User Generated Exercise Counters

Fabian Ahorner (k1255179), f.ahorner@gmail.com

October 9, 2016

1 Introduction

The goal of this project was to create an Android app that is capable of learning new exercise types and then counting the number of repetitions for the trained type. The focus was on using the minimum amount of training samples to allow the simple creation of new exercise types by the user. This approach does not force the user to do several repetitions of an exercise nor does it require them to train a machine learning algorithm on their phone. The number of training samples was set to one to allow the easiest creation of new exercise counters.
The implementation was done in Java. This includes a desktop Java application to record a dataset and to test iterations of the algorithm. The final algorithm was wrapped in an Android application that focuses on an easy creation and use of exercise counters. It can be downloaded at https://play.google.com/store/apps/details?id=com.bitflake.allcount

2 Dataset

All data was collected from phone acceleration sensors. Although more complicated sensors like gyroscopes are available on some phones, they are not widely available. As a result, only the direction and magnitude of gravity relative to the phone are known. Therefore only acceleration differences are available for evaluation. Additionally, rotations around the direction of gravity of the phone can not be detected. The acceleration sensor generates 20-100, depending on the phone, 3D-vectors per second. For testing purposes a total of 396 exercise repetitions were recorded. These include:

- 88 Dips by 4 different people (125+231+286+119 = 725 combinations)
- 98 Pushups by 3 different people (230+286+286 = 802 combinations)
- 59 Situps by 3 different people (156+81+216 = 453 combinations)
- 150 Squats by 5 different people (384+324+216+216+189 = 1329 combinations)

Each person was free to record the exercises in a different way. Some e.g. recorded them with their phone in the front or back pocket, or in their hand. Each person recorded several single repetitions that could be used to create a counter out of them and multiple sequences with several repetitions in a row. During the recordings the participants often had a break. Therefore the phones position was not guaranteed to be the same over all recordings.

3 Testing

For testing purposes each of the single recordings of one person was used to count the repetitions in all of his other recordings of the same exercise. When D is a list with the number of repetitions per recording, then the number of possible combinations can be calculated as follows:

$$(\sum^{rep \in D: c=1} c)! + \sum^{rep \in D: r>1} r * (\sum^{c \in D: rep=1} c)$$

This inflates the number of possible test combination to a total of 3309. During the development only a limited number of samples was used as a training set and the rest was reserved as a test set. For each combination the squared difference of counted number and the total number of repetitions was summed up to create a mean squared error value for each exercise per person. Then the average MSQE was calculated for each exercise and an equally weighted total MSQE was calculated for the final value. This means that each exercise is weighted equally for the final performance measure.

3.1 Training Dataset

The training dataset included the following data and had a total of 1203 combinations:

- 43 Dips by 2 different people (125+119 = 244 combinations)
- 46 Pushups by 1 person (230 combinations)
- 26 Situps by 1 person (156 combinations)
- 75 Squats by 2 different people (384+189 = 573 combinations)

3.2 Test Dataset

The test data had a total of 2106 combinations and was collected while working on the algorithm and provides more single recordings and has therefore more possible combinations. The test dataset included:

- 46 Dips by 2 different people (231+250 = 481 combinations)
- 52 Pushups by 2 different people (286+286 = 572 combinations)
- 33 Situps by 2 different people (81+216 = 297 combinations)
- 75 Squats by 3 different people (324+216+216 = 573 combinations)

4 Testing/Debugging application

For evaluating the results of the algorithm two Java applications were developed.

- The first application allows to find optimal values, for certain parameters defined in "CountSettings.java", by running a grid search on multiple processors at once.
- The second application ("Gui.java") is a GUI application that was created to simplify certain tasks and allow further explorations of the dataset. It was not written with userbility in mind and is mainly controlled by making changes to the source code. The tool was used for:
  - Looking at a graphical representation of recordings and extracted counters.
  - Recording acceleration values from a phone through a TCP connection and annotating/storing them in a file.
  - Testing combinations of counters and recordings and showing the current count value and other internal values of the algorithm in a graph.
  - Running the algorithm on the whole dataset and showing mistakes made in a list that allowed to further investigate the failed combinations of counters and recordings.

