UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS ASSIGNMENT No. 1887

TIME ENTRY PREDICTION FOR A TIME TRACKING SOFTWARE

Stjepan Petruša

Zagreb, July 2019.

Foremost, I would like to express my thanks to my mentor Tomislav Burić for continuous support and for making this thesis really exciting and interesting.

Besides my mentor, I would like to thank Tomislav Car, Jan Varljen, and Ivan Lučin from Productive for all the help understanding how Productive works. They pushed me forward to explore further and become a much better engineer than I was before. In addition, a thank you to Ivan Hribar, whose guidance helped me in the creation of figures in this paper. It was a real pleasure working with all of them.

My sincere thanks also go to my colleague and my friend Matej Janjić for all support in the last couple of years, for countless hilarious moments, excellent beer, and many great advice.

Table of content

Introduction

Tracking worked time is a crucial step in calculating the profitability of a company that works on many different projects. Examples of such companies are architecture studios, design studios, software development, and marketing agencies. To calculate the resources spent on a specific project in a time interval those companies rely on their employees to record the amount of time taken for a specific activity on a project. The process of recording that time is called time tracking.

Time tracking is tedious mostly because it required each employee to create time recordings called time entries for every working day. The whole process consists of many repetitive parts. Many designers, developers, and engineers work on the same or similar projects throughout the period of many days. Working day, in many cases, consists of activities based on a particular company, like lunch break, daily or weekly meetings. Time tracking also holds great importance. Using time entries managers can easily track and update estimates on projects, calculate the profitability of projects or give honest feedback to clients about the state the project currently is.

Problems with time tracking exist mainly on a lower level. Those are repetition, need for maintenance, and stress generation. On a higher level, time tracking provides precious information which can be of crucial help for project managers. There are many ways on how to mitigate those problems. One solution could be timers. Timers would allow employees, or in this case time trackers, an ability to start a timer when starting to work. By stopping the timer, they would not need to think about how much time went by. Problems with timers are that people often forget to stop one and start another timer. In that case, timers do not provide any help. A time tracker has to do a correction of that bad time entry which creates stress once again. The second solution would be suggestions. Suggestions work

in a way that they provide an ability to create complete time entry with one click. They function like drafts. If the user wants to create a time entry as suggested it needs to confirm suggestion. If suggestions are relevant and correct, employees may find time tracking much easier, which in the end benefits both employees and managers.

Our goal in this thesis is to create such an algorithm that will provide good and appropriate suggestions. As a source of recommendations, we can use bookings. Bookings are allocations made for each employee which show on what project that person should be working. Operations teams often create those bookings, and in theory, they have a high correlation with created time entries. For the second source of recommendations, we can use historical data to predict next time entry. If there is a common pattern between already created time entries, the algorithm should be able to find and use it.

Benefits of good time tracking experience are endless. Making time tracking process easier will lead to less stressed employees. Less stress can increase the quality of produced time entries what in the end can head to better project management and more on-point estimations. If time entries present real time spent on the project, project managers can make more less-impactful decisions and less big modifications to the project's flow. By making the project more consistent through time, engineers and designers feel less pressure which can improve their personal growth. In the end, improving time tracking experience can lead to better developers and designers, and consequently higher quality of work and better success of the company.

1 – **Time tracking software**

1.1 – **Available tools**

There are many time tracking tools currently available in the market. Even though certain agencies still use primitive tools like Microsoft Excel[1], we'll focus on professional solutions. Some of the most popular tools are Toggl[2] and Harvest[3]. They have created a great interface to allow users to track time with as little effort as possible. Some of their features include automated timers, simple and intuitive user experience and overview of historical data. Cons of those solutions are that they only do time tracking. They do it very well, but we are looking for a solution which can show greater benefits in the long term.

Many project management and reporting tools are already integrating time tracking in the core of their products. Good examples would be 10,000ft[4] and Productive[5]. Their solutions combine time tracking, resource planning, project management, and reporting. Improving their time tracking would benefit all other aspects of the tool, and not only time tracking plan like it would be with Toggl and Harvest who specialized only for time tracking. As we managed to get in contact with people at Productive, we decided that Productive would be a great fit for our project. This way we get an ability to implement our solution based on an existing software which can benefit from all positive incomes better time tracking can generate.

1.2 – **Productive**

Productive is a tool for managing sales pipeline, resource planning, project management, time tracking, and reporting. Their headline is "Productive is the only tool you need to run a profitable agency." and it provides coverage of a complete

project lifecycle which is great because it can illustrate all the benefits we mentioned before. Productive offers web, desktop, and mobile applications.

Figure 1.1 – Productive logo

Interface for time tracking module in Productive is shown on figure 1.2. It consists of a couple of elements:

1. New time entry form – consists of two modes: manual mode and timer mode. In manual mode, the user can quickly create a new time entry for the entered amount of time, while in timer mode user starts a new timer.
2. Navigation bar – contains all the days of the selected week for a swift change of the date. It also displays total time tracked for each day and cumulative time for the whole week.
3. Bookings bar – lists all bookings made for the current user and the selected date. Allows creating a new time entry or timer with time predefined in the resource planning module.
4. Time entries list – lists of created time entries for selected date grouped by project. There is an option to edit or delete each time entry.
5. Events list – displays all events scheduled in resource planning module. Examples of events are a vacation or call in sick.

