# Prediction of Accelerometer Activity through Statistical Modeling and Machine Learning

By

Ryan Moore

Master's Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
in
Biostatistics

August 7, 2020

Nashville, Tennessee

Approved:

Professor Leena Choi , Ph.D.

Professor Kristin Archer Swygert, D.P.T., Ph.D.

## ACKNOWLEDGMENTS

I would like to express my very great appreciation to Dr. Leena Choi, my research supervisor, for her valuable feedback and suggestions. Her willingness to generously give her time is very much appreciated. Additionally, I would like to offer my thanks to Dr. Kristin Archer Swygert for providing the accelerometer dataset and joining as my second committee member. Finally, I am grateful to Sophie Cape for providing me with both support and proofreading.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Introduction

A tri-axial accelerometer is a device that measures acceleration, or the rate of change in velocity, in three orthogonal directions. Accelerometers are widely used in industry, engineering, and consumer electronics; however, they have also been increasingly utilized in biology and healthcare research with applications in ecology (Shepard, Wilson, Quintana, Laich, Liebsch, Albareda, Halsey, Gleiss, Morgan, Myers et al. 2008), fall detection in the elderly (Bagalà, Becker, Cappello, Chiari, Aminian, Hausdorff, Zijlstra & Klenk 2012), physical activity energy expidenture (Crouter, Churilla & Bassett 2006), and activity recognition (Kwapisz, Weiss & Moore 2011).

The focus of this thesis is the analysis of accelerometry data, which are collected from accelerometers affixed to human participant's bodies for clinical research on human activity. This type of data presents unique challenges in its analysis due to the massive size, participant non-adherence to protocol, and the data being collected outside of controlled laboratory settings. Prior research has created algorithms for identifying when participants are not wearing the accelerometer (Choi, Ward, Schnelle & Buchowski 2012); however, another difficult issue in the analysis of accelerometry data is that accelerometers are often activated prior to shipment to participants and are not deactivated until they are returned to the laboratory. Due to this design, large portions of accelerometry datasets are often recorded while the accelerometers are in transit to the participant or laboratory. The purpose of this thesis is to explore and develop algorithms that can accurately classify a given day in an accelerometer dataset as a human wear day or a delivery day.

To the best of our knowledge, research has not been conducted on training algorithms to predict whether accelerometer activity is due to human wear or motion from delivery; however, the field of human activity recognition is highly applicable to this problem. The primary goals in human activity recognition research are to find the temporal partitions in a dataset in which the performed activities change and to classify temporal intervals as the correct activity (Kim, Helal & Cook 2009). The classification of days as delivery or human wear based on differences in acceleration is a similar classification problem to human activity recognition. Algorithms utilized to classify human activity can be highly applicable to the classification of human wear versus delivery activity.

The task of human activity recognition is performed with data from a wide variety of sensors such as video cameras, GPS, heart monitors, and thermometers; but wearable accelerometers are one of the most commonly utilized devices due to recent technical advances in microelectronics and the rich data they provide (Lara & Labrador 2012). In the context of human activity recognition, accelerometer data is traditionally analyzed by first extracting global features from the temporal intervals of the dataset. The average acceleration, standard deviation of acceleration, time between peaks, quantiles, and many other features are commonly used as inputs in a wide range of models (Kwapisz et al. 2011). The goal of feature extraction is to reduce massive datasets such that regression models or machine learning methods can be used with a reasonable number of variables; however, the advancement of neural networks in recent years has created the ability to analyze the raw data using algorithms that identify complex local features instead (Ignatov 2018).

In this thesis, we develop several machine and statistical learning algorithms to classify days in an accelerometry dataset as delivery or human wear in a supervised learning framework. In Chapter 1, the algorithms used in the analysis are introduced and described. In Chapter 2 of this thesis, we demonstrate the processing and analysis of a benchmark dataset in the field of human activity recogniton: the Wireless Sensor Data Mining (WISDM) dataset. For Chapter 3, we present several internally validated models to classify a given day as human wear or delivery activity. In Chapter 4, we present the conclusions drawn from the analysis of the datasets and potential future applications.

# Chapter 1

# Background

## 1.1 Machine and Statistical Learning

Machine learning is a data analysis method that allows computers to learn from data without explicit model structure, whereas in traditional statistical learning model structures are pre-specified. The driving force of machine learning is that the model structure adapts in response to data, allowing the process of "learning." Although machine learning methods can be extremely powerful, they are often difficult to interpret and require very large sample sizes for model training (Nilsson 2005).

The field of machine learning differs from statistics in that its primary focus is prediction, while statistical models perform both prediction and inference. Machine learning methods are usually appropriate when the sample sizes are huge with many events, the primary interest of the study is prediction rather than inference, the signal to noise ratio is high, and it is acceptable that the model is relatively uninterpretable. In these settings, machine learning can outperform traditional statistical methods and may be a suitable choice (Harrell Jr 2015). Some examples of settings where machine learning has had powerful performance is in mastering the game of go with the AlphaGo algorithm (Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton et al. 2017) and in image recognition (Krizhevsky, Sutskever & Hinton n.d.). Machine learning flourishes in these settings due to the relative ease of building a large training set and the inherently mechanistic outcomes.

Another difference between statistical methods and machine learning is flexibility. Although different methods within the fields of statistics and machine learning have different degrees of flexibility, machine learning as a whole tends to be much more flexible and therefore tends to be more prone to overfitting (Harrell Jr 2015). Subsequently, great care must be taken in both building and interpreting the predictions from machine learning models. On the other hand, techniques such as ensemble methods, cross-validation, and external validation can be utilized to help increase generalizability. When considering what model to use, one must also consider each model's trade off between interpretability and flexibility. If the primary goal of learning is inference, then interpretability of the model is much more important than flexibility. Conversely, when the primary goal is prediction, accuracy and flexibility are generally more important than interpretability (Hastie, Tibshirani & Friedman 2009).

Learning tasks can usually be divided into supervised or unsupervised learning. Unsupervised learning involves inputs that do not have associated outputs. Performing a regression is not possible and hence unsupervised learning often seeks to identify relationships between the inputs. Unsupervised learning tasks often involve clustering inputs into distinct groups using their relationships. Some examples of unsupervised learning models are clustering analyses such as nearest neighbors and principal component analysis (Hastie et al. 2009). Contrasting unsupervised learning, in supervised learning each input has an associated output. The goal of supervised learning may be to find a relationship between the response and predictor or to be able to predict a response in the future. Some examples of supervised learning models are random forest, neural networks, and logistic regression. If the output is continuous, regression can be utilized in supervised learning to predict the outcome. Alternatively, when the outcome has separate and distinct labels, classification can be utilized. Supervised learning generally utilizes a loss function that penalizes prediction errors. By minimizing a chosen loss function, a locally optimal solution can be found (Hastie et al. 2009).

In the context of supervised learning, the dataset of interest is often randomly split into a training set and a test set through a process known as data splitting. The training set is used to develop the model, while the test set is then used to validate the model's performance in order to reduce overfitting. Data splitting is the simplest and one of the weakest forms of model validation because it greatly reduces sample size and the results of the validation can vary depending how samples are randomly split into the training and test set. A preferable method of model validation is bootstrapping or cross-validation, in which the data is resampled or split and modeled repeatedly.

Once the model validation technique is selected, the model's predictive abilities can be measured by its calibration and discrimination. A calibration measure quantifies the model's ability to make unbiased estimates, while a measure of discrimination indicates the model's ability to correctly predict outcomes (Harrell Jr 2015).

## 1.2 Machine Learning Methods

### 1.2.1 Tree Based Methods

Decision trees are a supervised learning method that can be utilized in the context of regression and classification. As their name suggests, decision trees have the hierarchical form of a tree with layers of nodes and branches extending from an origin known as the root node. At each node, a decision rule is applied to the data in order to recursively partition the dataset into smaller and purer subsets. For continuous predictor variables, the decision rules at the respective decision node is an inequality or interval. For example, the data would pass through a decision node to the node's first branch if its value satisfies the interval or inequality. Otherwise, the data would pass through the node to the second branch. For categorical predictor variables, data from different categories pass through a given node to one of several different branches. After passing through multiple nodes and branches, the data is recursively subset until it reaches a final node, known as a leaf node, that has no branches extending from it and no decision rule. A terminal leaf node exists at the end of every possible path on the decision tree and provides the final classification (Myles, Feudale, Liu, Woody & Brown 2004).

When training a decision tree, the goal is to create an accurate and efficient tree that can correctly classify the data with a minimal number of decisions or nodes. The construction of the decision tree starts at the root node and includes the entire training set. In the simplest case of binary and univariate branching, a predictor variable and decision rule are chosen at the root node to split the training set into two subsets according to a splitting algorithm that optimizes a specified scoring criteria. The subsets of the data in the two different branches are then recursively subset at the following nodes until a stopping rule is reached (Myles et al. 2004).

Different decision tree algorithms utilize different variable selection methods, decision rules, and stopping rules. The variable utilized at a given node and its corresponding decision rule are chosen with the aim of maximizing the “purity” of the resulting nodes from the split. Purity is often measured with the mean square error, variance, entropy, or most commonly the gini index. Some commonly used stopping rules are a minimum number of observations in a leaf node and a maximum number of steps from the root node. Occasionally, a stopping rule is not enough to create an efficient decision tree and the tree is pruned to remove branches and nodes that are less informative (Song & Ying 2015).

Although decision trees are appealing in that they are visually interpretable and can effectively model complex interactions without an additivity assumption, they tend to perform poorly relative to other algorithms. Decision trees are often ungeneralizeable to the test set, tend to be unstable, often overfit, and are biased against continuous variables. Cross-validation can ameliorate some of the ungeneralizability of decision tree models; however, these models still tend to not be very accurate (Harrell Jr 2015).

Decision trees often perform poorly due to overfitting and instability; however, when multiple decision trees are utilized together in an ensemble method, their performance can be greatly improved. Ensemble methods such as bagging, boosting, and random forest grow multiple decision trees and then arrive at a single consensus on classification or prediction that is ideally more robust and accurate than a single decision tree’s prediction. When used for classification, ensemble methods collect the classifications from each individual tree, then select the mode as the consensus. In a regression context, they simply take the average output from all the individual trees. By averaging or taking the majority output of many high variance trees these ensemble methods are able to reduce instability, while modeling complex interactions (Hastie et al. 2009).

Bootstrap aggregation, or bagging, is one ensemble method that is commonly utilized on decision trees. To perform bagging, a sample is recursively taken with replacement from the original dataset and an individual decision tree is grown from each bootstrapped sample. The average output for regression or mode output for classification is taken as the final output for the bagging method. The proportion of misclassifications by the trees is then the bagging error (Breiman 1996).

Currently, the random forest model is one of the most popular bagging methods due to the ease of implementation, speed, and generally good performance. The random forest is performed similarly to the traditional bagging method described above, but with a key difference - at each split in the bootstrapped decision trees, the random forest method randomly selects a subset of the available variables used in the decision rule. In traditional bagging, the bootstrapped samples create an ensemble of highly correlated trees because any feature that is of high importance to the prediction will be utilized in the majority of the trees. By randomly selecting a subset of variables at each splitting point, the trees in a random forest are relatively decorrelated, making the trees less flexible and prone to overfitting (Breiman 2001).

### 1.2.2 Neural Networks

A neural network is a machine learning method composed of layers of multiple interconnected nodes, known as artificial neurons. Heavily inspired by the biological brain, these systems of neurons are designed to work collectively to learn how to perform various tasks. Although each artificial neuron is relatively simple, large networks of these neurons are able to solve a variety of complicated tasks by utilizing different architectures and connection patterns between the neurons. Due to their complexity, artificial neural networks are among the most flexible but least interpretable machine learning methods. The flexibility of these algorithms can make them ideal for solving classification and prediction problems when a large training set of data is available (Nielsen 2015).

#### 1.2.2.1 Multi-Layer Perceptron

A multitude of neural network architectures exist, but the earliest and simplest form is the traditional feed-forward neural network known as a multilayer perceptron. In a feed-forward neural network, information flows unidirectionally through each layer in a process known as forward propagation. Data are first received by the input layer, which has one neuron for each variable input from the data, and then distributed through the hidden layers to the output layer. The output layer is the final layer that returns the output to the user. For a binary classification problem, a single output neuron gives the probability of the inputs being from a target class. In a classification problem with 5 categories, 5 output neurons would be used with each estimating the probability of the inputs belonging to a respective class. The number of neurons in each hidden layer and the number of hidden layers are known as hyperparameters, which are chosen by the user depending on the task. Using more neurons and hidden layers can solve more complex tasks but also increase the risk of overfitting (Nielsen 2015).

In a traditional fully-connected network, a given neuron has a connection to every neuron in the subsequent layer to which it sends its output. This output is then received by the subsequent layer's neurons as their inputs (Figure 1.1). Every neuronal connection has a unique weight that is used in the calculation of a neuron's output. In general, a lower weighted neuron has less impact on the output than a neuron with a higher weight. Additionally, every neuron can have a bias, which functions similarly to an activation threshold. Both the biases and parameters that are learned from the data are not defined by the user (Nielsen 2015).

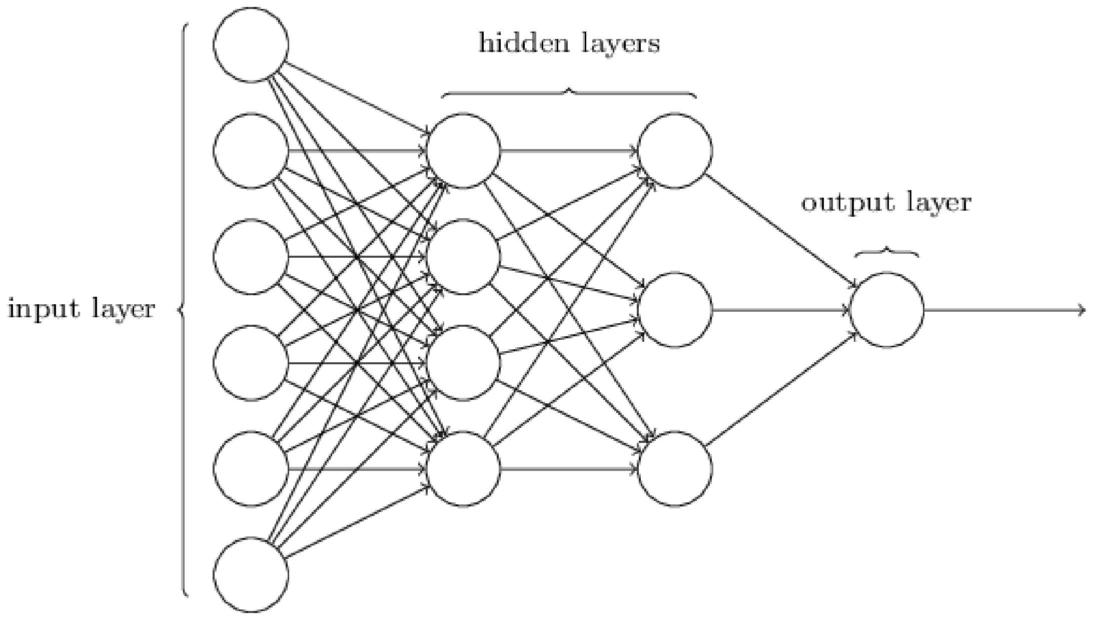


Figure 1.1: A simple example of the architecture of a fully-connected multilayer perceptron. Black lines indicate activations and circles indicate neurons. In practice, hidden layers usually include many more neurons than can be easily shown in a figure (Nielsen 2015).

The output or activation of the $j^{th}$ neuron of the $l^{th}$ layer is denoted as $a_j^{(l)}$. Similarly, the bias of this neuron is denoted as $b_j^{(l)}$. The weight of the connection between the $k^{th}$ neuron of the $(l-1)^{th}$ layer to the $j^{th}$ neuron of the $l^{th}$ layer is denoted as $w_{j,k}^{(l)}$. To calculate a neuron's output, the outputs of every previous layer's neurons are multiplied by each connection's weights and then summed. This value, which is denoted as $z$, is then used in an activation function, $f(z)$, to calculate the final output. The activation of the $j^{th}$ neuron of layer $l$ is calculated as:

$$a_j^{(l)} = f\left(\sum_k w_{j,k}^{(l)} a_k^{(l)-1} + b_j^{(l)}\right). \tag{1.1}$$

This function can also succinctly be expressed as a vector of activations in a given layer $l$ by:

$$a^{(l)} = f(w^{(l)} a^{(l-1)} + b^{(l)}). \tag{1.2}$$

Many different activation functions can be used with each having different desirable properties. For example, the rectified linear unit function (ReLU) makes all negative output values 0 while maintaining the identity of all positive outputs. The sigmoidal activation function scales all outputs between 0 and 1 with the following function:

$$f(z) = \frac{1}{1+e^{-z}}. \tag{1.3}$$

The activation function of the output neurons must be carefully chosen to reflect the nature of the task. For example, in a binary classification problem, a sigmoidal activation function for a single output neuron can be used to estimate the probability of the inputs belonging to a target class. For a multi-class problem, a softmax activation function is commonly used. The softmax activation function transforms the output of the last hidden layer into probabilities that are normalized such that they sum to 1. The class associated with the highest output neuron can then be chosen as the most probable class given an input (Bridle 1990). The softmax activation function for calculating a class probability, $P_j$ from output neuron $j$ and its input, $z_j$, across $k$ classes is:

$$P_j = \frac{e^{z_j}}{\sum_k e^{z_k}}. \tag{1.4}$$

Given an optimized matrix of weights and biases, a neural network can hypothetically solve complicated tasks such as audio or image recognition with a high degree of accuracy. The learning aspect of the neural network comes from its ability to choose these weights and biases such that the network is able to accomplish its task. In the context of supervised learning, training inputs, $x$, with already known correct outputs, $y$, are used in a cost function in order to identify which sets of weights and biases give optimal output. A lower value of the cost indicates that the neural network's outputs are more accurate. In order for a neural network to learn, an appropriate cost function must be selected and then minimized with respect to the weight and bias parameters (Nielsen 2015).

The quadratic cost function or mean square error is one of the most commonly used cost functions and has the form:

$$c(x,y) = \frac{1}{2n} \sum_{x} ||y(x) - a^L(x)||^2. \tag{1.5}$$

In the quadratic cost function, $n$ is the total number of training inputs, $L$ is the total number of layers in the neural network, and $a^L$ is a vector of the activations of the output layer given input $x$. Dividing the function by 2 may appear strange and is somewhat arbitrary, but ultimately cleans up the equation during later stages in the learning process when the derivative of the cost function is used (Nielsen 2015).

Typically in a classification setting, a cross-entropy loss function is used instead of mean square error due to increased training speed and its ability to easily adapt to multi-class settings. Cross entropy loss calculates the mean difference between the predicted and true probabilities across the entire training dataset, thus softmax and sigmoid are natural activation functions to pair with cross-entropy in classification settings (Senior, Vanhoucke, Nguyen, Sainath et al. 2012). In a binary classification setting, the formula for cross entropy loss where the correct target class is denoted as $y$ and the predicted probability of a given set of inputs being in that class is $\hat{y}$ is displayed below:

$$c\big(x,y\big) = -\frac{1}{N} \sum_{i}^{N} \big(y_i log\hat{y}_i + (1-y_i) log(1-\hat{y}_i)\big) \tag{1.6}$$

If any weight in the network is changed, then the output will also change. Neural networks are able to learn through a series of many small changes in weights and the effect or impact of these changes on the cost function. Different learning techniques are available to determine which weights to use. Hypothetically, one could even randomly choose weights, saving the matrix of weights that results in the lowest value of a cost function, but in practice a more sophisticated method called gradient descent is used to locally minimize the cost function (Nielsen 2015).

