LEARNING MODELS FROM AVIONICS DATA STREAMS

Sinclair Gurny

Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Dr. Carlos A. Varela, Chair
Dr. Barbara M. Cutler
Dr. Stacy E. Patterson

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[May 2020]
Submitted March 2020

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor, Prof. Carlos Varela, for his support and supervision on this research. I would like to thank my entire committee for teaching me so much over the years: Prof. Carlos Varela, Prof. Barbara Cutler, and Prof. Stacy Patterson. I also would like to thank my colleagues of World Wide Computing Laboratory: Paul and Jason. I would like to thank all my friends who were with me throughout my time at RPI. As well as my friends, Adam and Nampoina, who supported me on a daily basis during my time working on my thesis. Finally, I want to thank my family for all the love and moral support over the years.

ABSTRACT

In aviation, there are many values that are useful for pilots to know that cannot be measured from sensors and require calculation using various charts or other means to accurately estimate. Aircraft take-off distance requires knowing wind speed, pressure altitude, and temperature. However, it is possible for the inputs of these calculations to change during flight, or be calculated incorrectly by mistake, and it would be useful for pilots to know these values in real-time. PILOTS is a programming language for spatio-temporal data stream processing. We have added improved integration for machine learning algorithms as well as a linguistic abstraction for training these models. In data-driven systems, it can be useful to use distributed processes for computation. We have designed a declarative framework for federated learning and the aggregation of results from multiple related models within PILOTS. Furthermore, we built a model using PILOTS that is able to estimate weight in real-time during take-off of a fixed-wing aircraft using data available from the avionics. We evaluated the results of several models on accuracy and timeliness. Data was collected from the flight simulator X-Plane. Accidents such as the fatal crash of Cessna 172R N4207P could have been prevented using the weight estimation methods illustrated.

CHAPTER 1
Introduction

1.1 Motivations

More devices than ever are producing data, from cellphones to the thousands of aircraft in the sky. Collecting and processing this data can provide useful insights. Data from aircraft flight recorders can provide clues into how an accident occurred and how it could have been prevented. Aircraft sensors measure physical quantities to help pilots and flight automation systems with situational awareness and decision making. Unfortunately, some important quantities of interest (QoI), e.g. aircraft weight, cannot be directly measured from sensors. As a consequence, accidents can happen, exemplified by Tuninter Flight 1153 and the crash of Cessna 172R N4207P, where the airplanes were underweight (not enough fuel) and overweight (6% over maximum gross weight).

Take-off and landing are the most dangerous times of flight because the most incidents occur during these times. However, early into take-off is also the safest time to abort the flight. In large aircraft, V_1 is the take-off decision speed: the speed above which take-off should not be aborted. If anything occurs before the aircraft reaches V_1, the aircraft can safely abort and come safely to a stop on the runway. However, if anything occurs after V_1, the aircraft must try to continue take-off. During take-off, there is increased lift due to ground effect. This means that an aircraft that is able to take-off may not necessarily be able to operate properly during other phases of flight. Therefore, being able to detect dangerous conditions, such as being overweight, during take-off is critical to maximize safety.

The physics-based approach to estimating aircraft weight during take-off is based on modeling the forces acting on the aircraft. Aerodynamic forces, such as lift and drag, can be estimated fairly accurately given certain properties of the aircraft, such as coefficient of lift and cross-sectional area. Thrust of a propeller cannot be calculated easily; however, there are tables which give the thrust depending on atmospheric conditions, rpm, and other variables. The difficulty in this approach is that the rolling resistance depends on the coefficient of friction between the landing gear and the runway surface as well as the force being applied to the runway. The specific runway, presence of water, oil, or tire wear can impact the coefficient of friction. Most importantly, calculating the force on the runway requires knowing

the weight, which causes a circular dependency. However, using a data-driven approach does not have this issue and does not require knowing aerodynamic properties of the aircraft.

Machine learning allows learning models from data. There are various programming languages which are capable of machine learning each with different styles. Declarative programming is a style which focuses on what a computation should do while abstracting away details of how it should be done. Declarative programming is more concise and direct which is helpful for users who do not have a strong programming background. Removing unnecessary details of control flow makes declarative programs faster to write and allows for higher levels of reasoning on programs. Declarative programming more resembles mathematical logic than the low-level operations of a language like C. Creating a declarative machine learning interface allows for significantly faster prototyping of systems created by users who understand the domain but may not understand the way to use machine learning in other programming languages.

Aircraft communicate with a network of other aircraft and various air traffic control towers as they fly. Each aircraft has limited computational abilities but can leverage its communication abilities to make up for its limitations. Distributed machine learning allows for a system of small compute nodes to train models that would normally only be possible using a powerful server. Aircraft produce large amounts of data which may not be feasible to send to a central server to train a model. Federated learning is an method of distributed machine learning which allows each node to train on its private local training data and collaboratively learn a model. Furthermore, it can be useful to combine the results of distinct models to come up with a final result. Models created for error detection could be learned across multiple aircraft and when an aircraft detects an error other aircraft can predict the source of the error using their local models.

1.2 Contributions

In this thesis, we look at the use of declarative machine learning to learn models from data. As well as using avionics data during take-off to learn a model of aircraft weight. The contributions are summarized as follows:

- PILOTS is a highly declarative domain-specific programming language for spatio-temporal data stream processing [1]. We implement the proposed additions for declar-

ative supervised learning designed by Imai et. al [2].

- We propose an extension to our declarative supervised learning grammar for federated learning. This extension allows for models to be trained collaboratively without the need to transfer training data to a central server for training.

- We design several methods which allow for the creation of ensemble models within PILOTS. These methods improve the ability to create more advanced systems within PILOTS.

- We develop a data-driven weight estimation method for fixed-wing aircraft during take-off. This model provides another layer of safety to prevent accidents such as Tuninter Flight 1153 and the crash of Cessna 172R N4207P.

1.3 Structure of Thesis

In the following chapter, we discuss PILOTS and the advancements for stream analytics. Chapter 3 discusses our proposed data-driven aircraft weight estimation method and experimental results, Chapter 4 discusses related work, and Chapter 5 summarizes the thesis as well as explores avenues for further work.

CHAPTER 2
Declarative Machine Learning Programming

2.1 PILOTS Programming Language

PILOTS[1] is a ProgrammIng Language for spatiO-Temporal data Streaming especially designed for building data-driven applications on systems such as airplanes. PILOTS is a declarative language with a syntax similar to Pascal, see Figure 2.3. Users of PILOTS can specify how to use spatio-temporal data streams to produce other data streams, which can be seen in various works by Varela, Imai, and others [1], [3], [2], [4]. PILOTS programs define the high-level operations showing how the output data streams are based on the input data streams. As a data stream processing language, PILOTS is unique because of its first-class support for data selection, data interpolation, and error recovery.

2.1.1 Data Selection

Data selection is used to find the most applicable data when none is available at a specific location and/or time. PILOTS has three types of data selection methods: `closest`, `euclidean`, `interpolate`.

- `closest`: Takes the closest value in 1-D space from $\{x, y, z, t\}$ (both space and time) to find the data point closest to the given value. For example, finding the closest data point in t would find the most recent data point to the current time.

- `euclidean`: Extension of closest into 2-D and 3-D euclidean space. Finds the data point with the closest Euclidean distance using two or more values from $\{x, y, z\}$. For example, could be used to find closest neighboring aircraft's data.

- `interpolate`: Given any number of values from $\{x, y, z, t\}$ to linearly interpolate from multiple data points. It also takes another argument n_{interp} which specifies the maximum number of closest data points to interpolate. For example, given the two closest data points in the form (x, d): $(1, 4)$ and $(3, 8)$ and interpolating for $x = 2$ would give $d = 6$.

[1] http://wcl.cs.rpi.edu/pilots/

Figure 2.1 shows `Twice`, a simple PILOTS program[2] [1]. The program has two input data streams $a(t)$ and $b(t)$ and one output data stream, e, which outputs every second. The input data stream b is supposed to be twice the value of a, which are both functions of time. The output data stream e outputs the error of how far a and b are from the correct values, defined as $b - 2 * a$. This example will be advanced on further in later examples to show other features of PILOTS.

```
program Twice;
  inputs
    a(t) using closest(t);
    b(t) using closest(t);
  outputs
    e: b - 2 * a at every 1 sec;
end;
```

Figure 2.1: Simple PILOTS program Twice.

2.1.2 Operation Modes

PILOTS programs can be ran in two operational modes [1]:

- *real-time mode*: The default operational mode. In this mode, the program sends and receives data in real-time. The program terminates when either the input data streams end or a specified time has passed.

- *simulation mode*: This mode is used for simulations ran within a specified time span and often using data saved in files. The program ignores the output frequencies and runs to completion as fast as possible.

2.1.3 Error Detection and Error Recovery

An error function is a numerical function that can detect errors in redundant input data streams. An example of an error function is the output stream from the Twice program, Figure 2.1, $e = b - 2 * a$. This error function can be used to detect whether a and b are in sync or not.

Error detection is done by analyzing the values of error functions over time and comparing them to specified error signatures [5]. An error signature is defined as a constrained

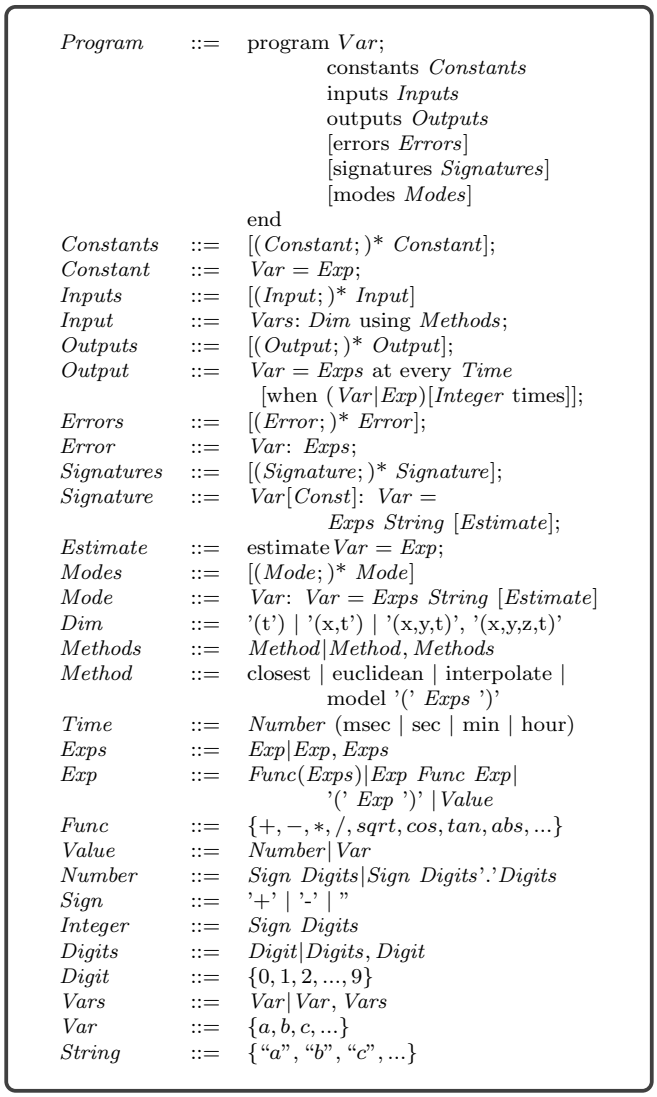
[2]PILOTS version 0.6: https://github.com/RPI-WCL/pilots/releases/tag/v0.6

mathematical function pattern that is used to capture the characteristics of an error function $e(t)$. More simply, an error signature represents a general shape that the error function's value over time can look like. Error signatures require domain knowledge to properly design. Error recovery is achieved when the value of an error function matches a specific error signature; each error signature can specify a way to correct associated data streams [3].

Figure 2.2 shows an advancement of the Twice example called `TwiceCorrect` [2]. It shows the error detection and correction of data streams in PILOTS. Just like the previous example, the program takes two inputs a and b, where b is supposed to be twice of a. However, this program includes `errors` and `signatures`. The `errors` section creates an expression that represents the error function and `signatures` creates expressions to represent error signatures which capture different types of error states. In this example, there are four signatures. The first signature $s0$ represents when the error function looks like the line $e(t) = 0$ over time, which means that both a and b are in sync and b is twice of a. The second and third signature occur when one of the input data streams fail: the stream is not producing any new data. The signature $s1(K) : e = 2 * t + K$, where K is any possible constant, matches when $e(t)$ looks like a line with a slope of 2. Similarly, the $s2$ signature matches when $e(t)$ looks like a line with a slope of -2. The last signature occurs when both a and b are working but the values are out of sync, where e looks like a horizontal line $e(t) = K$ such that $|K| > 20$.

```
program TwiceCorrect;
inputs
    a(t) using closest(t);
    b(t) using closest(t);
outputs
    e, mode at every 1 sec;
errors
    e: b - 2 * a;
signatures
    s0: e = 0                    "Normal mode";
    s1(K): e = 2 * t + K         "A failure"
      estimate a = b / 2;
    s2(K): e = -2 * t + K        "B failure"
      estimate b = a * 2;
    s3(K): e = K, abs(K) > 20 "Out-of-sync";
end;
```

Figure 2.2: PILOTS program specification with error signatures, TwiceCorrect.



