Step Counting Research Paper

Naman Bhargava

Abstract:

This paper revolves around creating improvements for a step counting algorithm for different gaits when the data comes from sensors in right pockets. The original algorithm relied on counting steps from peaks that were above the standard deviation value for the data set. The changes used were a peak smoothing deadzone and an adaptive threshold. The changes reduced the percent error by 1.6% from 18.7% to 17.1%. The changes worked, but still require improvement.

Introduction:

Counting steps can have large health benefits on people by quantifying the amount of exercising people do daily. According to Catrine Tudor-Locke, the director of the Walking Behavior Laboratory at Pennington Biomedical Research Center, The Centers for Disease Control and Prevention's recommendation for moderate aerobic physical activity in terms of walking steps translates to a minimum of 7,000 steps daily. However, on average, American adults only walk 5,900 steps daily (Rettner). Step counters can motivate people to improve the number of steps they take daily and show their daily progress. This may inspire them to continue improving their physical activity, thus improving their health and quality of life.

Common pedometer apps are only reliable in special cases. Inaccurate results may discourage people from continuing to improve their physical activity. The need for an accurate step counter accessible to everyone is steadily growing.

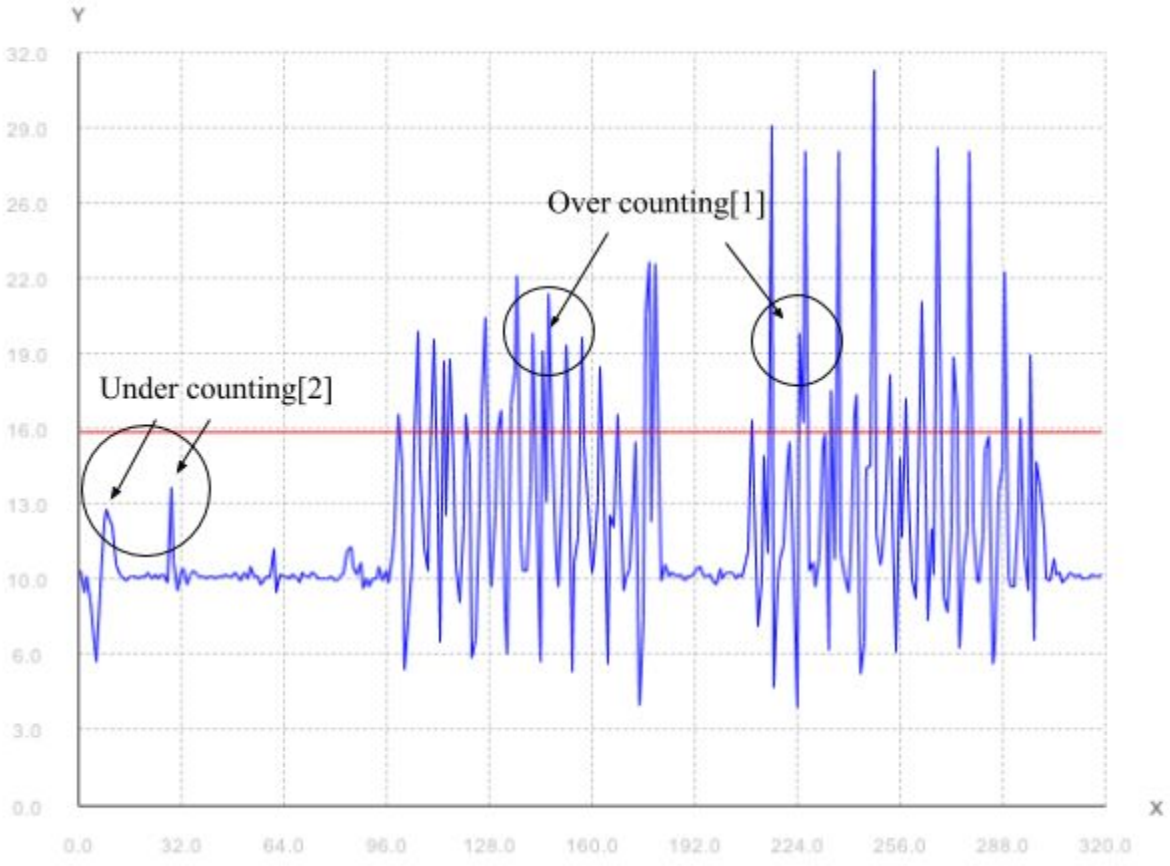
The evolution and spread of smartphones has led to accelerometers and gyrometers increasingly becoming common. This makes data collection readily accessible through phones. This study will focus on developing an algorithm which will accurately count the number of steps one takes in any motion, from walking to running to irregular walking, and more. In question form, what improvements can increase the accuracy of a step counting algorithm for different gaits if the sensor data comes from the right pocket?

