Bovine Event Detection: Analysis of Bovine Data and Cycles

by

Emily Medema

B.Sc. Hons., The University of British Columbia, 2022

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

B.SC. COMPUTER SCIENCE HONOURS

in

Faculty of Science

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2022

Abstract

Agriculture is a vast industry that has the potential for fascinating data analysis and research. However, a lot of data collected by farms and researchers goes to waste as it is inaccessible or focused on monetary gain rather than the interests of researchers or farmers. In an age of big data and data-driven development, it is unfortunate that a majority of the data collected within agriculture is stored within local or proprietary files and thus a daunting analysis task. Ensuring this data is accessible and performing an analysis that is understandable would be a major step forward for the agricultural academia and industry environment. Filling this gap of open-sourced data analysis will allow for a more customized analysis to be performed on bovine data, widening the potential research avenues utilizing sensor data. We have transferred the data provided from CSV files to a DBMS hosted on the cloud, in this case, MySQL on a Digital Ocean Ubuntu Server through a generic, customizable script that reads and uploads all of the data. We then utilized Machine Learning models, specifically k-means, K Nearest Neighbours (KNN), and Isolation Forests, to detect outliers and inform our exploration into event detection. Through our transfer of data and analysis, we can see that having accessible data allows for the development of new detection models and a new understanding of the data itself.

Table of Contents

List of Tables

List of Figures

Listings

Acknowledgements

Special thanks to Dr. Ramon Lawrence for being an exceptional supervisor. Additionally, I'd like to thank Dr. Ronaldo Cerri and his team for the use of their data in my thesis.

Dedication

Thanks to my family for supporting me through the stress of this last year and for pretending to have interest whenever I talked technical details.

Chapter 1

Introduction

In the age of big data and despite the progress within academia and industry in terms of data and cloud storage, it is an unfortunate fact that incredibly useful data remains in text files stored on personal computers. Analyzing this data manually or with little knowledge can be a daunting task. Currently within the agricultural community, there is a lack of analysis of data that could help farmers. While there is development of event detection models happening, the analysis and algorithms are typically considered proprietary and thus not shared with the farmers by agricultural companies. Providing more research into event detection and data analysis could allow farmers and researchers to gain more insight into agriculture and in particular, cows. In order for the agricultural sector to progress in terms of data and its analysis, we must move from the inefficient and volatile storage of data within Excel or text files. A generalized script requiring little customization and technical knowledge to run will allow for an easy transfer of data from a file-based system to a database system. After which, various unique methods and models can be run on the data. In this instance, we are focusing on event detection using the sensor data from cows.

1.1 Motivation

Agriculture is a vast industry that has the potential for fascinating data analysis and research. However, a lot of data collected by farms and researchers goes to waste, languishing in spreadsheets or behind paywalls. In fact, approximately 75 percent of herds within the United States and 21 percent in Canada have implemented Automated Activity Monitor (AAM) systems, specifically for *estrous* detection [BMS+17]. These smaller companies, farmers, and researchers have to rely upon the systems of these companies [CBM+21] for detection, analysis, and data collection. These systems and their models are proprietary which hinders the research and possible use of their technologies. Not to mention that these developments are focused on profitability and not furthering research resulting in limited use and potential in terms of new research. For example, there is an overwhelming

focus on *estrous* detection, however, it is unclear what measurements and thresholds are used for this detection within the AAM systems [CBM+21].

The data we are using was provided by Dr. Ronaldo Cerri and his team at UBC Vancouver Dairy. This data is sourced from a bovine farm and it consists of various measurements of bovine over a period of time. Currently, it is all stored in Excel files based on the date, source, and type of data. Once the data has been uploaded and is accessible, we will then perform data cleaning and analysis. After which our goal is to develop a model and a script to detect events within the data. In order to transfer the data, we will create a generic, customizable script to read and upload all the data within these spreadsheets to a DBMS hosted on the cloud, in this case MySQL on a Digital Ocean Ubuntu Server.

As mentioned, our major goal is developing techniques for event detection. In the bovine industry, there are certain events and cycles that require measurement, knowledge, or intervention of some kind. The ability to detect and predict when a cow will enter an *estrous* cycle is one such event as intervening would increase opportunities for reproduction and allow for better care of the animals [BPKC18]. *Estrus* is a state within the *estrous* cycle of a cow that precedes ovulation and only lasts between 6 and 20 hours [BPKC18]. As *estrus* is closely associated with breeding and the viability of impregnation, detecting *estrus* is incredibly important for the reproduction management of a dairy farm since it indicates the most opportune time for artificial insemination [CBM+21].

Luckily, *estrus* results in a marked change of the cow's behaviour, specifically in terms of restlessness. During this period of the *estrous* cycle, a cow will start to mount and let themselves be mounted by other cows as well as become quite restless which can be detected by the activity monitor sensors [CBM+21]. The sensors only measure the hourly baseline levels of activity from the cows. Traditionally, this is done using a rolling mean calculation over several days (so that each hour of the day has a mean number). Given a threshold, if there is a 80-100 percent relative increase or a 3 standard deviation relative change in activity (this can vary), it triggers an *estrus* alert. The hour block at the time of the alert is considered the beginning of estrus. The end of estrus is considered the hour block that observes the activity measurement dropping below the threshold level.

1.2 Contribution

The contribution of this thesis is as follows:

- the development of a generic script to automatically upload data from Excel ('xlsx') or CSV files to a MySQL database;
- data cleaning and analysis;
- model development for the detection of events;
- the translation of existing R code to Python for *estrus* detection.

Chapter 2

Background

2.1 Database Systems vs. File-base Systems

File-based systems have significant drawbacks when compared to database systems such as size constraints, inefficiency, and insufficient protections. However, a file-based system has a low barrier of entry for any person who has a base knowledge of how to work in Excel or with a spreadsheet. While being user-friendly has its advantages, you can quickly run into issues when using file-based systems.

Database systems, while having a steeper initial learning curve, are better in the long run for the data. Databases allow for the use of a query language, resulting in easy searches of the data. Similarly, databases are efficient [Law21]. Querying and storing a large data-set, without having to read every file and data item is a better use of time and processing power. Additionally, databases are easily secured and able to manage large amounts of data without having to worry about the storage capacity or safety of your local computer [Law21]. Databases are also persistent [Law21]. This means that rather than possibly losing your data when your computer shuts off or crashes, data will still be safely stored within the database.

In addition to the clear advantages of storing data in a database rather than a file-based system, the potential for future use of the data when stored in a database is much higher. For example, it is much better to create a dashboard or run analysis on data that is within a database as that data is independent and abstracted. This allows for the data within the database to change without affecting the programs or analysis relying upon it [Law21].

Data residing in files requires a data engineering process to load, clean, and update data into a database. Given that significant amounts of sensor data, specifically the bovine data studied in this work, is stored in files, it is clear that creating a script that will allow easy transitions between file-based systems and databases would be beneficial.