Program ::= program *Var*;
constants *Constants*
inputs *Inputs*
outputs *Outputs*
[errors *Errors*]
[signatures *Signatures*]
[modes *Modes*]
end
Constants ::= [(*Constant*;)* *Constant*];
Constant ::= *Var* = *Exp*;
Inputs ::= [(*Input*;)* *Input*]
Input ::= *Vars*: *Dim* using *Methods*;
Outputs ::= [(*Output*;)* *Output*];
Output ::= *Var* = *Exps* at every *Time*
[when (*Var*|*Exp*)[*Integer* times]];
Errors ::= [(*Error*;)* *Error*];
Error ::= *Var*: *Exps*;
Signatures ::= [(*Signature*;)* *Signature*];
Signature ::= *Var*[*Const*]: *Var* =
Exps String [*Estimate*];
Estimate ::= estimate *Var* = *Exp*;
Modes ::= [(*Mode*;)* *Mode*]
Mode ::= *Var*: *Var* = *Exps String* [*Estimate*]
Dim ::= '(t') | '(x,t') | '(x,y,t)', '(x,y,z,t)'
Methods ::= *Method*|*Method*, *Methods*
Method ::= closest | euclidean | interpolate |
model '(' *Exps* ')'
Time ::= *Number* (msec | sec | min | hour)
Exps ::= *Exp*|*Exp*, *Exps*
Exp ::= *Func*(*Exps*)|*Exp Func Exp*|
'(' *Exp* ')' |*Value*
Func ::= $\{+, -, *, /, sqrt, cos, tan, abs, ...\}$
Value ::= *Number*| *Var*
Number ::= *Sign Digits*|*Sign Digits*'.'*Digits*
Sign ::= '+' | '-' | ''
Integer ::= *Sign Digits*
Digits ::= *Digit*|*Digits*, *Digit*
Digit ::= $\{0, 1, 2, ..., 9\}$
Vars ::= *Var*| *Var*, *Vars*
Var ::= $\{a, b, c, ...\}$
String ::= {"*a*", "*b*", "*c*", ...}

Figure 2.3: PILOTS grammar.

2.2 Declarative Supervised Learning

Oftentimes, finding relationships between data streams can be extremely difficult without advanced domain knowledge. However, these relationships can also be inferred directly from data using machine learning. This approach is useful for its flexibility and rapid development.

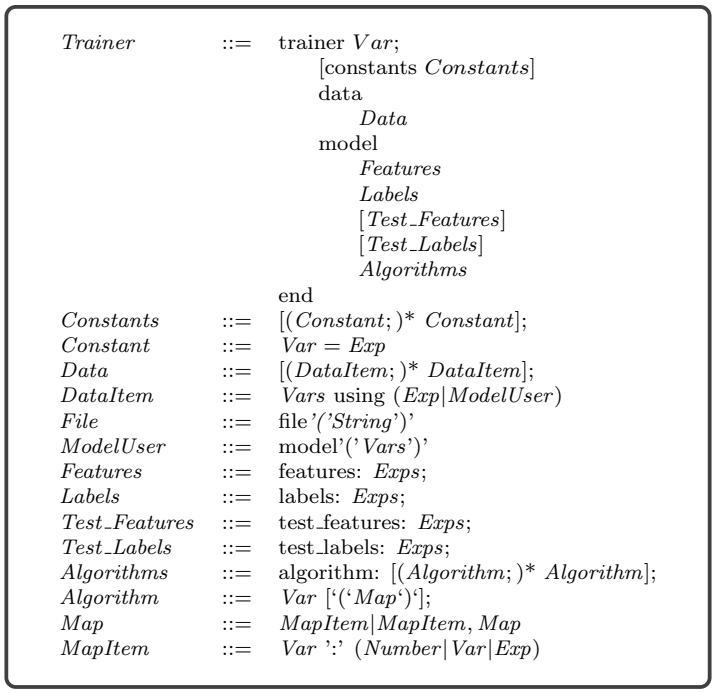
Machine learning has shown impressive results in many different fields from robotic dexterity [6] to board games [7]. We have implemented the grammar for training machine learning models, originally proposed by Imai et al. [2], making the system more modular and extensible for all users so that more machine learning algorithms can be easily incorporated.

This grammar, called trainers, allows for a declarative and extremely simple interface for supervised training of machine learning algorithms. The syntax of trainers are simple and consistent for every algorithm, allowing for the programmer and data scientist to focus on the application instead of the machine learning.

2.2.1 Trainer Language Abstraction

Machine learning models can either be trained offline or online. Offline training is completed before the use of the model in the application. Whereas online training is done in real-time while the model is being run. A model can be both trained in the offline phase while being updated during the online phase. The difference is in the implementation of the model and not in the syntax of its use. PILOTS trainers are used to train offline models as well create online models.

There are three main sections of a trainer file: `constants`, `data`, and `model`; Figure 2.4 shows the full grammar. The `constants` are an optional section that specifies constants. The `data` section is where data is collected to be sent to the model for training and validation. There are two ways to collect data: using previously created models and files. Comma-separated values (csv) files are the most common input method; the grammar also supports the selection of specific columns. Lastly, the model section specifies the data to be used to train, and possibly validate the model as well as specify any settings of the models. The `features` and `labels` subsections specify the inputs and the correct output values for the corresponding input. The model uses this information to train internal parameters. There is also an optional `test features` and `test labels`, which can be used to validate the accuracy of a trained model without needing to create another program to test it. Validating the model can ensure that the model has not over-fit on the training dataset. The last subsection of the model specifies the `algorithm` to use as well as specify any arguments to the model. The arguments can signify settings such as the learning rate of a neural network or what method of error calculation to use. Furthermore, if multiple models are listed then each of the models are trained and validated so the user can see which model resulted in the best performance on the given data.



```
Trainer        ::=  trainer Var;
                        [constants Constants]
                        data
                            Data
                        model
                            Features
                            Labels
                            [Test_Features]
                            [Test_Labels]
                            Algorithms
                    end
Constants      ::=  [(Constant; )* Constant];
Constant       ::=  Var = Exp
Data           ::=  [(DataItem; )* DataItem];
DataItem       ::=  Vars using (Exp|ModelUser)
File           ::=  file'('String')'
ModelUser      ::=  model'('Vars')'
Features       ::=  features: Exps;
Labels         ::=  labels: Exps;
Test_Features  ::=  test_features: Exps;
Test_Labels    ::=  test_labels: Exps;
Algorithms     ::=  algorithm: [(Algorithm; )* Algorithm];
Algorithm      ::=  Var ['('Map')'];
Map            ::=  MapItem|MapItem, Map
MapItem        ::=  Var ':' (Number|Var|Exp)
```

Figure 2.4: PILOTS trainer grammar.

2.2.2 PILOTS Architecture

Figure 2.5 shows the connection between the Machine Learning (ML) component and other PILOTS programs. Within the ML component, there are three types of files used to perform the operations: training data, algorithm implementations, and trained models. Training data consists of data used by models to update internal weights to reduce error in prediction. Model code files are implementations of machine learning algorithms written in Python using a specific API so it can used by the ML component. Lastly, trained models are serialized models that can be used by PILOTS programs to predict values. In summary, trainer files take model code files, pass them training data, and compile them into a trained model file which can be used by other PILOTS programs.

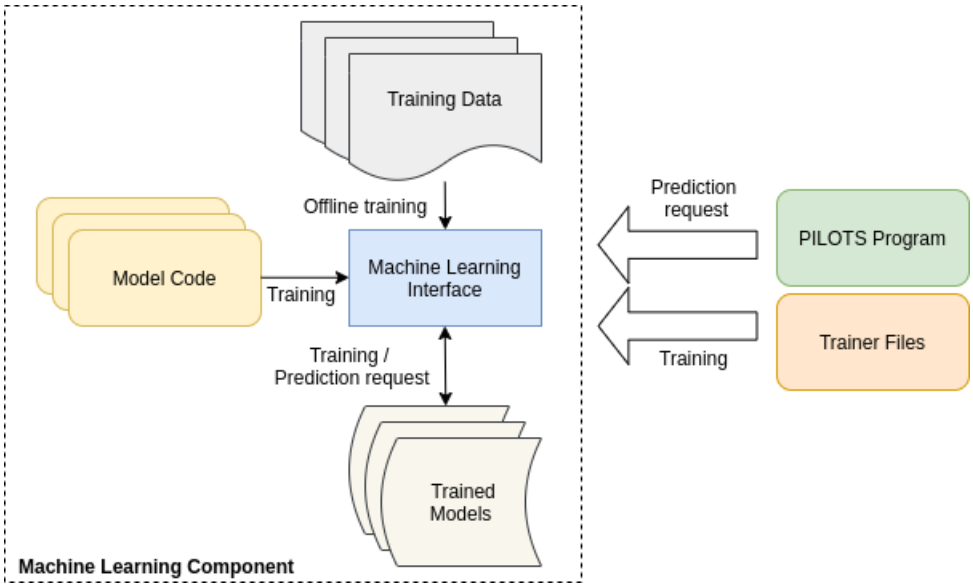


Figure 2.5: PILOTS and machine learning interface.

The machine learning component consists of an HTTP server which responds to requests from the PILOTS trainers and PILOTS programs.

Model code is the term used to describe the Python implementation of the machine learning algorithms. As long as the Python source code implements certain Application Programming Interface (API) functions, seen in Table 2.1, it is considered a valid model code file. This means any type of Python library or program with Python bindings can be used within PILOTS with ease.

Model files are the serialized representation of a trained model. They are created as the result of running a trainer file and can be used by any PILOTS program.

Table 2.1: Python API for implementing a machine learning algorithm.

Method	Arguments	Returns	Description
`make_model`	map of arguments	new model	Creates a new model with the given settings
`train`	untrained model, data	trained model, accuracy	Train the given model using the given data and return the trained model and the accuracy of running the trained model on the training set
`score`	trained model, data	accuracy	Return the average accuracy of running the trained model on the given data
`run`	trained model, data	result	Get the result of running the trained model on the given data
`is_live`	None	boolean	Return true if the model is an online model and is updated at runtime and false otherwise
`reset`	None	None	Resets the online model to a state that is ready to receive new data

The PILOTS compiler translates PILOTS programs and trainer files into Java for platform independent execution. The compiler consists of two parts: a parser and a code generator, each of which has a version meant for standard PILOTS programs and one for trainer files. The parser uses JavaCC[3] to convert the grammar into an abstract syntax tree, which the code generator then converts into a valid Java program.

Once a trainer file is compiled into the Java source file, it is compiled using the standard Java compiler. When the program is ran in conjunction with the machine learning component server, it creates a model file which can be used in other trainer files or PILOTS applications.

2.2.3 Applications

A simple example of a program that uses a machine learning model can be seen in Figure 2.6: the Twice program but using a linear regression model to estimate the b data stream [2]. It uses the a data stream as input to the "twice_model". The other input data stream, b, is twice the value of a and the output data stream o outputs the error of the model's estimate of b. This model was trained using the trainer file shown in Figure 2.7.

This trainer program trains a linear regression model using the columns a and b from a training dataset called "twice_training.csv". In this data, b, is approximately twice that of a. The linear regression model was also specified to use the ordinary least squares method.

Figure 2.8 shows the twice_model trainer when it is compiled into Java.

```
program prediction_twice;
  inputs
    a, b (t) using closest (t);
    b_prime using model(twice_model, a);
  outputs
    o: b - b_prime at every 1 sec;
end
```

Figure 2.6: PILOTS prediction_twice program.

[3]https://javacc.github.io/javacc/

```
trainer twice_model;
  data
    a, b using file("twice_training.csv");
  model
    features: a;
    labels: b;
    algorithm:
      LinearRegression(OrdLeastSq: true);
end;
```

Figure 2.7: PILOTS twice_model trainer program.

A more advanced trainer which trains a neural network to learn a three input XOR gate can be seen in Figure 2.9. It collects columns a, b, c, and x from the "data" file, where they are listed as columns "A", "B", "C", and "Q" in the file. Also $i1$, $i2$, $i3$, and o are collected from the "xor" file. The three neural network models using slightly differing settings are trained using a, b, and c as inputs and x as the output. Each model is also validated with the testing data given by $i1$, $i2$, $i3$, and o. The program will return the final results of training all of the models so that the user can see which model was most effective on the dataset. Training multiple models is no different in PILOTS which means that users can quickly try multiple models to see what works best on the data. These four models resulted in final accuracies of 89%, 0.1%, 99.9%, and 1.0%, respectively. These results shows several things very quickly: that the hyperbolic tangent and logistic activation functions severely decrease the accuracy of the model (comparing models 1 to 2, and models 3 to 4) and that increasing from one layer to two increased overall accuracy (comparing model 1 to 3).

```
import java.io.*;
import java.util.*;

import pilots.util.trainer.*;
import pilots.util.model.*;

public class Twice_model extends PilotsTrainer {

    public Twice_model() {
        super();

        super.model_name = "twice_model";
        super.addAlgorithm("LinearRegression");

        setupTrainer();
    }

    public void train() { super.train(); }

    private void setupTrainer() {
        // === Constants ===

        // === Data ===
        super.pullData( "file:twice_training.csv::a,b" );

        // === Model ===
        addFeature( super.get("a") );

        addLabel( super.get("b") );


        addAlgArg( 0, "OrdLeastSq", true);
    }

    public static void main(String[] args) {
        Twice_model tr = new Twice_model();
        tr.train();
    }
}
```

Figure 2.8: PILOTS trainer compiled into Java.

```
trainer xor_model;
  data
    a, b, c, x using file("data.csv", A, B, C, Q);
    i1, i2, i3, o using file("xor.csv");
  model
    features: a, b, c;
    labels: x;
    test_features: i1, i2, i3;
    test_labels: o;
    algorithm:
      NeuralNetwork(hidden_nodes: 100, hidden_layers: 1);
      NeuralNetwork(hidden_nodes: 100, hidden_layers: 1,
                    logistic: true);
      NeuralNetwork(hidden_nodes: 50, hidden_layers: 2);
      NeuralNetwork(hidden_nodes: 50, hidden_layers: 2, tanh: true);
end;
```

Figure 2.9: Training multiple models to learn XOR.

2.3 Distributed Model Training and Execution

Aircraft are highly distributed and each part may only have limited local computing ability. Therefore, it is useful to be able to leverage this network of systems to train or execute machine learning models.

One such method for distributed machine learning is federated learning. One benefit of federated learning is that each device does not need to share its local training data, which decreases privacy and security risks [8]. Federated learning trains models on heterogeneous training data and aggregates the results into a single model. One application of federated learning is in Google's keyboard application, Gboard, which trains a model for word prediction without sharing users' data with a central server [9]. We will present a declarative grammar for federated learning within PILOTS trainers.

Just as it can be useful to collect data from various sources, strategically aggregating the results of several models can be very beneficial. Ensemble systems combine results from several models to outperform any one model. One such example is the distributed Shogi (Japanese Chess) engine Akara which uses majority voting system between four separate programs to come up with a move [10]. Akara was able to beat a professional Shogi player in a public match. Xu-rui et al. created an ensemble learning algorithm to improve probabilistic aircraft conflict detection [11]. Their ensemble model was superior to a single model in

detection accuracy and false alarm rate. We propose several data selection methods within PILOTS to allow for the aggregation of results from multiple models.