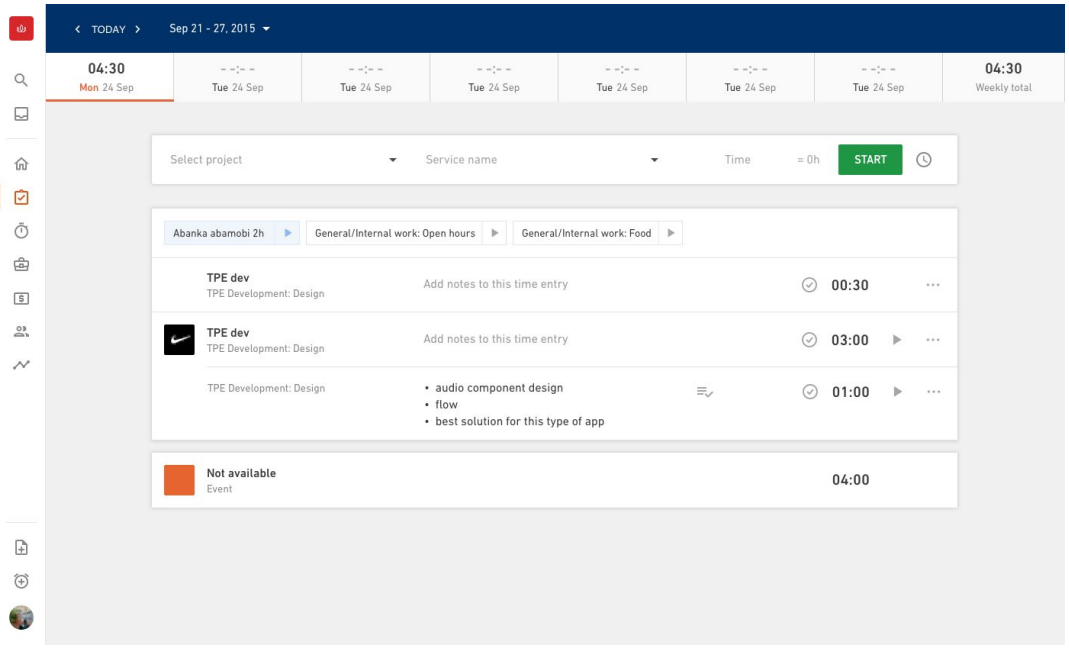

Figure 1.2 – Time tracking interface in Productive

Just looking at the interface, we can see that Productive already has a great time tracking section. Other features that are related to time tracking are automatic email reminders if a user has not tracked time for the previous day. There is also a timer widget which allows the user to create a new time entry or start a timer from anywhere in the application.

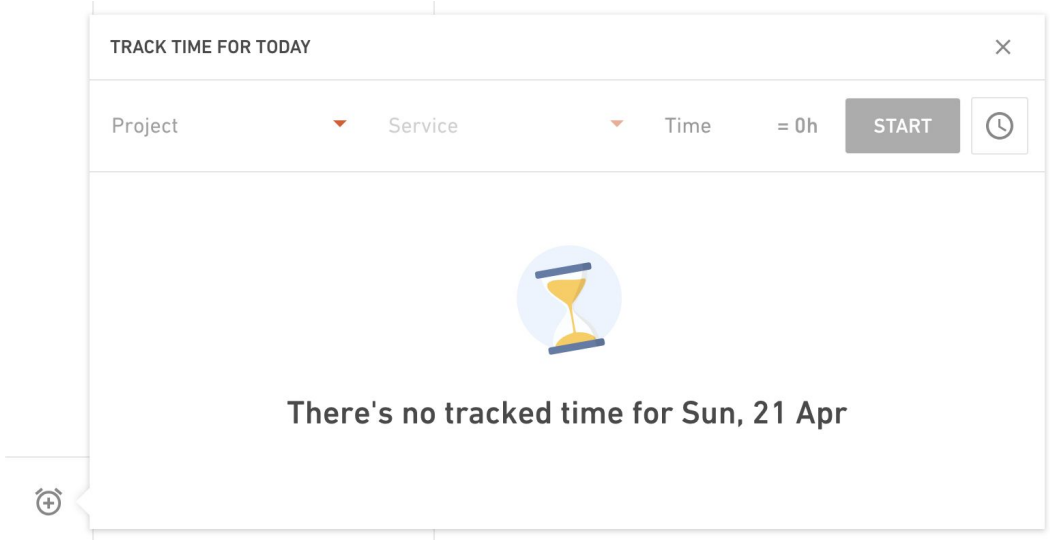

Figure 1.3 – Time tracking widget accessible from the sidebar navigation

1.3 – **Time entry structure in Productive**

Each time entry in Productive contains many fields:

1. Project/service – a reference to a service on a project. Time entry can be related to only one project/service.
2. Note – an optional short description for the time entry.
3. Time – the number of minutes spent working on the selected project.
4. Date – a date time entry associates to. Time entries can have a date in the past or the future.

In Productive time entries contain many other attributes like billable time, but they are not important for us as they do not affect our algorithm.

2 – **Architecture and infrastructure**

2.1 – **Current architecture**

Productive internal architecture consists of many small microservices which work together. For this project, we need only two of them: API service and APP service. The main purpose of the API microservice is to provide and control the data as it is the main endpoint to the database. It makes sure that all data are validated and saved. APP service provides a web application and the main interface for Productive. It serves as the main connection between users and the backend.