Figure 2.1: AfiCollar

2.2 Sensor Data

Along with the advent and development of the Internet of Things (IoT), came sensors and the corresponding data. The primary purposes of IoT sensor networks are to sense the critical information from the external physical environment, sample the internal system signals, and obtain meaningful information from sensor data to perform decision-making [KKG+20]. These IoT sensor networks can be found everywhere. As we become more and more connected, the more data and sensors we collect and maintain. However, for a long time this data was poorly used as sensors record very large amounts of data and "principled methods for efficient sensor data processing" were not widely applied [Agg13]. Nonetheless, despite the noisiness of the data, genuine insights can be made and processes aided and automated from this data.

Despite how helpful sensors are for a wide variety of real-life problems, there are a number of challenges. Some of the main issues are a deluge of unclean sensor data and a high resource-consumption cost [KKG+20]. In order to overcome these issues, we must know how to use processing techniques such as data "denoising", data outlier detection, missing data imputation and data aggregation [KKG+20]. Additionally, it is necessary to use data fusion methods such as direct fusion, associated feature extraction, and identity declaration data fusion [KKG+20].

There are several AAM systems available and while there are still improvements that could be made, using a combination of these sensors results in acceptable detection and transmission of data [CBM+21]. In terms of the data we are using, we are working with multiple sensors. This can reduce the error rate, false negatives, and increase the probability of *estrus* detection. Specifically, we have two main sensors: a collar and a pedometer,

Figure 2.2: AfiAct

both manufactured by Afimilk[1], an Israeli company. The first sensor (AfiAct II) measures physical activity (steps) and lying time and is located on the cow's foot as seen in Figure 2.2. A study in the 2005 from the Netherlands found that pedometers were able to detect between 51 and 87 percent of all *estrous* periods [CBM$^+$21]. The second sensor (AfiCollar) is a collar (see Figure 2.1) that measures physical activity (acceleration), feeding time (most likely based on head tilt), and rumination time. It has been shown that using a neck-mounted device has detected 71 percent of the preovulatory phases but missed 13 percent of the recorded ovulations, 72 percent of the preovulatory follicular phases with only 32 percent false positives, and 83 to 85 percent of cows in *estrus* [CBM$^+$21]. Over the course of these studies, it has become clear that the future of bovine data analysis and event detection, specifically *estrus*, lies within the use of AAM systems.

[1] https://www.afimilk.com/cow-monitoring

Chapter 3

Architecture

3.1 Technology Stack

This section overviews the technology used for the project.

3.1.1 Python

Python is the coding language chosen for the script, data analysis, and model development. Python is a versatile language, particularly suited for data analysis and automation. It has a host of well-documented libraries and features that are useful for all aspects of the project including `pandas`, `mysqlclient`, and `SciKit`. Both `pip` and `conda` are used for library management.

3.1.2 MySQL

The database selected is MySQL. This was due to the formatting of the data corresponding well to a relational model and it being a Database Management System (DBMS) that was familiar.

3.1.3 Digital Ocean

Digital Ocean hosts the database. Digital Ocean was selected as the web host as it allows freedom to host the data in anyway we chose and was a reasonable size and price for what is needed. We have chosen to create a Ubuntu Virtual Machine to host the data.

3.1.4 Docker

Docker was chosen as it allows quick set-up and tear down as well as multiple services at the same time. Through docker we are able to easily set-up MySQL, PHP, and a Web-server to host and see our data on the Ubuntu Virtual Machine. Docker will also allow for an easy switch to a different DBMS if required.

Chapter 4

Analysis: Script Development

The first key task was to develop a Python script to read and upload spreadsheets to a MySQL DBMS.

4.1 Set-up and Requirements

In terms of set-up and requirements, we must have the correct file structure, the requisite Python libraries installed, and a MySQL database set-up to receive the data.

4.1.1 File Structure

There are two specific folders needed for the script to work correctly: a 'datasets' folder and a 'processed' folder. These folders must be located correctly in relation to the script, otherwise there will be an error. Specifically, the 'datasets' folder must be located in the same folder as the script (*spreadsheet_conversion.py*) and the 'processed' folder located within the 'datasets' folder. In addition to the 'processed' folder, the 'datasets' folder will also hold all the Excel or CSV files we will be uploading.

4.1.2 Libraries

In order for the script to work, there are specific Python libraries that must be installed. To ensure ease of use, we have compiled most of the libraries into the 'requirements.txt' file. You can then simply run `pip install requirements.txt` within the terminal and all the libraries listed will be installed. Any additional libraries needed, specifically the `mysqlclient` and `pandas-profiling`, can be installed following the directions on the GitHub repository. In the case of the former, it will be installed via the terminal by running `sudo apt install libmysqlclient-dev`. For the latter, we will

use terminal as well with the command `conda install -c conda-forge pandas -profiling`. A prerequisite of note for `pandas-profiling` is that `conda` is also needed for this install.

`mysqlclient` is needed in order to connect the Python script to a MySQL database. `PyMySQL` is also used to connect to the MySQL database. The former is used to upload the data and the latter is used to retrieve the data for analysis. `pandas` and `numpy` are used to clean, handle, and analyze the data. `matplotlib` and `seaborn` are used to graph and depict the data. `pandas-profiling` is used to get a general understanding of the data and the relationships within it. Lastly, `pyod` and `scikit-learn` are used for the machine learning model development.

4.1.3 MySQL Database

For the script to work, a working MySQL database is required with the correct tables for the data. For the bovine data, this set-up will have four different tables (derived from the titles of the spreadsheets, the sensors, and the types of data). Specifically, there is "AfiCollar_Motion_Hourly", "AfiAct2_Steps_and_Activity_15_minutes", "AfiAct2_Rest_Hourly", and "AfiCollar_Rumination_and_Eating_Hourly". See Table 5.1 for more details. These tables need to be created and set-up in order to receive the data. This means that the attributes and keys need to be correctly typed and named in order for the data insertion to work.

4.2 Spreadsheet Conversion Script

The spreadsheet conversion (ie. reading of Excel or CSV files and upload of the data to the database) is done through one Python script, the *spreadsheet_conversion.py* file.

The Spreadsheet Conversion Script can be split into five main sections:

4.2.1 Connection to Database

The first section is connecting to the database, which uses the `mysqlclient` library. This code (see Code 4.1) connects to the database through the connection information specified in the *custom.py* file.

```
def __init__(self):
    self.db = MySQLdb.connect(host=hostname,
                              db=dbname,
                              user=username,
                              passwd=pword)
    self.c = self.db.cursor()
```

Listing 4.1: Connection Code within Constructor

The first section of code is the constructor for the `SpreadsheetConversion` class. This constructor connects to the database with the methods provided by `mysqlclient`. In Code 1, we then can customize the connection information (database name, user, and password are redacted for confidentiality) and the information regarding the tables. It is vitally important that the information in `table_info` is correct as this is how we determine which table to insert the data into.