2.3.1 Federated Learning

The process of a single round of federated learning would be as shown in Figure 2.10. Each node trains the local model on the local training data and sends the resulting internal training parameters to the central model. The central model aggregates the results and creates a new global model. Each client then updates the local model with the new global model and the process continues.

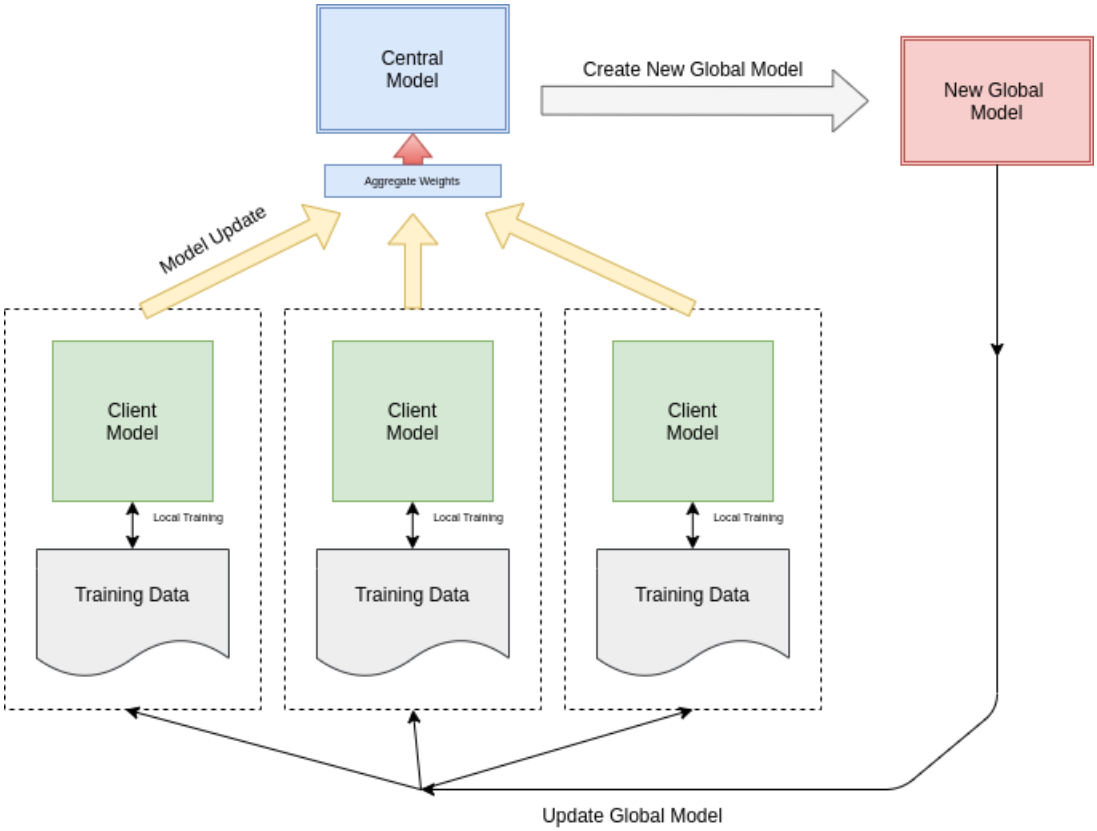


Figure 2.10: Federated learning round process.

Here we propose an extension of the trainer grammar which supports declarative model-agnostic federated learning, Figure 2.12. The federated trainer consists of one additional section to a standard trainer file, the federated section, which specifies the details of the federated training process. The user specifies the maximum number of total rounds to run, the number of nodes to use during each round, how to select nodes from the list of available

clients, and the method used to aggregate the values from all the clients. Possible selection methods for clients could include *fastest* which chooses the nodes with the best latency, *random* which chooses randomly for each round, and *priority* which gives each node a user-defined priority. Each node is an instance of the PILOTS machine learning server. Possible aggregation methods can include federated stochastic gradient descent, or *FedSGD*, which is a large-batch synchronous variation of stochastic gradient descent (SGD), *FedAvg* is another variation on SGD, as well as *custom* where the system uses a user defined function [8]. This declarative grammar gives users the functionality of federated learning without dealing with the details of communication and control flow. An example of a federated trainer for a neural network is shown in Figure 2.11. Using federated learning, aircraft could train models without contact with ground stations. For example, the weight estimation model discussed later in this paper can be trained using each aircraft's speed curve at take-off.

```
trainer fed_model;
  data:
    a, b, c, x using file("data.csv");
  federated
    rounds: 20
    nodes: 10
    selection: random
    aggregation: FedAvg
  model
    features: a, b, c;
    labels: x;
    algorithm:
      NeuralNetwork(hidden_nodes: 50, hidden_layers: 6);
end;
```

Figure 2.11: Federated PILOTS trainer.

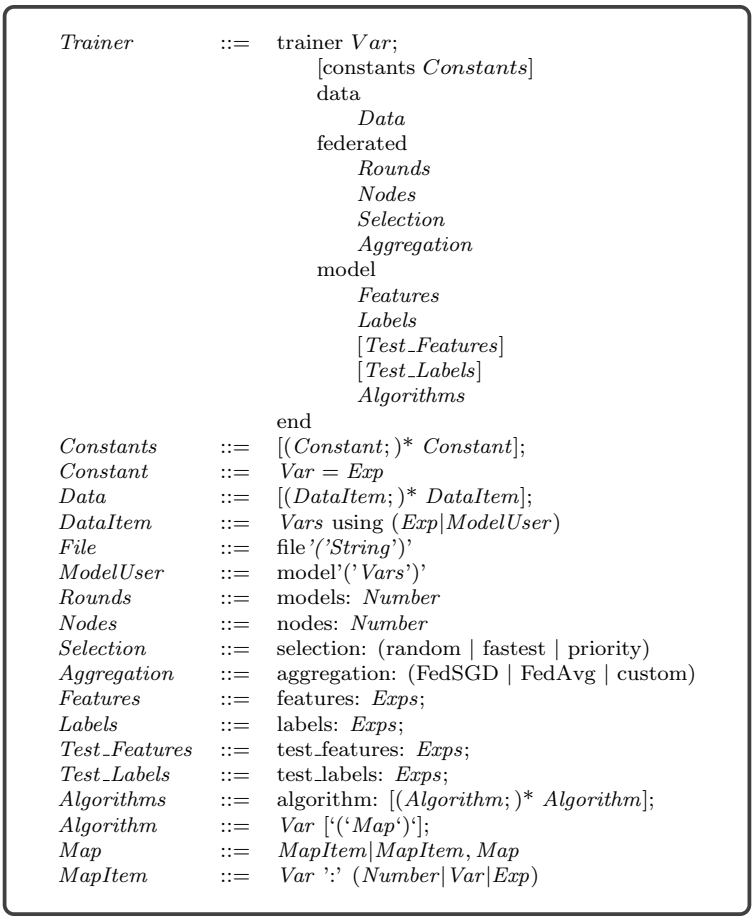


Figure 2.12: PILOTS federated trainer grammar design.

2.3.2 Model Aggregation

We propose several methods for aggregating the results of multiple models, *majority*, *weighted*, *average*, and *decision*. The full extended grammar is shown in Figure 2.14. An example of these methods can be seen in Figure 2.13, where multiple models estimating density altitude, altitude relative to standard atmospheric conditions, are aggregated. *Majority* takes three or more results and returns the majority if one exists; otherwise it returns the value signified by *none*. *Weighted* takes the weighted average of each input. The *average* method takes the average of the given values. Lastly, *decision* uses a boolean expression to determine when to use which value. These ensemble methods provide a simple way to combine the results of related models. In the following chapter, we investigate the use of an ensemble model for weight estimation.

```
program ensemble;
  constants
    w1 = 0.2;
    w2 = 0.5;
    w3 = 0.3;
  inputs
    // Pressure, temperature, and altitude
    prs, tmp, alt (t) using closest (t);
    // Get prediction from each model
    da1 using model(da_model1, prs, tmp, alt);
    da2 using model(da_model2, prs, tmp, alt);
    da3 using model(da_model3, prs, tmp, alt);
    // Combine results
    A using average(da1, da2, da3);
    M using majority(da1, da2, da3, none:A);
    W using weighted(w1:da1, w2:da2, w3:da3);
    D using decision( (prs < 29.00):da1,
              (29.00 < prs < 30.00):da2,
                      (30.00 < prs):da3 );
  outputs
    A, M, W, D at every 1 sec;
end;
```

Figure 2.13: PILOTS program displaying proposed ensemble methods.

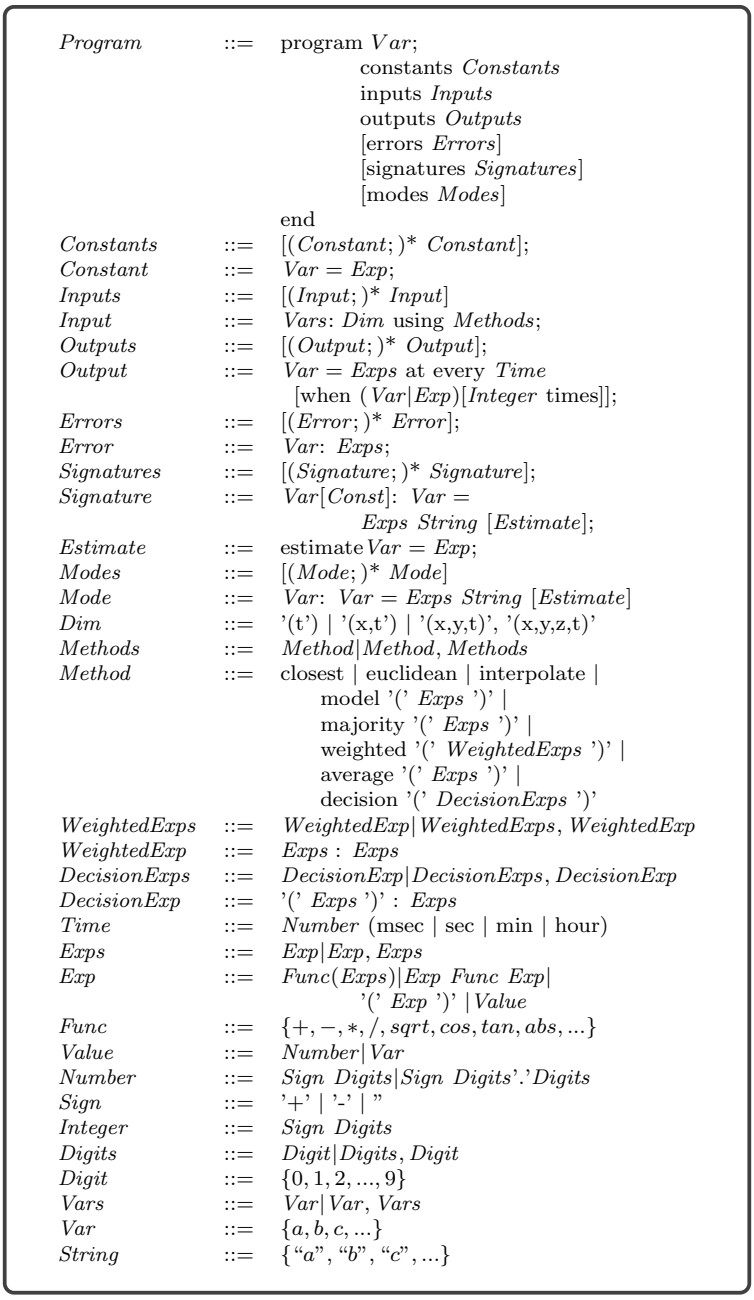


Figure 2.14: PILOTS grammar design with the proposed ensemble methods.

CHAPTER 3
Aircraft Weight Estimation During Take-off

3.1 Background

During take-off an aircraft is accelerating due to thrust and being slowed from friction with the air and the runway. Knowing these forces and the aerodynamic properties of the aircraft we can calculate the mass using Newton's second law of motion.

$$m = \frac{T - D - F}{dV/dt} \tag{3.1}$$

T is thrust, D is aerodynamic drag, F is rolling resistance, and V is airspeed. The complexity in this approach is that each force can be difficult to compute.

Thrust of the aircraft depends on air density, propeller pitch, propeller diameter, propeller rpm and airspeed. There is no simple equation to calculate it directly, and is often looked up in tables.

Aerodynamic drag, equation 3.2, depends on the coefficient of drag C_D, air density ρ, airspeed V, and cross sectional area S.

$$D = \frac{C_D * \rho * V^2 * S}{2} \tag{3.2}$$

Lastly, rolling resistance, equation 3.3, is dependent on the coefficient of friction between the landing gear and the runway surface μ, and the force being applied to the runway which is result of weight W and lift L.

$$F = \mu(W - L) \tag{3.3}$$

Whereas the other forces can be accurately calculated using properties of the aircraft, rolling resistance poses an issue. Different runway surfaces and weather conditions will change the friction between the landing gear and the runway. More importantly, the dependency on weight to calculate rolling resistance makes the physics approach more difficult during take-off. We will discuss a data-driven model which can estimate aircraft weight using only previous take-off trials that does not require knowledge of the aerodynamic properties

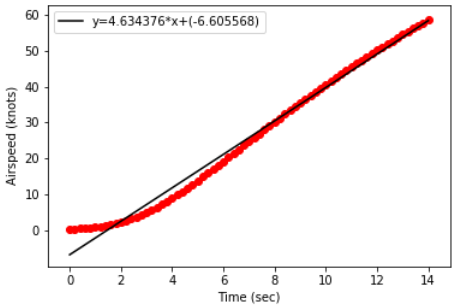
of an aircraft.

