Project for the Process Mining Course – academic year 2020-21

Prediction with Timestamp Encoding

Luca Dal Zotto - 1236343
luca.dalzotto.1@studenti.unipd.it

Assignment

In this project we aim at enriching classical control-flow encodings with information derived from timestamps. Having features capturing more than one aspect related to the timestamp of the execution of an event could improve the accuracy of the prediction and could be useful to make it clear what time-related aspect(s) influence(s) most the predictions.

1. Introduction

In this project, we were asked to face a prediction task starting from an event log, exploiting what we have learnt during the lectures. In particular, a certain emphasis will be given to the encoding phase, exploring the contribution of time related features.

The dataset considered for this purpose is the real-life event log used for the first Business Process Intelligence Challenge, in 2011. It consists in 1140 cases, and each trace collects the different phases that a patient of the Dutch Academic Hospital undergoes. As we will see, each trace has some attributes, which are features related to the patience or to the trace itself, that do not change. Besides, the events of the traces have their own attributes too. The event log has been provided already split in training and test set, with a proportion equal to 80% and 20% of the cases, respectively.

In the BPI scenario, great importance is given to the encoding phase, in which all relevant information are extracted from the event log and rewritten in a format that allows further analysis, as well as the training of machine learning algorithms. In this specific case, we were asked to perform two encodings: one using only the event names of the traces (simple index encoding) and the other including the timestamps features.

Then, some classifiers are trained on the encoded dataset, optimizing some of their hyper-parameters through a grid-search. The best models are then evaluated in the test set and compared.

The structure of this report is the following: in the next section, I will describe the methods adopted for the analysis, describing the overall structure of the code, and how I implemented the most relevant functions. In particular, I organized it in four subsections: one regarding the timestamp encoding, one reporting the data exploration results, one showing how I created the datasets, and one where I describe the training procedure. Then, in the third section, I will show the results obtained. Finally, in the last section, I will comment them, and draw some conclusions.

Along with this report, I provided the commented Python code in the notebook format. In this way, I managed to insert some text sections to better describe the rationale I followed. Moreover, the reader can already see the results even without running the code. In case the teacher desires to execute the code, I suggest to copy the folder "PM Project Dal Zotto" in Drive, open the notebook `PM Project Dal Zotto.ipynb` using Google Colab and follow the procedure described in it. Since I fixed some random seeds, the results should be reproducible.

2. Methods

The dataset provided is in the XES format. To manipulate it, I used the `pm4py` library. A data log consists on a list of traces. Each of them is composed by different activities (events), which represents the different steps of the process. As anticipated before,

we have both trace-level and event-level attributes. Among these, there is the timestamp. This attribute requires some extra work to provide some relevant information in a good format. Now I will describe the four functions suggested by the teacher, which together provide the desired encodings.

2.1. Timestamp Encoding

The first one is `encode_trace_simple_index`. It takes as input a trace and return a vector encoding the trace using simple index encoding. To achieve this result, I simply iterate over the events of the trace and append to a vector the name of the events. A crucial aspect is that the returned vector should be of the same length, regardless of the trace length, otherwise, some samples of the encoded dataset would have a different number of features. To avoid it, I computed the prefixes of length 20, meaning that I considered only the first 20 occurring events. Moreover, in case a trace was composed by less than 20 events, I used padding, namely, I added a certain number of empty strings until reaching a vector of length 20.

Secondly, I wrote an auxiliary function called `encode_timestamp` which takes a timestamp of an event and returns a vector containing different features extracted from it: the timestamp itself in epoch timestamp, the timestamp itself in human-readable format, the day of the year, the day of the week, the month, the hour of the day in local time, the minutes of the hour in local time and the concatenation of hour and minutes. To do so, I used the `Pandas` library, since it provides straightforward ways to access the desired information.

Thirdly, I implemented another auxiliary function, `encode_event_simple_index_with_timestamp`, which, given an event, encodes it using simple index with timestamp encoding. Essentially, it just concatenates the event name attribute with the timestamp vector encoding returned by the previous function.