4.2.2 Reading a Spreadsheet

After a database connection is established, the code reads the first spreadsheet found in the 'datasets' folder (see Code 4.2).

```
def read_files(self, file_name, file_type):
    if file_type == "xlsx":
        df = pd.read_excel(file_dir + "/" + file_name)
    elif file_type == "csv":
        df = pd.concat((chunk for chunk in pd.read_csv(file_dir
    + "/" + file_name,chunksize=5000)))
    else:
        print(file_name, file_type)
        print("File type not supported")
        df = pd.DataFrame()
    return df
```

Listing 4.2: Read Files Method

Based on the file_type, a different read method is used. CSV files are read chunk by chunk, with each chunk being 5000 rows. Whereas Excel files use the `pd.read_excel` method. By the end of this method, there is a dataframe (stored in the `df` variable) that contains all the data that was within the file.

4.2.3 Upload of Data

The next step is to upload the data from within the dataframe to the corresponding table within the database. The upload code is mainly Code 4.3, with the helper methods of Code 2 and Code 3:

```
def send_to_sql(self, data):
    """
    This function sends the data from dataframe to mysql
    """
    headers = list(data.columns.values)
    table = self.compare_headers(headers)

    for i in range(len(data)):
        row = data.iloc[i].tolist()
        sql = self.prepare_insert(table, headers, row)
        row = [item for item in row
            if not (pd.isna(item))==True]
        self.c.execute(sql, row)
        self.db.commit()
```

Listing 4.3: Send to SQL Method

Before uploading the data, the code determines to which table to upload the data. The `compare_headers` helper method does this by comparing the headers from the spreadsheet to the column names of the tables and returning the most similar match. In some rare instances, a header from the spreadsheet may not be suitable as a column name in MySQL. In this case, we utilize the `rename_headers` method and the `banned_headers` customization field to rename the headers to the predetermined suitable name (ie. the original name appended with "_Collected") and then compare and return.

Now that the data is stored in a dataframe and the code has determined which table to upload to, the next step is to prepare to insert into the table using the `prepare_insert` method in Code 4.4. This method uses string formatting to create a prepared statement that will insert the data into the correct table and columns. Once the SQL insert statement is created and returned, an insert is performed into the database (provided there are no NA values that may cause issues).

```
def prepare_insert(self, tablename, df_head, row):
    query = """INSERT INTO {0} ({1}) VALUES ({2});"""
    columns = ""
    placeholders = ""
    index = 0
    for col in df_head:
        if pd.isna(row[index]):
            index += 1
        elif index == 0:
            columns = columns + "`" + col + "`"
            placeholders = placeholders + "%s"
            index += 1
        else:
            columns = columns + ",`" + col + "`"
            placeholders = placeholders + ", %s"
            index += 1
    return query.format(tablename, columns, placeholders)
```

Listing 4.4: Prepare Insert Method

4.2.4 File Restructuring

After the file has been read and uploaded to the database, it is moved from the 'datasets' folder to 'processed' folder to ensure it will not be processed again. This is done within the main loop with the line `os.rename( file_dir + "/" + files[i], file_dir + "/processed/" + files[i])`.

4.2.5 Control Process

The main loop code (see Code 4.5) runs through all the files found in the 'datasets' folder (using the helper methods 4 and 5) and reads and uploads them one at a time. The code currently has some redundancy that was introduced as the processing power of the virtual machine we are using was limited. The goal is for this script to run endlessly (remove the break) and thus it will automatically upload any file that gets put into the 'datasets' folder without having to be run by a person each time. However, due to the aforementioned limitations that was not possible. In fact, we were limited to reading only three files at a time. This is all fairly easy to change but is nonetheless a limitation.

```
while(True):
    files = conv.get_files()
    file_types = conv.get_file_types(files)

    if len(files) != 0:
        processed = 1
        i = 0
        if 2 >= 3:
            while processed <= 3:
                print("Inserting: " + files[i])
                data = conv.read_files(files[i], file_types[i])

                if data.empty != True:
                    conv.send_to_sql(data)
                os.rename( file_dir + "/" + files[i],
                    file_dir + "/processed/" + files[i])
                processed += 1
                i += 1
        else:
            while processed <= len(files):
                print("Inserting: " + files[i])
                data = conv.read_files(files[i], file_types[i])

                if data.empty != True:
                    conv.send_to_sql(data)
                os.rename( file_dir + "/" + files[i],
                    file_dir + "/processed/" + files[i])
                processed += 1
                i += 1
    print("Processed some files! Run again to process more.")
    break
```

Listing 4.5: Controlling Loop

Chapter 5

Analysis: Bovine Data

5.1 Data Cleaning and Overview

5.1.1 Data Cleaning

Sensor data is known for noisiness and general uncleanliness [KKG+20]. In order to receive meaningful results, data cleaning is critical. This processing involves reducing data noise, imputing missing data, outlier detection, and data aggregation.

For each table within the bovine data provided (see Table 5.1) we need to determine how to clean and process the data.

5.1.2 Importing of the Data

Data can be imported in one of two ways. The first is to import them via CSV files using the *pandas* library. An example of this can be seen in the Code 5.1.

```
1 import pandas as pd
2 #if excel file
3 df_step_excel = pd.read_excel("path/to/file/filename.xlsx")
4 #if csv file
5 df_step_csv = pd.read_csv("path/to/csv_file/filename.csv")
```

Listing 5.1: Read Data from Excel or CSV

While possible to read from CSV files, as mentioned before, CSV files are not a reliable method of storage. Therefore, we will use the second option which is to read from a MySQL database into a pandas dataframe using the *pymysql* and *pandas* libraries. An example of this can be seen in following Code 5.2.

Table	Columns	Explanation
AfiAct2 Rest Hourly	Hour	Impute any missing hours of the day
	RestTime	Remove outliers and impute missing values.
	RestBout	Remove outliers and impute missing values.
AfiAct2 Steps and Activity 15 minutes	Step	Derive column for step difference.
	DateTime_Collected	Impute any missing 15 minutes of the day
AfiCollar Motion Hourly	Hour	Impute any missing hours of the day
	MotionHeatIndicator	Remove outliers and impute missing values.
	Motion	Remove outliers and impute missing values.
AfiCollar Rumination and Eating Hourly	Hour	Impute any missing hours of the day
	RuminationTimeInSeconds	Remove outliers and impute missing values.
	EatingTimeInSeconds	Remove outliers and impute missing values.

Table 5.1: Data Cleaning Performed on the Bovine Data

```
import pymysql
import pandas as pd
def mysqlconnect():
    # To connect MySQL database
    conn = pymysql.connect(
        host=hostname,
        user=username,
        password = pword,
        db=dbname,
        )

    return conn

def execute_sql(cur, sql):
    cur = conn.cursor()
    cur.execute(sql)
    output = cur.fetchall()
    return output

def close_connection(conn):
    conn.close()
conn = mysqlconnect()
aa2_rest = pd.read_sql("select * from AfiAct2_Rest_Hourly",
    conn)
```

Listing 5.2: Read from MySQL Database

Column	Explanation
Id	This id is from the insertion of the data into the database. It is a unique identifier but has no relation to the original data collected.
AnimalId	This id is the original identifier of each animal from the collection. There are multiple rows of data per animal.
Hour	This correlates to the hour of the day from which the data was collected.
Date_Collected	This correlates to the date from which the data was collected.

Table 5.2: Universal Attributes Across the Tables

Column	Explanation
RestTime	These are the amount of minutes spent resting at this hour on this date.
RestBout	These are the amount of bouts spent resting at this hour on this date.

