Towards Sequential Multivariate Fault Prediction for Vehicular Predictive Maintenance

***Abstract*—Predictive maintenance, which has traditionally used anomaly detection methods on sensory data, is now being replaced by event-based techniques. These methods utilise events with multiple temporal features, produced by diagnostic modules. This raises the need for predicting the next fault event in industrial machines, specially vehicles, that use Diagnostic Trouble Codes (DTCs). We propose a predictive maintenance approach, named Sequential Multivariate Fault Prediction (SMFP), for predicting the next multivariate DTC fault in an event sequence, using Long Short-Term Memory Networks (LSTMs) and jointly *learned* event embeddings. By performing an in-depth comparison of different architectural choices and contextual preprocessing techniques, we provide an initial baseline for SMFP that achieves top-3 accuracy of 63% on predicting multivariate fault with 3 collective output layers, using vehicle maintenance data as a case study.**

***Index Terms*—Predictive maintenance, LSTM, DTCs, Embeddings**

I. Introduction

Industrial systems, ranging from small machines to vehicles, rely on maintenance for their durability. In recent years, companies have started to invest in techniques that can prevent faults in advance. One such technique is *Preventive Maintenance* [1], which is a proactive monthly or biannual routine inspection of the system that tries to reduce major breakdowns and failures by solving minor problems ahead of time but requires increased labor and cost. In contrast, *Predictive Maintenance* [2] refers to the analysis of historical data, correlations, and complex patterns to predict the need for maintenance or to forecast failures of the system instead of periodic maintenance, hence resulting in cost reduction and a decrease in the downtime of equipment.

Traditional predictive maintenance systems apply different anomaly detection techniques and time-series forecasting algorithms to sensory data from vehicles, for example, using correlation to perform predictive maintenance of compact electric generators [3]. Time series models like the Autoregressive and Moving Average Model (ARMA) have also been used for anomaly detection [4], [5]. Others include clustering techniques for outlier detection [6], [7] and recurrent neural networks to detect anomalies in aircraft data [8].

Instead of sending sensory values to cloud platforms for analysis, modern vehicles have diagnostic modules installed in them, which detect the faults themselves. These modules provide sequences of multivariate fault events called Diagnostic Trouble Codes (DTCs), which convey information about the time, mileage, granularity level of fault, etc. Some features, especially those related to faults, are non-numeric (categorical) and have a different number of unique categories or classes (*cardinality*), for example, 70, 486, and 84. High cardinality of features and lack of numeric representation restrict the utilization of machine learning algorithms, which often rely on the numeric representation of features with low cardinality. Due to these problems and the complexity of multivariate sequential dependencies between the events in a sequence, researches up till now have either relied on DTC relationship analysis, focused on models that do not cater for sequential dependencies (i.e, considered events as independent), or have limited the predictions to a few DTC events only (i.e, few vehicle components) [9], [10], [11]. To overcome the complexity of performing predictive maintenance on multivariate events with sequential dependencies, a significant paradigm shift is necessary.

To this end, we propose a Sequential Multivariate Fault Prediction (SMFP)[1] approach where we predict the next DTC event at timestep $T+1$ given a sequence of multivariate DTC events up to the current timestep T. This enables us to perform predictive maintenance by taking a proactive action of checking or replacing the component predicted by SMFP. In addition, we use multiple features of the predicted event to trace the location and granularity of the fault. The contributions of SMFP are:

1) An embedding mechanism, learned jointly with the prediction task, to map multivariate events into a continuous representation space, where similar events are close to each other.
2) A multi-output model based on a Long Short-Term Memory (LSTM) network, which is learned on top of the embeddings to handle the sequential dependencies between the multivariate events and to simultaneously optimize the joint event representation space in embeddings.
3) A baseline for this new methodology by performing an extensive set of experiments with results and observations relating to different architectures, embeddings, hyperparameters, and contextual preprocessing approaches for SMFP.

[Acknowledgements omitted for submission].

