

İTÜ



UCK 358E: Term Project
22828 - Thursday

Due on Thursday, May 23, 2024

Assoc. Dr. Barış Başpınar

Ecem Sengel* - 110200149

*sengel20@itu.edu.tr

Contents

| | | |
|-----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Brief Information About The Data | 1 |
| 3 | Data Preprocessing | 3 |
| 3.1 | X-train Part | 3 |
| 3.2 | y-train Part | 3 |
| 4 | Models | 4 |
| 4.1 | Linear Regression Model | 4 |
| 4.2 | RNN Model | 4 |
| 4.3 | CNN-GRU Model | 5 |
| 5 | Random Forest (RF) Model | 6 |
| 6 | Support Vector Machine (SVM) Model | 6 |
| 7 | Gradient Boosting Machines (GBM) Model | 7 |
| 8 | K-Nearest Neighbourhood (KNN) Model | 7 |
| 9 | Confronted Challenges | 7 |
| 9.1 | Feature Selection Confusion | 7 |
| 9.2 | Creating a Suitable Model | 8 |
| 9.3 | Ensuring Time Synchronization | 8 |
| 9.4 | Training The Models | 8 |
| 9.5 | Creating Submission Format | 9 |
| 9.6 | Prediction Problem | 9 |
| 10 | Conclusion and Interpretations | 9 |
| 11 | References | 12 |

1 Introduction

The precision of positioning systems is critical in an age where location-based services are ubiquitous. The Kaggle-hosted "Google Smartphone Decimeter Challenge" is a key venue for pushing the boundaries of machine learning-based positioning technology. The main goal here is improving positioning accuracy by skillfully utilizing sensor data and raw GNSS measurements.

Kalman filtering process usually appears in such problems, in this study, the problem was tried to be solved by using artificial intelligence models without applying any Kalman filtering process. While trying to create reliable machine learning models that can greatly increase positioning accuracy over traditional approaches, other objectives of the study can be named as careful data preprocessing, feature engineering, and model selection, but also to obtain insights into the complex interactions between different features and model architectures.

The work developed for the stated purpose is explained in detail, step by step, together with the difficulties experienced, in the relevant sections that follow in the report. The code used in addition to the explanations is not shared in this report, but is shared as a separate file in '.ipynb' format.

2 Brief Information About The Data

The data required to be used are presented separately as train and test parts as csv files. Here, data for each device is filed, along with the date and location of the measurement and the device models where the measurement was made. When these csv files are opened, measurements including gnss and imu data are also seen. There are approximately 45000 lines in each gnss and imu file, 58 features in each gnss file and 9 features in each imu file. The first thing to do here was to decide which feature model would be used to train. First, I eliminated the columns that were the same in all rows or had NAN values in all rows. After this elimination process, I had 40 features left. After that, I examined the previous studies and selected the features that I thought would be most directly effective on location data according to reference [2]. The features selected in gnss and imu data are shared separately in the list below.

For gnss data:

- utcTimeMillis
- SvVelocityYEcefMetersPerSecond
- SvVelocityZEcefMetersPerSecond

- SvVelocityXEcefMetersPerSecond
- RawPseudorangeMeters
- SvClockBiasMeters
- IsrbMeters
- TroposphericDelayMeters
- IonosphericDelayMeters
- WlsPositionXEcefMeters
- WlsPositionYEcefMeters
- WlsPositionZEcefMeters

For imu data:

- utcTimeMillis
- MessageType
- MeasurementX
- MeasurementZ
- MeasurementY

During the research carried out to recognize the data, it was understood that many features presented in the GNSS data actually affected another feature and were used to calculate it, so specific column selection was seemed necessary.

3 Data Preprocessing

3.1 X-train Part

In both data, multiple measurements were taken for the same sensor at times with milliseconds of difference and sometimes even the same time. As seen in sample-submission, I reduced this measurement interval to a single line by taking the average of the data taken in the same second to separate it as if it were taken at one-second intervals. Since I did not need to do any further processing for the GNSS data, I set the time intervals and assigned them to a variable to combine with other sensor data to train the model.

