidelines. *Neural Computing and Applications, 14*(4), 273–281. https://doi.org/10.1007/s00521-004-0463-7

Faragher, R. (2012). Understanding the basis of the Kalman filter via a simple and intuitive derivation. *IEEE Signal Processing Magazine, 29*(5), 128–132. https://doi.org/10.1109/MSP.2012.2203621

FLIR Systems. (2010). Datasheet of SeaFLIR-II. https://www.psicompany.com/man-prod-info/FLIR/Thermal-Imaging-Cameras/SeaFLIR-II/SeaFLIR-II-Datasheet.pdf

Gao, X., Luo, H., Ning, B., Zhao, F., Bao, L., Gong, Y., Xiao, Y., & Jiang, J. (2020). Rl-AKF: An adaptive Kalman filter navigation algorithm based on reinforcement learning for ground vehicles. *Remote Sensing, 12*(11), 1704. https://doi.org/10.3390/rs12111704

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. http://www.deeplearningbook.org

Graves, A. (2012). Long short-term memory. In A. Graves (Ed.), *Supervised Sequence Labelling with Recurrent Neural Networks* (pp. 37–45). Springer. https://doi.org/10.1007/978-3-642-24797-2_4

Guerriero, M., Willett, P., Coraluppi, S., & Carthel, C. (2008). Radar/AIS data fusion and SAR tasking for maritime surveillance. *2008 11th International Conference on Information Fusion*, 1–5.

Hoehn, J.R. (2022). Joint all-domain command and control (JADC2) (CRS Report No. IF11493). Congressional Research Service. https://crsreports.congress.gov/product/pdf/R/R46725/2

Joint Chiefs of Staff. (2013). Joint tactics, techniques, and procedures for intelligence support to targeting. https://irp.fas.org/doddir/dod/jp2_01_1.pdf

Joint Chief of Staff. (2018). Joint targeting. https://www.justsecurity.org/wp-content/uploads/2015/06/Joint_Chiefs-Joint_Targeting_20130131.pdf

Jouaber, S., Bonnabel, S., Velasco-Forero, S., & Pilté, M. (2021). NNAKF: A neural network adapted Kalman filter for target tracking. *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4075–4079. https://doi.org/10.1109/ICASSP39728.2021.9414681

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S. A. A., Ballard, A. J., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., … & Hassabis, D. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, *596*(7873), 583–589. https://doi.org/10.1038/s41586-021-03819-2

Jung, S., Schlangen, I., & Charlish, A. (2020). Time-dependent state prediction for the Kalman filter based on recurrent neural networks. *2020 IEEE 23rd International Conference on Information Fusion (FUSION)*, 1–7. https://doi.org/10.23919/FUSION45008.2020.9190484

Karayev, S., Van Pelt, C., & Tobin, J. (2022). Full stack deep learning. *Full Stack Deep Learning*. https://fullstackdeeplearning.com/

Kendall, A., & Gal, Y. (2017). What uncertainties do we need in Bayesian deep learning for computer vision? (arXiv:1703.04977). arXiv. https://doi.org/10.48550/arXiv.1703.04977

Kim, S., Petrunin, I., & Shin, H.-S. (2022). A review of Kalman filter with artificial intelligence techniques. *2022 Integrated Communication, Navigation and Surveillance Conference (ICNS)*, 1–12. https://doi.org/10.1109/ICNS54818.2022.9771520

Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization (arXiv:1412.6980). arXiv. https://doi.org/10.48550/arXiv.1412.6980

Koch, W. (2014). *Tracking and sensor data fusion*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-39271-9

Koch, W. (2015). Hard & soft fusion cornerstone of information processing and management an introduction with defence and security examples. *NATO STO Lecture Series - Advanced Algorithms for Effectively Fusing hard and soft Information (IST-134)*. https://www.sto.nato.int/publications/STO%20Educational%20Notes/STO-EN-IST-134/EN-IST-134-01.pdf