Finally, I managed to implement the function `encode_trace_simple_index_with_timestamp`, which takes as input an execution trace and returns as output a vector encoding it using the simple index with timestamp encoding. Similarly to the first function, it iterates over the events of the trace, but instead of saving just the event names, it calls the function `encode_event_simple_index_with_timestamp` for each event. The same observations regarding prefixes and padding seen for the first function, hold also in this case, with the only difference that, to implement padding, I need to add also some zeros for the time features.

2.2. Data Exploration

Before creating the datasets, I performed a small data exploration, to obtain a better knowledge of the data at hand, and also to familiarize with an event log in XES format.

I started with the trace attributes, which essentially are features that do not change during the trace execution. Here I list them, reporting some quick observations. Further details, such as the distribution of the variables, can be observed in the provided notebook.

- name: it is just the ID of the trace, so it is not useful for prediction purposes;
- age: the age of the patient. It has 70 possible numerical values;
- diagnosis: 78 possible nominal values;
- diagnosis_code: 11 possible alphanumeric values;
- diagnosis_treatment_combination_ID: it has a different value for each trace, so it is not useful;
- specialism_code: 3 possible alphanumeric values;
- treatment_code: 34 possible alphanumeric values;
- start date: timestamp attribute;
- end date: timestamp attribute.

From the last 2 attributes, I extracted the time features listed in the previous subsection. I noticed that, in the training set, the month, hour and minute are always the same, so the related variables are not informative and should be discarded. Moreover, keeping all other time features could be a wrong choice, since in some cases they are redundant, and this may lead to an higher model complexity, affecting the performances.

I found out that some traces are without the attributes "treatment_code" and "end date". A possible explanation is that they are pre-mortem data.

In addition to them, there is also the label attribute, which represents the response value for the prediction task. It takes only 2 values (true or false), so we will be dealing with a binary classification problem. The dataset is just slightly unbalanced: around 60% of the samples are labelled as false. Therefore, the accuracy should be fairly reliable as classification metric. However, I will also compute other metrics: weighted accuracy, precision, sensitivity, specificity and F-score, along with the confusion matrix.

Then, I repeated a similar analysis for the event attributes. Here I report the most relevant information:

- name: 582 possible values;
- activity_code: 627 possible alphanumeric values;
- producer_code: 113 possible alphanumeric values;
- specialism_code: 23 possible alphanumeric values;
- number_of_execution: 36 possible alphanumeric values;
- lifecycle:transition: just one value, so it is useless;
- section: not present in all traces;
- group: not present in all traces;
- timestamp.

Even in this case, from the timestamp attribute, I extracted the time features listed in the previous subsection. I found that the hour has only 2 possible values, while minutes is always the same. The same considerations about redundant variables hold.

2.3. Dataset creation

Now that we have a deeper knowledge of the information available, it is finally possible to prepare the dataset in a format that can be used as input for the predictive models. However, before doing so, I also implemented two auxiliary functions that can be used to add also event level attributes, obtaining a complex index encoding. This step was not asked, but I found it interesting to explore the contribution of different levels of information. These two functions are:

- `encode_event_complex`: takes as input an event and returns a vector with all its attributes;
- `encode_trace_complex`: takes as input a trace and returns a vector encoding it using complex index encoding.

Since I need to convert the event name (and all the other non-numerical values) into a numeric format, I wrote the function `encode_categorical` which solves this task using the `sklearn` implementation of `LabelEncoder`. I prefered it to one-hot encoding, since this option would have generated to many variables, due to the high number of possible values, increasing the complexity of the problem.

The function used for creating the datasets is `create_dataset`. It takes as input the log and some parameters, which are used to choose the information that will be kept. The parameters are: event_timestamps, trace_attributes, trace_timestamps and complex_encoding. If they are all set to false, then, simple index encoding is applied. Otherwise, it is possible to include event timestamps, trace timestamps, trace attributes and event attributes. I decided to create five different datasets, to have the possibility to understand the contribution of different features. The information present in these datasets is reported in the Table 1.

dataset name	**event names**	**trace attributes**	**time features**	**other event attributes**
simple	yes			
simple_trace	yes	yes		
time	yes		yes	
time_trace	yes	yes	yes	
complex	yes	yes	yes	yes

Table 1: Variables considered in each dataset.