Since things were a little more complicated with the imu data, I had to separate the three types of sensor data (gyroscope, magnetometer, accelerometer) before making the time adjustment. Since the data received from the gyroscope, magnetometer, and accelerometer sensors were mixed, I first separated them all and recorded the measurements of all of them on the x, y, and z axis in the dataframes I created by specifying the sensor names. Then, I set the times to progress in seconds, as I set in the gnss data, and combined all the sensor data in the variable named total-imu, based on the times, as I set the gnss data.

Finally, I combined the gnss data and total-imu data, which will form the X part of my train set, to match the times and created my x-train data. Lines at intersecting times were matched with the help of the `.merge()` function.

The last thing I paid attention to before starting to train the models was to remove the time column and perform normalization.

Then I trained my alternative models respectively. I repeated all the specified processes for each .csv file given in the specified order and I completed x part of my model training process.

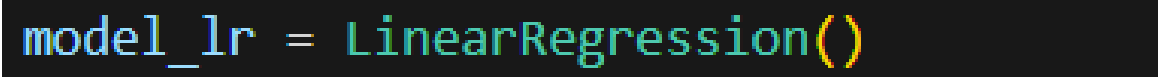
3.2 y-train Part

Here again I had to take into account the ground-truth files provided by the same filling process. Time adjustment was made as mentioned before, and this data used as the y-train data to train the models.

4 Models

4.1 Linear Regression Model

To create a reference I applied the linear regression model directly because I had a regression problem at hand. I used the ready-made Linear Regression function in the Sklearn library with default values. Simply the code is given below as:



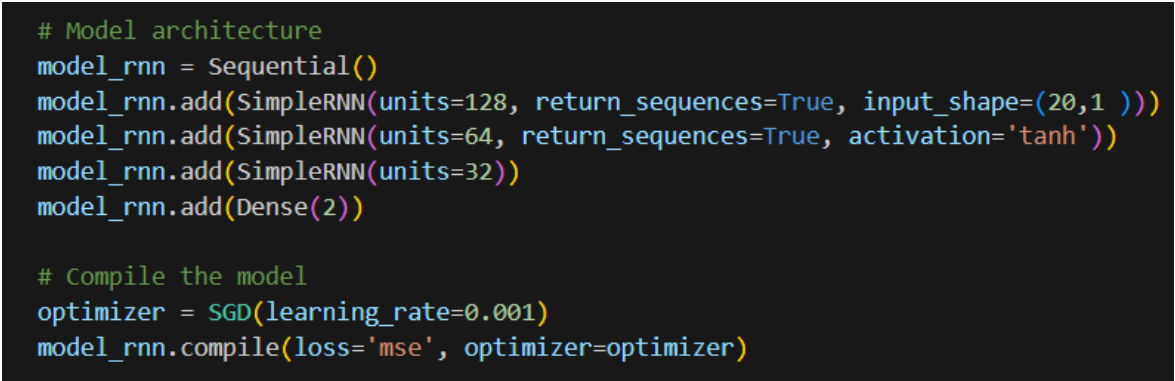
```
model_lr = LinearRegression()
```

Figure 1: Linear Regression Default Model

I predict that the values will most likely not change with a linear connection, but my main purpose here was to create a classical and complex model and then have a comparison value since it is a regression problem.

4.2 RNN Model

I know RNNs are useful for processing time data, and I also see that it is one of the most preferred methods in the ML part of the previous works. Based on those, an RNN model was created. The basic structure of the model, such as the number of layers and density values, can be easily seen in the figure shared below:



```
# Model architecture
model_rnn = Sequential()
model_rnn.add(SimpleRNN(units=128, return_sequences=True, input_shape=(20,1 )))
model_rnn.add(SimpleRNN(units=64, return_sequences=True, activation='tanh'))
model_rnn.add(SimpleRNN(units=32))
model_rnn.add(Dense(2))

# Compile the model
optimizer = SGD(learning_rate=0.001)
model_rnn.compile(loss='mse', optimizer=optimizer)
```

Figure 2: RNN Model

The loss function values were decreased up to 0.0008061218541115522 for the last version of the code, which was actually the given version above, but after I retried to run the whole code I got high values of losses and nan values after some point, I could not find the reason. Since

loss functions are just a method chosen by me for model training behavior representation, I had to continue ignoring this part after making sure that the model was trained.

