Noname manuscript No.
(will be inserted by the editor)

Dynamic data-driven learning for self-healing avionics

Shigeru Imai · Sida Chen · Wennan Zhu · Carlos A. Varela

Received: date / Accepted: date

Abstract In sensor-based systems, spatio-temporal data streams are often related in non-trivial ways. For example in avionics, while the airspeed that an aircraft attains in cruise phase depends on the weight it carries, it also depends on many other factors such as engine inputs, angle of attack, and air density. It is therefore a challenge to develop failure models that can help recognize errors in the data, such as an incorrect fuel quantity or an incorrect airspeed. In this paper, we present a highly-declarative programming framework that facilitates the development of *self-healing* avionics applications, which can detect and recover from data errors. Our programming framework enables specifying expert-created failure models using *error signatures*, as well as learning failure models from data. To account for unanticipated failure modes, we propose a new dynamic Bayes classifier, that detects outliers and upgrades them to new modes when statistically significant. We evaluate error signatures and our dynamic Bayes classifier for accuracy, response time, and adaptability of error detection. While error signatures can be more accurate and responsive than dynamic Bayesian learning, the latter method adapts better due to its data-driven nature.

Keywords data streaming, spatio-temporal data, declarative programming, linear regression, Bayesian statistics.

Shigeru Imai, Sida Chen, Wennan Zhu, and Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
110 Eighth Street, Troy NY USA 12180
Tel.: (518) 276-6912
E-mail: {imais, chens15, zhuw5}@rpi.edu, cvarela@cs.rpi.edu

1 Introduction

Detecting and recognizing patterns in data streams generated by multiple aircraft sensors has become an important research area for flight safety. In the Air France flight 447 (AF447) accident in 2009, iced pitot tubes caused an error in air speed data, and the pilots failed to react correctly, leading to the crash [6]. While there has been a large body of work on fault detection, isolation, and reconfiguration (FDIR) [14], information fusion [5], and anomaly detection [28,27,4,10], there is still a need for further research in methods that allow for fault detection and recovery techniques to be easily realized and implemented with minimal risk of software errors [15]. To facilitate the development of such *self-healing* flight systems that embody the DDDAS [9] concept, we have been developing the ProgrammIng Language for spatiO-Temporal data Streaming applications (PILOTS) and its runtime system for fault detection and correction in data streams [19,20,25,17,16,8,15].

For the AF447 accident, we successfully developed a PILOTS program that detects the incorrect airspeed and estimates the correct airspeed from ground speed and wind speed [17]. As part of the PILOTS program, we manually created a failure model in the form of *error signatures*, which are used to identify error patterns. However, there are some cases where creating failure models is non-trivial. For example, in the Tuninter 1153 (TU1153) flight accident in 2005, the fuel quantity indicator of a different aircraft model was erroneously installed, causing the instrument to display an incorrect amount of fuel, which led to fuel exhaustion of the aircraft [31]. This accident could have been avoided if the weight error could be detected by checking the relationship between lift and weight during the cruise phase of flight. Lift depends on airspeed, air density, wing sur-

face area, and coefficient of lift. The coefficient of lift itself depends on the angle of attack and this relationship can change with different aircraft types. Understanding such complex relationships from multiple sensor data streams is critical to accurately detecting sensor faults.

As we have seen in the AF447 and TU1153 accidents, there are failure models with different complexities. Thus, it is preferred to take different modeling approaches depending on the complexity of the target failure model. We consider two modeling approaches for self-healing data streaming systems: 1) models created by domain experts and 2) models learned from data. While we use error signatures for the former, we apply machine learning (ML) techniques to the latter. Specific research questions include the following:

RQ1 How to design a programming language for self-healing data streaming systems that supports both (i) models created by domain experts, and (ii) models learned from data?

RQ2 How to support data-driven machine learning of failure models in such a programming language?

RQ3 Since it is not possible to predict every failure mode, can we support dynamic data-driven learning of unanticipated failure modes?

In this paper, we propose a programming framework for self-healing data streaming applications which supports (i) detecting sensor faults from data, (ii) estimating new values for associated streams when possible, and (iii) dynamic data-driven learning of failure modes. The proposed framework addresses the research questions as follows. For the first question, we take a domain-specific approach to enable high-level description of both error signatures and ML-based failure modeling in the PILOTS programming language. We use separate abstractions for model learning and prediction for separation of concerns. Also, we provide a succinct declarative grammar for ease of use. For the second question:

- We integrate an existing ML framework into our PILOTS runtime infrastructure. For modularity, we define a general interface to access ML functionalities from the main PILOTS system.
- We support online/offline ML algorithms for classification and regression.

For the third question, to detect statistically significant new modes online in addition to pre-existing modes that are already learned offline, we develop a new dynamic Bayes classifier that extends a naive Gaussian Bayes classifier.

To illustrate the usage of our proposed framework, we analyze two real-world commercial accidents as case studies, namely the AF447 and TU1153 accidents. We apply our dynamic Bayes classifier to both accidents to detect outliers as new modes using unsupervised learning. In summary, our contributions are as follows:

- A highly-declarative programming framework to use machine learning algorithms and learn failure models for self-healing data stream processing.
- A new dynamic Bayes classifier: a naive Bayes based classifier that dynamically recognizes statistically significant new modes in addition to pre-existing modes.
- Evaluation of the proposed framework with real-world data and synthetic data from actual commercial flight accidents. Using these data, we successfully detected pitot tube and fuel quantity indicator faults and estimated new values for associated airspeed and weight streams.

The rest of the paper is organized as follows. Section 2 introduces spatio-temporal data stream processing implemented by the core component of PILOTS. Section 3 presents a framework for self-healing data stream processing and describes how we support both error signatures and machine learning techniques for error detection and correction. Section 4 introduces our new dynamic Bayes classifier and online learning. Sections 5 and 6 discuss the methods and results of the proposed learning approaches using case studies of airplane weight estimation and airspeed estimation. Section 7 describes related work. Section 8 discusses potential improvements for PILOTS and future research directions. Finally, we conclude the paper in Section 9.

2 Spatio-temporal data stream processing software

Today's most commonly used programming languages (*e.g.*, C/C++, Java, PHP, JavaScript, python, etc.) do not have first-class support for space and time since they are designed to be general-purpose. The downside of the general-purpose approach is the complexity and size of code. Since these programming languages are imperative, meaning that we have to use *for, if*, or *while* to control the flow of the programs and explicitly handle state, the code can get large and complex easily. In contrast to the general-purpose approach, if we know a specific problem domain very well and want to provide first-class support for key domain concepts (such as space and time), we can take a domain-specific approach. Domain-specific languages are less expressive than general-purpose programming languages; however, code is more declarative and therefore simpler to write, read, and reason about. We design PILOTS as a highly-declarative domain-specific programming language for

spatio-temporal data stream processing, aiming to provide a suitable programming environment for data scientists with less programming background.

In this section, we explain the design of our PILOTS system as spatio-temporal data stream processing software. We first show the runtime system followed by data selection criteria for homogeneous stream processing and compilation of PILOTS programs.

2.1 PILOTS programming language

Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level (declarative) program specification. The PILOTS program includes an inputs section to specify the data streams and how data is to be extrapolated from heterogeneous data, typically using declarative data selection criteria (*e.g.*, closest, interpolate, euclidean keywords, see Section 2.3 for details). It includes an outputs section to specify the data streams to be produced by the application, as a function of the input streams with a given frequency.

Figure 1 shows one of the simplest programs written in PILOTS, called Twice. As the name suggests, it takes two input streams, $a(t)$ and $b(t)$, where b is supposed to be twice as large as a, and outputs an error defined by $e = (b - 2 * a)$ every second. Note that these two input streams are not associated with any location information.

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

Fig. 1 Declarative specification of the Twice PILOTS program.

2.2 Core system architecture of PILOTS

Figure 2 shows the core system architecture of the PILOTS runtime system. The data manager first takes N raw input spatio-temporal data streams $d_i(x, y, z, t), i = 1, 2, ..., N$, which are *heterogeneous* in terms of granularity on space and time. The application model is derived from a user's original PILOTS program, for example, as shown in Figure 1. It requests the data manager to send data streams at a specified rate (*e.g.*, every second) with certain data selection criteria explained in Section 2.3. The data manager transfers requested data streams $\bar{d}_i(t)$, $i = 1, 2, ..., N$, which are now only functions of time (*i.e.*, *homogeneous* streams), and the application module computes output streams $o_j(t)$, $j = 1, 2, ..., M$ and error streams $e_k(t)$, $k = 1, 2, ..., L$ specified by an optional errors section (see Figure 4 for an errors section example).

Whereas spatio-temporal data is available with various spatial density and time frequency depending on sources of data in general (*e.g.*, weather forecast data can be given hourly/daily for a vast geographic area, GPS data can be given every second or at higher frequency for a specific geographic location), the application often needs to process data at a constant frequency. The data manager essentially allows the application to view a set of these heterogeneous data streams as a homogeneous data stream, and therefore enables a separation of concerns: application programmers can focus on their application model.

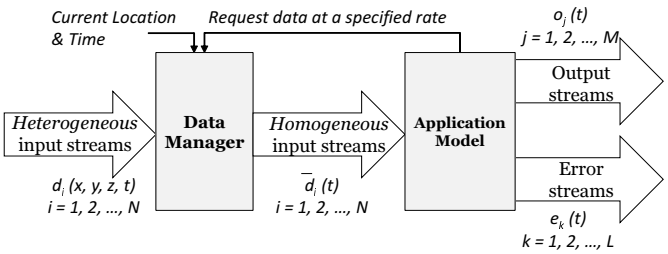

Fig. 2 System architecture of the core PILOTS runtime system.

2.3 Data selection criteria

PILOTS offers the following data selection criteria to the programmer so that the application can get data consistently regardless of its heterogeneity. We explain three data selection methods: closest, euclidean, and interpolate. PILOTS takes a view that the application is moving in the 3-D space and is provided with the current location and time as shown in Figure 2. In the following, we assume that we are given the current location and time as (x, y, z) and t respectively.

closest This method takes a 1-D argument from $\{x, y, z, t\}$ to find the closest data point to the given argument. This method is applicable to both space and time domains.

euclidean This method is an extension of closest to 2-D Euclidean plane and 3-D Euclidean space. euclidean takes any combination of two or more arguments

from $\{x, y, z\}$ and selects the data point that has the closest Euclidean distance to the specified point.

interpolate This method takes any combinations of locations $\{x, y, z\}$ or time t as arguments to linearly interpolate multiple data points. It also takes another argument n_{interp} to specify the maximum closest data points to interpolate. Suppose we have three 2-D data points $d_0(x_0, y_0)$, $d_1(x_1, y_1)$, $d_2(x_2, y_2)$, and $n_{\text{interp}} = 2$. If d_0 and d_1 are the two closest data points to the current location (x, y), we linearly interpolate d_0 and d_1.

2.4 Operational modes

There are two operational modes for how to run PILOTS programs.

- *real-time mode:* This mode is default and is intended to be used for receiving data from sensors and real-time processing. If the frequency of an output is specified as "at every 1 min" in the program specification, the program actually outputs data once every 1 minute. Also, in this mode, the program finishes if one of input data streams sends the last line marker (\n) or certain amount of time has elapsed, which can be specified by the user in the command line as -DtimeSpan=30min.
- *simulation mode:* This mode is used for simulations and is activated if the user gives a past time span in the command line as -DtimeSpan=t0∼t1. The PILOTS runtime sets its internal time as t0 and virtually progress the time as the program runs, and when the internal time reaches t1, the program finishes. The user can get outputs as fast as possible. This mode is intended to be used for processing recorded data in the past.

2.5 Compilation

PILOTS programs are translated into Java code for platform independent execution. The PILOTS compiler consists of two parts: a parser and a code generator. The parser is developed using JavaCC [21]. Depending on which operational mode (see Section 2.4) is specified, the compiler generates code either for real-time or simulation mode. Table 1 shows a comparison of lines of code between PILOTS and compiled Java programs for the real-time mode for three programs which we will show later: Twice (Figure 1), Twice_Signatures (Figure 4), WeightCheck_Signatures (Figure 12), and SpeedCheck_Signatures (Figure 21). It also shows the lines of code for the PILOTS common library that is used from all PILOTS programs.

Table 1 Comparison of lines of code between PILOTS, compiled Java programs, and the PILOTS common library (Note: the lines of code exclude comments and names of the programs are shortened to save space).