Gradient descent works by iteratively minimizing the cost function through a series of 'steps.' At each step, the partial derivatives of the cost function with respect to each weight and bias are approximated using an algorithm called backpropagation. These values are stored in a vector known as the gradient vector, $\nabla C$. The magnitude of each component of $\nabla C$ indicates the relative sensitivity of the cost function to each weight and bias in the network. Since the gradient of the cost function gives the direction of steepest increase in the cost function, the gradient vector can be multiplied by negative 1 to find what direction each weight and bias should change in order to most efficiently start locally minimizing the cost function. To calculate the first step in the minimization of the cost function, the gradient vector is multiplied by the learning rate, $\mathscr{N}$, in order to scale how much the weights and biases will move in a step. If $\mathscr{N}$ is too small, the neural net will take too long to learn. If $\mathscr{N}$ is too big, the neural net may continuously jump over the minimum at each iteration and will never be able to find a minimum in the cost function. $-\mathscr{N}\nabla C$ is calculated and the weights are stepped repeatedly until the cost function converges to a local minimum (Nielsen 2015).

Gradient descent and backpropagation are extremely resource intensive processes. In order to reduce training time, training sets are often evenly divided into smaller mini-batches. The algorithms are carried out over the first mini-batch, and the gradient vector calculated from this mini-batch is used as the next "step" in the gradient descent. Each step of the gradient descent is calculated in this manner until convergence to a minimum. This process, known as stochastic gradient descent, is less precise than training on the entire training set for each iteration, but the gradient descent converges to a local minimum much more efficiently (Nielsen 2015).

Stochastic gradient descent is the classical method of optimizing a neural network, but in recent years the Adam (adaptive moment estimation) algorithm has been gaining popularity due to several unique benefits. Unlike stochastic gradient descent, the Adam optimizer has an adaptive learning rate that decays if the performance of the model stabilizes or increases if the model performance does not improve across batches. One major advantage of Adam's adaptive learning rate is that a learning rate does not need to be specified by the user as a hyperparameter. In addition to having an adaptive learning rate, the Adam optimizer adds momentum to the learning process. Momentum utilizes a weighted decaying average from past weight updates to accumulate inertia and continue moving down the gradient in the direction of past updates. This process generally helps speed up learning, making it a popular and useful feature of an optimization algorithm (Kingma & Ba 2014).

#### 1.2.2.2 Convolutional Neural Network

The multilayer perceptron can function very well for relatively simple tasks but tends to struggle as the complexity and size of the task increases. As more neurons become necessary to solve a task, the computation slows down and the risk of overfitting increases. More sophisticated neural network architectures, such as the convolutional neural network, have been created in order to address more complicated data (O'Shea & Nash 2015).

Convolutional neural networks are a form of feed-forward neural networks that utilize specialized hidden layers in order to perform advanced tasks, often with much better results than the traditional feed-forward network previously described. The specialized layers in convolutional neural networks are able to scan large inputs such as colored images and collapse them down to a more manageable size for the rest of the network to analyze. Convolutional neural networks were initially developed in order to address the complexities of image analysis but have been successfully adapted to many other tasks such as natural language processing (Hu, Lu, Li & Chen 2014) and text data analysis (Kim 2014).

The first specialized hidden layer of a basic convolutional neural network is known as the convolutional layer. Each neuron of this layer scans the input data for features. For example, in the context of letter recognition, this feature may be a small concave curve associated with the lower region of the letters 'U' or'J.' In a process known as filtering, a kernel or filter is overlaid and then iteratively scanned across the input data, creating an activation map of where the kernel was found to best match the data. The convolutional layer then applies the filtering process on the data with many different filters and outputs a stack of filtered images (Cireşan, Meier, Masci, Gambardella & Schmidhuber 2011). In the case of 2-D convolution, the kernel travels across the image by row and column. This process is commonly used in image recognition algorithms (Krizhevsky et al. n.d.). For a 1-D convolutional layer, the kernels only travel in one direction. 1-D convolution is commonly used in time series such as tri-axial accelerometers (Ignatov 2018). Simplified examples of 1-D and 2-D convolution are presented in Figures 1.2 and 1.3 respectively.

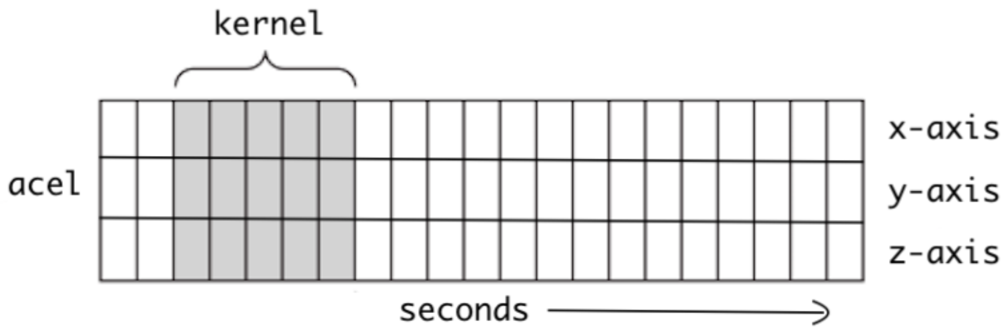


Figure 1.2: A simplified example of a kernel convolving one dimensionally across 3 axes of accelerometer data (Verma 2019).

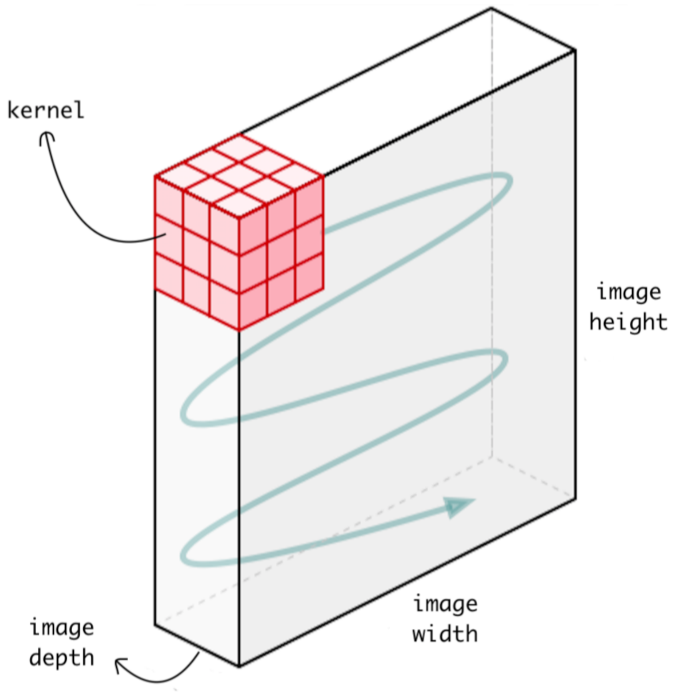


Figure 1.3: A simplified example of a kernel convolving two dimensionally across a three color deep image (Verma 2019).

Even with a relatively small input, convolutional layers can easily become extremely complex. Certain hyperparameters can be manipulated to reduce the computational rigor of a convolutional layer, but this can come at the cost of the model's performance. The depth of a convolutional layer is the number of neurons within the convolutional layer that are connected to a neuron in the input layer. Decreasing depth decreases the number of neurons, which speeds up the algorithm. The stride of the convolutional layer is the number of 'pixel steps' that the filter matrix takes between each scan of the input data. If the stride is increased, the filter matrix overlaps over less data and reduces the spatial dimension of the activation map. Additionally, the input can be zero-padded at the borders. This process allows the filter matrix to scan the corner and edge pixels with the same frequency as the pixels in the middle of a two dimensional image by adding false pixels around the borders of the input (O'Shea & Nash 2015).

The next specialized hidden layer of a basic convolutional neural network is the pooling layer, which functions by reducing the dimensionality of the convolutional layer's output. In the process of pooling, a small kernel is iteratively scanned across an activation map. There are multiple pooling functions that can be used, such as max-pooling or averaging as shown in Figure 1.4, but the end result is always a reduced activation map. The size and stride of the pooling kernel can be manually adjusted in order to change the speed of the pooling process. If the stride and size of the kernel are increased, pooling will occur more quickly and dimensionality will be further reduced, but more data will be lost, which can potentially affect the analysis (Cireşan et al. 2011).

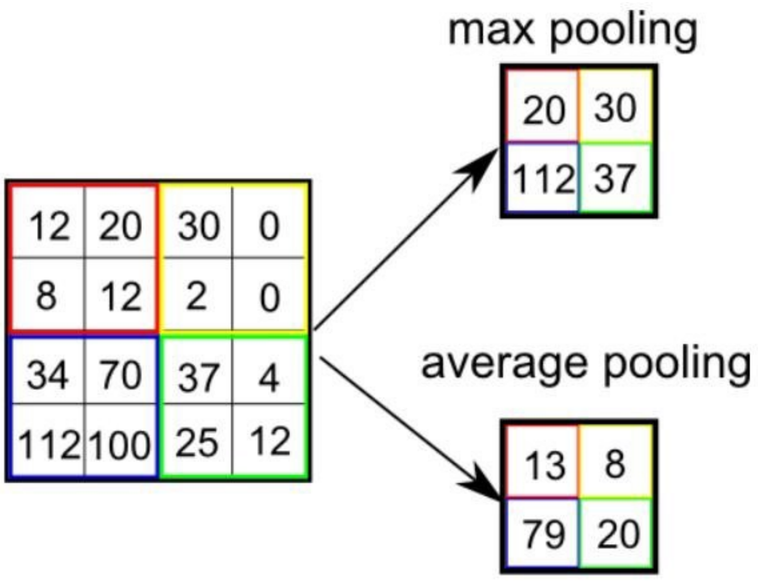


Figure 1.4: Examples of 2-D max and mean pooling functions. In this example, the pooling kernel has a size of 2 by 2 and a stride of 2 (Saha 2018).

After convolution and pooling, the pooled activation map is flattened and then transmitted to the fully connected layer, which has similar architecture and function to the multilayer perceptron neural network previously described. Ideally, the fully connected layer learns the correct output given a flattened and pooled activation map through an optimization algorithm such as gradient descent and propagation. The ideal number of neurons of the fully connected layer can vary with the complexity of its input (Cireşan et al. 2011).

In addition to the previously mentioned hyperparameters of each layer, the number and order of layers in the network itself can be varied. In some cases, it has been found more effective to send information through a series of repeated convolutional and pooling layers prior to sending through the fully connected layer. This series of layers can collapse large input data down into more manageable sizes for the fully connected layer. Having more than one fully connected layer can also be useful depending on the complexity of the input as a deeper layer of convolution can combine simple features into more complex shapes (Krizhevsky et al. n.d.). An example of the complete architecture of a 2-D convolutional neural network is presented in Figure 1.5.

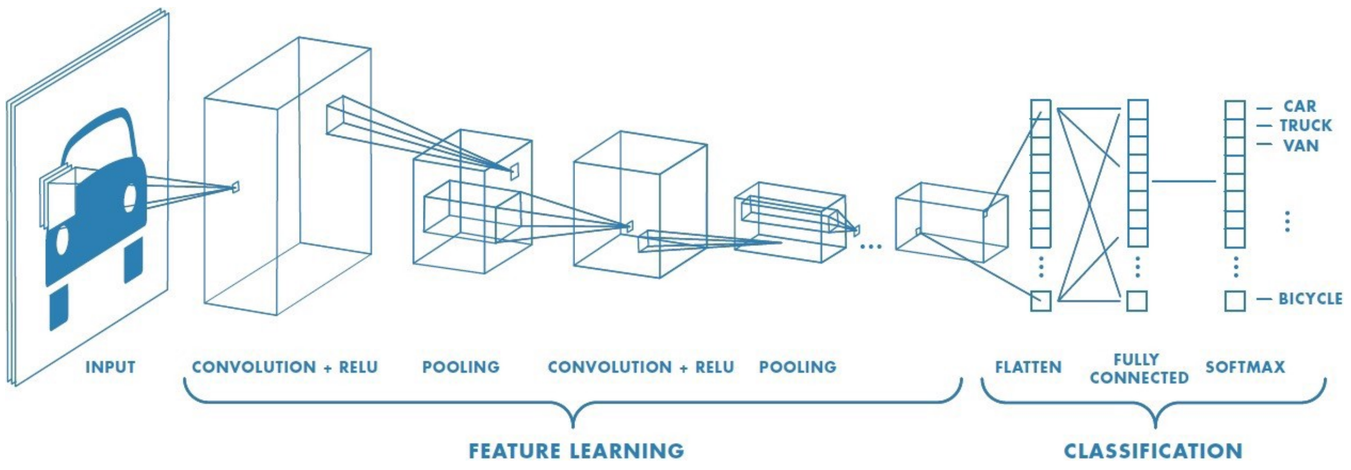


Figure 1.5: An example of a 2-D CNN designed to classify images of vehicles. This architecture uses two sets of convolutional and pooling layers followed by a flattening and fully connected layer (Saha 2018).

Convolutional neural networks have to learn not only the matrix of biases and weights in the fully connected layer but also the filters to use in the convolution layer. The process of learning in the fully connected layers and output layer is identical to the gradient descent and backpropagation processes described with traditional feed-forward networks. Learning which optimal filters to apply also uses backpropagation and an optimization algorithm, but the process performs convolutions across the input image (Cireşan et al. 2011).

#### 1.2.2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of feed-forward neural networks that have been adapted to work well with sequenced data such as time series, speech recognition, natural language processing, or predicting protein structure from genetic sequences (Baldi & Pollastri 2003). Unlike the previously discussed CNN and MLP architectures, RNNs are able to utilize information from previous inputs when calculating the output of the current input. A simplified example of this architecture is shown in Figure 1.6. This can be extremely useful in sequential data, where information from early in the sequence of data can be informative in predicting later sections of the sequence. For example, in natural language processing, RNNs are extremely useful as the first several words of a sentence contextualize the rest of the sentence (Sak, Senior & Beaufays 2014).

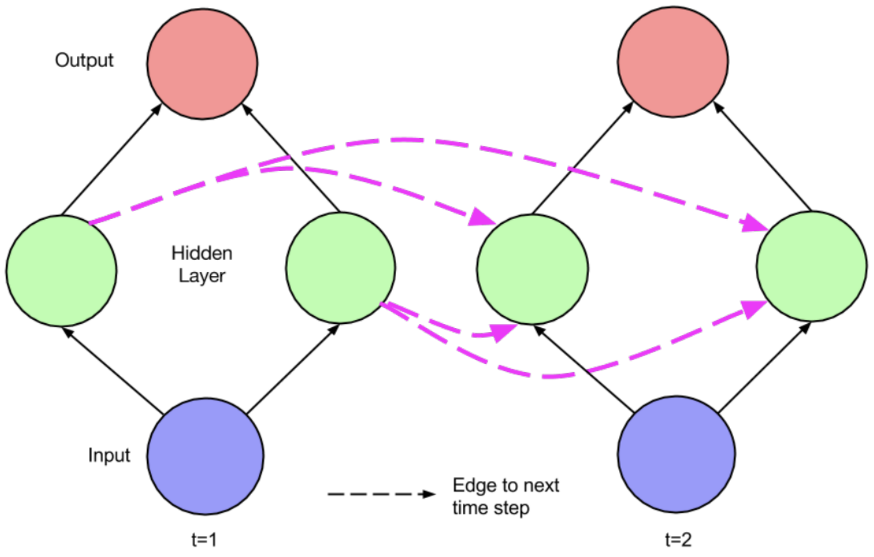


Figure 1.6: Example of a simple recurrent neural network across time steps t and t+1. Solid lines indicate activation passed within a time step and dashed lines indicate activation passed forward through time (Lipton et al. 2015).

Another major difference between RNNs and most neural networks is that each input can be a different length. This is exceptionally useful in the context of voice recognition or natural language processing, where the length of the input data is dependent upon the time span of the audio clip or number of letters in a word. A multi-layer perceptron or convolutional neural network could hypothetically be zero-padded to the maximum length of all the inputs, but this generally has poor performance relative to an RNN (Lipton et al. 2015).

RNNs are able to achieve a memory state by using the activations from the previous inputs to calculate the activations of the current input. In the forward pass, RNNs sequentially analyze each input while passing information from previous inputs to the following input through the activation function. The initializing activation, $a^{<0>}$, is generally just a vector of zeros. This initial activation is used along with the first input from the data, $x^{<1>}$, in order to calculate the first activation, $a^{<1>}$. Both the initializing activation and the first input, $x^{<1>}$, are multiplied by their respective matrices of weights, $\mathbf{W_{aa}}$ and $\mathbf{W_{ax}}$, and summed together along with a bias, $b_a$. This sum is then input into an activation function, $f(\cdot)$, such as a ReLU or tanh. When the input, $x^{<t>}$, is not the first input, the activation of the previous input, $a^{<t-1>}$, is used rather than the initialization activation (Lipton et al. 2015). The equation for the calculating the $t^{th}$ activation in the forward pass is then:

$$a^{<t>} = f\big(\mathbf{W_{aa}}a^{<t-1>} + \mathbf{W_{ax}}x^{<t>} + b_a\big). \tag{1.7}$$

Similarly to MLPs, this calculated activation is multiplied by a weight matrix, $\mathbf{W_{ya}}$, summed together with a bias, $b_y$, and input into another activation function, $g(\cdot)$. This activation function could be a sigmoid function for a binary output or softmax for categorical outputs (Lipton et al. 2015). The equation for calculating the $t^{th}$ prediction output from the $t^{th}$ input is then:

$$\hat{y}^{<t>} = g\big(\mathbf{W_{ya}}a^{<t>} + b_y\big). \tag{1.8}$$

The weight matrices and biases that are used to calculate the predictions and activation functions are calculated by backpropagating through time (BPTT). In backpropagation in traditional neural networks, a training input vector is first fed into the network and propagated forward to predict an output. Then an error is calculated in order to compare the prediction and truth. Partial derivatives of the chosen error equation are calculated with respect to the weights and biases (Werbos 1990).

Recurrent neural networks are unique in their ability to model dependencies across time or sequence, but in practice traditional RNNs tend to perform very poorly due to vanishing or exploding error gradients during backpropagation through time. Vanishing and exploding gradients occur as information from a prior input on a neuron in the hidden layer exponentially vanishes or explodes. These issues worsen as the distance in sequence increases due to weights being repeatedly multiplied by themselves. Several solutions ranging from different architectures to changes in optimization algorithms have been proposed in order to ameliorate this issue. The Long Term Short-Term Memory (LSTM) RNN is currently one of the most popular and successful of these adaptations to the RNN architecture. The LSTM was designed to avoid the vanishing gradient problem by utilizing a specialized memory cell instead of a traditional neuron in the hidden layer. Each memory cell is connected by a cell state, which flows sequentially from the first memory cell to the last memory cell. Unlike a normal activation, the cell state is carefully regulated by a series of gates that have their own activation function. Information is only added to the cell state through minor linear interactions and thus is able to avoid the vanishing and exploding gradient problem that occurs from repeatedly multiplying weights (Hochreiter & Schmidhuber 1997).

#### 1.2.2.4 Convolutional Recurrent Neural Networks

By combining convolutional and recurrent layers into the same neural network, one can take advantage of the local feature extraction and dimensionality reduction of a convolutional network while simultaneously utilizing the ability of temporal modeling from a recurrent neural network. Architectures of this type have successfully been used in analyzing videos but potentially have application in many different types of time series analyses (Donahue, Anne Hendricks, Guadarrama, Rohrbach, Venugopalan, Saenko & Darrell 2015).

The basic architecture of a hybrid convolutional recurrent neural network is essentially a convolutional neural network that feeds into a recurrent architecture rather than the standard fully connected layer after the flattening layer. By first processing the input through convolutional and pooling layers, only the reduced dimension features are fed into the recurrent layer. This can be exceptionally useful for large and complex data that have temporal correlation such as dynamic magnetic resonance imaging data (Qin, Schlemper, Caballero, Price, Hajnal & Rueckert 2018). A disadvantage of this method relative to plain recurrent neural networks is that the inputs all have to be the same size since the architecture begins with convolutional layers.