Table 5.3: AfiAct2 Rest Hourly Data

5.1.3 Overview of Data

The data is contained within four main tables: AfiAct2_Rest_Hourly, AfiAct2_Step_and_Activity_15_minutes, AfiCollar_Motion_Hourly, and AfiCollar_Rumination_and_Eating_Hourly. The "AfiAct" tables contain data collected by the AfiAct Pedometer and the "AfiCollar" tables contain data collected by the AfiCollar.

All the tables have some attributes in common (see Table 5.2).

AfiAct2_Rest_Hourly collected from the pedometer is data relating to the resting of the cows. See Table 5.3 for a more detailed explanation.

AfiAct2_Step_and_Activity_15_minutes is the table with the most inherent differences. The data is collected from the pedometer but on a 15 minute interval rather than an hourly one. See Table 5.4 for more detailed explanations.

AfiCollar_Motion_Hourly collected data related to the motion of the cows from the AfiCollar sensor around their neck. See Table 5.5 for a more detailed explanation of the attributes.

AfiCollar_Rumination_and_Eating_Hourly also collects data from the Afi-

Column	Explanation
Time_Collected	This correlates to the exact time from which the data was collected.
Fract	Value unclear, calculation proprietary
Tag	Number of the activity monitor (cycled between cows)
Step	These are the total amount of steps as measured at that interval.
Unnamed: 6	Value unclear, calculation proprietary
DateTime_Collected	This correlates to the exact datetime from which the data was collected.
Activity	Number of steps hourly.
HeatIndicator	This is the result of the companies algorithm in detecting a relative increase from the baseline activity monitor that is indicative of *estrous*.

Table 5.4: AfiAct2 Step and Activity 15 minutes Data

Column	Explanation
Motion	This is a measure of the cows acceleration in the hour interval.
MotionHeatIndicator	This is the result of the companies algorithm in detecting a relative increase from the baseline motion monitor that is indicative of *estrous*.

Table 5.5: AfiCollar Motion Hourly Data

Column	Explanation
RuminationTimeInSeconds	This is a measure of the cows rumination time in seconds.
EatingTimeInSeconds	This is a measure of the cows eating time in seconds.
Min_Collected	Value unclear, calculation proprietary
Unnamed: 6	Value unclear, calculation proprietary
Unnamed: 7	Value unclear, calculation proprietary

Table 5.6: AfiCollar Rumination and Eating Hourly Data

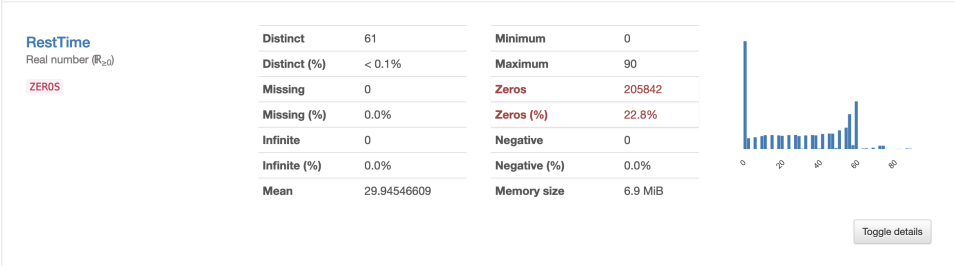

Figure 5.1: RestTime Attribute Overview

Collar around the neck of the cow on an hourly basis, but is instead focused on the cow's eating (see Table 5.6).

The data collected allows a good understanding of the daily life of a cow on this farm. The `pandas_profiling` library was used to understand the data, outliers, correlation between attributes, and create reports on each of the tables and their data. For example, the analysis of the AfiAct2_Rest_Hourly table provides information including a general overview, an overview of each attribute, interactions between the attributes, correlations between the attributes, and missing values.

The general overview provides information on the table as a whole, such as number of attributes, number of rows, any missing cells or duplicate rows, size in memory, and variable types. The overview of individual attributes gives a more detailed look at the attribute. This overview shows the number of distinct values, any missing values, the mean, the maximum, the minimum, and more. It also shows the general distribution of the attribute which is helpful in determining outliers (see Figure 5.1). There is also the interactions between variables as well as any correlations between them (see Figure 5.2).

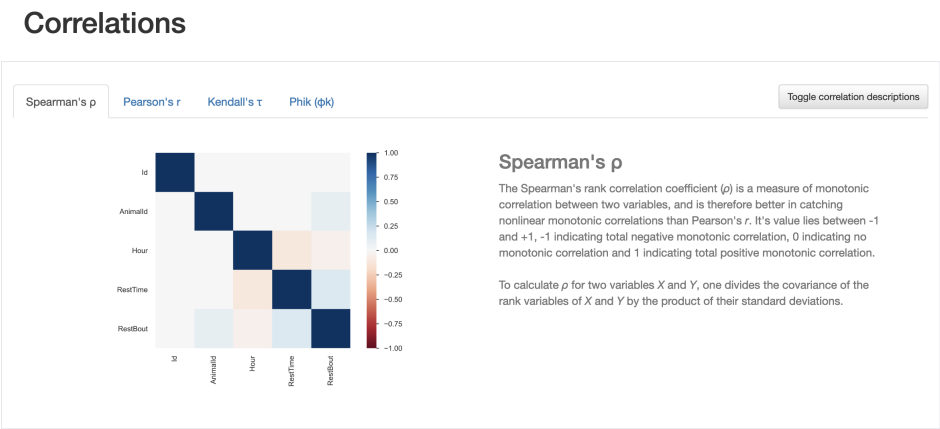

Figure 5.2: Correlation between AfiAct2_Rest_Hourly Attributes

5.2 Model Development

Model development proceeded in an exploratory fashion with no set end goal of the data analysis. However, outliers were detected during data cleaning, which prompted a focus on a model for outlier detection that would have practical use for farmers and researchers. Specifically, detecting outlier behaviour would either indicate something might be wrong with a cow or the equipment, both of which would be valuable for users. Potential models considered included supervised and unsupervised models as well as the use of different models within these subsections of machine learning models.

5.2.1 Supervised vs. Unsupervised

Within Machine Learning (ML), there are two forms of models: unsupervised or supervised models. The main difference between these two models is in the use of independent and target variables. Before the difference between the types of models can be explained, a bit of background in ML is required. First, ML statistical algorithms are shown historical data (or training data) to learn patterns within the data, this process is referred to as training the model [VPeE+20]. This data consists of both output and input variables, in general output variables are called target variables or dependent variables, and input variables are referred to as independent variables [VPeE+20].

Supervised learning is a form of Machine Learning (ML) that creates a statistical ML model to predict a value of a target variable. The data used to train this model, the input data, contains both the independent and

target variable [VPeE+20]. The training of this model is done in such a way as to generate an equation or understanding of the relationship between the input and output variables so that the model can then predict an output variable if given the input variables [VPeE+20]. In order to use this type of learning, one must have the output variable within the training data so that we can determine the predictive accuracy.

