Time-series classification for industrial applications: a brake pad wear prediction use case

Evgeny Burnaev
Skoltech, Moscow, Russia
E-mail: e.burnaev@skoltech.ru

Abstract.
Brake system is an important part for control of a vehicle. Hence condition monitoring of brake pads is essential for ensuring passenger’s safety. Many existing methods for brake pads wear assessment rely on specific sensors installed in the brake system, which could be expensive. Instead we use data from existing vehicle’s sensors and electronic control unit that are readily available in modern vehicles. We reduced the prediction problem to time-series classification problem and developed and tested several classification pipelines based on machine learning. We demonstrated that it is possible to predict a brake pad wear with an accuracy sufficient for real-life usage.

1. Introduction
1.1. Brake pad wear prediction problem
Brake pads is an important part of a vehicle brake system. Hence condition monitoring of the brake pad is essential for passenger’s safety [1]. The brake pad wear can be predicted through identification of some symptoms, manifested in measurements from specific sensors [2, 3], using approaches for predictive analytics [4] based on failure prediction [5, 6], anomaly detection [7, 8, 9, 10, 11] and time-series change-point detection [12, 13, 14, 15, 16, 17, 18].

However, mounting of the sensors could be expensive and not possible for vehicles used everyday. Thus the only way to predict brake pad wear is to use data from existing vehicle’s sensors and electronic control unit that are readily available in modern vehicles [1]. In this setup the problem can be reduced to time-series classification problem: during a brake event we observe signals from sensors and should predict the condition of the brake pad.

There exist a number of time-series classification approaches [19, 20]. However, many of these modern approaches exploit deep neural networks, especially convolutional neural networks, for end-to-end time series classification. Although being automatic these approaches lead to construction of non-interpretable black-boxes that often are prohibited in engineering applications, especially when the final aim is deployment of the predictive model in a real production system. Moreover, the resulting predictive models often are too heavy to be used in on-board systems.

To develop a robust and interpretable predictive system in this work we proposed a time-series classification pipeline based on data preprocessing and feature engineering combined with construction of time-series classifiers capable to utilize historic information about previous
braking events. We demonstrated that it is possible to predict a brake pad wear with an accuracy sufficient for real-life usage.

The paper has the following structure. In Section 2 we discuss the proposed approach. In Section 3 we present obtained results, and in Section 4 we draw conclusions.

1.2. Mathematical problem statement
We consider a time-series classification problem statement. We denote by $D = (X, y) = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ a dataset, where $x_k \in \mathbb{R}^{d \times n_k}$ is a $d$-dimensional time series of length $n_k$, and $y_k \in \{1, \ldots, K\}$ is a $K$-class label of the time series. The task is to construct the model that for a new unseen time series $x_*$ predicts its label.

In practice, the time series can be provided in a chronological order, that is $x_k$ is observed earlier than $x_m$ if $k < m$ for $\forall k, m = 1, N$, and the classes $\{1, \ldots, K\}$ are ordered as well. E.g., in case of a brake pad wear prediction in the beginning brake pads are new, then they become used and later they wear out, i.e. labels (New, Used and WornOut) naturally appear in a chronological order [1]. It means that to predict label for a current time series we can use several previous time series (history) to make a predictive model more accurate. Moreover, we also can utilize that the label Used can appear only after New and the label Worn Out appears only after Used.

1.3. Real-world data set description
Data set consists of $\sim 1500$ braking events with brake pad usage condition label (New, Used, WornOut), see [1]. Each braking event is described by a multivariate time series of different signals measured during braking. Some of them have straightforward physical meaning (e.g. acceleration, friction coefficient, pressure and so on), i.e. signals from sensors, and some of them are software parameters recorded by the electronic control unit (ECU) of the braking system. Overall number of parameters is equal to $\sim 400$, with length of each time-series ranging from $\sim 100$ to $\sim 10^3$. There are highly correlated parameters, e.g. characteristics that were measured for each wheel separately. An example of time series is given in Fig. 1 for different brake pads condition.