5 Algorithm

The original concept was to extract states from a recording, build a primitive probabilistic model out of them and then use a particle filter on the model during counting. Once the number of particles in the last state reaches a certain threshold, the counter is increased and the particles are reset. During the implementation, several pitfalls were found and the concept had to be adopted to deal with these problems. This section tries to explain the concepts and evolution of the first algorithm to the final implementation.

5.1 Extracting states from recording

The data from the acceleration sensor is available as list of 3D-vectors (=States). To reduce the computational effort the number of samples per second was reduced to 20. Thereby all samples in between were combined to their average value.
Listing 1 shows the Java code that was used to reduce the number of states further.

Listing 1: record/StateExtractor:compressStates

```java
    public static List<CountState> compressStates(List<CountState>
        states) {
        if (states.isEmpty())
            return new ArrayList<>();
        List<CountState> newStates = LandmarkExtractor.getLandmarks(
            states);
        removeSimilarStates(newStates);
        equalizeFirstAndFinalState(newStates);
        addTransientStates(newStates);
        return newStates;
    }
```

- First only the most important states are extracted. Thereby states are removed that can easily be interpolated or don't add further structure to the counter. The recursive implementation for this algorithm can be found in listing 2.

Listing 2: record/LandmarkExtractor.java

```java
public class LandmarkExtractor {
    private final List<CountState> originals;
    private final List<CountState> landmarks;
    private final double minDistance;

    public LandmarkExtractor(List<CountState> originals) {
        this.originals = originals;
        this.landmarks = new ArrayList<>();
        this.minDistance = getMaxStartDistance(originals) * 0.3;
    }

    private static double getMaxStartDistance(List<CountState>
        states) {
        if (states.size() == 0)
            return 0;
        CountState firstState = states.get(0);
        double distance = 0;
        for (int i = 1; i < states.size(); i++) {
            distance = Math.max(distance, firstState.getDistance(
                states.get(i).values));
        }
        return distance;
    }

    public void extractLandmarks() {
        landmarks.add(originals.get(0));
        addLandmarks(0, originals.size() - 1);
        landmarks.add(originals.get(originals.size() - 1));
    }

    public void addLandmarks(int from, int to) {
        double[] start = originals.get(from).values;
        double[] end = originals.get(to).values;
        double[] lastState = start;
        double totalGradient = 0;
        double[] interpolated = new double[start.length];
        double maxDistance = 0;
        int newLandmark = -1;

        for (int i = from + 1; i < to; i++) {
            CountState state = originals.get(i);

            // Find the state with the maximum distance to the interpolation of the start and the end
            double percentage = (i - from) / (double) (to - from);
            interpolate(interpolated, start, end, percentage);
            double distance = state.getDistance(interpolated);
            if (distance > maxDistance) {
                maxDistance = distance;
                newLandmark = i;
            }

            //Sum up the gradients of all states between the start and the end
            totalGradient += getGradient(lastState, state.values);
            lastState = state.values;
        }

        totalGradient += getGradient(lastState, end);
        double interpolatedGradient = getGradient(start, end);

        // Calculate how similar the gradient between the start
        // and end state is to the real gradient of all states in between
        double gradientSimilarity = Math.abs(totalGradient / interpolatedGradient - 1);

        if (maxDistance > minDistance && gradientSimilarity > 0.2) {
            addLandmarks(from, newLandmark);
            landmarks.add(originals.get(newLandmark));
            addLandmarks(newLandmark, to);
        }
    }

    private static double getGradient(double[] last, double[] current) {
        double gradient = 0;
        for (int j = 0; j < current.length; j++) {
            gradient += Math.abs(current[j] - last[j]);
        }
        return gradient;
    }

    private static void interpolate(double[] result, double[]
            start, double[] end, double percentage) {
        for (int j = 0; j < start.length; j++) {
            result[j] = start[j] + (end[j] - start[j]) *
                percentage;
        }
    }

    public List<CountState> getLandmarks() {
        return landmarks;
    }

    public static List<CountState> getLandmarks(List<CountState>
            states) {
        LandmarkExtractor extractor = new LandmarkExtractor(
            states);
        extractor.extractLandmarks();
        return extractor.getLandmarks();
    }
}
```

- In listing 3, states that have a less than average distance to its next state are removed.