Unsupervised learning algorithms do not have labelled data or a target variable it wishes to predict [VPeE+20]. These algorithms focus on looking for undetected patterns or underlying structures within the data. This is the key difference between these two type of models. Unsupervised models do not rely upon having an output or target variable. This research focused on unsupervised models as there was no target variable and the analysis was looking at the underlying relationships between variables and searching for outliers.

5.2.2 Outlier Detection Models

Clustering models are one technique for outlier detection. Cluster analysis is one of the most famous applications of unsupervised ML. Cluster analysis groups the data based on similar patterns and common attributes visible in the data [VPeE+20]. The goal was to identify outliers based on their belonging or lack of to certain groups (ie. their distance from other data points).

The first model investigated was k-means clustering. The objective of clustering is to group data points in such a way that observations within a group are similar to each other but different to those in other groups [PfHE19]. For k-means clustering, this requires specifying the number of groups (or clusters), k, then the algorithm will then assign each observation within the input data to a cluster [PfHE19]. In order to determine the number of clusters we should use, k-means clustering was performed with a varying number of clusters and then the number of groups at which the Within Cluster Sum of Square (WCSS) - sum of squared distance between each point and the centroid (the clusters central point) in a cluster - plateaus was selected. An example of the results of this iterative calculation of WCSS can be seen in Figure 5.3. The WCSS begins to plateau at 9, which was the number of clusters used. Executing k-means using 9 clusters and the `KMeans` model from the `sklearn` library (see Code 5.3) produces results in Figure 5.4 where the observations are depicted with the x-axis showing the date the observation was collected on and the y-axis showing the motion measurement. Each cluster of data has its own individual colour.

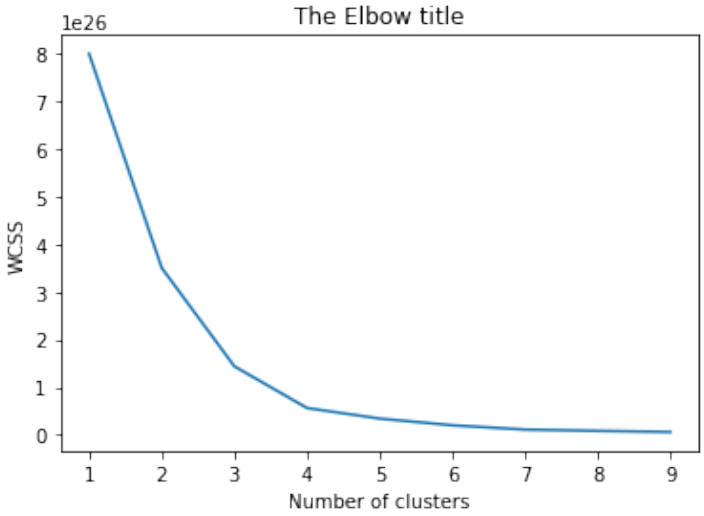

Figure 5.3: WCSS for K-Means of Animal 1098

```
kmeans = KMeans(
    init="random",
    n_clusters=9,
    n_init=10,
    max_iter=300,
    random_state=42
)
kmeans.fit(input_data_for_1098)
```

Listing 5.3: Code to Run KMeans on Animal 1098

In the results of the k-means model, it can be seen that the data is grouped in accordance to something other than date. This is a good indication for our model, as this means that there is a underlying structure to the data that is important in these groupings, especially as we can see that the outliers we can visually detect on the graph are grouped separately.

This k-means exploration was then followed up with experimentation using a K-nearest-neighbours (KNN) supervised learning model in an effort to predict outliers. KNN classifies data points based on simple majority vote from the neighbours, it does not construct a traditional model [VPeE+20]. This model can be used well when there is a complex relationship between the input variables and the target variable [VPeE+20]. However, while the code could detect outliers, there was no way to determine the model's accuracy without validation data. KNN also does not perform at its best when

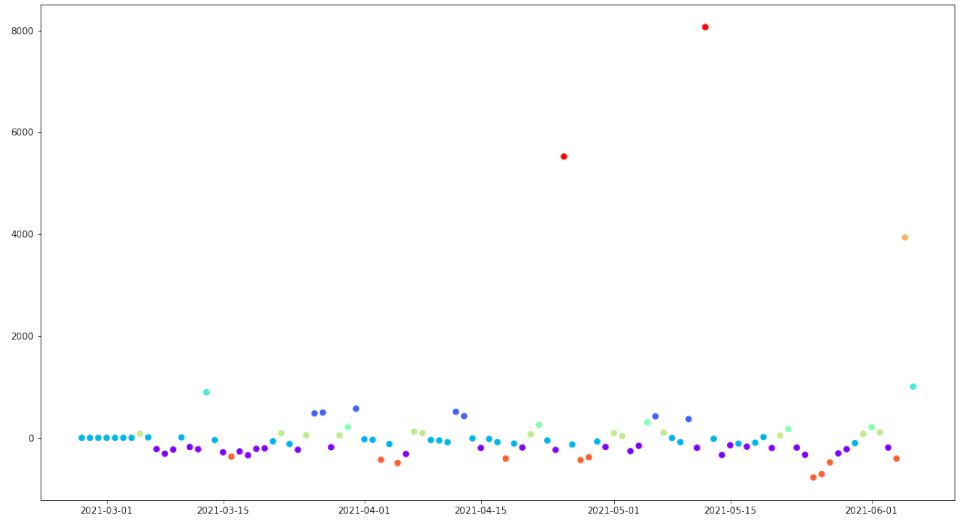

Figure 5.4: K-Means of Animal 8061

there is no clear target variable [VPeE+20]. The results of this exploration in KNN can be seen within Figure 5.5. The graph shows a subset (an informal training dataset) of Animal 8061's MotionHeatIndicator data as the y-axis and a integer version of the date as the x-axis. The red colour indicates outliers and green indicates inliers.

As KNN is a supervised technique and the data does not have a clear target output, the focus of the exploration shifted to unsupervised learning techniques, including DBSCAB and Isolation Forests. DBSCAN groups together closely packed points, where close together refers to a minimum number of points that must exist within a certain distance [PfHE19] which makes it a good model for outlier detection. However, the decision was to proceed with a more traditional method of anomaly or outlier detection, Isolation Forests, as it works a bit better with more time and date centered data. Isolation forests consist of a collection of individual tree structures that recursively partition the dataset [AAeP+19]. Within each iteration, a random feature is chosen and the data is then split based on a randomly chosen value between the minimum and maximum of the randomly chosen feature [AAeP+19]. This is repeated until the entirety of the dataset is partitioned. Anomalies within the data generally have much shorter paths from the root than normal observations, and thus are more easily detected [AAeP+19].