3.2 Data-driven Weight Estimation

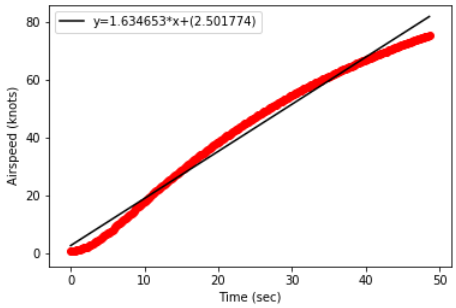
If an aircraft is heavier, then it takes longer to take-off, assuming all other variables are held constant. This can seen in practice and in theory. This is because a heavy aircraft, given the same engine thrust, takes longer to accelerate to take-off speed. The model that will be discussed in this chapter uses the relationship between weight and acceleration to estimate weight. Furthermore, it only uses data that is available from basic avionics on an aircraft such as the Cessna 172. This means that this model can be used in nearly any aircraft, as long as it has an accurate measurement of airspeed and atmospheric conditions such as temperature, air pressure, and altitude.

Looking at the airspeed over time gives a curve. Figure 3.1 shows the linear approximations of the velocity curve. As can be seen in Figure 3.1a, the curve takes time at the beginning until the curve accelerates at a roughly constant rate because during this time the propeller is getting up to speed. Due to this, it is possible to cut off the lowest airspeed values to wait until the propeller is up to speed, which gives a better linear approximation as seen in Figure 3.2. Higher order approximations, see Figure 3.3, can be used for more accurate approximations of acceleration; however, these approximations do not allow for the interpolation methods used within this model and their inclusion into the model will be looked into in the future. The acceleration of the aircraft would be approximated to the derivative of the fitted curve. As mentioned previously, if the aircraft is heavier then it takes longer to take-off, because the heavier aircraft has less acceleration. This relationship between weight and the velocity curve can be shown in Figures 3.4 for fixed atmospheric conditions.

Furthermore, in real-life situations, aircraft do not take-off in identical atmospheric conditions. Even within a single day, weather conditions can change greatly, so it is important to be able to account for atmospheric conditions. This model uses density altitude, which is defined as the altitude with respect to standard atmospheric conditions, as an encapsulation of all atmospheric properties since it includes air pressure, temperature, and altitude within a single value. However, we are disregarding the effect of humidity and wind. For a fixed weight, Figure 3.5 shows the effect of density altitude on the take-off velocity curve.

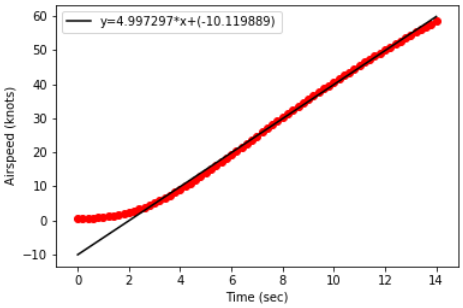


(a) Trial with density altitude of -1,809ft and weight of 1,811lbs.

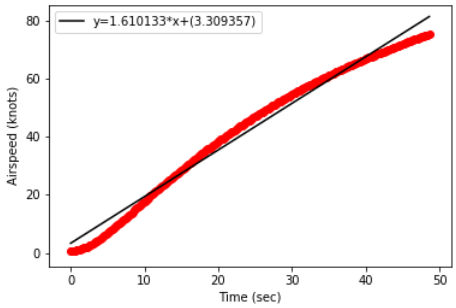


(b) Trial with density altitude of 6,551ft and weight of 2,934lbs.

Figure 3.1: Linear approximation of velocity curve.

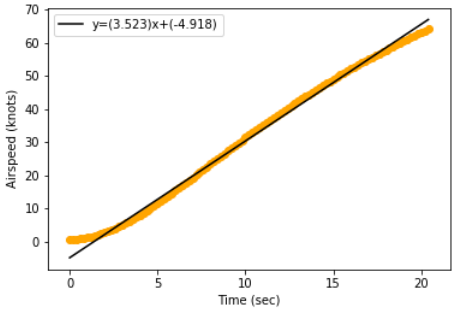


(a) Trial with density altitude of -1,809ft and weight of 1,811lbs.

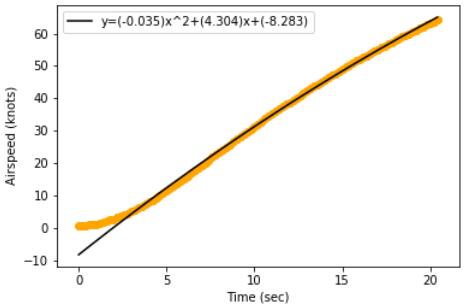


(b) Trial with density altitude of 6,551ft and weight of 2,934lbs.

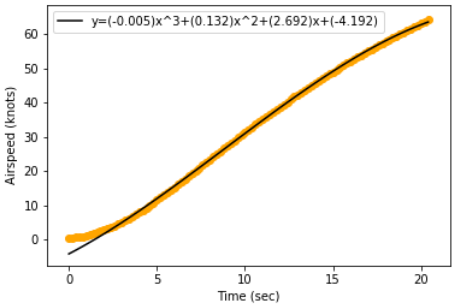
Figure 3.2: Linear approximation of velocity curve with a cutoff below 2.5 knots.



(a) First order approximation

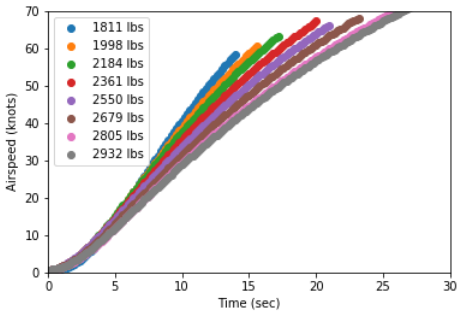


(b) Second order approximation

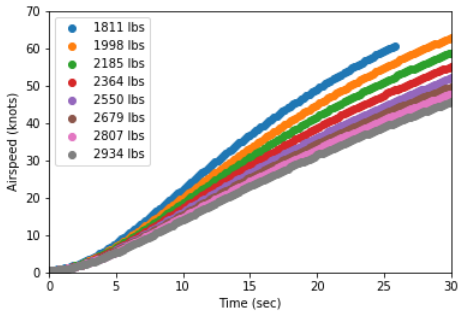


(c) Third order approximation

Figure 3.3: First, second, and third order approximation of velocity curve of a trial with density altitude of 4155ft and weight of 1,811lbs.

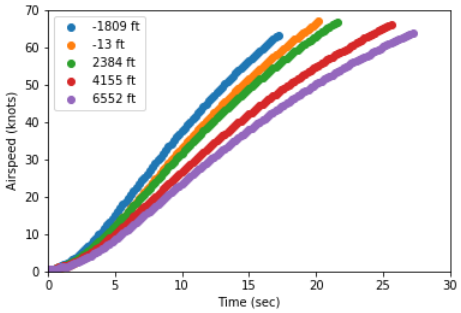


(a) At density altitude of -1,809ft.

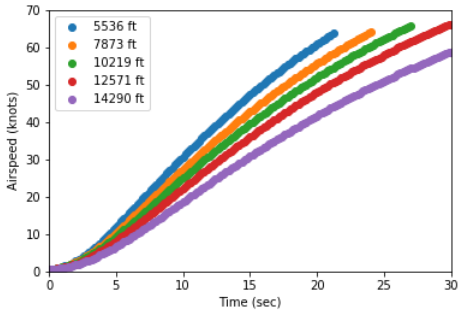


(b) At density altitude of 14,291ft.

Figure 3.4: Effect of weight on velocity curve.



(a) At weight of 2181lbs at KALB.



(b) At weight of 2181lbs at KRNO.

Figure 3.5: Effect of density altitude on velocity curve.

3.3 Weight Estimation

In this section, we will discuss how the weight estimation model predicts the weight of an unknown trial. Including how the model uses the training data, and interpolates between training data points to get a final estimate.

3.3.1 Density Altitude

Density altitude represents what altitude the current conditions would be at in standard atmospheric pressure and temperature. The calculation used assumes completely dry air (0% humidity). Similar to density altitude, pressure altitude (PA) is the altitude with respect to standard air pressure, disregarding the effect of temperature.

$$PA = \text{Elevation} + 1000 \text{ ft} * (29.92 \text{ inHG} - \text{Pressure}) \tag{3.4}$$

Elevation is the altitude of the aircraft above mean sea level in feet and pressure is the atmospheric pressure at the location of the aircraft in inches of mercury.

$$ISA = 15°\text{C} - (1.98°\text{C}) * \left(\frac{PA}{1000 \text{ ft}}\right) \tag{3.5}$$

International Standard Atmosphere (ISA) temperature is the temperature at a specific altitude in standard atmospheric conditions.

$$DA = PA + (118.8 \text{ ft}/°\text{C}) * (OAT - ISA) \tag{3.6}$$

Outside air temperature (OAT) is the current air temperature. Finally, to calculate density altitude, we use equation 3.6 for its ease of calculation and very good approximation of true density altitude which is sufficient for our model.

3.3.2 Supervised Training

Training data of this model consists of take-off trials labelled with the true weight. Each take-off trial consists of atmospheric conditions (temperature, pressure, and altitude) which are used to calculate the density altitude and the velocity throughout the take-off which is used to estimate the acceleration of the aircraft. The stored data after training consists of three values: density altitude, estimated acceleration, and true weight for each take-off trial.

3.3.3 Interpolation

Once all training data is collected, the model calculates two values used to interpolate the final result. The two values represent the effect that a change of density altitude has on the acceleration, and the effect that a change in acceleration has on the weight. These two values are called δ_{da} and δ_{acc}, given by equations 3.7 and 3.8.

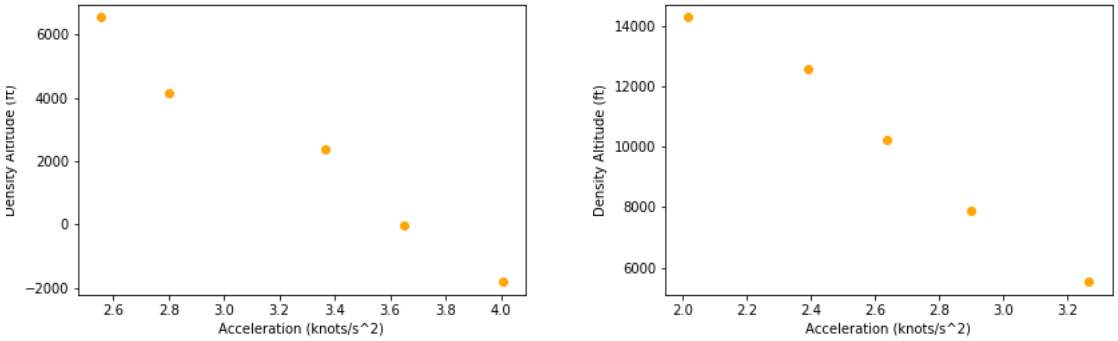
$$\delta_{\text{da}} = \frac{\Delta\text{acceleration}}{\Delta\text{density altitude}} \tag{3.7}$$

$$\delta_{\text{acc}} = \frac{\Delta\text{weight}}{\Delta\text{acceleration}} \tag{3.8}$$

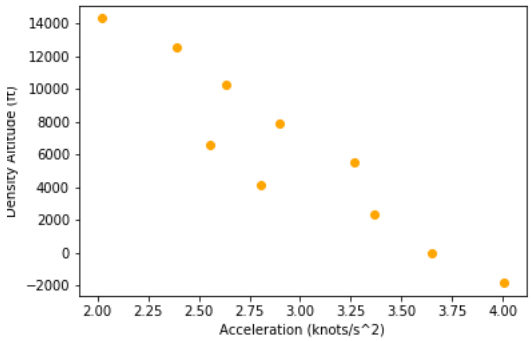
These values can be understood as quantifying the relationships shown in Figures 3.5 and 3.4, respectively. However, instead of showing the relationship in terms of the velocity curve, it is with respect to the approximated acceleration. Figure 3.6 shows the relationship between density altitude and acceleration and Figure 3.7 shows the relationship between weight and acceleration. The δ_{da} is the slope of the linear approximation of the relationship shown in Figure 3.6 and, similarly, δ_{acc} is the inverse of the slope of the linear approximation of Figure 3.7.

To calculate δ_{da}, the model finds the average of values of δ_{da} between pairs of training data points with similar weights, $\pm 0.1\%$. Similarly, to calculate δ_{acc}, the model finds the average values of δ_{acc} between pairs of points with similar density altitudes, $\pm 0.1\%$. If the

dataset does not have enough data points with similar weights and density altitudes to calculate δ_{acc} and δ_{da}, then the model will not be able to accurately estimate weight. This can be solved by adding structure to data collection by collecting data using specific weights and density altitudes, or increasing the size of the dataset.



(a) KALB and KRNO trials, respectively.



(b) Complete dataset.

Figure 3.6: Relationship between density altitude and estimated acceleration.

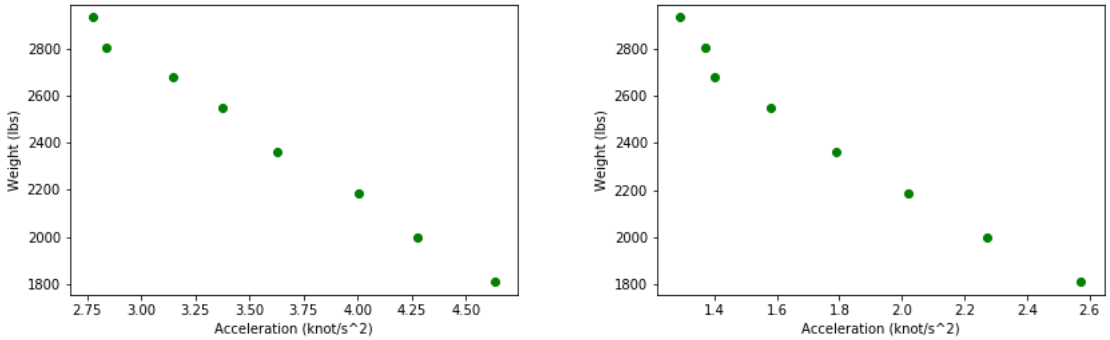


Figure 3.7: Relationship between aircraft weight and estimated acceleration for density altitudes of -1,809ft and 14,291ft, respectively.