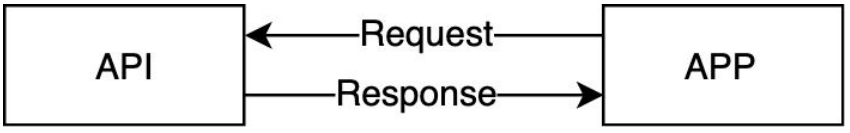

Figure 2.1 – Communication between API and APP microservices

In figure 2.1 we can see how those two microservices work together. API service provides data, and APP service displays them to the user. Communication between those two is done using HTTPS (Hypertext Transfer Protocol Secure), and the data is formatted using JSON API standard. JSON API standard works great here because we have a standardized way of how data should be formatted. All requests that are not in JSON API format are automatically rejected. As those two services work as microservices, it is logical that our solution works as a microservice too.

2.3 – **Time entry prediction microservice – Helga**

Our microservice will use API and APP microservices as providers for input and output data. For easier comparison, we will give our microservice an

accessible name: Helga. In figure 2.2 is new architecture with our microservice Helga.

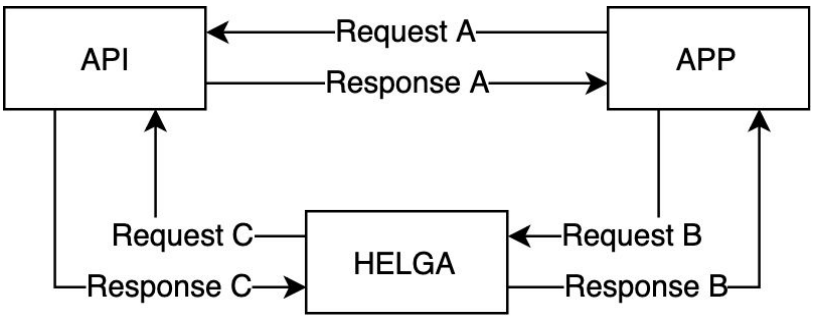

Figure 2.2 – Communication between API, APP and HELGA microservices

Helga microservice will work together with APP and API microservice in a particular way. User loads the website served from APP microservice. App then communicates with API through request A and response A displayed in image 2.2. When APP needs to display suggestions, it will make an HTTPS request to our new Helga microservice. Once Helga will have an answer it will return it to the APP. Request B and response B portrait that communication. To create prediction Helga will need historical data from API service. Using request C and response C, over HTTPS, Helga will fetch necessary data.

2.3 – **Authorization**

API service authenticates each request using secret API token saved in cookies in the user's browser. If we implement the communication between Helga service and API service as described before, we will always get an error from the API because our requests will not be authenticated. To authenticate them we need to send a secret token with the request using query parameters or cookies. To simplify implementation, we will just use cookies that we received in the request from front-end client (APP service). Of course, there are more secure ways to do authorization between two microservices. One way would be to have a secret token, or even a pair of public and private keys, just for those two microservices.

That way both API and Helga will know for sure if the request or the response is legit. Because we do not have direct access to API service, we are forced to use the token provided from the front-end application.

2.4 – **Infrastructure**

Helga microservice will work as a regular web server accepting requests through HTTPS and returning predictions in response formatted following JSON API standard. A basic web server can be implemented using many different programming languages like Java, Python, PHP or JavaScript. Predictions will be made using historical data which tells us we will have a lot of data manipulation and processing. Therefore Python is an obvious choice as it provides a clear interface for data processing. Flask[8] is a framework for Python which allows creating simple and easy to use web servers without big overhead. As our server only needs to do one thing, provide time entry suggestions, big and heavy frameworks like Django, would be overkill in this situation.

3 – **Data preprocessing**

With the arrival of new methodologies in project management, like agile development, more time is spent on planning work ahead. That way project managers can easily predict the state of the project in a couple of weeks, or even months. That also brings more meetings for all members of the project. Our algorithm could use data from a calendar application and give suggestions based on scheduled bookings. That process would require us to create integrations with many other calendar 3rd party applications. To simplify our process, we will only use time entries from the past and not bookings from a calendar application. Using this approach we can ensure our algorithm works fine even when tasks are not scheduled in advance. Integration with calendar applications could be a further improvement for this system.

3.1 – **Patterns in time tracking**

To implement a good time entry prediction system, we first need to understand the data we will use. As described in section 1.3 each time entry has a time component in the form of a date field. We can utilize that field to try and find a common pattern between time entries with the same project/service but a different date.

There could be a countless number of patterns that can occur with time entries. Even if we narrow historical data to a couple of weeks, the number of different patterns is too big. Therefore, we will focus only on common patterns and on patterns that occur within our training data. Many patterns have a different duration. Some can exist for only one week, while others can be seen through all data. Examples of those two are shown in figure 3.1. Pattern A is an example of a structure with a duration longer than the observed time period, while pattern B has a duration of only two weeks.

Figure 3.1 – Patterns with different duration

Our first hyperparameter will be the size of the window we use for looking at historical data. In case our window size is too big, the algorithm will be too slow because it will need to get and process a lot of data. If the window size is too small, the algorithm might not detect all patterns. Training data will contain time entries from a significantly longer period than our window size, just to be sure our algorithm learns all patterns in the learning phase.