The fitting of an isolation forest can be seen in Code 5.4. This code

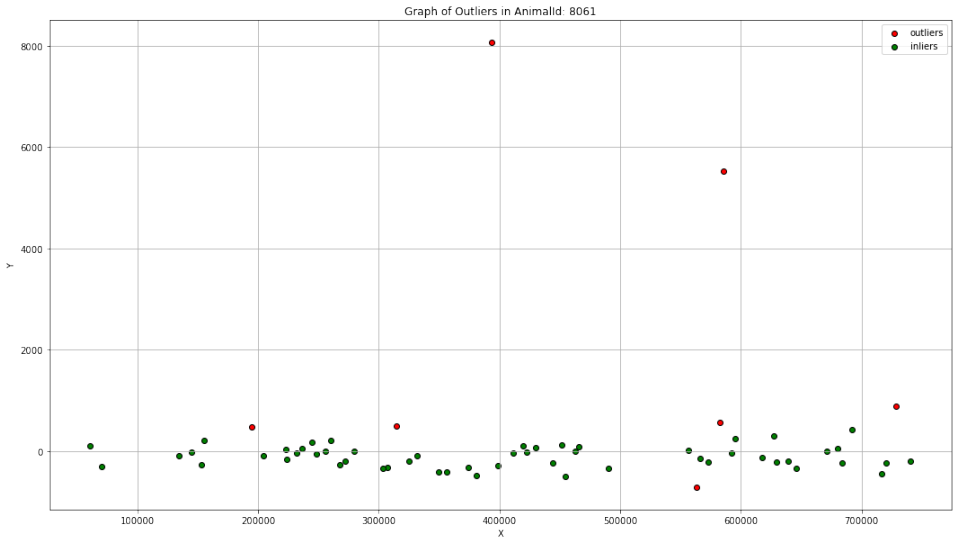

Figure 5.5: KNN of Animal 8061

takes the dataframe containing the data, in this case the data from AfiCollar_Motion_Hourly, the column for which it is trying to detect the outlier (in this example MotionHeatIndicator), and a percentage of outliers allowed in a sequence. The results of fitting the model can be found in Figure 5.6. The isolation forest clearly identified outliers within the MotionHeatIndicator attribute for Animal 8061.

Interestingly, after closer examination of the data and outliers on variables such as steps and motion, these spikes actually indicated *estrus* and were not technically outliers but expected spikes.

```
def isolation_forest_anomaly_detection(df,
                                       column_name,
                                       outliers_fraction):
    """
    In this definition, time series anomalies are detected
    using an Isolation Forest algorithm.
    Outputs:
        df: Pandas dataframe with column for detected
    Isolation Forest anomalies (True/False)
    """
    #Scale the column that we want to flag for anomalies
    min_max_scaler = preprocessing.StandardScaler()
    np_scaled = min_max_scaler.fit_transform(df[[column_name
    ]])
    scaled_time_series = pd.DataFrame(np_scaled)
    # train isolation forest
    model =  IsolationForest(contamination =
    outliers_fraction)
    model.fit(scaled_time_series)
    #Generate column for Isolation Forest-detected anomalies
    isolation_forest_anomaly_column = column_name+"
    _Isolation_Forest_Anomaly"
    df[isolation_forest_anomaly_column] = model.predict(
    scaled_time_series)
    df[isolation_forest_anomaly_column] = df[
    isolation_forest_anomaly_column].map( {1: False, -1:
    True} )
    return df
```

Listing 5.4: Method to Fit Isolation Forest

5.3 Event Detection

Using the outlier detection models on specific attributes, especially those related to behaviours that change around *estrus*, we were able to unintentionally create an event detection model, specifically an *estrus* event detection model.

5.3.1 Previous Event Detection

Currently, *estrus* detection is done either through the company that provided the sensors or through the calculation of rolling averages and other measures. Traditionally a 7 day rolling average was used, so each hour of the day had a mean number. Then, a threshold was set to detect 80-100 percent relative increases or a 3 standard deviation relative change in activity (these

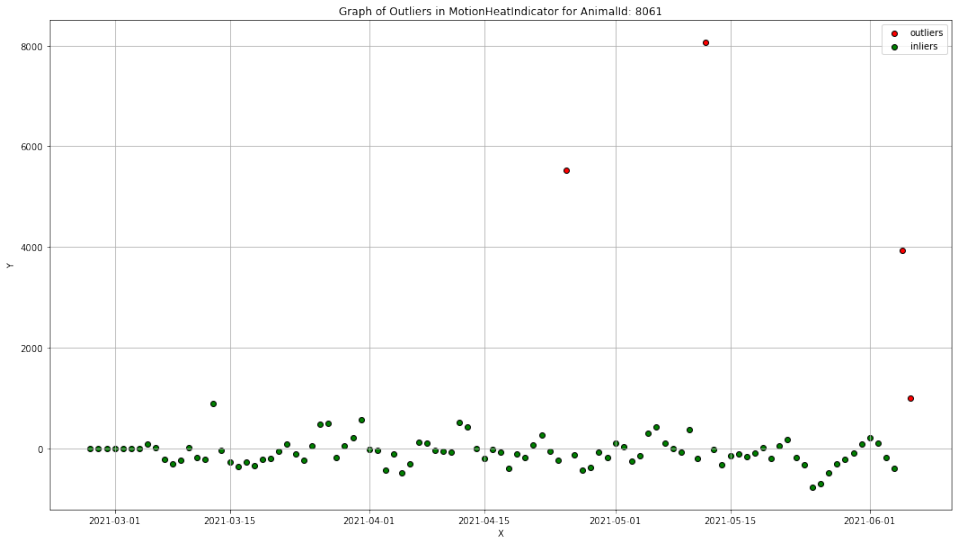

Figure 5.6: Isolation Forest of Animal 8061

thresholds can vary), which when triggered would send out an *estrus* alert. The hour block at the time of the alert was considered the beginning of estrus. The end of estrus was considered the hour block that observes the activity measurement dropping below the threshold level.

In prior research this event detection was done retrospectively from the data in the CSV files and through the coding language R. First, the algorithm would identify the estrus event based on the rolling average and threshold levels. Second, it would lay out several variables related to this event, such as maximum relative and absolute increase, total duration, hour block of the beginning and end of estrus. All of these were then used to inform the research.

5.3.2 R Translation

R as a coding language originated from statistical analysis. In fact, R can perform many complex models and visualizations. There are more than 13,000 R packages were available via the Comprehensive R Archive Network (CRAN) for deep analytics [CTC22]. It is very popular among researchers and scholars for its ease of use and many libraries for data cleaning, visualizations, and training and evaluating machine learning and deep learning algorithms [CTC22].

R, while great for statistical learning, does not work as well as Python

for machine learning and data analysis [Ped22]. Similarly, while R is popular amongst academics and statisticians, people who will be working on this in the future may not be familiar with R, but Python is more regularly taught in all quantitative disciplines [CTC22].

Python also accepts more formats of data than R does. Likewise, it is easier to create web interfaces and connect to databases in Python compared to R [Ped22] as Python is also a general-purpose coding language. Therefore, it makes sense to translate the existing R code to Python so that in the future, we not only have working copies of both for anyone to use, we can also easily continue into machine learning models for event detection. Lastly, as there is already a model that is seemingly detecting estrus running in Python, in order to compare the effectiveness of the ML model and the current detection method, removing the language barrier between the two ensures a more equal comparison. All in all, translating the current method from R to Python is a reasonable decision.

5.3.3 Challenges

While R and Python perform similar tasks and there are some similarities in terms of how they do so, it is not a one-to-one match. There are certain features within the R code that do not have an exact Python equivalent.

