AI-based Multi-Sensor Fusion

Final Report Binder, 12/02/2022

TABLE OF CONTENTS

Definitions

Document Definitions

Table 01-1 Document Definitions

Term	Definition
Design History File	A compilation of records containing the complete design history of a finished device or service
Verification	Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled
Validation	Establishing objective evidence that system specifications conform to user needs and intended uses
Component	One of the parts that make up a system. A component may be hardware or software and may be subdivided into components
Functional Testing	Testing that ignores the internal mechanism of a system or component and focuses on the outputs generated in response to selected inputs
Training Data	The set of data that is provided by Aerospace Corporation, in order to both develop and provide verification and validation to the product

Machine Learning Model	A program that has the ability to learn given input data

Document Acronyms

Table 01-2 Document Acronyms

Acronym	Description
DHF	Design History File
IEEE	Institute of Electrical and Electronics Engineers
CR	Customer Requirement
SR	System Requirement
PNT	Position, Navigation, Time
VT	Virginia Polytechnic University
MSF	AI based Multi-Sensor Fusion Project

Objective and Requirements Information

Customer Needs

Problem Statement

Aerospace Corporation is currently contemplating the utilization of a Kalman filter model implementation to improve PNT accuracy. By fusing GPS and IMU sensor inputs, a potentially enhanced solution can be employed rather than the inputs individually. By the end of next semester, our team is expected to produce a neural network by applying deep learning techniques to the training data provided by Aerospace Corp in order to emulate and potentially improve upon a standard Kalman Filter. Time permitting, a Kalman Filter made adaptive by the supplement of a neural network will also be investigated.

Customer Needs Description

A GPS will have a certain amount of error in computing the position, speed, and acceleration of some arbitrary object being tracked. Software applications are one of the tools that can be used to minimize the error, by combining an estimation of the tracked object's state and the sensor data. The customer wants the function of a standard Kalman Filter, the standard tool for this sort of task, to be replicated with a neural network. This process is shown below. The desired product is a new software system, in the form of a Machine Learning Model, that has equivalent or greater performance.

Key Stakeholders

- Virginia Tech: VT wants projects that they sponsor to do well. It reflects well on them as an institution giving them more credibility. This then correlates to higher enrollment numbers, donations, and government funding since VT is a federal institution.
- VT Multi-Sensor Fusion Student Group Members: The students working on the VT sponsored MSF will gain real world marketable experience. Working with industry professionals with a timeline and implementable final deliverables will give the students invaluable experiences working on real world projects.
- Aerospace Corp: Aerospace Corporation sponsoring the MSF means they have a vested interest in the success of the project. Most likely they benefit from sponsoring the project by receiving a working product they can implement or scouting talent to recruit and work for them.
- United States Federal Government: Aerospace Corporation is a federally funded non-profit organization which provides consulting in a number of sectors. Developing more options for GPS error correction could be of interest to the government funding such research.

General Constraints

External Factors - Global, Cultural, and Environmental

GPS, or global positioning system, has become a natural tool in numerous societies. It allows for the average person to get from point A to point B by entering a destination and the ideal route comes up in seconds. Technology has advanced at such a rapid pace that many people in today's world would struggle to travel without GPS. It has become such a profitable and advantageous industry that many organizations across the globe dedicate a large portion of their time and money to advancing their GPS systems. This is why our project that involves developing enhanced PNT solutions will have global, cultural, and environmental constraints.

Globally, our project constraints are limited to potentially advancing foreign powers with advanced GPS technology. It's an extreme case, but on a global scale it is always something to consider. Especially since our customer is a defense contracting organization, keeping materials safeguarded is a key component to their everyday activities. In terms of cultural constraints, our model will be difficult to utilize in areas that don't have access to large GPS and IMU sensor data.

It takes time and money to organize data for a model so if an organization has limited resources it will be difficult to implement the model to its fullest potential.

Environmental constraints aren't entirely related to our model but it can be in some aspects. Our project involves collecting data such as GPS, which includes location, velocity, and time data as well as IMU data which includes acceleration, angular velocity, and magnetic field measurements. All of these can be affected by environmental constraints such as will the environment for the data we are collecting be affected by outside factors that impact the model results. If this is the case, then the data will not be as precise as it could be, and the model output will be negatively affected. In conclusion, these constraints allow us to plan and adapt for our project implementation as we continue to progress.

Social Factors - Public Health, Safety, and Welfare

Personal safety is not an issue of great concern to a project entirely developed as software. However, there are implications for public safety present in the project. A malfunctioning GPS can potentially lead drivers into dangerous situations. Thus, there is enormous importance placed on ensuring that the system functions reliably.

When considering public safety, one of the most important things to consider about our project is the fact that a convolutional neural network is a statistical construct. Its quality is therefore determined by the variety, quantity, and quality of the data provided to test it. In order to ensure that our network operates safely, we have to confirm that the data is of decent quality, through the use of software that can calculate statistical information about the data, and plot it out in graphical forms, allowing analysis.

Malware could represent a risk to our development environment. It could corrupt files on a computer, prevent people from being able to work, and risk exposing the data we are being provided. For this reason, we should ensure that all our work is backed up, and that we use secure connections with trusted internet providers when working.

Ethical Factors - Global, Societal, Economic, and Environmental

Since the MSF team is relatively small, two of the three IEEE code of ethics principles do not apply as much. In regards to the second IEEE ethics principle about discrimination, naturally we will treat each other and advisors with respect and free of discrimination. However, being such a small group means the risk of such condemnable actions is highly unlikely so such violations are not a big concern. Similarly the concern for lack of ethical accountability amongst the team (third IEEE ethics principle) is also low due to the group size being so small. It is difficult to pass on accountability when failing to meet responsibilities in small groups is very evident. The most applicable IEEE ethics principle to the MSF group would be about upholding standards. More specifically the principle about, seeking, accepting, and offering honest criticism and being honest about estimates and results.

Given the short discussion on ethics above as pertaining to this project, there are a few notable areas for further ethical discussion. Since the core members of the MSF group have limited experience working on any project from its inception through to its ends while also having limited experience working with Machine Learning there will likely be numerous mishaps throughout development as this project incorporates both. Thus, it is of utmost importance that the MSF team learn the most from mistakes made so that they may be corrected, while maintaining a professional demeanor in terms of giving and receiving criticism. More applicable though is the importance of being honest about data collection and testing. Machine learning models function by taking in some data set and tuning its parameters to fit some desired results. Accepting dubious data to construct Neural Networks would be a violation of the Team Charter all core team members signed prior to beginning work on this project and would lead to poor real world application of any final product constructed in this manner.

As students of Virginia Tech we are the embodiment of our motto “Ut Prosim”, so that I/we may serve. This means we will do our best to conduct ourselves ethically and hold one another accountable to do the same. It is also our responsibility as soon-to-be fully fledged members of society to create a world we’d be proud to live in, and this starts with living by the principles we say we stand for (1).

Product Requirements

Target Specifications

Target specifications are prioritized 1 to 5, 5 being the highest priority.

Table 01-5 Target Specifications

Req. #	Metric	Priority	Units	Marginal Value	Target Value
SR-1	Model Variance	1	R^2	.90	.95
CR-1, CR-3	Program Execution time	4	seconds	15	5
CR-2, CR-6	Comparison with GPS only data	2	Root Mean Squared Error	Less than 10m	.Better than the GPS data
CR-3, CR-5	GPU usage	5	%GPU used	More than 80%	95%
CR-4	Training Time	3	Seconds per dataset	300	120

			used in training		

Standards and Statutory Requirements

Table 01-6 Standards and Statutory Requirements

Req. #	Requirement	Source Document (e.g., standard, regulatory requirements)	Details
SR-1	Implement extensive and transparent testing(1)	Link i.5	To ensure that our products meet sponsor requirements we will implement rigorous testing so that our model meets expectations when given a data set that was not used for training the model
SR-2	Develop ML Learning Models(Linear Regression, KNN, Perceptron, Neural Network) using PyTorch framework(4)	PyTorch	Customer wants the model to be built in either PyTorch or TensorFlow
SR-3	A New Standard for Assessing the Well-Being Implications of Artificial Intelligence(3)	IEEE 7010 - 20277880	General requirements regarding the concern of rapid AI integration into society
SR-4	Software, systems and enterprise Architecture evaluation framework(2)	IEEE 42030-2019 19557621	Identifying principles and constructs that will ensure the architecture implemented will have a maximized impact towards success
SR-5	Benchmarking Deep Learning Frameworks: Design	IEEE INSPEC #: 17936825	Benchmarking Metrics for DL

	Considerations, Metrics and Beyond(5)		

Detailed Design

Schedule and Potential Costs

Our schedule was designed in two parts, for the two respective semesters, with the second schedule being made using the website Jira, for easy task subdivision and updating information.

Semester 1 Schedule

Task Name	Duration	Start	Finish
Major Design Experience	**76 days?**	**Mon 1/17/22**	**Fri 4/29/22**
1 My MDE Project Artifacts			
1.1 Requirements Specifications	30 days	Mon 1/17/22	Fri 2/25/22
1.2 Preliminary Design Review	51 days	Mon 1/17/22	Mon 3/28/22
1.3 Critical Design Review	70 days	Mon 1/17/22	Thu 4/21/22
1.4 Detiled Design Report	75 days	Mon 1/17/22	Thu 4/28/22
2 MDE Design Processes			
2.1 Architecture Definition			
2.2.1 Top Level Diagram	51 days	Mon 1/17/22	Mon 3/28/22
2.2.2 Initial Design Concepts	25 days	Mon 1/17/22	Fri 2/18/22
2.2.3 Architecture Defintion	51 days	Mon 1/17/22	Mon 3/28/22
2.2.4 Architecture Evaluation	75 days	Mon 1/17/22	Thu 4/28/22
2.2.6 Preliminary Design Review	51 days	Mon 1/17/22	Mon 3/28/22
2.2 Detailed Design			
2.2.1 Collect Data	12 days	Tue 3/1/22	Wed 3/16/22
2.2.2 Preprocess Data	16 days	Fri 3/18/22	Fri 4/8/22
2.2.3 Feature Engineering	69 days	Mon 1/17/22	Wed 4/20/22
2.2.4 Train Model	69 days	Mon 1/17/22	Wed 4/20/22
2.2.5 Tune Model	72 days	Mon 1/17/22	Mon 4/25/22
2.3 Implementation & Verification			
2.5 Project Management	**75 days**	**Mon 1/17/22**	**Thu 4/28/22**
2.5.1 Project Control	75 days	Mon 1/17/22	Thu 4/28/22
2.5.2 Decision Management	19 days	Mon 1/17/22	Thu 2/10/22
2.5.3 Risk Management	75 days	Mon 1/17/22	Thu 4/28/22
2.5.4 Process Performance Measurement	64 days	Mon 1/17/22	Thu 4/14/22
3 Class Sessions			
3.1 4805 Classes	**63 days**	**Tue 1/18/22**	**Thu 4/14/22**
3.1.1 L1 Intro & Overview	0 days	Tue 1/18/22	Tue 1/18/22
3.1.2 L2 Project Management	0 days	Thu 1/20/22	Thu 1/20/22
3.1.3 L3 Coaching & Resumes	0 days	Tue 1/25/22	Tue 1/25/22
3.1.4 L4 Engineering Requirements	0 days	Sun 2/27/22	Sun 2/27/22
3.1.5 L5 Communication and Teamwork	0 days	Tue 2/8/22	Tue 2/8/22
3.1.6 L6 Architecture Design	0 days	Tue 2/22/22	Tue 2/22/22
3.1.7 L7 Ethics	0 days	Tue 3/1/22	Tue 3/1/22
3.1.8 L8 Detailed Design	0 days	Tue 3/15/22	Tue 3/15/22
3.1.9 L9 Intellectual Property	0 days	Mon 1/31/22	Mon 1/31/22
3.1.10 L10 AAR	0 days	Thu 4/14/22	Thu 4/14/22