Kong, L., Peng, X., Chen, Y., Wang, P., & Xu, M. (2020). Multi-sensor measurement and data fusion technology for manufacturing process monitoring: A literature review. *International Journal of Extreme Manufacturing, 2*(2), 022001. https://doi.org/10.1088/2631-7990/ab7ae6

Kreuzberger, D., Kühl, N., & Hirschl, S. (2022). Machine learning operations (MLOps): Overview, definition, and architecture (arXiv:2205.02302). arXiv. https://doi.org/10.48550/arXiv.2205.02302

Labbe, R. (2022). Filterpy [Python]. https://github.com/rlabbe/filterpy (Original work published 2014)

Lau, B. P. L., Marakkalage, S. H., Zhou, Y., Hassan, N. U., Yuen, C., Zhang, M., & Tan, U.-X. (2019). A survey of data fusion in smart city applications. *Information Fusion, 52*, 357–374. https://doi.org/10.1016/j.inffus.2019.05.004

Li, G., Yan, Z., Fu, Y., & Chen, H. (2018). Data fusion for network intrusion detection: A review. *Security and Communication Networks*, https://doi.org/10.1155/2018/8210614

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training. ArXiv Preprint ArXiv:1807.05118.

Liermann, V. (2021). Overview machine learning and deep learning frameworks. In C. Stegmann (Ed.), *Data storage, data processing and data analysis: Vol. III.* Springer International Publishing. https://doi.org/10.1007/978-3-030-78821-6

Llinas, J. (2008). Assessing the performance of multisensor fusion processes. In M. I. Liggins & D. Hall (Eds.), *Handbook of multisensor data fusion (2nd ed.)*. CRC Press.

Maltese, D., & Lucas, A. (1998). Data fusion: Principles and applications in air defense. *Signal Processing, Sensor Fusion, and Target Recognition VII, 3374,* 329–336. https://doi.org/10.1117/12.327110

MathWorks. Tracking and tracking Filters—MATLAB & Simulink. (2022). Retrieved July 8, 2022, from https://www.mathworks.com/help/fusion/gs/tracking-and-tracking-filters.html

Matrix Games. (2022a). Command Lua API documentation v1147.34. https://commandlua.github.io/index.html

Matrix Games. (2022b). Command: Modern operations (1.04.114745) [Windows]. https://www.matrixgames.com/game/command-modern-operations

Matrix Games. (2022c). *Command professional edition user manual*, version 2.0.

Merten, S. (2014). Employing data fusion in cultural analysis and COIN in tribal social systems. In *Culture, conflict, and counterinsurgency*. Stanford University Press. https://doi.org/10.11126/stanford/9780804785952.003.0004

Miller, S., Blais, C., & Green, J. (2020). Modeling the operational value of data fusion on ASW and other missions. [Master's thesis, Naval Postgraduate School]. NPS Archive: Calhoun. https://calhoun.nps.edu/handle/10945/67927

Murashov, D. (2021). A review and proposal for developing of data fusion models and frameworks for decision making systems. *Proceedings of the 33rd European Modeling & Simulation Symposium*, 116–125. https://doi.org/10.46354/i3m.2021.emss.016

Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., López García, Á., Heredia, I., Malík, P., & Hluchý, L. (2019). Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey. Artificial Intelligence Review, 52(1), 77–124. https://doi.org/10.1007/s10462-018-09679-z

Nicholas, M. L. (2008). Survey of multisensor data fusion systems. In L. I. Martin, D. Hall, & J. Llinas (Eds.), *Handbook of multisensor data fusion (2nd ed.)*. CRC Press.

Nix, D. A., & Weigend, A. S. (1994). Estimating the mean and variance of the target probability distribution. *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94), 1,* 55–60. https://doi.org/10.1109/ICNN.1994.374138

Northrop Grumman Sperry Marine B.V. (2005). BridgeMaster E radar ship's manual. Northrop Grunmman Sperry Marine B.V. https://www.marinsat.com/en/product/bridgemaster-e

Pandas Development Team. (2020). Pandas-dev/pandas: Pandas (latest). Zenodo. https://doi.org/10.5281/zenodo.3509134

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems, 32*. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-Learn: Machine learning in python. *Journal of Machine Learning Research, 12,* 2825–2830.