Listing 3: record/StateExtractor:removeSimilarStates

```
    private static void removeSimilarStates(List<CountState>
            states) {
        double minDistance = computeSimilarityBoundary(states);
        Iterator<CountState> it = states.iterator();
        CountState last = it.next();
        while (it.hasNext()) {
            CountState next = it.next();
            if (last.getDistance(next) > minDistance) {
                last.setNext(next);
                last = next;
            } else {
                it.remove();
            }
        }
        last.setNext(null);
    }

 private static double computeSimilarityBoundary(List<CountState>
          states) {
        double distSum = 0;
        for (int i = 0; i < states.size() - 1; i++) {
            CountState current = states.get(i);
            current.setNext(states.get(i + 1));
            distSum += current.getDistanceToNext();
```

```java
        }
        states.get(states.size() - 1).setNext(null);

        double distMean = distSum / states.size();
        double sd = 0;
        for (int i = 0; i < states.size() - 1; i++) {
            CountState current = states.get(i);
            sd += Math.pow(current.getDistanceToNext() - distMean, 2);
        }
        return distMean - Math.sqrt(sd / states.size()) / 3;
    }
```

- To prevent particles jumping over the next state to a more convenient later one in case the current value is in between two states additional states were introduced. A additional transient state was added between all neighbouring states. This transient state was computed from the mean of its two new neighbours and was allowed to be overstepped in the later generated model (Listing 4).

Listing 4: record/StateExtractor:addTransientStates

```java
    private static void addTransientStates(List<CountState> newStates) {
        int i = 0;
        int id = 0;
        CountState last = newStates.get(i++);
        while (i < newStates.size()) {
            CountState next = newStates.get(i);
            last.setId(id++);
            CountState l = last;
            for (int j = 0; j < CountSettings.TRANSIENT_STATES; j++) {
                double p = 1. / (1 + CountSettings.TRANSIENT_STATES) * (j + 1);
                CountState transientState = new CountState(last, next, id++, p);
                l.setNext(transientState);
                newStates.add(i++, transientState);
                l = transientState;
            }
            l.setNext(next);
            last = next;
            i++;
        }
        last.setId(id);
    }
```

- Finally the first and last state were both set to the same value to prevent particles moving from the start to the goal state over time. Since both states should already be a very similar value their mean was used (Listing 5).

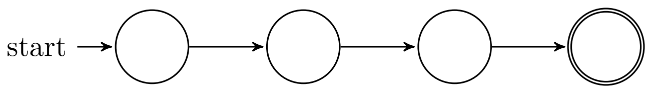
Listing 5: record/StateExtractor:equalizeFirstAndLastState

```java
    private static void equalizeFirstAndFinalState(List<CountState> newStates) {
        CountState firstState = newStates.get(0);
        CountState finalState = newStates.get(newStates.size() - 1);
        for (int j = 0; j < finalState.values.length; j++) {
            firstState.values[j] = finalState.values[j] = (firstState.values[j] + finalState.values[j]) / 2;
        }
    }
```

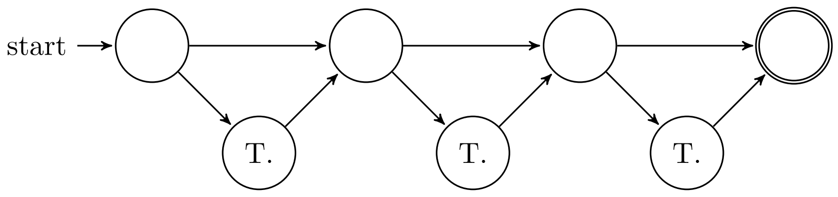
5.2 Probabilistic model

Throughout the implementation several different models were tested:

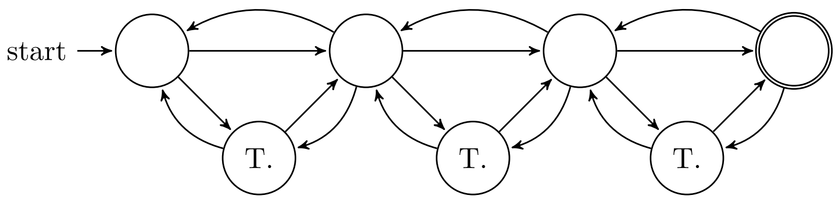
1. The first version was a simple sequence that only allowed particles to travel from one state to the next.



2. In the second version transient states were introduced that could be skipped by particles. Different numbers of transient states in between normal states were tested but the best results were achieved with one transient state. **This was also the final version used.**



3. The next version allowed particles to travel backwards. Again, transient states could be skipped.



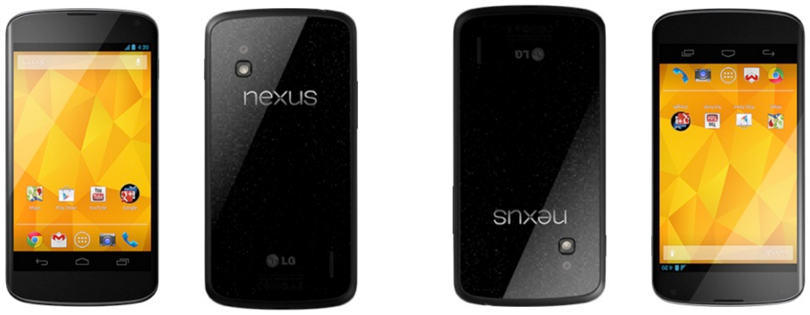


Figure 1: Easy to mix up phone positions

5.3 Particle Filter

Different versions of particle filters were used throughout the development of the application. Up to 100 Particles were moved by calculating a score for each reachable state. In most cases this score was calculated by using the euclidean distance of the current acceleration and the states acceleration. This score was then used for a roulette wheel selection. Different re-sampling strategies were tested. One of them can be found in the package *com.bitflake.counter.algo.shared.old.SensorCounter*. The main problems of this approach were that particles started to jump over states by chance and then ending up in a closer state than the actual one. To compensate this it was tried to use the cumulated error of each particle for re-sampling. This resulted in all particles being re-sampled to previous closer states during breaks in the exercise sequence. Additionally the issue of different initial starting positions (Further discussed in section 5.4) would require a high number of particles for each possible possible starting position. This would have required a number of particles that was not computational efficient anymore. Because of all these issues a different approach was chosen in the end.

5.4 Initial phone rotations

During the development of the particle filter it was discovered that the main issue while recognising movements was that the phone was not perfectly moved back in the starting position. This could easily happen when the phone moved in someone pocket or it was simply put in slightly rotated. This initial rotation of the phone meant that all reference states from the recording were off and the particles were more likely to jump to other less likely states. An additional problem was that it was easy for users to forget which of the four phone positions, shown in figure 1, was used while recording the exercise.

The first method to deal with this issue was to use the same particle filter described in section 5.3, but using differently rotated versions of the reference states. In particular whenever the phone was moved a new version was created by calculating a rotation matrix that would move the first reference state to the current position of the phone. This rotation matrix was then applied to all remaining reference states which were then assigned to a particle in the first state. If one of these newly generated particles was

successful it would be re-sampled a lot and other particles would be assigned the same initial rotation. Although this seemed to work better on some test recordings, especially with slightly rotated starting positions, the different particle versions meant that less particles were available to find the right position within the current exercise routine.
Additionally it turned out that this method did not work for all possible phone rotations. The problem with the first approach was that it only used the first reference state to rotate the remaining states. For the rotation of a vector to another direction it is sufficient to rotate the vector only around two axis. But if it is a group of vectors it can be rotated around the X,Y and Z axis to create a new version. For the phone this means that if it was also rotated around the direction of the first state, the rotation is ignored in this version. To clarify, if there is an additional rotation around the first state, the first state is not altered but all other states that point in a different direction are rotated. Using the acceleration sensor only it is not possible to detect if the phone is rotated around the vector of acceleration. To get this additional information a gyroscope that provides the phones state as a quaternion would be required.
A primitive extension to the discussed particle filter would be to also create around the start state rotated counter versions and assigning them to particles. To counteract the smaller number of particles focusing on the rightly rotated version the number of total particles would have to be increased again. Since the application is supposed to run on a phone, a different approach was required.