Fig. 1: Semester 1 schedule

Semester 2 Schedule

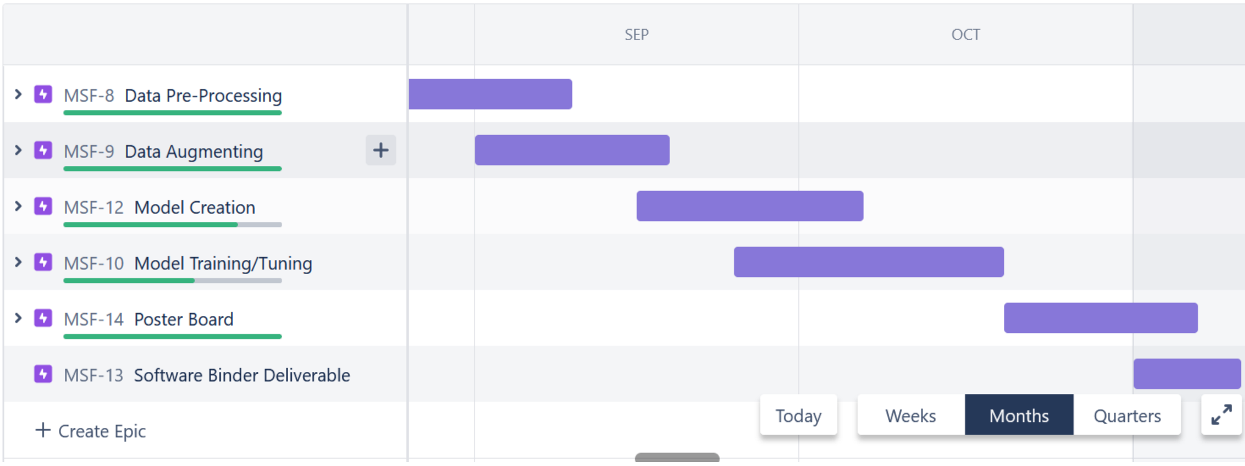


Fig. 2: Semester 2 Schedule

Cost and Risk Analysis

	Impact				A. Data is not cooperating with preprocessing
Risk Probability	3	2	1		B. Need cloud space
3					C. Data generated is inaccurate
2	A	E			D. Inaccurate Kalman-Filter implementation
1	C, D		B		E. Issues with storing data

Fig. 3: cost risk analysis

The only monetary cost in Fig. 3 would be for cloud space, approximately $0.20 for 1GB.

Top Level Design

On the most abstract level, our project took in raw data, and with the code we created initially on Google Collab and on script files using PyTorch, and would produce augmented datasets and logs of both the models themselves and their performances using the online service Weights and Biases

Top Level Diagram

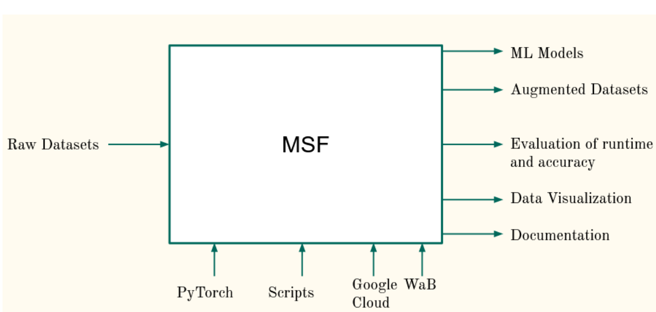


Fig. 4: abstract high level design of project

System Level Design

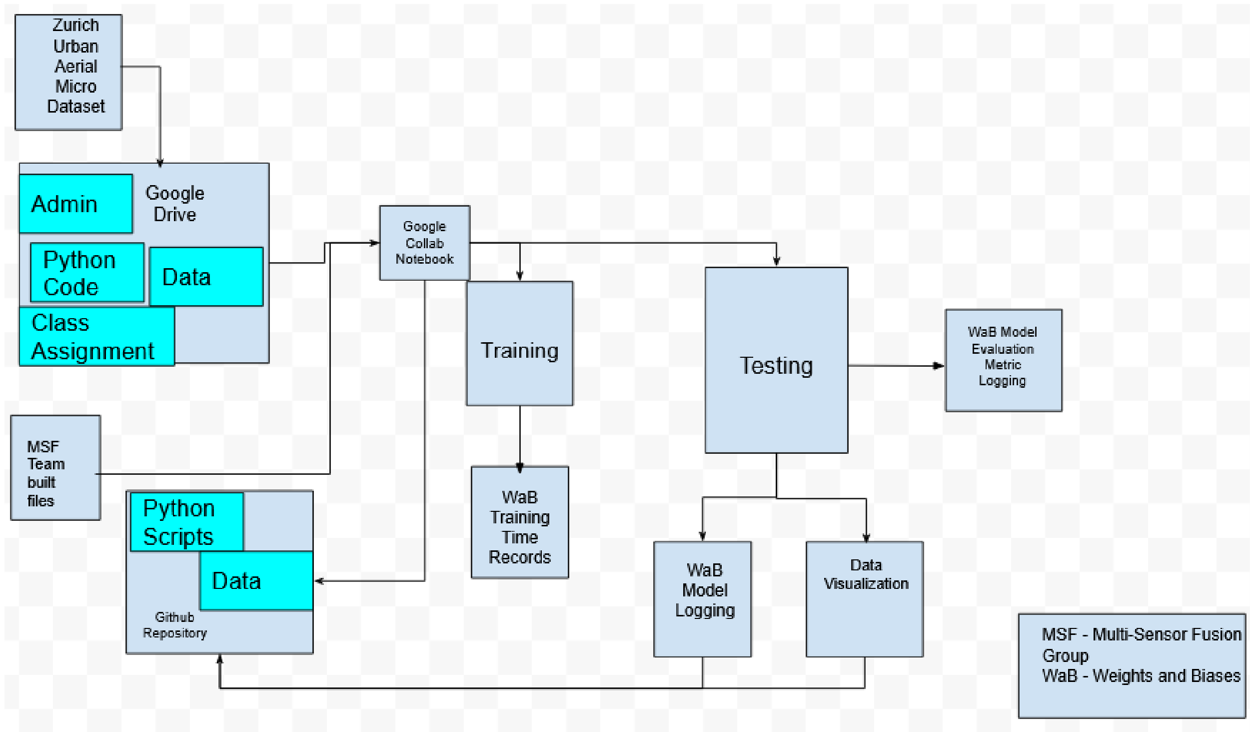


Fig. 5: in-depth system level diagram

Architectural Diagram

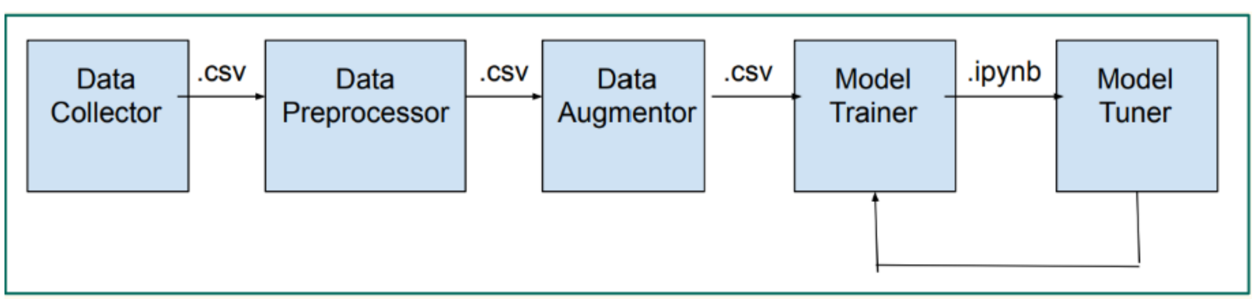


Fig. 6: abstract architectural design graph

Tools and Interfaces

The data, both original and augmented, will be stored as .csv files, both on-machine, in Our Git Repository to facilitate sharing files between members that are meant to be on-machine), and in our shared Google drive. Our work with training and tuning the models would be done initially through a .ipynb file, a Python notebook that could be integrated with our files stored on our drive using Google Collab. By customer request, that notebook would be decomposed into multiple scripts for final delivery

Data Collection

We went through a process of drawing data from a couple of different sources, our customer, SME, Kaggle, etc. Eventually we settled on the Zurich Urban Micro Aerial Vehicle Dataset, a dataset composed of the flight path taken by a drone, with true coordinates verified by images taken by the drone. Along with this path are a number of sensor readings, including GPS and IMU data. The data is, however, somewhat messy, and there are sensors that are irrelevant to our project, thus necessitating preprocessing.

Data Collection Diagram

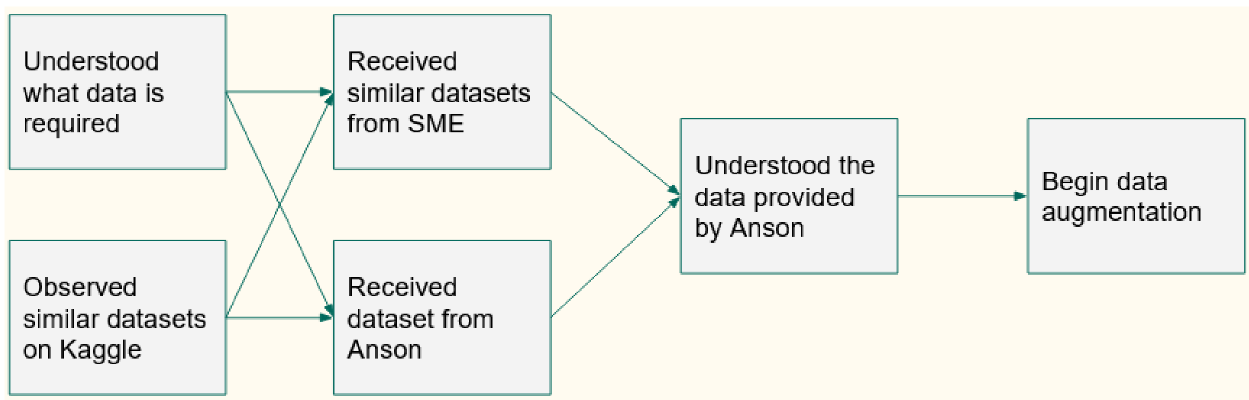


Fig. 7: in-depth chart describing the data collector

Data Preprocessing

The preprocessing implementation Involved extracting the columns of data that we care about (the true ground coordinates, GPS, and IMU data) from the original dataset and storing them in a csv file on our shared drive. The data would ideally be normalized for a machine learning model, so we would set up code for scaling the dataset, and returning the scalars which could be used to reverse the process. The normalized data could be put into the model, and the model predictions would be scaled back.

Data Augmentation

The dataset we will be using is relatively small, which creates the risk of overfitting our models. To solve this problem, we implemented data augmentation techniques. Augmented datasets were generated from the ground truth coordinates of the drone from the Zurich dataset. A set of three transformations were applied in sequence to this path to generate a new path, a linear translation of random length corresponding to the entire path being shifted to a new location, a rotational transformation that spun the path around in space, and a transformation from the open source Tsaug library called timewarp that simulates changes in frequency when using time series data to simulate variations in velocity/acceleration. To simulate the error of the sensors, we first calculated the error between the Zurich GPS and ground truth coordinates. We then got the standard deviation of that error, and applied noise with multiples of that standard deviation to the new path to derive simulated sensor readings to use as the input, with the path itself being the target label.

Data Augmentation Diagram

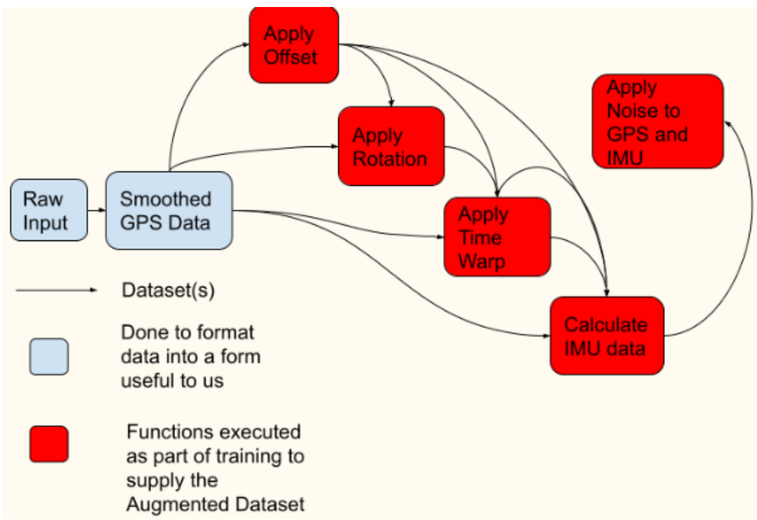


Fig. 8: in-depth chart describing the data augmentation

Model Development

We looked into a couple different kinds of models, Linear Regression, KNN (K Nearest Neighbors), and both “vanilla” RNNs (recurrent neural networks) and LSTMs (Long Short-Term Models), which are a type of RNN. As we studied more, it became clear that RNN model types would be what we used for the final product, they have a “memory” that persists through their application which makes them unanimously acclaimed as the type of model to use with time-series data. Often this is in the context of predicting the future of things like stocks or weather, but predicting a current value given the past predictions and flawed input data about the present works the same, and is what a Kalman filter does, thereby making them the optimal path for development. LSTMs proved better than the simpler kind of RNN, so we further directed our energy there. KNNs however, still held some promise to group together sections of the GPS data based on their precision, to detect major errors in the data. The results of these could then be logged using the service Weights and Biases, except for the KNN which doesn't have learned parameters and just categorizes information based on how it is defined in code.