3.2 – **Preprocessing flow**

In order to make extraction of patterns easier, we will transform our input data into a consistent structure. That structure will be in the form of an array containing only two values — 0 and 1. Transformation of raw data will take several steps. Figure 3.2 is an example of raw data of six entries before any alteration.

Figure 3.2 – Example time entries

3.3 – **Grouping and ordering**

First two steps in data preprocessing are:

1. Split time entries into groups based on the project and the person who created it.
2. Order time entries in each group by its date.

Splitting is important because we do not want data from one person, or from one project to mix with data from another person or project. Ordering is here to help us find duplicates and missing values in the next step.

Figure 3.3 – Grouped and ordered time entries

3.4 – **Duplicate and missing data**

To represent time entries as zeros or ones, we must first remove all duplicate entries. We use those boolean values mostly because we want to know if there was time entry for a specific day, project and person. Some people like to split large chunks of work into smaller parts and create separate time entries for each part, while others create only one larger time entry. To eliminate that problem, we remove all duplicate time entries.

Figure 3.4 – Duplicate time entries

One of the restrictions with using boolean values for representation is that we cannot have any other additional data. That is why we ordered time entries by date in step 2 because now we know that time entry later in array happened after all time entries before it. There could be a case when a person did not work on a project on a specific date, which creates gaps in our data. To remove that gaps we create dummy time entries so that we could have data for every combination of person, project, and date if there is at least one entry.

Figure 3.5 – Adding a dummy time entry

3.5 – **Sliding window**

Last two steps in data preprocessing are final data structure extraction using the sliding window technique and mapping to binary values. Extraction using the sliding window is helping us have data structures of the same length. In figure 3.6 is an example of the output of the sliding window method using two as a length of the window. As described before, the length of the sliding window is a hyperparameter and we will need to optimize it later. Conversion to binary values is necessary as it simplifies algorithm and speeds up prediction as the algorithm needs to understand two different values. Also, we do not care about other parts of the data like who created time entry or to what project/service the entry refers to.

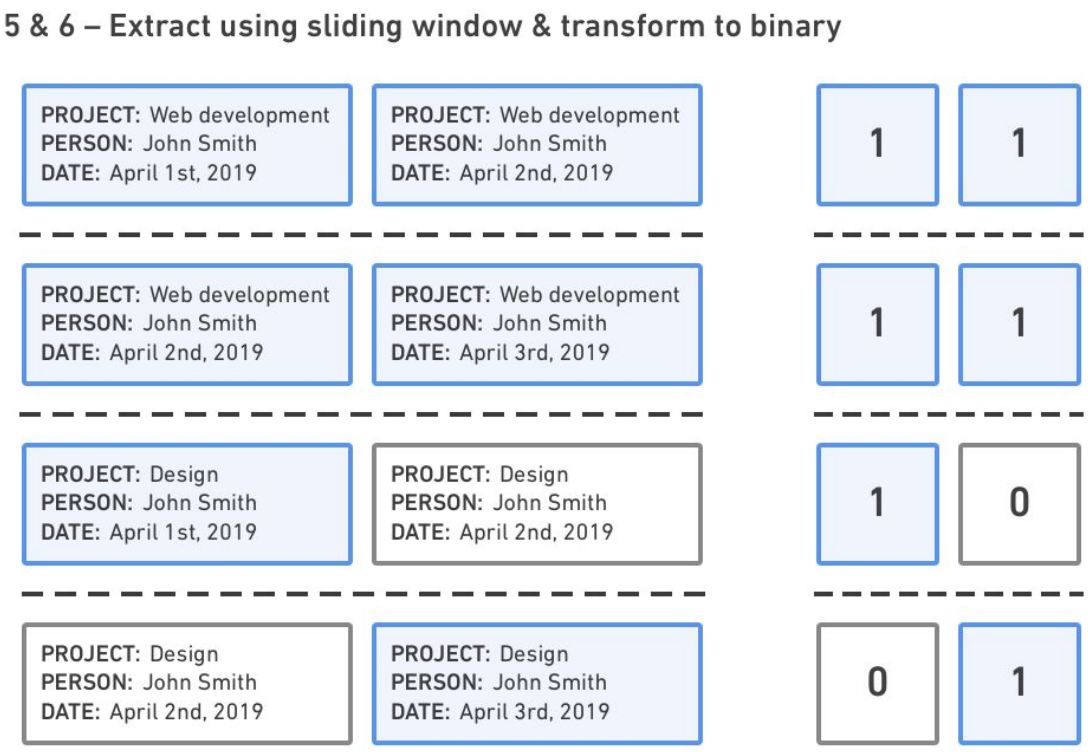

Figure 3.6 – Sliding window extraction and transformation to binary values

3.6 – **Training data**

This data preprocessing method will be applied to raw data provided by the Productive team. Granted raw data contains information about more than 50 000 time entries in a time range of one month. Raw data consists of only two values: date and group key. The date is needed for ordering step in preprocessing, while group key is used to know which time entries are connected and can be used for grouping step.