## 1.3 Statistical Learning Algorithms

### 1.3.1 Binary Logistic Regression

Binary logistic regression is a generalized form of linear regression that is used to model the log odds or probability of an input belonging to a target class of a binary dependent variable. Unlike the previously described machine learning methods, the inputs or independent variables used in a binary logistic regression have interpretable parameters that can used for both inference and prediction (Harrell Jr 2015).

The logistic regression model can be expressed as linear in $\mathbf{X}\beta$ by:

$$\text{logit}\big(E[Y|X]\big) = \mathbf{X}\beta, \tag{1.9}$$

where $\mathbf{X}\beta = \beta_0 + \beta_1 X_1 + ... + \beta_k X_k$.

To estimate the probability of Y belonging to the target class, the $\beta$ coefficients are first estimated using the maximum likelihood estimation and a numerical estimation technique such as Newton-Raphson algorithm (Hastie et al. 2009). The estimation of the probability of belonging to the target class based on inputs, $\mathbf{X_i}$, can then be expressed as:

$$\hat{P}_i = [1 + exp(-X_i\hat{\beta}]^{-1}, \tag{1.10}$$

where $\hat{P}_i$ is the estimate for Prob$(Y_i = 1|X_i)$.

Logistic regression is a powerful inferential tool, but it is also commonly used as a classification algorithm. By creating a decision threshold at $P(Y = 1|X) \geq 0.5$, logistic regression can be used to classify sets of inputs based on the coefficients.

### 1.3.2 Mixed Effects Logistic Regression

Logistic regression makes the assumption that observations are independent of each other. When data is obtained from repeated measurements of the same individual, this assumption is violated since the samples from within the same individual will be correlated with each other. One way to account for correlation in repeated measurement data is to induce a correlation structure using subject-specific random-effects in a mixed-effects regression model (Diggle, Heagerty, Liang, Zeger et al. 2002).

A mixed-effects regression model assumes that each subject has a regression model that is characterized by a combination of random and fixed effects. The fixed effects are analogous to the coefficients in a standard logistic regression and are common to every individual. The random effects are subject-specific parameters that are drawn from a distribution and used to induce a correlation structure for the data. The random effects explain natural differences between subjects for a subset of the coefficients (Diggle et al. 2002). Often the random effects are assumed for the intercept to explain differences in the inherent trait of the subjects or for the slopes to explain differences in the rate of change among the subjects (Schildcrout 2020).

In the case of logistic regression with random intercepts, the probability of the outcome of subject $i$ at time $j$, $Y_{ij}$, belonging to the target class can be given by:

$$\text{logit}\big(E[Y_{ij}|X_{ij},U_i]\big) = \beta_0 + U_i + \mathbf{X}_{ij}\beta U_i \sim N(0,\sigma^2), \tag{1.11}$$

where $\beta_0$ is the intercept common to every individual and $U_i$ is the random effect for subject $i$, assuming a gaussian distribution with mean 0. The rest of the notation is the same as in Section 1.3.1.

Inference for a mixed effect model can be made using a maximum likelihood approach in which the random effects are treated as nuisance variables and integrated over. For generalized mixed effects models such as the logistic mixed effect model, a numerical approximation technique is necessary to estimate the coefficients as no closed-form solution exists (Schildcrout 2020).

# Chapter 2

# WISDM Dataset and Analysis

## 2.1 Introduction

The Wireless Sensor Data Mining (WISDM) project was started by the Computer and Information Science Department at Fordham University with the goal of exploring various research issues related to sensor data from mobile devices. Thus far, the WISDM project has been focused on human activity recognition using data from tri-axial accelerometer sensors in a controlled laboratory setting. Due to the high quality and open source nature of the WISDM data it has become a benchmark dataset in the field of human activity recognition (Kwapisz et al. 2011). The goal of both this analysis and the original analysis of the WISDM dataset is to predict which activity is being performed based on the tri-axial acceleration data. For the purposes of this thesis, the WISDM data was utilized as a benchmark to refine model development on data that already has been analyzed for comparison. The techniques refined on the WISDM dataset are then expanded upon in the analysis of the accelerometry dataset, which is the primary focus of this thesis.

The accelerometer data for the WISDM dataset was collected for a supervised learning context in a laboratory setting. Thirty-six total participants were asked to perform six different activities while a smartphone in their front pants pocket collected accelerometer data. A custom designed phone application was used under supervision of a WISDM team member to start, stop, and label the data in real time during the data collection. Measurements were taken every 50 milliseconds, resulting in 20 measurements per second. After aggregating all participant's data, the final form of the WISDM dataset was a large sequence of time-series acceleration measurements on the x, y, and z axis, which are both activity and participant labeled (Kwapisz et al. 2011). The entire dataset contains 1,098,203 measurements, or slightly over 15 hours, of labeled tri-axial accelerometer data. Examples of the first 10 seconds of each activity on only the x-axis are presented in Figure 2.1.

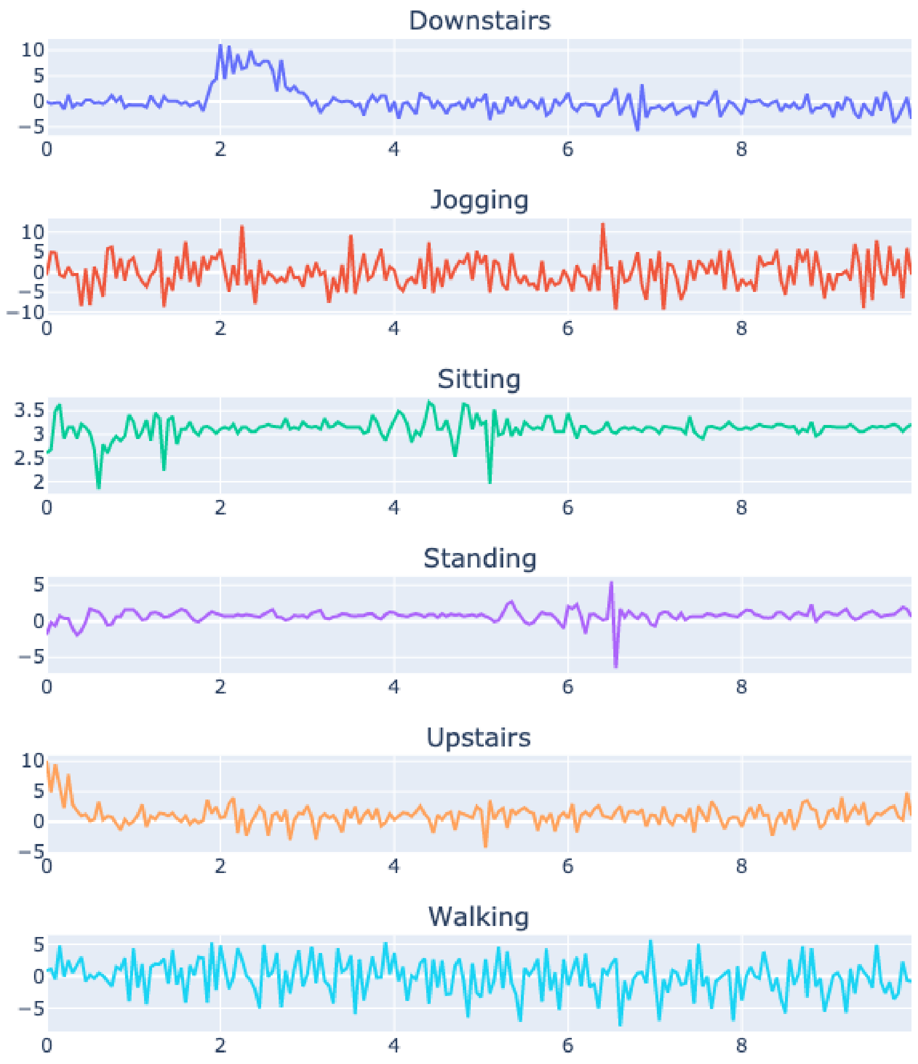


Figure 2.1: 10 second examples of accelerometer data by activity from the x axis. The x axis captures left to right motion or vice versa from the participant's leg.

## 2.2 Methods

The data processing and modeling of the human activity accelerometer laboratory WISDM dataset was performed using Python 3 with assistance from a human activity recognition tutorial (Ackermann 2018) for the import and segmentation process. Code for the processing and modeling of the data can be found in Appendix A.

### 2.2.1 Data Processing

The data was first randomly split into a test set and training set such that participants with data in the training set were not included in the test set. Twenty-five of the participants were randomly assigned to the training set, while the remaining 11 unique IDs were assigned to the testing set. The datasets were fairly large with approximately 10 and 5 hours of data in the training and test sets, respectively. After splitting the data, the training and test set had their x, y, and z axes separately min-max scaled. This step is important for efficient convergence in the neural network algorithms (Orr & Müller 2003).

Each participant's trial in the data is formatted as one contiguous dataset regardless of the activity being performed. The different activities were performed for varying amounts of time for each participant. In order to model the WISDM data, the dataset was first segmented into sections of identical 80 measurement lengths (4 seconds) that were each staggered by 40 measurements (2 seconds). The segments needed to be identical lengths because the convolutional neural network requires inputs of the same size and dimensionality. Hypothetically, the algorithms that do not incorporate convolutional layers could use varying length inputs; however, the inputs were kept the same length between models not only to allow comparisons in model performance but also because the time of the boundary between activities is rarely known in practice. The segmentation process inevitably caused some segments to include data from two different activities when the segment spanned a change in activity. In order to account for this, a given segment was labeled as the activity that occurred most often within that segment. After segmentation, the training set consisted of a 18,646 deep stack of 80 by 3 segments and the test set consisted of a 8,806 deep stack of 80 by 3 segments.

The segmented data was processed into two sets of three-dimensional arrays. The first set was composed of a stack of the segmented and min-max scaled raw data from the three axes. The second set was composed of a stack of statistical features that were extracted from the first set's segments. Features were extracted from each of the three axes. These features included: mean, variance, minimum, maximum, absolute energy, sum of absolute change, kurtosis, skewness, 25th quantile, 50th quantile, 75th quantile, and complexity with a lag of 1, 2 and 10.

Several of the more complicated time series features were found as a part of the tsFresh time series analysis Python package (Christ, Braun, Neuffer & Kempa-Liehr 2018) and warrant more explanation. Complexity, which roughly quantifies how complicated a time series is in regards to the relative amount of peaks and valleys (Batista, Keogh, Tataw & De Souza 2014), was calculated as:

$$\sqrt{\sum_{i=1}^{n-2lag} (x_i - x_{i+1})^2}. \tag{2.1}$$

The absolute energy was calculated as the sum over the squared values of each time series segment:

$$\sum_{i=1} x_i^2 \tag{2.2}$$

and the absolute sum of changes was calculated as:

$$\sum_{i=1}^{n-1} \mid x_{i+1} - x_i \mid . \tag{2.3}$$

### 2.2.2 Model Fitting and Assessment

The analysis of the WISDM dataset was performed using Python 3 with Keras (Chollet 2015) and Scikit Learn packages (Pedregosa 2011). Five different models were developed in the analysis: random forest, multi-layer perceptron, 1-D convolutional neural network, recurrent neural network, and a convolutional recurrent neural network. The random forest and multi-layer perceptron models utilized extracted features from each data segment as inputs, while the convolutional and recurrent neural networks used the segmented and min-max scaled raw data as inputs.

The random forest modeling was performed on the array of 14 features from each of the three axes and was composed of 500 trees. Gini impurity was utilized as the criterion for measuring the quality of each split in the individual trees and the number of features considered at each split was the rounded square root of the total number of features, 6.

The traditional neural network, sometimes known as a multi-layer perceptron (MLP), consists of 4 layers. The input layer is composed of 42 neurons - one for each of the 14 features from the 3 axes. The input layer feeds into the first hidden layer, a dense layer composed of 200 neurons with ReLU activation functions. The second hidden dense layer contains 100 neurons with ReLU activation functions. The second hidden layer has 50% dropout as it feeds into the output layer, which is composed of six output neurons with a softmax activation function. Each output neuron corresponds to one of the six unique activities. The MLP was fit with a categorical cross-entropy loss function and an adam optimizer. Training was performed over 10 epochs. Details on the architecture of the MLP are shown in Table 2.1.

| **Layer** | **Layer Hyperparameters** | **Activation Function** | **Output Shape** |
|---|---|---|---|
| Dense | 200 Neurons | ReLU | (,200) |
| Dense | 100 Neurons | ReLU | (,100) |
| Dropout | 50% Dropout Rate | | (,100) |
| Dense Output | 6 Neurons | Softmax | (,6) |

Table 2.1: The architecture of the multi-layer perceptron used in the analysis of the WISDM dataset.

The 1-D convolutional neural network consists of an input layer, an output layer, and six hidden layers. The input layer contained a neuron for every measurement in a given segment. The first two hidden layers were 1-D convolutional layers that were composed of 100 filters each, with a span of 5 measurements, a stride of 1 measurement, and ReLU activation functions. After a max pooling layer with a patch size of 3, the model has another pair of convolutional layers that are composed of 160 filters with a span of 5 and a stride of 1. After a global averaging layer, the model has a 50% dropout and feeds into the 6 neurons in the output layer. The 1-D CNN was fit with a categorical cross-entropy loss function and an adam optimizer. The model was trained over 10 epochs. Details on the architecture of the convolutional neural network are shown in Table 2.2.

| **Layer** | **Layer Hyperparameters** | **Activation Function** | **Output Shape** |
|---|---|---|---|
| Convolutional | 100 5×3 Filters | ReLU | (,76,100) |
| Convolutional | 100 5×1 Filters | ReLU | (,72,100) |
| Max Pooling | 3 Unit Pool size | | (,24,100) |
| Convolutional | 160 5×1 Filters | ReLU | (,20,160) |
| Convolutional | 160 5×1 Filters | ReLU | (,16,160) |
| Global Average Pooling | | | (,160) |
| Dropout | 50% Dropout Rate | | (,160) |
| Dense Output | 6 Neurons | Softmax | (,6) |

Table 2.2: The architecture of the one dimensional convolutional neural network used in the analysis of the WISDM dataset.

The recurrent neural network is composed of an input layer, an output layer, and two hidden layers. The first hidden layer contains 200 long short-term memory cells with hyperbolic tangent activation functions and sigmoid recurrent activations functions. After 50% dropout, the first hidden layer feeds into a dense layer of 100 neurons with ReLU activation functions. The output layer is composed of 6 neurons with a softmax activation function. The model was fit with a categorical cross-entropy loss function and an adam optimizer. Model training was performed over 10 epochs. Details regarding the architecture of the recurrent neural network are shown in Table 2.3.

| **Layer** | **Layer Hyperparameters** | **Activation Function** | **Output Shape** |
|---|---|---|---|
| LSTM | 200 Neurons | TanH and Sigmoid | (,200) |
| Dropout | 50% Dropout Rate | | |
| Dense | 100 Neurons | ReLU | (,100) |
| Dense Output | 6 Neurons | Softmax | (,6) |

Table 2.3: The architecture of the long short-term memory recurrent neural network used in the analysis of the WISDM dataset.

The convolutional recurrent neural network is composed of an input layer, an output layer, and 4 hidden layers. The first and second hidden layers are 1-D convolutional layers that have 100 filters each, with spans of 20 and 10, respectively. Both layers have strides of 1 and ReLU activation functions. The 3rd hidden layer is a max pooling layer with a patch size of 2. The last hidden layer contains an LSTM layer with 10 cells, tanh activation functions, and sigmoid recurrent activation functions. This last layer feeds into the six neurons of the output layer, which have a softmax activation functions. The convolutional recurrent neural network was fit with a categorical cross-entropy loss function and an adam optimizer. Model training was performed over 10 epochs. Details regarding the architecture are shown in Table 2.4.

| Layer | Layer Hyperparameters | Activation Function | Output Shape |
|---|---|---|---|
| Convolutional | 100 5 $\times$ 3 Filters | ReLU | (,76,100) |
| Convolutional | 100 5 $\times$1 Filters | ReLU | (,72,100) |
| Max Pooling | 2 Unit Pool size | | (,36,100) |
| LSTM | 10 Neurons | TanH and Sigmoid | (,10) |
| Dense Output | 6 Neurons | Softmax | (,6) |

Table 2.4: The architecture of the convolutional long short-term memory (LSTM) recurrent neural network used in the analysis of the WISDM dataset.

Since the goal of this analysis of the WISDM dataset was to reproduce benchmark results, no cross-validation or bootstrapping was performed for model validation. Instead, results from the test set were compared to published results. Specifically, the sensitivity, positive predictive value (PPV), F1 score, and Brier score for the test set results were utilized in the assessment of model performance. Model accuracy is not presented as the data set is highly unbalanced.

Sensitivity, or recall, is calculated as the average ratio of true positives to true positives plus false negatives for all 6 classes. Positive predictive value, also known as precision, is calculated as the average ratio of true positives to true positives plus false positives for all 6 classes. The F1 score, which is commonly used in the field of machine learning, is the macro averaged harmonic mean of recall and precision for all 6 classes. For a binary outcome, the Brier score is commonly formulated as $\frac{1}{N}\sum_{t=1}^{N}(f_t - o_t)^2$ (Harrell Jr 2015); however, the original definition by Brier (Brier 1950) is used in this case as the original definition can be used in multi-class scenarios. The multi-class Brier score in Table 2.5 was calculated as $\frac{1}{N}\sum_{t=1}^{N}\sum_{i=1}^{R}(f_{ti} - o_{ti})^2$. In this formulation, $f$ is the forecasted probability, $o$ is the true outcome, $N$ is the number of samples in the test set, and $R$ is the number of categories in the outcome. A macro averaged multi-class Brier Score could be obtained by dividing this definition of Brier score by the number of classes; however, this analysis uses the original definition by Brier instead. The Brier score is somewhat challenging to interpret, but in general better performing models have Brier scores closer to 0.

## 2.3 Results

### 2.3.1 Data Description

Each participant in the WISDM study collected data for varying time intervals of approximately 8 to 45 minutes (Figure 2.2). Although 70% of the subjects were randomly assigned to the training set, the total length of the training set is only approximately 5 hours compared to the 10 hours of the test set. This is due to imbalances in the amount of data collected for each participant but would likely not cause a problem due to the large amount of data in each set.

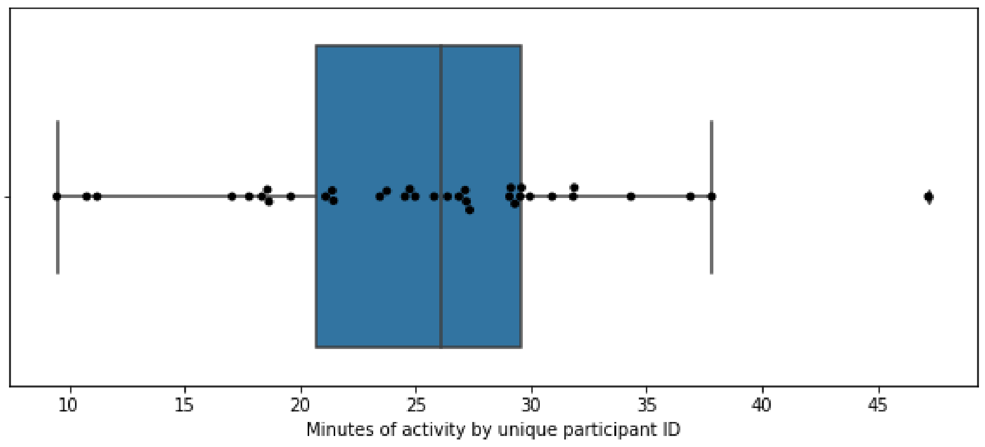


Figure 2.2: Minutes of total activity by participant in the WISDM dataset.