Original Algorithm Analysis:

The original basic algorithm to count steps had many faults. It began by calculating the magnitudes of the 3d acceleration data from the x, y, and z components. Then it found the peaks, which are defined as a value which is larger than its left and right neighbors. Next, the algorithm calculated a standard deviation by looping over the array of magnitudes and adding up all the squares of the difference of each value and the mean. Then, it divides the sum by one less than the number of values. The standard deviation is equal to the square root of that value. Finally, the

algorithm finds a threshold value equal to two (a scalar constant to ensure there is a high peak) times the standard deviation. Peaks higher than the threshold were counted as steps.

The old algorithm failed due to overcounting steps during little peaks from noise[1], as well as failing to count light steps[2]. The steps over counted were little peaks within an unrealistically short interval. This meant the peaks identification needed to be modified. The steps under counted were lighter steps, but were under the threshold indicated by the horizontal line. This meant a threshold calculated for the full data will likely fail to count all the steps.



In order to resolve the issue of overcounting peaks, one could ignore a certain time interval around the tallest peaks. By ignoring values very close to peaks, the noise resulting in peaks won't be counted as steps. One could instead only count the peaks as a step, if there is a drop larger than a certain magnitude before the second peak. This could be a distance like returning to the baseline, where the accelerations are constant.

In order to fix the issue of undercounting peaks, one could separate the data into sections where the magnitudes of the peaks are similar. This way, each section would have its own threshold, so patches of smaller magnitude peaks could still be counted as steps, even in the presence of harder steps with higher magnitudes. A second idea would be to identify a constant

frequency of the peaks. The algorithm could approximate where other peaks are expected by the frequency. If there is a peak in the next location, a step would be counted. This process would repeat until the algorithm does not find a peak in the next expected location. This idea works since a constant frequency of peaks might indicate the rate of a constant walking motion. Once the peaks at regular frequencies stop, the constant motion has ended. This would count the steps even if there was a light step while walking normally. A third idea would be to create an adaptive threshold, which would use data in a certain range around the index value to calculate a threshold. This would allow a wider range of magnitudes, as well as irregular pacing.

The modifications actually implemented were the dead zone time interval around peaks, and adaptive thresholds. The dead zone time interval around the peaks made sure steps would only be counted if they were realistically spaced apart. This was the best solution to over counting peaks, since it avoids extra small peaks from noise in the data. It also allows the net magnitude to be increasing or decreasing, as with using stairs. The time interval allows the largest input from other data, while still solving the problem. The adaptive threshold was an important solution, since it allows the largest range of inputs. It would work while individuals have net magnitudes sloped to increasing or decreasing, like during stairs, and still accommodates irregular pacing, like from children. The adaptive threshold relies on data points around the peak, and would be lower if most of the data points reflected the constant baseline of the magnitudes. This could lead to extra peaks identified as steps. The segmented threshold groups corrects the adaptive threshold, the best solution to undercounting peaks.

Testing Plan:

The two types of test conducted were ideal scenario tests, and non-optimal setting tests. The tests compared the actual number of steps to the calculated number by the algorithm. That accuracy determined if the changes to the algorithm increased the accuracy. The ideal scenario tests were run to make sure the algorithm completes what it was designed to do. All simple logic bugs were made apparent here, since the results showed an estimate of the accuracy of the algorithm. The non-optimal setting tests were conducted to increase the range of motions. They were tests that covered the types of motion in an average day. Those tests showed the overall performance of the algorithm, and determined the direction for the next step.

The ideal scenario tests included the forward walking test and the forward running test. Those tests fulfilled the ideal scenario range, since they were the most periodic motions and produced the most constant data. The expected frequencies and magnitudes for peaks were approximately the same. These tests gave a quick estimate for the accuracy of the algorithm.