Last step, sliding window, changes drastically how much data could be used for testing. Small sliding window sizes produce a lot of training data, while long windows give fewer data. How that number of data changes depending on the sliding window size is shown in figure 3.7

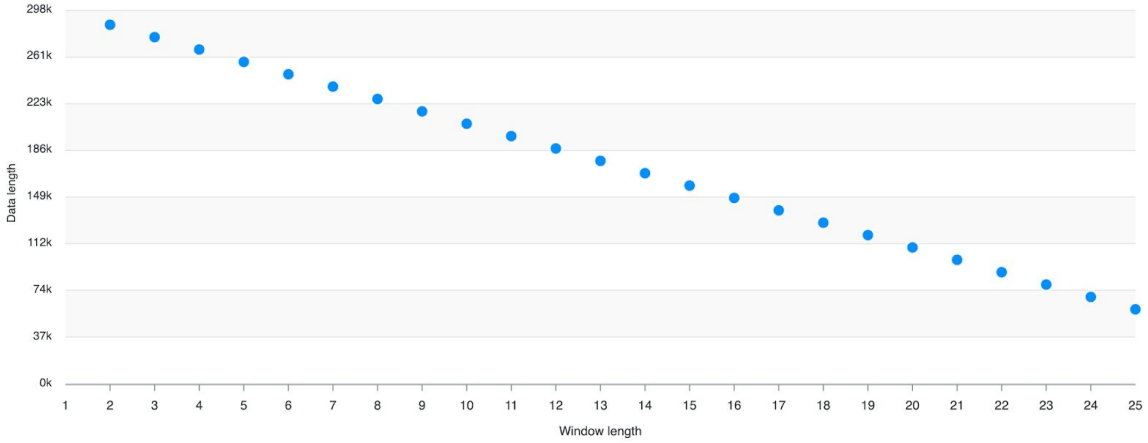

Figure 3.7 – Data length depending on the sliding window length

4 – **Algorithms**

4.1 – **Training and evaluation**

As we are dealing with an abundance of data, our algorithm will need to consist of two steps. The first step will extract common patterns from training data, and store those patterns in a structure that can be used later in the second step. The second step will use the saved structure to promptly report the final prediction. These two steps behave in diagonally different ways. The first step, or training step, is slow and does a lot of computation. It works as an optimizer for the second step. Training will be run periodically when needed, as it is not efficient to run it with every request. Evaluation step needs to return the prediction as soon as possible because we do not want users to wait for predictions every time they want to time track. It is also important that we save the state after training because we want to keep the training data even after server restart or any similar inconvenience.

4.2 – **Heuristic search**

Our first attempt to build the system for predicting time entries will be based on a heuristic search algorithm. The algorithm will try and match a given pattern with a pattern from the past. If it does not find any match it will try and find a similar pattern in the neighborhood of the original pattern. The neighborhood of the pattern is defined as a set of all patterns that differ from the original pattern by only one item. In figure 4.1 is an example of a neighborhood for a pattern with a length of 5. The training step for this algorithm is quite primitive — after data preprocessing, we just count the number of occurrences for each pattern. In the final structure, we will only save patterns with the largest number of appearances in the training data.

Figure 4.1 – Neighborhood for pattern 10011

Structure for saving patterns is quite simple. It is a hashmap where the keys are patterns and values are numbers of occurrences. Code for generating this structure in Python is in figure 4.2.

```
patterns_map = dict()
for pattern in patterns:
    key = ''.join(str(x) for x in pattern)
    if key not in patterns_map:
        patterns_map[key] = 0
    patterns_map[key] += 1
```

Figure 4.2 – Python code for generating pattern map

Evaluation algorithm gets a pattern with reduced length as an input. It has length by one smaller than window length from training data because its goal is to predict the last item in the pattern. As we are working with binary values in patterns, there could be only two possibilities — pattern ending with 1 or 0. To decide which one is more probable ending, we look in the hashmap of training patterns and see which one has a greater count number. If there are no matching patterns, we create a neighborhood for the given pattern and find a pattern with the greatest count. For a pattern of length N, there are 2*N patterns in its neighborhood. This is important to know as window length is our hyperparameter and the largest it is the more lookups we will need to do as we have more items in the neighborhood. Luckily our data is stored in a hashmap so lookup is done in constant time — *O(1)*.

As the length of the sliding window is our hyperparameter, meaning that we should optimize it before our service is ready. For measurement and comparison of different lengths, we will use accuracy. Accuracy is a ratio of good classification versus the total number of items. On figure 4.3 shows how accuracy depends on sliding window length. There are two charts, one with calculating only one neighborhood, and second, calculating neighborhood of neighborhood if needed. The first chart with only one neighborhood shows the best results for the length of 13, while the second chart for the length of 16. Difference between those accuracies is not that big, only 0,56% which can be negligible. In the training process, all patterns that occurred less than 4 times were discarded as it drastically increases memory usage, while not giving a significant amount of useful data.