The dataset is highly unbalanced with vast differences in the proportion of time spent performing each activity. The majority of the dataset is composed of jogging and walking with relatively little time spent on the upstairs, downstairs, sitting, and standing activities (Figure 2.3).

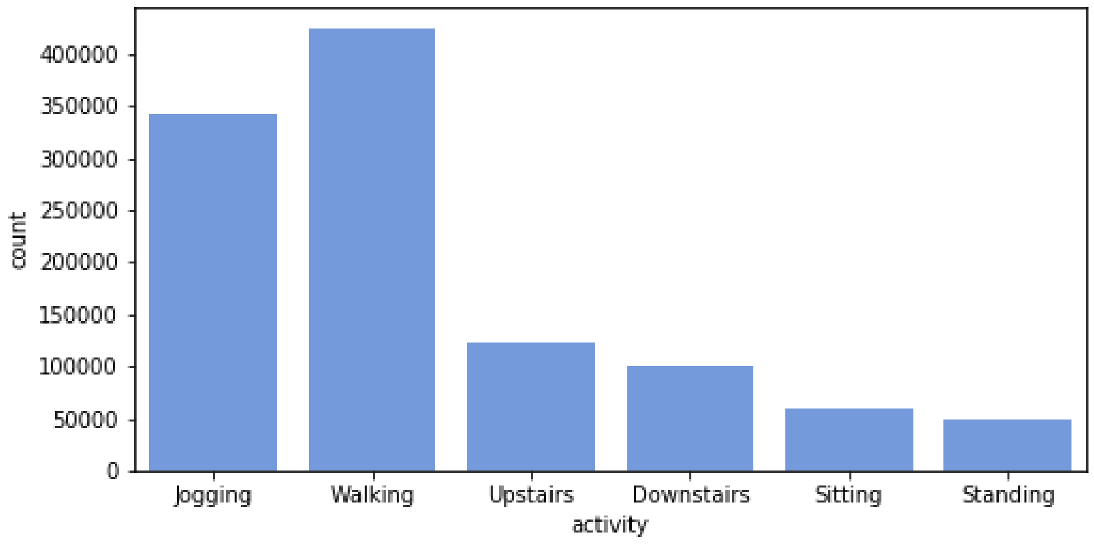


Figure 2.3: The number of measurements by activity in the WISDM dataset.

### 2.3.2 Model Performance

The macro averaged multi-class sensitivity, positive predictive value, F1 score, and multi-class Brier score of the test set on each of the 5 models is presented in Table 2.5. Across all metrics of model performance, the models that use extracted statistical features performed worse than the neural networks that use the scaled raw data as inputs. Both the MLP and random forest models performed similarly. Of particular note is the convolutional neural network model, which outperformed all the other models on every metric.

| **Model** | **Sensitivity** | **PPV** | **F1 Score** | **Brier Score** |
|---|---|---|---|---|
| **Feature Input Models** | | | | |
| Random Forest | 0.67 | 0.72 | 0.67 | 0.33 |
| Multilayer Perceptron | 0.68 | 0.763 | 0.69 | 0.33 |
| **Scaled Data Input Models** | | | | |
| Convolutional Neural Network | 0.88 | 0.89 | 0.88 | 0.13 |
| Recurrent Neural Network | 0.78 | 0.80 | 0.78 | 0.24 |
| Convolutional Recurrent Neural Network | 0.82 | 0.80 | 0.80 | 0.24 |

Table 2.5: The multi-class macro-averaged sensitivity, positive predictive value (PPV), F1 score, and Brier score for the model performance with the test set of the WISDM analysis.

## 2.4 Discussion

WISDM is a benchmark accelerometer dataset in the field of human activity recognition. The goal of the analysis of the WISDM dataset is to predict which activity out of six classes is being performed based entirely on tri-axial accelerometer readings. For the purposes of this thesis, WISDM was utilized as a benchmark dataset to apply commonly used machine learning algorithms on data that already have been analyzed for comparison.

The analysis of the WISDM data utilized several different machine learning algorithms that had two different types of inputs: manually extracted global features or scaled raw data. The MLP and random forest models used 42 extracted features as inputs. Both models had similar performance with a multi-class Brier scores of 0.33 and macro averaged F1 scores of 0.67 and 0.69, respectively. Across all model metrics, the models developed with global features performed worse than the models developed with scaled raw data as inputs. This difference in performance could be due to the local features, that are found through backpropagation, containing more relevant information than the manually extracted global features.

Of particular interest is the convolutional neural network, which outperformed all other models across every performance metric with a macro averaged F1 score of 0.88 and a multi-class Brier score of 0.13. Interestingly, this model even outperformed the convolutional recurrent neural network, that hypothetically is expected to work best as it both extracts local features in the convolutional layers and uses long short-term memory to utilize information from past segments to inform the predictions of current segments. Adding a convolutional layer to the recurrent network's architecture only marginally improved performance in terms of the sensitivity and F1 score.

One goal of the analysis of this dataset was to compare model performance with published results. Unfortunately, methodological differences between our analysis and published results make this task challenging. The original publication of the WISDM data analysis presented decision tree, logistic regression, and MLP models developed with 43 features from 10 second epochs of data. Their analysis did not split the data into a test or training set and evaluated the models' performances on the training data. Additionally, the original publication used average classification accuracy to compare with a straw man model that only predicts the most commonly occurring activity - walking. They found that the MLP performed the best, with 91.7% accuracy relative to the straw man model that had 37.2% accuracy (Kwapisz et al. 2011). We avoided presenting accuracy as a model metric due to the dataset being highly unbalanced, and instead presented F1 Score and Brier Score as metrics for model performance. Additionally, the metrics presented in our analysis were based on the models' performance on a randomly selected test set that contained data from different participants who were not in the training set in order to reduce over-optimism in the results.

Another study analyzed the WISDM data with a random forest and obtained a macro averaged F1 Score of 0.981; however, this study did not split the data into a test and training set and is not comparable to our results (Walse, Dharaskar & Thakare 2016). A more recent study analyzed the WISDM dataset using convolutional neural networks (Ignatov 2018). This study used the novel approach of appending statistical features to neurons in the fully connected layer after the convolutional layer. Additionally, this study evaluated the performance of a test set similarly as we have done. The convolutional neural network in their study had an accuracy of 91% in the test set, but they did not present any other model performance metric. Although accuracy was not presented in our results as it is a poor metric for unbalanced data, the convolutional neural network in our study did perform similarly, with approximately 91% accuracy.

A major weakness in the previous analysis of the WISDM dataset is a general lack of thoroughness in the model validation. The data was only split into a single test and training set with no attempts at cross-validating or bootstrapping to validate the models. Changing the random sampling for participants in the test set could potentially alter the results. This problem is especially critical for the recurrent neural network, which displayed unstable training between epochs.

Even with the lack of validation and difficulty in comparing results to prior research, we were able to develop several models that performed reasonably well or as good as the published results, although completely fair comparison was not possible due to the different evaluation methods. The feature based models performed fairly well for a 6 class outcome with average sensitivity near 70% and multi-class Brier score of 0.33. The convolutional neural network performed extremely well with sensitivity and positive predictive value close to 90%.

# Chapter 3

# Analysis of Accelerometry Data

## 3.1 Introduction

The accelerometry dataset is composed of 779 unique trials in which 251 participants were mailed a tri-axial Actigraph accelerometer to wear for approximately one week per trial. The participants were requested to wear the Actigraph for the entire duration of the trial, except when sleeping. Additionally, participants were requested to keep a timestamped log of when they received the Actigraph in the mail, when they returned the Actigraph to postal services, and any other potential issues such as non-compliance. Physical activity was measured with 1 minute epoch from the x, y, and z axes. Figure 3.1 demonstrates a visualization of one trial's vector magnitude, which is calculated by taking the square root of the sum of the squared values of the x, y, and z axes, $\sqrt{x^2+y^2+z^2}$.

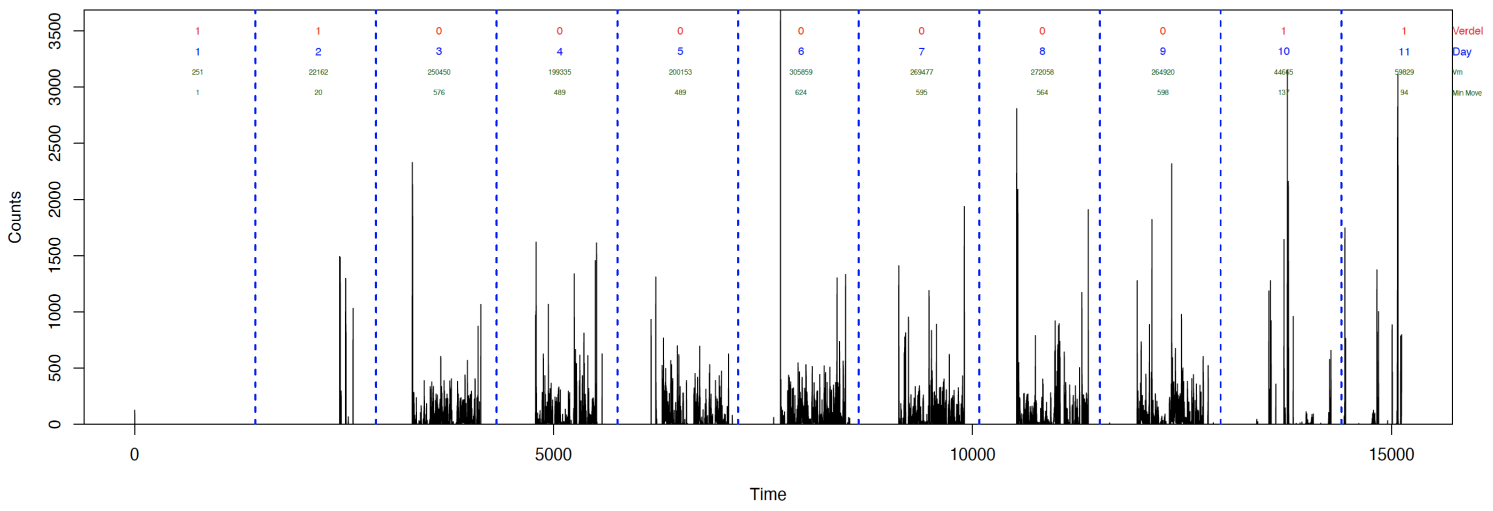


Figure 3.1: Example of data for accelerometry trial. The black lines represent the count of measurements on the x axis with a one minute epoch. Vertical dashed blue lines indicate midnight. The red Verdel label indicates 0 for a human wear day and 1 for a delivery day. The blue text enumerates the day of the trial. The first line of green text gives the sum of the vector magnitudes across the entire day and the second line of green text indicates the minutes of wearing during that day.

Although the accelerometry dataset is rich in information, the dataset has several problems that make the analysis challenging. Unlike the WISDM dataset, the accelerometry data was collected outside of a laboratory setting, spans a massive amount of time, and relies upon unsupervised participant adherence to protocol. In addition to some adherence problems, Actigraphs were activated prior to being mailed to the participant and only deactivated once they were returned to the laboratory by postal services. Due to this design, over half of the acceleration data across all of the trials is recorded data from movement while the Actigraph was in transit to the participant or laboratory. The difference in the acceleration data between human wear and delivery day can be seen in Figure 3.1, where the first couple days and last couple days of data collection occurred during delivery and the days in between occurred during human wearing.

Prior research has developed algorithms for detecting when participants are not wearing the accelerometer (Choi et al. 2012), but to the best of our knowledge, research has not yet been conducted on training algorithms to predict whether a given day of activity is a delivery or human wear day. The primary goal of this study is to develop algorithms for the classification of a day as ”delivery” or ”human wear” in the context of supervised learning. These algorithms can be used in future research to automate the removal of days with high probability of being delivery data so that analyses can focus on human wear activity.

## 3.2 Methods

The data processing and modeling of the accelerometry dataset was performed using R version 4.0.0. The modeling was repeated in Python 3. R code for the processing and modeling of the data can be found in Appendix B.

### 3.2.1 Data Preprocessing

The accelerometry dataset required extensive preprocessing prior to preparing the data for modeling. Four trials had accelerometer data measured with 30 second epoch instead of 1 minute epoch. These data were collapsed to be formatted as 1 minute epoch using the Physical Activity R package. Sixty-six trials were discarded as they had no associated activity labels. Six more trials were removed because they had associated ”rewear” trials that were collected due to data collection issues in the original trials. Additionally, 40 trials were removed as the participant and trial identification indicated they were duplicates.

Due to the self-reported nature of the data labeling, some trials had issues from user recording error or non-adherence to protocol. For example, 40 trials had every day labeled as human wear days even when the study design and the plots of the trial clearly indicated a lack of human activity during many days of the trial. An example of one such trial is presented in Figure 3.2. These trials likely occurred from participants not filling out the requested log sheet. All 40 trials that contained only human wear days were removed from the analysis.

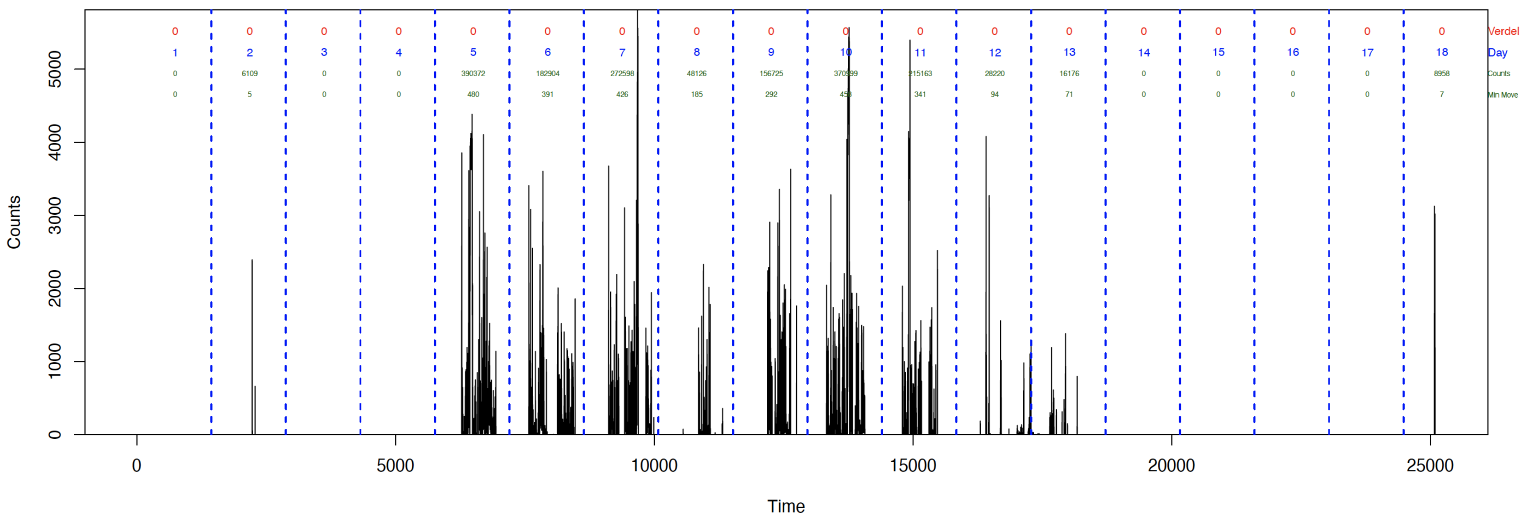


Figure 3.2: Example of spuriously labeled accelerometry trial. Only the days 5 through 11 display activity that is indicative of human movement; however, every day of the trial is labeled 0 (i.e., human wearing) due to a recording error.

Another issue caused by user error, though less common, was clearly mislabeled delivery days that were adjacent to the period of human activity labeled days. An example of a trial with a potentially mislabeled day is presented in Figure 3.3. Every trial in the dataset was examined visually, and data labeled as delivery days that clearly resembled human activity were removed from the dataset. In total, only 36 of these type of days were identified. Due to the subjective nature of this omission, the analyst was careful to be conservative in the exclusion of potentially mislabeled days.

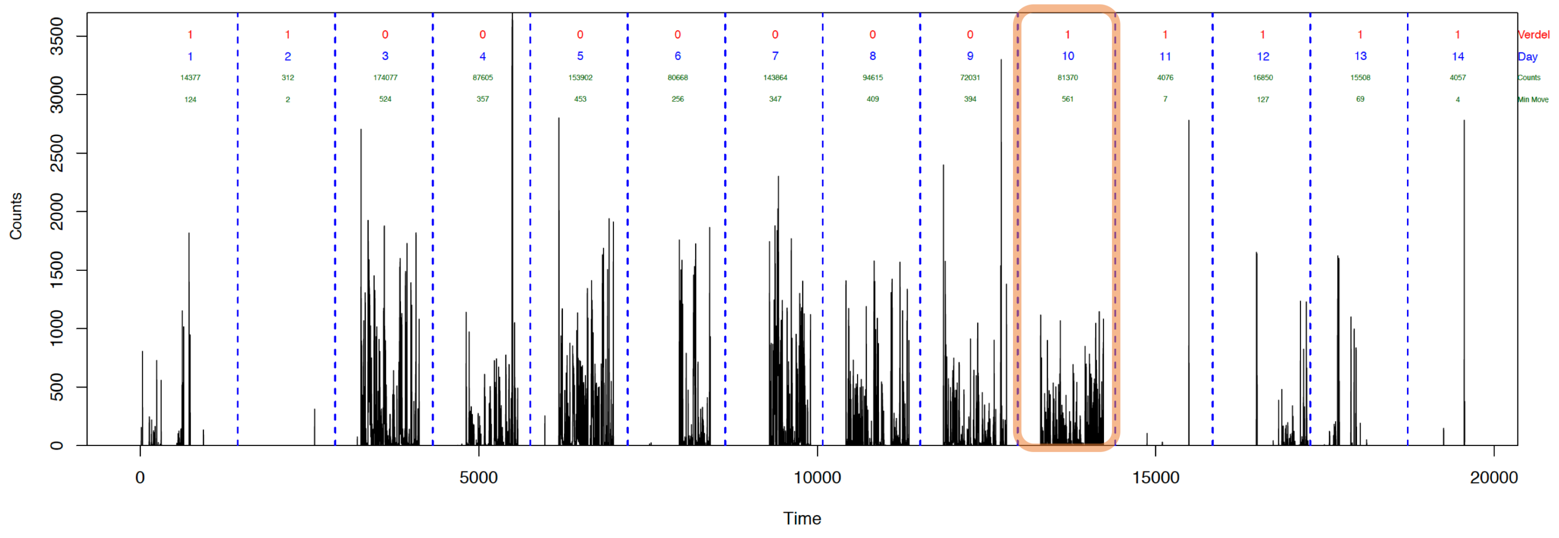


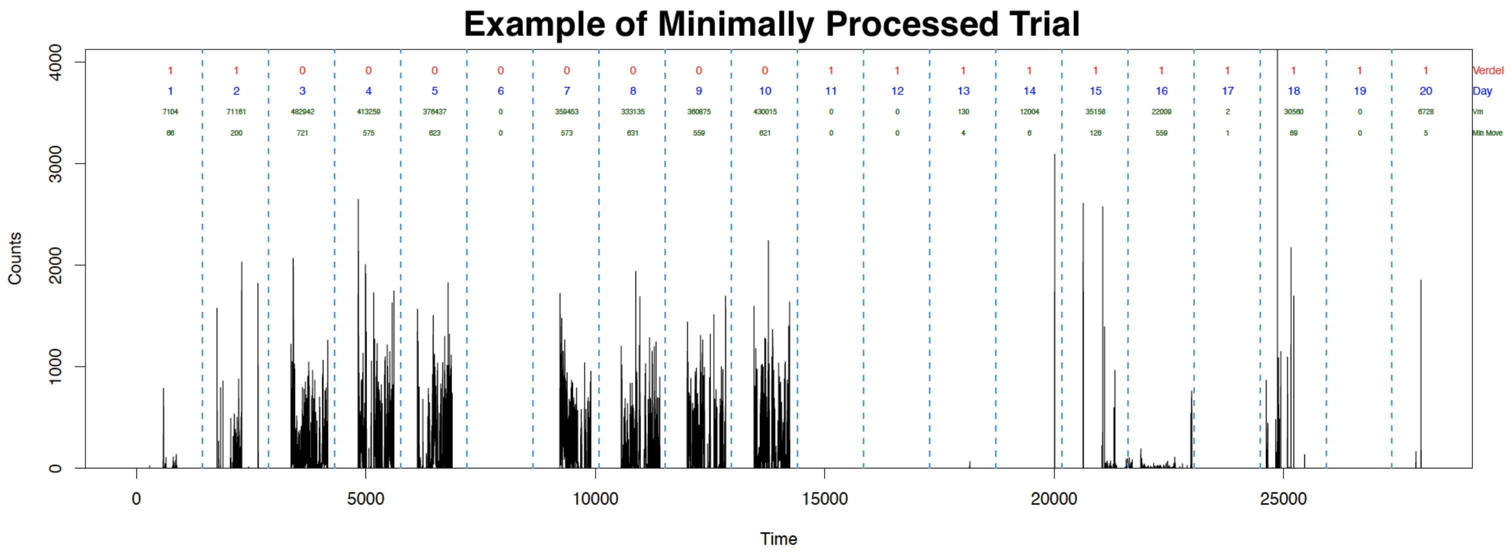
Figure 3.3: Example of spuriously labeled day in accelerometry trial. The orange highlighted day was labeled as delivery due to absence of a log entry; however, the activity clearly suggests human wearing.