[1][14] also worked in events prediction in a recommender system, but, unlike in our proposal, events are univariate and sequentially independent, and follow a different algorithmic approach (Empirical Risk Minimization ranking model).

Our experiments achieve a baseline of 63% top-3 test accuracy for the next event prediction, which is a combined accuracy of all three output layers, each corresponding to a different event feature with varying cardinalities. We interpret 63% top-3 accuracy as identifying the actual fault in the top-3 predictions, 63% of the time.

In the next section, we define the methodology for learning joint event representations, SMFP with LSTMs, and several architectural choices. In section III, we present results from our experiments. We shall finish with a conclusion in section IV.

II. Methodology

In this section, we will first share the details about the dataset, the data preprocessing and the problem. We will then discuss how LSTMs handle the sequential dependencies of such multivariate events and how we use them in our work for performing SMFP. Next, we describe how neural embeddings jointly represent multivariate events and how they affect the overall architecture of the model, including the choice between multiple output layers (one for each feature) and a single output layer (with concatenated raw features). Lastly, we compare different contextual preprocessing approaches and their impact on sequential dependency modelling.

A. Dataset and preprocessing

Diagnostic data, which is located in the memory of electronic modules and control units, is either collected in a diagnostic session or streamed to the cloud periodically. DTC events data for this work is provided by *[omitted for submission]*, which is a *[omitted for submission]* for vehicles. Before preprocessing (grouping) data, a single event row in a dataset corresponds to a fault that occurred in a module of a car. Besides event related information, each row has some attributes related to the car and the session where this event is recorded, for example mileage and time. We initially preprocess the dataset such that all observations (multivariate events) corresponding to one car are grouped into one sequence, ordered by occurrence and mileage. This preprocessing produced 250,000 such sequences. We further preprocessed the data by separating individual features f^i from all the events of a sequence, to form feature vectors $\vec{f}^i$. These two different preprocessing steps are reflected in figure 1 and are used differently for embeddings later.

In figure 1-b, we can see a single sequence containing multiple multivariate events from a car. The first feature, which is at the highest granular level, can be seen as the main control source of error like Transmission Module (TM), Object-Detection Module (OD), Break-System Module (BS), etc. The second feature, which is less granular, provides location information such as the part of vehicle (pedal, chassis area or power area). The third and lowest granular feature can be thought of as fault-type, for example missing the high or low value of some chip. This single arbitrary sequence with the last 5 multivariate events looks like [..., (TM,Power,01), (OD,Camera,12), (BS,Pedal,29), (TM,Power,01), (TM,Battery,101)]. As illustrated in the figure 1-b, the goal is to predict the next DTC event with all the three features, given previous multivariate events. Next, we define this process formally.

B. Sequential Multivariate Fault Prediction

Let $s = (e_T, e_{T-1}, .., e_1)$ be a single such sequence containing multivariate fault events from timestep T to timestep 1 (i.e., in descending order of occurrence), where $e_t = (f_t^1, f_t^2, f_t^3)$ is an event at timestep t with three variables (features) and f_t^i represents the *i-th* feature of this event at timestep t. If we denote a sequence $s = (e_T, e_{T-1}, .., e_1)$ compactly as $s_{T:1}$ and θ be the parameters of the model that we aim to learn, then the candidate event $\widetilde{e}$ from all possible events E that maximises the following probability, is selected as the next predicted event at timestep $T + 1$.

$$\underset{\widetilde{e}}{\operatorname{argmax}} P(s_{T+1} = \widetilde{e} | s_{T:1}, \theta) \tag{1}$$