5.5 Rotation Collection

The implementation of this final algorithm can be seen in listing 6. For the computation of the rotations the **Apache Commons Math** library was used.

Listing 6: count/RotationCollection:analyseValues

```
    @Override
    public void analyseValues(double[] values) {
        currentValues = new Vector3D(values).normalize();
        updatePotentialRotations(currentValues);
        addPotentialRotation(currentValues);
        updateLikelihoods(currentValues);
        sortOutRotations();
        checkCount();
        if (countListener != null)
            countListener.onCountProgress(0);
    }
```

5.5.1 Finding the right rotation

When counting is started all four rotated versions of the start state (See figure 1) are generated. When a new acceleration value is received from the sensor a new potential rotation is created, if the following conditions apply (Listing 7):

Listing 7: count/RotationCollection:addPotentialRotation

```java
    private void addPotentialRotation(Vector3D v) {
        if (isValidStartingPoint(v) && isNewStartingPoint(v)) {
            CounterRotation rotation = new CounterRotation(
                stateValues, v);
            potentialRotations.add(rotation);
        }
    }
```

- The direction of the vector is within a certain angle of one of the four flipped start versions created in the beginning. It is assumed that if the phones rotation diverges too much from the recorded starting position it can be ignored (Listing 8).

Listing 8: count/RotationCollection:isValidStartingPoint

```java
    private boolean isValidStartingPoint(Vector3D v) {
        bestStartAngle = Double.MAX_VALUE;
        for (int i = 0; i < flippedStartVersions.length; i++) {
            double angle = Math.abs(Math.acos(v.dotProduct(
                flippedStartVersions[i])));
            bestStartAngle = Math.min(bestStartAngle, angle);
            if (angle < CountSettings.VALID_START_ANGLE) {
                return System.currentTimeMillis() > 0;
            }
        }
        return false;
    }
```

- There is no other previously created potential rotation that points in a similar direction. This limits the number of the rotations that have to be observed (Listing 9).

Listing 9: count/RotationCollection:isNewStartingPoint

```java
    private boolean isNewStartingPoint(Vector3D v) {
        for (CounterRotation pR : potentialRotations) {
            double d = pR.getFirstStateDistance(v);
            if (d < CountSettings.MIN_START_DISTANCE) {
                return false;
            }
        }
        return true;
    }
```

If these conditions apply the rotation from the start reference state to the current value is saved as an potential rotation. Since this does not include a rotation around the start state additional information is required. Therefore, whenever the sensor generates a new value all potential rotations are checked in listing 10.

Listing 10: count/RotationCollection:updatePotentialRotations

```java
    private void updatePotentialRotations(Vector3D v) {
        Iterator<CounterRotation> it = potentialRotations.iterator();
        while (it.hasNext()) {
            CounterRotation rotation = it.next();
            double originalAngle = rotation.getOriginalAngle();
            double currentAngle = rotation.getCurrentAngle(v);
            if (currentAngle > originalAngle * CountSettings.
                MIN_FIRST_ANGLE) {
                rotation.rotateSecondState(v);
                double angle = Math.abs(rotation.getSecondRotation())
                    ;
                if (angle < CountSettings.VALID_START_ANGLE || angle
                    > Math.PI - CountSettings.VALID_START_ANGLE) {
                    activeRotations.add(new RotatedCounterVersion(
                        rotation, states));
                }
                it.remove();
            }
        }
    }
```

Then the angle between the creation of the potential rotation and the current measured acceleration are computed. If it is at least within a certain range of the angle between the first and second reference state, it is removed from the potential rotations. Additionally this information can now be used to calculate the missing rotation around the start state. If this rotation is again within a certain range the potential rotation is used to create a new rotated counter version.
These counter versions are fed with all sensor values to compute the likelihood of being in a certain state.