This became clear in a few specific instances. The first can be seen in Code 5.5, specifically line 5.

```
1 df_step3 <- df_step2 %>%
2   filter(!is.na(date_time1)) %>%
3   as_tsibble(key = id,index =  date_time1) %>%
4   index_by(date_time1) %>%
5   fill_gaps(.full = FALSE)
```

Listing 5.5: Filling in Missing Hours

This code fills in any missing hour blocks within the dataframe. For example, if on December 12th the hour attribute jumps from 3 to 5, it would insert a row of NAs on December 12th at 4. The python equivalent of this code can be seen in Code 5.6. These four lines of code is the equivalent of one line of R code.

```
1 df_step3 = (df_step2.set_index("dt1").groupby("id").apply(
      lambda x: x.asfreq("H")))
2 df_step3 = df_step3.rename(columns={"id":"animalid"}).
      reset_index(drop=False)
3 df_step3 = df_step3.drop("animalid", axis = 1)
```

```
df_step3["hour"] = df_step3["dt1"].dt.hour
```

Listing 5.6: Filling in Missing Hours - Python

The next instance of differences was found during the calculation of step difference. The step attribute is a sum of all the steps until it reaches a threshold and is reset. In order to determine relative difference between hours, the step difference between hours must be calculated. Code 5.7 shows how this is done in R.

```
df_step3 <- df_step3 %>%
  mutate(step = step1) %>%
  as_tibble() %>%
  arrange(id,date_time1) %>%
  group_by(id) %>%
  mutate(step_diff = case_when(step<lag(step)~ 65536-lag(
    step)+step, T ~ step-lag(step)))
```

Listing 5.7: Calculation of Step Difference in R

The calculation of step difference in Python can be seen in Code 5.8. As Python does not have a lag function like R, the solution requires iterating through the dataframe, separating by animal, and then calculating the difference. It is also important to make sure that if there are any negative values it is set to the correct difference as that is indicative of the pedometer resetting.

```
def diff_steps(aa2_steps):
    diffs = []
    aa2_steps = aa2_steps.sort_values(by=["id", "dt1"],
    ascending = [True, True])
    animalids = aa2_steps.drop_duplicates(subset = ["id"]).id.
    tolist()
    for i in range(len(animalids)):
        temp = aa2_steps[aa2_steps.id == animalids[i]]
        step_count = temp["step"].tolist()
        diff = temp["step"].diff().tolist()
        for j in range(len(diff)):
            if diff[j] < -1000:
                diffs.append((step_count[j]+65535)-step_count[j
    -1])
            else:
                diffs.append(diff[j])

    aa2_steps["step_diff"] = diffs
    return aa2_steps
```

Listing 5.8: Calculation of Step Difference in Python

The third instance of R to Python differences was found when the R code used a Kalman Filter to impute the missing values for step difference.

```
library(imputeTS)
df_step5 <- df_step4

# we use Kalman method for imputation of NAs
df_step5 <- df_step5 %>%
  as_tibble() %>%
  as_tsibble(id,date_time1) %>%
  group_by(id) %>%
  mutate(across(c(step_diff2) ,imputeTS::na_kalman ,smooth=T
    ,.names = "{.col}_imputed" ))
```

Listing 5.9: Kalman Filter Imputation

In fact, an exact equivalent for this piece of code has not been found. Every Kalman filter that was tried resulted in wildly different imputations. The closest equivalent that was found can be seen in Code 5.10

```
df_step4["step_diff2_imputed"] = df_step4["step_diff2"]
df_step4["step_diff2_imputed"].fillna(df_step4.groupby(["id","
    hour"])["step_diff2_imputed"].transform("mean"), inplace=
    True)
df_step4["step_diff2_imputed"].fillna(df_step4["
    step_diff2_imputed"].mean(), inplace=True)
```

Listing 5.10: Imputation of Step_Diff in Python

The last two main differences were the rolling mean, which suffers from the same situation as the Kalman Filter Imputation, and the Run-length Encoding which had to be implemented manually.

Essentially, anytime a specific package or library is not cross-implemented or the code requires iteration through the dataframe (in the brute-force implementation) there will most likely be difficulties in translating from R to Python.

Chapter 6

Results

The results are as follows: a script to upload data from a CSV file to a MySQL database is working and customizable, the temporary MySQL database can be accessed remotely and is hosted securely on a Digital Ocean server, data overview and cleaning are completed, the developed isolation forest model detects outliers and depending on the variable used *estrus*, and the previous event detection code has been translated from R to Python.

One interesting result of the R translation - which is possibly the result of the aforementioned in-equivalencies - was the sheer amount of false-positives and even a few false negatives for event detection. There were approximately 173 events detected via the R code, however, the Python code detected around 500. This is most likely due to the rolling mean and Kalman Filter imputation calculations as well as the thresholds. Specifically, I propose that adjusting the thresholds will reduce the amount of false positives and that correcting any issues found with the rolling mean and Kalman Filter imputation will reduce the false negatives.

This is especially interesting as the models developed for outlier detection, rather than event detection, had less false positives than the translation of the code that is currently used for *estrus* detection. We cannot say for certain as we have no validation data, however, based on current understanding of *estrus*, it is a period of 6-20 hours characterized by restlessness, in other words high step counts, every approximately 21 days. When the Isolation Forest model is run on Animal 66, we can see that it detect "outliers" of a high amount of steps in a potential 24 day cycle. This can be seen in Figure 6.1, where we can clearly see the detection of the high number of steps in March and April. There are evidently some outlier values and errors that might still be within these, but in comparison to the Event Detection now, there are significantly less false positives. In the R translation for Animal 66, *estrus* is being detected 10 times within March and April as opposed to the two times with the Isolation Forest and original R code. Evidently, the current Python translation needs more refinement, specifically in the translation and thresholds, but it clear that there is a potential for a machine learning model as well.

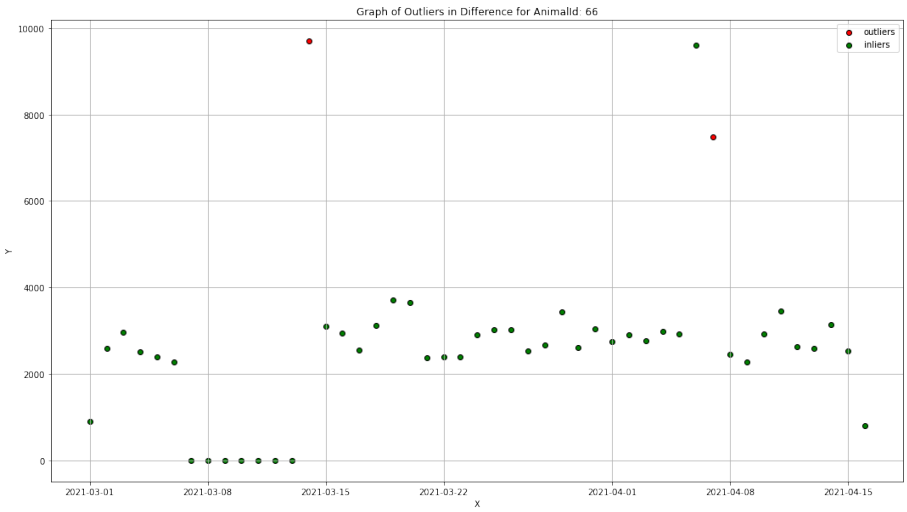

Figure 6.1: Isolation Forest of Steps for Animal 66

Chapter 7

Conclusion and Future Work

Due to storing data in spreadsheets and conflicts between academia and industry on the desired outcome (ie. real-time event detection vs. the integration and analysis of multiple datasets for more in-depth and complex analysis of multiple variables), analyzing data in the desired way for academics was challenging. However, now academics and researchers can easily upload and access data and thus utilize it in a multitude of different ways. Now that the data is accessible, the data can be used in a variety of new ways and allows the input of researchers and farmers. Thus the future development of detection models and the analysis of the data can become better and more useful than ever.