An example PILOTS program to train a weight estimation model is shown in Figure 3.8. This program trains a model using data from KRNO. Furthermore, the model does not use a cutoff of any data and uses data points with density altitude values within ±0.5% to calculate δ_{acc} and uses data points with weights within ±0.1% to calculate δ_{da}.

```
trainer weight_model_krno;
  data
    // Airspeed, pressure, temperature,
    // altitude, and actual weight
    va, prs, tmp, alt, curr_w using
      file("data_krno.csv", );
  model
    features: va, prs, tmp, alt;
    labels: curr_w;
    algorithm:
      WeightEstimator( cutoff: 0.0, da_close: 0.5, w_close: 0.1 );
end;
```

Figure 3.8: Weight estimation model trainer.

3.3.4 Weight Prediction

When the model is given new data with an unknown weight, it calculates the density altitude and the acceleration from the slope of the velocity curve. It then finds the closest stored data point, closest being the point that minimizes the difference in the density altitudes and the difference in the acceleration values as in equation 3.9. The model then takes that data point and interpolates to get a final estimate.

$$e = (\text{DA}_1 - \text{DA}_2)^2 + (a_1 - a_2)^2 \tag{3.9}$$

Given the closest stored data point from the training data, equation 3.10 shows how to interpolate this value to improve accuracy.

$$\text{weight} = (\text{closest weight}) + (\delta_{\text{acc}} * (\delta_{\text{da}} * E_{\text{da}} - E_{\text{acc}})) \tag{3.10}$$

Error in density altitude, E_{da}, is the difference between the closest point's density altitude and the input's density altitude. Similarly, error in acceleration, E_{acc}, is the difference in the two data points' accelerations.

Multiplying E_{da} and δ_{da} gives the error in the acceleration due to the density altitude.

Therefore, adding the value with E_{acc} gives us the total error in acceleration. Note, the negative sign on E_{acc} is due to the inverse relationship between the error in acceleration and weight. Meaning a positive value of E_{acc} would result in a negative adjustment in the weight. Lastly, multiplying the total error in the acceleration by δ_{acc} gives the weight adjustment due to the error in the slope, which gives us the updated final weight estimate.

3.4 Experimental Results

3.4.1 Albany, NY and Reno, NV Airports

The data was collected in X-Plane 9 using an Cessna 172SP. Data streams such as airspeed, temperature, air pressure, and altitude were collected during standard procedure[4] take-offs at two airports: Albany International Aiport (KALB) and Renoe-Tahoe International Aiport (KRNO), using a selection of density altitudes given in Figure 3.1, and a selection of weight classes listed in Table 3.2. There are 80 total take-off trials in the training dataset.

Table 3.1: Density altitudes collected at each runway (feet above mean sea level).

KALB	**KRNO**
-1,809	5,336
-13	7,873
2,384	10,219
4,155	12,571
6,552	14,290

The density altitude values for each airport are a selection of atmospheric conditions that could happen in that location.

[4]According to Cessna 172 Information Manual [12]

Table 3.2: Weight classifications.

Classification	**Cargo**	**Fuel (hrs)**[5]	**Total (lbs)**
LOW	123	1	1811
LOW-MED	261	2	1998
MED	400	3	2181
MED-HIGH	460	5.5	2360
MAX	596	6.6	2550
105% MAX	725	6.6	2679
110% MAX	852	6.6	2806
115% MAX	979	6.6	2933

These weight classes were selected as a uniform partitioning of weights that represent a reasonable possible low-end take-off weight up to a very extreme overweight condition.

The model was validated using a wide selection of weights and density altitudes, as can be seen in table 3.3. This included take-offs from KALB, KRNO, and Minot International Airport (KMOT). KMOT was used to see how well the model can be used on runways that the model had no training with.

[5] 1 hour of fuel weights $\tilde{4}8$ lbs

Table 3.3: Dataset of testing trials.

Trial Name	**Weight (lbs)**	**Density Alt (ft)**
KALB_1	2076	-385
KALB_2	2484	1393
KALB_3	2169	2105
KALB_4	2093	59
KALB_5	2458	59
KALB_6	2717	59
KALB_7	1859	1245
KALB_8	2181	1245
KALB_9	2549	1245
KALB_H1	3001	59
KALB_H2	3001	1245
KRNO_1	2256	8247
KRNO_2	2608	6512
KRNO_3	1949	11441
KRNO_4	1916	7970
KRNO_5	2192	7970
KRNO_6	2799	7970
KRNO_7	1810	9710
KRNO_8	2181	9710
KRNO_9	2679	9710
KRNO_10	1972	12620
KRNO_11	2364	12620
KRNO_12	2399	11454
KRNO_13	1998	11454
KRNO_H1	3001	10290
KRNO_H2	3001	11454
KMOT_1	2817	2521
KMOT_2	2083	5075
KMOT_3	2267	9243

The results of five different models were tested. There are two models trained using data from only one airport (KALB and KRNO), to see how the model can transfer from one airport to another without direct knowledge. Referred to as the KALB and KRNO models. Another two models are trained on the complete dataset, one of which ignores all airspeed values below 2.5 knots to improve the linearity of the velocity curve (cutoff model). The other, which ignores no values, is referred to as the total model. Lastly, there is an ensemble model that aggregates the results from the two models trained only on one airport. This model uses the average density altitude between all training trials as a transition point (approximately 6000ft). When given a density altitude below this point the model uses the KALB model and above which uses the KRNO model. Figure 3.9 shows PILOTS program which creates the ensemble model.

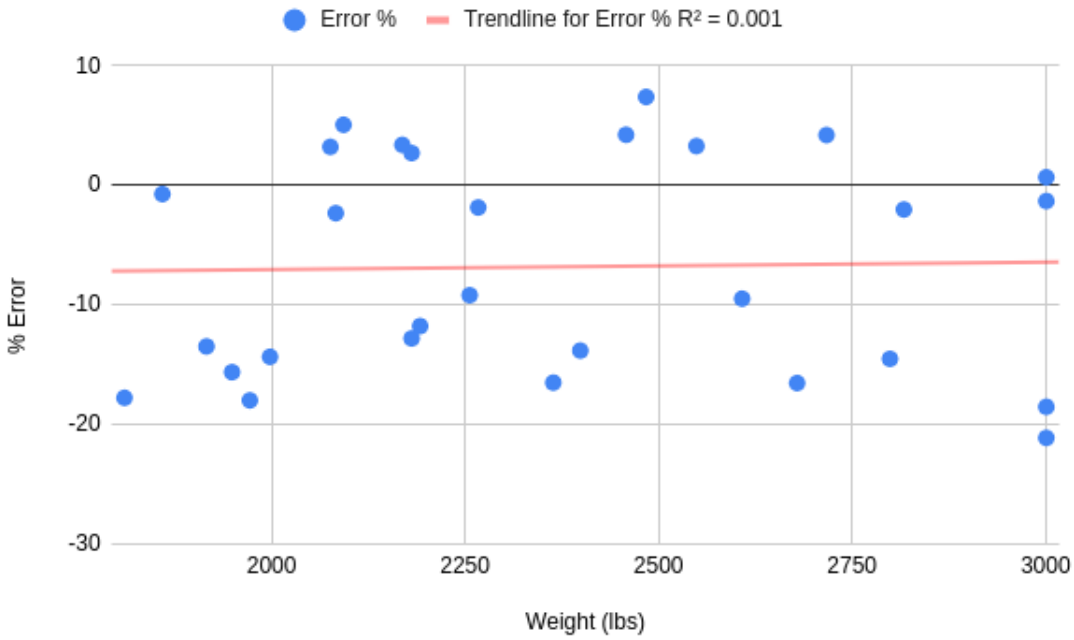
```
program ensemble_estimator;
  inputs
    // Airspeed, temperature, air pressure,
    // altitude above sea level, and actual weight
    v_a, prs, tmp, alt, curr_w (t) using closest(t);
    // Calculate Density Altitude
    dens_alt using model(density_altitude, prs, tmp, alt);
    // Get estimates from two models
    est_w_kalb using model(weight_model_kalb, v_a, dens_alt);
    est_w_krno using model(weight_model_krno, v_a, dens_alt);
    // Decide which model to use
    est_w using decision( (dens_alt < 6000):est_w_kalb,
                          (dens_alt >= 6000):est_w_krno );
  outputs
    v_a, curr_w, est_w at every 200 msec;
end;
```

Figure 3.9: Ensemble weight estimator PILOTS program.

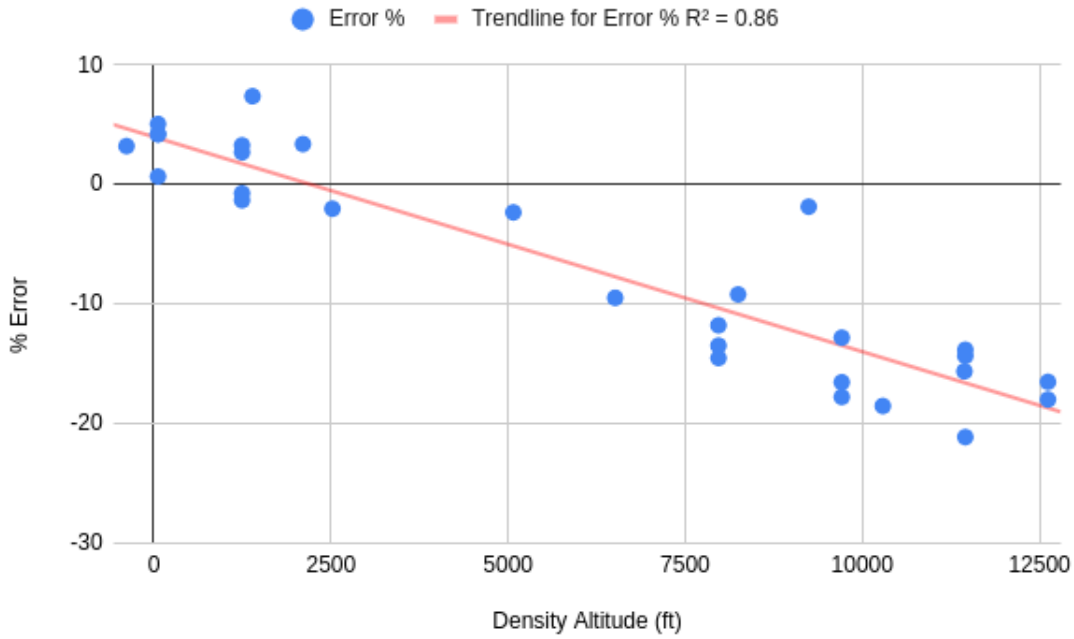
Table 3.4 shows the average final results of each model. This table also shows that some models performed better on certain runways than others. As can be seen in Table 3.4, the KALB model underestimated estimates on KRNO trials and the KRNO model overestimated on KALB trials. This is likely due to some differences between the two runways that is not properly accounted for by the model, such as runway slope or surface. Furthermore, this is likely the cause of the correlation between error and density altitude seen in Figures 3.10 and 3.11. The performance of the cutoff model shows that removing the non-linearity of the start of the velocity curve does not improve performance of the model. Lastly, the ensemble model outperforms the other models. It can also be seen that the ensemble model outperforms all other models.

Table 3.4: Average results based on runway of testing trial.

Testing Airport	**KALB Model**	**KRNO Model**	**Total Model**	**Cutoff Model**	**Ensemble Model**
KALB Accuracy	3.22	13.00	10.24	11.63	3.32
KRNO Accuracy	-14.90	1.10	1.51	4.27	2.26
KMOT Accuracy	-2.07	12.34	11.63	12.98	4.68
Overall Average	-6.81	6.78	5.87	7.96	2.9

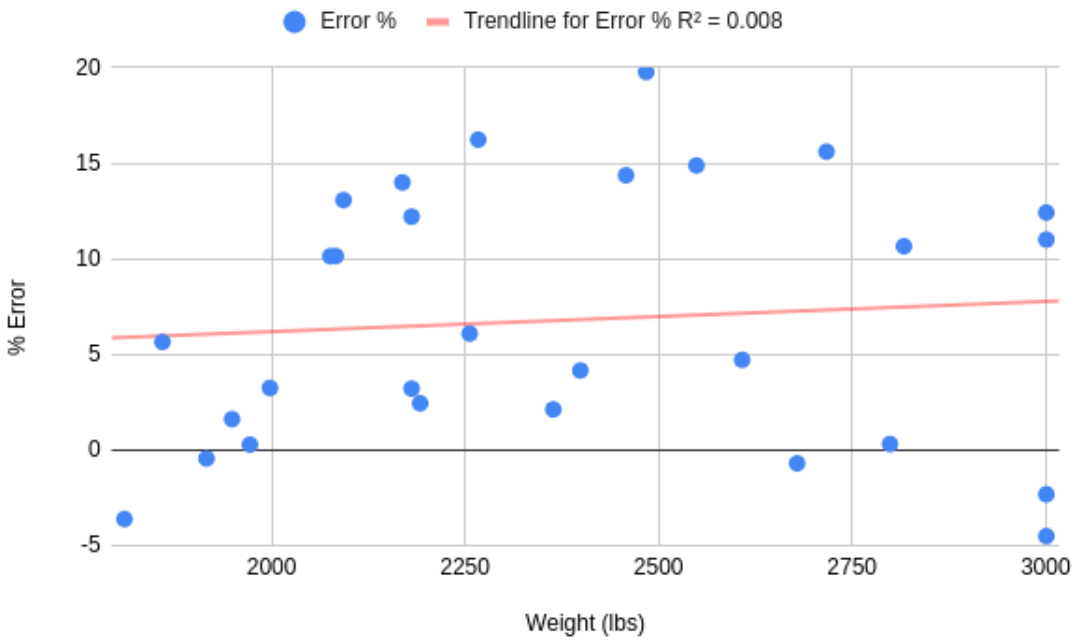


(a) Relation between error and weight.