The unit of interest for this analysis is the prediction of an entire day as either delivery or human wear. As the data are measured with 1 minute epoch, a complete day consists of 1440 measurements for each of the three axes. Every trial contained at least one incomplete day during which less than 1440 measurements were taken. This occurred due to the accelerometer being activated or deactivated at any time other than midnight. Since the convolutional layers of a neural network require all inputs to be the same shape, the days that contained less than 1440 measurements were zero-padded to a length of 1440. If the truncated day occurred at the start of the trial, the zero-padding occurred from midnight to the time the Actigraph was activated. If the truncated day occurred at the end of the trial, the zero-padding occurred from the deactivation time to the next midnight.

### 3.2.2 Data Processing

In addition to the preprocessing, the data was also processed using procedures designed to remove days that contain little information. Any day that had a total vector magnitude of less than 5000 or less than 10 minutes of movement was removed from the dataset. Additionally, any day that was labeled as human wear was removed if less than 120 minutes of total activity occurred as this indicated a large amount of non-compliance with the protocol.

Data that was only preprocessed as described in Section 3.2.1. was denoted as "minimally processed" data, while data that was both preprocessed and processed as described in this section was denoted as "fully processed." In this analysis both the minimally and fully processed datasets were modeled in order to explore different algorithm's capabilities of handling messier data. The minimally processed data approximates accelerometer data that is confounded with participant non-adherence, while the fully processed data is a much cleaner dataset. An example of the differences between the minimally and fully processed data is presented in Figure 3.4.



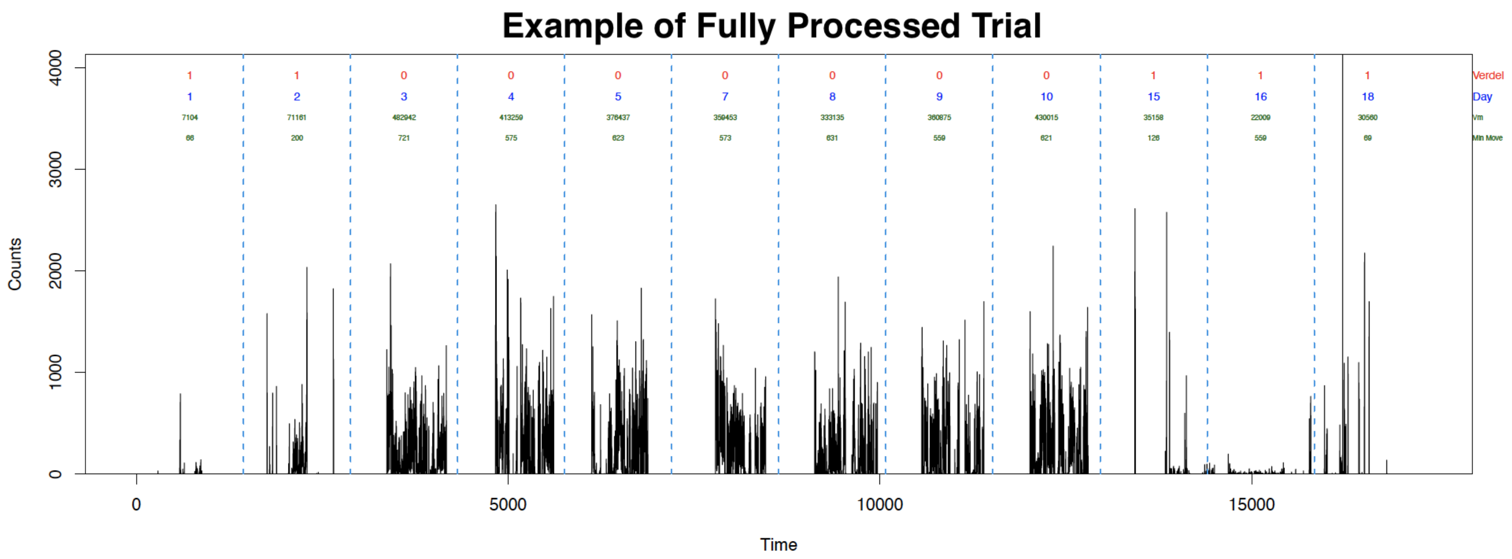


Figure 3.4: Example of minimally and fully processed data. The non-compliance human activity day 6 is removed in the fully processed data. Additionally, the large stretch of several zero activity delivery days from day 11 to 14, as well as days 17,19, and 20, are removed in the full processing.

To prepare the data for modeling, the data from each day was segmented into lengths of 1440 measurements between the hours of 0:00 and 23:59. These segments were reshaped into three dimensional arrays of stacks of 1440 by 3. The day long segments were used as inputs in the convolutional and recurrent neural networks or features were extracted from them for use in the multilayer perceptron, random forest, and logistic regression models. Features were only extracted from the vector magnitude of all three axes. The extracted features include: mean, variance, maximum, 95th quantile, absolute energy, absolute change in energy, kurtosis, and skewness. The features are defined as previously described in Section 2.2.1. All the features and the data from the x,y, and z axes were mean centered and scaled by their standard deviation. This scaling step is critical for achieving convergence in many of the models used in this analysis. Scaling was performed separately for the test and training sets as described in Section 3.2.4.

### 3.2.3 Modeling

Seven different models were developed in the analysis: random forest, multi-layer perceptron, logistic regression, mixed-effects logistic regression, 1-D convolutional neural network, recurrent neural network, and a convolutional recurrent neural network. The random forest, regression models, and multi-layer perceptron utilized extracted features from each data segment as inputs, while the convolutional and recurrent neural networks used the scaled raw data as inputs.

The random forest model was developed using an array of 9 features that were extracted from the vector magnitude of all three axes and was composed of 500 trees. Gini impurity was utilized as the criterion for measuring the quality of each split in the individual trees and the number of features considered at each split was the rounded square root of the number of total features, 3.

The logistic and mixed-effects logistic regression models were fit with the 9 extracted features. Each of the 9 features was flexibly fit with a restricted cubic spline with three knots. The mixed effects model was fit with a random intercept for participant.

The multi-layer perceptron (MLP) consists of 3 layers. The input layer is composed of 9 neurons - one for each of the features. The input layer feeds into the first hidden layer, a dense layer consisting of 200 neurons with ReLU activation functions. This dense layer has 50% dropout as it feeds into the output layer, which is composed of a single output neuron with a sigmoid activation function. The MLP was fit with a binary cross-entropy loss function and an adam optimizer. Training was performed over 10 epochs. Details regarding the architecture of the MLP can be found in Table 3.1.

| **Layer** | **Layer Hyperparameters** | **Activation Function** | **Output Shape** |
|---|---|---|---|
| Dense | 200 Neurons | ReLU | (,200) |
| Dropout | 50% Dropout Rate | | (,200) |
| Dense Output | 1 Neuron | Sigmoid | (,1) |

Table 3.1: Architecture of multi-layer perceptron neural network.

The 1-D convolutional neural network consists of an input layer, output layer, and six hidden layers with ReLU activation functions. The input layer contains a single neuron for every measurement in a given day. The first hidden layer is a 1-D convolutional layer with 200 filters that span 5 measurements with a stride of 1. After a max pooling layer with a patch size of 4, the model has another pair of convolutional and max pooling layers that are composed of 64 filters with a span of 5 and a pooling size of 4, respectively. The output is then flattened and fed into a 64 neuron dense layer. The dense layer feeds into an output layer with a single sigmoid activated neuron. The 1-D CNN was fit with a binary cross-entropy loss function and an adam optimizer. Training was performed over 10 epochs.. Details on the architecture can be found in Table 3.2.

| Layer | Filters/Pool Size/Units | Activation Function | Output Shape |
|---|---|---|---|
| 1-D Convolutional | 200 5 $\times$ 3 Filters | ReLU | (,1431,200) |
| 1-D Max Pooling | 4 Unit Pool Size | | (,357,200) |
| 1-D Convolutional | 64 5 $\times$ 1 Filters | ReLU | (,355, 64) |
| 1-D Max Pooling | 4 Unit Pool Size | | (,88,64) |
| Flattening | | | (,5632) |
| Dense | 64 Neurons | ReLU | (,64) |
| Dense Output | 1 Neuron | Sigmoid | (,1) |

Table 3.2: Architecture of convolutional neural network.

The recurrent neural network consists of an input layer, an output layer, and two hidden layers. The first hidden layer is composed of 30 long short-term memory cells with hyperbolic tangent activation functions and sigmoid recurrent activation functions. After a 40% dropout, the first hidden layer feeds into a dense layer of 100 neurons with ReLU activation functions. After 30% dropout, this layer fed into the output layer, which was composed of a single neuron with a sigmoid activation function. The model was fit with a binary cross-entropy loss function and an adam optimizer. Training was performed over 10 epochs. Details on the architecture can be found in Table 3.3.

| Layer | Layer Hyperparameters | Activation Function | Output Shape |
|---|---|---|---|
| LSTM | 30 Neurons | TanH and Sigmoid | (,30) |
| Dropout | 40% Dropout Rate | | (,30) |
| Dense | 100 Neurons | ReLU | (,100) |
| Dropout | 30% Dropout Rate | | (,100) |
| Dense Output | 1 Neuron | Sigmoid | (,1) |

Table 3.3: Architecture of long short-term memory (LSTM) recurrent neural network.

The convolutional recurrent neural network consists of an input layer, an output layer, and 4 hidden layers. The first and second hidden layers contain a 1-D convolutional layer and a max pooling layer with 200 filters of size 5 by 3 and a pooling size of 4 by 1, respectively. The 3rd hidden layer is another 1-D convolutional layer with 64 filters with a span of 5. Both convolutional layers had strides of 1 and ReLU activation functions. The 4th hidden layer is an LSTM layer with 30 memory cells with tanH activation functions and sigmoid recurrent activation functions. After 30% dropout, the LSTM layer fed into the output layer, which was composed of a single neuron with a sigmoid activation function. The convolutional recurrent neural network was fit with a binary cross-entropy loss function and an adam optimizer. Training was performed over 10 epochs. Details on the convolutional recurrent model architecture can be found in Table 3.4.

| **Layer** | **Layer Hyperparameters** | **Activation Function** | **Output Shape** |
|---|---|---|---|
| Convolutional | 200 5×3 Filters | ReLU | (,1431,200) |
| Max Pooling | 4 Unit Pool Size | | (,357,200) |
| Convolutional | 64 5×1 Filters | ReLU | (,355,64) |
| LSTM | 30 Neurons | TanH and Sigmoid | (,30) |
| Dropout | 30% Dropout Rate | | (,30) |
| Dense Output | 1 Neuron | Sigmoid | (,1) |

Table 3.4: Architecture of convolutional long short-term memory (LSTM) neural network.

### 3.2.4 Model Validation

Five-fold Monte Carlo cross-validation was performed for the analysis of both the minimally and fully processed datasets. For each repetition of the validation, the test and training sets were selected by randomly sampling 30% and 70% of participants' data, respectively. The previously described models were then fit with the training set and tested with the test set. The sensitivity, positive predictive value, F1 score, and Brier score were calculated for each of the models for every cross-validation. The mean and standard deviation of each model metric from the test set were then calculated.

## 3.3 Results

### 3.3.1 Data Description

The minimally processed data had a total of 10,546 days of information, while the fully processed data had a total of 7,433 days. Both forms of processing result in unbalanced data; however, the data is unbalanced in opposite directions because most of the low activity days removed in the full processing occurred on empty delivery days. The days removed during human wear were likely caused by non-adherence. Figure 3.5 shows that 54% of the days in the minimally processed dataset are human wear, while only 40% of the days in the fully processed dataset are human wear.

Each subject participated in a range of one to three trials in which they were asked to record their activity for a week. On average, the accelerometer for each trial was active for approximately 17 days. Across all trials, the average number of days per participant was approximately 42 days. After fully processing the data, the average number of days per participant was reduced to approximately 30 days. The count of days by participant in the minimally and fully processed dataset is shown in Figure 3.6. Although 70% of the subjects were randomly assigned to the training set for each cross-validation repetition, the size of the training set relative to the test set varied from a 7:3 split. This is due to imbalances in the amount of data collected for each participant but would likely not cause a problem due to the large amount of data in each set.

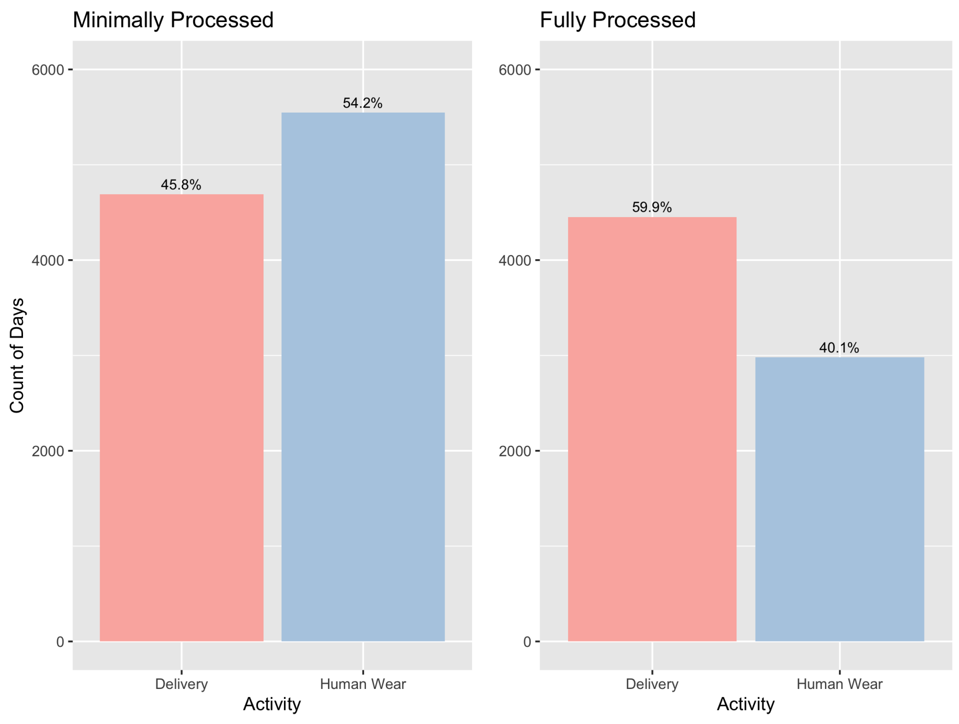


Figure 3.5: The count of days by activity in the minimally and fully processed datasets. The percentage of each activity is presented above each bar. The proportion of delivery days is decreased in the fully processed data.

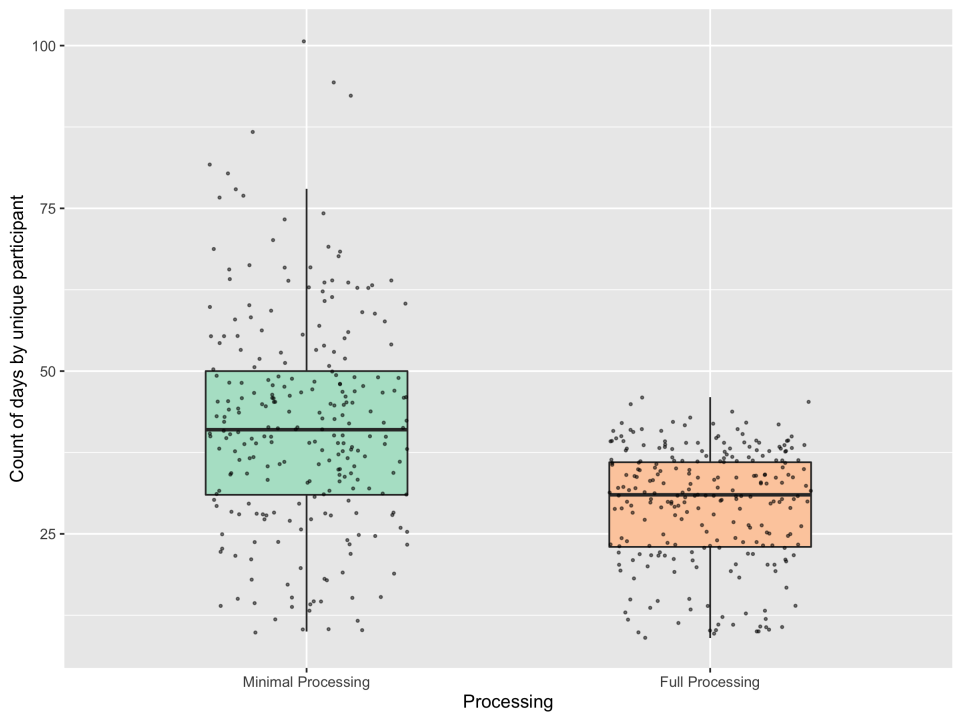


Figure 3.6: The number of days by participant for the minimally and fully processed datasets.

### 3.3.2 Model Performance

The average and standard deviation of the sensitivity, positive predictive value, F1 score, and Brier score across the 5 Monte Carlo cross-validations are presented in Tables 3.5 and 3.6 for the minimally and fully processed data, respectively. The corresponding results are presented in Figures 3.7 and 3.8. All models had a lower Brier score for the fully processed dataset compared to the minimally processed dataset; however, several models have a better F1 score in the minimally processed dataset. For both forms of processing, the recurrent architecture had the worst performance, while the convolutional recurrent architecture marginally outperformed the other models.