5.5.2 Computing state likelihoods

The computation of the likelihood of being in a certain state for each rotated version can be seen in listing 13. By enabling line 55 it is possible to allow the most likely state to move backwards but it was determined through experiments that this decreases the performance. Additionally it can be seen that transient states have a limited reach.

Listing 11: count/RotatedCounterVersion:updateLikelihoods

```java
    public void updateLikelihoods(Vector3D values) {
        updateStateScores(values);
        for (int i = 0; i < likelihoods.length; i++) {
            CountState state = states.get(i);
            int reach = state.isTransientState() ? 1 : 2;
            int from = i;
//            int from = Math.max(0, i - reach);
            int to = Math.min(i + reach, likelihoods.length - 1);
            updateLikelihood(likelihoods[i], from, to);
        }
        swapTmpLikelihoods();
    }
```

First a score is calculated for each state in listing 12. This is done in the same way as it used to be calculated for the particle filter. Thereby the parameter "MOVE_DRIVE" in line 102 controls with what probability a particle moved to a neighbouring state with a similar/better probability.

Listing 12: count/RotatedCounterVersion:updateStateScores

```
91      private void updateStateScores(Vector3D values) {
92          double cumulated = 0;
93          for (int i = 0; i < likelihoodsTmp.length; i++) {
94              stateDistances[i] = getStateDistance(values, i);
95              stateScores[i] = getStateScore(stateDistances[i]);
96              cumulated += stateScores[i];
97              stateScoresCumulated[i] = cumulated;
98          }
99      }
100
101     private double getStateScore(double distance) {
102         return Math.exp(-CountSettings.MOVE_DRIVE * distance);
103     }
104
105     private double getStateDistance(Vector3D values, int state) {
106         return stateValues[state].distance(values);
107     }
```

The first tested approach was to create a behaviour that would be have in the same way as the particle filter but with less computational effort. This implementation can see in the commented section in listing 13. By conducting several test to find the right "MOVE_DRIVE" it was determined that the results were improving the higher the drive was. This means that particles would move with a high likelihood to the state with the best score (=smallest euclidean distance). As an result of this observation the multiple particle approach was given up and the state with the lowest distance to the current acceleration was selected. This did not work well when no rotated versions of the counter were used because the particles got stuck in local minimals to easily. But by using differently rotated versions of the counter all states are closer to the observed values and investing unlikely current states becomes less important.

Listing 13: count/RotatedCounterVersion:updateLikelihood

```
    private void updateLikelihood(double likelihood, int from, int to
        ) {
```

```java
//          double scoreSum = stateScoresCumulated[to] -
    stateScoresCumulated[from] + stateScores[from];
//          for (int i = from; i <= to; i++) {
//              likelihoodsTmp[i] += stateScores[i] / scoreSum *
    likelihood;
//          }
        double bestScore = 0;
        int bestScoreIndex = -1;
        for (int i = from; i <= to; i++) {
            if (bestScore < stateScores[i]) {
                bestScore = stateScores[i];
                bestScoreIndex = i;
            }
        }
        likelihoodsTmp[bestScoreIndex] += likelihood;
    }
```

5.5.3 Detecting one repetition

After updating the likelihood of all active rotated counter versions, versions who's most likely state has a too high distance to the currently observed state are being sorted out in listing 14.

Listing 14: count/RotationCollection:sortOutRotations

```java
    private void sortOutRotations() {
        Iterator<RotatedCounterVersion> it = activeRotations.iterator
            ();
        while (it.hasNext()) {
            RotatedCounterVersion version = it.next();
            double d = version.getBestDistance();
            if (d > CountSettings.SORT_OUT_DISTANCE) {
                it.remove();
            }
        }
    }
```

Finally it is checked if the likelihood of being in the last state is above a certain boundary for any rotated version. In the final version this just translates to checking if the predicted current state of any version is the last state. If this is the case one count is performed and all potential and active rotations are being removed (Listing 15).