Language	Twice	Twice_Sigs	WC_Sigs	SC_Sigs
PILOTS	7	15	18	17
Java	62	95	122	139
Common Lib.	5365			

3 Self-healing streams using error signatures and machine learning

In this section, we present our framework for self-healing data stream processing using PILOTS. We define error signatures for expert-created models, and a machine learning component to support data-driven model training and prediction.

3.1 System architecture

Figure 3 shows the system architecture of the self-healing PILOTS runtime system. To add a self-healing feature (*i.e.*, error detection and correction) in PILOTS, we add the mode estimation subsystem, which consists of two components: the error analyzer and ML component, to the core PILOTS runtime system in Figure 2. For error detection, the error analyzer uses error signatures and the ML component uses machine learning. Whereas a domain expert designs error signatures as part of a PILOTS program, for the ML component, models are purely trained with data by the ML engine either offline or online. Once an error mode is estimated, error correction is performed by the application model regardless of the error detection method. The types of streams exchanged between the core PILOTS system and the mode estimation subsystem are: *input*, *error*, *prediction*, and *mode*. Whereas the ML component directly takes input streams from the data manager and generates prediction streams or estimated error mode streams, the error analyzer takes error streams from the application model and returns estimated error mode streams (for the definition of a mode, see Section 3.2.1). Also, using the same architecture, the user can also run a PILOTS program without the self-healing feature (*e.g.*, Figure 1), with error signature-based error detection only (*e.g.*, Figure 4), or with ML-based error detection only (*e.g.*, Figure 7).

For better modularity, the ML engine is accessible through the ML interface with an adaptation layer, which is implemented for a target ML library. The ML engine exposes the following functionalities: offline/online model training, value prediction for regression, and mode prediction for classification. We use *scikit-learn*

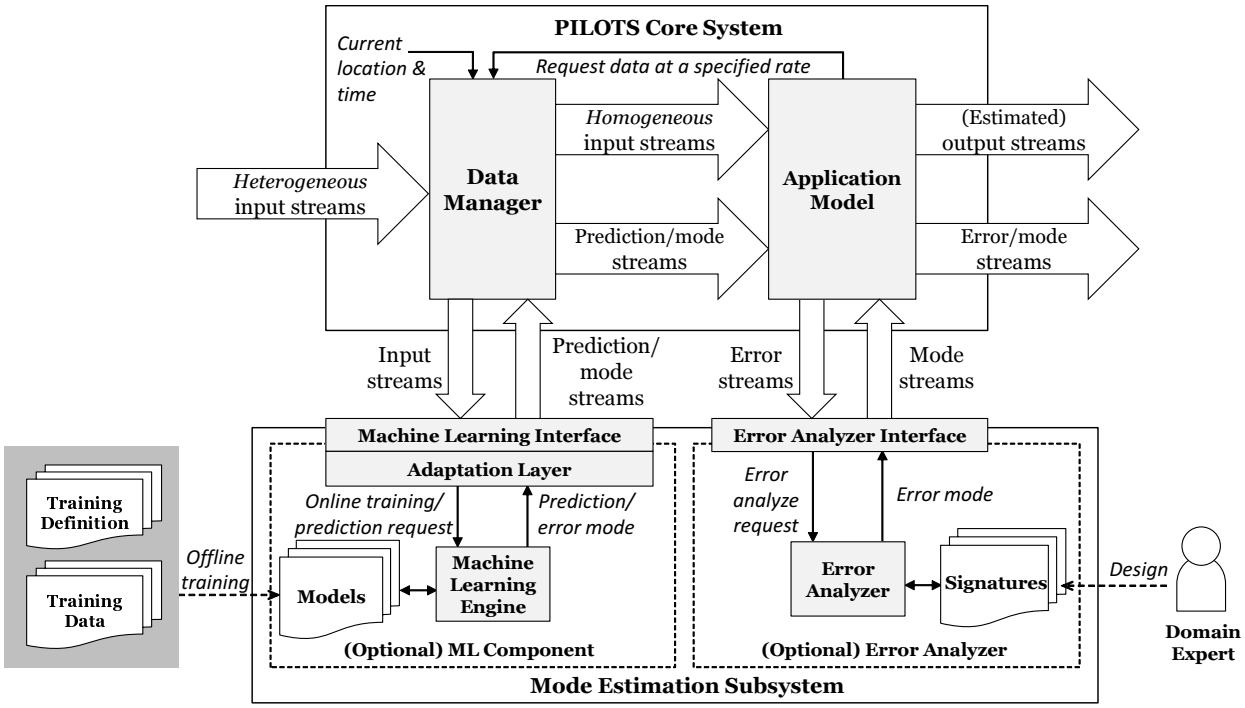

Fig. 3 System architecture of the proposed self-healing PILOTS runtime system.

as the ML engine in this work [30]. Similar to the ML-engine, the error analyzer exposes its functionalities through the error analyzer interface, which is used to request mode estimation by error signatures. Currently, there is no need to define an adaptation layer for the error analyzer since there is no alternative implementation.

3.2 Error signatures

3.2.1 Definition of error signature

Error functions Error functions are used to detect possible faults among redundant input data streams. An error function should have the value zero if there is no error in the input data, when the whole system is working in the normal mode.

For example, in the cruise phases of a flight, the lift equals the weight of the airplane. The lift can also be calculated using other independently measured inputs, including angle of attack, air density, temperature, pressure, and airspeed. In this case, an error function can be defined as follows.

$$e(lift, weight) = lift - weight. \tag{1}$$

In cruise phases, this equation should be zero. If there is an error in the weight indicator, and the input weight data is lower than the real weight, Equation (1) should be greater than zero. Similarly, if the input weight data is higher than the real weight, Equation (1) should be smaller than zero. Thus, the validity of the input data can be determined from the value of the error function. As shown in Figure 2, error functions can be computed from homogeneous input streams periodically and can be represented as $e(t)$.

Error signatures An *error signature* is a constrained mathematical function pattern that is used to capture the characteristics of an error function $e(t)$. Using a vector of constants $\bar{K} = \langle k_1, \ldots, k_m \rangle$, a function $f(t, \bar{K})$, and a set of constraint predicates $\bar{P} = \{p_1(\bar{K}), \ldots, p_l(\bar{K})\}$, the error signature $S(\bar{K}, f(t, \bar{K}), \bar{P}(\bar{K}))$ is defined as follows:

$$S(f(t, \bar{K}), \bar{P}(\bar{K})) \triangleq \{\ f \mid p_1(\bar{K}) \wedge \cdots \wedge p_l(\bar{K})\}. \tag{2}$$

Mode likelihood vectors Given a vector of error signatures $\langle S_0, \ldots, S_n \rangle$, we calculate $\delta_i(S_i, t)$, the distance between the measured error function $e(t)$ and each error signature S_i by:

$$\delta_i(S_i, t) = \min_{g(t) \in S_i} \int_{t-\omega}^{t} |e(t) - g(t)| dt. \tag{3}$$

where ω is the window size. Note that our convention is to capture "normal" conditions as signature S_0. The smaller the distance δ_i, the closer the raw data is to the

theoretical signature S_i. We define the *mode likelihood vector* as $L(t) = \langle l_0(t), l_1(t), \ldots, l_n(t) \rangle$ where each $l_i(t)$ is:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t),\ldots,\delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases} \quad (4)$$

Mode estimation Using the mode likelihood vector, the final mode output is estimated as follows. Observe that for each $l_i \in L, 0 < l_i \leqslant 1$ where l_i represents the ratio of the likelihood of signature S_i being matched with respect to the likelihood of the best signature.

Because of the way $L(t)$ is created, the largest element l_j will always be equal to 1. Given a threshold $\tau \in (0, 1)$, we check for one likely candidate l_j that is sufficiently more likely than its successor l_k by ensuring that $l_k \leqslant \tau$. Thus, we determine j to be the correct mode by choosing the most likely error signature S_j. If $j = 0$ then the system is in *normal mode*. If $l_k > \tau$, then regardless of the value of k, *unknown error* mode (-1) is assumed.

Error correction Whether or not a known error mode i is recoverable is problem dependent. If there is a mathematical relationship between an erroneous value and other independently measured values, the erroneous value can be replaced by a new value estimated from the other independently measured values.

3.2.2 Error detection and correction

Figure 4 shows an example PILOTS program with error signatures called Twice_Signatures. This program is an enhancement of the Twice program in Figure 1. A signatures section is defined to capture the shape of the error computed in e under the errors section. Both a and b are expected to increase by one for a and by two for b every second (*i.e.*, $a(t) = t + k$ and $b(t) = 2t + k$, where k is a constant). Thus, the error is zero in the Normal case. Suppose a is failed and keeps producing the last observed value, the error keeps increasing with the slope of 2. Similarly, when b is failed and keeps producing the last observed value, the error keeps decreasing with the slope of -2. We can express these behaviors as error signatures: $e = 2 * t + k$ and $e = -2 * t + k$ respectively for A failure and B failure. Using the error detection method described in Section 3.2.1 with these error signatures, the error analyzer detects error modes. Once a mode is determined, the original data is estimated by the application model as shown in the estimate clauses: $a = b/2$ for A failure and $b = 2 * a$ for B failure.

```
program Twice_Signatures;
    inputs
        a (t) using closest(t);
        b (t) using closest(t);
    outputs;
        o: (b - 2 * a) at every 1 sec;
    errors
        e: b - 2 * a;
    signatures
        s0: e = 0          "Normal";
        s1: e = 2*t + k   "A failure"
            estimate a = b / 2;
        s2: e =-2*t + k   "B failure"
            estimate b = 2 * a;
end
```

Fig. 4 Error signatures-based declarative specification of the Twice_Signatures PILOTS program.

3.3 Machine learning

3.3.1 Offline and online training

The ML engine takes a trainer file [1], as shown in Figure 5 and Figure 6 for example, to train a model regardless of whether the training is online or offline. The trainer file consists of three sections: optional constants, and mandatory data and model sections. The constants section defines constants. The data section contains information for training data file and variables used in the model section. The model section further contains subsections: features, labels, and algorithm. Figure 5 is an example of linear regression trainer that learns linear relationship $b = \beta_1 a + \beta_0$ from variables a and b offline. Figure 6 is another example of Bayes classifier trainer, which is used to learn a label specified by the mode from a computed feature $b - 2 * a$.

For the offline training, the user manually trains a model using a training data file specified in the trainer file. For the online training, training is automatically performed together with prediction at runtime if online training is enabled for the specified model. Every time the data manager requests a prediction, the ML engine updates a learning model and gives prediction results back to the data manager, which sends the prediction results to the application model as requested.

```
trainer twice_b_regression;
  data
    a, b using file(twice_ab.csv);
  model
    features: a;
    labels: b;
    algorithm: LinearRegression;
end
```

Fig. 5 Example training specification for linear regression.

[1] We have designed, but not fully implemented the trainer abstraction in PILOTS version 0.4 as of June 2017.

```
trainer twice_mode_bayes;
  data
    a, b, mode using file(twice_abmode.csv);
  model
    features: b - 2*a;
    labels: mode;
    algorithm: BayesClassifier;
end
```

Fig. 6 Example training specification for Bayes classifier.

3.3.2 Error detection and correction

As shown in Figure 7, to support prediction in the PILOTS programming language, a new data selection method model is defined in addition to the methods described in Section 2.3. The model method takes an identifier of the model, as defined in the trainer file (*e.g.*, twice_mode_bayes in Figure 6), and input variables as arguments. This method works as the main interface between the data manager and the ML-engine. Figure 7 shows a simple example PILOTS program called Twice_Bayes, which is a Bayes classifier version of the Twice PILOTS program. Same as in the examples shown in Figure 1 and Figure 4, b is expected to be twice as large as a. We assume a Bayes classifier is already trained with the trainer shown in Figure 6 with three modes: Normal(=0), A failure(=1), B failure(=2). Also, we can refer to this classifier as twice_mode_bayes. This program obtains a mode directly from the model method with input variables a and b. In case of failures, either a or b is recomputed by the estimate clauses similar to the error signatures-based error correction shown in Figure 4.

```
program Twice_Bayes;
  inputs
    a, b (t) using closest (t);
    mode (t) using model(twice_mode_bayes, a, b);
  outputs
    mode_out: mode at every 1 sec;
  modes
    m0: mode = 0  "Normal";
    m1: mode = 1  "A failure"
      estimate a = b / 2;
    m2: mode = 2  "B failure"
      estimate b = 2 * a;
end
```

Fig. 7 Declarative specification of the Twice_Bayes PILOTS program.