RNN/LSTM Mechanics

RNNs are neural networks, meaning they are models that have a variable number of "layers", each layer being essentially an equation that the model learns which outputs to the next layer, with the last layer ordinarily being the prediction. RNNs are different in that they store previous predictions and use them as part of the input for the next prediction. LSTMs are RNNs with additional values called cell values stored in memory that are used to make predictions; these additional values are calculated much like the prediction, but remain as hidden values that influence further decisions without needing to correspond with the actual prediction, thereby giving the model an additional capacity to file away values for the future. These models have a number of different hyperparameters, or attributes that define the "shape" of the model, such as the number of layers, the size of each layer, the number of epochs trained, and the learning rate.

RNN/LSTM Diagram

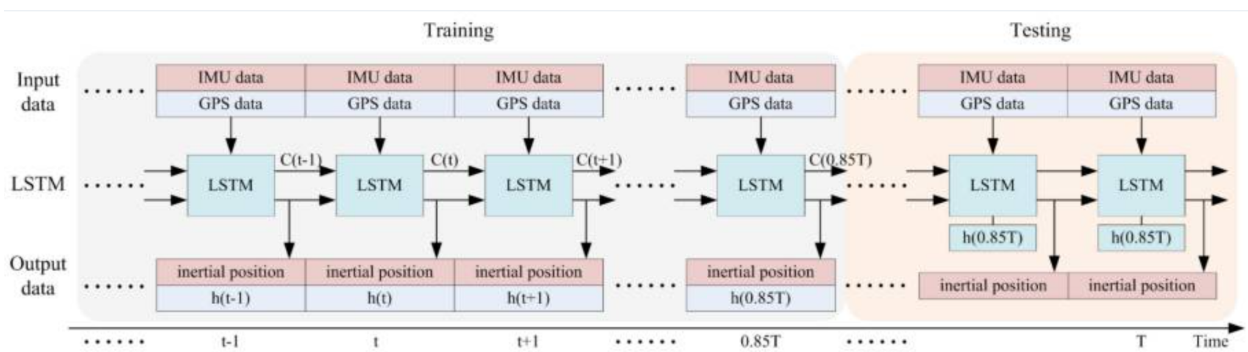


Fig. 9: depiction of an RNN's internal logic

KNN Mechanics

KNN models group data points based on how close the data point is to other, known data points. It gets the distance between the new data point and every known value, and "votes" on the category the new data point should be slotted into based on the K nearest data points, where K is a parameter set by the programmer. This way we would use this kind of model is to identify the points of greatest error in the initial dataset and categorize measurements as being outliers based on clusters of GPS measurements that have more to do with one another than the real path

Training and Tuning

The data for the RNN models will be split into training, testing, and validation data for evaluating model performance, with 30% being used for testing. PyTorch allows for dataloaders that group the data into batches for optimal testing, notably these batches can have their order randomly shuffled. This will allow us to reduce bias and improve accuracy, especially when coupled with the database we can generate. The library Optuna provides code to automate the process of fine tuning a model, it takes in a range of values for each hyperparameter that ought to be varied, and begins training models with every possible combination of those hyperparameters, ending the training sessions for models that are underperforming and not showing substantial improvement early.

Evaluation and Delivery Method

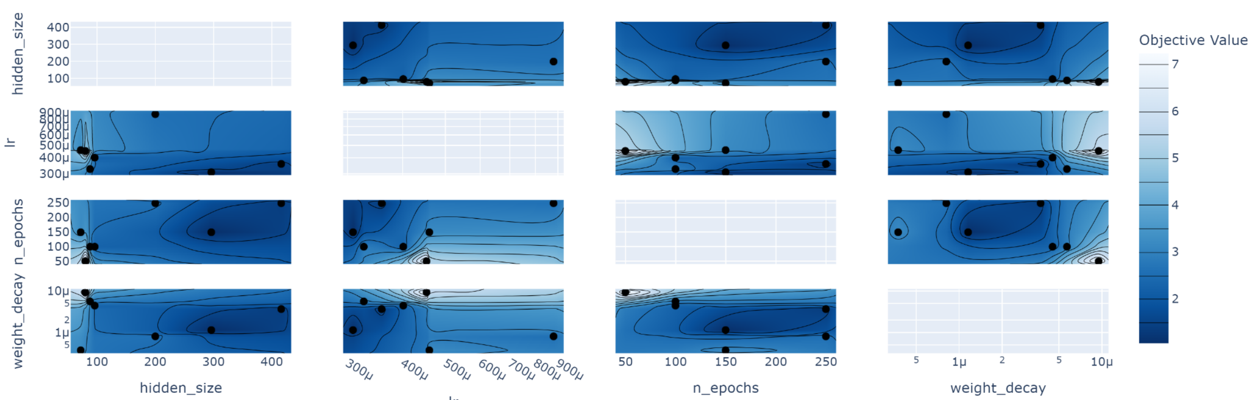
The online service Weights and Biases can be used to both log the models themselves, as well as any data from the training and testing runs that the model was run on. This can include data pulled from Python timers, GPU usage records, and any metrics we want to use; in our case the mean squared error and R2 value. These logs can also include data like the error between the GPS and ground truth coordinates to evaluate the models against. These metrics can then be checked against the requirements and standards set above in this document. These logs and models, which can be shared through W&B, are one key component of our deliverable. We will also provide the notebook that we had written our code on. Following customer request, we will also take the code in the notebook and compartmentalize it into Python scripts that can be run locally on-machine for the final delivery.

Build and Test

Project Status/Results

Test Results & Analysis

The model test results were originally visualized in Google Collab in order to observe the predicted path against the ground truth data. Once the model architecture was well developed, the models began being logged in Weights and Biases so model plots and key hyperparameters could be analyzed. After comparing the RNN and LSTM model outputs, it was determined that LSTM was producing more accurate results. This was concluded by comparing the plotted predicted path as well as calculated metrics such as root mean squared error, mean absolute error, and the R squared score. Furthermore, Weights and Biases stored every model execution and each plot and calculated metrics were able to be visually compared against each other. This included creating contour plots where the hyperparameters were plotted against each other and the performance was able to be visualized for that aspect. An example of the contour plot can be observed below.



The objective value represents the sum of the mean absolute error of the x and y values.

Additionally, by incorporating tools such as Optuna, the hyperparameters have been tuned such that the model can have optimal accuracy performance and predicted positions within one meter of the ground truth values. Below is an example output in which the model's predicted output is in blue and the ground truth values are in red.

Key Accomplishments

The main accomplishment of this project was streamlining a process for tuning an LSTM model in PyTorch with the inclusion of logging and visualization. The project is set up such that modifying what data sets to use for training and validation is easy. Notably this is done by passing a list of strings containing the paths for the files the user would like to use for training and validation. Changing the range of hyperparameters to test for is also trivial. This involves modifying the range of suggestInt functions in the Optuna library. When run, the results are automatically logged into Weights & Biases.

Delivery

Overall Project Performance

We have models that have a good sense of the “shape” of the data, and are able to make predictions with an average error of around 1 meter. However, the average error of the GPS only data is 30 centimeters. Although the model did not outperform the raw GPS data, it was still relatively accurate and precise. Moreover, the implementation of Optuna and Weights and Biases was only added within a recent window before the deadline, meaning that if these tools were discovered earlier on, then more accurate outputs would have likely been produced. Additionally, models, more often than not, can always be improved with time, for example by using the KNN to help with data augmentation by categorizing the areas where the GPS was high and low performing, and using that data to simulate the GPS. Therefore, the model produced during this time frame has the potential to outperform the GPS sensor data. In conclusion, the outcome of the project was a machine learning based model that could accurately predict position, given GPS and IMU data as input. The project could be further enhanced in the future by having the model supplement a Kalman Filter. Regardless, accurate PNT systems are a necessity in modern society and the model produced demonstrates machine learning capabilities when given the right data to learn from.

Customer Satisfaction

The customer had few questions, and positive comments. They liked the transition from notebook to scripts, and after being shown the scripts was approving of the command line commands, as well as of the Optuna plots on Weights and Biases.

Challenges/Issues

The main challenges of the project resulted from inexperience with the frameworks utilized as well as the overall objective. But it was a great experience to be able to learn a highly used tool such as PyTorch in order to develop a machine learning model, as well as be able to gain an understanding of how important navigational system accuracy is. Other challenges included time, mainly regarding model program runtime in Google Collab as after a certain time is reached, Collab will quit the program. This allowed the team to learn how to adjust hyperparameters in order to overcome the obstacle.

Lessons Learned

The main lesson learned was how to develop a machine learning model in a PyTorch framework. PyTorch is one of the most utilized tools in the artificial intelligence industry and if the goal is to work in that field one day, then this project was a major milestone in gaining knowledge for that field. The team was able to fully grasp the effect of hyperparameter tuning and how sensitive a model’s output is according to the data it receives. By appending files to have the model train on larger datasets, it was an immense difference in output compared to a smaller dataset. This allowed the team to understand that the amount of data and the quality of it is most likely the main factor for a model’s success or its downfall. Overall, the project was a terrific learning experience and knowledge was gained that will likely be applicable in the nearby future.

Code Description:

Folder Description:

The following libraries have to be installed on-machine for our code to run: Optuna (for hyperparameter optimization), Tsaug (for time series data augmentation tools), PyTorch Lightning (for sophisticated high-level ML code), Wandb (for accessing the Weights and Biases website), Pyproj (for data visualization).

Our scripts include data_visualization.py for viewing the original dataset, data_preprocessing.py for running the preprocessing to make the data usable, Data_Augmentation.py for producing the augmented datasets, lstm.py for training and logging models.

Read Me as Appears in Folder:

Run the following srcipts from within the 'AI-Multi_Sensor-Fusion/scripts' folder

1. Data visualization

To view the GroundTruthAGL plot you can simply run the following from the scripts folder:
python data_visualization.py
To specify the path you can run the follwing:
python data_visualization.py -path ../datasets/GroundTruthAGL.csv

To view the onBoardGPS plot you need to specify both the path and the type which by default are the following:
python data_visualization.py -path ../datasets/onBoardGPS.csv -type onBoard

Note: These function assumes that the labels for the ground truth and gps positions are as follows ' x_gt',' y_gt',' x_gps',' y_gps', ' lon', ' lat'.
There is an extra space in at the start of the labels

2. Data Preprocessing

The data_preprocessing.py script is used to generate the LR_processed_data.csv file.