4.3 CNN-GRU Model

Models such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Units), which are a special type of RNNs, can effectively learn long-term dependencies, which I know will increase location prediction accuracy. I decided to use this model after reviewing the article mentioned in ref[1]. I saw that the CNN (Convolutional Neural Network) model was used in a few studies I reviewed, and in this study [1] an improvement was added to the CNN model with GRU (gated recurrent unit). I tried to develop a model similar to the model described in the article. I tried different combinations for activation functions and intermediate layer numbers. Based on the loss values, the one with the lowest loss value can be seen in the figure below, and can also be examined in the code in the .ipynb file.

According to the article, this method utilizes a CKF (Central Kalman Filter) module for precise navigation information and integrates it with CNN-GRU for enhanced navigation during GNSS signal outages. The CNN-GRU network effectively processes multi-sensor inputs and time-dependent motion data. CNN aids in feature extraction, while GRU handles sequential analysis efficiently. For my own work, I tried to create an application without the Kalman filtering part.

```
# Define the combined CNN-GRU model
def create_cnn_gru_model(input_shape, l2_reg=0.01):
    model = tf.keras.Sequential([
        # CNN layers
        tf.keras.layers.Reshape((-1, 1), input_shape=input_shape), # Reshape for Conv1D input
        tf.keras.layers.Conv1D(filters=128, kernel_size=3, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg)),
        tf.keras.layers.MaxPooling1D(pool_size=2),
        tf.keras.layers.Conv1D(filters=64, kernel_size=3, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg)),
        tf.keras.layers.MaxPooling1D(pool_size=2),
        tf.keras.layers.Flatten(),

        # GRU layer
        tf.keras.layers.Reshape((-1, 64)), # Reshape for GRU input
        tf.keras.layers.GRU(48),

        # Dense output layer
        tf.keras.layers.Dense(2) # Output layer for predicting 2 new features
    ])
    return model

# Define the model
input_shape = (None, 21) # Adjust according to your data dimensions (21 features)
model_cnn_gru = create_cnn_gru_model(input_shape)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001) # Set the learning rate to 0.001

# Compile the model with the customized optimizer
model_cnn_gru.compile(optimizer=optimizer, loss='mae')

# Function to train the model on a single file with gradient clipping
def train_model_on_file(model, x_train, y_train, clip_norm=1.0):
    model.fit(x_train, y_train, epochs=10, batch_size=16) # Train the model
```

Figure 3: CNN-GRU Model

When I tried to train my model created as shared above, I noticed that there was a fluctuation in the loss values. In general, there seems to be a decrease in the loss values, but I saw that the values continued to decrease, peaking in between and falling back to the remaining values. Normally, I would expect such behavior not to occur, I think it may be a problem caused by poor data processing or the data provided is unstable, but I still could not solve this problem. In general, I observed that the average loss values decreased to around 0.09 at final iterations.

5 Random Forest (RF) Model

With the help of the Sklearn library, algorithms such as random forest and decision tree are expected to give relatively good results without much effort. Since the Random forest model is actually a model that includes the decision tree model, I chose to use it for the study. Since I had a lot of data and data close to each other, I thought a slightly more complicated model might be more useful. The model is shaped as seen in the figure below. In the model, I only increased the n-estimators value without playing much with the default values.

```
model_rf = MultiOutputRegressor(RandomForestRegressor(n_estimators=150, random_state=42))
```

Figure 4: Random Forest Model

6 Support Vector Machine (SVM) Model

SVMs can be effective on high-dimensional datasets where the number of features exceeds the number of data samples. Since the data I need to process is high-dimensional, I expect the svm model to perform well, at least close to the Random Forest model. Here again, I made sure to change the kernel variable to 'poly' in the model I called from the Sklearn library. Since it would normally do a single variable regression, it became a multi-regressor to make predictions in latitude and longitude. To create a more complex model, I slightly increased the degree variable. The final version of the model is as follows:

```
svr = SVR(kernel='poly', degree=5, C=1.0, epsilon=0.1)
model_svm = MultiOutputRegressor(svr)
```