Out of the feature input models, the mixed-effects logistic regressions generally performed the worst, while the random forest marginally outperformed the other models in most of the metrics. The mixed effect model performed fairly well when used on the fully processed dataset, but performed poorly relative to the other feature input models when used to model the minimally processed data. Out of the scaled raw data input models, the recurrent neural network performed the worst. Similarly to the mixed effect model, the recurrent neural network's performance was particularly poor relative to other models when used to model the minimally processed dataset. The convolutional neural network performed very well in both the minimally and fully processed data.

| **Minimally Processed Dataset** | | | | |
|---|---|---|---|---|
| **Model** | **Sensitivity** | **PPV** | **F1 Score** | **Brier Score** |
| **Feature Input Models** | | | | |
| Random Forest | 0.981 (0.008) | 0.931 (0.006) | 0.955 (0.006) | 0.041 (0.004) |
| Generalized Linear Model | 0.984 (0.002) | 0.931 (0.004) | 0.957 (0.002) | 0.041 (0.002) |
| Generalized Mixed Effects Model | 0.944 (0.040) | 0.889 (0.033) | 0.916 (0.036) | 0.066 (0.019) |
| Multilayer Perceptron | 0.981 (0.004) | 0.927 (0.004) | 0.953 (0.004) | 0.043( 0.003) |
| **Scaled Data Input Models** | | | | |
| Convolutional Neural Network | 0.980 (0.003) | 0.926 (0.012) | 0.952 (0.007) | 0.044 (0.008) |
| Recurrent Neural Network | 0.936 (0.055) | 0.741 (0.163) | 0.815 (0.090) | 0.221 (0.190) |
| Convolutional Recurrent Neural Network | 0.988 (0.005) | 0.936 (0.006) | 0.961 (0.005) | 0.038 (0.004) |

Table 3.5: Average model performance metrics from 5 fold Monte Carlo cross-validation with standard deviation in parentheses for the minimally processed data. PPV: Positive Predictive Value.

| **Fully Processed Dataset** | | | | |
|---|---|---|---|---|
| **Model** | **Sensitivity** | **PPV** | **F1 Score** | **Brier Score** |
| **Feature Input Models** | | | | |
| Random Forest | 0.970 (0.007) | 0.944 (0.004) | 0.957 (0.005) | 0.023 (0.001) |
| Generalized Linear Model | 0.964 (0.005) | 0.937 (0.006) | 0.951 (0.005) | 0.026 (0.003) |
| Generalized Mixed Effects Model | 0.966 (0.007) | 0.935 (0.008) | 0.950 (0.006) | 0.027 (0.004) |
| Multilayer Perceptron | 0.958 (0.005) | 0.933 (0.005) | 0.945 (0.005) | 0.028 (0.002) |
| **Scaled Data Input Models** | | | | |
| Convolutional Neural Network | 0.962 (0.015) | 0.938 (0.009) | 0.950 (0.010) | 0.026 (0.004) |
| Recurrent Neural Network | 0.915 (0.047) | 0.839 (0.068) | 0.873 (0.035) | 0.074 (0.040) |
| Convolutional Recurrent Neural Network | 0.970 (0.008) | 0.949 (0.006) | 0.960 (0.006) | 0.021 (0.002) |

Table 3.6: Average model performance metrics from 5 fold Monte Carlo cross-validation with standard deviation in parentheses for the fully processed data. PPV: Positive Predictive Value.

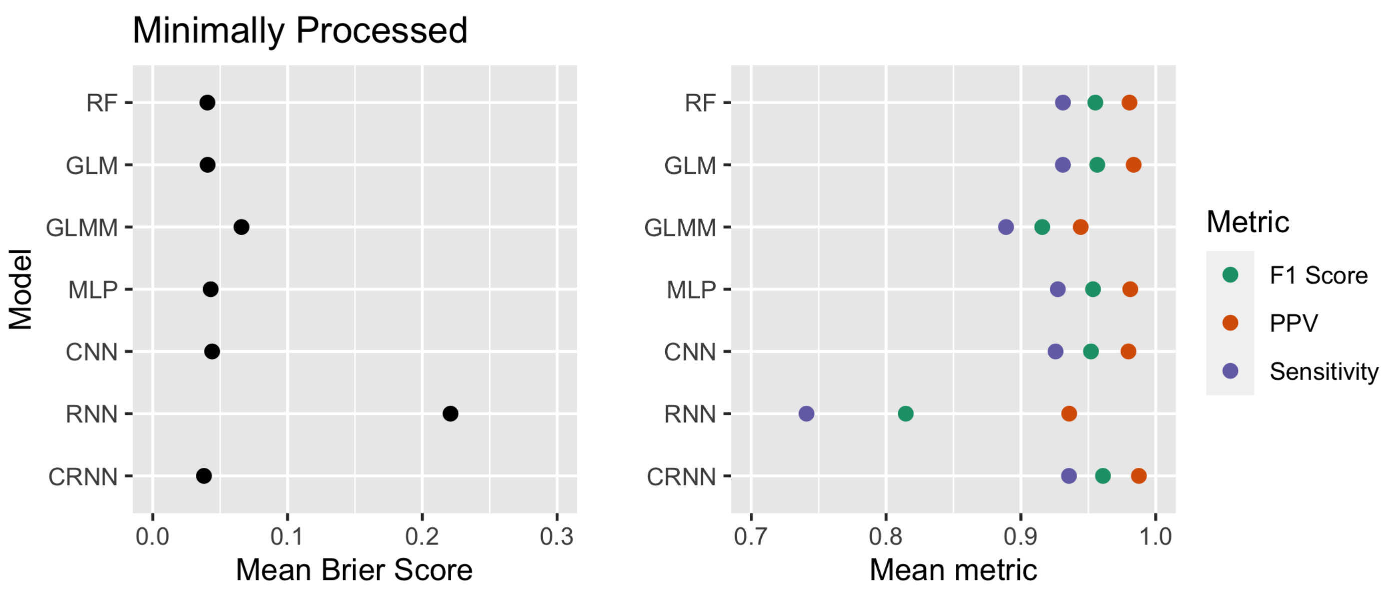


Figure 3.7: Cross-validated average model performance metrics on the minimally processed data. RF: Random Forest, GLM: Generalized Linear Model, GLMM: Generalized Linear Mixed Effects Model, MLP: Multilayer Perceptron, CNN: Convolutional Neural Network, RNN: Recurrent Neural Network, CRNN: Convolutional Recurrent Neural Network, PPV: Positive Predictive Value.

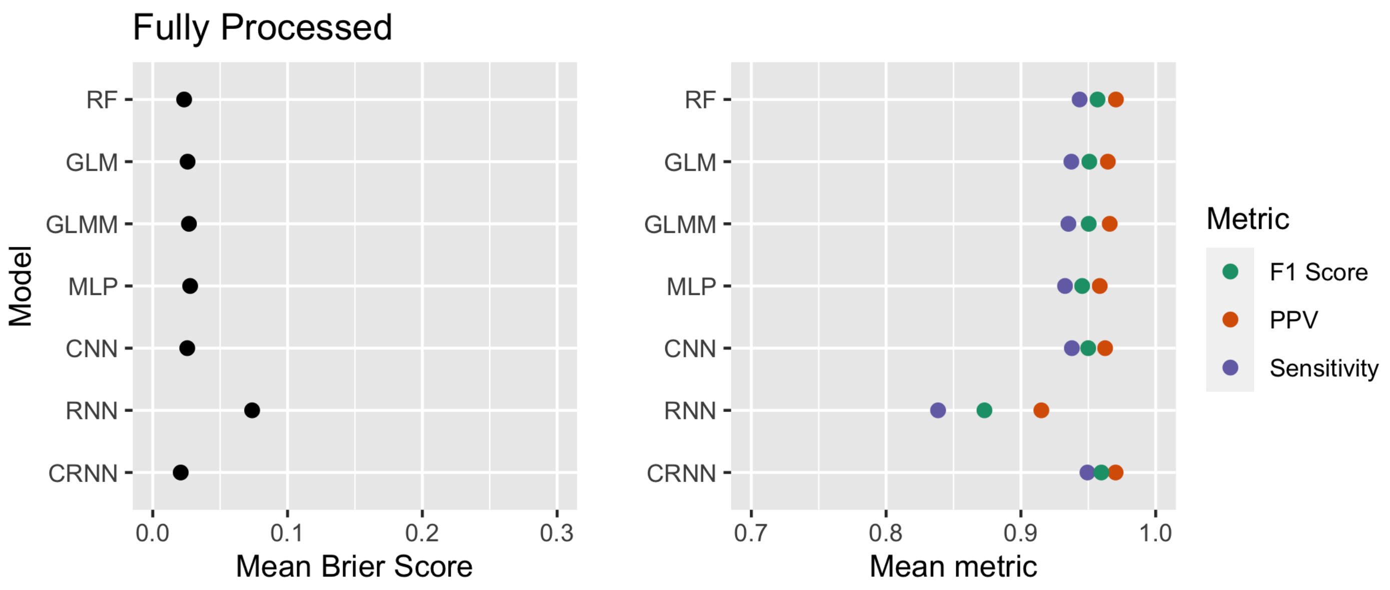


Figure 3.8: Cross-validated average model performance metrics on the fully processed data. RF: Random Forest, GLM: Generalized Linear Model, GLMM: Generalized Linear Mixed Effects Model, MLP: Multilayer Perceptron, CNN: Convolutional Neural Network, RNN: Recurrent Neural Network, CRNN: Convolutional Recurrent Neural Network, PPV: Positive Predictive Value.

## 3.4 Discussion

The accelerometry dataset used to develop our models consists of 779 trials in which 251 participants wore an Actigraph for approximately one week per trial. The participants were requested to wear the accelerometer for the entire duration of the trial, except when sleeping. Additionally, they were requested to keep a timestamped log of when they received the Actigraph in the mail, when they returned the Actigraph to postal services, and any other potential issues such as non-adherence. The analysis of the dataset is challenging due to the inclusion of accelerometer activity from the delivery of the Actigraph. The goal of this study was to develop and compare the performances of several models that can discriminate between human wear and delivery activity on a given day. The purpose of classifying delivery days is to allow a user to easily process accelerometer data that is confounded with delivery data.

Several machine and statistical learning models were developed in order to address our goal. Additionally, the models were fit to the dataset after two types of processing: minimal processing and full processing. For the fully processed data, days with less than either 10 minutes of activity or 5000 activity counts were removed in addition to the minimal processing. Human wear days with less than 2 hours of activity were also removed.

The models fit to the dataset can broadly be classified into two types: models that use extracted features as inputs and models that use the scaled raw data as inputs. Out of the models that utilize extracted features as inputs, the random forest and logistic regression marginally outperformed the multilayer perceptron. The mixed effects model performed approximately as well as the the random forest and logistic regression model for the fully processed data, but performed poorly when modeling the minimally processed data.

Out of the scaled raw data input models, the convolutional and convolutional recurrent neural networks performed very well, while the recurrent neural network performed the worst out of all models using both the minimally processed and fully processed datasets. The slightly stronger performance of the convolutional recurrent neural network relative to the convolutional neural network indicates that incorporating time dependencies is helpful; however, the poor performance of the recurrent neural network indicates that the data greatly benefits from being reduced in dimensionality through convolutional layers before the recurrent layer processes the sequence. It is likely that the LSTM recurrent neural network has difficulties processing the thousands of days input with a length of 1440 measurements per day.

Although the convolutional recurrent neural network had the best performance, the structure of the dataset makes the model somewhat naive in that it cannot differentiate between unique trials. Ideally, the data would have been zero padded between trials in order to reset the internal memory of the recurrent layers. A potential improvement to the convolutional recurrent neural network would be the inclusion of a bidirectional LSTM layer. These layers incorporate information from both future and past states in an input sequence and have recently been shown to have improved performance over traditional LSTMs in certain contexts (Chiu & Nichols 2016).

Models fit to the minimally processed dataset had worse Brier scores than those fit to the fully processed data. The decrease in Brier score is likely due to models mistaking non-adhering human wear days as delivery days. This can be seen in the decrease in positive predictive value for the minimally processed dataset relative to the fully processed. On the other hand, the sensitivity decreases in almost all models for the fully processed analysis compared to the minimally processed analysis. This is likely due to changes in the proportion of human wear days relative to delivery days in the dataset. Thus, when comparing models between the differentially processed datasets, the Brier score may be a better metric to use than F1.

Although the convolutional recurrent neural network had the best performance, many of the other models still had excellent performance in terms of both Brier and F1 score. A possible reason that the majority of the models accurately performed the classification task so similarly is that a ceiling effect in the model performance was reached. Many of days that are being misclassified by the models may be exceedingly difficult to correctly classify, while still accurately modeling the rest of the data.

One of the limitations to this study is that the hyperparameters of the neural network models were tuned based on the results of a single test set. The cross-validation results indicate that overfitting from this modeling approach is not likely to be an issue. However, it could still be a concern, especially considering only 5 repetitions of Monte Carlo cross-validation were performed. Ideally, more resamplings would have been performed, but the large amount of computing power required to fit the convolutional and recurrent neural networks made more resamplings difficult.

Another limitation to the generalizability of this study is that the models were built for and fit to accelerometer data collected with one minute epoch. It is possible that some of the models fit to our data would not be applicable to datasets with different temporal resolution such as 1 second epoch. The models based on scaled extracted features would likely perform similarly, but the convolutional and recurrent neural networks would likely need retraining and possibly architectural updates to model other resolutions accurately. Fortunately, if another dataset is collected at a higher resolution, the data can be easily collapsed to a one minute epoch.

# Chapter 4

# Conclusion

The focus of this thesis is the analysis of data collected from tri-axial accelerometers for human activity research. This type of data presents unique challenges due to massive size, participant non-adherence, and the data being collected outside of controlled laboratory settings. Another difficult issue in the analysis of accelerometry data is that accelerometers are often activated prior to shipment to participants and are not deactivated until they are returned to the laboratory. Due to this design, large portions of accelerometry datasets are often recorded while the accelerometers are in transit to the participant or laboratory. The purpose of this thesis is to explore and develop algorithms that can accurately classify a given day in an accelerometer dataset as a human wear day or a delivery day. These algorithms can then be implemented by users in the automated cleaning of accelerometry data that is adulterated with delivery activity.

Using our large dataset of 7,433 days of activity after processing, we trained several statistical and machine learning models in a supervised learning context. Our models were able to discriminate between human wear and delivery days with a high degree of accuracy. A hybrid convolutional recurrent neural network had the best performance, with a mean 5 fold cross-validated Brier score of 0.021 and F1 score of 0.96. The logistic regression and random forest models also performed well with mean Brier scores of 0.026 and 0.023, respectively. The models in this analysis have not been externally validated, but the mechanistic nature of delivery activity and high performance of the models during internal validation suggest that the models would have good performance when applied to new data.

Although the convolutional recurrent neural network had the best performance for our dataset, it has a few barriers to widespread implementation. First, the model was trained on a dataset with a temporal resolution of one measurement per minute. The trained neural network would likely perform well for accelerometry datasets with the same temporal resolution but may not be generalizable to other resolutions since the model learned to recognize local features only at one minute epoch. Fortunately, if the data was recorded at a higher resolution, it could easily be collapsed to a lower resolution. Another issue in the widespread implementation of the convolutional recurrent neural network is its high computational cost and dependency on the Keras package. Although it is fairly simple to export and import Keras models, the requirement of both installing Keras and running the model could deter some users.

The random forest and logistic regression models would be fairly simple to implement on a different dataset but do require certain statistical features to be extracted. One advantage of the feature extraction and scaling is that the models may be more easily applicable to other temporal resolutions. The logistic regression model would be especially easy to import for use in any programming language as it has a closed form solution and would not require any package dependencies.

In choosing the best model for application in identifying delivery days, the user could choose a model based on whether they want to use raw data or utilize manual feature extraction. Additionally, the user could weigh the high computational cost and greater performance of the convolutional recurrent neural networks against the much faster but slightly less powerful random forest or logistic regression models.

# REFERENCES

Ackermann, N. (2018), 'Har tutorial'. Accessed: 5 July 2020.
**URL:** *https://github.com/ni79ls/har-keras-cnn*

Bagalà, F., Becker, C., Cappello, A., Chiari, L., Aminian, K., Hausdorff, J. M., Zijlstra, W. & Klenk, J. (2012), Evaluation of accelerometer-based fall detection algorithms on real-world falls, *PloS one* **7**(5), e37062.

Baldi, P. & Pollastri, G. (2003), The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem, *Journal of Machine Learning Research* **4**(Sep), 575–602.

Batista, G. E., Keogh, E. J., Tataw, O. M. & De Souza, V. M. (2014), Cid: an efficient complexity-invariant distance for time series, *Data Mining and Knowledge Discovery* **28**(3), 634–669.

Breiman, L. (1996), Bagging predictors, *Machine learning* **24**(2), 123–140.

Breiman, L. (2001), Random forests machine learning. 45: 5–32, *View Article PubMed/NCBI Google Scholar* .

Bridle, J. S. (1990), Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition, *in* 'Neurocomputing', Springer, 227–236.

Brier, G. W. (1950), Verification of forecasts expressed in terms of probability, *Monthly weather review* **78**(1), 1–3.

Chiu, J. P. & Nichols, E. (2016), Named entity recognition with bidirectional lstm-cnns, *Transactions of the Association for Computational Linguistics* **4**, 357–370.

Choi, L., Ward, S. C., Schnelle, J. F. & Buchowski, M. S. (2012), Assessment of wear/nonwear time classification algorithms for triaxial accelerometer, *Medicine and science in sports and exercise* **44**(10), 2009.

Chollet, F. (2015), ‘Keras’. Github Repository.

**URL:** *https://github.com/fchollet/keras*

Christ, M., Braun, N., Neuffer, J. & Kempa-Liehr, A. W. (2018), Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package), *Neurocomputing* **307**, 72–77.

Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M. & Schmidhuber, J. (2011), High-performance neural networks for visual object classification, *arXiv preprint arXiv:1102.0183* .

Crouter, S. E., Churilla, J. R. & Bassett, D. R. (2006), Estimating energy expenditure using accelerometers, *European journal of applied physiology* **98**(6), 601–612.

Diggle, P., Heagerty, P., Liang, K.-Y., Zeger, S. et al. (2002), *Analysis of longitudinal data*, Oxford University Press.

Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K. & Darrell, T. (2015), Long-term recurrent convolutional networks for visual recognition and description, *in* ‘Proceedings of the IEEE conference on computer vision and pattern recognition’, 2625–2634.

Harrell Jr, F. E. (2015), *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*, Springer.

Hastie, T., Tibshirani, R. & Friedman, J. (2009), *The elements of statistical learning: data mining, inference, and prediction*, Springer Science & Business Media.

Hochreiter, S. & Schmidhuber, J. (1997), Long short-term memory, *Neural computation* **9**(8), 1735–1780.

Hu, B., Lu, Z., Li, H. & Chen, Q. (2014), Convolutional neural network architectures for matching natural language sentences, *in* ‘Advances in neural information processing systems’, 2042–2050.

Ignatov, A. (2018), Real-time human activity recognition from accelerometer data using convolutional neural networks, *Applied Soft Computing* **62**, 915–922.

Kim, E., Helal, S. & Cook, D. (2009), Human activity recognition and pattern discovery, *IEEE pervasive computing* **9**(1), 48–53.

Kim, Y. (2014), Convolutional neural networks for sentence classification, *arXiv preprint arXiv:1408.5882* .

Kingma, D. P. & Ba, J. (2014), Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* .

Krizhevsky, A., Sutskever, I. & Hinton, G. (n.d.), Pereira f, burges cjc, bottou l, weinberger kq, editors, *ImageNet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems* **25**, 1097–1105.

Kwapisz, J. R., Weiss, G. M. & Moore, S. A. (2011), Activity recognition using cell phone accelerometers, *ACM SigKDD Explorations Newsletter* **12**(2), 74–82.

Lara, O. D. & Labrador, M. A. (2012), A survey on human activity recognition using wearable sensors, *IEEE communications surveys & tutorials* **15**(3), 1192–1209.

Lipton, Z. C., Berkowitz, J. & Elkan, C. (2015), A critical review of recurrent neural networks for sequence learning, *arXiv preprint arXiv:1506.00019* .