The non-optimal tests included the irregular walking pace test, the hill walking test, and the walking on stairs test. The irregular walking pace test showed the effect of different frequencies of steps. The hill walking test and the walking upstairs test both showed the effect of different magnitudes of steps. The hill test is needed since it is gradual changes in magnitude. The walking on stairs test is needed since it has periodic changes in magnitude. These tests cover

a majority of normal motions from real life. The range is large enough to cover one's motions throughout the day.

The walking forward test involved a participant walking forward in a straight line. Their height, weight, phone model, and data collection app were noted down. This test was performed on a flat path. The phone was placed into the participant's front right pocket. A proctor instructed the participant to walk at a comfortable pace and counted the number of steps. This number was recorded for later use.

The running forward test involved a participant running forward in a straight line. The height, weight, phone model, and data collection app were noted down. This test was performed on a flat path. The phone was placed into the participant's front right pocket. The proctor instructed the participant to run at a comfortable pace and counted the number of steps. The number was recorded for later use.

The irregular walking pace test involved a participant walking by taking irregularly paced steps. Their height, weight, phone model, and data collection app were noted down. The phone was placed into the participant's front right pocket. A proctor instructed the participant to walk forward and take irregularly paced steps. A proctor counted the steps and recorded them for later use.

The hill walking test involved a participant walking uphill in a straight line. Their height, weight, phone model, and data collection app were noted down. The phone was placed into the participant's front right pocket. A proctor instructed them to walk from the base of the hill to the height of the hill, wait at the top of the hill, wait two seconds, then walk down the hill. The location of the hill was noted down. The proctor counted the number of steps to walk up, the number of steps to walk down and recorded them.

The walking on stairs test involved a participant walking up stairs. Their height, weight, phone model, and data collection app were noted down. The phone was placed into the participant's front right pocket. A proctor instructed them to walk up five stairs, wait two seconds, then walk down the stairs. The proctor verifies the number of stairs.

New Algorithm Analysis:

The improvements of the new algorithm increased the accuracy of the project in most cases. The results of the original algorithm were compared to the results of the new algorithm and displayed in the following table:

Position: right front pocket

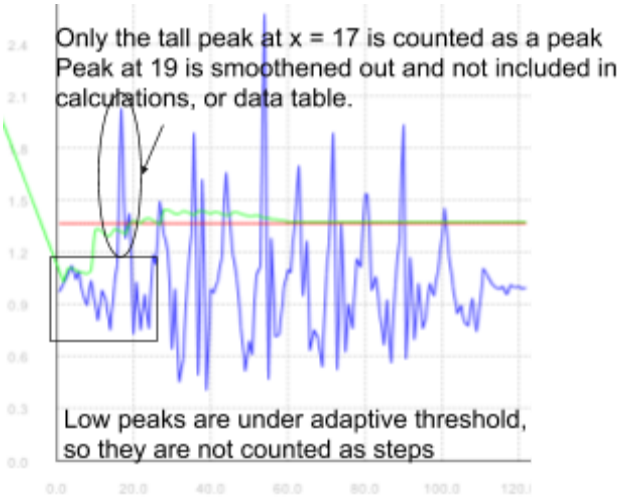
Actual # of steps	Gait	Person who recorded the data	Original # of steps calculated	# of steps calculated after changing algorithm	Which algorithm counted closest to the amount of steps actually taken? (original or fixed?)	Most accurate (counted exact number of

					[percent error of original]	actual steps) [percent error of improved]
10	Jogging	Sabrina	10	11	Original [0%]	Original [10%]
11	Walking down stairs	Sabrina	10	12	Both close [20%]	None [20%]
11	Walking up stairs	Sabrina	10	11	Fixed [20%]	Fixed [0%]
13	Walking up and down stairs	Sabrina	12	12	Both Same [7.7%]	None [7.7%]
15	Forward walking	Sabrina	11	12	Fixed [26.7%]	None [20%]
24	Irregular walking	Sabrina	21	22	Fixed [12.5%]	None [8.3%]
32	Walking up and down hill	Sabrina	29	31	Fixed [9.4%]	None [3.1%]
64	Jogging with phone in pocket	Mr. Dobervich	50	58	Fixed [21.9%]	None [9.4%]
35	Forward walking and	Mr. Dobervich	35	43	Original [0%]	Original [22.9%]

	changing directions					
30	Free conditions with phone in pocket	Classmate	20	20	Both same [33.3%]	None [33.3%]
30	Fast walking with phone in pocket	Classmate	24	25	Fixed [20%]	None [16.7%]
30	Jogging with phone in pocket	Classmate	14	14	Both Same [53.3%]	None [53.3%]
		Average Error	Original:	18.7%	Improved:	17.1%