4 Dynamic Bayes classifier

Naive Bayes classifiers [33,22] are commonly used in supervised training and classification. For continuous data, if the values of samples in each class are assumed to be normally distributed, the classifiers are called Gaussian naive Bayes classifiers [23]. In the training phase, tagged samples of different classes are processed to train the parameters of the classifier. The parameters include the mean value, standard variance, and prior probability of each class. In the testing phase, the trained Bayes classifier decides the class of untagged input samples.

One limitation of the traditional naive Bayes classifier is that the input samples in the testing phase will only be classified into classes that appeared in the training phase. If a sample of a previously unknown class appears, it will be classified into one of the known classes, even if the probability that it belongs to that class is very low. However, with dynamic stream data, new modes not in the training set could occur in some complex situations. For example, if a Bayes classifier is trained to recognize the "normal weight" and "underweight" modes of the weight indicator on an airplane during flights, and a previously unknown mode "overweight" appears in testing phase, the classifier will not be able to detect this new mode, but will classify the samples to "normal weight" or "underweight" based on the value and prior probability of the modes.

To tackle this limitation of the naive Bayes classifier, we extend it into a *dynamic Bayes classifier* that has two phases: (1) *Offline*: Supervised learning, which is the same as Gaussian naive Bayes classifiers. (2) *Online*: Unsupervised dynamic incremental learning, that classifies samples in known modes, updates parameters of the model, and create new modes for samples in previously unknown modes. Because we focus on processing data streams during flights and deciding the normal or error operational modes of an airplane, the words "mode" and "class" are used interchangeably and have the same meaning in this paper.

4.1 Offline supervised learning

4.1.1 Gaussian naive Bayes classifier

In a Gaussian naive Bayes classifier [23], each input sample X is described by a feature vector $(x_1, \ldots, x_n)$, and each sample is classified into a target class $y \in \{y_1, \ldots, y_m\}$. In this paper, we consider samples of only one feature x, but the results can be generalized to n features. By Bayes' theorem, the conditional probability $P(y|x)$ is:

$$P(y|x) = \frac{P(y)P(x|y)}{P(x)} \tag{5}$$

As the samples in each feature are assumed to be normally distributed, $P(x|y)$ is calculated by:

$$P(x|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(x-\mu_y)^2}{2\sigma_y^2}} \quad (6)$$

Where μ_y is the mean of the values in x associated with class y, and σ_y is the standard deviation of the values in x associated with class y.

The corresponding classifier $\hat{y}$ is:

$$\hat{y} = \arg\max P(y|x) \quad (7)$$

Because $P(x)$ is the same for each class, $\hat{y}$ is:

$$\hat{y} = \arg\max P(y|x) = \arg\max \ P(y)P(x|y) \quad (8)$$

4.1.2 Offline learning phase

In the offline supervised learning phase, input data tagged with mode labels are processed by a Gaussian naive Bayes classifier. The mean value μ_y, standard deviation σ_y, and the prior probability $P(y)$ of each mode y, are calculated by the classifier, as in Figure 8.

4.2 Dynamic online unsupervised learning

4.2.1 Major and minor modes

To support dynamically changing modes during the online learning phase, the concepts of *major* and *minor* modes are introduced. The modes in the offline supervised learning phase are major modes. In the prediction and online learning phase, before a sample is processed by the Bayes classifier, the value is checked by a preprocessor to decide if it is in the range of each major mode. As 95% of the values in a normal distribution lie within $\mu \pm 2\sigma$, if the sample is not within that range of any major mode, a new minor mode is created. As more data are processed, when the number of samples in a minor mode exceeds a threshold, it will be turned into a major mode. Minor modes are used to keep samples differentiated from known major modes. A threshold is used to diminish the impact of noise. Mode ID is automatically assigned when a new mode is detected.

4.2.2 Online learning phase

The process of dynamic online unsupervised learning is shown in Figure 8. The parameters of initial major modes are from the training results of the offline training phase. As untagged samples are processed, if the value is within $\mu \pm 2\sigma$ of any major mode, the sample will be classified by naive Bayes classifier, and the parameters are incrementally updated. If the value is not within $\mu \pm 2\sigma$ of any major mode, but is within $\mu \pm 2\sigma$ of a minor mode, it will be classified into the closest minor mode, and the parameters of minor modes are updated accordingly. Finally, if the value of the sample is not within $\mu \pm 2\sigma$ of any major or minor mode, a new minor mode will be created for this sample. σ of the new minor mode is initially set as the average σ of the existing major modes. When the size of the minor mode is greater than a threshold, we start to calculate and use the real σ of the minor mode. The reason is that the σ might be biased if the number of samples is too small. Each time when the parameters of a minor mode are updated, if the number of samples exceeds a certain threshold, it will be upgraded into a major mode.

5 Case study 1: Airplane weight estimation

To help prevent accidents caused by fuel quantity indicator errors such as the Tuninter 1153 flight accident, we use the X-Plane flight simulator to generate flight sensors data and simulate airplane weight error scenarios. Since during cruise phases, weight is equal to lift, we can estimate weight by using the lift equation from aerodynamics theory. Lift depends on airspeed, air density, wing surface area, and coefficient of lift. The coefficient of lift itself depends on the angle of attack and this relationship can change with different aircraft types. Figure 9 suggests a linear relationship between angle of attack and coefficient of lift for the expected range of angle of attack in cruise conditions. Therefore, we choose to use linear regression to predict coefficient of lift. Figure 11 confirms that a linear model was an appropriate choice.

We first estimate the actual weight based on error signatures with a linear model learned using linear regression in Section 5.1. Then, in Section 5.2, we use our dynamic Bayes classifier to detect erroneous weight and estimate the true weight.

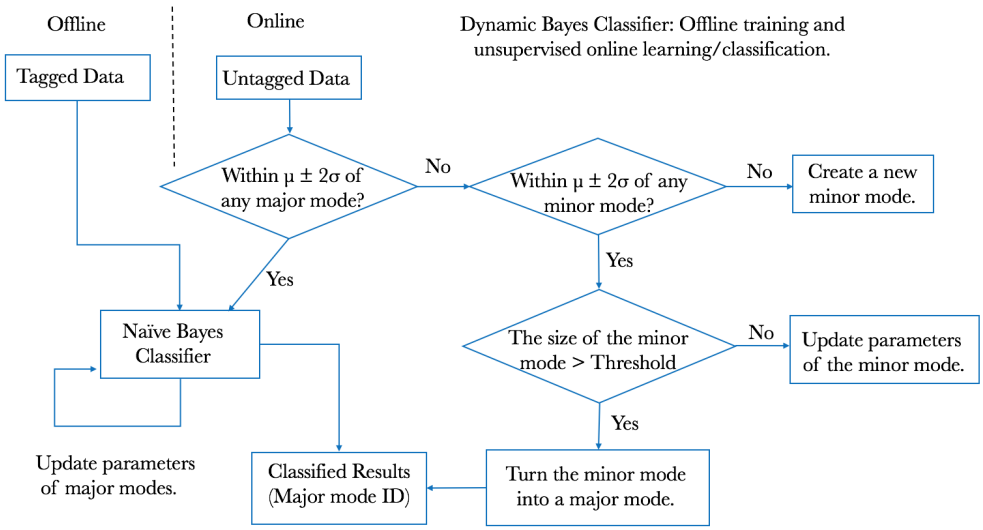

Fig. 8 Online classification and incremental learning using dynamic Bayes classifier.

5.1 Estimation by error signatures

5.1.1 Training weight model using linear regression

Weight model design In cruise phase, when yaw, roll angles are close to zero and pitch is small, we assume $L = W$, in which L is total lift and W is gross weight. Based on the assumption, we can estimate W by the lift equation:

$$W = L = \frac{1}{2}v^2 S\rho C_l, \tag{9}$$

where v is true air speed, S is wing surface area, ρ is air density and C_l is coefficient of lift. From ideal gas law, we know $\rho = \frac{p}{R'T}$, where p is ambient air pressure, R' is special gas constant and T is ambient temperature, and replace ρ with $\frac{p}{R'T}$ in Equation (9) to get:

$$W = \frac{pv^2 SC_l}{2R'T} \tag{10}$$

and by transforming Equation (10), C_l, coefficient of lift could be represented by:

$$C_l = \frac{2WR'T}{pv^2 S}. \tag{11}$$

Generally C_l depends on the shape of airfoil and the shape of an aircraft. To roughly estimate C_l, the complex physical model is simplified using Thin-Airfoil theory, which predicts a linear relationship [3] between coefficient of lift, C_l, and the angle of attack, α, for low values of α, as shown in Figure 9 between dashed vertical lines. This relationship can be expressed as:

$$C_l = \beta_1 \alpha + \beta_2 + \epsilon \tag{12}$$

where ϵ is noise and α is known while β_1 and β_2 are distinct values for different aircrafts. A linear model could be formulated as the following:

$$y = X\beta + \epsilon \tag{13}$$

$$y = \begin{pmatrix} C_{l_1} \\ C_{l_2} \\ \vdots \\ C_{l_n} \end{pmatrix}, C_{l_i} = \frac{2W_i R'T_i}{p_i v_i^2 S}, X = \begin{pmatrix} \alpha_1 & 1 \\ \alpha_2 & 1 \\ \vdots & \\ \alpha_n & 1 \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}$$

Because each column in X is independent, we could use the least squares method defined to retrieve $\hat{\beta}$, and predict $\hat{W}$ using the following equation:

$$\hat{W} = \frac{pv^2 S(\beta_1 \alpha + \beta_2)}{2R'T} \tag{14}$$

where we have substituted the linear estimation of C_l, in Equation (10).

Experimental settings Synthetic data with simple relationships are used to verify the integration of machine learning approaches into PILOTS. In this example, simulated ATR-72 500 airplane data is used for PILOTS to detect and correct for weight error in the data streams, and the relation between coefficient of lift and angle of attack is investigated. To simulate and test linear regression implemented in PILOTS machine learning component, we assume certain known variables. It is required that the following variables are correctly measured and known: gross weight W, ambient pressure p, true airspeed v, wing surface area S, special gas constant for dry air R', and ambient temperature T.

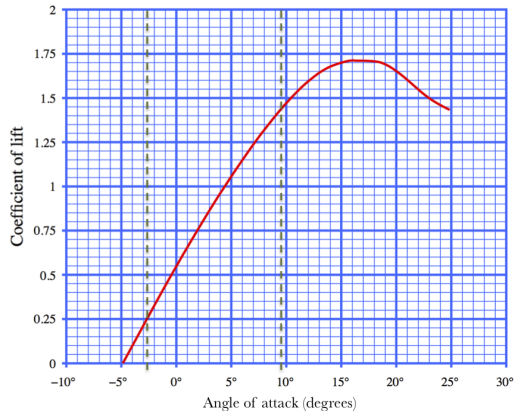

Fig. 9 Coefficient of lift as a function of angle of attack for a cambered airfoil, adapted from https://en.wikipedia.org/wiki/Lift_coefficient.

A linear model to predict coefficient of lift C_l is trained with synthetic data using offline training definition file as shown in Figure 10. The trainer file will generate a prediction model with the ID cl_regression. In this case, a linear regression model will be trained by data defined in data section, in which also the variables are initialized. schema defines the additional attributes to the variables and preprocess defines the actions to be taken on input dataset before training. The generated model will be used for estimation of coefficient of lift from angle of attack.

```
trainer cl_regression;
  /* v: true air speed,
     a: angle of attack,
     p: pressure, t: temperature,
     w: gross weight, S: wing area,
     R: Gas constant
   */
  constants
    R = 286.9;
    S = 61.0;
  data
    v, p, t, w, a using file(weightcheck.csv );
    schema
      unit(v:knot, p:in_Hg, t:celsius,
        w:force_pound, a:degree);
  model
    preprocess
      changeunit using
        unit(v:m/s, p:pascal, t:kelvin,
             w:newton, a:radian);
    features: a;
    labels: 2*w/(v^2 * (p/R/t)*S);
    algorithm: LinearRegression;
end
```

Fig. 10 Offline training parameters for coefficient of lift prediction model with linear regression.

Training result Figure 11 shows the training result of linear relationship between angle of attack and coefficient of lift, where the learned parameters are $\beta_1 = 6.3888$ and $\beta_2 = 0.3589$. The evaluation of the trained model gives $R^2 = 0.9949, RMSE = 0.00794$, showing a strong linear relationship with low in-sample error. Using Equation (9), we compute the training error between measured weight and estimated weight, resulting in $RMSE = 2687N$.