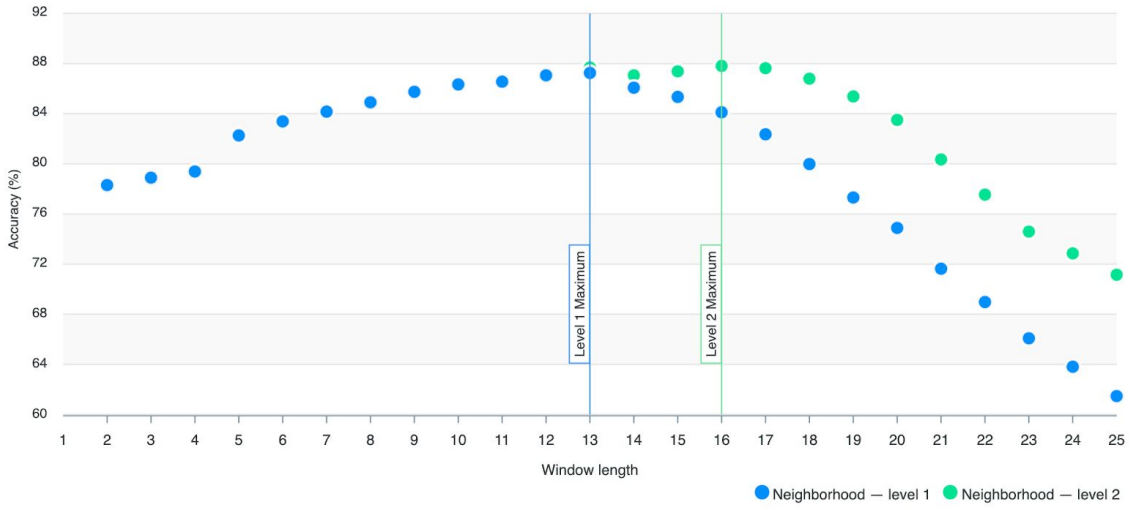

Figure 4.3 – Accuracy for different window lengths

In figure 4.4 we can see how much time is spent, on average, for evaluation of one pattern. If we do lookups only to first neighborhood complexity is linear, but if we do lookup up to the second level that complexity rises exponentially. Because of the rise of execution time, lookup to the second level is not efficient enough, as it does not justify longer execution time with better accuracy.

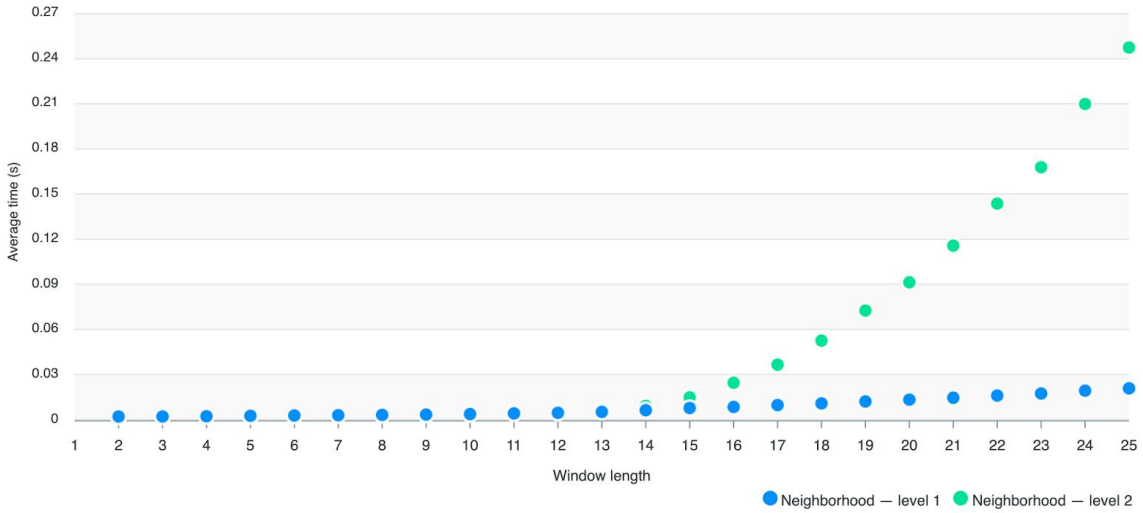

Figure 4.4 – Average execution time for different window lengths

4.3 – **Deep neural network**

Search for a solution using a heuristic was fast, but its accuracy was not great. To improve we need an algorithm that has greater capacity but does not sacrifice execution time. In industry more solutions are turning towards machine learning and deep learning. Deep learning is rising in popularity due to its great capacity to learn. With hardware improvements more operations could be done in less time, making training times lower. Deep learning is a very generative term because it could be used for image recognition, generative modeling, sequence modeling or reinforced learning. In our case, we will focus on sequence modeling, as we are working with patterns and trying to predict the next item in the sequence.

5 – **Recurrent neural network**

Recurrent neural networks are a type of neural networks with a property that outputs of a node are also inputs to the same node. Therefore RNNs contain a state that is kept and modified throughout a certain time. Today they are used extensively for language modeling. The core of such a network is a long short-term memory cell or LSTM. LSTMs are known for reaching the best metric scores among other recurrent cells. LSTMs, like other recurrent cells, has its output connected to its input. That recurrent behavior can occur a static or dynamic number of times. In our case that number will be equal to the size of our patterns or to the length of the sliding window. To optimize training and evaluation, recurrent neural networks with a static number of layers use a method called unfolding. That method unpacks one core cell into a feedforward network. Example of such unfolding is shown in figure 5.1.