The script matches rows from the onBoardGPS.csv file to rows from the GroundTruthAGL.csv
file by their imgid value into one localized LR_processed_data.csv file. The script can be run as follows:

```
python data_preprocessing.py
```

Note: Coordinates are translated from lat, lon to x, y (in the onBoardGPS.csv file) using the pyproj library

3. Data Augmentation

This script runs the data augmentation process. the file 'LR_processed_data.csv' should be in the same file as this, as that file contains the preprocessed data drawn from the dataset.
With no arguements, or a single numerical argument, the script generates a new dataset, plots out the old and new datasets, and asks the user if they want to save it after the plots are closed. The numerical argument is a scaling factor for the noise of the GPS data. Ex:

```
python Data_Augmentation.py 1.2
```

If the argument "view" is passed, followed by a path, the script will display information about the dataset. Ex:

```
python Data_Augmentation.py view c:/datatsets/Training_Data/Training_Set_14.csv
```

Otherwise, if a string argument that is not view is given, it will create a folder with that name and fill it with multiple augmented datasets. If a numerical argument is given after that folder, it will be used as a scale for the intensity of noise used to generate the GPS data, as above. Ex:

```
python Data_Augmentation.py datasets
python Data_Augmentation.py smallError 0.3
python Data_Augmentation.py noisy 1.5
```

4. RNN

The rnn script runs the RNN model. The rnn model was developed in the beginning of the creation process but is mainly still here for original comparison reasons.The LSTM has proven to produce better results so that model is the final product. Running this model will still be helpful for improvement possibilities and visual comparisons. Ex:

```
python rnn.py
```

5. LSTM

The lstm script runs our lstm model and logs the results to WandB. The script will prompt the user to log into wandb or create an account if not already signed in. The -n flag sets the number of trials the user would like optuna to optimize for. In WandB the script generates two projects, MSF which contains numerical information about the run and MSF Optuna which contains visuals about the runs

```
python lstm.py -n 100
```

6. Outlier Detection (low thresholded KNN)

This script plots a comparison of prediction data and true data when given a path. The current model is currently has the outlier quartile range set to 0.75, which means that it will have at most 25% of the data be outliers. Another condition was added that checks if the distance is a minimum distance to be an error. Both of these are adjustable variables based on what the user is attempting to observe. By default it uses the Ground Truth AGL file from the dataset, but if you want to evaluate points of concentrated error in models, give it a path to a model prediction and the parameter -m after said path. Ex:

```
python outlier.py c:/datatsets/GroundTruthAGL.csv
```

References

1. "Ethics and member conduct committee 2021 ... - ieee.org," *IEEE Code of Ethics*, Jun-2020. [Online]. Available: https://www.ieee.org/content/dam/ieee-org/ieee/web/org/about/example-ethics-cases.pdf. [Accessed: 24-Feb-2022].
2. "42030-2019 - ISO/IEC/IEEE international standard - software, Systems and Enterprise -- Architecture Evaluation Framework," *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/document/8767001. [Accessed: 24-Feb-2022].
3. "IEEE 7010: A new standard for assessing the well-being implications of Artificial Intelligence," *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/document/9283454. [Accessed: 24-Feb-2022].
4. "PYTORCH documentation¶," *PyTorch documentation - PyTorch 1.10 documentation*. [Online]. Available: https://pytorch.org/docs/stable/index.html. [Accessed: 25-Feb-2022].
5. L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin and Q. Zhang, "Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond," *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1258-1269, doi: 10.1109/ICDCS.2018.00125.

Appendices (code):

Data Visualization:

```
import sys
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def main():
	args = sys.argv[0:]

	if len(args) == 1:
	plot_original_GTdataset('../datasets/GroundTruthAGL.csv')
	else:
	argc = 1
	path ='../datasets/GroundTruthAGL.csv'
	file = 'GT'

	while argc < len(args):
	if args[argc] == '-path':
		assert(argc + 1 <= len(args) - 1)
		path = args[argc + 1]
	elif args[argc] == '-type':
		assert(argc + 1 <= len(args) - 1)
		file = 'onBoard'
	argc += 1

	assert(file == 'GT' or file == 'onBoard')

	if file == 'GT':
	plot_original_GTdataset(path)
	else:
	plot_original_onBoarddataset(path)


def plot_original_GTdataset(path):

	# read data from .csv in to dataframe
	df = pd.read_csv(path)

	# plot lines
	plt.plot(df[' x_gt'], df[' y_gt'], label = 'Ground Truth Position', color='red')
	plt.plot(df[' x_gps'], df[' y_gps'], label = 'GPS Position', color='blue')
```

```
        plt.title('GroundTruthAGL Plot')
        plt.legend()
        plt.show()

def plot_original_onBoarddataset(path):

        # read data from .csv in to dataframe
        df = pd.read_csv(path)

        # plot line
        plt.plot(df[' lon'], df[' lat'], label = 'Coordinates', color='red')

        plt.title('onBoardGPS Plot')
        plt.legend()
        plt.show()


if __name__ == '__main__':
        main()
```

Data Preprocessing:

```
import pandas as pd
import numpy as np
from pyproj import Proj, transform

# This script takes the onBoardGPS values and maps them to groundTruthAGL
# values (while translating from lat, lon to x, y)
def main():

        # read data from .csv to dataframe
        df_agl = pd.read_csv('../datasets/GroundTruthAGL.csv')
        df_onboardGPS = pd.read_csv('../datasets/OnboardGPS.csv')

        new_df = pd.DataFrame()
        new_df['timestep'] = np.zeros(len(df_agl['imgid']))
        new_df['imgid'] = np.zeros(len(df_agl['imgid']))
        new_df['x_gt'] = np.zeros(len(df_agl['imgid']))
        new_df['y_gt'] = np.zeros(len(df_agl['imgid']))
        new_df['onboard_lat'] = np.zeros(len(df_agl['imgid']))
        new_df['onboard_lon'] = np.zeros(len(df_agl['imgid']))
        new_df['translated_onboard_x'] = np.zeros(len(df_agl['imgid']))
        new_df['translated_onboard_y'] = np.zeros(len(df_agl['imgid']))
```

```
        new_csv(new_df, df_agl, df_onboardGPS)
        latlon_to_xy(new_df)

        new_df.to_csv('../datasets/LR_processed_data.csv')

# Combines relevant data across different .csv's into one .csv in order to have localized data
def new_csv(new_df, df_agl, df_onboardGPS):
  for idx, elem in enumerate(df_agl['imgid']):
        row_gt = df_agl.loc[df_agl['imgid'] == elem].to_numpy()
        row_onboard = df_onboardGPS.loc[df_onboardGPS[' imgid'] == elem].to_numpy()

        if row_onboard.size > 0 and row_gt.size:
        new_df['x_gt'][idx] = row_gt[0][1]
        new_df['y_gt'][idx] = row_gt[0][2]
        new_df['onboard_lat'][idx] = row_onboard[0][2]
        new_df['onboard_lon'][idx] = row_onboard[0][3]
        new_df['timestep'][idx] = row_onboard[0][0]
        new_df['imgid'][idx] = row_onboard[0][1]
        else:
        new_df['x_gt'][idx] = new_df['x_gt'][idx - 1]          # If missing data use previous row
        new_df['y_gt'][idx] = new_df['y_gt'][idx - 1]
        new_df['onboard_lat'][idx] = new_df['onboard_lat'][idx - 1]
        new_df['onboard_lon'][idx] = new_df['onboard_lon'][idx - 1]
        new_df['timestep'][idx] = new_df['timestep'][idx -1]
        new_df['imgid'][idx] = new_df['imgid'][idx - 1]

# Transform coordinates from longitude, latitude to x, y
def latlon_to_xy(new_df):
  outProj = Proj('epsg:32632') # WGS 84 / UTM zone 32N coordinate system
  inProj = Proj('epsg:4326') # Latitude, longitude

  new_df['translated_onboard_x'] = new_df.apply(lambda row: transform(inProj, outProj,
row['onboard_lat'], row['onboard_lon'])[0], axis=1)
  new_df['translated_onboard_y'] = new_df.apply(lambda row: transform(inProj, outProj,
row['onboard_lat'], row['onboard_lon'])[1], axis=1)

if __name__ == '__main__':
        main()
```