The results from the tests indicate that the new algorithm improved on most trials. They showed that the percent error for the new algorithm's prediction was less than the old algorithm's errors. The highest percent error was found with a classmate's data, where the error was 53.3%. It was likely an outlier, since all the other tests provided errors of less than 34%. Overall, the new algorithm performed better, as it had an average error rate of 1.6% less. It had an average accuracy of ±4 steps. It was successfully able to count the lower peaks through the adaptive threshold, yet still made sure to avoid counting peaks unrealistically close together.

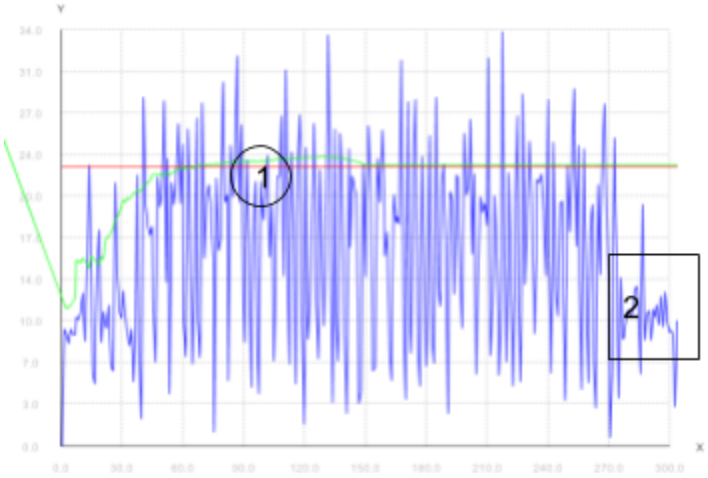
When observing the output graphs of the magnitudes and the thresholds, and the table of the peaks, the algorithm demonstrated it counted the correct peaks. The graph of 100% accuracy in the walking up stairs demonstrated this.



Note:
Green line = adaptive threshold
Red line = old threshold
Blue line = Accelerometer Magnitudes

Console output:
(Magnitudes with * after them meant they were above the threshold and counted as a step)

Peak time	Thresholds	Magnitude
4	1.0947144848477293	1.121293391727207*
9	1.3265192626198807	1.0337484269406294
17	1.304862033856748	2.0236782138771754*
21	1.3651409233752192	1.0252178399435747
27	1.4471548397491452	1.48855699751583*
31	1.424329372810718	0.9918837977757816
36	1.4363196301691326	1.8871022415340493*
40	1.4264347102463633	0.9860736806918463
44	1.4098663259123525	1.6552428459570938*
50	1.4075421516119246	0.6867096406552373
54	1.394757019119384	2.567377280216971*
63	1.3708880146400668	1.698620972275259*
67	1.3708880146400668	0.7518744019379413
72	1.3708880146400668	1.8849512560403312*
80	1.3708880146400668	1.535103786672652*
86	1.3708880146400668	0.9274418896263263
90	1.3708880146400668	1.9316305423017255*
94	1.3708880146400668	1.0219721803156492
101	1.3708880146400668	1.4562112716229627*
111	1.3708880146400668	1.1063887161955654
118	1.3708880146400668	1.0155381945782367



As shown by the graph and console display, the algorithm did a good job ignoring double peaks unrealistically close together, and didn't count the extremely low peaks.

Although the new algorithm performed better, it still had areas of improvement. The adaptive threshold was often not the correct size for the specific data set. This inaccuracy is demonstrated in the 64 steps in pocket jogging test. The graph is shown to the left. In circle 1, the peaks which seem like steps are below the threshold and are not counted. The threshold had too large a window to search the values to calculate the threshold. This led to the undercounting of these steps. In box 2, the threshold did not drop off at the end, which meant the back end range was too large. Continuing improvement, the algorithm would need a method which calculates an appropriate threshold range for the data. This could rely on a combination of the fluctuations between the peak heights and time, rather than just time.