![Figure 1. Example of time series](image)
2. Proposed Approach
Model construction algorithm consists of the following steps:

1. Preprocessing (cleaning data, removing unimportant columns, etc.).
2. Feature extraction (calculate new features, extract some statistics from time series, etc.).
3. Build a predictive model.

2.1. Preprocessing
Preprocessing of the dataset consists of the following steps

(i) Remove almost constant parameters: some parameters don’t change their values for 99% of braking events.

(ii) Average measurements that correspond to the same parameter but for different wheels.

(iii) Drop some columns that were marked as unimportant by classifier (see below).

(iv) Cut beginning and end of the time series.

(v) Drop observations with constant speed.

Now let us describe each step of preprocessing in more detail.

Removing constant parameters. To identify constant parameters we do the following

(i) For a given parameter calculate its standard deviation for each braking event.

(ii) Calculate number of events for which the standard deviation is close to 0.

(iii) If number of such events is greater than 99% of events mark the parameter as constant.

Averaging parameters over wheels. Some parameters were measured separately for each wheel. For such parameters we do the following

(i) Calculate average over different wheels.

(ii) Put the average to the data set and remove separate measurements.

Cut beginning of time series. A criterion to cut is \( \text{RollingStd}_{50}(V) \leq \frac{1}{5} \max \{ \text{RollingStd}_{50}(V) \} \), where \( V \) is the velocity and \( \text{RollingStd}_{50} \) is a rolling standard deviation of \( V \) with window size 50 (in time ticks, where time tick is the interval between two consequent measurements). This criterion is used to cut all time series of a braking event (i.e. the same split points are chosen for all time series of a braking event). All the constants here are chosen manually, just by evaluating the performance quality of the result: we need to cut the beginning of the time series where the braking doesn’t begin.
Cut end of time series. A criterion to cut is $\text{RollingStd}_{50}(V) \leq \frac{1}{10} \max V \{ \text{RollingStd}_{50}(V) \}$ and $V < 0.2$. This criterion is used to cut all time series of a braking event (i.e. the same split points are chosen for all time series of a braking event). All the constants here are chosen manually, just by evaluating the performance quality of the result: we need to cut the end of the time series where the braking has already ended. Example of cut time series is given in Figure 2.

Drop observations with constant speed. It turned out that after preprocessing there are braking events with constant speed. We find such events by checking standard deviation of the velocity. If it is close to zero, then we drop the observation from the data set. After this procedure 1290 observations left.

2.2. Feature extraction
Standard classification models require feature vectors of fixed dimension as input. Therefore we need to extract feature vectors from time series characterizing their properties relevant for the classification problem.

2.2.1. Global and local features. We distinguish two types of features: global and local.

Global features. By global features we mean features that are calculated using the whole time series, so they contain some aggregated information about the whole signal. We calculate the following global features

- **Absolute energy.** Let $x_i, i = 1, \ldots, T$ be the signal. Then the absolute energy is defined as $\text{abs\_energy} = \sum_{i=1}^{T} x_i^2$. Actually absolute energy is an area under the square of absolute value of the signal over time.
- **Binned entropy.** First we bin the signal values into $K$ bins. Then we compute entropy $\text{binned\_entropy} = -\sum_{k=1}^{K} p_k \log p_k 1_{p_k > 0}$, where $p_k$ is percentage of values in the $k$-th bin. It is a measure of disorder or uncertainty in signal. Higher values indicates higher uncertainty (or complexity) of the signal. We set $K = 10$.

Local features are statistics that we calculate on some intervals of the time series.

To identify reasonable intervals we make adaptive split for each braking event as follows. Typically, absolute values of acceleration increases rapidly at the beginning of the braking, after that it remains more or less constant for some time and in the end it rapidly decreases (see example of absolute value of acceleration in Figure 3). That is why we split the time series into three intervals. Split points remain the same for all time series inside the braking event.

For each brake we find two split points such that each of three intervals are well approximated by a line. As a result the braking event is represented by 3 multi-dimensional but shorter time series. In more details the procedure is performed using brute force, i.e.

(i) Choose 2 split points on time series, construct approximation in each of the 3 intervals and calculate total mean squared error.
(ii) Do this for each pair of points and choose the best one in terms of mean squared error.