Figure 5: Support Vector Machine Model

7 Gradient Boosting Machines (GBM) Model

Gradient Boosting Machines (GBM) algorithm is a powerful and flexible method for regression and classification problems. This model generally provides high performance by combining many weak learners (typically decision trees) into a strong predictor. Since it can adapt to different datasets and problems due to its many hyperparameters (learning rate, number of weak learners, depth of weak learners, etc.), I could easily modify it to use for my work. The structure is shared below at the figure:

```
gbr = GradientBoostingRegressor(n_estimators=80, learning_rate=0.01, max_depth=5, random_state=42)
model_gbm = MultiOutputRegressor(gbr)
```

Figure 6: Gradient Boosting Machines Model

8 K-Nearest Neighbourhood (KNN) Model

K-Nearest Neighbors (KNN) is a simple yet effective algorithm for regression and classification problems. In fact, this model can be considered as the advanced model after linear regression among the models used. Even though it doesn't give as good a result as the others, I expect it to give a good result from linear regression. One advantage can be told as it is straightforward to understand and implement. It makes predictions based on the closest training samples in the feature space. Model parameters were set and used as shown below.

```
knn = KNeighborsRegressor(n_neighbors=6, weights = 'distance')
model_knn = MultiOutputRegressor(knn)
```

Figure 7: K-Nearest Neighbourhood Model

9 Confronted Challenges

9.1 Feature Selection Confusion

First of all, it was necessary to understand the given data and examine the feature properties and decide whether all or some of it should be used. After doing some research on this subject, I thought that instead of using all the features, it would be enough to select a few of them. When I did a little research on how GPS data and GNSS data are received and processed, I saw that they used some of the parameters I saw in the column names to find the parameter

I saw in another column. Instead of using all of them, I selected the latest features that I thought would be most effective in finding the location data and continued in this way.

9.2 Creating a Suitable Model

In such problems, it is observed that Kalman filtering is generally used to find location (latitude and longitude data) data by processing imu sensor fusion or gnss data. I have seen ML algorithms encountered in some hybrid Kalman filtering applications that have been created. This is how I decided which models would make more sense for the models I needed to create.

9.3 Ensuring Time Synchronization

One of the problems that I had the most difficulty understanding and solving was ensuring time synchronization and adjusting the data accordingly. First of all, I examined the ground-truth data to understand in which intervals I should sort the data and make predictions. I saw that each data was given one second apart. In order to be able to process only seconds in time columns in both GNSS and IMU data, I calculated the remaining integers from 10^{10} divisions and assigned them as time values. Thus, I took the first ten characters and could reference the last digits as seconds. Then, I averaged the data at the same second for both datasets and reduced them to a single line. Thus, the data was adjusted to advance by one second. Then, I made a match based on the 'utcTimeMillis' column with the `pd.merge()` function.

9.4 Training The Models

First, while training my models, I opened the files under training one by one and then tried to process the data in the current file opened to train my models and put it into the model. During these processes, I realized that it took me a lot of time to train each model, and my models were not trained properly, probably because there was some confusion, and I was encountering false values in the data I prepared as x-train. To resolve this, after opening the files and doing the preprocessing phase, I named them as I opened them, made them ready to train the model, and stored them as .csv files again. I created a more organized system by checking the nan values at each stage and filling them in. When it came to model training, I opened these files again and did the training processes separately for each model. Thus, the problems I encountered before were eliminated and I was saved from having to

read all the train files each time and repeat the preprocessing phase. Since I trained the models separately, I was able to detect and solve errors more easily. happened.

