MC520 Machine Learning – WS 2018/19
Project Report (Airbender)

Oliver Adam, Robert Gstöttner, Anastasiia Mishchenko, Roland Spindelbalker

15th February, 2019

Introduction

Airbender is a gesture control application for the apple watch. The main focus of this application is to assist people in their life, by controlling different devices only with simple gestures. Potentially it could be applied in different areas of medicine where hands free actions are necessary, or in areas of healthcare where people are limited to moving their hands. The goal is to have a general user interface which requires a very low learning curve and is very simple to use.

The idea is to create a smart watch application which can be used to control different devices via air–gestures. The Gestures can be recorded via the accelerometer and gyroscope sensors. The problem with this sensor bases approach is that the signals vary in length and in magnitude between different users. Therefore data–preprocessing is very challenging in this case. The gestures should also be orientation independent, which basically means that the users position should be independent of the derived gestures. Another problem is the gesture segmentation. It is necessary to detect the start point of the gesture and the end point int time. In order to do that it would be possible to use a binary classifier. Figure 1 shows the basic toolchain required to solve the problem.

The goal is to create a system which is able to distinguish between several predefined gestures. These gestures can then be used to perform different actions onto various devices. Basically, a new type of user interface for computers should be created which is highly robust against gesture variations.

Toolchain

As mentioned above, before evaluating which gesture was done, a binary classifier is needed to distinguish between random actions and wanted gestures (can be seen in figure 1).

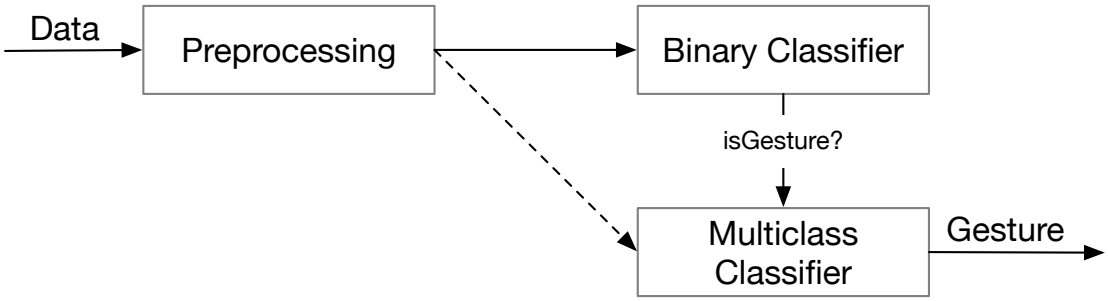

Figure 1: Basic toolchain

Figure 2 shows the toolchain in more detail. First we get the data of the accelerometer and gyroscope. Then we resample the data to 50 samples so we can perfectly derive features. Then we take the data and calculate the mean and median in time for these samples and then the first 20hz of the frequency band. These are our features. We reduce the features further to eliminate correlations. The next step is to train a RF model. For evaluating if a valid gesture has occured, we generate random movements (invalid gestures) out of the data, so we can train and test a OC-SVM binary classifier. The last step is to export the models into our mobile application.