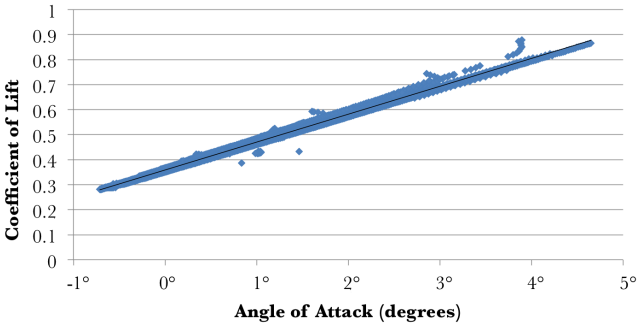

Fig. 11 The linear relation between angle of attack and coefficient of lift in cruise phase of training set.

5.1.2 Performance metrics for error mode detection

We evaluate the performance of error detection based on *accuracy* and *response time*, which are defined as follows:

- *Accuracy*: This metric is used to evaluate how accurately the algorithm determines the true mode. Assuming the true mode transition $m(t)$ is known for $t = 0, 1, 2, \ldots, T$, let $m'(t)$ for $t = 0, 1, 2, \ldots, T$ be the mode determined by the error detection algorithm. We define $accuracy(m, m') = \frac{1}{T}\sum_{t=0}^{T} p(t)$, where $p(t) = 1$ if $m(t) = m'(t)$ and $p(t) = 0$ otherwise.
- *Maximum/Minimum/Average Response Time*: This metric is used to evaluate how quickly the algorithm reacts to mode changes. Let a tuple (t_i, m_i) represent a mode change point, where the mode changes to m_i at time t_i. Let

$$M = \{(t_1, m_1), (t_2, m_2), \ldots, (t_N, m_N)\},$$

and

$$M' = \{(t'_1, m'_1), (t'_2, m'_2), \ldots, (t'_{N'}, m'_{N'})\},$$

where M and M' are the sets of true mode changes and detected mode changes respectively. To remove noise in the detected changes, we further apply the following operation to M': if $t'_{i+1} - t'_i \leqslant \delta$, remove a tuple (t'_{i+1}, m'_{i+1}) from M', where δ is a small

threshold value to eliminate frequent mode changes. For each $i = 1 \ldots N$, we can find the smallest t'_j such that $(t_i \leqslant t'_j) \wedge (m_i = m'_j)$; if not found, let t'_j be t_{i+1}. The response time r_i for the true mode m_i is given by $t'_j - t_i$. We define the maximum, minimum, and average response time by $\max_{1 \leqslant i \leqslant N} r_i$, $\min_{1 \leqslant i \leqslant N} r_i$, and $\frac{1}{N} \sum_{i=1}^{N} r_i$ respectively.

5.1.3 Error detection and correction using error signatures

Error signatures The error function e is given by Equation (15) as the percentage of discrepancy between predicted weight $\hat{W}$ and measured weight W. The error signatures use a threshold of 3.5% because this number is more rigorous than the percentage of discrepancy between error weight and actual weight in Tuninter 1153 accident, which is about 10%.

$$e = \frac{W - \hat{W}}{W} \tag{15}$$

Mode	Error Signature	
	Function	Constraints
Normal	$e = k$	$-0.035 < k < 0.035$
Overweight	$e = k$	$k > 0.035$
Underweight	$e = k$	$k < -0.035$

Table 2 Error signatures set for weight correction.

PILOTS program The WeightCheck_Signatures PILOTS program implementing the error signatures in Table 2 is shown in Figure 12. If the error signature s1 or s2 is detected, the program estimates weight using Equation (14). The data selection module computes v', a', p', te', w' using data points with the closest time stamp, and uses a' as an input matrix to predict cl' using model the with the ID cl_regression.

Experimental settings X-Plane 9.7 is used to generate flying data of ATR72-500 in different altitudes, gross weights and power settings. The data is split by selecting three flights' 25 cruise phases, 1251 minutes in total, as training set, and one 20-minutes flight with four cruise phases as testing set. The model is trained by 25 cruise phases in the training set and tested by four cruise phases in the testing set. To evaluate the PILOTS error detection accuracy, the whole testing set is modified to introduce artificial measurement errors as follows: weight data in the range from 1 to 100 and 750 to 800 seconds is multiplied by 0.9, from 1025 to 1099 seconds is multiplied by 1.1, from 390 to 490 seconds is multiplied by 1.05, from 570 to 648 seconds is multiplied by normal distribution of error with mean at 1 and standard deviation at 0.1, from 291 to 377 seconds are multiplied by uniform distribution of error ranging from 0.9 to 1.1. The cruise phases of testing set lie between 5 to 164, 230 to 395, 470 to 688 and 780 to 1108 seconds. We can visualize this data as "measured" in Figure 13.

```
program WeightCheck_Signatures;
  /* v: true air speed [m/s],
     a: angle of attack [rad],
     p: pressure [Pa], te: temperature [K],
     w: gross weight [N], S: wing area [m^2],
     R: Gas constant [J kg^-1 K^-1],
     cl: coefficient of lift
   */
  constants
    R = 286.9;
    S = 61.0;
  inputs
    v, a, p, te, w (t) using closest(t);
    cl (t) using model(cl_regression, a);
  outputs
    corrected_weight: w at every 1 sec;
  errors
    e: (w - p*(v*v)*S*cl/(2*R*te)) / w;
  signatures
    s0: e = K, -0.035 < K < 0.035   "Normal";
    s1: e = K, K > 0.035            "Underweight"
      estimate w = p*(v*v)*S*cl/(2*R*te);
    s2: e = K, K < - 0.035          "Overweight"
      estimate w = p*(v*v)*S*cl/(2*R*te);
end
```

Fig. 12 Declarative specification of WeightCheck_Signatures PILOTS program using error signature.

PILOTS program WeightCheck_Signatures in Figure 12 is executed with different combinations of window sizes $\omega \in \{1, 2, 4, 8, 16\}$ and thresholds $\tau \in \{0.2, 0.4, 0.6, 0.8, 0.99\}$ to investigate the accuracy and average response time, which is computed using $\delta = 0$, because of the instability of error modes in the dataset.

Results Figure 13 shows the estimated weight and measured weight during the 18 minutes flight where $\omega = 1$ and $\tau = 0.99$, the best combination among all combinations in accuracy and response time. The PILOTS program can successfully detect and correct underweight and overweight conditions in cruise phases, with root mean squared error close to 1617N. The program performs the best in system failure simulation regions of cruise phase 1 and 4, where the weight drifts by 10% or 5%. The accuracy is 100% and max/min/average response time is 0/0/0 seconds. In random error simulation regions of cruise phase 2 and 3, performance is decreased to have accuracy 89.1% and max/min/average response time 84/0/1.76 seconds. The overall accuracy is 95.2% and max/min/average response time is 84/0/1.65 seconds. Outside cruise phases, the program

does not estimate weight properly since the assumption $L = W$ does not hold.

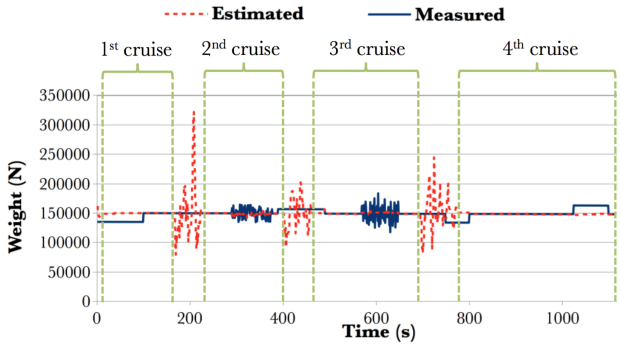

Fig. 13 Error detection and correction using $\omega = 1, \tau = 0.99$ for X-Plane simulated data.

```
trainer weightcheck_mode_bayes;
  /* v: true air speed,
     a: angle of attack,
     p: pressure, t: temperature,
     w: gross weight, S: wing area,
     R: Gas constant, mode: labeled mode for training,
     cl: coefficient of lift
   */
  constants
    S = 61.0;
    R = 286.9;
  data
    v, p, t, w, a, mode using file(weightcheck.csv);
    cl using model(cl_regression, a);
    schema
      unit(v:knot, p:in_Hg, t:celsius,
        w:force_pound, a:degree);
  model
    preprocess
      changeunit using
        unit(v:m/s, p:pascal, t:kelvin,
          w:newton, a:radian);
    features: w - (1/2)*v^2*S*(p/(R*t))*cl;
    labels: mode;
    algorithm:
      DynamicBayesClassifier(
       sigma_scale:2, threshold:100);

end
```

Fig. 14 Offline training parameters for the dynamic Bayes classifier.

```
program WeightCheck_Bayes;
  /* v: true air speed [m/s],
     a: angle of attack [rad],
     p: pressure [Pa], te: temperature [K],
     w: gross weight [N], S: wing area [m^2],
     R: Gas constant [J kg^-1 K^-1],
     cl: coefficient of lift
   */
  constants
    S = 61.0;
    R = 286.9;
  inputs
    v, a, p, te, w (t) using closest(t);
    cl (t) using model(cl_regression, a);
    mode (t) using model(weightcheck_mode_bayes,
                  v, p, te, w, cl);
  modes
    m0: mode = 0 "Normal";
    m1: mode = 1 "Underweight"
      estimate w = p*(v*v)*S*cl/(2*R*te);
end
```

Fig. 15 A declarative specification of the WeightCheck_Bayes PILOTS program using the dynamic Bayes classifier.

5.2 Estimation by dynamic Bayes classifier

5.2.1 Offline training

We use $W - \hat{W}$ as the feature for the dynamic Bayes classifier. Estimated weight is calculated by Equation (9) using the method described in Section 5.1.1. The dynamic Bayes classifier is trained with both "normal" and "underweight" tagged data in the offline learning phase. Figure 14 shows the parameters setting for the offline training phase. PILOTS uses this trainer file to generate a prediction model with the ID weightcheck_mode_bayes, in which the model is defined in model with dynamic Bayesian classifier using sigma scale 2, which represents $\mu \pm 2\sigma$ for range of a mode for classification, and threshold 100 for turning a minor mode into a major mode. The prediction model cl_regression is used for estimating $\hat{C}_l$, which is then used for estimation of $\hat{W}$.

5.2.2 Online error detection and correction

PILOTS program The WeightCheck_Bayes PILOTS program is shown in Figure 15. This program is used for the online learning and classification to detect and give weight estimation to different weight error modes.

Experimental settings We use the same performance metrics for major mode prediction: accuracy and response time as in Section 5.1.2.

See Section 5.1.3 for data generation. We use the same testing data, and 8000 seconds training data in cruise phase modified as follows: weight data in the range from 1526 to 3129 second are multiplied by 1.1 to simulate underweight mode. There are two major modes in the tagged training data: mode 0 for normal status and mode 1 for underweight status. For online learning, we set the threshold of the sample number to turn a minor mode into a major mode to 100. Instead of using average σ, the sample number threshold for calculating σ of a new mode is also set as 100. Figure 16 (a) shows the feature and Figure 16 (b) shows tagged mode of training data. The major modes of the classifier after training phase is shown in Table 3 where n is the number of data point classified to a mode.

Results Figure 17 and Table 4 show the results of weight error mode detection by dynamic Bayes classifier. Using the same testing data as in Figure 13, the dynamic Bayes classifier successfully detects three major modes in the cruise phases: mode 0 for normal, mode 1 for underweight, and mode 3 for overweight. Mode 0 and mode 1 are major modes that appeared in the tagged training data, mode 3 is a new major mode detected by the classifier during the online incremental learning and prediction phase. There are also 22 minor modes generated by the noise and non-cruise phase data in the testing set. Using the metric described in Section 5.1.2 on only major modes, in systematic failure simulation regions of cruise phase 1 and 4, the accuracy is 95.3% and max/min/average response time is 19/0/3.83 seconds, while in random error simulation regions of cruise phase 2 and 3, the accuracy is largely decreased to 66.7% and max/min/average response time is increased to 98/0/5.38 seconds. The overall accuracy of major mode detection on all cruise phases is 82.7% and max/min/average response time is 98/0/5.28 seconds. Figure 18 illustrates the result of weight estimation in detected underweight mode. When in underweight mode and cruise phase, the estimation gives root mean squared error of approximately 1420N.