Additionally, we impose some constraints on the length of intervals, namely, we restrict the intervals to be longer than 5% of total length of time series. The adaptive split can be speeded up by using dynamic programming (details can be found in [21]). The result of adaptive split is 3 intervals in each of which the time series is more or less homogeneous. Extracting features from homogeneous regions allows to obtain more accurate classification.

In each interval following statistics are calculated (see Figure 4): median, standard deviation, maximum, minimum, linear trend: slope, bias, error of approximation. As a result we obtain
a dataset of size 575 features: after preprocessing 25 parameters left, for each parameter we calculate 2 global features and 21 local features (7 features for each of 3 intervals).

2.3. Models

In this section 2 models are described. The first model corresponds to the first problem statement, where we don’t take into account chronological order of the events. We will refer to this type of models as models without history or the first level models.

The second type of models correspond to the second problem statement where we take into account time stamps of the measurements. These models are based on the first level models: it takes predictions of the first level model (or some features extracted from the model) and constructs another model on top of them (the second level model). We will also refer to this type of models as models with history.

2.3.1. Models without history. Model construction is performed in three steps
(i) Fit Gradient Boosted Decision Trees model (using XGBoost package).
(ii) Calculate feature importances and select several the most important features.
(iii) Refit model using selected features.

Here decision tree is a model that tries to make prediction by learning simple decision rules inferred from the data; Gradient Boosted Decision Tree is an ensemble of decision trees, so that decision trees are constructed iteratively one-by-one and each new decision tree tries to correct approximation error of already constructed ensemble. Details on Gradient Boosted Decision Tree and its implementation in XGBoost package can be found in [22].

Feature Importance can be measured using Gain. Each split of variable improves loss on the training set. The sum of improvements over all splits of variable and all trees in the ensemble is called Gain. The higher the Gain the more important is the variable. In this study we supposed that variable is important if Gain > 10 for this variable.

XGBoost model has a lot of hyper-parameters that allows to fine-tune the model. We used the following set of parameters: we set maximum depth of each decision tree in the ensemble max_depth to 3, as we keep this value small in order to obtain more robust model; we set a number of trees in the ensemble n_estimators to 400, as according to our experiment larger number of trees gives only a marginal improvement (if gives it at all) but makes the model much more complex; we set a weight of each newly constructed tree in the ensemble learning_rate to 0.05 as reducing learning rate allows to avoid overfitting at the cost of slower training and larger number of trees.

2.3.2. Models with history. Brake pad is not worn out immediately. It is a continuous process, therefore, if during several brakings brake pads are not worn out then with high probability they are not worn out at current braking. This means that we can use several consecutive brakings to improve prediction for current one. It is supposed that we already have the first level model (model without history). Then we can do the following procedure:

1. For braking events $t, t - 1, \ldots t - k + 1$ calculate probabilities $p_i = (p_{t-i}^1, p_{t-i}^2, p_{t-i}^3)$ for $i = 0, k - 1$ using the first level model, where $p_j^c$ is a probability of class $c$ for braking event $j$.

2. Use the probabilities $p_i$ as features for the second level model to make predictions (see Figure 5).

For the second level we considered the following models.
Majority voting. The prediction is given by $\hat{y} = \text{Majority}(\arg\max_i p_i)$. This can be seen as a simple baseline model. In this model each $p_i$ is interpreted as an expert. The model assumes that all experts are independent and if they are not, the quality can be lower than expected. Note, that when we use the predictions of the first level models they are not independent. The model has no parameters.

Gradient Boosted Decision Trees (XGBoost). See details in section 2.3.1. In this case we used slightly different parameters: $n_{\text{estimators}} = 100$, $\text{max_depth} = 3$, $\text{learning_rate} = 0.1$.

Logistic Regression (LogReg). This is a simple linear model. The main assumption of the model is that the probability of the target label is expressed by $p(y = c|x) = \frac{\exp(-(w_c,x)+b_c)}{\sum_k \exp(-(w_k,x)+b_k)}$, where $\{w_c, b_c\}_{c=1}^C, w_k \in \mathbb{R}^d, b_k \in \mathbb{R}$ are parameters of the algorithm, $d$ is the dimensionality of the feature vector. We used implementation of LogReg from scikit-learn Python package\(^1\).