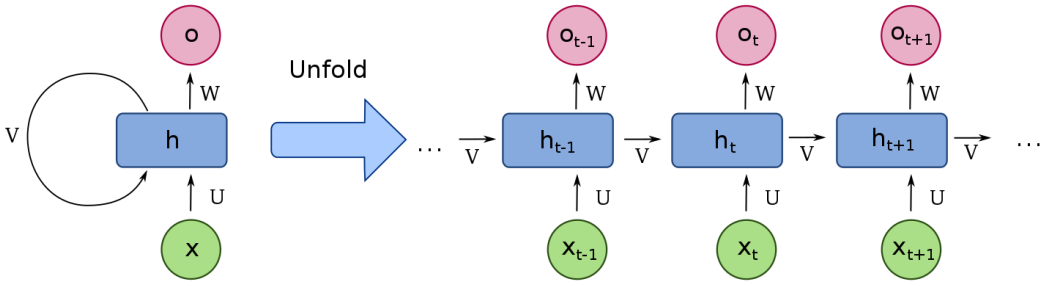

Figure 5.1 – Unfolding of recurrent neural network

All machine learning problems require training before they could be put in use. Training neural networks are pricier than using heuristic algorithms, but it can lead to greater results. Training is often done periodically or, in some cases, only once. Unlike with heuristic algorithm, training neural network is not deterministic and can yield different results. Loss function serves as a guide in training. In our case, the loss function is binary cross-entropy, and its calculation is in figure 5.2.

y_i represents real and $\widehat{y}_i$ predicted value. As the real value is binary, can only have values 0 and 1, only one part of the formula is active at the moment. If the real value for prediction is 1, the inner part of loss function reduces to $log(\widehat{y}_i)$. In case real value is 0, the inner part is $log(1-\widehat{y}_i)$.

$$L(y,\ y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\widehat{y}_i) + (1-y_i) log(1-\widehat{y}_i)$$

Figure 5.2 – Loss function for binary cross-entropy

Neural networks are harder to train than algorithms doing a heuristic search mainly because of the increased complexity and number of parameters. There are also more hyperparameters that need to be optimized before the model is ready for use. With the recurrent neural network, there are more hyperparameters that can affect the final outcome. Apart from sliding window length, new hyperparameters include learning rate, the depth of the network, size of LSTM inner memory, and others.

5.1 – **Architecture of the model**

Our neural network will consist of a single or double layer with LSTM and a dense layer just before the output. LSTM cells will have a dimension of 64 or 128 and a hyperbolic tangent or sigmoid as an activation function. The complete structure is in figure 5.3. To decide which of those settings will work the best with our data, we need to test each of those possibilities.

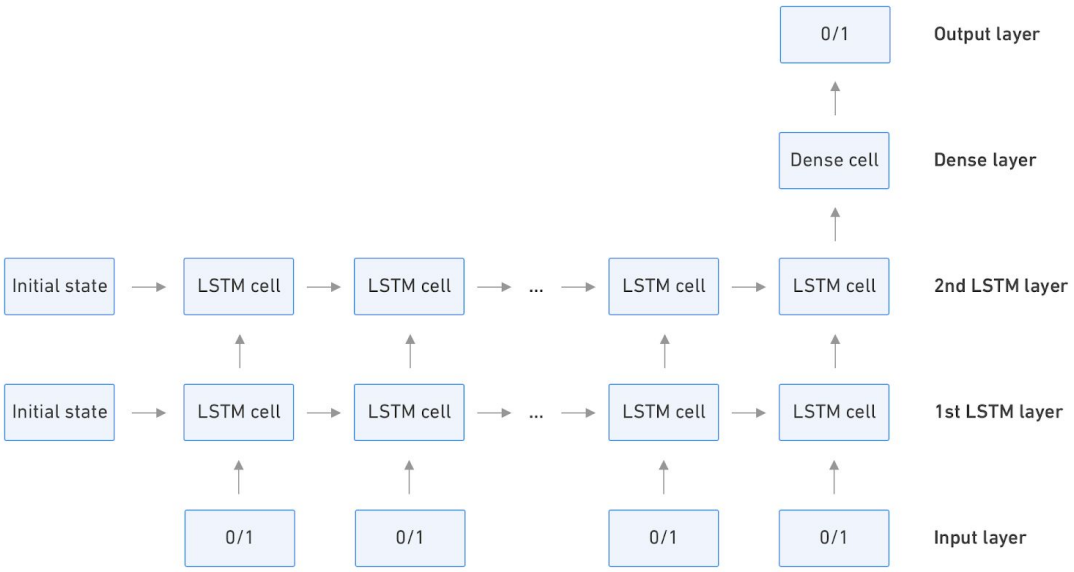

Figure 5.3 – Structure of our model for time entry prediction

5.2 – **Software solutions**

Before any testing, we need to create our neural network. To help us with that task, we will use Tensorflow. Tensorflow is a framework, which works with Python, that allows us to create various neural networks. It provides low-level support with its tensor objects and high-level support with Keras models. We will use high-level models from Keras as it eases model creation. Our structure is not that exotic that it would need any low-level support. Keras is, just like Tensorflow, a framework which contains many ready-made models and layers and we will use it to create out complete model.

```
model = Sequential()
model.add(Embedding(input_dim=2000, output_dim=1,
                    input_length=WINDOW_LENGTH-1))
model.add(LSTM(64, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop', metrics=['accuracy'])
```

Figure 5.4 – Python code for generating our neural network with Keras

Code for our model is in figure 5.2 and it first describes that our network will be sequential — data will flow from one layer to the next one. Next, there is an embedding layer which controls input data and feeds that data to the next layer, LSTM. For an LSTM layer, we need to define dimensions and activation function. After the LSTM layer is a standard dense layer which will provide us the output. The last step is to compile the model. Compiling will wrap the model with the loss function and optimizer after which the model is ready for training and evaluation. Here we will also list all the metrics we want to track, and in our model, we will track only the accuracy.