Listing 15: count/RotationCollection:checkCount

```java
    private void checkCount() {
        RotatedCounterVersion bestVersion = null;
        double bestLikelihood = 0;
        for (RotatedCounterVersion version : activeRotations) {
            if (bestVersion == null || version.getGolaLikelihood() >
                bestLikelihood) {
```

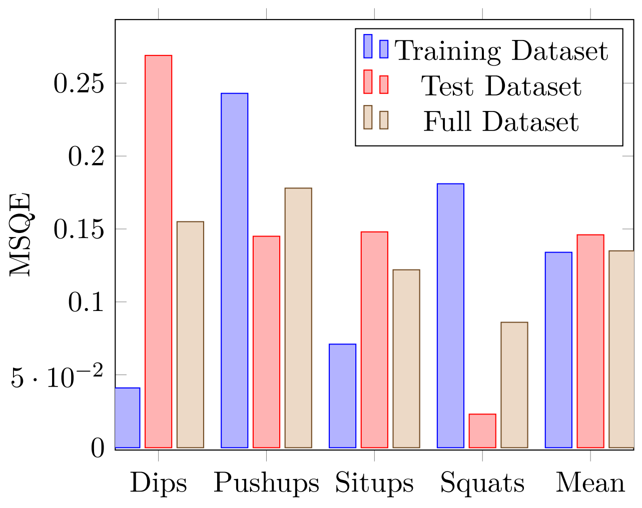


Figure 2: Results of the final algorithm

```
                bestVersion = version;
                bestLikelihood = version.getGolaLikelihood();
            }
        }
        if (bestLikelihood > CountSettings.MIN_GOAL_LIKELIHOOD) {
            performCount(bestVersion);
        }
    }

    private void performCount(RotatedCounterVersion bestVersion) {
        bestVersion.reset();
        activeRotations.clear();
        potentialRotations.clear();
        count++;
        if (countListener != null)
            countListener.onCount(count);
    }
```

6 Results

Figure 2 shows the results of the final algorithm being applied to the training, test and the full dataset. It can be seen that the algorithms performance is different for different exercises on the test and training set. This can either indicate that the algorithm does not overfit or that the dataset is too small. By looking at the results of the algorithm on all different recorded versions in table 1 it can be seen that some exercises seem to be recognised better then the same exercise executed by a different person. Since the number of people was limited it has a high effect on the final result if one execution can

Dips	Pushups	Situps	Squats
0.02	0.24	0.07	0.01
0.49	0.08	0.22	0.05
0.04	0.21	0.07	0.02
0.06			0.00
0.06			0.35

Table 1: Detailed results per person on the full dataset

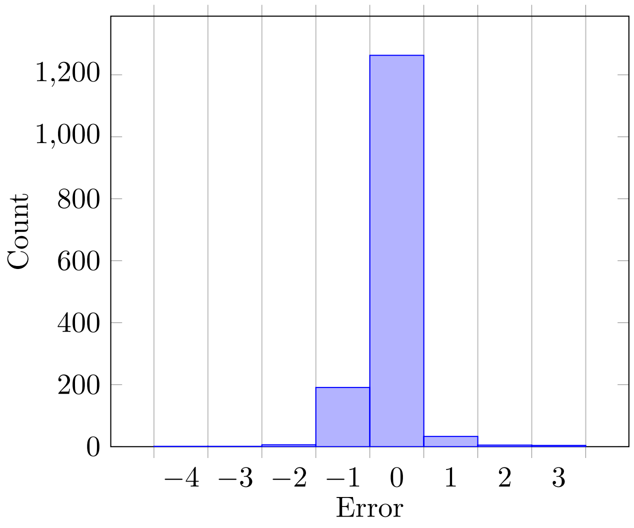


Figure 3: Error distribution of the full dataset

not recognised as good as the other ones.