Multi-layered perceptron (MLP) is a feed-forward neural network. The output of the model is given by $z_L = a_L(W_{L-1}z_{L-1} + b_{L-1}) + b_L$, where $W_l \in \mathbb{R}^{h_l \times h_{l-1}}, b_l \in \mathbb{R}^{h_l}$, $z_0 = x$ for $l = 1, \ldots, L$ and $a_l(\cdot)$ is an activation function. For $a_l(\cdot)$ we used ReLU activation function $a_l(x) = x \cdot 1_{x > 0}$. So, MLP feeds the data to a sequence of layers where each layer performs simple non-linear mapping (activation function) of the input. Result of the composition of the mappings is highly non-linear transformation of the input data. MLP with one layer is equivalent to Logistic Regression. The parameters of the MLP, namely $W_l, b_l$ for all $l$, are tuned by minimizing cross-entropy loss via Stochastic Gradient Descent (see [23]).

In an architecture of MLP that we used we set number of layers $L$ to 3, as small number of layers reduce overfitting; we set hidden layer sizes $h_l$ to $(100, 100)$, as larger hidden layer sizes provide more complex networks. In our experiments we tried different combinations of $L$ and $h_l$ and provide results only for the best one. For MLP we used implementation from scikit-learn Python library.

Recurrent neural network (RNN) is a type of artificial neural network that is applicable to sequential data due to its architecture. RNNs are called recurrent because they perform the same task for every element in a sequence, with the output being dependent on previous computations. Typically RNN looks like the one depicted in Figure 6. We used a special RNN architecture called Gated Recurrent Unit (GRU, first proposed in [24]). This model sequentially calculates the following values $z_t = \sigma(W_{zh}h_{t-1} + W_{zx}x_t), r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t), \tilde{h}_t = \tanh(W_r(r_t \cdot h_{t-1}) + W_zx_t), h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$, where $h_t$ is a hidden state and also the output of RNN which is then passed to the Logistic Regression or MLP to perform classification (so, we can think of $h_t$ as features extracted from $x_t$ taking into account all the previous values $x_{t-1}, x_{t-2}, \ldots$, $W_{zh}, W_{zx}, W_{rh}, W_{rx}, W_r, W_z$ are matrices which are tuned during

\(^1\) https://scikit-learn.org/
training. Vector $h_{t-1}$ is a feature vector for the previous element $x_{t-1}$ and can be seen as a memory (past). The meaning of other values is the following

- **Update gate** $z_t$. $\sigma$ in equation for update gate is a sigmoid function. So, it takes values in $(0, 1)$ interval and defines how much information to pass to the future.
- **Reset gate** $r_t$. It also takes values in $(0, 1)$ interval and defines how much the past information to forget.
- **Current memory content** $\tilde{h}_t$. It takes values in $(-1, 1)$ interval and, thus, not a gate. It contains useful information about the $x_t$ taking into account the past.
- **Final memory content** $h_t$. It’s a linear combination of the past and current memory content.

We can stack several GRU units to create deep RNN. We used the following structure of the RNN: $\text{hidden dim} = 128$ — this is a dimension of hidden state $h_t$; $\text{n_layers} = 5$ — number of layers, i.e. stacked GRU units; $\text{bidirectional} = \text{True}$ — the same sequence can be passed to the RNN in both straight and reversed order and then the results are concatenated. This makes sense because usually RNN better memorizes the most recent elements of the sequence, but passing the sequence in both directions allows to better memorize older elements in the sequence; $\text{n_mlp_layers} = 2$, $\text{mlp_hidden_sizes} = (64, 64)$ — parameters of MLP that is used on top of GRU.