C. Sequential dependencies and recurrent neural networks

Since sequences consist of multiple DTC events up to timestep T, it is necessary to capture sequential order and dependencies between events in order to predict the next DTC event. Unlike regular feed-forward networks, Recurrent Neural Networks (RNNs) use a recurrent loop that acts as a memory and helps treating a history of events as a latent hidden state (h). At timestep T, instead of modelling based on a complete sequence $s_{T:1}$, the goal is to efficiently store all the history in the current hidden state h_T and model following

$$P(s_{T+1} | s_{T:1} = (e_T, e_{T-1} .., e_1)) = P(s_{T+1} | h_T) \tag{2}$$

The previous hidden state h_{T-1} and the current input timestep s_T contribute to calculating h_T with different learned weights: the weights between the inputs and the hidden layer W_{sh}, the bias weight for the hidden layer b_h and the weights between hidden states W_{hh}. The hidden state h_T is then computed with an activation function a as

$$h_T = a(s_T, h_{T-1}) = a(s_T \cdot W_{sh} + h_{T-1} \cdot W_{hh} + b_h) \tag{3}$$

Similarly, we have output weights W_{hq} and a bias weight for the output layer b_q. We use W_{hq} and h_T to calculate the output Y_T as

$$Y_T = a(h_T \cdot W_{hq} + b_q) \tag{4}$$

Learning long term dependencies with RNNs involves multiplying gradients, which results in either very large exploding gradient values or very small vanishing ones. LSTM networks, introduced in [12], are a type of RNNs that addresses the problem of remembering long-term dependencies in sequences by incorporating additional components, namely a forget gate, an input gate, a cell state, and an output gate. Three gates with their respective weight matrices for input layers

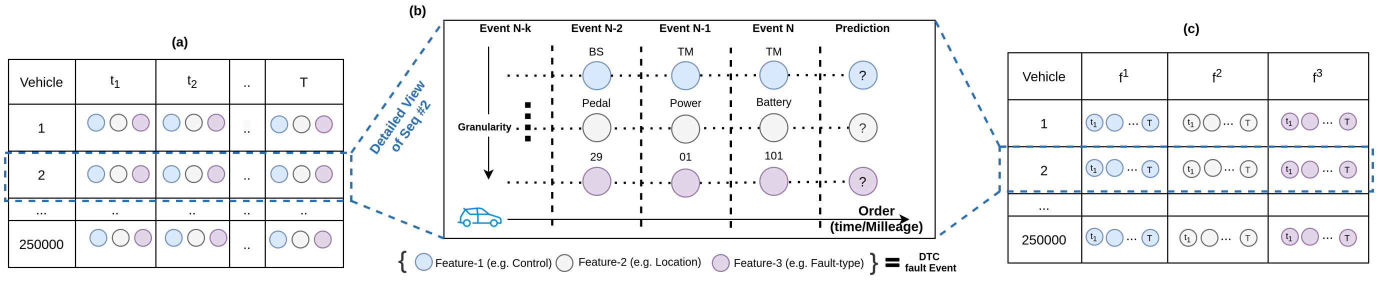

Fig. 1. (a) An initial preprocessing (grouping) of data resulting in single sequence per vehicle ordered by occurrence and mileage. (b) A detailed view of a single DTC event sequence having an event containing all the three features at each timestep (e.g., the last timestep shows a Transmission Module (TM) related fault in the battery component, with fault type 101). (c) Further preprocessing done on (a) to separate individual features for applying embeddings.

(W_{sf},W_{si},W_{so}), bias weights (b_f,b_i,b_o) and hidden layers weights (W_{hf},W_{hi}, and W_{ho}) are defined as

$$f_T = a(s_T \cdot W_{sf} + h_{T-1} \cdot W_{hf} + b_f) \tag{5}$$

$$I_T = a(s_T \cdot W_{si} + h_{T-1} \cdot W_{hi} + b_i) \tag{6}$$

$$O_T = a(s_T \cdot W_{so} + h_{T-1} \cdot W_{ho} + b_o) \tag{7}$$

The forget gate (f_T) provides a reset mechanism for the content of a cell state C_T. The input gate (I_T) helps to decide how much information should be read into the candidate cell state $\widetilde{C_T}$, which has a weight W_{sc} with the input, a bias weight b_c and a weight W_{hc} with the hidden layer. The output gate (O_T) channels how much information should the network process, based on C_T. LSTMs prevent unnecessary updates of content, by avoiding the linear transformation of the cell state C_T and replacing it by point-wise multiplication of the candidate memory and the gates.