Table 3 Major modes of dynamic Bayes classifier after running training dataset.

Mode description	Mode	μ	σ	n
Normal	0	49	2534	5937
Underweight	1	16068	3249	1604

Table 4 Major and minor modes of dynamic Bayes classifier after error mode detection on testing dataset.

Mode status	Mode description	Mode	μ	σ	n
Major	Normal	0	-50	2481	6522
	Underweight	1	15961	3201	1746
	Overweight	3	-14472	1151	121
Minor	Noise	2, 4-24	N/A	N/A	N/A

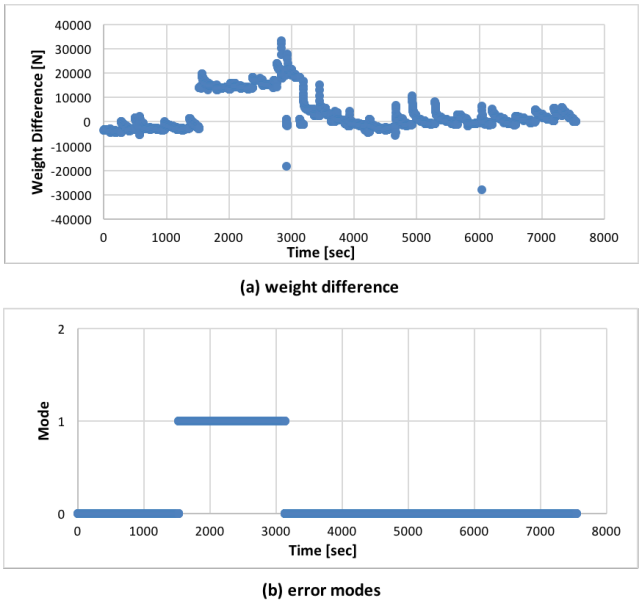

Fig. 16 Weight error mode training data for the dynamic Bayes classifier.

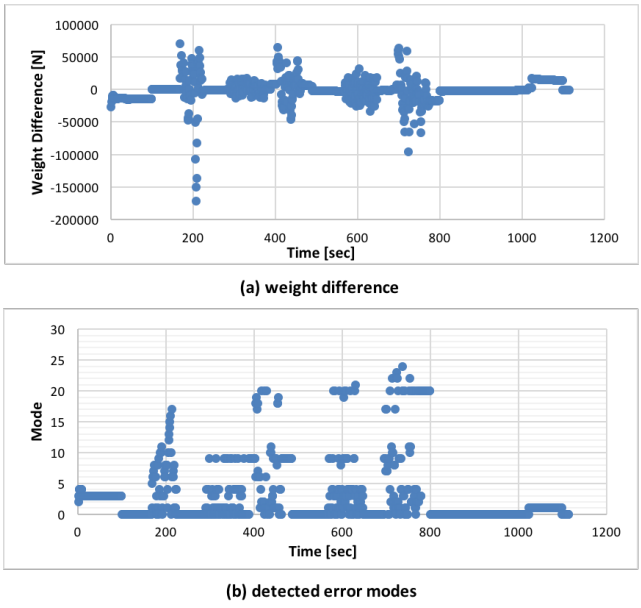

Fig. 17 Weight error mode detection using dynamic Bayes classifier.

5.3 Summary of results

In systematic failure simulation regions, the average response time of the error signatures approach with 0.035 as threshold, $\omega = 1$, and $\tau = 0.99$, is 3.83 seconds shorter than that of the dynamic Bayes classifier, and the error signatures approach is 4.7% more accurate than the dynamic Bayes classifier. And in random error simulation regions, the error signature approach produces 3.62 seconds shorter average response time and 22.4% more accurate than those of the dynamic Bayes classifier. One of the reasons is that in random error simulation regions, many minor modes are created instead of major modes, resulting in poor performance of major modes on the regions. However, the dynamic Bayes classifier discovers discrete error modes dynamically and automatically while the error signatures approach is static, that is, every signature must be predefined manually.

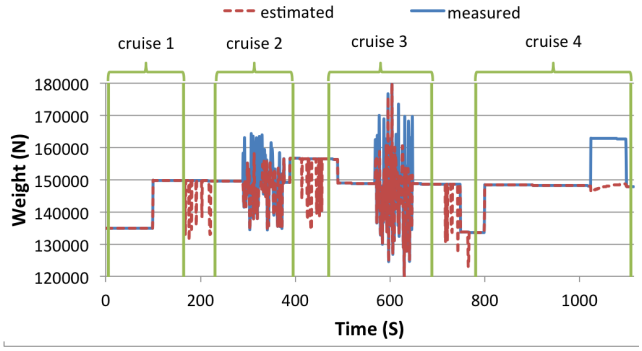

Fig. 18 Weight error correction using dynamic Bayes classifier.

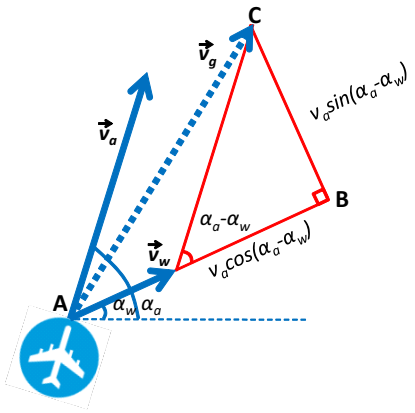

Fig. 19 Trigonometry applied to the ground speed, airspeed, and wind speed.

6 Case study 2: Airplane speed estimation

To help prevent accidents caused by pitot tube failures such as the Air France 447 flight accident, we use actual speed data recovered from the Air France 447's flight data recorder for evaluation. First, we estimate the true airspeed based on error signatures that we derive from a geometrical model in Section 6.1. Next, we train our dynamic Bayes classifier with synthetic data streams which consist of airspeed, ground speed, and wind speed and use it to estimate the correct airspeed in Section 6.2.

6.1 Estimation by error signatures

6.1.1 Derivation of speed checking model

Unlike the weight model in Section 5, the relationship between airplane speeds can be modeled in a simpler way. The relationship between ground speed, airspeed, and wind speed follows the following formula:

$$\overrightarrow{v_g} = \overrightarrow{v_a} + \overrightarrow{v_w}. \tag{16}$$

where $\overrightarrow{v_g}, \overrightarrow{v_a}$, and $\overrightarrow{v_w}$ represent the ground speed, the airspeed, and the wind speed vectors. A vector $\overrightarrow{v}$ can be defined by a tuple (v, α), where v is the magnitude of $\overrightarrow{v}$ and α is the angle between $\overrightarrow{v}$ and a base vector. Following this expression, $\overrightarrow{v_g}, \overrightarrow{v_a}$, and $\overrightarrow{v_w}$ are defined as $(v_g, \alpha_g), (v_a, \alpha_a)$, and (v_w, α_w) respectively as shown in Figure 19.

To examine the relationship in Equation (16), we can compute $\overrightarrow{v_g}$ by applying trigonometry to $\triangle ABC$ as shown in Figure 19. We can define an error function as the difference between measured v_g and computed v_g as follows:

$$\begin{aligned} e(\overrightarrow{v_g}, \overrightarrow{v_a}, \overrightarrow{v_w}) &= |\overrightarrow{v_g}| - |\overrightarrow{v_a} + \overrightarrow{v_w}| \\ &= v_g - \sqrt{v_a^2 + 2v_a v_w \cos(\alpha_a - \alpha_w) + v_w^2}. \end{aligned} \tag{17}$$

6.1.2 Error detection and correction using error signatures

Error signatures We consider the following four error modes: 1) normal (no error), 2) pitot tube failure due to icing, 3) GPS failure, 4) both pitot tube and GPS failures. Suppose the airplane is flying at airspeed v_a. For computing error signatures for different error conditions, we will assume that other speeds as well as failed airspeed and ground speed can be expressed as follows.

- ground speed: $v_g \approx v_a$.
- wind speed: $v_w \leqslant a v_a$, where a is the wind to airspeed ratio.
- pitot tube failed airspeed: $b_l v_a \leqslant \bar{v}_a \leqslant b_h v_a$, where b_l and b_h are the lower and higher values of pitot tube clearance ratio and $0 \leqslant b_l \leqslant b_h \leqslant 1$. 0 represents a fully clogged pitot tube, while 1 represents a fully clear pitot tube.
- GPS failed ground speed: $\bar{v}_g = 0$.

We assume that when a pitot tube icing occurs, it is gradually clogged and thus the airspeed data reported from the pitot tube also gradually drops and eventually remains at a constant speed while iced. This resulting constant speed is characterized by ratio b_l and b_h. On the other hand, when a GPS failure occurs, the ground speed suddenly drops to zero. This is why we model the failed ground speed as $\bar{v}_g = 0$.

In the case of pitot tube failure, let the ground speed, wind speed, and airspeed be $v_g = v_a, v_w = a v_a$,

and $\bar{v}_a = bv_a$. The error function (17) can be expressed as follows:

$$e = v_a - \sqrt{v_a^2(b^2 + 2ab\cos(\alpha_a - \alpha_w) + a^2)}.$$

Since $-1 \leqslant \cos(\alpha_a - \alpha_w) \leqslant 1$, the error is bounded by the following:

$$v_a - \sqrt{v_a^2(a+b)^2} \leqslant e \leqslant v_a - \sqrt{v_a^2(a-b)^2}$$
$$(1-a-b)v_a \leqslant e \leqslant (1-|a-b|)v_a. \quad (18)$$

In the case of GPS failure, let the ground speed, wind speed, and airspeed be $\bar{v}_g = 0, v_w = av_a$, and $v_a = v_a$. The error function (17) can be expressed as follows:

$$e = 0 - \sqrt{v_a^2(1 + 2a\cos(\alpha_a - \alpha_w) + a^2)}.$$

Similarly to the pitot tube failure, we can derive the following error bounds:

$$-(a+1)v_a \leqslant e \leqslant -|a-1|v_a. \quad (19)$$

We can derive error bounds for the normal and both failure cases similarly. Applying the wind to airspeed ratio a and the pitot tube clearance ratio $b_l \leqslant b \leqslant b_h$ to the constraints obtained in Inequalities (18) and (19), we get the error signatures for each error mode as shown in Table 5.

Table 5 Error signatures for speed data.

Mode	Error Signature	
	Function	Constraints
Normal	$e = k$	$k \in [-av_a, av_a]$
Pitot tube failure	$e = k$	$k \in [(1-a-b_h)v_a, (1-\lvert a-b_l\rvert)v_a]$
GPS failure	$e = k$	$k \in [-(a+1)v_a, -\lvert a-1\rvert v_a]$
Both failures	$e = k$	$k \in [-(a+b_h)v_a, -\lvert a-b_l\rvert v_a]$

When $a = 0.1, b_l = 0.2$, and $b_h = 0.33$, the error signatures shown in Table 5 are visually depicted in Figure 20.

PILOTS program The SpeedCheck_Signatures PILOTS program implementing the error signatures shown in Table 5 is presented in Figure 21 [2]. This program checks if the wind speed, airspeed, and ground speed are correct or not, and computes a crab angle, which is used to adjust the direction of the aircraft to keep a desired ground track. For this program to be applicable to the aircraft used for Air France 447, we use a cruise speed of 470 knots as v_a.

[2]We have not implemented the when clause support in PILOTS version 0.4 as of June 2017.