The improved algorithm performed better than the original six times, while performing just as well four times in twelve tests. It was able to perform well in all of the gaits, also showing improvement. The ideas implemented in the new algorithm took a step in the right direction, but still need development for a really accurate step counting algorithm.

Conclusion:

The step counting algorithm's performance was improved with the implementation of a peak deadzone, as well as an adaptive threshold. The original algorithm relied on a threshold calculated by finding the standard deviation from the whole data set. The improvements lowered the average error rate by 1.6%. For future improvements, the algorithm needs a method to calculate the adaptive threshold. The algorithm could also be tested with data from different phone positions once the gait tests were returning reliable accuracy. The improvements took a step in the right direction and will allow people to remain motivated to stay active, and improve their health.

Appendix and References

Code:

```
import java.util.ArrayList;
import java.util.Arrays;

import javax.swing.JFrame;
import org.math.plot.Plot2DPanel;

public class CountSteps {

	private static final int DEADZONE_THRESHOLD = 50;
	private static final int ADAPTIVE_THRESHOLD_RANGE = 20;
	private static final int TIME_THRESHOLD = 150;

	/***
	 * Counts the number of steps based on sensor data.
	 *
	 * @param times
	 *		a 1d-array with the elapsed times in milliseconds for each row
	 *		in the sensorData array.
	 * @param sensorData
	 *		a 2d-array where rows represent successive sensor data
	 *		samples, and the columns represent different sensors. We
	 *		assume there are 6 columns.Columns 0-2 are data from the x, y,
	 *		and z axes of an accelerometer, and 3-5 are data from the
	 *		x,y,z axes of a gyro.
	 * @return an int representing the number of steps
	 */
	public static int countStepsByMagnitudes(double[][] sensorData, double[] times) {
		int stepCount = 0;
		double[] magnitudes = calculateMagnitudesFor(sensorData);
		int[] peaks = findPeaks(magnitudes, times);
		double[] thresholds = calculateThresholds(magnitudes, times);

		for (int i = 0; i < magnitudes.length; i++) {
			if (thresholds[i] > 0.5 && magnitudes[i] > thresholds[i] && peaks[i] == 1)
				stepCount++;
		}

		return stepCount;
	}

	/***
	 * calculates a threshold range using the time data
	 *
	 * @param times the times that each data point was collected
	 * @return the number of values to use as a range
	 */
```

```
private static int calculateThresholdRange(double[] times) {
	int range = 0;
	while (times[range] < ADAPTIVE_THRESHOLD_RANGE)
		range++;
	return range;
}

/***
 * Calculates a threshold value for each magnitude with a new window and returns an
 * array containing all the values
 *
 * @param magnitudes the magnitudes of the acceleration data
 * @param times the times from the data
 * @return an array with threshold values recalculated for each vale
 */
public static double[] calculateThresholds(double[] magnitudes, double[] times) {
	double[] thresholds = new double[magnitudes.length];
	int range = calculateThresholdRange(times);

	for (int i = 0; i < magnitudes.length; i++) {
		double[] magCluster = getMagnitudeCluster(magnitudes, range, i);
		thresholds[i] = calculateThreshold(magCluster, calculateMean(magCluster));
	}
	return thresholds;
}

public static double calculateMagnitude(double x, double y, double z) {
	return Math.pow(x * x + y * y + z * z, 0.5);
}

public static double[] calculateMagnitudesFor(double[][] sensorData) {
	double[] output = new double[sensorData.length];
	for (int i = 0; i < output.length; i++) {
		output[i] = calculateMagnitude(sensorData[i][0], sensorData[i][1], sensorData[i][2]);
	}
	return output;
}

public static double calculateStandardDeviation(double[] arr, double mean) {
	double sum = 0;

	for (Double dataValue : arr)
		sum += Math.pow(dataValue - mean, 2);
	sum /= (arr.length - 1);

	return Math.pow(sum, 0.5);
}

public static double calculateMean(double[] arr) {
	double sum = 0;
```

```
		for (int i = 0; i < arr.length; i++) {
			sum += arr[i];
		}
		return sum / arr.length;
}

/***
 * Locates the peaks within the data
 *
 * @param times the times of the data