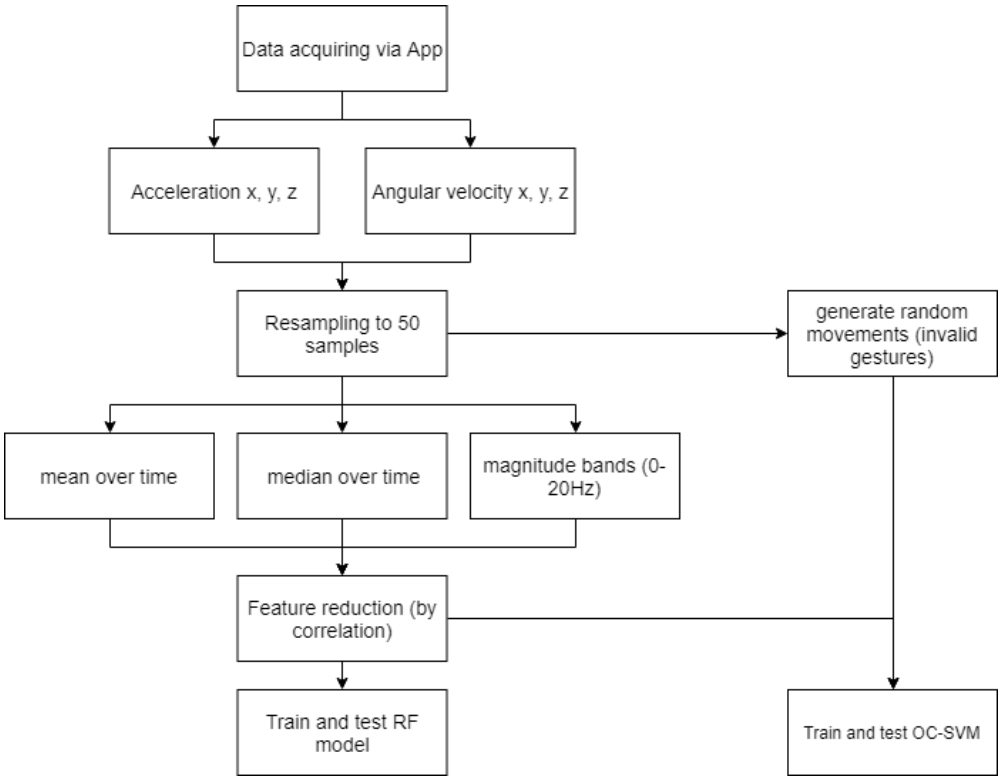

Figure 2: Toolchain in detail

Data analysis

In total the data has 600 gestures which consist of 12 different gestures, five participants and ten repeats. For every gesture the acceleration in the x, y and z axis and the angular velocity in the x, y and z axis was measured with 100 Hz.

The various gestures are the following:

- rotate
- double rotate
- swipe left
- swipe right
- swipe up
- swipe down
- circle clockwise
- circle counterclockwise
- clap
- double clap
- imaginary button
- check mark

The following figures 3 and 4 show the acceleration and the angular velocity of such a gesture, in this example of a double rotation.