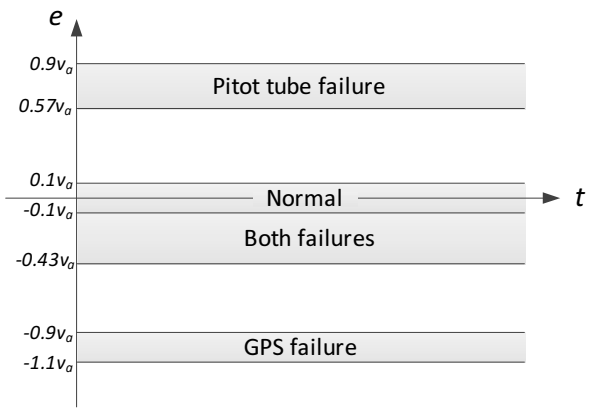

Fig. 20 Error Signatures for speed data ($a = 0.1, b_l = 0.2$, and $b_h = 0.33$).

```
program SpeedCheck_Signatures;
  /* va: airspeed,      aa: airspeed angle,
     vw1: wind speed,
     aw1: wind speed angle,
     vw2: estimated wind speed,
     aw2: estimated wind speed angle,
     vg: ground speed,  ag: ground angle */
  inputs
    va, vg, vw1 (t) using closest(t);
    aa, ag, aw1 (t) using closest(t);
  outputs
    va_out: va at every 1 sec;
    vw2: sqrt(va*va + vw2*vw2 -
        2*va*vw2*cos((PI/180)*(ag-aa)))
            at every 1 sec when s0 10 times;
    aw2: cos((PI/180)*(ag-aa)
            at every 1 sec when s0 10 times;
  errors
    e: vg - sqrt(va*va + vw1*vw1 +
          2*va*vw1*cos((PI/180)*(aw-aa)));
  signatures
    /* v_a=470, a=0.1, b=0.2...0.33 */
    s0: e = K, -47 < K, K < 47    "No error";
    s1: e = K, 220.9 < K, K < 517 "Pitot tube failure"
      estimate va =
        sqrt(vg*vg + vw2*vw2 -
         2*vg*vw2*cos((PI/180)*(ag-aw2)));
    s2: e = K, -517 < K, K < -423 "GPS failure"
      estimate vg =
        sqrt(va*va + vw2*vw2 +
            2*va*vw2*cos((PI/180)*(aw2-aa)));
    s3: e = K, -203.66 < K, K < -47
      "Pitot tube + GPS failure";
end
```

Fig. 21 Declarative specification of the SpeedCheck_Signatures PILOTS program.

Experimental settings The ground speed and airspeed are collected based on Appendix 3 in the final report of Air France flight 447 [6].

Note that the (true) airspeed was not recorded in the flight data recorder so that we computed it from recorded Mach (M) and static air temperature (SAT) data. The airspeed was obtained by using the relationship: $v_a = a_0 M\sqrt{SAT/T_0}$, where a_0 is the speed of sound at standard sea level (661.47 knots) and T_0 is the temperature at standard sea level (288.15 Kelvin).

Independent wind speed information was not recorded either. According to the description from page 47 of the final report: "(From the weather forecast) the wind and temperature charts show that the average effective wind along the route can be estimated at approximately *ten knots tail-wind*." We followed this description and created the wind speed data streams (*i.e.*, vw1 and aw1 in Figure 21) as ten knots tail wind.

While wind speed data streams from the accident report are useful to compute the error, it may be more accurate to use the wind data derived from measured ground speed and airspeed [15]. Since wind speed data can be estimated by $\overrightarrow{v_w} = \overrightarrow{v_g} - \overrightarrow{v_a}$, we can save this estimated wind speed as long as we detect the normal mode consistently. In Figure 21, we define vw2 and aw2 as the estimated wind speed streams, which values are updated when we observe the normal mode (*i.e.*, s0 signature is matched) for 10 times. Once we detect an error in airspeed (va) or ground speed (vg), we use vw2 and aw2 to recompute estimated airspeed or ground speed respectively.

According to the final report, speed data was provided from 2:09:00 UTC on June 1st 2009 and it became invalid after 2:11:42 UTC on the same day. Thus, we examine the valid 162 seconds of speed data including a period of pitot tube failure which occurred from 2:10:03 to 2:10:36 UTC. so the set of true mode changes is defined as $M = \{(1,0),(64,1),(98,0)\}$. The accuracy and average response time are investigated for window sizes $\omega \in \{1,2,4,8,16\}$ and threshold $\tau \in \{0.2,0.4,0.6,0.8\}$ with the true mode changes $M = \{(1,0),(64,1),(98,0)\}$.

Results From the metric described in Section 5.1.2 with $\delta = 5$ for response time, the best results, accuracy = 94.4%, max/min/average response times = 5/0/1.67 seconds, are observed when $\omega = 1$ and $\tau = 0.8$. The transitions of the corrected speed and detected modes that show the best accuracy with $\omega = 1$ and $\tau = 0.8$ are shown in Figure 22. Looking at Figure 22(b), the pitot tube failure is successfully detected from 70 to 96 seconds except for the interval 63 to 69 seconds due to the slowly decreasing airspeed. The response time for the normal to pitot tube failure mode is 5 seconds and for the pitot tube failure to normal mode is 0 seconds (thus the average response time is 1.67 seconds). From Figure 22(a), the airspeed successfully starts to get corrected at 70 seconds and seamlessly transitions to the normal airspeed when it recovers at 99 seconds.

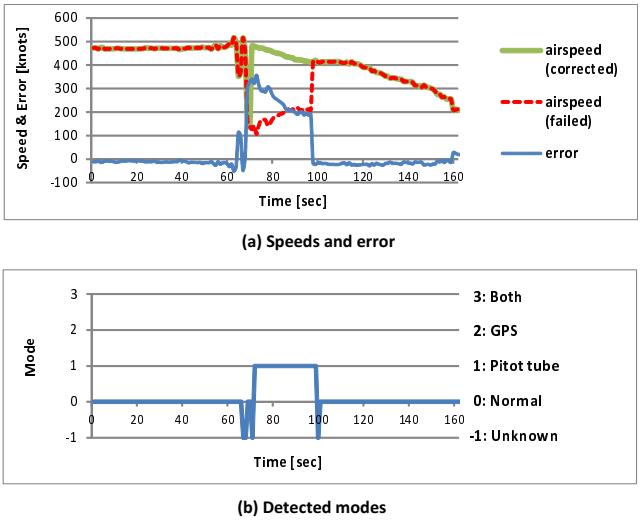

Fig. 22 Corrected airspeed and detected modes for AF447 flight.

6.2 Estimation by dynamic Bayes classifier

6.2.1 Offline training

We create a training dataset for three error modes (*i.e.*, normal, pitot tube failure, GPS failure) by simulation using the parameters as shown in Table 6. Note that all the speeds are relative to airspeed v_a and all the angles are in degrees. $\mathcal{N}(\mu,\sigma)$ is a normal distribution with mean μ and standard deviation σ. $\mathcal{U}(a,b)$ is a uniform distribution that randomly choose a number between $[a,b]$. ϵ is a Gaussian measurement noise. $f(v_a,\alpha_a,v_w,\alpha_w)$ gives ground speed v_g such that the error defined in Equation (17) becomes zero. Similarly, $g(v_a,\alpha_a,v_w,\alpha_w)$ gives ground speed angle α_g such that an error defined for angles becomes zero.

Table 6 Simulation parameters to generate speed estimation model training data.

Parameters	Error modes		
	Normal	Pitot tube failure	GPS failure
Number of samples	10,000	10,000	10,000
v_a: airspeed	$1.0+\epsilon$	$\mathcal{N}(0.5,0.067)$	$1.0+\epsilon$
α_a: airspeed angle	$45+\epsilon$		
v_w: wind speed	$0.1+\epsilon$		
α_w: wind speed angle	$\mathcal{U}(0,360)$		
v_g: ground speed	$f(v_a,\alpha_a,v_w,\alpha_w)+\epsilon$		$0.0+\epsilon$
α_g: ground speed angle	$g(v_a,\alpha_a,v_w,\alpha_w)+\epsilon$		

Similar to what we define the error signatures set for speed data in Section 6.1.2, we make some assumptions on how pitot tube and GPS failures occur. In the case of pitot tube failure, we assume the pitot

tube clearance ratio b has some randomness and follows the normal distribution $\mathcal{N}(0.5, 0.067)$. We define failed airspeed $\bar{v}_a = b \cdot v_a$, but since $v_a = 1.0$, v_a follows $\mathcal{N}(0.5, 0.067)$. In the case of GPS failure, the ground speed drops to zero with a small measurement error ϵ.

Offline training of the dynamic Bayes classifier is performed with training parameters in Figure 23, where the feature is chosen to be the difference between measured ground speed and estimated ground speed calculated using other variables.

```
trainer speedcheck_mode_bayes;
  /* vw: wind speed, aw: wind speed angle
     va: airspeed,   aa: airspeed angle
     vg: ground speed, ag: ground angle,
     mode: the error mode
   */
  data
    vw, va, vg, aa,
    ag, aw, mode using file(speedcheck.csv);
  model
    features: vg - sqrt(va^2 + vw^2
      + 2*va*vw*cos((PI/180)*(aw-aa)));
    labels: mode;
    algorithm:
       DynamicBayesClassifier(
        sigma_scale:2, threshold:100);
end
```

Fig. 23 Offline training parameters for the dynamic Bayes classifier.

6.2.2 Online error detection and correction

PILOTS program Figure 24 shows the SpeedCheck_Bayes PILOTS program using the dynamic Bayes classifier to detects three modes: Normal, Pitot tube failure, and GPS failure. The program outputs the airspeed or estimated airspeed depending on the detected mode.

Experimental settings The same dataset used in error signature experiment in Section 6.1.2 is utilized as testing set for PILOTS program described in Figure 24 with trained dynamic Bayes classifier in Section 6.2.1. For dynamic Bayes classifier, we set the threshold of creating new mode as 2σ and threshold of converting minor mode to major mode as 200. Table 7 shows the major modes contained in the dynamic Bayes classifier.

Results Using Section 5.1.2 as metric with delta = 5 for response time, the accuracy is 91.0%, max/min/average response times is 11/0/3.67 seconds. Since the dynamic Bayes classifier changes its states as new data comes in, the updated states are described in Table 8. Because the training set is large and only the data points residing

```
program SpeedCheck_Bayes;
  /* vw: wind speed, aw: wind speed angle
     va: airspeed,   aa: airspeed angle
     vg: ground speed, ag: ground angle */
  inputs
    va, vg, vw (t) using closest(t);
    aa, ag, aw (t) using closest(t);
    mode (t) using model(speedcheck_mode_bayes,
           va, vg, vw, aw, aa);
  outputs
    va_out: va at every 1 sec;
  modes
      m0: mode = 0 "Normal";
      m1: mode = 1 "Pitot tube failure"
          estimate va =
             sqrt(vg*vg + vw*vw
             - 2*vg*vw*cos((PI/180)*(ag-aw)));
      m2: mode = 2 "GPS failure"
          estimate vg =
             sqrt(va*va + vw*vw
             + 2*va*vw*cos((PI/180)*(aw-aa)));
end
```

Fig. 24 Declarative specification of the SpeedCheck_Bayes PILOTS program.

Table 7 Major modes of dynamic Bayes classifier after trained by training data.

Mode description	Mode	μ	σ	n
Normal	0	0.11	14.19	10000
Pitot tube failure	1	233.61	39.50	10000
GPS failure	2	-471.14	39.78	10000

inside 2σ of a mode are capable of changing attributes of the mode, for major modes with id 0 to 2, the standard deviation and mean stay nearly the same. The classifier discovers new modes with id 3 to 5, which may suggest some unconsidered cases are not covered in the training dataset or the threshold of creating new mode is low and results in noises and outliers to become new modes. Since the threshold of converting minor mode to major mode is 200, no new major modes are created.

The measured and estimated airspeed is illustrated in Figure 25(a) and the detected modes are shown in Figure 25(b). Pitot tube failure is captured and the airspeed is estimated during 70, 75 to 97 seconds but not 64 to 69, 71 to 74 seconds since the error function gives numbers greater than 2σ of every state in trained dynamic Bayes classifier, and being classified to new modes. The classifier successfully detects end of pitot tube failure immediately under the resolution of 1 second. Interestingly, at 124 second, the classifier gives mode 3, which is created at 62 second.

6.3 Summary of results

The max/min/average response times for both dynamic Bayes classifier and error signatures method for PILOTS program are identical. However, since the dynamic Bayes classifier contains newly created modes

Table 8 Major and minor modes of dynamic Bayes classifier after running testing data.