2.4. Models training

For each of these datasets, I will train 3 different supervised classifiers: Logistic Regression, Decision

Tree and Random Forest, all implemented using the sklearn library.

In all cases, I will optimize some hyper-parameters performing a grid search: essentially, for each hyper-parameter to optimize, a list of candidate values is defined, and all possible combinations are tried. To have more reliable results in the optimization phase, I used a 5-fold cross-validation on the training set. Note that, the cross-validation has been implemented only in this optimization phase, while for the final tests, I kept a separate test set. Another option would have been to perform cross-validation also for the testing phase. However, a nested cross-validation would have increased a lot the computational requirements, so I discarded this approach.

Logistic Regression is one of the most simple machine learning classifiers. Basically, it combines linear regression with the logistic function, which takes values in [0,1]. In this way, it returns the probability for a sample to belong to a certain class. The hyper-parameters considered for this model are the maximum number of iterations and the regularization term, which is very important since it allows to find the correct complexity level, avoiding both overfitting and underfitting.

Decision Tree consists in splitting recursively the whole dataset according to the values of a variable. The variable and the threshold value can be chosen according to the entropy criterion or the Gini index. I opted for the latter, since it is the default value in the sklearn implementation. An important hyper-parameter is the maximum depth of the tree, which is the maximum number of splits. The other one that I optimized was the the minimum number of samples required to split an internal node.

One of the limits of a Decision Tree is that it tends to overfit the training data, reducing the generalization capacity. To improve this aspect, we can use Random Forest, which essentially combines different Decision Trees, obtaining a better prediction. The hyper-parameters considered in this case are the number of trees and the maximum depth of each tree.

An important observation is that, due to the way the learning algorithms are defined, both Decision Tree and Random Forest provide a feature ranking, which explains which variables gave the higher contribution in solving the classification task.

3. Results

In order to evaluate the performances of the models, I wrote two auxiliary functions:

- `metrics_results`, which takes as input the target vector and the predictions and computes the confusion matrix;
- `metrics`, which takes as input the four entries of a confusion matrix and computes some of the most relevant classification metrics: accuracy, weighted accuracy, precision, sensitivity, specificity and F-score.

For each of the five previously defined datasets and for each of the three classifier, I performed the following steps:

1. take the hyper-parameters giving the better results in the training set (remember that I used cross-validation in the training set to optimize the hyper-parameters);
2. re-train the model in the whole training set, using the optimal hyper-parameter configuration;
3. compute the model predictions both on the training and test set;
4. print the training accuracy (to have an indication telling whether the model is overfitting or underfitting);
5. print the confusion matrix and the classification metrics of the test set predictions.

In Figure 1, I represented the test set accuracy reached by each model: I grouped the histograms by dataset (see the label on the x-axis) and for each group, I plotted side by side the results of the three classifiers using different colors (see legend).

To make a more direct evaluation of the effect of adding the time features, in Figure 2, I plotted in blue the histograms representing the accuracy reached by the models using simple index encoding, while in orange the accuracy obtained using simple index with timestamps encoding. In this case, the histograms were grouped by models.

To conclude I would like to focus on two of the best models: Decision Tree and Random Forest trained on