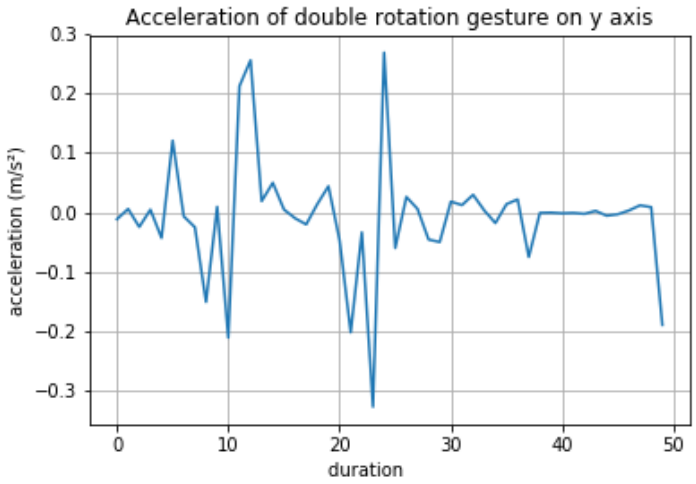

Figure 3: Acceleration of double rotation gesture on y axis

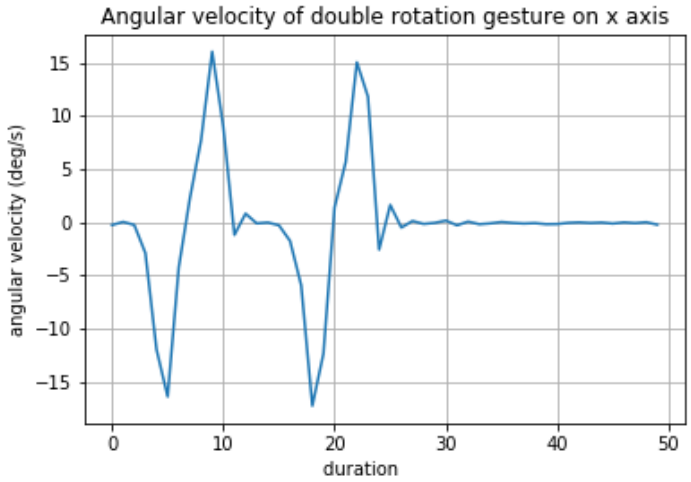

Figure 4: Angular velocity of double rotation gesture on x axis

Data preprocessing

Before deriving the features the noise has to be filtered. Because the iPhone doesn't support advanced filtering techniques we used resampling to reduce noise and get better samples. So we down-sampled the recordings to 50 samples.

Feature derivation

Because the iPhone has a lack of libraries for machine learning we decided to use basic features like:

- mean in time
- median in time
- band frequencies

Time domain

In the time domain we basically calculated the mean and median of the samples.

Frequency domain

First we applied the fft on every recording and then calculated the magnitude by applying the modulus. This can be seen in the code snippet 5.

```python
def get_magnitude_band(data, band_width):
    temp_feature = pd.DataFrame()
    def get_band(row, band_width, label):
        fft = np.fft.fft(row)
        magnitude = np.abs(fft)
        return pd.Series(magnitude[1:band_width+1],index=[label + "_" + str(x) for x in range(1,band_width+1)])

    count=0
    for frame in data:
        bands = frame.iloc[:,2:].apply(lambda x:get_band(x, band_width, data_names[count]), axis=1)
        temp_feature = pd.concat([temp_feature, bands], axis=1, sort=False)
        count+=1

    return temp_feature
```

Figure 5: FFT code

After that we got a beautiful magnitude of the first 20 bands which are then used for identifying gestures (can be seen in figure 6).

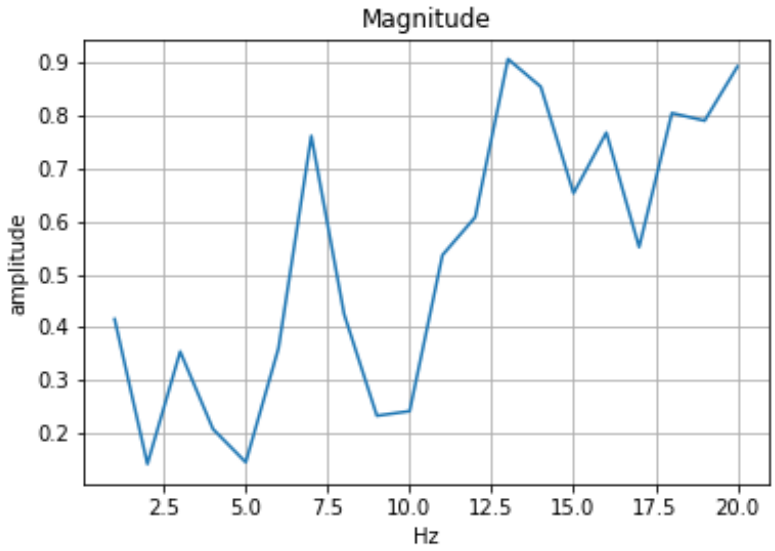

Figure 6: Magnitude from 0 to 20 Hz of a sample

Dimensionality reduction

Dimensionality reduction is very important first of all for better results when testing in real life situations and also for reducing the training time of the model. First we made a correlation plot of all features. As you can see in figure 7 x, y and z axis of the acceleration is highly correlating with each other. So it is not necessary to take all of them.