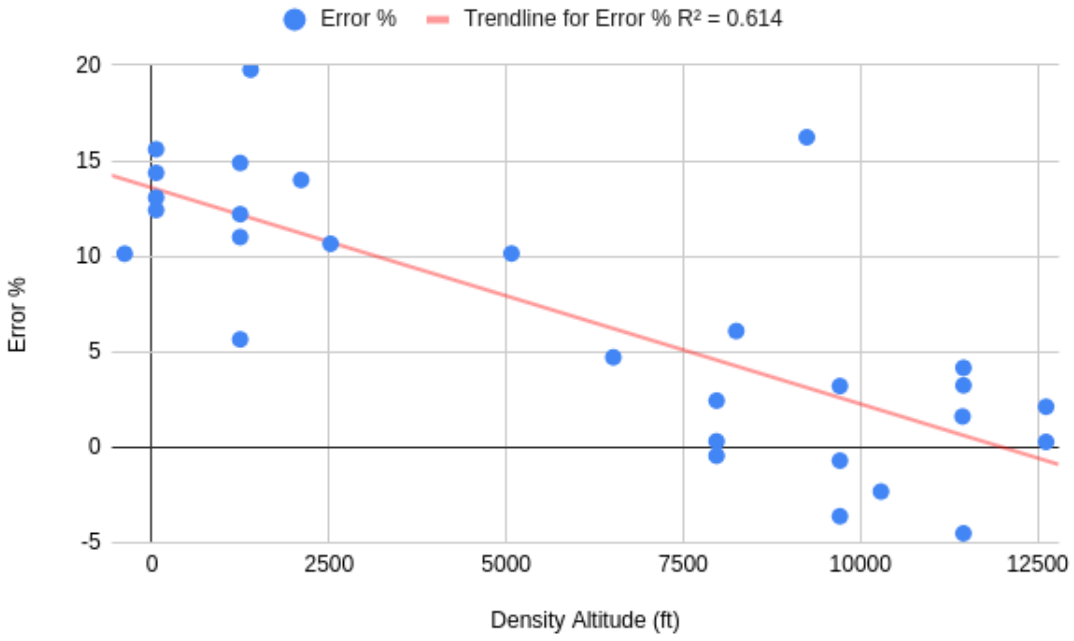


(b) Relation between error and density altitude.

Figure 3.10: Results of the KALB model.

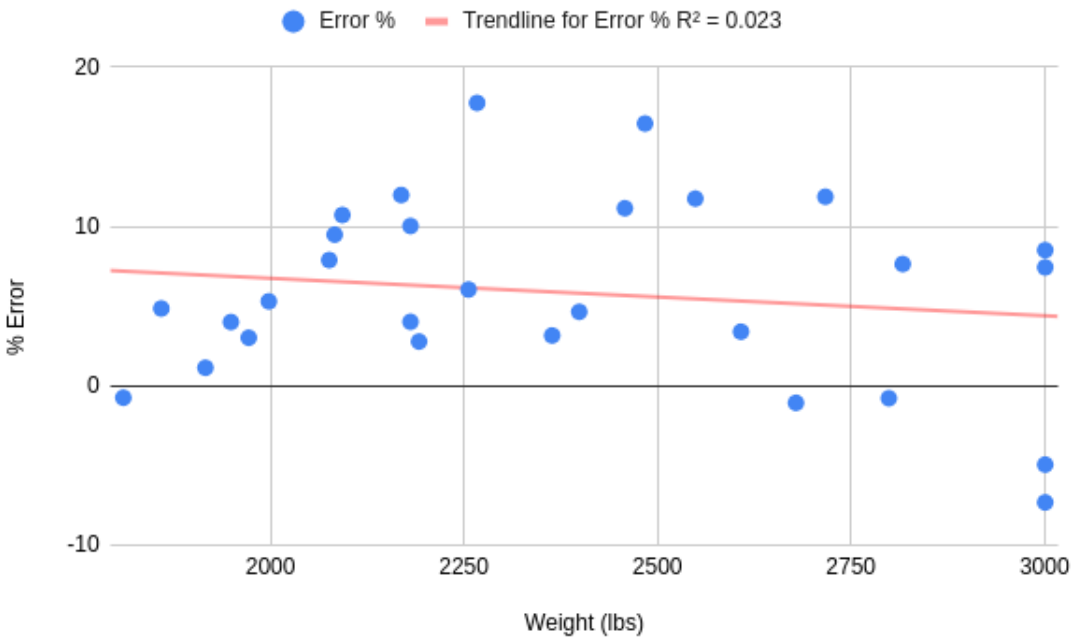


(a) Relation between error and weight.

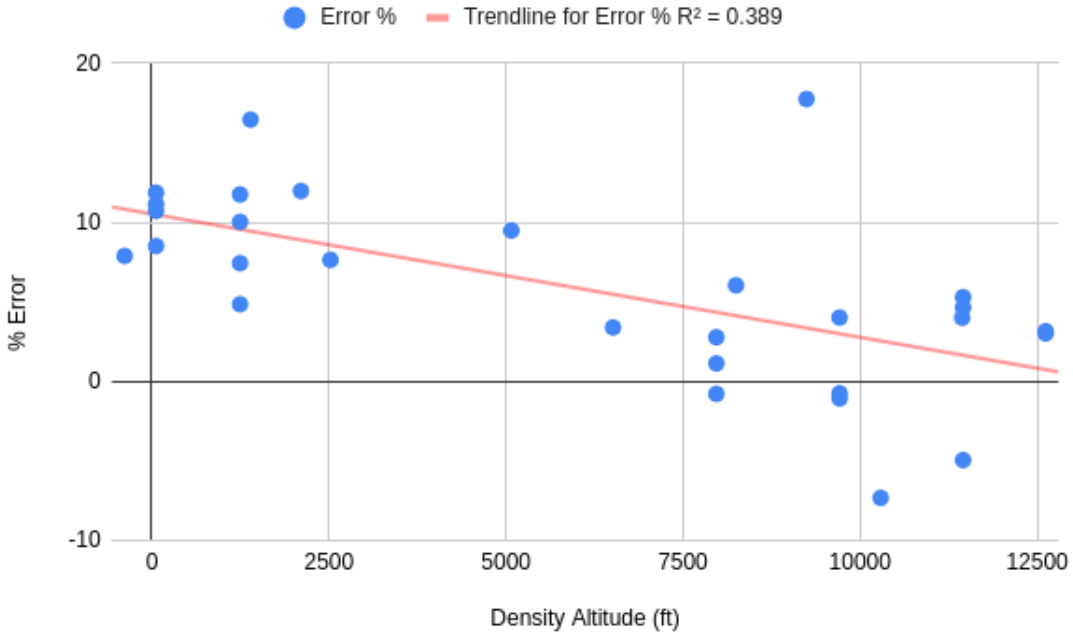


(b) Relation between error and density altitude.

Figure 3.11: Results of the KRNO model.

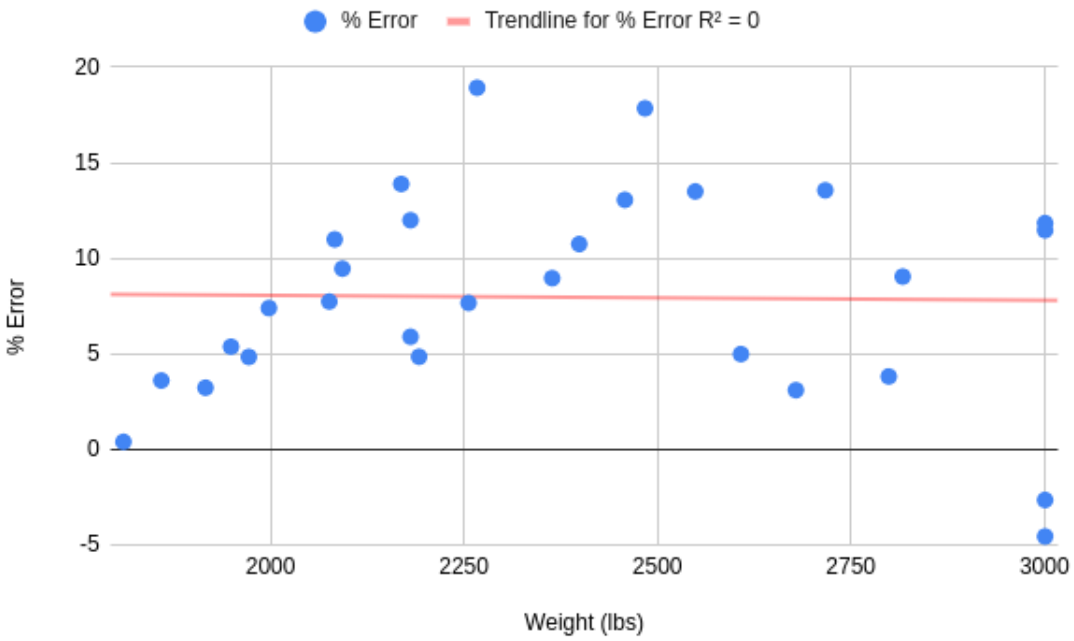


(a) Relation between error and weight.

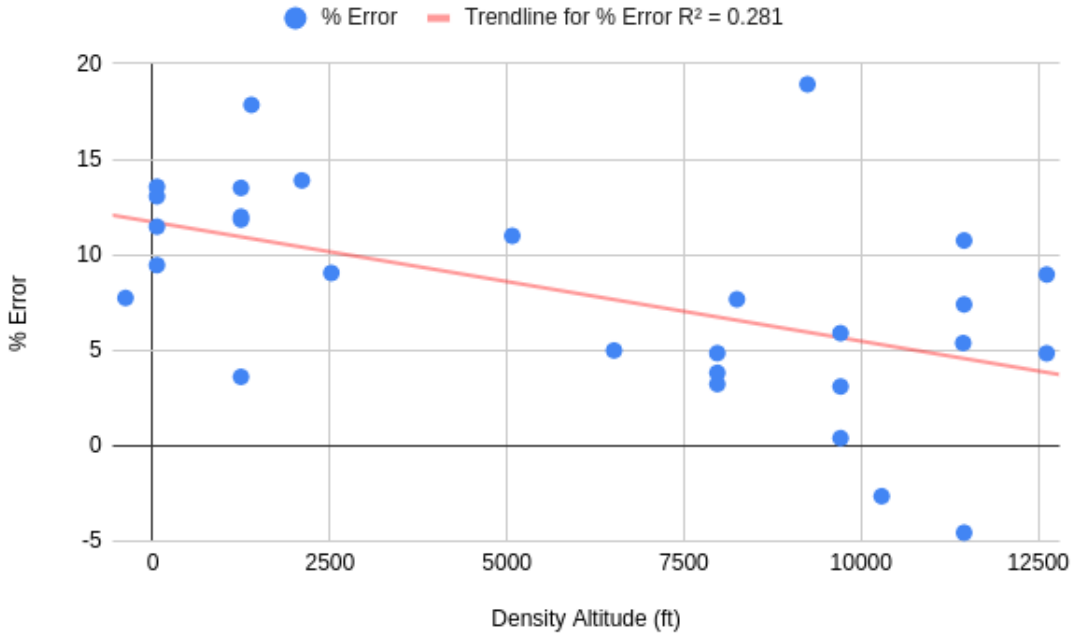


(b) Relation between error and density altitude.

Figure 3.12: Results of the Total model.

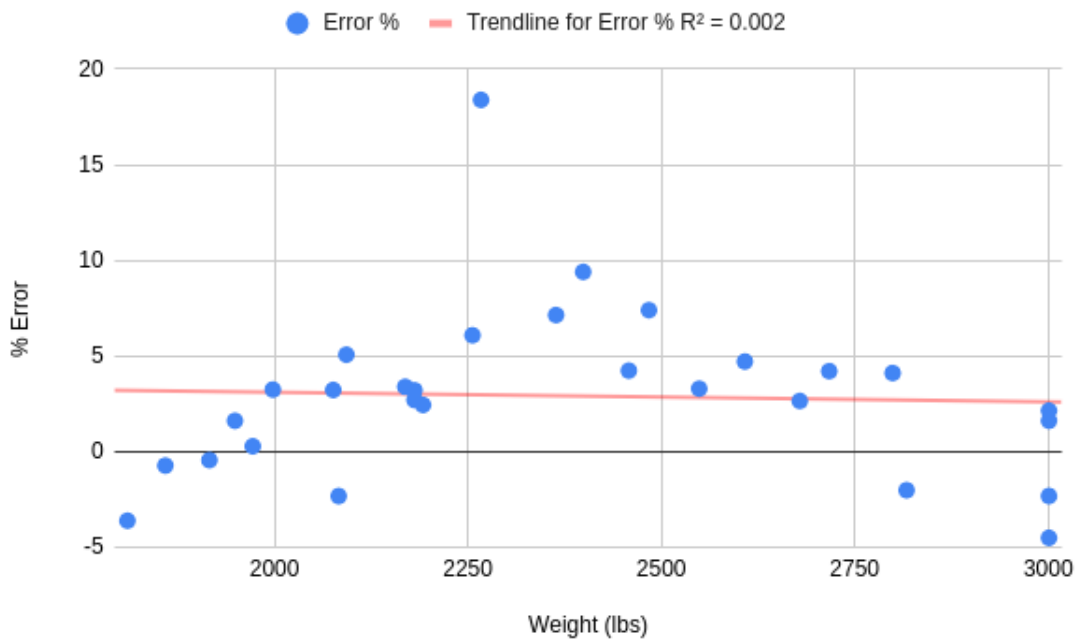


(a) Relation between error and weight.

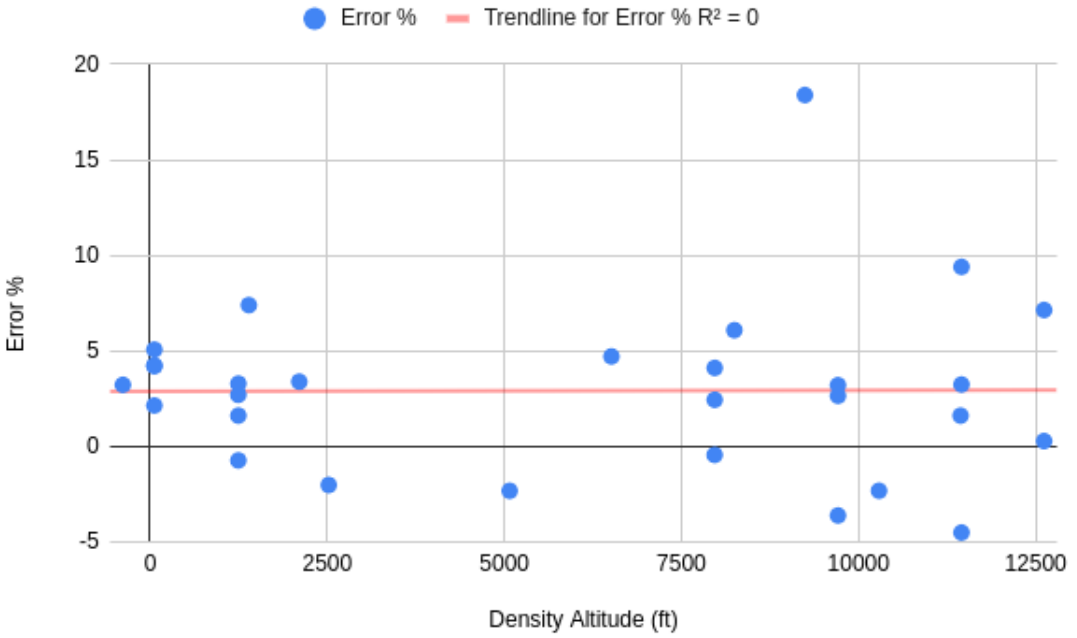


(b) Relation between error and density altitude.