Python. (2022). Python 3-D coordinate conversions [Computer Software]. Geospace code. https://github.com/geospace-code/pymap3d

Raytheon Technologies. (2005). AN/SLQ-32(V) shipboard EW system. Raytheon Technologies. http://www.raytheon.com/products/stellent/groups/sas/documents/asset/slq32.pdf

Rothenhaus, K. J. (2008). Data strategies to support automated multi-sensor data fusion in a service oriented architecture [Doctoral Dissertation, Naval Postgraduate School]. NPS Archive: Calhoun. https://calhoun.nps.edu/handle/10945/10348

Russell, S., Norvig, P., & Davis, E. (2010). *Artificial intelligence: A modern approach (3rd ed.)*. Prentice Hall. http://aima.cs.berkeley.edu/

Schachter, B. J. (2020). *Automatic target recognition*. SPIE Press. https://books.google.com/books?id=EfvIzAEACAAJ

Seitzer, M., Tavakoli, A., Antic, D., & Martius, G. (2022). On the pitfalls of heteroscedastic uncertainty estimation with probabilistic neural networks. https://arxiv.org/abs/2203.09168

Sharma, J., & Stokes, G. H. (2002). *Toward Operational Space-Based Space Surveillance. 13*(2), 26.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature, 550*(7676), 354–359. https://doi.org/10.1038/nature24270

Singapore Armed Forces [SAF]. (2022). *Fighting the COVID-19 pandemic—Leadership and reflections from the SAF*. Singapore Armed Forces.

Smith, D., & Singh, S. (2006). Approaches to multisensor data fusion in target tracking: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *18*(12), 1696–1710. https://doi.org/10.1109/TKDE.2006.183

Steinberg, A. N., Bowman, C. L., & White, F. E. (2017). Revisions to the JDL data fusion model. In *Handbook of multisensor data fusion* (pp. 65–88). CRC press.

Ullah, I., Fayaz, M., & Kim, D. (2019). Improving accuracy of the Kalman filter algorithm in dynamic conditions using ANN-based learning module. Symmetry, 11(1), 94. https://doi.org/10.3390/sym11010094

Ullah, I., Fayaz, M., Naveed, N., & Kim, D. (2020). ANN based learning to Kalman filter algorithm for indoor environment prediction in smart greenhouse. IEEE Access, 8, 159371–159388. https://doi.org/10.1109/ACCESS.2020.3016277

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., … & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature, 575*(7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z

Wang, P., Yang, L. T., Li, J., Chen, J., & Hu, S. (2019). Data fusion in cyber-physical-social systems: State-of-the-art and perspectives. *Information Fusion, 51*, 42–57. https://doi.org/10.1016/j.inffus.2018.11.002

Watson, F. O. (2021). Design methodologies for 21st century entity correlation [Master's thesis, Naval Postgraduate School]. NPS Archive: Calhoun. https://calhoun.nps.edu/handle/10945/68396

West, T. D., & Birkmire, B. (2019). AFSIM: The air force research laboratory's approach to making M&S ubiquitous in the weapon system concept development process. *CSIAC Journal, 7*(3), 50–55.

Winder, P. (2019, July 1). A comparison of reinforcement learning frameworks. https://winder.ai/a-comparison-of-reinforcement-learning-frameworks-dopamine-rllib-keras-rl-coach-trfl-tensorforce-coach-and-more/

Yeong, D. J., Velasco-Hernandez, G., Barry, J., & Walsh, J. (2021). *Sensor and sensor fusion technology in autonomous vehicles: A review. 21*(6), 2140. https://doi.org/10.3390/s21062140

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California