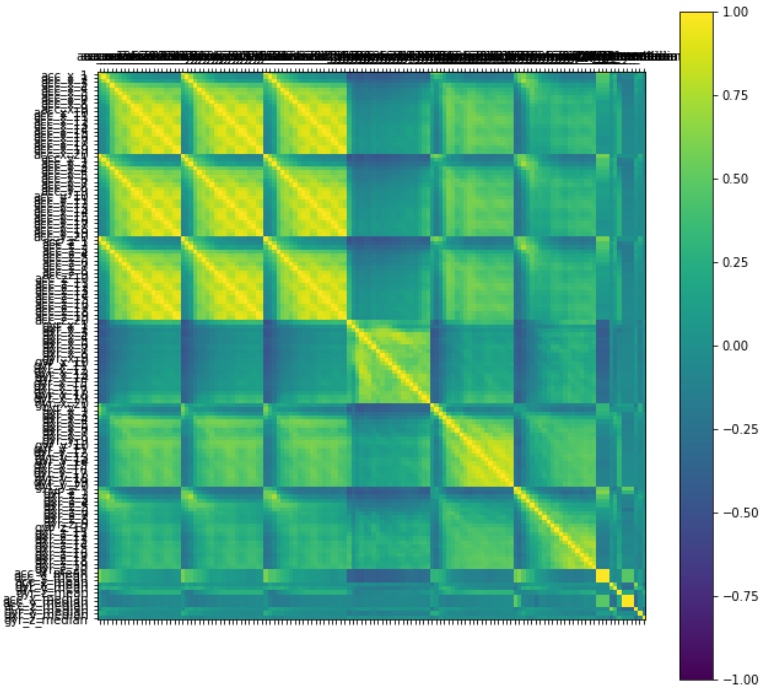

Figure 7: Correlation of features

So after removing correlating features only the acceleration of the x axis is used for further training. The following figure 8 shows all features which are used later on.

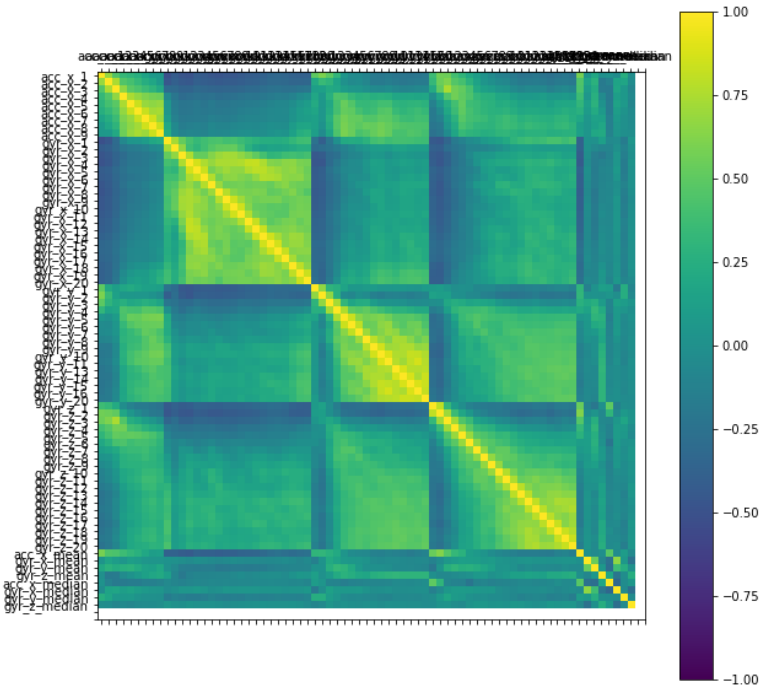

Figure 8: Final features by removing highly correlating ones

Model training and testing

Detect gestures

For detecting gestures we trained a Random Forest and a KNN model. For the RF model we used a grid search with hyper-parameters:

- estimators with values from 3^1 to 3^{18}
- maximum features with auto and sqrt
- maximum depth with values from 3^1 to 3^8