Data Augmentation:

```
"""
Created on Thu Nov  3 11:48:26 2022
```

```
@author: fwoff

This script runs the data augmentation process. the file
'LR_processed_data.csv' should be in the same file as this, as that file
cotains the preprocessed data drawn from the dataset.
With no arguements, or a single numerical argument, the script generates a
new dataset, plots out the old and new datasets, and asks the user if they
want to save it after the plots are closed. The numerical argument is a scaling
factor for the noise of the GPS data. If the argument "view" is given,
followed by a path, the script will display information about the dataset.
Otherwise, if an argument that is not view is given, it will create a folder
with that name and fill it with multiple augmented datasets. If a numerical
argument is given after that folder, it will be used as a scale for the
intensity of noise used to generate the GPS data.
"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tsaug as ts
import random
import sys
import os

"""
Functions used for data augmentation
"""
def TranslateCoordinates(coordList):
	xAdd = random.uniform(-3000.0, 3000.0)
	yAdd = random.uniform(-3000.0, 3000.0)
	#we copy the values and add to all of them to create the new coordinate set
	newList = np.copy(coordList)
	newList[:, 0] = newList[:, 0] + xAdd
	newList[:, 1] = newList[:, 1] + yAdd
	return newList #we return the entirety of the list we made

def rotateCoordinateSet(coords):
	#we extract the first coordinate of the set as the origin, which we will rotate around
	origin = coords[0, :]
	angle = np.random.randint(0, 360)
	angle = np.deg2rad(angle)
	R = np.array([[np.cos(angle), -np.sin(angle)],
		[np.sin(angle),  np.cos(angle)]])
	o = np.atleast_2d(origin)
```

```
        p = np.atleast_2d(coords)
        return np.squeeze((R @ (p.T-o.T) + o.T).T)

#function for translation
def TranslateAugmentation(df):
  #create the new dataframe that we will return
  new_df = df.copy(deep = True)
  #extract the ground truth values as a numpy array,
  #pass it through the tranformation function, and convert
  #the output to a dataframe
  gtValuesFrame = df[["x_gt", "y_gt"]]
  gtArray = gtValuesFrame.to_numpy()
  new_gt = TranslateCoordinates(gtArray)
  new_gt_frame = pd.DataFrame(new_gt, columns = ["x_gt","y_gt"])
  #apply the dataframe to the output, and return
  new_df["x_gt"] = new_gt_frame["x_gt"]
  new_df["y_gt"] = new_gt_frame["y_gt"]
  return new_df

#function for rotation
def RotationAugmentation(df):
  #create the new dataframe that we will return
  new_df = df.copy(deep = True)
  #extract the ground truth values as a numpy array,
  #pass it through the tranformation function, and convert
  #the output to a dataframe
  gtValuesFrame = df[["x_gt", "y_gt"]]
  gtArray = gtValuesFrame.to_numpy()
  new_gt = rotateCoordinateSet(gtArray)
  new_gt_frame = pd.DataFrame(new_gt, columns = ["x_gt","y_gt"])
  #apply the dataframe to the output, and return
  new_df["x_gt"] = new_gt_frame["x_gt"]
  new_df["y_gt"] = new_gt_frame["y_gt"]
  return new_df

#function for time warp
def TimeWarpAugmentation(df):
  new_df = df.copy(deep = True)
  warp_number = random.randint(1, 12)
  gtValuesFrame = df[["x_gt", "y_gt"]]
  gtArray = gtValuesFrame.to_numpy()
  #Where T is the number of elements in each series, and N is the
  #number of series', tsaug's augmentation takes in an array of
  #shape (N,T), while our data is organized columnwise (T, N),
```

```
  #so we reshape before and after applying noise
  T, N = gtArray.shape
  gtArray.reshape((N, T))
  ts.TimeWarp(n_speed_change=warp_number).augment(gtArray)
  gtArray.reshape((T, N))
  new_gt_frame = pd.DataFrame(gtArray, columns = ["x_gt","y_gt"])
  new_df["x_gt"] = new_gt_frame["x_gt"]
  new_df["y_gt"] = new_gt_frame["y_gt"]
  return new_df

#function to get acceleration data, that would be measured by an IMU
def GetIMU(df):
  new_df = df.copy(deep = True)
  gt_x_frame = df[["x_gt"]]
  gt_y_frame = df[["y_gt"]]
  gtx = gt_x_frame.to_numpy()
  gty = gt_y_frame.to_numpy()
  #double derivarives for accelerometer data
  acc_x = np.diff(gtx, n=2, axis=0)
  acc_y = np.diff(gty, n=2, axis=0)
  #getting the angular velocity by taking the derivative of the array of angles
  fractions = gty / gtx
  angles = np.arctan(fractions)
  ang_v = np.diff(angles, axis=0)
  #since the derivative for the first two instances don't have accelerometer
  #data, we remove them, as well as the first angular velocity
  #instance, then add the data
  gtValuesFrame = df[["x_gt", "y_gt"]]
  gtArray = gtValuesFrame.to_numpy()
  new_gt_frame = pd.DataFrame(gtArray[2:,:], columns = ["x_gt","y_gt"])
  new_df["x_gt"] = new_gt_frame["x_gt"]
  new_df["y_gt"] = new_gt_frame["y_gt"]
  final_ang_v = ang_v[1:]
  final_ang_v = np.reshape(final_ang_v,(final_ang_v.size, 1))
  new_x_frame = pd.DataFrame(acc_x, columns = ["acc_x"])
  new_y_frame = pd.DataFrame(acc_y, columns = ["acc_y"])
  new_ang_frame = pd.DataFrame(final_ang_v, columns = ["ang_v"])
  new_df["x_acc_true"] = new_x_frame["acc_x"]
  new_df["y_acc_true"] = new_y_frame["acc_y"]
  new_df["ang_v"] = new_ang_frame["ang_v"]
  return new_df


#function to derive sensor data through noise
```

```
#In Progress
def SensorData(df, std, scaling=1/2):
  #create the new dataframe that we will return
  new_df = df.copy(deep = True)
  #extract the ground truth values and true acceleration
  #values as a numpy array
  gtValuesFrame = df[["x_gt", "y_gt"]]
  array = gtValuesFrame.to_numpy()
  #Where T is the number of elements in each series, and N is the
  #number of series', tsaug's augmentation takes in an array of
  #shape (N,T), while our data is organized columnwise (T, N),
  #so we reshape before and after applying noise
  T, N = array.shape
  array.reshape((N, T))
  #by default, AddNoise calculates 0 mean gaussian noise (white noise)
  #independantly for each series.
  gt_sensor_readings = ts.AddNoise(scale=(std*scaling), normalize=False).augment(array)
  gt_sensor_readings.reshape((T, N))
  new_gt_frame = pd.DataFrame(gt_sensor_readings, columns = ["x_gps","y_gps"])
  #apply the dataframe to the output, and return
  new_df["x_gps"] = new_gt_frame["x_gps"]
  new_df["y_gps"] = new_gt_frame["y_gps"]
  imuValuesFrame = df[["x_acc_true", "y_acc_true"]]
  array = imuValuesFrame.to_numpy()
  T, N = array.shape
  array.reshape((N, T))
  acc_sensor_readings = ts.AddNoise().augment(array)
  acc_sensor_readings.reshape((T, N))
  new_acc_frame = pd.DataFrame(acc_sensor_readings, columns = ["IMU_x","IMU_y"])
  new_df["IMU_x"] = new_acc_frame["IMU_x"]
  new_df["IMU_y"] = new_acc_frame["IMU_y"]
  #similar process for angular velocity
  gyroValuesFrame = df[["ang_v"]]
  array = gyroValuesFrame.to_numpy()
  T, N = array.shape
  array.reshape((N, T))
  gyro_sensor_readings = ts.AddNoise().augment(array)
  gyro_sensor_readings.reshape((T, N))
  new_gyro_frame = pd.DataFrame(gyro_sensor_readings, columns = ["gyro"])
  new_df["gyro"] = new_gyro_frame["gyro"]
  return new_df

def GetAugmentedDataset(df, scale=1/2, translate = False, rotate = False, warp = False, save =
False, file_name = ''):
```

```
  new_df = df.copy(deep = True)
  #calculating the standard deviation of the error of the sensors from the dataset
  difference_x = df[["x_gt"]].to_numpy() - df[["translated_onboard_x"]].to_numpy()
  difference_y = df[["y_gt"]].to_numpy() - df[["translated_onboard_y"]].to_numpy()
  std_x = np.std(difference_x)
  std_y = np.std(difference_y)
  std = (std_x + std_y)/2


  #apply whatever data tranformations have been selected
  if(translate):
      new_df = TranslateAugmentation(new_df)
  if(rotate):
      new_df = RotationAugmentation(new_df)
  if(warp):
      new_df = TimeWarpAugmentation(new_df)

  #calculate simulated IMU and sensor data
  new_df = GetIMU(new_df)
  new_df = SensorData(new_df, std, scaling=scale)

  #get rid of empty rows from previous step
  new_df['x_gt'].replace('', np.nan, inplace=True)
  new_df.dropna(subset=['x_gt'], inplace=True)

  #if the new dataset is meant to be saved somewhere, do so, then return
  if(save and (file_name != '')):
      new_df.to_csv(file_name)
  return new_df

def isfloat(num):
    try:
    float(num)
    return True
    except ValueError:
    return False

# read data from .csv to dataframe
df = pd.read_csv('LR_processed_data.csv')
scaling = 1/2
makeWithScale = False
if len(sys.argv) == 2:
    if isfloat(sys.argv[1]):
    scaling = float(sys.argv[1])
```

```
        makeWithScale = True
elif len(sys.argv) > 2:
        if isfloat(sys.argv[2]):
        scaling = float(sys.argv[2])

if len(sys.argv) == 1 or makeWithScale:
        # Create plot from df
        df.plot( x = 'translated_onboard_x', y = 'translated_onboard_y', kind = 'line', label = 'GPS')
        df.plot( x = 'x_gt', y = 'y_gt', kind = 'line', label = 'Ground Truth')
        df_new = GetAugmentedDataset(df, translate=True, rotate=True, warp=True,
scale=scaling)
        df_new.plot( x = 'x_gps', y = 'y_gps', kind = 'line', label = 'New GPS Readings')
        df_new.plot( x = 'x_gt', y = 'y_gt', kind = 'line', label = 'New Ground Truth')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()
        saveState = " "
        while saveState != "n" and saveState != "y":
        saveState = input("Do you want to save the new dataset? (y/n) ")
        if saveState == "y":
        newName = input("Type the file name, leaving out the .csv ") + ".csv"
        df_new.to_csv(newName)
elif sys.argv[1] != "view":
        if not os.path.exists(sys.argv[1]):
        os.makedirs(sys.argv[1] + "/Training_Data")
        os.makedirs(sys.argv[1] + "/Validation_Data")
        trainingBase = sys.argv[1] + "/Training_Data/Training_Set_"
        validationBase = sys.argv[1] + "/Validation_Data/Validation_Set_"
        for i in range(24):
        if i > 13:
        trainingName = trainingBase + str(i) + ".csv"
        newData = GetAugmentedDataset(df, scale=scaling, translate=True, rotate=True,
warp=True, save=True, file_name=trainingName)
        for j in range(14):
        if j > 6:
        validationName = validationBase + str(j) + ".csv"
        newData = GetAugmentedDataset(df, translate=True, rotate=True, warp=True, save=True,
file_name=validationName)
elif len(sys.argv) >= 3:
        path = sys.argv[2]
        df = pd.read_csv(path)
        df.plot( x = 'x_gt', y = 'y_gt', kind = 'line', label = 'Ground Truth')
        df.plot( x = 'x_gps', y = 'y_gps', kind = 'line', label = 'GPS Readings')
        fig, (ax1, ax2, ax3, ax4, ax5, ax6) = plt.subplots(6, 1)
```

```
ax1.plot(df['Unnamed: 0'], df['x_acc_true'], label = 'True X Acceleration', color = 'red')
plt.xlabel('Time Step')
plt.ylabel('Acceleration')
ax2.plot(df['Unnamed: 0'], df['y_acc_true'], label = 'True Y Acceleration', color = 'blue')
plt.xlabel('Time Step')
plt.ylabel('Acceleration')
ax3.plot(df['Unnamed: 0'], df['ang_v'], label = 'True Angular Acceleration', color = 'blue')
plt.xlabel('Time Step')
plt.ylabel('Velocity')
ax4.plot(df['Unnamed: 0'], df['IMU_x'], label = 'IMU X Readings', color = 'yellow')
plt.xlabel('Time Step')
plt.ylabel('Acceleration')
ax5.plot(df['Unnamed: 0'], df['IMU_y'], label = 'IMU Y Readings', color = 'purple')
plt.xlabel('Time Step')
plt.ylabel('Acceleration')
ax6.plot(df['Unnamed: 0'], df['gyro'], label = 'Gyro Readings', color = 'purple')
plt.xlabel('Time Step')
plt.ylabel('Velocity')
plt.show()
```

RNN:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import pytorch_lightning as pl
import matplotlib.pyplot as plt
import torchmetrics
import torch.optim as optim
import sys

from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer
from torch.utils.data import TensorDataset, DataLoader
from tqdm import tqdm
from sklearn import preprocessing
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Note: The RNN functions were taken and built upon from the example provided by Kaan
Kuguoglu at towardsdatascience.com
```

```
#
https://towardsdatascience.com/building-rnn-lstm-and-gru-for-time-series-using-pytorch-a46e5b09
4e7b

# ---------------------------------------------------------------------------------------------------
# Module and Trainer classes defined below
# ---------------------------------------------------------------------------------------------------

# A class for a simple PyTorch RNN Model
class RNN_Model(nn.Module):
	def __init__(self, input_size, hidden_dim, n_layers, output_size):
	super(RNN_Model, self).__init__()

	# Defining the number of layers and the nodes in each layer
	self.hidden_dim = hidden_dim
	self.layer_dim = n_layers

	# RNN layers
	self.rnn = nn.rnn(
	input_size, hidden_dim, n_layers, batch_first=True)

	# Fully connected layer
	self.fc = nn.Linear(hidden_dim, output_size)

	def forward(self, x):

	# Initializing hidden state for first input using method defined below
	h0 = torch.zeros(self.n_layers, x.size(0), self.hidden_dim).requires_grad_()

	# Passing in the input and hidden state into the model and obtaining outputs
	out, h0 = self.rnn(x, h0.detach())

	# Reshaping the outputs such that it can be fit into the fully connected layer
	out = out[:, -1, :]

	# Convert the final state to our desired output shape (batch_size, output_dim)
	out = self.fc(out)

	return out


# A class to train a PyTorch RNN model
class Trainer(pl.LightningModule):
	def __init__(self, model, loss_fn, optimizer):
```

```
super(Trainer, self).__init__()
self.model = model
self.loss_fn = loss_fn
self.optimizer = optimizer
self.train_losses = []
self.val_losses = []

def train_step(self, x, y):
# Sets model to train mode
self.model.train()

# Makes predictions
yhat = self.model(x)

# Computes loss
loss = self.loss_fn(y, yhat)

# Computes gradients
loss.backward()

# Updates parameters and zeroes gradients
self.optimizer.step()
self.optimizer.zero_grad()

# Returns the loss
return loss.item()

def train(self, trial, train_loader, val_loader, batch_size, n_epochs, n_features=1):

for epoch in range(1, n_epochs + 1):
batch_losses = []
for x_batch, y_batch in train_loader:
        x_batch = x_batch.view([batch_size, -1, n_features])
        y_batch = y_batch
        loss = self.train_step(x_batch, y_batch)
        batch_losses.append(loss)
training_loss = np.mean(batch_losses)
self.train_losses.append(training_loss)

with torch.no_grad():
        batch_val_losses = []
        for x_val, y_val in val_loader:
        x_val = x_val.view([batch_size, -1, n_features])
        y_val = y_val
```

```
            self.model.eval()
            yhat = self.model(x_val)
            val_loss = self.loss_fn(y_val, yhat).item()
            batch_val_losses.append(val_loss)

            validation_loss = np.mean(batch_val_losses)
            self.val_losses.append(validation_loss)

    if (epoch <= 10) | (epoch % 50 == 0):
            print(
            f"[{epoch}/{n_epochs}] Training loss: {training_loss:.8f}\t Validation loss:
{validation_loss:.8f}"
            )


    def evaluate(self, test_loader, batch_size, n_features):
    with torch.no_grad():
    predictions = []
    values = []
    for x_test, y_test in test_loader:
            x_test = x_test.view([batch_size, -1, n_features])
            y_test = y_test
            self.model.eval()
            yhat = self.model(x_test)
            predictions.append(yhat.detach().numpy())
            values.append(y_test.detach().numpy())

    return predictions, values

    def plot_losses(self):
    plt.plot(self.train_losses, label="Training loss")
    plt.plot(self.val_losses, label="Validation loss")
    plt.legend()
    plt.title("Losses")
    plt.show()
    plt.close()



# ----------------------------------------------------------------------------------------------------
# Helper functions defined below
# ----------------------------------------------------------------------------------------------------
```

```
# functions (1 of 2) to get data stored in the drive into the dummy csv file
def new_csv(df, df_append_list=[]):
  new_df = df[['x_gt', 'y_gt', 'x_gps', 'y_gps', 'IMU_x', 'IMU_y', 'gyro']]
  new_df.to_csv('../datasets/rnn_dummy.csv', index=False)

  if len(df_append_list) != 0:
        for _df in df_append_list:
        append_df = _df[['x_gt', 'y_gt', 'x_gps', 'y_gps', 'IMU_x', 'IMU_y', 'gyro']]
        append_df.to_csv('../datasets/rnn_dummy.csv', mode='a', index=False, header=False)