$$\widetilde{C_T} = tanh(s_T \cdot W_{sc} + h_{T-1} \cdot W_{hc} + b_c) \tag{8}$$

$$C_T = F_T \odot \widetilde{C_T} + I_T \odot \widetilde{C_T} \tag{9}$$

D. Learning to represent events

Representing textual events numerically is required to apply deep learning algorithms and calculate similarity metrics. Some representations are human-interpretable as they use simple metrics like count, presence and absence, etc., of the events. For example, in a Bag-Of-Words (BOW) model [13], the count of occurrences of each word is used as a feature for training. In our case, we can denote a sequence of a feature f^2 with the count of occurrences of its categories (Camera, Pedal, Chassis, etc.). Similarly, One-Hot Encoding (OHE) represents each event in a sequence as an N-dimensional vector, where N is the cardinality of the event. Each vector uses 1 only at the index of that particular word and 0 elsewhere. Unlike BOW, OHE preserves the order of token but suffers from the curse of dimensionality (for instance, f^2 of every event at each timestep will be replaced by a 486 dimensional vector, where 486 is the cardinality of f^2). Other types of representation are derived representations, produced by some algorithm or mathematical formula and hence they might not be directly interpretable (for example, Frequency-Inverse Document Frequency (TF-IDF), in [14]).

Neural embedding is a type of derived representation that maps data to continuous low-dimensional representations learned using neural networks. They provide meaningful representations by preserving the similarity of events and mapping each event to a continuous space, such that the events that appear together are close in the new representation. There are many efficient embedding algorithms that produce representations with neural networks, for example by predicting the next words given their context (words appearing together) like Word2Vec [15].

1) Representation with neural embeddings: Due to the high cardinality of features, for each feature f^i, a neural embedding matrix $f^i{}_{EMB}$ is learned jointly with a prediction task such that $size(f^i{}_{EMB}) < size(f^i)$. A linear projection $(f^i{}_{EMB} \cdot f^i_{OHE})$ is applied to obtain a new reduced representation of an OHE *i-th* feature (f^i_{OHE}). In our study, there were two architectural choices for learning joint embeddings (F_{EMB}) for all three features and both have an impact on the number of output layers and on the performance of the resulting architecture. We briefly describe them next.

a) Feature concatenation and single entity embedding: In the first approach, we concatenated 3 raw features per time step ($f_{concat} = f^1 \parallel f^2 \parallel f^3$) and used it to calculate an embedding for the event. We also used the OHE of raw concatenation as outputs, which resulted in one output layer, as shown in figure 2.

b) Multi-input multi-output model with separate embeddings: The drawback of concatenating raw features is that if a feature is missing in the concatenation (f_{concat}), the whole concatenation becomes invalid. This problem reduces the dataset size and the quality of the model, and is expected to get worse as new categories arise for each feature. To handle this issue, we changed the approach of preprocessing the data as well as the architecture involving embedding layers. We applied separate embedding layers to individual feature vectors $\vec{f}^i$. After obtaining separate embeddings for each $\vec{f}^i$, all three embeddings were horizontally stacked to form (F_{EMB}), before passing to LSTMs. We then introduced separate output layers for each feature. The architecture for this approach is depicted in figure 3.