The best hyper-parameters, as you can see in figure 9, is maximum depth of 27 with maximum features auto and 512 estimators, which ends in a score of about 90%.

```
# RF
rf = get_rf(X_train, y_train)
print("[INFO] grid search best score: {}".format(rf.best_score_))
print("[INFO] grid search best parameters: {}".format(rf.best_params_))
acc = rf.score(X_test, y_test)
print("[INFO] grid search test accuracy: {:.2f}%".format(acc * 100))
```

```
Fitting 3 folds for each of 238 candidates, totalling 714 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed:   52.2s
[Parallel(n_jobs=-1)]: Done 349 tasks      | elapsed:  2.5min
[Parallel(n_jobs=-1)]: Done 632 tasks      | elapsed:  4.8min
[Parallel(n_jobs=-1)]: Done 714 out of 714 | elapsed:  5.8min finished
```

```
[INFO] grid search best score: 0.9010989010989011
[INFO] grid search best parameters: {'max_depth': 27, 'max_features': 'auto', 'n_estimators': 512}
[INFO] grid search test accuracy: 83.33%
```

Figure 9: Training RF

The tests were done with a gallery independent test set because we wanted to simulate a real-life scenario. The Random Forest model achieved a test accuracy of 83.33%. As shown in figure 10 the Random Forst predicts the gestures very good but has some problems when it comes to swiping left or right. But overall the model is very strong when prediciting gestures.

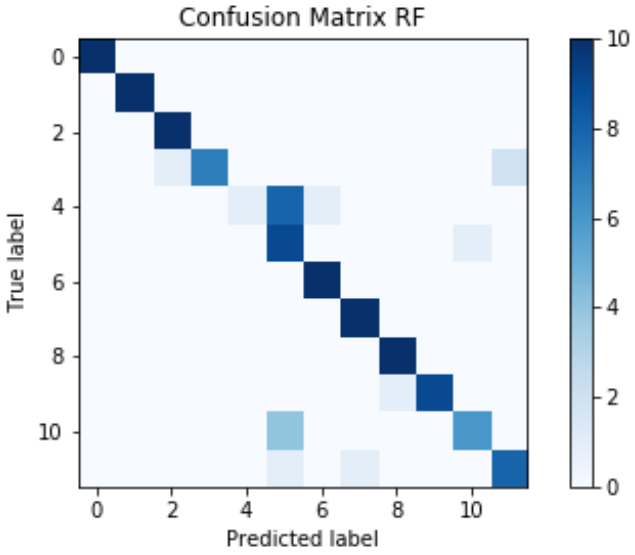

Figure 10: Testing RF

For the KNN model we used a grid search with nearest neighbours ranging from 1 to 100. The best hyper-parameters, as you can see in figure 11, is nearest neighbor of 4, which ends in a score of about 92%.

```
# KNN
knn = get_knn(X_train, y_train)
print("[INFO] grid search best score: {}".format(knn.best_score_))
print("[INFO] grid search best parameters: {}".format(knn.best_params_))
acc = knn.score(X_test, y_test)
print("[INFO] grid search test accuracy: {:.2f}%".format(acc * 100))
```

```
Fitting 5 folds for each of 99 candidates, totalling 495 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 160 tasks      | elapsed:    0.9s
```

```
[INFO] grid search best score: 0.9208333333333334
[INFO] grid search best parameters: {'n_neighbors': 4}
[INFO] grid search test accuracy: 65.83%
```

```
[Parallel(n_jobs=-1)]: Done 495 out of 495 | elapsed:    2.2s finished
```

Figure 11: Training KNN

The KNN model is better than the Random Forest one, but only in the training set. As the score of the test result shows (also in figure 11) the model is overfitting, because the test result is only 65.83%. Therefore the confusion matrix (figure 12) is also not quite good.