**Training of RNN.** We used pytorch\(^2\) Python library to build RNN model. RNN model is usually hard to train because we have a lot of parameters in our model and rather small data set, so it overfits easily. In order to make the procedure more stable and prevent overfitting we applied several tricks

- **Dropout.** This is a special regularization technique to reduce overfitting (see [25]). The idea is that during training random weights of the neural network are set to zero. The intuition is the following. When one set of weights is zeroed-out we have one network with smaller number of parameters, with another set of zeroed-out weights we have another network. Applying dropout is in some sense equivalent to training an ensemble of networks with smaller number of parameters (which are less prone to overfitting). We used $\text{dropout rate} = 0.5$ — average number of weights zeroed-out at each learning iteration.

- **Learning rate scheduler.** Learning rate controls how large is an update to the neural network after each step of stochastic gradient descent. We change learning rate at each iteration — start from some initial value, then monotonically decrease for several iterations, after that set it back to initial value (restart) and repeat. Such restarts allows to escape from local minimum and find more robust one with less overfitting.

- **Adding noise to the input data.** The input of the RNN is a prediction of XGBoost model which is not perfect. To take it into account and be more robust we incorporate noise to the inputs. Predictions of XGBoost is a vector of probabilities $p_i$. We generate noisy probabilities by sampling them from Dirichlet Distribution with parameter $p_i/\tau$, where $\tau > 0$ is a parameter. The higher its value the more noisy the samples. For values close to zero, samples almost coincide with the original vector of probabilities.

2.4. **Quality assessment**
The training set is rather small, therefore quality assessment is performed via $N$-by-$M$ cross-validation procedure

(i) Split the training set into $M$ non-intersecting random subsets.

\(^2\) [https://pytorch.org/](https://pytorch.org/)
Figure 6. RNN and the unfolding in time of its forward computation.

(ii) Use $M - 1$ subsets to fit the model and use the remaining one (validation set) to calculate predictions of the model.

(iii) Repeat step 2 $M$ times each time choosing different validation set. After this we obtain predictions for each point from the initial training set.

(iv) Calculate quality metrics using obtained predictions.

(v) Repeat the whole procedure $N$ times.

(vi) Calculate mean and standard deviation of quality metrics.

2.4.1. Splitting into train/validation for models with history. There is some subtlety how the data set should be splitted into training and validation folds when we use a model with history (see Figures 7 and 8).

- Splitting for model without history: validation fold is a random subset from the given data set, training fold is the rest of the data set. In this case if $x_t$ is in the validation fold, then with high probability $x_{t-1}, x_{t+1}$ are in the training fold. So, the model just needs to interpolate one sample between ($t - 1$)-th and ($t + 1$)-th elements.

- Splitting for model with history: choose random $x_t$ from the data set and put to the validation set the sequence $(x_t, x_{t-1}, \ldots, x_{t-h})$ (this procedure is repeated while the desired validation fold size is not reached). The training fold is the rest of the data set. In this case the model has to interpolate large gap between $x_{t-h-1}$ and $x_{t+1}$.

This means that the quality of the first level models will differ for different schemes of splitting: in the first case we expect the quality to be higher than in the second case.

2.4.2. Quality metrics. The following metrics are calculated:

Accuracy. Accuracy = \( \frac{1}{N} \# \{ \hat{y}_i = y_i, i = 1, \ldots, N \} \), where $\hat{y}_i$ is a predicted label, $y_i$ is a reference label and $N$ is a dataset size. The value of Accuracy is between 0 and 1. For ideal classification Accuracy is equal to 1.
**Figure 9.** ROC-curve and confusion matrices for models without history: before feature selection.

**F1-score** for binary classification (there are two labels – positive and negative) is equal to F1 = 2 * \( \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \), where TP is true positives, i.e. number correctly predicted positive labels, FP is false positives, i.e. number of incorrectly predicted positive labels, FN is false negatives, i.e. number of incorrectly predicted negative labels.

In multiclass case F1 score can be calculated for each class separately and then aggregated. In this work we used weighted aggregation which means that we take weighted sum of F1 scores for each class, weights are inversely proportional to class size. The value of F1-score is between 0 and 1. Good classification has F1-score close to 1.