 * @param magnitudes the magnitudes of the data
 * @return a double array with values of 1 where there are peaks, and 0 otherwise
 */
public static int[] findPeaks(double[] magnitudes, double[] times) {
		int[] peaks = new int[magnitudes.length];

		for (int i = 1; i < magnitudes.length - 1; i++)
			if (magnitudes[i] > magnitudes[i - 1] && magnitudes[i] > magnitudes[i + 1]) {
				peaks[i] = 1;
			}
		clearExtraPeaks(peaks, magnitudes, DEADZONE_THRESHOLD, times);

		return peaks;
}

/***
 * finds all the peaks
 * @param magnitudes the array with the magnitude values from acceleration data
 * @return all the peak locations
 */
public static int[] findRawPeaks(double[] magnitudes) {
		int[] peaks = new int[magnitudes.length];

		for (int i = 1; i < magnitudes.length - 1; i++)
			if (magnitudes[i] > magnitudes[i - 1] && magnitudes[i] > magnitudes[i + 1]) {
				peaks[i] = 1;
			}
		return peaks;
}

/***
 * Clears extra peaks
 *
 * @param peaks the peak locations
 * @param magnitudes the magnitudes of the peaks
 * @param deadzone the absolute value range of values to check
 */
public static void clearExtraPeaks(int[] peaks, double[] magnitudes, int deadzone, double[] times) {
		int range = calculateTimeRange(times);
```

```
		for (int i = 0; i < peaks.length; i++) {
			if (peaks[i] == 1)
				checkDeadzoneForTallestPeak(peaks, i, range, magnitudes, times);
		}
	}

	/***
	 * Removes extra peaks that are in very close vicinity
	 *
	 * @param peaks an array containing the peak locations
	 * @param index the index to search values around
	 * @param deadzone the number of values in front of the index to search (also searches deadzone number of values
behind it)
	 * @param magnitudes the magnitudes for the peaks
	 */
	public static void checkDeadzoneForTallestPeak(int[] peaks, int index, int deadzone, double[] magnitudes, double[]
times) {
		int startIndex = index - deadzone, endIndex = index + deadzone;
		double currentMag = magnitudes[index];

		if (startIndex < 0) startIndex = 0;
		if (endIndex >= peaks.length) endIndex = peaks.length - 1;

		for (int i = startIndex; i < endIndex; i++) {
			if (i != index && peaks[i] == 1) {
				if (magnitudes[i] > currentMag)
					peaks[index] = 0;
				else
					peaks[i] = 0;
				break;
			}
		}
	}

	/***
	 * returns the range of times
	 * @param times the times from the data
	 * @return the integer value of the times to contain the range
	 */
	private static int calculateTimeRange(double[] times) {
		int range = 0;
		while (times[range] < TIME_THRESHOLD)
			range++;
		return range;
	}

	/***
	 * returns a small array of the current window of magnitudes to observe
	 * @param magnitudes the array of the magnitudes of acceleration data
	 * @param range the radius of values to look around the current value
```

```
	 * @param currentValue the current value
	 * @return a small array of the current window of magnitudes to observe
	 */
	public static double[] getMagnitudeCluster(double[] magnitudes, int range, int currentValue) {
		int startIndex = range - currentValue;
		int endIndex = range + currentValue;
		int currentIndex = 0;

		if (startIndex < 0) {
			endIndex -= startIndex;
			startIndex = 0;
		}
		if (endIndex >= magnitudes.length) endIndex = magnitudes.length - 1;

		double[] output = new double[endIndex - startIndex];

		for (int i = startIndex; i < endIndex; i++) {
			output[currentIndex++] = magnitudes[i];
		}
		return output;
	}

	/***
	 * Calculates a threshold value
	 * @param magnitudes the array with the magnitudes of acceleration
	 * @param mean the average of the data
	 * @return the threshold value
	 */
	public static double calculateThreshold(double[] magnitudes, double mean) {
		return (calculateStandardDeviation(magnitudes, mean) + mean);
	}

	/***
	 * Finds the next peak
	 * @param currentIndex the current index of the peak
	 * @param peaks the array with the locations of all the peaks
	 * @return the index of the next peak
	 */
	public static int getNextPeak(int currentIndex, int[]peaks) {
		for (int j = currentIndex; j < peaks.length; j++) {
			if(peaks[j] == 1){
				return j;
			}
		}
		return currentIndex;
	}
}
```

References:

- Rettner, Rachael. "The Truth About '10,000 Steps' A Day." *Live Science*. N.p., 7 Mar. 2014. Web.