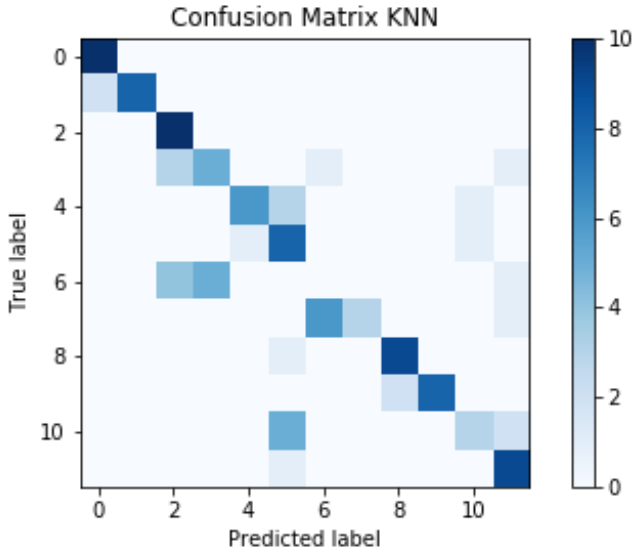

Figure 12: Testing KNN

Binary classifier

Since we defined a set of gestures our system has to be able to distinguish, we as well faced a common problem in such "recognition" applications – we needed a way to find out whether a pre-processed data sample is part of the target population or not. In our case, this meant we had to find out if the recorded movement could be a valid gesture or not, before doing the actual classification.

To do so we decided to use a one-class support vector machine (OC-SVM) as binary classifier. Similar to other classifiers, the OC-SVM is used to "range in" a new data

value into an existing set of classes. But in difference to other classifier – and as the name suggests – the OC-SVM does not have many different classes. In fact, it only decides whether a new value is similar to the ones in the training set or not. Based on this "similarity score" it decides whether the value is valid or not. Again, in our case, this means the distinction between actual gestures (we want to classify) and random movement. An example of such a classification can be seen in the figure 13. Here, the two valid ranges of data and the invalid outliers are clearly visible.

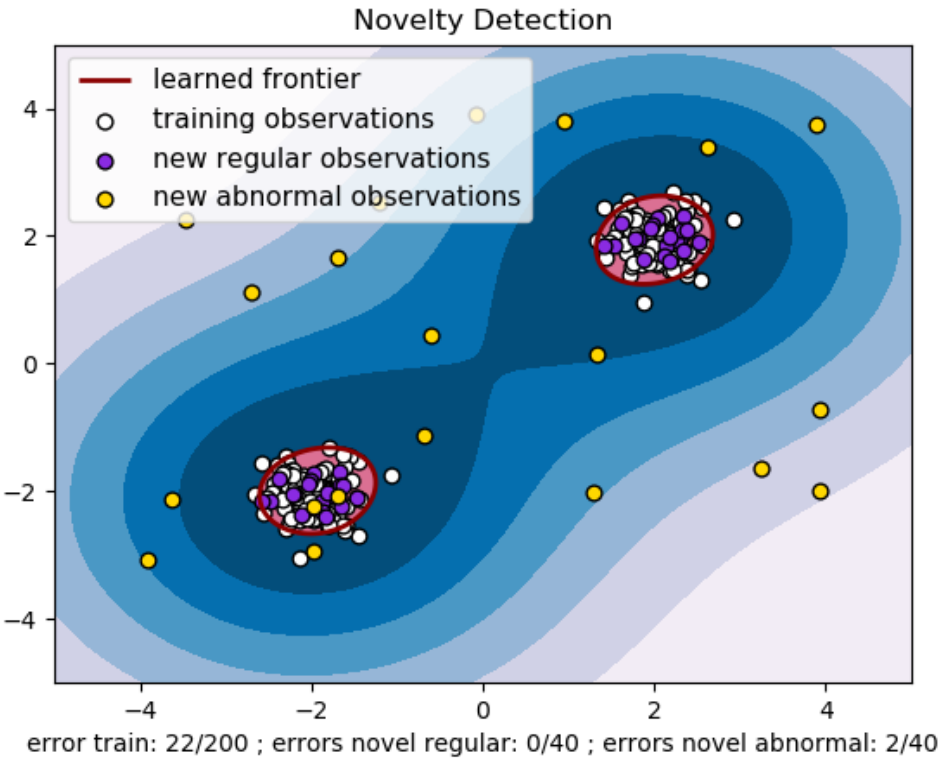

Figure 13: Example of a one-class SVM classification (https://scikit-learn.org/stable/auto_examples/svm/plot_oneclass.html)