**Area under ROC curve.** In a binary case a classifier usually returns probability \( p \) of positive class. Decision on the predicted label is made by the following rule: \( \hat{y} = \text{positive} \) if \( p > c \), and negative otherwise, where \( c \) is some threshold. Depending on \( c \) different TPR and FPR can be obtained (\( TPR = \frac{TP}{TP+FN} \) is a true positive rate, \( FPR = \frac{FP}{FP+TN} \) is a false positive rate).

ROC curve is FPR vs. TPR curve for \( c \in [0,1] \). In multiclass case we can do the following: calculate \( TP_k \) and \( FP_k \) for each class \( k \), then compute \( TP = \sum_k TP_k, FP = \sum_k FP_k, FN = \sum_k FN_k, TN = \sum_k TN \) and FPR and TPR as usual (this procedure is also called micro-averaging). ROC curve for random guess corresponds to \( y = x \) line. The higher lies the curve the better works the algorithm. Qualitatively this effect is measured by calculating area under the curve (AUC).

**Confusion matrix** is a matrix, \( (i,j) \) element of which is a number of cases when \( i \)-th class was predicted as \( j \)-th class. For ideal classifier the matrix should be diagonal.

### 3. Results

In this section experimental results of application of the described approach are presented.

#### 3.1. Models without history

In Figures 9 and 10 ROC-curve and confusion matrices are given for models before and after feature selection, the Table 1 presents the quantitative results of quality assessment. If we perform feature selection we obtain model with almost the same quality but with lower number of parameters, thus, more robust. It means that we overfit less.

Note from confusion matrix that there are brakings with **Worn Out** brakes that were classified as **New** or **Used**. The cost of such misclassification can be large. Such misclassification we will call False Negatives (FN) (in this case we suppose that positive class is **Worn Out** and all other classes are negative).
Figure 10. ROC-curve and confusion matrices for models without history: after feature selection.

Table 1. Quality metric values for models without history

| Metric      | Before feature selection | After feature selection |
|-------------|--------------------------|-------------------------|
| ROC AUC     | 0.86 ± 0.008             | 0.86 ± 0.009            |
| F1-score    | 0.70 ± 0.019             | 0.70 ± 0.014            |
| Accuracy    | 0.70 ± 0.019             | 0.71 ± 0.014            |

Table 2. Accuracy of models with history for history size 10

| Model      | Accuracy          | Accuracy of 1st level model |
|------------|-------------------|-----------------------------|
| Majority Voting | 0.76 ± 0.037   | (theoretical 0.94)          |
| LogReg     | 0.85 ± 0.036     |                             |
| XGBoost    | 0.84 ± 0.032     | 0.69 ± 0.020               |
| MLP        | 0.80 ± 0.043     |                             |
| RNN        | 0.85 ± 0.033     |                             |

Besides XGBoost approach we also used models based on convolutional and recurrent neural networks, and support vector machine classification. However, these models provided significantly lower accuracy.

3.2. Models with history

Table 2 provides quality of different models with history for history size 10 (here we report only accuracy as some models don’t provide probabilities as predictions, only labels). We also report the accuracy of 1st level model on top of which the 2nd level model was built. For MajorityVoting we also provide the theoretical estimate of the quality assuming that 1st level models predictions are independent. Theoretical estimate can be computed as follows. Suppose that the accuracy (the probability of correct prediction) of 1st level model is $p$. Then, the probability that the majority of predictions is correct is equal to $P_h = \sum_{i=\lfloor h/2 \rfloor}^h \binom{h}{i} p^i (1-p)^{h-i}$.

For top-2 best performing models we conducted experiments with various history sizes (results are given in Table 3). From the table it can be seen that LogReg models accuracy drops with larger history size, while RNN’s remains at the same level. Decrease in accuracy is explained by the decrease in accuracy of the first level model. For history size 10 the accuracy of LogReg and RNN models is the same, so, in this case LogReg is preferred as it is much more simple. However, the accuracy of LogReg model largely depends on the accuracy of the first level model, while RNN is more robust to it.
Table 3. Accuracy for different history sizes

| Model   | History size | Accuracy   | Accuracy of 1st level model |
|---------|--------------|------------|----------------------------|
| RNN     | 10           | 0.85 ± 0.033 | 0.67 ± 0.063 |
| LogReg  | 10           | 0.85 ± 0.036 |                          |
| RNN     | 20           | 0.84 ± 0.060 | 0.63 ± 0.072 |
| LogReg  | 20           | 0.82 ± 0.086 |                          |
| RNN     | 30           | 0.81 ± 0.100 | 0.52 ± 0.140 |
| LogReg  | 30           | 0.77 ± 0.120 |                          |
| RNN     | 50           | 0.82 ± 0.145 | 0.57 ± 0.124 |
| LogReg  | 50           | 0.76 ± 0.122 |                          |