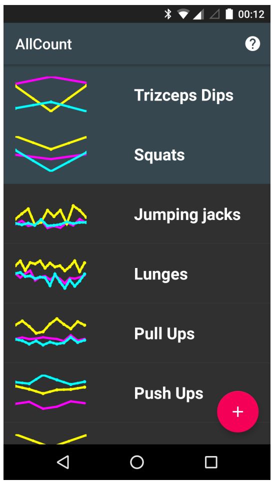
Figure 3 shows the error distribution. It can be seen that the most common error was that one exercise was not counted. Therefore it has more a tendency to count less repetitions than executed.

7 Android application

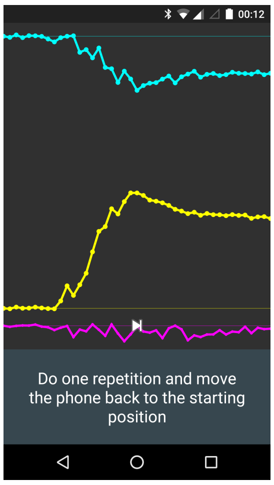
The first time a user opens the app they are welcomed by a tutorial screen that allows them to record new counters. As soon as they have saved one counter the default start screen is figure 4a. This screen allows the user to record a new counter and to select a previously recorded counter. The top of the list shows the most recently used counters.

7.1 Recording

When the user records a new counter they are asked to hold the phone still in its starting position to start the recording. This allows them to put the phone in a pocket or any other convenient place during the exercise. When the phone is kept still for several

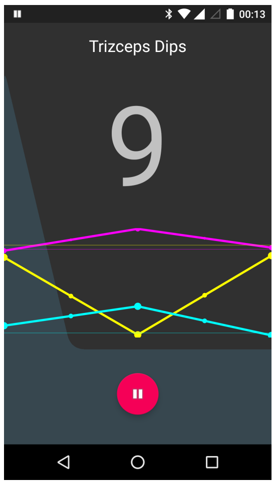


(a) Start screen



Do one repetition and move the phone back to the starting position

(b) Recording screen



(c) Counting screen

Figure 4: Main screens of the android application

seconds it automatically starts recording the users movements. The user is informed to start moving on the screen and by a voice command (Figure 4b). After one repetition of the exercise they want to record for a counter they are asked to move back into their starting position. The phone automatically detects when the phone is moved back to its original position and ends the recording when it has been held still for a couple of seconds again.
Then the application extracts the information that is required to create a new counter from the recording and opens the count screen to allow the user to count further repetitions immediately. Since they have already done one repetition the counter starts with the value 1. To see further details about the count screen see the next section.

7.2 Counting

The counting screen (Figure 4c) shows a simplified version of the movement done when recording the exercise. Three lines show the current acceleration sensor value (X,Y and Z coordinates) and allow the user to see possible diversions from the recording. When one repetition is done, the count value is increased and the according voice feedback is given. Additionally, the user can pause the counter and set or change its name. Counters with a name are automatically saved for later use. To prevent accidental touches of the screen when the phone is in a pocket the counting and recording also work if the screen is turned off.

7.3 Android Wear application

It was planned to provide an accompanying Android Wear application. Since the watch is worn on the wrist it is better capable to record some movements that can not be recorded by a phone in a pocket. This would have provided the opportunity to count further exercises like weight lifting. The goal was to send the sensor value to the phone and evaluate them there. Several different APIs were tested but none of them worked sufficiently. The main reason for this was that the android wear support for fast connections is very limited. Its focus lies on battery saving. Therefore all messages, in this case sensor values, were bundled and only sent after a certain delay. This delay was unpredictable and the the general connection was very unreliable. Because of this a good user experience could not be guaranteed and the Android Wear support was dropped.

8 Summary

Although the basic particle concept implemented in the beginning seemed promising in the beginning, it had some issues that could not be improved by just optimising the basic algorithm. These limitations were responsible for a lot of time wasted trying to optimise different concepts that were not powerful enough to deal with initial phone rotations. The final algorithm seems to be well suited to deal with this. Additionally the algorithm was implemented in an Android application that was published and can be downloaded for free from the Google play store.