When training the OC-SVM, we encountered a problem, namely: To be able to distinguish between valid and invalid gestures, we needed examples for both. Sadly, our dataset only consisted of valid gestures, so we had to improvise a bit to generate invalid data from our dataset. In the end, we decided to just use the valid, preprocessed data as basis. By randomly combining chunks of the data, we created new samples, which represent invalid gestures and – generally speaking – random movement. But sadly, this limits the validity of the testing results a little bit, since real data could potentially look different. Here is a detailed list of steps we took to train the OC-SVM:

1. Add a new column to the data for discerning between valid and invalid gestures
2. Generate invalid gestures (approx. 10% of the data, batch size 3)
3. Separate the target label (the new column) and remove it from the dataset again

4. Determine the outlier ratio (approx. 10%)
5. Do a train- test-split
6. Train the OC-SVM and test it afterwards

The random data creation works as follows:

1. Draw a random row of data from the dataset
2. Take a batch of data out of the drawn row (e.g. three values)
3. Add those to the newly created, invalid row
4. Draw another random row of data and go back to step 1
5. Quit after the complete row has been filled with data

The evaluation results of the trained OC-SVM can be seen in figure 14. They seem okay, but as already mentioned above, more evaluations with real data values would be helpful to further clarify and verify the results. Since the invalid gestures are generated randomly, the results change a bit when re-running the training.

```
### One-Class SVM
preds = ocsvm.predict(test_data)
targs = test_target

print("accuracy: ", metrics.accuracy_score(targs, preds))
print("precision: ", metrics.precision_score(targs, preds))
print("recall: ", metrics.recall_score(targs, preds))
print("f1: ", metrics.f1_score(targs, preds))
print("area under curve (auc): ", metrics.roc_auc_score(targs, preds))
```

```
accuracy:  0.8863636363636364
precision:  0.9487179487179487
recall:  0.925
f1:  0.9367088607594937
area under curve (auc):  0.7125
```

Figure 14: Results of the OC-SVM evaluation

Exporting models for iOS

After training the models they have to be converted to coreml models in order to be used in an iOS application. So the coremltools for python takes an existing model and converts it. Then it can be imported into XCode and used by the application. The whole chain can be seen in figure 15.

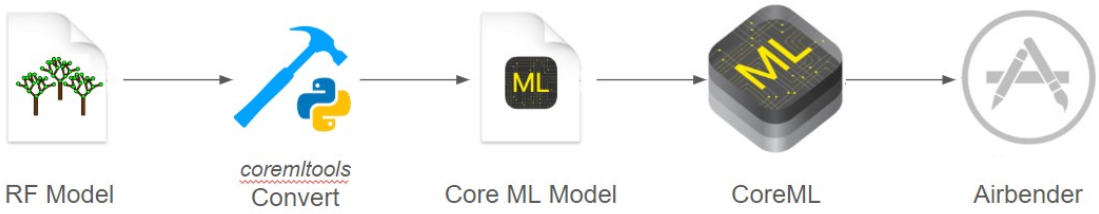

Figure 15: Procedure of exporting a model for iOS

Conclusion

The project shows that detecting gestures is basically not a really hard task. You have to know your data and which features are appropriate for this kind of challenge. As we found out only the acceleration of one axis is necessary for our gestures, because the other two mirror the exact same results and therefore correlate with each other. Removing those features was a important step to improve accuracy in test and training results and to lower the training time.

The major challenge was to find a way to separate random actions from valid gestures. Our choice was the OC-SVM binary classifier. After generating random actions, which are not truly random, we hit a high accuracy, but it doesn't really reflect a real-life scenario. Wearing a watch a whole day while recording would offer much more usable random data, that would be better suited for this case. That's one thing that could be improved.

Based on the gallery independent test result the best performing model was the RF. Therefore we assume that this is the best model for this task. We are confident that using RF and better feature engineering the application could be also improved.