# functions (2 of 2) to get data stored in the drive into the dummy csv file
def prepareDataset(N, training, files_to_append):
  if training:
        folder = 'Training_Data/Training_Set_'
        folderType = 'Training Data #'
  else:
        folder = 'Validation_Data/Validation_Set_'
        folderType = 'Validation Data #'

  path = '../datasets/' + folder + str(N) + '.csv'
  df = pd.read_csv(path)

  dfs_to_append = []
  if len(files_to_append) != 0:
        for file_path in files_to_append:
        dfs_to_append.append(pd.read_csv(file_path))

  new_csv(df, dfs_to_append)



# Gets the dataset in the current rnn dummy file, and returns a tuple
# containing the scaled dataframe, the original, and the scalars in the order
# x1, x2, x3, x4, x5, y1, y2
def readData():
        # read data from .csv to dataframe
        df_rnn = pd.read_csv('../datasets/rnn_dummy.csv')
        #df_rnn.drop(df_rnn.columns[[0]], axis=1, inplace=True)
        # Create a copy of the raw .csv file the RNN model will use (don't want to overwrite original
values)
        rnn_df_scaled = df_rnn.copy()
        # Scalers for all the features (independent of each other)
        x1_scaler = preprocessing.MinMaxScaler() # GPS X coordinate scalar
        x2_scaler = preprocessing.MinMaxScaler() # GPS Y coordinate scalar
```

```
        x3_scaler = preprocessing.MinMaxScaler() # Accelerometer X scalar
        x4_scaler = preprocessing.MinMaxScaler() # Accelerometer Y scalar
        x5_scaler = preprocessing.MinMaxScaler() # Gyro angular velocity reading scalar
        # Scalers for the outputs x,y (independent of each other)
        y1_scaler = preprocessing.MinMaxScaler() # x_gt scaler
        y2_scaler = preprocessing.MinMaxScaler() # y_gt scaler
        # Normalize X and Y of the RNN data to have values between 0 and 1
        rnn_df_scaled['x_gt'] = y1_scaler.fit_transform(df_rnn['x_gt'].to_numpy().reshape(-1, 1))
        rnn_df_scaled['y_gt'] = y2_scaler.fit_transform(df_rnn['y_gt'].to_numpy().reshape(-1, 1))
        # X (features)
        rnn_df_scaled['x_gps'] = x1_scaler.fit_transform(df_rnn['x_gps'].to_numpy().reshape(-1, 1))
        rnn_df_scaled['y_gps'] = x2_scaler.fit_transform(df_rnn['y_gps'].to_numpy().reshape(-1, 1))
        rnn_df_scaled['IMU_x'] = x3_scaler.fit_transform(df_rnn['IMU_x'].to_numpy().reshape(-1,
1))
        rnn_df_scaled['IMU_y'] = x4_scaler.fit_transform(df_rnn['IMU_y'].to_numpy().reshape(-1,
1))
        rnn_df_scaled['gyro'] = x5_scaler.fit_transform(df_rnn['gyro'].to_numpy().reshape(-1, 1))

        return (rnn_df_scaled, df_rnn, x1_scaler, x2_scaler, x3_scaler, x4_scaler, x5_scaler,
y1_scaler, y2_scaler)




# Takes in the scaled and unscaled dataset, getting the features and labels of
# each instance in a tuple, and returning a tuple of tuples
def splitData(rnn_df_scaled, df_rnn, train_size=1624, val_size=541, test_size=541):
        train_data_scaled = rnn_df_scaled[:train_size]
        val_data_scaled = rnn_df_scaled[train_size : train_size + val_size]
        test_data_scaled = rnn_df_scaled[train_size + val_size : ]

        X_train_scaled = train_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_train_scaled = train_data_scaled.filter(['x_gt','y_gt'], axis=1)
        train_tuple_scaled = (X_train_scaled, Y_train_scaled)
        X_test_scaled = test_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_test_scaled = test_data_scaled.filter(['x_gt','y_gt'], axis=1)
        test_tuple_scaled = (X_test_scaled, Y_test_scaled)
        X_val_scaled = val_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_val_scaled = val_data_scaled.filter(['x_gt','y_gt'], axis=1)
        val_tuple_scaled = (X_val_scaled, Y_val_scaled)

        train_data = df_rnn[:train_size]
        val_data = df_rnn[train_size : train_size + val_size]
```

```
        test_data = df_rnn[train_size + val_size : ]

        X_train_raw = train_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_train_raw = train_data.filter(['x_gt','y_gt'], axis=1)
        train_tuple = (X_train_raw, Y_train_raw)
        X_test_raw = test_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_test_raw = test_data.filter(['x_gt','y_gt'], axis=1)
        test_tuple = (X_test_raw, Y_test_raw)
        X_val_raw = val_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_val_raw = val_data.filter(['x_gt','y_gt'], axis=1)
        val_tuple = (X_val_raw, Y_val_raw)

        return (train_tuple_scaled, test_tuple_scaled, val_tuple_scaled, train_tuple, test_tuple,
val_tuple)
```

```
# Takes in a set of 3 tuples containing the scaled inputs and splits the data
# into randomly shuffled batches, returned  in a tuple of train, validation,
# test, and unshuffled test
def dataLoading(training_data, validation_data, testing_data, batch_size=24):
        train_features = torch.Tensor(training_data[0].to_numpy())
        train_targets = torch.Tensor(training_data[1].to_numpy())

        test_features = torch.Tensor(testing_data[0].to_numpy())
        test_targets = torch.Tensor(testing_data[1].to_numpy())

        val_features = torch.Tensor(validation_data[0].to_numpy())
        val_targets = torch.Tensor(validation_data[1].to_numpy())

        train = TensorDataset(train_features, train_targets)
        val = TensorDataset(val_features, val_targets)
        test = TensorDataset(test_features, test_targets)

        train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, drop_last=True)
        val_loader = DataLoader(val, batch_size=batch_size, shuffle=True, drop_last=True)
        test_loader = DataLoader(test, batch_size=batch_size, shuffle=True, drop_last=True)
        test_loader_one = DataLoader(test, batch_size=1, shuffle=False, drop_last=True)
        return (train_loader, val_loader, test_loader, test_loader_one)
```

```
# Functions to transform the predictions back to original scale
def inverse_transform(df, y1_scaler, y2_scaler):
	df_gt = df.filter(['x_gt','y_gt'], axis=1)
	df_pred = df.filter(['x_pred','y_pred'], axis=1)

	arr_x_gt = y1_scaler.inverse_transform(df_gt['x_gt'].to_numpy().reshape(-1 ,1))
	arr_y_gt = y2_scaler.inverse_transform(df_gt['y_gt'].to_numpy().reshape(-1 ,1))

	arr_x_pred = y1_scaler.inverse_transform(df_pred['x_pred'].to_numpy().reshape(-1 ,1))
	arr_y_pred = y2_scaler.inverse_transform(df_pred['y_pred'].to_numpy().reshape(-1 ,1))

	df_gt['x_gt'] = arr_x_gt
	df_gt['y_gt'] = arr_y_gt

	df_pred['x_pred'] = arr_x_pred
	df_pred['y_pred'] = arr_y_pred

	return pd.concat([df_gt, df_pred], axis=1)


# Formats the prediction results
def format_predictions(predictions, values, df_test, y1_scaler, y2_scaler):
	vals = np.concatenate(values, axis=0)
	preds = np.concatenate(predictions, axis=0)
	df_result = pd.DataFrame(data={'x_gt': vals[:,0], 'y_gt': vals[:,1], 'x_pred': preds[:,0],
'y_pred': preds[:,1]}, index=df_test.head(len(vals)).index)
	df_result = df_result.sort_index()
	df_result = inverse_transform(df_result, y1_scaler, y2_scaler)
	return df_result


# Function that calculates metrics
def calculate_metrics(df):
	return {
	'mae_x' : mean_absolute_error(df.x_gt, df.x_pred),
	'rmse_x' : mean_squared_error(df.x_gt, df.x_pred) ** 0.5,
	'r2_x' : r2_score(df.x_gt, df.x_pred),
	'mae_y' : mean_absolute_error(df.y_gt, df.y_pred),
	'rmse_y' : mean_squared_error(df.y_gt, df.y_pred) ** 0.5,
	'r2_y' : r2_score(df.y_gt, df.y_pred)
	}
def gps_metrics(df):
```

```
        return {
        'mae_x' : mean_absolute_error(df.x_gt, df.x_gps),
        'rmse_x' : mean_squared_error(df.x_gt, df.x_gps) ** 0.5,
        'r2_x' : r2_score(df.x_gt, df.x_gps),
        'mae_y' : mean_absolute_error(df.y_gt, df.y_gps),
        'rmse_y' : mean_squared_error(df.y_gt, df.y_gps) ** 0.5,
        'r2_y' : r2_score(df.y_gt, df.y_gps)
        }



def plotResults(df_rnn, df_result):

        plt.figure(figsize=(12, 8))
        for row in df_result.iterrows():
        plt.scatter(row[1][0], row[1][1], color = 'r', s = 12)
        plt.scatter(row[1][2], row[1][3], color = 'b', s = 12)

        plt.show()



def trainModelStartToFinish(trial, lr, epochs, num_layers, hidden_size, append_dataset_list=[],
N=0, display=True):
        batch_size = 1024
        n_epochs = epochs
        #n_epochs = 100
        # learning_rate = 1e-4
        learning_rate = lr
        #weight_decay = weight_decay
        n_features=5

        prepareDataset(N, True, append_dataset_list)
        processedData = readData()

        rnn_df_scaled = processedData[0]
        df_rnn = processedData[1]
        y1_scaler = processedData[7]
        y2_scaler = processedData[8]
        #big_tuple = splitData(rnn_df_scaled, df_rnn)

        #manually change the sizes
        #big_tuple = splitData(rnn_df_scaled, df_rnn, train_size=13530, val_size=5412,
test_size=5412)
```

```
        big_tuple = splitData(rnn_df_scaled, df_rnn, train_size=1624, val_size=541, test_size=541)
        training = big_tuple[0]
        testing = big_tuple[1]
        validation = big_tuple[2]
        X_test_scaled = testing[0]

        train_loader, val_loader, test_loader, test_loader_one = dataLoading(training, validation,
testing)
        model = RNN_Model(input_size=n_features, output_size=2, hidden_dim=hidden_size,
n_layers=num_layers)
        loss_fn = nn.MSELoss(reduction="mean")
        optimizer = optim.Adam(model.parameters(), lr=learning_rate)
        opt = Trainer(model=model, loss_fn=loss_fn, optimizer=optimizer)
        opt.train(trial, train_loader, val_loader, batch_size=batch_size, n_epochs=n_epochs,
n_features=n_features)

        predictions, values = opt.evaluate(test_loader_one, batch_size=1, n_features=n_features)
        df_result = format_predictions(predictions, values, X_test_scaled,  y1_scaler, y2_scaler)
        result_metrics = calculate_metrics(df_result)
        result_metrics_gps = gps_metrics(df_rnn)
        opt.plot_losses()
        plotResults(df_rnn, df_result)

        mae_x = result_metrics['mae_x']
        mae_y = result_metrics['mae_y']
        rmse_x = result_metrics['rmse_x']
        rmse_y = result_metrics['rmse_y']
        r2_x = result_metrics['r2_x']
        r2_y = result_metrics['r2_y']

        mae_x_gps = result_metrics_gps['mae_x']
        mae_y_gps = result_metrics_gps['mae_y']
        rmse_x_gps = result_metrics_gps['rmse_x']
        rmse_y_gps = result_metrics_gps['rmse_y']
        r2_x_gps = result_metrics_gps['r2_x']
        r2_y_gps = result_metrics_gps['r2_y']

        print(f"MAE X: {result_metrics['mae_x']}\t MAE Y: {result_metrics['mae_y']}")

        return (model, result_metrics)
```

```
def run_model(trial=0,lr=0.00008, epochs=100, num_layers=1, hidden_size=256,
append_dataset_list=[], N=0, display=True):
        return trainModelStartToFinish(trial=trial, lr=lr, epochs=epochs, num_layers=num_layers,
        hidden_size=hidden_size, append_dataset_list=append_dataset_list, N=N,
display=display)