Since weights for these embeddings are learned jointly with the downstream prediction task using LSTMs, they are opti-

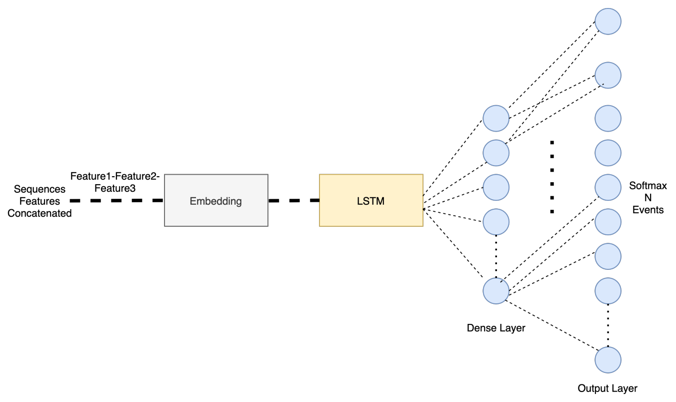

Fig. 2. Architecture diagram with a single embedding for concatenated features and a single output.

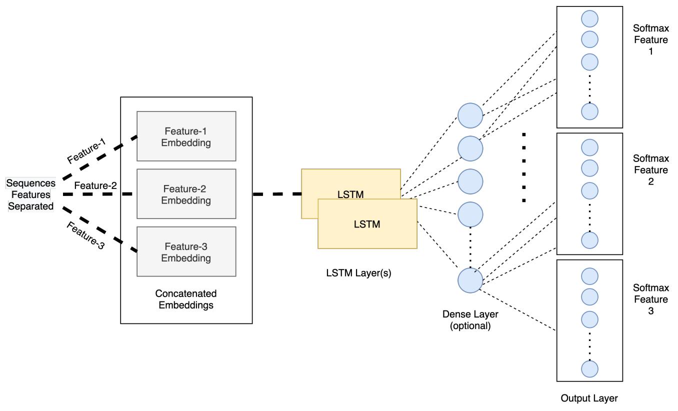

Fig. 3. Architecture diagram for the approach of concatenating separate embeddings for every feature and multiple-outputs.

mized such that they retain sequential and semantic similarities in the learned continuous representation space.

E. On context length and its influence

Theoretically, LSTMs can handle sequences of variable lengths with long-term dependencies without encountering vanishing or exploding gradients. Despite this, due to the large variations in the length of sequences and computational constraints, the number of events to consider (i.e., context) to predict the next event is an important decision for the architectural choice of an LSTM network. Moreover, within a single sequence corresponding to one vehicle, there can be different unrelated event sub-sequences. For example, if we aim to predict the next DTC event in a sequence containing 15 total events, it is possible that only the last 7 events (a sub-sequence from t_8 to T) are related and will result in failure of power. We compare taking only the last N events to different sequence preprocessing techniques, in order to ascertain if an LSTM can learn these relations without additional context.

Using the *Fixed Window Split* technique [16], we divided long sequences into separate sequences after a fixed window size of N timesteps. For instance, if one sequence has a length of 80, it will be divided into 8 sub-sequences with a split size of N=10. The *Sliding window* approach is used in many areas of information theory such as in time-series prediction [17] and NLP. In our study, for each sequence, we took a fixed window of N timesteps of a sequence at a time and then slid this window to the right with stride size 1.

We also used *Overlapping windows*, where for each window of size N, instead of sliding it with stride size 1, we included the last M timesteps of the previous window into the window, giving an effect of smooth overlap and transition. Finally, we used *Last-N Timesteps*, where we considered only the last N timesteps of each sequence. In most of the approaches mentioned above, we had some sequences which were shorter than the window size and had to be padded by 0.

III. Experiments and Results

Our model predicts a multivariate event using three dense outputs layers. We use OHE to represent actual output features (ground truth labels) of an event and a softmax activation function in output layers. These layers provide K_{f^i} probabilities, where K_{f^i} is the cardinality of a particular feature f^i. All output layers have cross-entropy loss as the measure of errors on predicted outputs. The overall loss is the sum of the three individual losses for each feature f^i. If $\hat{y}$ denotes the predicted class, y corresponds to an actual class, and K_{f^i} is the size of the output layer for a particular feature, the overall categorical loss of all F features is calculated as

$$L(\hat{y}, y) = \sum_{f^i}^{F} (-\sum_{k}^{K_{f^i}} y^{(k)} \log(\hat{y^{(k)}})) \tag{10}$$