Table 4. Characteristics of the second level models. Here $d_{in}$ is dimensionality of input $x_t$, $n_l$ is total number of layers in neural network, $n_t$ is a number of trees in XGBoost, $d$ is a tree depth, $d_h$ is max dimensionality of hidden layers in neural network, $h$ is history size. Measurements were recorded on Intel(R) Core(TM) i7-7820X CPU @ 3.60GHz. Concrete values of parameters see in Section 2.3.2.

| Model      | # parameters | Computational complexity | Evaluation time | Space on disk |
|------------|--------------|--------------------------|-----------------|---------------|
| Logistic Regression | $\mathcal{O}(d_{in})$ | $\mathcal{O}(hd_{in})$   | 0.4 ms          | 41 Kb         |
| MLP        | $\mathcal{O}(n_l \times d_h \max(d_h, d_{in}))$ | $\mathcal{O}(hn_t d_h \max(d_h, d_{in}))$ | 2.4 ms          | 110 Kb        |
| XGBoost    | $\mathcal{O}(n_l \times (2^{d+1}))$ | $\mathcal{O}(hn_t d_h)$ | 0.33 ms         | 815 Kb        |
| RNN        | $\mathcal{O}(n_l \times d_h \max(d_h, d_{in}))$ | $\mathcal{O}(hn_t d_h \max(d_h, d_{in}))$ | 0.07 s          | 5 Mb          |

3.3. Computational and Memory Complexity

Average braking event from the given training set requires 1980 Kb. All brakings have different length, so actual value can be larger or smaller. After feature extraction 4.5 Kb memory is required to store one braking event. This value is constant for all braking events.

Characteristics of different models are given in Table 4. The most lightweight model is LogReg. Moreover, this model provides one of the best accuracies with history size 10. Therefore, this model can be used for on-board implementation. RNN model is the heaviest one, although it provides quality comparable to LogReg. However, we expect that this model will work better for larger training set. This model can be used for cloud implementation.

4. Conclusion

We considered the problem of predicting brake pad usage condition using the characteristics measured during braking. The optimal model (optimal in terms of trade-off between accuracy and the memory required to store the model) that we have constructed is based on Gradient Boosted Decision Trees (XGBoost) with preprocessing and feature extraction (accuracy $\sim 0.70$).

Accuracy can be improved by using history of braking events (accuracy $\sim 0.85$). We developed two models, that shows the same quality for a given data set. The first one is lightweight and is based on XGBoost + Logistic Regression. This model can implemented on-board. The second model is more complex and is based on XGBoost + Deep Recurrent Neural Network. It can be implemented in a cloud. We expect that this model will provide higher accuracy if trained on larger data set. We also tried several other models, though they didn’t show good enough accuracy.

Acknowledgments

Evgeny Burnaev would like to thank Efim Boeru for a fruitfull collaboration in the framework of this project, and Yermek Kapushev for performing computational experiments.
References

[1] Hoffmann A, Lehner S, Weissenbacher C, Zechlin H 2018 Audi A3 e-tron Boxberg Driving Test Technical report Robert Bosch GmbH, CC-AS/EYB3, 74232 Abstatt, Germany.