Figure 3.13: Results of the Cutoff (below 2.5 knots) model



(a) Relation between error and weight.



(b) Relation between error and density altitude.

Figure 3.14: Results of the ensemble model.

From a safety standpoint, it is important to be able to quickly alert the pilot about an overweight condition so that they have enough time to take appropriate action. For large aircraft, V_1 is the take-off decision speed: the speed above which take-off should not

be aborted. Meaning, if there is an issue before reaching V_1, the pilot has enough time to abort take-off and stop safely; however, if an issue arises after V_1, the aircraft must complete take-off to not overrun the runway. However, in small aircraft such as the Cessna 172SP this speed is not calculated. To mitigate this, we have came up with a reasonable decision point for a small aircraft: either before the aircraft reaches 55 knots or 1000ft before the runway ends. This is because 55 knots is the speed at which standard procedure says to begin lifting the nose of the airplane. However, with heavy aircraft in high density altitudes it may take time for the aircraft to reach 55 knots. The aircraft needs to abort take-off with enough time to come to a full stop; 1000ft is more than enough runway to stop even on fairly short runways [6].

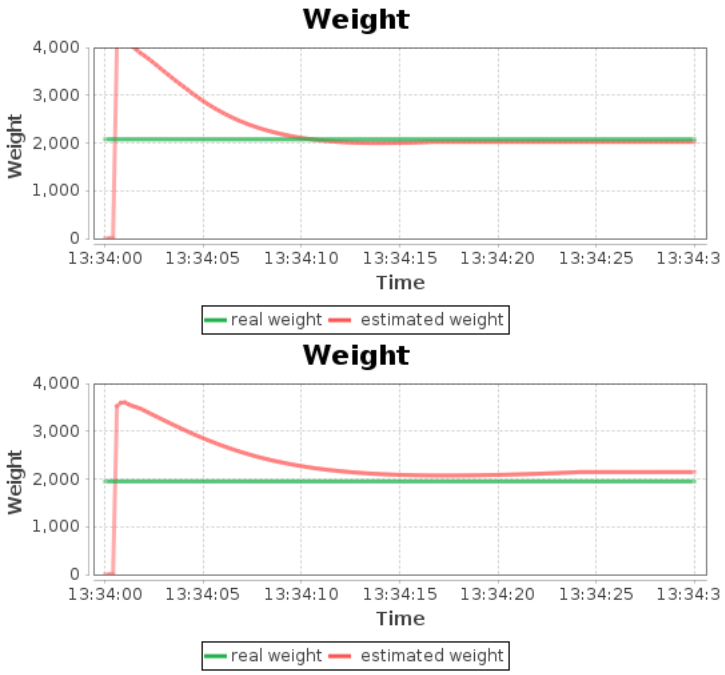


Figure 3.15: Weight estimation curves.

We used two points to indicate when the model produces useful results. This is when the model converges within 10% of the actual weight and when the model converges within 5%. These points are sooner than the point where the curve reaches a minimum which could also be used as a point of convergence. As can be seen in the weight estimation curve in Figure 3.15, the model first overestimates the weight of the aircraft then converges to a final value. When the model first converges within 10%, the model gives a rough estimate. This

[6] based on maximum landing distance ground roll in Cessna operating manual

gives us an idea of whether the estimate will be overweight or underweight. The estimate then stabilizes and converges within 5% and gives an estimate that is close to its final estimate. Table 3.5 shows how many testing trials converged and how quickly. Considering only the trials which converged, Tables 3.6 and 3.7 show the speeds and distances of the slowest trials to converge to 10% and 5%, respectively. It can be seen that the slowest trials still converged before the decision distance point even if the runways were half as long.

Table 3.5: Speed of model convergence.

Testing Airport	% of Trials Converged	Avg Time <10%	% of Trials Converged	Avg Time <5%	Avg Take-off Length
KALB	100	8.4	90	12.6	23.5
KRNO	100	10.5	80	18.1	29.3
KMOT	75	11.5	75	17.5	25
Overall	96.7	9.7	83.3	15.75	26.75

Table 3.6: Distance traveled and speed at slowest 10% convergence point.

Testing Airport	Slowest Trial Speed (knots)	Slowest Trial Distance (ft)	Runway Length (ft)
KALB	40.6	328	8500
KRNO	45.6	919	11001
KMOT	40	492	7700

Table 3.7: Distance traveled and speed at slowest 5% convergence point.

Testing Airport	Slowest Trial Speed (knots)	Slowest Trial Distance (ft)	Runway Length (ft)
KALB	56	1083	8500
KRNO	64	2297	11001
KMOT	61.9	1444	7700

3.4.2 Cessna 172R N4207P Accident

On August 25, 2014 a Cessna 172R airplane, N4207P, crashed in Willoughby Hills, Ohio shortly after take-off from Cuyahoga County Airport (KCGF)[7]. The private pilot and three passengers died in the crash. The aircraft was loaded to 2,622lbs, approximately 166lbs over its maximum gross weight of 2,457lbs (106% max gross weight), and was within safe ranges for the center of gravity. The airplane became airborne approximately 2000ft down runway 6 which is 5500ft long. The pilot noted that the aircraft was slow to climb and tried to turn back to land. The increased weight and steep turning angle used likely caused the aircraft to stall.

We created a scenario using X-Plane with a Cessna 172SP loaded to 106%, total of 2720lbs, taking off from KCGF runway 6. We replicated the atmospheric conditions of the

[7]see NTSB report CEN14FA453

day of the crash, 24°C and 30.09 inHG. Wind was reported as 10 knots from 140°, which would be a crosswind and would have negligible effect on take-off distance. However, we omitted wind entirely from our simulation as we currently solely use density altitude to encapsulate atmospheric conditions; this is an area of future work. The report noted that the aircraft had approximately 36 gallons of fuel at the time of take-off. The center of gravity was calculated from the information given in the report, 4.3 inches forward from the standard center of gravity which is within the safe range.

We collected 24 take-offs from KCGF runway 6. We used three distinct density altitudes: 860ft, 2049ft, and 3237ft, and the same eight weight categories from the previous experiments, see Figure 3.2. We trained a weight estimation model using this data, see Figure 3.17. Figure 3.16 shows the full PILOTS pilots program.

We then ran several take-off trials using the exact conditions mentioned previously to replicate the N4207P take-off. Figure 3.18 shows the velocity curve. The entire take-off was approximately 27s, and around 1800ft long, slightly faster then the approximate reported 2000ft of take-off distance of N4207P. The model produces the weight estimation curve as shown in Figure 3.19. However, on average, the model was able to estimate with a final accuracy of 3% and was able to estimate within 10% error after only 6s and within 5% error after 8s, as can be seen in Figure 3.20. After 8 seconds, our simulation only traveled 200ft. This means that the model could have alerted the pilot of an overweight condition well before the halfway point of the take-off giving ample time to react safely to this alert.

```
program weight_experiment;
  inputs
    // Airspeed, pressure, temperature, altitude,
    // and actual weight
    va, prs, tmp, alt, curr_w (t) using closest (t);
    // Estimate weight
    est_w (t) using model(N4207P_model, va, prs, tmp, alt);
  outputs
    va, curr_w, est_w, e at every 200 msec;
  errors
    e: (est_w-curr_w)/curr_w * 100;
end;
```

Figure 3.16: PILOTS N4207P program.

```
trainer N4207P_model;
  data
    // Airspeed, pressure, temperature, altitude,
    // and actual weight
    va, prs, tmp, alt, curr_w using
      file("N4207P_training_data.csv");
  model
    features: va, prs, tmp, alt;
    labels: curr_w;
    algorithm:
      WeightEstimator( cutoff: 0.0 );
end;
```

Figure 3.17: N4207P weight estimation model trainer.

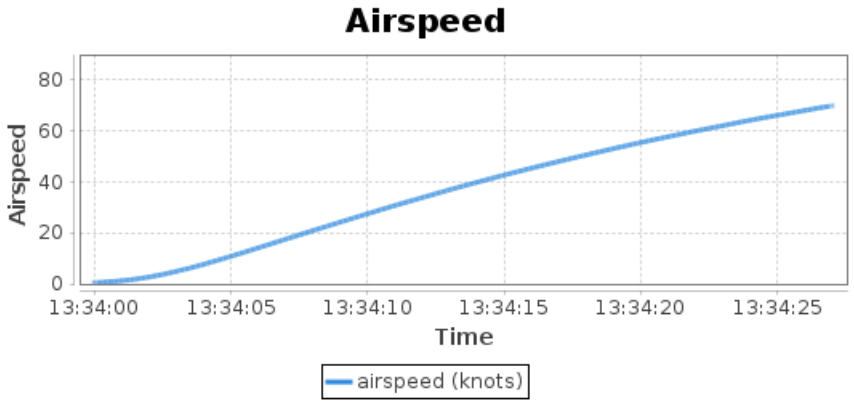


Figure 3.18: N4207P experiment velocity curve.

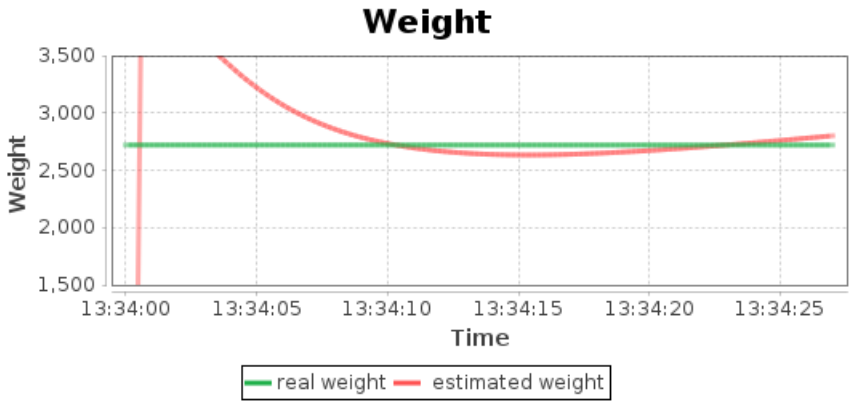


Figure 3.19: N4207P weight estimation curve.

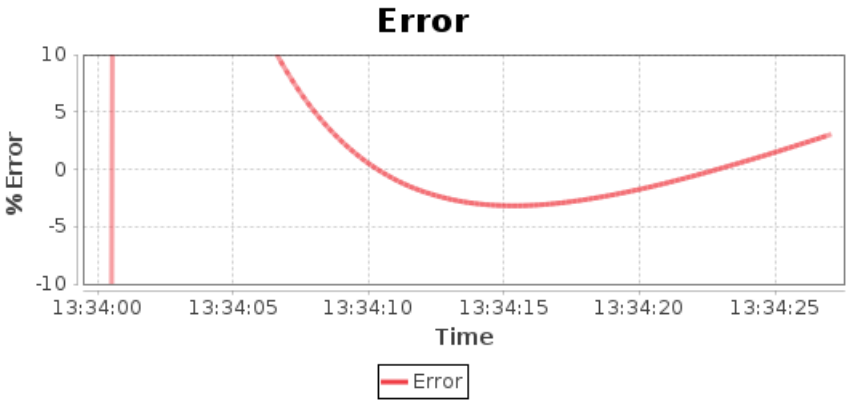


Figure 3.20: N4207P weight estimation error curve.

CHAPTER 4
Related Work

4.1 Data Stream Processing Languages

There are many Domain Specific Languages (DSLs) for machine learning applications [13]. OptiML is a DSL for machine learning which is implicitly parallel because it produces code which takes advantage of heterogeneous resources (CPUs and GPUs) [14]. OptiML has a functional programming style, as well as first-class support for graph, vector, and matrix operations. CVXPY is a DSL for convex optimization [15]. Convex optimization has applications in many fields, including aerospace [16], and CVXPY gives a simple and intuitive way to solve these problems. The language is embedded within Python, making it easy to use within PILOTS. MXNet is a machine learning library which combines symbolic expression with tensor computation [17]. MXNet can be ran in a distributed setting using a distributed synchronized key value store; it also has bindings in several languages including Python, R, Julia, and Go. The method of distributed computation is very different than the data stream processing model used in PILOTS. Furthermore, the previously mentioned DSLs are not declarative like PILOTS. SystemML is a system for development of large-scale machine learning algorithms which translates the scripts into an execution which uses MapReduce [18]. Similarly, ScalOps [19] and Pig Latin [20] uses the low-level procedural style of MapReduce but with the high-level declarative style of SQL. SCOPE is a scripting language which focuses on efficient processing of massive datasets which also is inspired by SQL [21]. The declarative SQL-like style of these DSLs is similar to the declarative nature of PILOTS; however, these DSLs are focused on processing large-scale data whereas PILOTS is focused on dynamic data-driven systems, error detection, and error recovery.

There are very few programming languages which support federated learning directly, as it is a fairly new technique. PySyft is a library that supports federated learning and encrypted communication on top of deep learning frameworks such as PyTorch [22]. TensorFlow Federated is a framework for computation on decentralized data, which supports federated learning [23]. Furthermore, ffl-erl is a federated learning framework written entirely in Erlang [24]. Erlang has certain performance penalties when compared to more optimized systems but the speed of development in Erlang may be more important in cer-

tain applications. However, these languages are non-declarative in nature and do not focus on a model-agnostic and simple interface. PILOTS aims for ease of use without the need to deal with procedural details of the model and communication.