Binary and multi-class classification problems use *accuracy* to evaluate the performance of the model. This metric measures the fraction of correct predictions out of total predictions. Due to the high cardinality of features (70, 486, and 84, respectively), we used a top-3 accuracy metric. This metric measures if the actual values of all three features fall in their respective top-3 predictions, which are probabilities sorted in descending order. For example, imagine that the actual fault that occurred was Camera (f^2 in the event) and that the model assigns the highest probabilities to Battery, Chassis, and Camera in descending order (i.e., Camera is in the top-3 probabilities for this feature f^2): if the same stands true for the other two features, then we count it as a true prediction.

A. Experimental setup and hyperparameter tuning

We have trained our model using 250,000 sequences, each of which corresponds to a unique vehicle. We performed training and test split of the dataset before applying any contextual preprocessing technique. We kept aside 12,500 sequences for testing, 4,750 sequences for validation, and used the remaining 232,750 sequences for training. Adam optimizer, with a learning rate of 0.001, was used for optimizing the network. Data preprocessing, including ordering the sequences with mileage and time, is done using Apache Spark [18] while the rest of the implementation uses Keras [19].

Hyperparameter choices, which we explain in detail in individual result sections, include (i) using one wide LSTM layer instead of two LSTM layers, (ii) using 60 to 240 units in each LSTM layer, (iii) whether to use a dense layer on top of the LSTM layer (iv) dropout values (both recurrent dropout and dropout after LSTM layers), and (v) the output size of the embedding (5 to 20 percent of the original feature cardinality) for each feature. All hyperparameters are selected using Keras Tuner, which provides different hyperparameter tuning algorithms like Random Search and Bayesian Optimization [20]. Common hyperparameter choices are listed in Table I.

TABLE I
COMMON HYPERPARAMETER CHOICES

Choice	Values
Optimizer	Adam (beta-1 = 0.9 & beta-2 = 0.999)
Learning rate	0.001
Dropout & Recurrent Dropout	0.1 to 0.3

B. Comparing feature concatenation with separate embeddings and selecting embedding dimension

Before running all experiments with the different hyperparameters and architectural choices, we evaluated the performance of two basic feature representation mechanisms; concatenating all three features before applying embedding and concatenating separately learned embeddings of each feature. The first row in Table II shows the performance of SMFP with a single concatenated input feature and a single output, while the second row reflects the result of a multi-input and multi-output model. We achieve 63% top-3 test accuracy for the separate embedding concatenation approach, while the alternative approach resulted in 51% top-3 test accuracy. Besides the difference in the results, the embedding concatenation approach is also flexible for feature preparation, as explained in II-D, so we have used separate feature embeddings in all other experiments.

The impact and choice of embedding dimensionality, which is currently a much-focused research area [21], depends on the cardinality of the feature being embedded. Although embeddings avert the curse of dimensionality, a very low dimension can also fail to find the latent features. With hyperparameter tuning, 10% of the original feature cardinality (e.g., the cardinality of $f^2 = 486$ maps to $486 \cdot 0.10 \approx 48$ dimensions vector) was selected as the embedding dimension for the experiments.

TABLE II
RESULTS OF SINGLE EMBEDDING OF CONCATENATED FEATURES AND SEPARATE FEATURE EMBEDDINGS, WHILE KEEPING THE OTHER CONFIGURATIONS CONSTANT

Representation Type	Top-3 Accuracy
Concatenated Features and Embedding	51%
Separate Embeddings	63%

C. Contextual preprocessing approaches

We experimented with the multiple contextual preprocessing designs, discussed in section II-E. We preprocessed all the 250,000 sequences. Each preprocessing approach resulted in a different number of sub-sequences. For a single sequence corresponding to one vehicle, while the Last-N Timesteps approach produces only one sub-sequence, the Sliding Window and Overlapping Window approaches produce a large number of sub-sequences due to overlapping repeated events and hence they start to overfit very early. We compared these alternatives to ascertain which one captures the context better. As shown by Table III, the Last-N Timesteps method reached the lowest validation loss of 4.621 and the highest top-3 test accuracy of 61%.