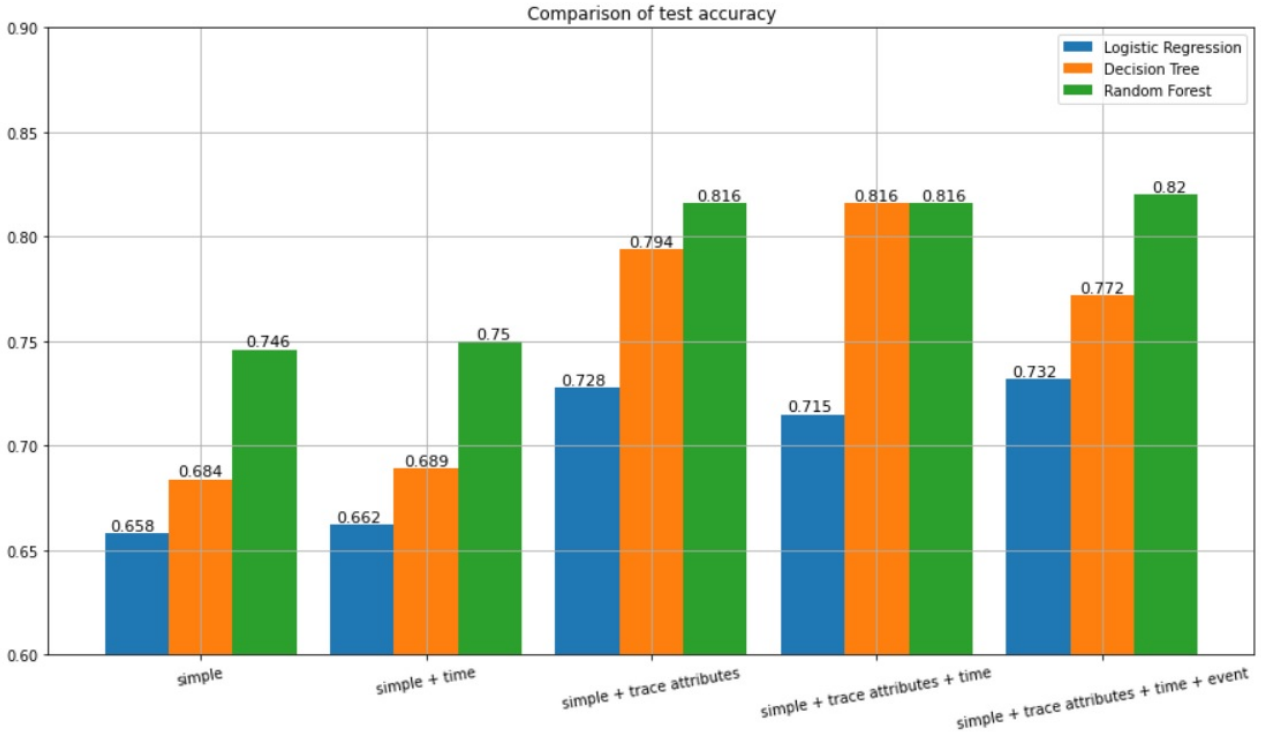

Figure 1: Comparison of the results of all models.

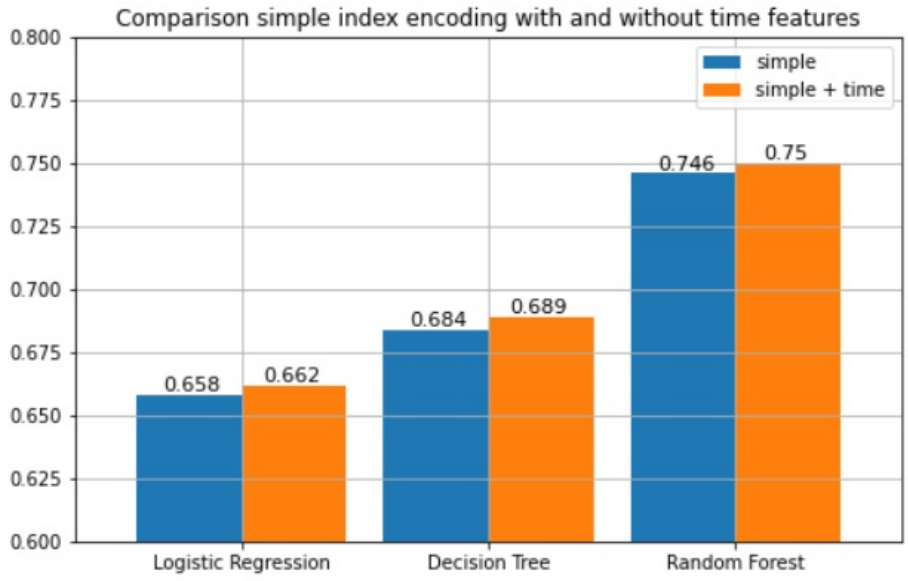

Figure 2: Direct comparison of simple index encoding with and without time features.