[2] Jegadeeshwaran R, Sugumaran V 2015 Brake fault diagnosis using clonal selection classification algorithm (csca) — a statistical learning approach Engineering Science and Technology, an International Journal 18 1 pp 14–23

[3] Chetan C Harlapur, Priyatamkumar Kadiyala, and Ramakrishna S 2019 Brake pad wear detection using machine learning International Journal of Advence Research, Ideas and Innovations in Technology pp 498–501

[4] Burnaev E 2019 Algorithmic foundations of predictive analytics in industrial engineering design Journal of communications technology and electronics 64 12 pp 1485–92

[5] Burnaev E 2019 On construction of early warning systems for predictive maintenance in aerospace industry Journal of communications technology and electronics 64 12 pp 1473–84

[6] Burnaev E 2019 Rare failure prediction via event matching for aerospace applications In Proceedings of the 3rd International Conference on Circuits, System and Simulation (ICCSS—2019) pp 214–20

[7] Smolyakov D, Sviridenko N, Burikov E, and Burnaev E 2018 Anomaly pattern recognition with privileged information for sensor fault detection In Proc. ANNPR vol 11081 (Springer LNCS Proceedings) pp 320–32

[8] Burnaev E, Erofeev P, and Papanov A 2015 Influence of resampling on accuracy of imbalanced classification In Proc. SPIE vol 9875

[9] Burnaev E, Erofeev P, and Smolyakov D 2015 Model selection for anomaly detection In Proc. SPIE vol 9875

[10] Burnaev E and Smolyakov D 2016 One-class svm with privileged information and its application to malware detection In Proc. IEEE 16th ICDM Workshops pp 273–80

[11] Smolyakov D, Korotin A, Erofeev P, Papanov A, and Burnaev E 2019 Meta-learning for resampling recommendation systems In Proc. SPIE vol 11041

[12] Burnaev E and Golubev G 2017 On one problem in multichannel signal detection Problems of Information Transmission 53 4 pp 368–80

[13] Artemov A and Burnaev E 2015 Ensembles of detectors for online detection of transient changes In Proc. SPIE vol 9875

[14] Artemov A, Burnaev E, and Lokot A 2015 Nonparametric decomposition of quasi-periodic time series for change-point detection In Proc. SPIE vol 9875

[15] Volkonskii D, Burnaev E, Nourotdinov I, Gammerman A, and Vovk V 2017 Inductive conformal martingales for change-point detection In Proc. of the 6th Workshop on Conformal and Probabilistic Prediction and Applications vol 60 (Proceedings of Machine Learning Research) pp 132–53

[16] Safin A and Burnaev E 2017 Conformal kernel expected similarity for anomaly detection in time-series data Advances in Systems Science and Applications 17 3 pp 22–33

[17] Ishimtsev V, Bernstein A, Burnaev E, and Nazarov I 2017 Conformal k-nn anomaly detector for univariate data streams In Proc. of the 6th Workshop on Conformal and Probabilistic Prediction and Applications vol 60 (Proceedings of Machine Learning Research) pp 213–27

[18] Smolyakov D, Sviridenko N, Ishimtsev V, Burikov E, and Burnaev E 2019 Learning ensembles of anomaly detectors on synthetic data In Proc. of the 6th Workshop on Conformal and Probabilistic Prediction and Applications vol 60 (Proceedings of Machine Learning Research) pp 292–306

[19] Burnaev E, Koptelov I, Novikov G, and Khanipov T 2017 Automatic construction of a recurrent neural network based classifier for vehicle passage detection In Proc. SPIE vol 10341

[20] Wang Z, Yan W, and Oates T 2017 Time series classification from scratch with deep neural networks: A strong baseline In Proc. International Joint Conference on Neural Networks (IJCNN) pp 1578–85

[21] Burnaev E and Menshikov I 2009 A model of the functional state of participants of laboratory markets Journal of Computer and Systems Sciences International 48 1002

[22] Chen T and Guerlin C 2016 XGBoost: A scalable tree boosting system In Proc. of the 22nd ACM SIGKDD, KDD’16 pp 785–94

[23] Bishop C 2006 Pattern recognition and machine learning (Springer)

[24] Cho K, Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, and Bengio Y 2014 Learning phrase representations using rnn encoder-decoder for statistical machine translation arXiv preprint arXiv:1406.1078

[25] Srivastava N, Hinton G, Krizhevsky A, Sutskever I, and Salakhutdinov R 2014 Dropout: a simple way to prevent neural networks from overfitting The journal of machine learning research 15 1 pp 1929–58