def main():
        args = sys.argv[0:]
        argc = 1
        run_model(trial=0,lr=0.00008, epochs=100, num_layers=1, hidden_size=256,
append_dataset_list=[], N=0, display=True)

if __name__ == '__main__':
        main()
```

LSTM:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import pytorch_lightning as pl
import matplotlib.pyplot as plt
import torchmetrics
import torch.optim as optim
import wandb
import optuna
import sys

from optuna.visualization import plot_contour
from optuna.visualization import plot_edf
from optuna.visualization import plot_intermediate_values
from optuna.visualization import plot_optimization_history
from optuna.visualization import plot_parallel_coordinate
from optuna.visualization import plot_param_importances
from optuna.visualization import plot_slice
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer
from torch.utils.data import TensorDataset, DataLoader
from tqdm import tqdm
from sklearn import preprocessing
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Note: The LSTM functions were taken and built upon from the example provided by Kaan
Kuguoglu at towardsdatascience.com
#
https://towardsdatascience.com/building-rnn-lstm-and-gru-for-time-series-using-pytorch-a46e5b09
4e7b


# -----------------------------------------------------------------------------------------------------
# Module and Trainer classes defined below
# -----------------------------------------------------------------------------------------------------

# A class for a simple PyTorch LSTM Model
class LSTM_Model(nn.Module):
	def __init__(self, input_size, hidden_dim, n_layers, output_size):
	super(LSTM_Model, self).__init__()

	# Defining the number of layers and the nodes in each layer
	self.hidden_dim = hidden_dim
	self.layer_dim = n_layers

	# LSTM layers
	self.lstm = nn.LSTM(
	input_size, hidden_dim, n_layers, batch_first=True)

	# Fully connected layer
	self.fc = nn.Linear(hidden_dim, output_size)

	# Sigmoid activation layer
	#self.sigmoid = nn.Sigmoid()

	def forward(self, x):
	# Initializing hidden state for first input with zeros
	h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

	# Initializing cell state for first input with zeros
	c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

	# We need to detach as we are doing truncated backpropagation through time (BPTT)
	# If we don't, we'll backprop all the way to the start even after going through another batch
	# Forward propagation by passing in the input, hidden state, and cell state into the model
	out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
```

```
        # Reshaping the outputs in the shape of (batch_size, seq_length, hidden_size)
        # so that it can fit into the fully connected layer
        out = out[:, -1, :]

        # Convert the final state to our desired output shape (batch_size, output_dim)
        out = self.fc(out)

        #sigmoid_out = self.sigmoid(out)

        #return sigmoid_out
        return out


# A class to train a PyTorch LSTM model
class Trainer(pl.LightningModule):
        def __init__(self, model, loss_fn, optimizer):
        super(Trainer, self).__init__()
        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.train_losses = []
        self.val_losses = []
        # log hyperparameters
        #self.save_hyperparameters(ignore=['loss_fn', 'model'])

        def train_step(self, x, y):
        # Sets model to train mode
        self.model.train()

        # Makes predictions
        yhat = self.model(x)

        # Computes loss
        loss = self.loss_fn(y, yhat)

        # Computes gradients
        loss.backward()

        # Updates parameters and zeroes gradients
        self.optimizer.step()
        self.optimizer.zero_grad()

        # Returns the loss
        return loss.item()
```

```
def train(self, trial, train_loader, val_loader, batch_size, n_epochs, n_features=1):

for epoch in range(1, n_epochs + 1):
batch_losses = []
for x_batch, y_batch in train_loader:
    x_batch = x_batch.view([batch_size, -1, n_features])
    y_batch = y_batch
    loss = self.train_step(x_batch, y_batch)
    batch_losses.append(loss)
training_loss = np.mean(batch_losses)
self.train_losses.append(training_loss)

with torch.no_grad():
    batch_val_losses = []
    for x_val, y_val in val_loader:
    x_val = x_val.view([batch_size, -1, n_features])
    y_val = y_val
    self.model.eval()
    yhat = self.model(x_val)
    val_loss = self.loss_fn(y_val, yhat).item()
    batch_val_losses.append(val_loss)

    validation_loss = np.mean(batch_val_losses)
    self.val_losses.append(validation_loss)

    intermediate_value = 1.0 - validation_loss
    trial.report(intermediate_value, epoch)

    # Handle pruning based on the intermediate value.
    if trial.should_prune():
    raise optuna.TrialPruned()

#wandb.log({"Train_loss": {"training loss": training_loss}})
wandb.log(data={"Validation Loss": validation_loss}, step=epoch)
wandb.log(data={"Training Loss": training_loss}, step=epoch)

if (epoch <= 10) | (epoch % 50 == 0):
    print(
    f"[{epoch}/{n_epochs}] Training loss: {training_loss:.8f}\t Validation loss:
{validation_loss:.8f}"
    )
    # wandb.log({"Train_loss": {"training loss": training_loss}})
    # wandb.log({"Val_loss": {"validation loss": validation_loss}})
```

```
        #wandb.run.summary["Final Validation Loss"] = validation_loss

        def evaluate(self, test_loader, batch_size, n_features):
        with torch.no_grad():
        predictions = []
        values = []
        for x_test, y_test in test_loader:
                x_test = x_test.view([batch_size, -1, n_features])
                y_test = y_test
                self.model.eval()
                yhat = self.model(x_test)
                predictions.append(yhat.detach().numpy())
                values.append(y_test.detach().numpy())

        return predictions, values

        def plot_losses(self):
        plt.plot(self.train_losses, label="Training loss")
        plt.plot(self.val_losses, label="Validation loss")
        #wandb.log({"Plot": {"Losses": plt}})
        plt.legend()
        plt.title("Losses")
        plt.show()
        plt.close()



# -------------------------------------------------------------------------------------------------------
# Helper functions defined below
# -------------------------------------------------------------------------------------------------------


# functions (1 of 2) to get data stored in the drive into the dummy csv file
def new_csv(df, df_append_list=[]):
  new_df = df[['x_gt', 'y_gt', 'x_gps', 'y_gps', 'IMU_x', 'IMU_y', 'gyro']]
  new_df.to_csv('../datasets/lstm_dummy.csv', index=False)

  if len(df_append_list) != 0:
        for _df in df_append_list:
        append_df = _df[['x_gt', 'y_gt', 'x_gps', 'y_gps', 'IMU_x', 'IMU_y', 'gyro']]
        append_df.to_csv('../datasets/lstm_dummy.csv',
                mode='a', index=False, header=False)
```

```
# functions (2 of 2) to get data stored in the drive into the dummy csv file
def prepareDataset(N, training, files_to_append):
  if training:
        folder = '../datasets/Training_Data/Training_Set_'
        folderType = 'Training Data #'
  else:
        folder = '../datasets/Validation_Data/Validation_Set_'
        folderType = 'Validation Data #'

  path = folder + str(N) + '.csv'
  df = pd.read_csv(path)

  dfs_to_append = []
  if len(files_to_append) != 0:
        for file_path in files_to_append:
        dfs_to_append.append(pd.read_csv(file_path))

  new_csv(df, dfs_to_append)



# Gets the dataset in the current lstm dummy file, and returns a tuple
# containing the scaled dataframe, the original, and the scalars in the order
# x1, x2, x3, x4, x5, y1, y2
def readData():
        # read data from .csv to dataframe
        df_lstm = pd.read_csv('../datasets/lstm_dummy.csv')
        #df_lstm.drop(df_lstm.columns[[0]], axis=1, inplace=True)
        # Create a copy of the raw .csv file the RNN model will use (don't want to overwrite original
values)
        lstm_df_scaled = df_lstm.copy()
        # Scalers for all the features (independent of each other)
        x1_scaler = preprocessing.MinMaxScaler() # GPS X coordinate scalar
        x2_scaler = preprocessing.MinMaxScaler() # GPS Y coordinate scalar
        x3_scaler = preprocessing.MinMaxScaler() # Accelerometer X scalar
        x4_scaler = preprocessing.MinMaxScaler() # Accelerometer Y scalar
        x5_scaler = preprocessing.MinMaxScaler() # Gyro angular velocity reading scalar
        # Scalers for the outputs x,y (independent of each other)
        y1_scaler = preprocessing.MinMaxScaler() # x_gt scaler
        y2_scaler = preprocessing.MinMaxScaler() # y_gt scaler
        # Normalize X and Y of the RNN data to have values between 0 and 1
        lstm_df_scaled['x_gt'] = y1_scaler.fit_transform(df_lstm['x_gt'].to_numpy().reshape(-1, 1))
        lstm_df_scaled['y_gt'] = y2_scaler.fit_transform(df_lstm['y_gt'].to_numpy().reshape(-1, 1))
        # X (features)
```

```
        lstm_df_scaled['x_gps'] = x1_scaler.fit_transform(df_lstm['x_gps'].to_numpy().reshape(-1,
1))
        lstm_df_scaled['y_gps'] = x2_scaler.fit_transform(df_lstm['y_gps'].to_numpy().reshape(-1,
1))
        lstm_df_scaled['IMU_x'] = x3_scaler.fit_transform(df_lstm['IMU_x'].to_numpy().reshape(-1,
1))
        lstm_df_scaled['IMU_y'] = x4_scaler.fit_transform(df_lstm['IMU_y'].to_numpy().reshape(-1,
1))
        lstm_df_scaled['gyro'] = x5_scaler.fit_transform(df_lstm['gyro'].to_numpy().reshape(-1, 1))

        return (lstm_df_scaled, df_lstm, x1_scaler, x2_scaler, x3_scaler, x4_scaler, x5_scaler,
y1_scaler, y2_scaler)




# Takes in the scaled and unscaled dataset, getting the features and labels of
# each instance in a tuple, and returning a tuple of tuples
def splitData(lstm_df_scaled, df_lstm, train_size=1624, val_size=541, test_size=541):
        train_data_scaled = lstm_df_scaled[:train_size]
        val_data_scaled = lstm_df_scaled[train_size : train_size + val_size]
        test_data_scaled = lstm_df_scaled[train_size + val_size : ]

        X_train_scaled = train_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_train_scaled = train_data_scaled.filter(['x_gt','y_gt'], axis=1)
        train_tuple_scaled = (X_train_scaled, Y_train_scaled)
        X_test_scaled = test_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_test_scaled = test_data_scaled.filter(['x_gt','y_gt'], axis=1)
        test_tuple_scaled = (X_test_scaled, Y_test_scaled)
        X_val_scaled = val_data_scaled.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_val_scaled = val_data_scaled.filter(['x_gt','y_gt'], axis=1)
        val_tuple_scaled = (X_val_scaled, Y_val_scaled)

        train_data = df_lstm[:train_size]
        val_data = df_lstm[train_size : train_size + val_size]
        test_data = df_lstm[train_size + val_size : ]

        X_train_raw = train_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_train_raw = train_data.filter(['x_gt','y_gt'], axis=1)
        train_tuple = (X_train_raw, Y_train_raw)
        X_test_raw = test_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_test_raw = test_data.filter(['x_gt','y_gt'], axis=1)
        test_tuple = (X_test_raw, Y_test_raw)
```

```
        X_val_raw = val_data.filter(['x_gps','y_gps','IMU_x','IMU_y','gyro'], axis=1)
        Y_val_raw = val_data.filter(['x_gt','y_gt'], axis=1)
        val_tuple = (X_val_raw, Y_val_raw)

        return (train_tuple_scaled, test_tuple_scaled, val_tuple_scaled, train_tuple, test_tuple,
val_tuple)
```

```
# Takes in a set of 3 tuples containing the scaled inputs and splits the data
# into randomly shuffled batches, returned  in a tuple of train, validation,
# test, and unshuffled test
def dataLoading(training_data, validation_data, testing_data, batch_size=24):
        train_features = torch.Tensor(training_data[0].to_numpy())
        train_targets = torch.Tensor(training_data[1].to_numpy())

        test_features = torch.Tensor(testing_data[0].to_numpy())
        test_targets = torch.Tensor(testing_data[1].to_numpy())

        val_features = torch.Tensor(validation_data[0].to_numpy())
        val_targets = torch.Tensor(validation_data[1].to_numpy())

        train = TensorDataset(train_features, train_targets)
        val = TensorDataset(val_features, val_targets)
        test = TensorDataset(test_features, test_targets)

        train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, drop_last=True)
        val_loader = DataLoader(val, batch_size=batch_size, shuffle=True, drop_last=True)
        test_loader = DataLoader(test, batch_size=batch_size, shuffle=True, drop_last=True)
        test_loader_one = DataLoader(test, batch_size=1, shuffle=False, drop_last=True)
        return (train_loader, val_loader, test_loader, test_loader_one)
```

```
# Functions to transform the predictions back to original scale
def inverse_transform(df, y1_scaler, y2_scaler):
        df_gt = df.filter(['x_gt','y_gt'], axis=1)
        df_pred = df.filter(['x_pred','y_pred'], axis=1)

        arr_x_gt = y1_scaler.inverse_transform(df_gt['x_gt'].to_numpy().reshape(-1 ,1))
```

```
        arr_y_gt = y2_scaler.inverse_transform(df_gt['y_gt'].to_numpy().reshape(-1 ,1))

        arr_x_pred = y1_scaler.inverse_transform(df_pred['x_pred'].to_numpy().reshape(-1 ,1))
        arr_y_pred = y2_scaler.inverse_transform(df_pred['y_pred'].to_numpy().reshape(-1 ,1))

        df_gt['x_gt'] = arr_x_gt
        df_gt['y_gt'] = arr_y_gt

        df_pred['x_pred'] = arr_x_pred
        df_pred['y_pred'] = arr_y_pred

        return pd.concat([df_gt, df_pred], axis=1)


# Formats the prediction results
def format_predictions(predictions, values, df_test, y1_scaler, y2_scaler):
        vals = np.concatenate(values, axis=0)
        preds = np.concatenate(predictions, axis=0)
        df_result = pd.DataFrame(data={'x_gt': vals[:,0], 'y_gt': vals[:,1], 'x_pred': preds[:,0],
'y_pred': preds[:,1]}, index=df_test.head(len(vals)).index)
        df_result = df_result.sort_index()
        df_result = inverse_transform(df_result, y1_scaler, y2_scaler)
        return df_result


# Function that calculates metrics
def calculate_metrics(df):
        return {
        'mae_x' : mean_absolute_error(df.x_gt, df.x_pred),
        'rmse_x' : mean_squared_error(df.x_gt, df.x_pred) ** 0.5,
        'r2_x' : r2_score(df.x_gt, df.x_pred),
        'mae_y' : mean_absolute_error(df.y_gt, df.y_pred),
        'rmse_y' : mean_squared_error(df.y_gt, df.y_pred) ** 0.5,
        'r2_y' : r2_score(df.y_gt, df.y_pred)
        }
def gps_metrics(df):
        return {
        'mae_x' : mean_absolute_error(df.x_gt, df.x_gps),
        'rmse_x' : mean_squared_error(df.x_gt, df.x_gps) ** 0.5,
        'r2_x' : r2_score(df.x_gt, df.x_gps),
        'mae_y' : mean_absolute_error(df.y_gt, df.y_gps),
        'rmse_y' : mean_squared_error(df.y_gt, df.y_gps) ** 0.5,
        'r2_y' : r2_score(df.y_gt, df.y_gps)
        }
```

```
def plotResults(df_lstm, df_result):
	plt.figure(figsize=(12, 8))

	#gt_path = pd.DataFrame(df_lstm, columns = ['x_gt','y_gt']).drop(list(range(drop_range[0],
drop_range[1])), axis=0)
	#gps_path = pd.DataFrame(df_lstm, columns =
['x_gps','y_gps']).drop(list(range(drop_range[0], drop_range[1])), axis=0)

	#plt.scatter(gt_path['x_gt'], gt_path['y_gt'], color = 'g')

	for row in df_result.iterrows():
	plt.scatter(row[1][0], row[1][1], color = 'r', s = 12)
	plt.scatter(row[1][2], row[1][3], color = 'b', s = 12)
	#wandb.log({"Plot": {"Model Output": plt}})

	plt.show()



def trainModelStartToFinish(trial, lr, epochs, num_layers, hidden_size, append_dataset_list=[],
N=0, display=True):
	#batch_size = 1024
	batch_size = 24
	n_epochs = epochs
	learning_rate = lr
	n_features=5

	prepareDataset(N, True, append_dataset_list)
	processedData = readData()

	lstm_df_scaled = processedData[0]
	df_lstm = processedData[1]
	y1_scaler = processedData[7]
	y2_scaler = processedData[8]

	#manually change the sizes
	#big_tuple = splitData(lstm_df_scaled, df_lstm, train_size=13530, val_size=5412,
test_size=5412)
	big_tuple = splitData(lstm_df_scaled, df_lstm, train_size=1624, val_size=541,
test_size=541)
	training = big_tuple[0]
```

```
        testing = big_tuple[1]
        validation = big_tuple[2]
        X_test_scaled = testing[0]

        train_loader, val_loader, test_loader, test_loader_one = dataLoading(training, validation,
testing)

        #model = LSTM_Model(input_size=n_features, output_size=2, hidden_dim=36,
n_layers=1)
        model = LSTM_Model(input_size=n_features, output_size=2, hidden_dim=hidden_size,
n_layers=num_layers)
        loss_fn = nn.MSELoss(reduction="mean")
        optimizer = optim.Adam(model.parameters(), lr=learning_rate)
        opt = Trainer(model=model, loss_fn=loss_fn, optimizer=optimizer)
        opt.train(trial, train_loader, val_loader, batch_size=batch_size, n_epochs=n_epochs,
n_features=n_features)

        predictions, values = opt.evaluate(test_loader_one, batch_size=1, n_features=n_features)
        df_result = format_predictions(predictions, values, X_test_scaled,  y1_scaler, y2_scaler)
        result_metrics = calculate_metrics(df_result)
        result_metrics_gps = gps_metrics(df_lstm)
        #opt.plot_losses()
        #plotResults(df_lstm, df_result)

        mae_x = result_metrics['mae_x']
        mae_y = result_metrics['mae_y']
        rmse_x = result_metrics['rmse_x']
        rmse_y = result_metrics['rmse_y']
        r2_x = result_metrics['r2_x']
        r2_y = result_metrics['r2_y']

        mae_x_gps = result_metrics_gps['mae_x']
        mae_y_gps = result_metrics_gps['mae_y']
        rmse_x_gps = result_metrics_gps['rmse_x']
        rmse_y_gps = result_metrics_gps['rmse_y']
        r2_x_gps = result_metrics_gps['r2_x']
        r2_y_gps = result_metrics_gps['r2_y']

        wandb.log({"result table": df_result})
        wandb.log({"Result Metrics": result_metrics, "GPS Metrics": result_metrics_gps})

        return (model, result_metrics)
```

```
# ------------------------------------------------------------------------------------------
# Optuna functions
# ------------------------------------------------------------------------------------------



def objective(trial):
	lr = trial.suggest_float("lr", 0.0001, 0.001, log=True)
	n_epochs = trial.suggest_int("n_epochs", 50, 250, step=50)
	hidden_size = trial.suggest_int("hidden_size", 16, 512, step=8)
	num_layers = 1

	config = dict(trial.params)
	config["trial.number"] = trial.number
	wandb.init(
	project="MSF Optuna",
	#entity="nzw0301",
	config=config,
	#group=STUDY_NAME,
	reinit=True,
	)

	model, result_metrics = trainModelStartToFinish(trial, lr=lr, epochs=n_epochs,
num_layers=num_layers,
				hidden_size=hidden_size, append_dataset_list=[], N=0,
display=False)

	wandb.finish(quiet=True)

	#return (1 - result_metrics['r2_x']) + (1 - result_metrics['r2_y'])
	return result_metrics['mae_x'] + result_metrics['mae_y']

# value: 8.822477340698242 and parameters: {'lr': 0.00018868435800919247, 'n_epochs': 250,
'hidden_size': 256, 'weight_decay': 2.0676999665529862e-07}

def run_optuna(num_trials):
	study = optuna.create_study(pruner=optuna.pruners.SuccessiveHalvingPruner())
	study.optimize(objective, n_trials=num_trials)
	return study

def run_wandb(study):
	wandb.init(
```

```
        project="MSF",
        #entity="nzw0301",
        #config=config,
        #group=STUDY_NAME,
        reinit=True,
        )

        wandb.log({"Optimization History Plot": plot_optimization_history(study)})
        wandb.log({"Parallel Coordinate Plot": plot_parallel_coordinate(study)})
        wandb.log({"Contour Plot": plot_contour(study)})
        wandb.log({"Slice Plot": plot_slice(study)})
        wandb.log({"Hyperparameter Importance Plot": plot_param_importances(study)})
        wandb.log({"Emperical Distribution Function Plot": plot_edf(study)})

        wandb.finish(quiet=True)

def run_model(trial=0,lr=0.00008, epochs=10, num_layers=1, hidden_size=512,
append_dataset_list=[], N=0, display=True):
        return trainModelStartToFinish(trial=trial, lr=lr, epochs=epochs, num_layers=num_layers,
        hidden_size=hidden_size, append_dataset_list=append_dataset_list, N=N,
display=display)

def main():
        args = sys.argv[0:]
        argc = 1

        while argc < len(args):
        if args[argc] == '-n':
        assert(argc + 1 <= len(args) - 1)
        print(int(args[argc + 1]))
        run_wandb(run_optuna(int(args[argc + 1])))
        return

if __name__ == '__main__':
        main()
```