While there is a need for even more open-source and accurate event detection for bovine researchers, it is clear that there is potential for a multitude of different projects and paths. In future works, we must create a more permanent data storage solution as the current one does not have enough storage. Along those lines, there are some issues with the R translation/event detection through the rolling average and thresholds that must be resolved. Additionally, it would be beneficial to focus more on the analysis of the data rather than the accuracy of detection. In fact, an expansion of the data through external validation and feature engineering as well as classification of the animals would be a great next step toward creating an amalgamation of all the animals and their physiology and genetic classifications. After which, answering questions such as how much more likely a cow is to great pregnant based off of certain factors would be much simpler.

Bibliography

[AAeP+19] Sridhar Alla, Suman K. Adari, SpringerLINK ebooks Professional, Applied Computing, SpringerLink (Online service), and O'Reilly for Higher Education. *Beginning Anomaly Detection Using Python-Based Deep Learning: With Keras and PyTorch*. Apress, Berkeley, CA, 1st 2019. edition, 2019. → pages 22

[Agg13] Charu C. Aggarwal. *Managing and Mining Sensor Data*. Springer-Verlag, New York, NY, 2013. → pages 5

[BMS+17] Tracy A. Burnett, Augusto M. L. Madureira, Bruna F. Silper, A. C. C. Fernandes, and Ronaldo L. A. Cerri. Integrating an automated activity monitor into an artificial insemination program and the associated risk factors affecting reproductive performance of dairy cows. *Journal of Dairy Science*, 100(6):5005–5018, 2017. → pages 1

[BPKC18] Tracy A. Burnett, Liam Polsky, Manveen Kaur, and Ronaldo L. A. Cerri. Effect of estrous expression on timing and failure of ovulation of holstein dairy cows using automated activity monitors. *Journal of Dairy Science*, 101(12):11310–11320, 2018. → pages 2

[CBM+21] R. L. A. Cerri, T. A. Burnett, A. M. L. Madureira, B. F. Silper, J. Denis-Robichaud, S. LeBlanc, R. F. Cooke, and J. L. M. Vasconcelos. Symposium review: Linking activity-sensor data and physiology to improve dairy cow fertility. *Journal of Dairy Science*, 104(1):1220–1231, 2021. → pages 1, 2, 5, 6

[CTC22] IBM Cloud Team and IBM Cloud. Python vs. R: What's the Difference?, 2022. → pages 25, 26

[KKG+20] Rajalakshmi R. Krishnamurthi, Adarsh A. Kumar, Dhanalekshmi D. Gopinathan, Anand A. Nayyar, and Basit B. Qureshi.

An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. *Sensors (Basel, Switzerland)*, 20(21):6076, 2020. → pages 5, 14

[Law21] Ramon Lawrence. Database Introduction, 2021. → pages 4

[Ped22] Priya Pedamkar. R vs Python — Top 11 Differences You Should Know, 2022. → pages 26

[PfHE19] Ankur A. Patel and O'Reilly for Higher Education. *Hands-On Unsupervised Learning Using Python*. O'Reilly Media, 2019. → pages 20, 22

[VPeE+20] Vaibhav Verdhan, Springer Professional, Applied Computing eBooks 2020 English/International, SpringerLink (Online service), and O'Reilly for Higher Education. *Supervised Learning with Python: Concepts and Practical Implementation Using Python*. Apress, Berkeley, CA, 1st 2020. edition, 2020. → pages 19, 20, 21, 22

Appendix

```
hostname="127.0.0.1"

file_dir = "/Users/emilymedema/honours/datasets"

table_info = {
    "AfiCollar_Motion_Hourly": ["AnimalId",
                                "Date_Collected",
                                "Hour",
                                "Motion",
                                "MotionHeatIndicator"],
    "AfiAct2_Steps_and_Activity_15_minutes": ["Time_Collected",
                            "Date_Collected",
                            "Fract",
                            "AnimalId",
                            "Tag",
                            "Step",
                            "Unnamed: 6",
                            "DateTime_Collected",
                            "Activity",
                            "HeatIndicator"],
    "AfiCollar_Rumination_and_Eating_Hourly": ["AnimalId",
                            "Date_Collected",
                            "Hour",
                            "RuminationTimeInSeconds",
                            "EatingTimeInSeconds",
                            "Min",
                            "Unnamed: 6",
                            "Unnamed: 7"],
    "AfiAct2_Rest_Hourly": ["AnimalId",
                              "Hour",
                              "Date_Collected",
                              "RestTime",
                              "RestBout"]}
banned_headers = ["Date", "DateTime", "Time", "Min"]
convert_headers = {"RestTime": "RestTime(minutes)", "RestBout":
     "RestBout(#)"}
```

Listing 1: Custom Information for Connection

```
def compare_headers(self, df_head):
    """
    This function compares the headers of the read file to the
    column names of the mysql tables
    Returns the name of the table that has those columns
    """
    df_head = self.rename_headers(df_head)
    length = 0
    table = ""
    for key, value in table_info.items():
```

```
        temp = len(list(set(df_head) & set(value)))
        if temp > length:
            length = temp
            table = key

    return table
```

Listing 2: Compare Headers Helper Method

```
def rename_headers(self, df_head):
    """
    This function renames columns of the tables that need it.
    Returns the df with the renamed headers.
    """
    for i in range(len(df_head)):
        if df_head[i] in banned_headers:
            df_head[i] = df_head[i] + "_Collected"
        else:
            vals = list(convert_headers.values())
            keys = list(convert_headers.keys())
            for j in range(len(vals)):
                if df_head[i] == vals[j]:
                    df_head[i] = keys[j]

    return df_head
```

Listing 3: Rename Headers Helper Method

```
def get_files(self):
    """This function gets all the pure file names"""
    files = []
    for file in os.listdir( file_dir ):
        if file.endswith(".xlsx") or file.endswith("csv"):
            files.append(file)

    return files
```

Listing 4: Get Files Method

```
def get_file_types(self, files):
    """
    This function will get all the file types within the
    specified directory.
    """
    file_type = []
    for name in files:
        temp = name.rpartition(".")[-1]
        file_type.append(temp)
    return file_type
```

Listing 5: Get File Types Method