Pycobra is a Python library for ensemble learning as well as visualisation [25]. The inclusion of ensemble methods within PILOTS aims to improve the usability of models within PILOTS. Therefore, Pycobra can be used within PILOTS models for more advanced ensemble learning functionality. LibEDM is a C++ library for ensemble-based data mining and provides many methods for data mining from pre-processing to classifier evaluation [26]. LibEDM is focused on being a data mining platform, whereas PILOTS is meant to be a declarative and extensible system for stream analysis.

4.2 Aircraft Weight Estimation

There exists some prior work on aircraft weight estimation. Imai, Galli, and Varela [3] used PILOTS to detect underweight conditions during cruise phase using data from the Tuninter 1153 accident. Lee and Chatterji [27] created a model which estimates the take-off weight required to complete a given flight plan, while accounting for limits on take-off weight, payload weight, and fuel capacity. It works by calculating the fuel usage of each segment of the flight plan and adding the needed fuel to the total weight. Similarly, Sun et al. [28] created a model which can estimate, with 4.3% accuracy the weight of an aircraft by analyzing the fuel consumption at various phases of flight and combining each estimate into a final value. The model calculates the probability distribution of weight for each phase based on equations for fuel consumption. Alligier et al. [29] created a mass estimation system used to help predict the rate of climb of an aircraft. It learns the thrust settings, which is how much thrust the aircraft is producing compared to standard, and using this estimates a weight distribution. Each of these models uses detailed physics-based models to accurately calculate values to be used by the model. However, our model uses the relationships between certain values, such as acceleration to weight, and density altitude to acceleration to estimate the weight in real-time. Furthermore, our model aims to provide information that may be critical to the safety of the pilot and passengers and provide that information before the aircraft becomes airborne, which none of the previous models can do.

There are several patents which detail systems to estimate aircraft weight before or during take-off. Several use the landing gear in some manner, either by using pressure

sensors within the landing gear [30] [31], or by measuring the bending of structural members of the aircraft which includes landing gear and wings [32]. Another details a system which automates the weighing of passengers and luggage and feeds this directly to a computer which calculates weight and center of gravity [33]. Each of these systems may work well for large commercial jets, which can afford installing new sensors and automated systems; however, they may not be as useful for smaller aircraft which cannot have these systems installed and are more likely to make mistakes such as accidentally loading too much cargo.

One system [34] describes measuring the acceleration of an aircraft, using an accelerometer, and using Newton's second law of motion to estimate weight of an aircraft during take-off. This systems uses a priori knowledge of aerodynamic properties such as coefficient of drag and aircraft wing area to calculate thrust and lift but ignores aerodynamic drag. Furthermore, it details a system to estimate the rolling resistance using a weighted average over time. This model is similar to our model in that it uses a measured acceleration to estimate weight during take-off. However, there are significant differences in the implementation. Our model has no prior knowledge of the aircraft and only uses previous take-off data. Furthermore, our model requires no hardware such as an accelerometer and can be used in various aircraft with no changes.

CHAPTER 5
Discussion

5.1 Conclusion

This paper discussed declarative extensions to PILOTS to learn models from data. Our extensions allow users of PILOTS to create systems using machine learning algorithms quickly and easily without a strong background in programming. Furthermore, it allows for users with a background in programming to easily integrate machine learning algorithms into PILOTS. We also proposed further extensions which allow for distributed machine learning and the use of ensemble models. These features in PILOTS are more declarative and model-agnostic than other similar domain specific languages and libraries. These tools can be used to make systems where aircraft train models collaboratively and create ensemble models based on the results from values proposed by other aircraft.

We also discussed our method of data-driven aircraft weight estimation. Our method can be used on nearly any aircraft with no additional hardware and provides useful results to the pilot in real-time during take-off. We showed that our method is capable of preventing accidents such as the August 2014 N4207P accident where an overweight condition caused the aircraft to crash shortly after take-off.

5.2 Future Directions

The ability of PILOTS to learn models from data has many possible improvements. Firstly, efforts can be made to implement our proposed features for federated learning and ensemble methods. Furthermore, each machine learning algorithm is limited in its ability to be adjusted by the training grammar. Currently, only boolean and numeric settings can be adjusted whereas the ability to enter expressions that will be used by the model or vectors could be useful to improve the clarity of the code.

Our weight estimation model was trained on data from small fixed-wing single-engine aircraft. Further research can look at other types of aircraft, such as larger multi-engine aircraft and commercial jets, and see if our model can produce accurate results before take-off decision speed. Appendix A shows example velocity curves for other aircraft. Furthermore,

the ability to take wind into account would also greatly improve the usefulness of the model in real-world applications.

Another avenue for future work is looking into creating an ensemble system to estimate weight during all phases of flight such as climbing, descending, landing, and turning. Even taxiing could also potentially provide enough information to detect errors. Besides overweight conditions, such a model would also be able to detect certain failures of the aircraft. For example: fuel leaking which decreases aircraft weight, an engine failure which decreases thrust, or wing surfaces icing which decreases lift. The exact source of such an error may not always be able to be determined; however, being able to alert the pilot to an error is critical to maintaining safety.

In data-driven systems, there is a lot of inherent randomness and uncertainty: sensor errors, communication delays and failures, and much more. Without the ability to maintain correctness in these conditions, an application is not useful in the real world. However, it is unfeasible to be able to be correct in all possible types of conditions, due to computational, temporal, or model-based constraints. Making models that have a clearly defined envelope of correctness allows the user, or other applications, to know when the result may not be correct. Furthermore, modeling this uncertainty can also be useful to know when one model is producing a more correct output. Ensuring transparency of models is important to improve trust of the system.

REFERENCES

[1] S. Imai and C. A. Varela, "Programming spatio-temporal data streaming applications with high-level specifications," in *3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012*, Redondo Beach, California, USA, November 2012.

[2] S. Imai, S. Chen, W. Zhu, and C. A. Varela, "Dynamic data-driven learning for self-healing avionics," *Cluster Computing*, Nov 2017. [Online]. Available: http://rdcu.be/yJNh [Accessed: Nov. 11, 2019]

[3] S. Imai, A. Galli, and C. A. Varela, "Dynamic data-driven avionics systems: Inferring failure modes from data streams," in *Dynamic Data-Driven Application Systems (DDDAS 2015)*, Reykjavik, Iceland, June 2015.

[4] S. Imai, F. Hole, and C. A. Varela, "Self-healing data streams using multiple models of analytical redundancy," in *The 38th AIAA/IEEE Digital Avionics Systems Conference (DASC 2019)*, San Diego, CA, September 2019. [Online]. Available: http://wcl.cs.rpi.edu/papers/DASC2019_imai.pdf [Accessed: Feb. 7, 2020]

[5] R. S. Klockowski, S. Imai, C. Rice, and C. A. Varela, "Autonomous data error detection and recovery in streaming applications," in *Proceedings of the International Conference on Computational Science (ICCS 2013). Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop*, May 2013, pp. 2036–2045.

[6] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving Rubik's cube with a robot hand," 2019.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[8] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," 2016.

[9] A. Hard, C. M. Kiddon, D. Ramage, F. Beaufays, H. Eichner, K. Rao, R. Mathews, and S. Augenstein, "Federated learning for mobile keyboard prediction," 2018. [Online]. Available: https://arxiv.org/abs/1811.03604 [Accessed: Mar. 18, 2020]

[10] K. Hoki, T. Kaneko, D. Yokoyama, T. Obata, H. Yamashita, Y. Tsuruoka, and T. Ito, "A system-design outline of the distributed-shogi-system Akara 2010," in *2013 14th*

ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, July 2013, pp. 466–471.

[11] J. Xu-rui, W. Ming-gong, W. Xiang-xi, and W. Ze-kun, "Application of ensemble learning algorithm in aircraft probabilistic conflict detection of free flight," in *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, May 2018, pp. 10–14.

[12] C. A. Company, *Information manual, Cessna Aircraft Company 1981 model 172P*. Cessna Aircraft Company, 1980.

[13] I. Portugal, P. Alencar, and D. Cowan, "A preliminary survey on domain-specific languages for machine learning in big data," in *2016 IEEE International Conference on Software Science, Technology and Engineering (SWSTE)*, June 2016, pp. 108–110.

[14] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: An implicitly parallel domain-specific language for machine learning," in *ICML*, 2011, pp. 609–616. [Online]. Available: https://icml.cc/2011/papers/373_icmlpaper.pdf [Accessed: Mar. 8, 2020]

[15] S. Diamond and S. Boyd, "CVXPY: A python-embedded modeling language for convex optimization," *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 2909–2913, Jan. 2016.

[16] X. Liu, P. Lu, and B. Pan, "Survey of convex optimization for aerospace applications," *Astrodynamics*, vol. 1, 02 2017.

[17] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.

[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *2011 IEEE 27th International Conference on Data Engineering*, April 2011, pp. 231–242.

[19] M. Weimer, T. Condie, R. Ramakrishnan *et al.*, "Machine learning in ScalOps, a higher order cloud computing language," in *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, vol. 9, 2011, pp. 389–396.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1099–1110. [Online]. Available: https://doi.org/10.1145/1376616.1376726 [Accessed: Mar. 8, 2020]

[21] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1265–1276, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454166 [Accessed: Mar. 8, 2020]

[22] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," 2018.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/ [Accessed: Mar. 15, 2020]

[24] G. Ulm, E. Gustavsson, and M. Jirstrand, "Functional federated learning in Erlang (ffl-erl)," in *Functional and Constraint Logic Programming*, J. Silva, Ed. Cham: Springer International Publishing, 2019, pp. 162–178.

[25] B. Guedj and B. S. Desikan, "Pycobra: A python toolbox for ensemble learning and visualisation," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 6988–6992, Jan. 2017.

[26] Q. Zhao and Y. Jiang, "LibEDM: A platform for ensemble based data mining," in *2014 IEEE International Conference on Data Mining Workshop*, Dec 2014, pp. 1250–1253.

[27] H. tae Lee and G. Chatterji, *Closed-Form Takeoff Weight Estimation Model for Air Transportation Simulation.* Aerospace Research Central, 2012. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2010-9156 [Accessed: Mar. 9, 2020]

[28] J. Sun, J. Ellerbroek, and J. M. Hoekstra, "Aircraft initial mass estimation using Bayesian inference method," *Transportation Research Part C: Emerging Technologies*, vol. 90, pp. 59 – 73, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X18302626 [Accessed: Mar. 9, 2020]

[29] R. Alligier, D. Gianazza, and N. Durand, "Learning the aircraft mass and thrust to improve the ground-based trajectory prediction of climbing flights," *Transportation Research Part C: Emerging Technologies*, vol. 36, pp. 45 – 60, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X13001708 [Accessed: Mar. 9, 2020]

[30] C. K. Nance, "Aircraft weight and center of gravity indicator," U.S. Patent 5,548,517, Oct. 22, 1998.

[31] M. A. Long and G. E. Gouette, "Aircraft weight and balance system," U.S. Patent 7,967,244, Nov. 16, 2006.

[32] C. D. Bateman, "Weight, balance, and tire pressure detection systems," U.S. Patent 4,312,042, Jan. 17, 1992.

[33] R. Stefani, "Aircraft weight and balance system," U.S. Patent 6,923,375, Sep. 29, 2003.

[34] H. Miller, "Takeoff weight computer apparatus for aircraft," U.S. Patent 4,490,802, Jan. 04, 1982.

APPENDIX A
Weight Estimation

Examples of different aircraft velocity curves.

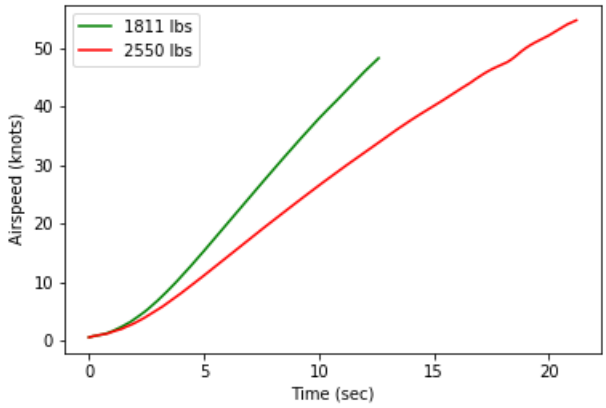


Figure A.1: Velocity curve of Cessna 172SP take-off from grass runway NY46.

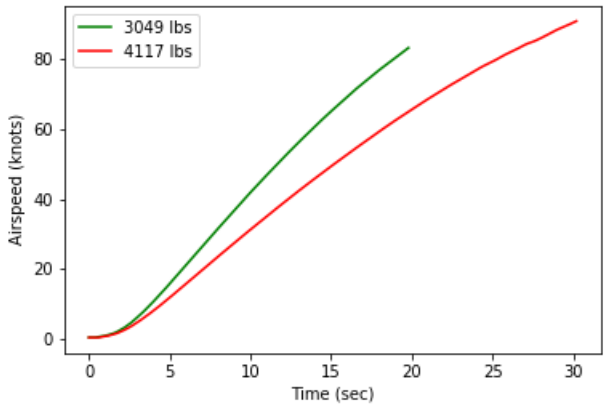


Figure A.2: Velocity curve of Piper PA 46 310P take-off from KALB.

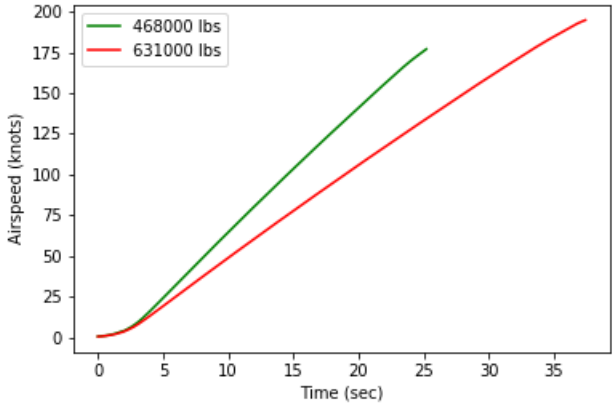


Figure A.3: Velocity curve of Boeing 777 takeoff from KALB.