KNN:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import pytorch_lightning as pl
import matplotlib.pyplot as plt
```

```
import sys
import os

'''
  The first argument should be the path to the file. By default this looks at the GPS
  data from the AGL file as the prediction, if the second argument is -m, then
  it compares a model's predictions instead.
'''

def outlierDetection(df_result, model):

  if (model):
        x_gt = torch.tensor(data = df_result['x_gt'], dtype = torch.float)
        y_gt = torch.tensor(data = df_result['y_gt'], dtype = torch.float)
        x_pred = torch.tensor(data = df_result['x_pred'], dtype = torch.float)
        y_pred = torch.tensor(data = df_result['y_pred'], dtype = torch.float)
  else:
        x_gt = torch.tensor(data = df_result[' x_gt'], dtype = torch.float)
        y_gt = torch.tensor(data = df_result[' y_gt'], dtype = torch.float)
        x_pred = torch.tensor(data = df_result[' x_gps'], dtype = torch.float)
        y_pred = torch.tensor(data = df_result[' y_gps'], dtype = torch.float)

  data_gt = torch.stack((x_gt, y_gt), axis = 1)
  data_pred = torch.stack((x_pred, y_pred), axis = 1)

  distance = torch.norm(data_gt - data_pred, dim = 1, p = None) #get distances between
corrosponding coordinates
  min_d = distance.min()
  max_d = distance.max()

  hist = torch.histc(distance, bins = 100, min = min_d, max = max_d) #make pytorch histogram
  bins = 100   #adjustable, must change ^ if changed
  r = range(bins)

  quantiles = torch.tensor([0.75]) #get upper 90% of distance values
  threshold = torch.quantile(distance, quantiles, dim=0, keepdim=True) #set threshold for
determining outliers

  #quartile threshold for model comparison
  #Distance = 13 for overall accuracy has proven most consistent

  plt.figure(figsize=(12, 8))
  plt.title("Detecting Outliers")
```

```
  plt.scatter(data_gt[:, 0], data_gt[:, 1], color = 'b') #ground truth points
  for i in range(len(distance)):
        if distance[i] > threshold and distance[i] > 15:
        plt.scatter(data_pred[i][0], data_pred[i][1], color = 'r', s = 20) #values that are outliers
        else:
        plt.scatter(data_pred[i][0], data_pred[i][1], color = 'g', s = 8) #values within max distance


   #if you just want to see the data point distances, uncomment the code below and comment the
code below
  '''
  plt.figure(figsize=(12, 8))
     distances = []
     if len(df_result['x_gt']) == len(df_result['x_pred']):
          for row in df_result.iterrows():
            dist = np.sqrt((row[1][0] - row[1][2])**2 + (row[1][1] - row[1][3])**2)
            distances.append(dist)
          df = pd.DataFrame(distances, columns = ['distance'])
          new_df = df.sort_values(by = 'distance')
          new_df.hist(column='distance', bins=50, grid=False, figsize=(12,8))
  '''

  plt.show()


if len(sys.argv) < 2:
        print("Please provide a file path to GroundTruthAGL.csv or a model's predictions")
else:
        df = pd.read_csv(sys.argv[1])
        modelPredict = False
        if len(sys.argv) > 2:
        modelPredict = (sys.argv[2] == "-m")
        outlierDetection(df, modelPredict)
```