Having good performance using Last-N Timesteps suggests that the LSTM is able to learn dependencies between events as well as sub-sequence breakpoints within sequences, which is a concern that we discussed in section II-E.

D. Using dense layer on top of LSTM layers

We also added a dense hidden layer (with 120 neurons) after the LSTM layers to the best performing preprocessing (Last-N Timesteps) to see if it would help in learning more relations. But as shown in Table III, it didn't result in an improvement in accuracy or validation loss. Despite the increment in the number of parameters, top-3 test accuracy scored 62% and validation loss 4.60.

E. Dropout and recurrent dropout

Dropout, originally introduced in [22], is a regularization technique used in deep learning to prevent overfitting and improve generalization. For each training iteration, dropout randomly selects a set of neurons to be deactivated. It prevents the network from relying only on a few units and forces all units to learn independently. In [23], it is argued that applying the same dropout mask at each time step is more efficient than applying it randomly. The authors also suggest that recurrent activations should be masked with a constant mask, which is often called recurrent dropout.

In Table III, it can be seen that, without dropout, the validation error stopped decreasing in all contextual preprocessing variations, while the training loss was still lower than the validation loss, which showed overfitting in the network. Using recurrent dropout and dropout of 0.2, which means deactivating 20% of the units randomly in the LSTM layer at the time of training, helped controlling overfitting in the network. These dropouts produced the lowest validation error of 4.52 among all the experiments.

F. Comparing LSTM units and wider-deeper networks

The number of LSTM units corresponds to the size of the hidden state, where one state captures one latent feature, for example, the presence or absence of an event that leads to engine malfunction. For the first layer of the LSTM, a final value of 120 units in a single LSTM layer was selected using hyperparameter tuning. One of the architectural choices for

TABLE III
RESULTS WITH MULTIPLE CONTEXTUAL PREPROCESSING APPROACHES, DIFFERENT HYPERPARAMETERS AND ARCHITECTURAL CHOICES

Method	LSTM Layers	LSTM Units	Dense Layer	Dense Units	Embedding Output Dimension	Dropout	Training Loss	Valid Loss	Top-3 Test Acc
Last-N Timesteps	1	120	0	-	$10\% \cdot size(f^i)$	-	4.44	4.62	0.61
Last-N Timesteps	2	64+84	0	-	$10\% \cdot size(f^i)$	-	4.42	4.63	0.60
Last-N (with Dense Layer)	1	120	1	120	$10\% \cdot size(f^i)$	-	4.38	4.60	0.62
Last-N Timesteps	1	120	0	-	$10\% \cdot size(f^i)$	0.2,0.2	4.58	4.52	0.63
Fixed Window	1	120	0	-	$10\% \cdot size(f^i)$	-	5.43	5.20	0.50
Sliding Window	1	120	0	-	$10\% \cdot size(f^i)$	-	5.37	5.33	0.47
Overlapping Window	1	120	0	-	$10\% \cdot size(f^i)$	-	5.38	5.57	0.49
Overlapping Window	2	64+84	0	-	$10\% \cdot size(f^i)$	-	5.39	5.61	0.48

this study was to use a wider network instead of a deeper network [24] by adding more units in the first layer instead of stacking a second LSTM layer on top of the first layer. We noticed that the wider network showed the same or slightly better performance than the deeper network. These results were consistent across all preprocessing approaches in Table III. To perform a balanced comparison for two-layered LSTMs, we kept the number of LSTM parameters close to the single-layered experiment, using 64 units in the first LSTM layer and 84 neurons in the second layer.