Classifier	Accuracy	Weighted Accuracy	Precision	Sensitivity	Specificity	F-score
Decision Tree	0.816	0.807	0.778	0.761	0.853	0.769
Random Forest	0.816	0.816	0.75	0.815	0.816	0.781

Table 2: Classification metrics of two models.

the dataset with both timestamps and trace attributes. In Table 2, I reported their classification metrics.

For these two models, I also analyzed the feature ranking that is automatically computed during training. These are the 15 variables contributing the most to the Decision Tree classifier:

1) diagnosis_code - importance = 0.192
2) age - importance = 0.0301
3) day_of_year_s - importance = 0.0273
4) event_5 - importance = 0.0254
5) event_10 - importance = 0.0237
6) day_of_year_20 - importance = 0.0224
7) event_20 - importance = 0.0208
8) readable_format_8 - importance = 0.0206
9) readable_format_20 - importance = 0.0201
10) specialism_code - importance = 0.0178
11) readable_format_s - importance = 0.0171
12) diagnosis - importance = 0.0144
13) event_8 - importance = 0.0137
14) day_of_week_8 - importance = 0.0137
15) day_of_week_17 - importance = 0.0126

Instead, this is the ranking provided by the Random Forest classifier:

1) diagnosis - importance = 0.0369
2) event_15 - importance = 0.0338
3) event_16 - importance = 0.0298
4) event_19 - importance = 0.0297
5) age - importance = 0.0257
6) event_20 - importance = 0.0242
7) event_17 - importance = 0.0234
8) readable_format_20 - importance = 0.0231
9) day_of_year_20 - importance = 0.0199
10) month_19 - importance = 0.0185
11) day_of_year_19 - importance = 0.0157
12) event_18 - importance = 0.0157
13) readable_format_s - importance = 0.0155
14) readable_format_19 - importance = 0.0154
15) day_of_year_18 - importance = 0.0142

4. Comments and Conclusions

Let's now comment the results obtained, starting from comparing the performances on the dataset using simple index encoding and the one adding also timestamps. From Figure 2, we can see that adding time features leads to a small increment in performances, on average around 0.5%. Therefore, we can say that time variables improve the predictive models, but not that much. Here I list some observations and possible explanations about this fact:

- some time features may not give useful information (for example, the timestamp in epoch timestamp, the concatenation of hour and minutes, ...);
- some time features may be redundant;
- I have added the time features for all events: maybe, only those regarding the last events or the most important events are relevant;
- adding different time features for each event increases a lot the dimensionality of the problem, and therefore, the computational complexity. For this reason, an algorithm such as Logistic Regression may need more iterations to reach better results.

A possible solution to this problem could be to perform a feature selection, or to apply some dimensionality reduction techniques.

Secondly, it is possible to notice the positive contribution given by the inclusion of the trace attributes. Their number is limited, so they can bring useful information without increasing too much the complexity of the problem. As a result, I observed an increment of the accuracy larger than 10% in some cases.

On the other hand, adding all event level attributes does not lead to a large improvement of the

performances. Instead, in the case of Decision Tree models, the accuracy decreases. Even in this case, I think that the predictive power brought by all the event attributes is counterbalanced by the higher dimensionality of the problem.

Then, analyzing the feature ranking reported before, we can notice that there are some trace attributes in the first positions (diagnosis_code, age, specialism_code, diagnosis) and also some event names. In the case of the Random Forest model, the last five events have higher importance. Finally, we can also see that some time features have a relevant role, such as the day of the year, day of the week and the timestamp in human-readable format of the last events.

Finally, we can observe that for all datasets, the best model is Random Forest, followed by Decision Tree. This seems reasonable, since the former is a sort of generalization of the latter. Instead, Logistic Regression is not effective in this case, leading to the lowest accuracy results.

To sum up, in this project I have faced a prediction task, exploring the contribution of different attributes and trying different machine learning techniques. According to the results obtained, it is possible to conclude that the inclusion of time features slightly improves the predictive capacity of the models. A larger increment is given by the inclusion of other trace attributes. Moreover, among the considered classifiers, Random Forest is the best one, in terms of test accuracy. To conclude, in order to better exploit the time related variables, it may be useful to perform some feature selection, or consider other models, such as Recurrent Neural Networks, even though Deep Learning architectures may require a larger dataset in terms of number of traces.