Mode status	Mode description	Mode	μ	σ	n
Major	Normal	0	-0.07	14.0	10130
	Pitot tube failure	1	233.62	39.50	10024
	GPS failure	2	-471.14	39.78	10000
Minor	New mode	3	-38.26	9.59	5
	New mode	4	104.01	8.80	2
	New mode	5	332.25	14.30	5

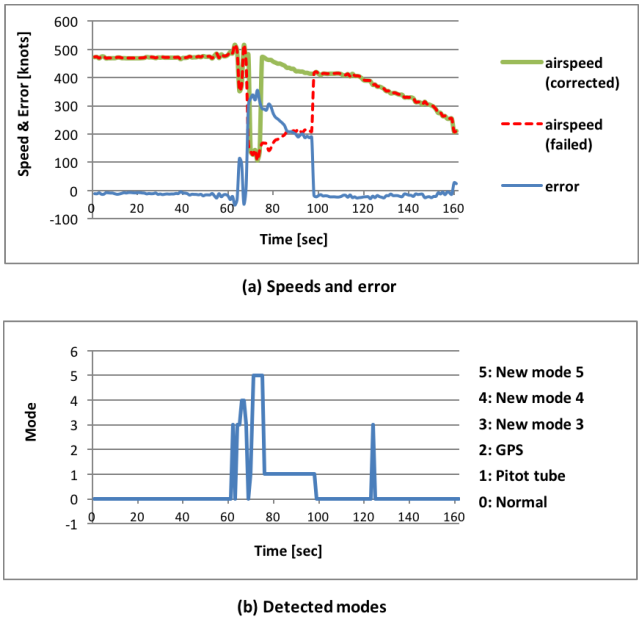

Fig. 25 Speed error mode detection and correction using dynamic Bayes classifier.

that have no corresponding estimation function provided, the failure mode accuracy is less than the one produced by error signatures, and thus, the estimation of airspeed is less complete than that computed by error signatures. The dynamic Bayes classifier discovers new modes, which could be attached to a new estimation function, and requires offline training, while the error signatures are not able to change dynamically but have no requirement of training.

7 Related work

There is a large body of work in data-driven anomaly detection. Mack et al. applied a data mining approach to flight data to obtain aircraft failure models in the form of a Bayesian network [28, 27]. Scalable Bayesian Network Learning (SBNL) [37] is a workflow framework to learn Bayesian networks in a scalable manner based on data-parallel ensemble learning. The map-reduce like learning process is built using the Kepler scientific workflow management system [26]. While Bayesian networks can represent probabilistic relationships between multiple random variables in a graphical way, our dynamic Bayes classifier is derived from a single probability function. However, since the idea of detecting new modes used in the dynamic Bayes classifier is orthogonal to Bayesian networks, dynamic new mode detection and Bayesian networks can complement each other. Biswas et al. proposed a method to detect anomalies for spacecraft operation using both unsupervised and supervised learning [4]. They first detect new operating modes or potential anomalies by clustering input data, and then use supervised learning to distinguish new operating modes from anomalies. Their approach is similar to our dynamic Bayes classifier in that both approaches detect outliers as potential new modes. They consult experts from NASA to confirm if there are anomalies in the detected new modes; we also argue for semi-supervised learning in our discussion (Section 8). Das et al. use a multiple kernel learning method to incorporate discrete (categorical) and continuous data, which are commonly observed in recorded flight data, for anomaly detection [10]. Our PILOTS framework can convert continuous data into discrete data and its modular approach enables plugging in different machine learning algorithms, so Das et al.'s work could be modeled using our framework.

There are several domain-specific languages (DSLs) for machine learning (ML) and parallel processing [32]. OptiML [35] is a DSL designed for ML and can generate target code for heterogeneous machines (*i.e.*, CPUs and GPUs). OptiML programs support vector, matrix, and graph data types and provide a functional programming style grammar. ScalOps [38] is a DSL for data analytics, which enables map-reduce computation iteratively. In addition to the map and reduce functions, it provides an update function to update a global state before the next iteration. SCOPE [7] is a declarative scripting language targeting for massive data analyses. It provides parallel data processing capabilities through an SQL-like grammar. It also supports the map-reduce programming model, in which the user can implement her own ML algorithms. These DSLs target data parallel processing, which is not the main goal of PILOTS. PILOTS is a DSL for data stream processing with error detection and correction, which enables modular data-driven failure model learning. As opposed to some of these DSLs for ML, PILOTS is purely declarative, which trades expressiveness for ease of use.

PyMC [13] is a Python library for Bayesian statistical models and fitting algorithms, including Markov chain Monte Carlo. It also provides a lot of pre-defined probability distributions as well as an extensible mech-

anism that allows users to write their own stochastic processes. Its flexibility and capability for Bayesian analysis exceeds the current Bayesian model support in PILOTS. PyMC and PILOTS can complement each other, by integrating PyMC as one of the ML engines in PILOTS through the adaptation layer. There is an increasing number of ML algorithms and libraries (*e.g.*, we use *scikit-learn* in Python) and our modular PILOTS architecture is designed to easily integrate new ML approaches to sensor failure modeling from data as they become available.

In recent years, several works have been proposed as data processing infrastructures for Dynamic Data Driven Applications Systems (DDDAS) [9]. Ditzler et al. presented the design of the High Performance Machine Learning (HPML) framework as a unified framework for Big Data processing [12] with a focus on batch machine learning. The HPML framework hides the complexity of underlying computing resources (*e.g.*, cluster of GPUs, cloud computing systems) from the user and intends to provide high-level abstraction for machine learning applications. Shekhar and Gokhale proposed the concept of Dynamic Data Driven Cloud and Edge Systems (D^3CES) as a model for adaptive cloud and edge computing resource management [34]. The D^3CES targets cyber physical systems (CPS) and Internet of Things (IoT) applications distributed across edge and cloud computing resources. It continuously monitors the application performance and elastically scales resources to meet given task deadlines. Kamburugamuve et al. studied the optimization of communication topology between worker processes for Apache Storm [36], which is used as a base application platform for Dynamic Data Driven Applications (DDDA) [24].

8 Discussion and future work

Semi-supervised learning for dynamic Bayes classifier When using the dynamic Bayes classifier to detect weight error, we noticed that the system could not only detect "normal" and "underweight" modes, but also classifies "5% overweight" and "15% overweight" as two different modes as shown in Figure 26 and Table 9. This information is useful if different strategies need to be taken for different extent of weight errors, otherwise it would be unnecessarily misleading to classify them into different modes. Thus, the dynamic Bayes classifier should be adjusted to the requirements of various use cases. This would result in a semi-supervised online learning approach, in which human experts give ground truth to distinguish anomalies from new operating modes.

Future work for our dynamic Bayes classifier includes to involve human feedback in the online learning phase, especially when a new major mode is detected, to get more accurate failure mode classification and decision making. Our dynamic Bayes classifier produces results that depend on the order it sees new data. We need to devise new techniques to make data ingestion associative and commutative, so that classification (especially new modes) is data-order independent. Support for multiple features and techniques for determining thresholds for converting minor modes into major modes will also be explored.

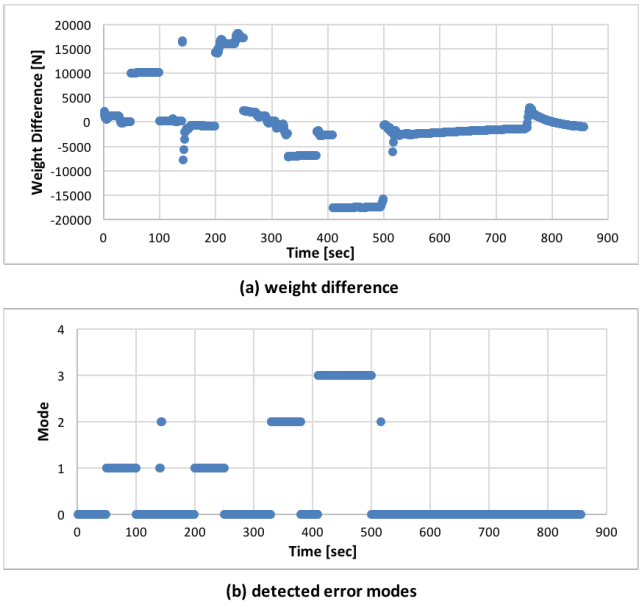

Fig. 26 Another weight error mode detection using dynamic Bayes classifier.

Table 9 Major and minor modes of dynamic Bayes classifier after running testing data.

Mode status	Mode description	Mode	μ	σ	n
Major	Normal	0	-33	2462	6544
	Underweight	1	15898	3312	1708
Minor	5% Overweight	2	-6922	239	54
	15% Overweight	3	-17450	277	91

Real-time execution of PILOTS programs For the real-time mode, the PILOTS runtime system is currently designed to execute programs in a best-effort basis. For example, even if the output rate is every second, the application could take more than one second to compute outputs, depending on the complexity of computation.

Table 10 shows average latency per one output cycle for three types of error detection methods measured on a laptop computer (CPU: Intel Core i5-6200U 2.30GHz, Memory: 8GB). They are in the order of milliseconds, and therefore, there have been no issues to keep the output rate of one second that we use throughout this paper.

Table 10 Comparison of latency per one output cycle.

Error Signatures	Linear Regression	Dynamic Bayes Classifier
2.41 ms	7.78 ms	8.55 ms

The current compiler implementation generates a single thread per output stream. If we were to analyze hundreds of input streams producing hundreds of output streams, we would need to allocate enough processes to these hundreds of threads to maintain latency and throughput requirements. Moreover, the machine learning component could be a bottleneck when processing such a large number of streams simultaneously. Thus, we would have to run it on a scalable data processing system and scale up its performance to keep up with the increasing number of input streams. To maintain the required performance in streaming systems, it is important to obtain a performance prediction model and proactively allocate enough computing resources [18].

Also, internal communication between the mode estimation subsystem and the core PILOTS system can hurt the realtime-ness of the application execution. Suppose the core PILOTS system runs on an airplane while the mode estimation subsystem runs in the cloud for scalability, latency could be up to hundreds of milliseconds. To guarantee stronger realtime processing, we could implement a progressive mode estimation mechanism with increasing estimation accuracy and timeout the mode estimation before the deadline. Distributed, scalable, and fault-tolerant data streaming systems include MillWheel [1], Storm [36], and Spark Streaming [39]. SAMOA [29] provides a streaming ML programming framework that supports multiple stream processing engines. Since these systems are expected to run over many computer nodes, they are designed to continue producing correct results with reasonably degraded performance even in the case of node failures.

Other research directions Machine learning techniques could also be used to learn parameters in error signatures from data. Another possible direction is to combine logic programming and probabilistic programming, as in ProbLog [11], to help analyze spatio-temporal data streams. Finally, uncertainty quantification [2] is another important future direction to associate confidence to data and error estimations in support of decision making.

9 Conclusion

In this paper, we presented a highly-declarative programming framework that facilitates the development of self-healing avionics applications, which can detect and recover from data errors. Our programming framework enables specifying expert-created failure models using error signatures, as well as obtaining failure models from data by machine learning. Also, we investigated both linear regression and dynamic Bayesian learning approaches to data-driven fault detection in avionics. We applied a linear regression approach to learn the relationship between coefficient of lift and angle of attack during cruise flight. With the training results, and Equation (9) to calculate airplane weight during cruise phase, the PILOTS program successfully detects and corrects underweight and overweight conditions in simulated flight data by using error signatures. Using our dynamic Bayes classifier, when the system is trained with normal and underweight data, the PILOTS program is able to detect a new mode when an overweight situation occurs in the online learning phase. Error signatures require more specific patterns for detecting faults, while our dynamic Bayes classifier uses data to learn different operating modes. Our dynamic Bayes classifier detects statistically significant new modes during the online learning phase, while our error signatures approach can only detect pre-defined modes.

Acknowledgements This research is partially supported by the DDDAS program of the Air Force Office of Scientific Research, Grant No. FA9550-15-1-0214, NSF Grant No. 1462342, and a Yamada Corporation Fellowship. We would like to thank anonymous reviewers for their valuable feedback. We would also like to acknowledge co-authors of previous PILOTS papers: Erik Blasch, Richard Klockowski, Alessandro Galli, Frederick Lee, and Colin Rice.

References

1. Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
2. Douglas Allaire, David Kordonowy, Marc Lecerf, Laura Mainini, and Karen Willcox. Multifidelity DDDAS methods with application to a self-aware aerospace vehicle. In *DDDAS 2014 Workshop at ICCS'14*, pages 1182–1192, June 2014.