9.5 Creating Submission Format

Since the required format for submitting here is very strict, even though I did the guesswork, I realized that I had serious row losses due to opening my test data and applying the `.merge()` operation and adding time-dependent intersection sets, and therefore I could not reach the sufficient number of rows that I needed to submit. To solve this problem, I realized that I needed to change the way I processed test data to open and make predictions. I also used this method for training, but since I did not have enough time to revise either of them, I realized that the inadequacy of my training process would be insufficient due to the preprocessing process. To avoid losing rows, I used the file given as submission-data to reference the time values. In this way, I created a dataframe filled with the time values in the submission file and containing the columns required for the x part of the test data. I tried to fill in the values that needed to be there as much as I could by opening the test files, and fill the remaining empty values with the values above and below. Since I thought that the number of Nan values would not be very high, I thought that it would not affect the accuracy rates of the models much. The code for this section is shown under the Test heading in the .ipynb file.

9.6 Prediction Problem

I did not encounter any problems while training and obtaining loss values for the models I created, but when I reached the final stage and started making predictions, I got some values, although their accuracy was low at first. Later, after a few tries, I realized that I was getting nan values when I tried to make predictions and I could not solve this problem. To fix this, I tried changing my model and revisiting the preprocessing part of the test data. Even though I applied the same preprocessing process as I do for training data, I could not make any predictions and I saw that I saved files full of nan. Later, I did not have enough time to fix my corrupted system, and I had to leave it that way.

10 Conclusion and Interpretations

The first result I got actually includes the predictions I made with the model I created for trial purposes. In order to prepare a data set suitable for the submission format, I made a

prediction and submission with the undeveloped versions of the models and got the result as shared below:



Figure 8: First Submission Result

The scores obtained as a result of the predictions I made and the submissions I made after making the corrections and the final versions of my models mentioned above are shared respectively in the figures below.



Figure 9: RNN Submission Result



Figure 10: CNN-GRU Submission Result



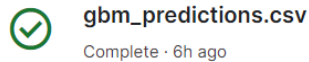
Figure 11: Linear Regression Submission Result



Figure 12: Random Forest Submission Result



Figure 13: SVM Submission Result



97298.301

Figure 14: GBM Submission Result

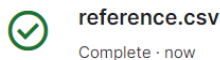


98090.676

Figure 15: KNN Submission Result

Linear regression got a worse score than other advanced models, as I expected. Among the results, what I did not expect was the score of the RNN model. I noticed that there were some problems while training the model, but I got this result because I could not find the source of the error to fix it. While other models generally score close to each other, I see that the most successful model is CNN-GRU. As the most advanced model among them, I actually got the result I expected. Although the differences are small, we can still say that the model with the highest accuracy is CNN-GRU.

In fact, the most logical way to compare how successful you are is to provide a reference value by directly converting the values given in x, y, and z coordinates in the test data into longitude and latitude values, without making any position improvements. The score obtained as a result of the submission of the data created by converting latitude and longitude from ECEF coordinates can be seen in the figure below.



198413.213

Figure 16: Given GNSS Values Submission Result Before Improvement

In general, it was observed that most of the models had some position improvement, albeit slightly. It seems that the predictions made with linear regression are worse than the predictions without any improvement. I expected that linear regression would not be useful in such a complex data set, and it turned out this way.

An unexpected result was that my RNN model had a much higher error rate than expected, I could not solve the fundamental problem with this model, it is most likely an implementation problem rather than a failure of the model.

11 References

- [1] Akyazı, Ö. (2019). IMU Sensor Fusion With Machine Learning [Application project report, Department of Informatics, Technische Universität München].
- [2] UCF IAHR. (2015, February 27). GPS RTK Surveying Workshop [Video]. YouTube. https://www.youtube.com/watch?v=v0J3u7Zd_i0
- [3] taroz1461. (2022, October 7). 1st Place Winner of the Smartphone Decimeter Challenge: Two-Step Optimization of Velocity and Position Using Smartphone's Carrier Phase Observations [Video]. YouTube. <https://www.youtube.com/watch?v=mxan4k9xY4U&t=127s>
- [4] Brena, R. F., Aguilera, A. A., Trejo, L. A., Molino-Minero-Re, E., and Mayora, O. (2020). Choosing the Best Sensor Fusion Method: A Machine-Learning Approach. *Sensors*, 20(8), 2350. <https://doi.org/10.3390/s20082350>