IV. Discussion and Conclusion

In this work, we propose a new event-based predictive maintenance model, Sequential Multivariate Fault Prediction (SMFP), which differs from the anomaly detection approaches that rely on sensor data having numeric representations. We show how complex multivariate events, which are non-numeric, can be mapped to continuous representations by jointly learned embeddings. We propose an LSTM based architecture that uses these representations for SMFP.

In order to set a baseline for SMFP, we show various contextual preprocessing approaches and architectural choices including multiple-output modelling, embeddings on raw feature concatenations, and stacking separate embedding layers. Our experiments achieve a baseline of 63% top-3 test accuracy for SMFP. We aim to further improve these results by trying other algorithms such as Seq2Seq models and Attention methods [25], [26].

[Due to confidentiality, associated code can be made public after the paper is accepted.]

References

[1] S. Duffuaa, M. Ben-Daya, K. Al-Sultan, and A. Andijani, "A generic conceptual simulation model for maintenance systems," *Journal of Quality in Maintenance Engineering*, vol. 7, pp. 207–219, 09 2001.

[2] F.M.Discenzo, "Motor diagnostics: technological drivers leading to 21st century predictive diagnostics."

[3] M. Sarnovsky, P. Kostelnik, P. Butka, J. Hreno, and D. Lacková, "First demonstrator of hydra middleware architecture for building automation," 02 2008.

[4] Z. Liang, M. A. C. Martell, and T. Nishimura, "A personalized approach for detecting unusual sleep from time series sleep-tracking data," pp. 18–23, 10 2016.

[5] P. Gaikwad, A. Mandal, P. Ruth, G. Juve, D. Król, and E. Deelman, "Anomaly detection for scientific workflow applications on networked clouds," 07 2016.

[6] H. N. Akouemo and R. J. Povinelli, "Data improving in time series using arx and ann models," *IEEE Transactions on Power Systems*, vol. PP, pp. 1–1, 01 2017.

[7] H.-x. Tian, X.-j. Liu, and M. Han, "An outliers detection method of time series data for soft sensor modeling," pp. 3918–3922, 05 2016.

[8] A. Nanduri and L. Sherry, "Anomaly detection in aircraft data using recurrent neural networks (rnn)," in *2016 Integrated Communications Navigation and Surveillance (ICNS)*, 2016, pp. 5C2–1–5C2–8.

[9] P. Pirasteh, S. Nowaczyk, S. Pashami, M. Löwenadler, K. Thunberg, H. Ydreskog, and P. Berck, "Interactive feature extraction for diagnostic trouble codes in predictive maintenance: A case study from automotive domain," 02 2019, pp. 1–10.

[10] L. Virkkala and J. Haglund, "Modelling of patterns between operational data, diagnostic trouble codes and workshop history using big data and machine learning," 2016.

[11] M. Fransson and L. Fåhraeus, "Finding patterns in vehicle diagnostic trouble codes : A data mining study applying associative classification," 2015.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[13] A. Mccallum and K. Nigam, "A comparison of event models for naive bayes text classification," *Work Learn Text Categ*, vol. 752, 05 2001.

[14] H. P. Luhn, "The automatic creation of literature abstracts," *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 159–165, 1958.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," pp. 1–12, 01 2013.

[16] Pham *et al.*, "Recurrent neural network for classifying of hpc applications," 04 2019, p. 15.

[17] F. A. Gers, D. Eck, and J. Schmidhuber, "Applying LSTM to time series predictable through time-window approaches," pp. 669–676, 2001.

[18] Apache Software Foundation, "Apache spark." [Online]. Available: https://spark.apache.org/

[19] F. Chollet, "open-source library that provides a Python interface for artificial neural networks," https://keras.io/.

[20] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," 2012.

[21] Z. Yin and Y. Shen, "On the dimensionality of word embedding," 2018.

[22] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012.

[23] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," 2016.

[24] M. Z. Alom, T. Josue, M. N. Rahman, W. Mitchell, C. Yakopcic, and T. M. Taha, "Deep versus wide convolutional neural networks for object recognition on neuromorphic system," 2018.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.