3. John David Anderson Jr. *Fundamentals of aerodynamics*. Tata McGraw-Hill Education, 2010.
4. Gautam Biswas, Hamed Khorasgani, Gerald Stanje, Abhishek Dubey, Somnath Deb, and Sudipto Ghoshal. An application of data driven anomaly identification to spacecraft telemetry data. In *Prognostics and Health Management Conference*, 2016.
5. Erik P Blasch, Dale A Lambert, Pierre Valin, Mieczyslaw M Kokar, James Llinas, Subrata Das, Chee Chong, and Elisa Shahbazian. High level information fusion (hlif): survey of models, issues, and grand challenges. *IEEE Aerospace and Electronic Systems Magazine*, 27(9):4–20, 2012.
6. Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile. Final Report: On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris.
7. Ronnie Chaiken, Bob Jenkins, PerÅke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.
8. Sida Chen, Shigeru Imai, Wennan Zhu, and Carlos A. Varela. Towards learning spatio-temporal data stream relationships for failure detection in avionics. In *Dynamic Data-Driven Application Systems (DDDAS 2016)*, Hartford, CT, Aug 2016. To appear.
9. Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *Computational Science-ICCS 2004*, pages 662–669. Springer, 2004.
10. S Das, BL Matthews, Ashok N Srivastava, and Nikunj C. Oza. Multiple kernel learning for heterogeneous anomaly detection: algorithm and aviation safety case study. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 47–55, 2010.
11. Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, volume 7, pages 2462–2467, 2007.
12. Gregory Ditzler, Salim Hariri, and Ali Akoglu. High Performance Machine Learning (HPML) Framework to Support DDDAS Decision Support Systems: Design Overview. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 360–362. IEEE, 2017.
13. Christopher J. Fonnesbeck. PyMC 2.3.6 documentation. https://pymc-devs.github.io/pymc/. Accessed 06-12-2017.
14. Inseok Hwang, Sungwan Kim, Youdan Kim, and Chze Eng Seah. A survey of fault detection, isolation, and reconfiguration methods. *IEEE Transactions on Control Systems Technology*, 18(3):636–653, 2010.
15. Shigeru Imai, Erik Blasch, Alessandro Galli, Wennan Zhu, Frederick Lee, and Carlos A. Varela. Airplane flight safety using error-tolerant data stream processing. *IEEE Aerospace and Electronics Systems Magazine*, 32(4), 2017.
16. Shigeru Imai, Alessandro Galli, and Carlos 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.
17. Shigeru Imai, Richard Klockowski, and Carlos A. Varela. Self-healing spatio-temporal data streams using error signatures. In *2nd International Conference on Big Data Science and Engineering (BDSE 2013)*, Sydney, Australia, December 2013.
18. Shigeru Imai, Stacy Patterson, and Carlos A. Varela. Maximum sustainable throughput prediction for data stream processing over public clouds. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*, Madrid, Spain, May 2017.
19. Shigeru Imai and Carlos A. Varela. A programming model for spatio-temporal data streaming applications. In *Dynamic Data-Driven Application Systems (DDDAS 2012)*, pages 1139–1148, Omaha, Nebraska, June 2012.
20. Shigeru Imai and Carlos 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.
21. JavaCC Development Group. JavaCC - The Java Parser Generator. http://javacc.org/. Accessed 06-12-2017.
22. Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.
23. George H John and Pat Langley. Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.
24. Supun Kamburugamuve, Saliya Ekanayake, Milinda Pathirage, and Geoffrey Fox. Towards high performance processing of streaming data in large data centers. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1637–1644. IEEE, 2016.
25. Richard S. Klockowski, Shigeru Imai, Colin Rice, and Carlos 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*, pages 2036–2045, May 2013.
26. Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
27. Daniel LC Mack, Gautam Biswas, Xenofon D Koutsoukos, and Dinkar Mylaraswamy. Learning bayesian network structures to augment aircraft diagnostic reference models. *IEEE Transactions on Automation Science and Engineering*, 14(1):358–369, 2017.
28. Daniel LC Mack, Gautam Biswas, Xenofon D Koutsoukos, Dinkar Mylaraswamy, and George Hadden. Deriving bayesian classifiers from flight data to enhance aircraft diagnosis models. In *Annual Conference of the Prognostics and Health Management Society*, 2011.
29. Gianmarco De Francisci Morales and Albert Bifet. SAMOA: Scalable Advanced Massive Online Analysis. *Journal of Machine Learning Research*, 16:149–153, Jan 2015.
30. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

31. Agenzia Nazionale per la Sicurezza del Volo. Final Report: Accident involving ATR 72 aircraft registration marks TS-LBB ditching off the coast of Capo Gallo (Palermo - Sicily), August 6th, 2005.
32. 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.
33. Irina Rish. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.
34. Shashank Shekhar and Aniruddha Gokhale. Dynamic resource management across cloud-edge resources for performance-sensitive applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 707–710. IEEE Press, 2017.
35. Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.
36. The Apache Software Foundation. Apache Storm. http://storm.apache.org/, January 2015. Accessed 06-14-2017.
37. Jianwu Wang, Yan Tang, Mai Nguyen, and Ilkay Altintas. A scalable data science workflow approach for big data bayesian network learning. In *Big Data Computing (BDC), 2014 IEEE/ACM International Symposium on*, pages 16–25. IEEE, 2014.
38. Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Machine learning in ScalOps, a higher order cloud computing language. In *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, volume 9, pages 389–396, 2011.
39. Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 2012.

Appendix A Core PILOTS runtime library

The core PILOTS runtime library is in charge of starting a data receiving server, storing received data, providing data selection/interpolation service to the application, and sending processed data to output/error hosts.

Primary classes included in the PILOTS runtime library shown in Figure 27 are explained as follows.

- PilotsRuntime class is extended by the application and provides all basic functions to run a PILOTS application other than application-specific processing. It starts DataReceiver to start receiving data, requests stored data from DataStore, and sends calculated outputs and errors to other hosts.
- DataReceiver class receives data from data input clients from a port specified in the command-line arguments. Upon accepting data, it launches a new worker thread to receive data and the created thread requests to add these data to DataStore.
- DataStore class accepts data from DataReceiver as a string, and then it asks SpatioTempoData to parse the string and stores the parsed data. It also implements getData() method supporting closest, euclidean, interpolate for data selection. When comparing locations and time for data selection/interpolation, it asks for the current time and location from CurrentLocationTimeService. Stored data are accessed from multiple threads (*i.e.*, threads for adding data from DataReceiver vs. threads getting data from PilotsRuntime), so the data have to be protected from simultaneous data access.
- CurrentLocationTimeService class is an interface class for providing the current time and location. Users have to implement this class for the system to work (*e.g.*, SimpleTimeService and SimulationService). The implemented class can either return the actual current time for the real-time mode or past time for the simulation mode.

Appendix B PILOTS with machine learning component grammar design

The grammar of the PILOTS programming language with machine learning component is designed as shown in Figure 28. Also, the grammar of the PILOTS trainer file is shown in Figure 29[3].

[3]Note: the current version (v0.4) of PILOTS does not fully support the grammar shown here as of October 2017.

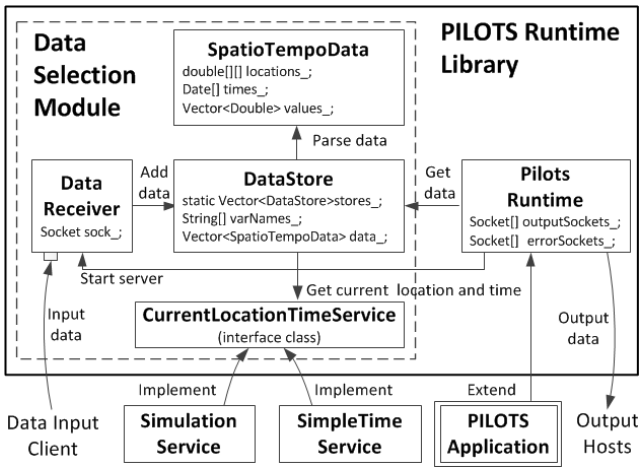

Fig. 27 Class diagram of PILOTS runtime library.

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	::=	*Vars*: *Exps* at every *Time* [when (*Var* | *Exp*) [*Integer* times]];
Errors	::=	[(*Error*;)* *Error*];
Error	::=	*Vars*: *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, sin, 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", ... }

Fig. 28 PILOTS with machine learning component grammar design.

Trainer	::=	trainer *Var*; constants *Constants* data *Data* [*Schema*] model [*Preprocess*] *Features* *Labels* *Algorithm* end
Constants	::=	[(*Constant*;)* *Constant*];
Constant	::=	*Var* = *Exp*
Data	::=	[(*DataItem*;)* *DataItem*];
DataItem	::=	*Vars* using (*File* | *ModelUser*)
File	::=	file'('*String*')'
ModelUser	::=	model'('*Vars*')'
Schema	::=	schema unit'('*Map*')';
Preprocess	::=	preprocess [changeunit using unit'('*Map*')'];
Features	::=	features: *Exps*;
Labels	::=	labels: *Exps*;
Algorithm	::=	*Var* ['('*Map*')'];
Map	::=	*MapItem* | *MapItem, Map*
MapItem	::=	*Var* ':' (*Number* | *Var* | *Exp*)

Fig. 29 PILOTS trainer file grammar design (Note: for the non-terminals not defined here, refer to Figure 28).

```
public class Twice extends PilotsRuntime {
  private Timer timer_;
  private SlidingWindow win_e_;
  private Vector<ErrorSignature> errorSigs_;
  private ErrorAnalyzer errorAnalyzer_;

  public Twice(String args[]) {
    parseArgs(args);
    timer_ = new Timer();
    win_e_ = new SlidingWindow(getOmega());
    errorSigs_ = new Vector<ErrorSignature>();
    errorAnalyzer_ = new ErrorAnalyzer(
      errorSigs_, getTau());
  }

  public void startOutput_e() {
    openSocket(OutputType.Output, 0, "e");
    timer_.scheduleAtFixedRate(new TimerTask() {
      public void run() {
        Value a = new Value();
        Value b = new Value();
        a.setValue(getData( "a",
          new Method(Method.Closest, "t")));
        b.setValue(getData( "b",
          new Method( Method.Closest, "t")));
        double e = (b.getValue() - 2*a.getValue());
        sendData(OutputType.Output, 0, e);}
    }, 0, 1000);
  }

  public static void main(String[] args) {
    Twice app = new Twice(args);
    app.startServer();
    app.startOutput_e();
  }
}
```

Fig. 30 Generated Java code for the Twice PILOTS program shown in Figure 1.

Appendix C Example generated Java code

Generated Java code for the Twice PILOTS program (Figure 1) is shown in Figure 30. Also, the difference in generated Java code between Twice (Figure 1) and Twice_Signatures (Figure 4) PILOTS programs is shown in Figure 31. Note that the difference is generated by the following Linux command:

```
$ diff Twice.java Twice_Signatures.java
```

```
< public class Twice extends PilotsRuntime {
---
> public class Twice_Signatures extends PilotsRuntime {
14c14
<   public Twice(String args[]) {
---
>   public Twice_Signatures(String args[]) {
26a27,30
>     errorSigs_.add(new ErrorSignature(ErrorSignature.CONST, 0.0, "Normal mode"));
>     errorSigs_.add(new ErrorSignature(ErrorSignature.LINEAR, 2.0, "A failure"));
>     errorSigs_.add(new ErrorSignature(ErrorSignature.LINEAR, -2.0, "B failure"));
>
29a34,56
>   public void getCorrectedData(SlidingWindow win,
>                                Value a, Value a_corrected,
>                                Value b, Value b_corrected,
>                                Mode mode, int frequency) {
>     a.setValue(getData("a", new Method(Method.Closest, "t")));
>     b.setValue(getData("b", new Method(Method.Closest, "t")));
>     double e = (b.getValue()-2*a.getValue());
>
>     win.push(e);
>     mode.setMode(errorAnalyzer_.analyze(win, frequency));
>
>     a_corrected.setValue(a.getValue());
>     b_corrected.setValue(b.getValue());
>     switch (mode.getMode()) {
>     case 1:
>       a_corrected.setValue(b.getValue()/2);
>       break;
>     case 2:
>       b_corrected.setValue(a.getValue()*2);
>       break;
>     }
>   }
>
40a68
>           Value a_corrected = new Value();
41a70,71
>           Value b_corrected = new Value();
>           Mode mode = new Mode();
43,45c73,74
<           a.setValue(getData("a", new Method(Method.Closest, "t")));
<           b.setValue(getData("b", new Method(Method.Closest, "t")));
<           double o = (b.getValue()-2*a.getValue());
---
>           getCorrectedData(win_o_, a, a_corrected, b, b_corrected, mode, frequency);
>           double o = (b_corrected.getValue()-2*a_corrected.getValue());
57c86
<     Twice app = new Twice(args);
---
>     Twice_Signatures app = new Twice_Signatures(args);
```

Fig. 31 Difference in generated Java code between Twice (Figure 1) and Twice_Signatures (Figure 4) PILOTS programs.