Myles, A. J., Feudale, R. N., Liu, Y., Woody, N. A. & Brown, S. D. (2004), An introduction to decision tree modeling, *Journal of Chemometrics: A Journal of the Chemometrics Society* **18**(6), 275–285.

Nielsen, M. A. (2015), *Neural networks and deep learning*, Vol. 2018, Determination press San Francisco, CA.

Nilsson, N. J. (2005), 'Introduction to machine learning'.

Orr, G. B. & Müller, K.-R. (2003), *Neural networks: tricks of the trade*, Springer.

O'Shea, K. & Nash, R. (2015), An introduction to convolutional neural networks, *arXiv preprint arXiv:1511.08458* .

Pedregosa, F. (2011), Scikit-learn: Machine learning in python, *the Journal of machine Learning research* **12**, 2825–2830.

Qin, C., Schlemper, J., Caballero, J., Price, A. N., Hajnal, J. V. & Rueckert, D. (2018), Convolutional recurrent neural networks for dynamic mr image reconstruction, *IEEE transactions on medical imaging* **38**(1), 280–290.

Saha, S. (2018), 'A comprehensive guide to convolutional neural networks'. Accesed: 5 July 2020.

**URL:** *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*

Sak, H., Senior, A. W. & Beaufays, F. (2014), Long short-term memory recurrent neural network architectures for large scale acoustic modeling.

Schildcrout, J. (2020), 'Longitudinal data: Linear mixed-effects models'. Course Lecture.

Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. et al. (2012), Deep neural networks for acoustic modeling in speech recognition, *IEEE Signal processing magazine* .

Shepard, E. L., Wilson, R. P., Quintana, F., Laich, A. G., Liebsch, N., Albareda, D. A., Halsey, L. G., Gleiss, A., Morgan, D. T., Myers, A. E. et al. (2008), Identification of animal movement patterns using tri-axial accelerometry, *Endangered Species Research* **10**, 47–60.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. et al. (2017), Article mastering the game of go without human knowledge, *Nature Publishing Group* **550**(7676), 354–359.

Song, Y.-Y. & Ying, L. (2015), Decision tree methods: applications for classification and prediction, *Shanghai archives of psychiatry* **27**(2), 130.

Verma, S. (2019), ‘Understanding 1d and 3d convolution neural network’. Accessed: 5 July 2020.
**URL:** *https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610*

Walse, K., Dharaskar, R. & Thakare, V. (2016), Performance evaluation of classifiers on wisdm dataset for human activity recognition, *in* ‘Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies’, 1–7.

Werbos, P. J. (1990), Backpropagation through time: what it does and how to do it, *Proceedings of the IEEE* **78**(10), 1550–1560.

# Appendices

# Appendix A

## WISDM Dataset Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Jun 27 15:35:00 2020

@author: ryan
"""
### Libraries
import numpy as np
import pandas as pd
import random
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, classification_report, confusion_matrix
from scipy import stats
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
import keras
from keras import optimizers
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers import Dense, Dropout, Conv1D, Flatten,
GlobalAveragePooling1D, MaxPooling1D
from keras.utils import np_utils

### Preprocessing
def import_wisdm(file_path):
    names = ['id','activity','timestamp','x-axis','y-axis','z-axis']
    df = pd.read_csv(file_path, header=None, names = names)
    df['z-axis'].replace(regex=True,inplace=True,to_replace=r';', value=r'')
    df['z-axis'] = df['z-axis'].apply(np.float)
    df.dropna(axis=0, how='any', inplace=True)
    return (df)
def label_split_scale(df, split):
    #Encodes labels numerically
    labelencoder = LabelEncoder()
    df['label'] = labelencoder.fit_transform(df['activity'].values.ravel())
    #Split into test/train set
    split_id = round(split * len(df.id.unique()))
    df_test = df[df['id'] > split_id]
    df_train = df[df['id'] <= split_id]
    #Scale
    pd.options.mode.chained_assignment = None
    df_train['x-axis'] = df_train['x-axis'] / df_train['x-axis'].max()
    df_train['y-axis'] = df_train['y-axis'] / df_train['y-axis'].max()
    df_train['z-axis'] = df_train['z-axis'] / df_train['z-axis'].max()
    df_train = df_train.round({'x-axis': 4, 'y-axis': 4, 'z-axis': 4})
    df_test['x-axis'] = df_test['x-axis'] / df_test['x-axis'].max()
    df_test['y-axis'] = df_test['y-axis'] / df_test['y-axis'].max()
    df_test['z-axis'] = df_test['z-axis'] / df_test['z-axis'].max()
    df_test = df_test.round({'x-axis': 4, 'y-axis': 4, 'z-axis': 4})
    return(df_train, df_test)
def segment_label(df, segment_length, step_distance):
    segments = []
    labels = []
    for i in range(0, len(df) - segment_length, step_distance):
        xs = df['x-axis'].values[i: i + segment_length]
        ys = df['y-axis'].values[i: i + segment_length]
        zs = df['z-axis'].values[i: i + segment_length]
        label = stats.mode(df['label'][i: i + segment_length])[0][0]
        segments.append([xs, ys, zs])
        labels.append(label)
    reshaped_segments = np.asarray(segments, dtype= np.float32).reshape(-1, segment_length, 3)
    labels = np.asarray(labels)
    return reshaped_segments, labels
def Two_to_One(data):
    data = data.reshape(data.shape[0], (data.shape[1]*data.shape[2]))
    return(data.astype('float32'))

#Arguements
random.seed(1612)
file_path = '/Users/ryan/Desktop/Python/datasets/WISDM_ar_v1.1_raw.txt'
split = 0.7 #Percent of ID's in training set
segment_length = 80 # Number of measurements within each segment
step_distance = 40 # Number of measurements shifted per step

#Import and Clean data
df = import_wisdm(file_path)
df_train, df_test = label_split_scale(df, split)
x_train, y_train = segment_label(df_train, segment_length, step_distance)
x_test, y_test = segment_label(df_test, segment_length, step_distance)
x_train_2D, x_test_2D = x_train.copy(), x_test.copy()
x_train, x_test = Two_to_One(x_train), Two_to_One(x_test)
y_train = np_utils.to_categorical(y_train, len(df.activity.unique()))
y_test = np_utils.to_categorical(y_test, len(df.activity.unique()))

#Feature Extraction
def cid_ce(x, normalize):
    if not isinstance(x, (np.ndarray, pd.Series)):
        x = np.asarray(x)
    if normalize:
        s = np.std(x)
        if s!=0:
            x = (x - np.mean(x))/s
        else:
            return 0.0

    x = np.diff(x)
    return np.sqrt(np.dot(x, x))
def abs_energy(x):
    if not isinstance(x, (np.ndarray, pd.Series)):
        x = np.asarray(x)
    return np.dot(x, x)
def absolute_sum_of_changes(x):
    return np.sum(np.abs(np.diff(x)))
def kurtosis(x):
    if not isinstance(x, pd.Series):
        x = pd.Series(x)
    return pd.Series.kurtosis(x)
def skewness(x):
    if not isinstance(x, pd.Series):
        x = pd.Series(x)
    return pd.Series.skew(x)
def cid(x, lag=1):

    holds = list()
    for i in range((len(x))-lag):
        hold=x[i+lag]-x[i]
        holds.append(hold)

    return(np.sqrt(np.dot(holds, holds)))
def feature_extract(data):
    data_features = np.empty((data.shape[0],14*3))
    for i in range(data.shape[0]):
        #Features of interest
        avgs = np.apply_along_axis(np.mean, axis = 0, arr=data[i])
        var = np.apply_along_axis(np.var, axis = 0, arr=data[i])
        minimum = np.apply_along_axis(np.min, axis = 0, arr=data[i])
        maximum = np.apply_along_axis(np.max, axis = 0, arr=data[i])
        complexity = np.apply_along_axis(cid_ce, axis = 0, arr=data[i], normalize=True)
        cid2 = np.apply_along_axis(cid, axis = 0, arr=data[i], lag=2)
        cid10 = np.apply_along_axis(cid, axis = 0, arr=data[i], lag=10)
        #cid50 = np.apply_along_axis(cid, axis = 0, arr=data[i], lag=50)
        energy = np.apply_along_axis(abs_energy, axis = 0, arr=data[i])
        sum_change = np.apply_along_axis(absolute_sum_of_changes, axis = 0, arr=data[i])
        kurt = np.apply_along_axis(kurtosis, axis = 0, arr=data[i])
        skew = np.apply_along_axis(skewness, axis = 0, arr=data[i])
        q25 = np.apply_along_axis(np.quantile, axis=0, arr=data[i], q=0.25)
        median = np.apply_along_axis(np.quantile, axis=0, arr=data[i], q=0.5)
        q75 = np.apply_along_axis(np.quantile, axis=0, arr=data[i], q=0.75)
        args = (avgs, var, minimum, maximum, complexity, cid2, cid10, #cid50,
                energy,sum_change,
                kurt, skew, q25, median, q75)
        feature = np.hstack(args)
        data_features[i,:] = feature
    return(data_features)
x_labels = ['x-axis avg', 'y-axis avg', 'z-axis avg',
            'x-axis var', 'y-axis var', 'z-axis var',
            'x-axis min', 'y-axis min', 'z-axis min',
            'x-axis max', 'y-axis max', 'z-axis max',
            'x-axis cidce', 'y-axis cidce', 'z-axis cidce',
            'x-axis cid2', 'y-axis cid2', 'z-axis cid2',
            'x-axis cid10', 'y-axis cid10', 'z-axis cid10',
            #'x-axis cid50', 'y-axis cid50', 'z-axis cid50',
            'x-axis energy', 'y-axis energy', 'z-axis energy',
            'x-axis abs_change', 'y-axis abs_change', 'z-axis abs_change',
            'x-axis kurtosis', 'y-axis kurtosis', 'z-axis kurtosis',
            'x-axis skewness', 'y-axis skewness', 'z-axis skewness',
            'x-axis 25th Quantile', 'y-axis 25th Quantile', 'z-axis 25th Quantile',
            'x-axis 50th Quantile', 'y-axis 50th Quantile', 'z-axis 50th Quantile',
            'x-axis 75th Quantile', 'y-axis 75th Quantile', 'z-axis 75th Quantile']
y_labels = ['Downstairs','Jogging','Sitting','Standing','Upstairs','Walking']

#Extract Features
x_train_feat = feature_extract(x_train_2D)
x_test_feat = feature_extract(x_test_2D)
x_train_feat = pd.DataFrame(x_train_feat, columns = x_labels)
x_test_feat = pd.DataFrame(x_test_feat, columns = x_labels)
y_train_feat = pd.DataFrame(y_train, columns = y_labels)
y_test_feat = pd.DataFrame(y_test, columns = y_labels)

### Modeling
#Random Forest Model
np.random.seed(1612)
model_rf=RandomForestClassifier(n_estimators=500)
model_rf.fit(x_train_feat,y_train_feat)
y_pred=model_rf.predict(x_test_feat)
print("Accuracy:",metrics.accuracy_score(y_test_feat, y_pred))
#MLP
random.seed(1612)
model_mlp_features = Sequential()
model_mlp_features.add(Dense(200, activation='relu',input_shape=(x_train_feat.shape[1],)))
model_mlp_features.add(Dense(100, activation='relu'))
model_mlp_features.add(Dropout(0.5))
model_mlp_features.add(Dense(len(df.activity.unique()), activation='softmax'))
model_mlp_features.compile(loss='categorical_crossentropy',
                optimizer='adam', metrics=['accuracy'])
model_mlp_features.fit(x_train_feat,
              y_train_feat,
              epochs = 10,
              verbose = 0)
model_mlp_features.evaluate(x_test_feat, y_test_feat)
#CNN
random.seed(1612)
model_cnn = Sequential()
model_cnn.add(Conv1D(100, 5, activation='relu', input_shape=(segment_length, 3)))
model_cnn.add(Conv1D(100, 5, activation='relu'))
model_cnn.add(MaxPooling1D(3))
model_cnn.add(Conv1D(160, 5, activation='relu'))
model_cnn.add(Conv1D(160, 5, activation='relu'))
model_cnn.add(GlobalAveragePooling1D())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(len(df.activity.unique()), activation='softmax'))
model_cnn.build()
model_cnn.compile(loss='categorical_crossentropy',
                optimizer='adam', metrics=['accuracy'])
model_cnn.fit(x_train_2D,
              y_train,
              epochs = 10,
              validation_data=(x_test_2D, y_test),
              verbose = 0)
model_cnn.evaluate(x_test_2D, y_test)
#LSTM
random.seed(1612)
model_lstm = Sequential()
model_lstm.add(LSTM(200, input_shape=(x_train_2D.shape[1],x_train_2D.shape[2])))
model_lstm.add(Dropout(0.5))
model_lstm.add(Dense(100, activation='relu'))
model_lstm.add(Dense(len(df.activity.unique()), activation='softmax'))
model_lstm.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model_lstm.fit(x_train_2D,
               y_train,
               epochs=10,
               validation_data=(x_test_2D, y_test),
               verbose=0)
model_lstm.evaluate(x_test_2D, y_test)
#CNNLSTM
random.seed(1612)
model_cnnlstm = Sequential()
model_cnnlstm.add(Conv1D(100, 5, activation='relu', input_shape=(segment_length, 3)))
model_cnnlstm.add(Conv1D(100, 5, activation='relu'))
model_cnnlstm.add(MaxPooling1D(2))
model_cnnlstm.add(LSTM(10))
model_cnnlstm.add(Dense(len(df.activity.unique()), activation='softmax'))
model_cnnlstm.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model_cnnlstm.fit(x_train_2D,
                  y_train,
                  epochs=10,
                  validation_data=(x_test_2D, y_test),
                  verbose=0)
model_cnnlstm.evaluate(x_test_2D, y_test)
#Obtain predictions from test set
rf_prs = model_rf.predict_proba(x_test_feat)
rf_pr = np.empty((rf_prs[0].shape[0],6))
for i in range(0,5):
    pr = rf_prs[i][:,1]
    rf_pr[:,i] = pr
mlp_pr = model_mlp_features.predict(x_test_feat)
cnn_pr = model_cnn.predict(x_test_2D)
lstm_pr = model_lstm.predict(x_test_2D)
cnnlstm_pr = model_cnnlstm.predict(x_test_2D)
```
WISDManalysis.py

# Appendix B

## Accelerometry Dataset Code

```r
library(dplyr)
library(data.table)
library(stringr)
library(PhysicalActivity)
library(lubridate)
library(ggplot2)
library(e1071)
library(tidyverse)
library(tensorflow)
library(abind)
library(keras)
library(lme4)
library(randomForest)
library(Hmisc)
library(rms)
`%notin%` <- Negate(`%in%`)
workingDir <- '/Users/ryan/Desktop/Projects/PA/Data'
verdat <- read.csv(paste0(workingDir, '/delivery-verified-comparison-all2-300.csv'))
orthdat <- get(load(paste0(workingDir, '/orthDat-2017-12-06.RData')))
#### Pre-processing
#Aggregate 30sec trials to 60 sec
ix <- which(grepl("30sec", names(orthdat)))
for(i in ix){
  df <- orthdat[[i]]
  orthdat[[i]] <- dataCollapser(df, TS="TimeStamp",by=60, col=c("axis1", "axis2", "axis3", "vm"))
}
#Drop trials with no label
ix <- intersect(verdat$id, names(orthdat))
orthdat <- orthdat[names(orthdat) %in% ix]
day_creator <- function(df){
  df$day <- format(df$TimeStamp, "%m-%d")
  df$day <- as.numeric(mapvalues(df$day, from=unique(df$day),
                                 to=1:length(unique(df$day))))
  return(df)
}
#Index days
orthdat <- lapply(orthdat, day_creator)
for(i in 1:length(orthdat)){
  name <- names(orthdat)[i]
  df <- orthdat[[i]]
  t <- verdat[verdat$id == name,]
  df$verdel <- NA
  for(j in unique(df$day)){
    df$verdel[df$day == j] <- t$verdel[t$day == j]
  }
  orthdat[[i]] <- df
}
#Repair naming for regex
ixOrth <- which(names(orthdat) == "12498-222-T1 (2016-01-15)60sec.agd")
names(orthdat)[ixOrth] <- "12498-222_T1 (2016-01-15)60sec.agd"
ixVer <- which(verdat$id == "12498-222-T1 (2016-01-15)60sec.agd")
verdat$id <- as.character(verdat$id)
verdat$id[ixVer] <- "12498-222_T1 (2016-01-15)60sec.agd"
#Remove trials with corresponding "rewears"
orthdat$`12498-239_T3 (2016-11-13)60sec.agd` <- NULL
orthdat$`446_T2 (2016-07-24)60sec.agd` <- NULL
orthdat$`487_T2 (2016-10-31)60sec.agd` <- NULL
orthdat$`492_T2 (2016-12-05)60sec.agd` <- NULL
orthdat$`493_T2 (2016-12-05)60sec.agd` <- NULL
#Extract "ID trial"
ID <- str_extract_all(names(orthdat), "^[^_]+(?=_)")
ID   <- unlist(ID,use.names=FALSE)
trial <- str_extract_all(names(orthdat), "(?<=_T)[0-9]+")
trial <- unlist(trial,use.names=FALSE)
ID_trial <- data.frame(ID_trial=as.character(paste(ID, trial)), names(orthdat))
ID_trial$ID_trial <- as.character(ID_trial$ID_trial)
ID_trial$names.orthdat. <- as.character(ID_trial$names.orthdat.)
#Search for duplicated ID_trial
counts <- data.frame(table(ID_trial$ID_trial))
```

```
counts$Freq <- as.numeric(counts$Freq)
dupID_trial <- as.character(counts$Var1[counts$Freq > 1])
#Delete duplicate
for(i in 1:length(dupID_trial)){
  name <- ID_trial[which(ID_trial$ID_trial == dupID_trial[i])[1],]$names.orthdat.
  ix <- which(names(orthdat) == name)
  orthdat <- orthdat[-ix]
}
dropnotin <- function(df){
  ix <- names(df) %in% c("TimeStamp", "axis1", "axis2", "axis3", "vm","day", "verdel")
  return(df[,ix])
}
orthdat <- lapply(orthdat, dropnotin)
firstDayNot1440 <- function(df){
  count <- aggregate(vm ~ day, df, FUN = "length")[1,]
  return(count$vm != 1440)
}
#Remove the trials that were labeled all 0
ix <- NULL
for(i in names(orthdat)){
  x <- aggregate(orthdat[[i]]$verdel, list(orthdat[[i]]$day), mean)$x
  if(all(x ==0)){
    ix <- append(ix, i)
  }
}
orthdat <- orthdat[names(orthdat) %notin% ix]
#Add Proportion of trial days feature
propDayCreator <- function(df){
  df$propDay <- df$day/max(df$day)
  return(df)
}
orthdat <- lapply(orthdat, propDayCreator)

### Processing
#Zeropatch first and last days
zeropatch <- function(df){
  counts <- table(df$day)
  days <- unique(df$day)
  if(counts[1] != 1440){
    tmp <- df[df$day == days[1],]
    tmp$TimeStamp <- force_tz(tmp$TimeStamp, "UTC")
    yearmonthday <- substr(tmp$TimeStamp[1], start = 1, stop = 10)
    timerange <- seq.POSIXt(from=as.POSIXct(paste0(yearmonthday, " 00:00:00 UTC")),
                            to=as.POSIXct(paste0(yearmonthday, " 23:59:00 UTC")),
                            by="min")
    timerange <- force_tz(timerange, "UTC")
    tmp2 <- data.frame(
      TimeStamp = timerange[timerange %notin% tmp$TimeStamp],
      axis1 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      axis2 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      axis3 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      vm = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      day = rep(1, length(sum(timerange %notin% tmp$TimeStamp))),
      verdel = rep(df[df$day == days[1],]$verdel[1],
                   length(sum(timerange %notin% tmp$TimeStamp))),
      propDay = rep(df[df$day ==days[1],]$propDay[1],
                    length(sum(timerange %notin% tmp$TimeStamp)))
    )
    tmp3 <- rbind(tmp2, tmp)
    tmp4 <- df[df$day != days[1],]
    df <- rbind(tmp3, tmp4)
  }
  if(counts[length(counts)] != 1440){
    tmp <- df[df$day == tail(days,1),]
    tmp$TimeStamp <- force_tz(tmp$TimeStamp, "UTC")
    yearmonthday <- substr(tmp$TimeStamp[1], start = 1, stop = 10)
    timerange <- seq.POSIXt(from=as.POSIXct(paste0(yearmonthday, " 00:00:00 UTC")),
                            to=as.POSIXct(paste0(yearmonthday, " 23:59:00 UTC")),
                            by="min")
    timerange <- force_tz(timerange, "UTC")
    tmp2 <- data.frame(
      TimeStamp = timerange[timerange %notin% tmp$TimeStamp],
      axis1 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      axis2 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      axis3 = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      vm = rep(0, length(sum(timerange %notin% tmp$TimeStamp))),
      day = rep(tail(days,1), length(sum(timerange %notin% tmp$TimeStamp))),
      verdel = rep(df[df$day == tail(days,1),]$verdel[1],
                   length(sum(timerange %notin% tmp$TimeStamp))),
      propDay = rep(df[df$day ==days[1],]$propDay[1],
                        length(sum(timerange %notin% tmp$TimeStamp)))
      )
      tmp3 <- rbind(tmp, tmp2)
      tmp4 <- df[df$day != tail(days,1),]
      df <- rbind(tmp4, tmp3)
    }
}
#Remove days with less than min count on axis1
removeLow <- function(df, minLow=0, minTime=0){
  ix1 <- aggregate(df$vm, list(df$day), sum)[aggregate(df$vm, list(df$day), sum)$x < minLow,1]
  ix2 <- aggregate(df$vm != 0, list(df$day), sum)[aggregate(df$vm !=0, list(df$day), sum)$x <
      minTime,1]
  ix <- append(ix1, ix2)
  df <- df[df$day %notin% ix,]
  return(df)
}
#Add fake timestamp and days for plots
fakeTimeStamp <- function(df){
  df$FakeTimeStamp <- seq.POSIXt(from= as.POSIXct(strptime("2000-01-01 00:00:00",
                                                           "%Y-%m-%d %H:%M:%S"),
                                               tz="UTC"),
                                 length.out=1440*length(unique(df$day)),
                                 by="min")

  df$FakeDays <- as.numeric(mapvalues(df$day, from=unique(df$day),
                                      to=1:length(unique(df$day))))
  return(df)
}
#Collapse list to 2D df
twoD <- function(dat){
  orthdat2d <- rbindlist(dat, idcol=TRUE, fill=TRUE)
  ID <- str_extract_all(orthdat2d$.id, "^[^_]+(?=_)")
  ID   <- unlist(ID,use.names=FALSE)
  orthdat2d <- cbind(orthdat2d, ID)
  trial <- str_extract_all(orthdat2d$.id, "(?<=_T)[0-9]+")
  trial <- unlist(trial,use.names=FALSE)
  orthdat2d <- cbind(orthdat2d, trial)
  orthdat2d$ID_trial <- paste(orthdat2d$ID, orthdat2d$trial, sep="_")
  orthdat2d$ID_trial_day <-paste(orthdat2d$ID_trial, orthdat2d$day, sep="_")
  orthdat2d <- subset(orthdat2d, select=-c(trial))
  return(orthdat2d)
}
lowCountWearDrop <- function(df, minWearCount=0){
  df1 <- df[df$verdel == 0,]
  ix <- aggregate(df1$vm !=0, list(df1$day), sum)[aggregate(df1$vm !=0, list(df1$day), sum)$x <
      minWearCount,1]
  df <- df[df$day %notin% ix,]
  return (df)
}
preprocess <- function(dat, zeropatch = TRUE, removeLow = TRUE, lowCountWear=TRUE,
                       minLow=5000, minTime = 0, minWearCount=0, twoD){
  if(zeropatch == TRUE){dat <- lapply(dat, zeropatch)}
  if(removeLow == TRUE){dat <- lapply(dat, removeLow,
                                      minLow=minLow, minTime=minTime)}
  if(zeropatch == FALSE & removeLow == FALSE){
    dat <- lapply(dat, function(dat){dat$FakeTimeStamp <- dat$TimeStamp; return(dat)})
    dat <- lapply(dat, function(dat){dat$FakeDays <- dat$day; return(dat)})
  } else {dat <- lapply(dat, fakeTimeStamp)}

  if(lowCountWear==TRUE){
    dat <- lapply(dat, lowCountWearDrop, minWearCount=minWearCount)
  }
  if(twoD == TRUE){dat <- twoD(dat)}
  return(dat)
}
ori <- preprocess(orthdat, zeropatch = FALSE, removeLow = FALSE, lowCountWear = FALSE,
                  twoD = TRUE)
pro <- preprocess(orthdat, zeropatch = TRUE, removeLow = TRUE, lowCountWear = TRUE,
                  minLow = 5000, minTime=10, minWearCount=60*2, twoD = TRUE)
oriPatched <- preprocess(orthdat, zeropatch = TRUE, removeLow = FALSE, lowCountWear = FALSE,
                         twoD = TRUE)
#Manually remove mislabeled days identified from plotting###
mis <-c("102_3_10", "12498-138_2_17", "12498-138_2_18" ," 12498-138_2_19",
  "12498-138_2_20" , "12498-153_3_14", "12498-243_2_11", "12498-24_3_13",
  "126_1_8", "181_1_9", "318_3_14" ," 318_3_15", "322_3_11", "383_1_12",
  "387_2_33", "387_3_31", "387_3_32", "429_3_10", "42_2_11", "459_1_9",
  "12498-188_1_8", "12498-188_1_9", "12498-188_1_10", "380_1_7",
  "380_1_8","08_3_17","121_1_7","12498-110_2_16","12498-112_3_13", "12498-198_2_10",
  "12498-47_2_13", "12498-56_1_9", "222_3_10","479_1_8", "03_2_1",
  "393_1_21")
ori <- ori[ori$ID_trial_day %notin% mis,]
```

```r
pro <- pro[pro$ID_trial_day %notin% mis,]
oriPatched <- oriPatched[oriPatched$ID_trial_day %notin% mis,]
#Split into test and train set by ID
ttSplit <- function(df, percentIDTrain=0.7){
  IDs <- unique(df$ID)
  trainID <- sample(IDs, size=round(percentIDTrain*length(IDs)))
  testID <- IDs[IDs %notin% trainID]
  return(list(train=trainID, test=testID))
}
#Extract Features
featExtract <- function(df){
  dt <- as.data.table(df)
  dat <- split(dt, dt$ID_trial_day)
  feats <- data.frame(
    ID = sapply(dat, function(df){df$ID[1]}),
    ID_trial = sapply(dat, function(df){df$ID_trial[1]}),
    day = sapply(dat, function(df){df$day[1]}),
    ID_trial_day = sapply(dat, function(df){df$ID_trial_day[1]}),
    verdel = sapply(dat, function(df){mean(df$verdel)}),
    mean = sapply(dat, function(df){mean(df$vm)}),
    variance = sapply(dat, function(df){var(df$vm)}),
    q95 = sapply(dat, function(df){unname(quantile(df$vm, 0.95))}),
    max = sapply(dat, function(df){max(df$vm)}),
    absChange = sapply(dat, function(df){sum(abs(diff(df$vm)))}),
    absEnergy = sapply(dat, function(df){sum((df$vm)^2)}),
    propDay = sapply(dat, function(df){df$propDay[1]}),
    skewness = sapply(dat, function(df){e1071::skewness(df$vm)}),
    kurtosis = sapply(dat, function(df){e1071::kurtosis(df$vm)})
  )
  #Nan occurs due to dividing by 0 if vector of vm is all 0. Replace Nan with 0.
  feats$skewness[is.nan(feats$skewness)] <- 0
  feats$kurtosis[is.nan(feats$kurtosis)] <- 0
  feats$verdel.factor <- fct_rev(as.factor(ifelse(feats$verdel == 0, "Wear", "Delivery")))
  return(feats)
}
#Scale Features
scaleFeatures <- function(df){
  dfScale <- scale(df[,c("mean","variance", "max", "absChange",
                        "absEnergy","q95","skewness","kurtosis")])
  dfScale <- cbind(as.data.frame(dfScale), df$propDay, df$ID, df$ID_trial, df$verdel.factor, df$
      verdel)
  colnames(dfScale) <- c("mean","variance", "max", "absChange", "absEnergy","q95",
                        "skewness", "kurtosis",  "propDay","ID","ID_trial", "verdel.factor", "
                            verdel")
  return(dfScale)
}
#Format data for neural network
nnFormat <- function(dat, make4d=FALSE){
  dt <- lapply(X=dat, FUN=as.data.table)
  dt <- lapply(X=dt, FUN= function(dt){
    split(dt[,c("axis1","axis2","axis3","verdel")], dt$ID_trial_day)})
  dt_Y <- lapply(X=dt, FUN= function(df){
    as.matrix(sapply(df, function(df){df$verdel[1]}))})
  dt_X <- lapply(X=dt, FUN= function(dat){
    lapply(dat, function(df){as.matrix(df[,-4])})})
  dt_X <- lapply(X=dt_X, FUN= function(dat){abind(dat, along=3)})
  dt_X <- lapply(X=dt_X, FUN= function(dt){aperm(dt, c(3,1,2))})
  if(make4d==TRUE){
    dt_X <- lapply(X=dt_X, FUN= function(dt){
      array(dt, dim=append(dim(dt),1))})
  }
  return(list(train = list(X=dt_X$train, Y=dt_Y$train),
              test = list(X=dt_X$test, Y=dt_Y$test)))
}
runModels <- function(CrossValReps = 1,scaleRaw=TRUE, epochs=10, data=pro){

  if(CrossValReps > 1 & importNeuralNets==TRUE){
    warning("Cannot import single model and cross validate \n")
    stop()
  }

  results <- list()
  for(i in 1:CrossValReps){
    ### Train/Test Split ###
    IDs <- ttSplit(data, percentIDTrain=0.7)
    dat <- list(train = data[data$ID %in% IDs$train,],
                test = data[data$ID %in% IDs$test,])
    #Feature Extraction
    feats <- list(train = featExtract(dat$train),
                  test = featExtract(dat$test))
    featsScale <- list(train = scaleFeatures(feats$train),
                       test = scaleFeatures(feats$test))
    #Scale raw data
    if(scaleRaw==TRUE){
      dat <- lapply(dat, FUN=function(df){
        df$axis1 <- scale(df$axis1)
        df$axis2 <- scale(df$axis2)
        df$axis3 <- scale(df$axis3)
        return(df)
      })
    }
    #MLP Formatting
    mlpData <- list(
      train = list(X=as.matrix(featsScale$train[,c("mean", "variance", "q95", "max","absChange",
                                                    "absEnergy", "propDay", "skewness", "kurtosis"
                                                        )]),
                   Y=as.matrix(featsScale$train$verdel)),
      test = list(X=as.matrix(featsScale$test[,c("mean", "variance", "q95", "max","absChange",
                                                  "absEnergy", "propDay", "skewness", "kurtosis")
                                                      ]),
                  Y=as.matrix(featsScale$test$verdel))
    )
    #CNN/naiveRNN Formatting
    cnn1dData <- nnFormat(dat=dat, make4d=FALSE)
    cnn2dData <- nnFormat(dat=dat, make4d=TRUE)
    ### Modeling ###
    #Random Forest
    model_RF <- randomForest(verdel.factor ~ mean+variance+max+absChange+absEnergy+q95+propDay+
        skewness+kurtosis,
                            data=featsScale$train, ntree=500, keep.forest=TRUE,
                            xtest = featsScale$test[,c("mean","variance", "max",
                                                       "absChange", "absEnergy", "q95",
                                                       "propDay","skewness","kurtosis")],
                            ytest = featsScale$test$verdel.factor)
    #Logistic Regression
    dd <<- datadist(featsScale$train)
    options(datadist = "dd")
    model_GLM <- lrm(verdel.factor ~ rcs(mean,3) + rcs(variance,3) + rcs(max,3) +
                     rcs(absChange,3) + rcs(q95,3) + rcs(propDay,3) +rcs(skewness,3)+rcs(
                         kurtosis,3),
                   data=featsScale$train , x=TRUE, y=TRUE)
    #Logistics Mixed Effects Regression
    model_GLMM <- tryCatch(glmer(verdel.factor ~ rcs(mean,3) + rcs(variance,3) + rcs(max,3) +
                                   rcs(absChange,3) + rcs(absEnergy,3) + rcs(q95,3) + rcs(propDay
                                       ,3) +
                                   rcs(skewness,3) + rcs(kurtosis,3) + (1 | ID),
                                 data = featsScale$train, family = binomial,
                                 control = glmerControl(optimizer = "bobyqa"), nAGQ = 10), error=
                                     function(e){})
      #MLP
      model_MLP <- keras_model_sequential() %>%
        layer_dense(units = 200, activation = 'relu', input_shape = c(9)) %>%
        layer_dropout(rate = 0.5) %>%
        layer_dense(units = 1, activation = "sigmoid")
      model_MLP %>% compile(optimizer = 'adam',
                            loss='binary_crossentropy',
                            metrics = 'accuracy')
      model_MLP %>% fit(x = mlpData$train$X, y = mlpData$train$Y,
                        epochs = epochs,verbose = 0)
      #1-D CNN
      model_CNN1D <- keras_model_sequential() %>%
        layer_conv_1d(filters = 200, kernel_size = 10, strides=1,
                      activation = "relu", input_shape = c(1440,3)) %>%
        layer_max_pooling_1d(pool_size = 4) %>%
        layer_conv_1d(filters = 64, kernel_size = 3, activation = "relu") %>%
        layer_max_pooling_1d(pool_size = 4) %>%
        layer_flatten() %>%
        layer_dense(units = 64, activation = "relu") %>%
        layer_dense(units = 1, activation = "sigmoid")
      model_CNN1D %>% compile(
        optimizer = 'adam',
        loss='binary_crossentropy',
        metrics = 'accuracy'
      )
      model_CNN1D %>%
        fit(x = cnn1dData$train$X, y = cnn1dData$train$Y,
            epochs = epochs,verbose = 0)

      #2-D CNN
      model_CNN2D <- keras_model_sequential() %>%
        layer_conv_2d(filters = 100, kernel_size = c(8,2), strides=1,
                      activation = "relu", input_shape = c(1440,3,1)) %>%
        layer_max_pooling_2d(pool_size = c(4,2)) %>%
        layer_conv_2d(filters = 100, kernel_size = c(4,1), strides=2,
                      activation = "relu") %>%
        layer_max_pooling_2d(pool_size = c(2,1)) %>%
        layer_flatten() %>%
        layer_dense(units = 100, activation = "relu") %>%
        layer_dense(units = 1, activation = "sigmoid")

      model_CNN2D %>% compile(
        optimizer = 'adam',
        loss='binary_crossentropy',
        metrics = 'accuracy')

      model_CNN2D %>%
        fit(x = cnn2dData$train$X, y = cnn2dData$train$Y,
            epochs = epochs,verbose = 0)

      #RNN
      model_LSTM <- keras_model_sequential() %>%
        layer_lstm(units=30, input_shape = c(1440,3), return_sequences=F) %>%
        layer_dropout(rate = 0.4) %>%
        layer_dense(units = 100, activation = 'relu') %>%
        layer_dropout(rate = 0.3) %>%
        layer_dense(units = 1, activation = "sigmoid")

      model_LSTM %>% compile(
        optimizer = 'adam',
        loss='binary_crossentropy',
        metrics = 'accuracy'
      )

      model_LSTM %>%
        fit(x = cnn1dData$train$X, y = cnn1dData$train$Y, epochs = epochs, verbose = 0)

      #CRNN
      model_CRNN <- keras_model_sequential() %>%
        layer_conv_1d(filters = 200, kernel_size = 10, strides=1,
                      activation = "relu", input_shape = c(1440,3)) %>%
        layer_max_pooling_1d(pool_size = 4) %>%
        layer_conv_1d(filters = 64, kernel_size = 3, activation = "relu") %>%
        layer_max_pooling_1d(pool_size = 4) %>%
        layer_lstm(units=30 , return_sequences=F) %>%
        layer_dropout(rate = 0.4) %>%
        layer_dense(units = 50, activation = 'relu') %>%
        layer_dropout(rate = 0.3) %>%
        layer_dense(units = 1, activation = "sigmoid")

      model_CRNN %>% compile(optimizer = 'adam',loss='binary_crossentropy',metrics = 'accuracy')

      model_CRNN %>%
        fit(x = cnn1dData$train$X, y = cnn1dData$train$Y,epochs = epochs,verbose = 0)

    }



#Model Predictions
res <- data.frame(ID_trial = featsScale$test$ID_trial,
                      verdel = featsScale$test$verdel,
                      rf = as.vector(predict(model_RF, newdata=featsScale$test, type="prob")[,2])
                          ,
                      glm = predict(model_GLM, newdata=featsScale$test, type="fitted"),
                      glmm = predict(model_GLMM, newdata=featsScale$test,
                                     allow.new.levels=TRUE, type="response"),
                      mlp = predict(model_MLP, mlpData$test$X),
                      cnn1d = predict(model_CNN1D, cnn1dData$test$X),
                      cnn2d = predict(model_CNN2D, cnn2dData$test$X),
                      rnn = predict(model_LSTM, cnn1dData$test$X),
                      crnn = predict(model_CRNN, cnn1dData$test$X))
  print(paste0("Cross Validations Completed: ", i))
  results[[i]] <- res
  }
  return(results)
}
#Calculates mean and sd for brier score, PPV, sensitivity, and F1 score for each model
getCvResults <- function(ls){
  res <- lapply(ls, FUN = function(df){
    res <- data.frame(matrix(NA, nrow=4, ncol=ncol(df)-2))
    colnames(res) <- colnames(df[,3:ncol(df)])
    rownames(res) <- c("PPV", "Sensitivity", "F1 Score", "Brier Score")
    for(j in 3:ncol(df)){
      pred <- ifelse(df[,j] < 0.5, 0, 1)
      ppv <- sum(pred == 1 & df$verdel == 1)/
        (sum(df$verdel == 1) + sum(pred == 1 & df$verdel == 0))
      sens <- sum(pred == 1 & df$verdel == 1)/sum(df$verdel == 1)
      f1 <- (2*ppv*sens)/(ppv+sens)
      brier <- (1/nrow(df))*(sum((pred-df$verdel)^2))
      res[,j-2] <- c(ppv, sens, f1, brier)
    }
    return(res)}
  )
  arr <- array( unlist(res), c(nrow(res[[1]]),ncol(res[[1]]),length(res)))
  means <- apply(arr, 1:2, FUN=mean)
  sds <- apply(arr, 1:2, FUN=sd)
  finalRes <- data.frame(matrix(NA, nrow=nrow(means), ncol(means)))
  colnames(finalRes) <- colnames(res[[1]])
  rownames(finalRes) <- rownames(res[[1]])
  for(i in 1:nrow(finalRes)){
    for(j in 1:ncol(finalRes)){
      finalRes[i,j] <- paste0(round(means[i,j],3), " (", round(sds[i,j],3), ") ")
    }
  }
  return(finalRes)
}
###Modeling and crossvalidation
set.seed(1612)
cvResProScaled <- runModels(dat=pro, CrossValReps = 5, importNeuralNets=F, epochs=10,scaleRaw=T)
set.seed(1612)
cvResOriScaled <- runModels(dat=oriPatched, CrossValReps = 5, importNeuralNets=F, epochs=10,
    scaleRaw=T)
```
PAProcessingModeling.R