5.3 – **Optimization of hyperparameters**

The neural network, unlike our algorithm with heuristic search, has a lot more hyperparameters that can drastically change the learning rate or final accuracy. Unlike regular parameters, hyperparameters are not changed during the training. That is why we need to optimize them before the model is ready for evaluation, and eventually for production. With every new hyperparameter complexity of optimization rises exponentially mainly because we cannot assume all hyperparameters are independent. Hyperparameters that we will focus on are:

- length of the sliding window
- the dimension of LSTM memory
- a number of LSTM layers
- activation function at the output of the dense layer
- activation function for LSTM cell
- learning rate

We will not consider and test all possible combinations of these hyperparameters. To find good enough values we will test different values for hyperparameters for different values of the sliding window size. We already did optimization of the length of the sliding window in chapter 1.1 when we were working with an

algorithm with heuristic search, so we will take a similar approach. We will examine accuracy for values between 2 and 25.

We are planning to see an impact on accuracy for different values of hyperparameters. First, we will examine the impact of memory dimension for the LSTM cell. As small changes in dimension do not change a lot in the output, we will test values: 2, 4, 8, 16, 32, 64, 128 and 256.

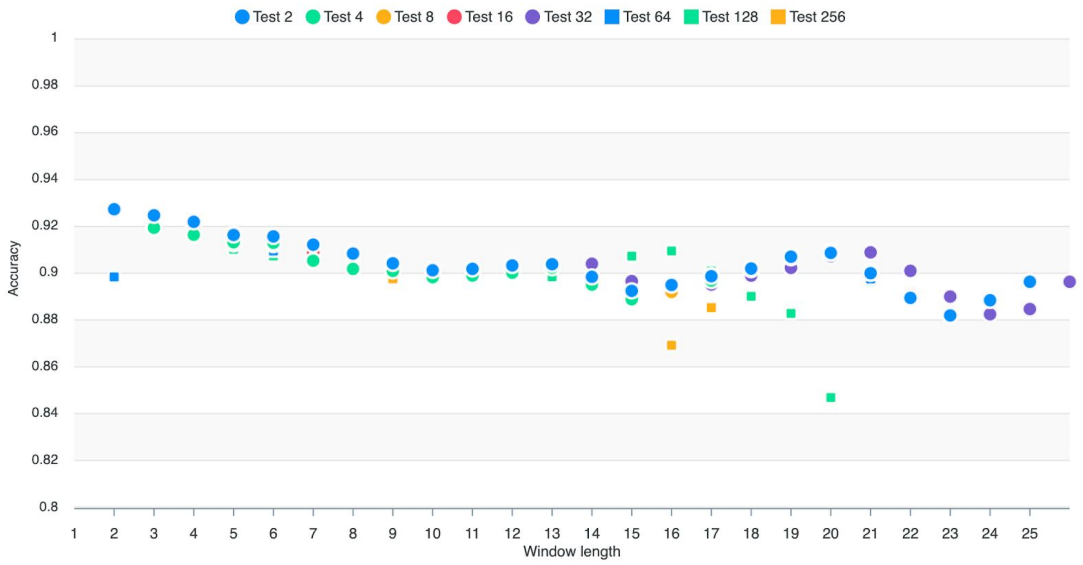

Figure 5.5 – Accuracy for different window lengths with different LSTM memory size

As it is shown in figure 5.5, LSTM memory size does not have an impact on the accuracy of the model. All values of the dimensionality for LSTM give similar values of accuracy on train dataset. Different memory size yields a different number of parameters that need to be optimized. In table 5.6 is a comparison of a number of parameters for LSTM and dense layers depending on the memory size.

LSTM dimensionality	Number of trainable parameters		
	LSTM layer	Dense layer	Total
2	32	3	35
4	96	5	101
8	320	9	329
16	1 152	17	1 169
32	4 352	33	4 385
64	16 896	65	16 961
128	66 560	129	66 689
256	264 192	257	264 449

Table 5.6 – Number of parameters depending on LSTM dimensionality

As the memory size of the LSTM does not change the accuracy, but it adds more trainable parameters, we will keep memory size of 2. This way we have fewer parameters to train, the training process is faster and we do not sacrifice any accuracy.

The next hyperparameter we would like to optimize would be the depth of the LSTM layer. LSTM layers can be stacked just like dense or any other layer. Outputs of LSTM cells in the first layer are passed as inputs to another LSTM layers, and so on until the last LSTM layer whose output is forwarded to a dense layer. In our model, we will test only two configurations: single and double LSTM layers. In figure 5.7 is accuracy scores for networks with one and two LSTM layers. Difference between those two is practically nonexistent. That lack of improvement could be because our model already has a large capacity, but the data it received is inconsistent and contains noise which interferes with the learning process. This can also point out that the model is overfitted. If we look at the accuracy for the validation dataset, we can see it only rises with small fluctuations which disprove our assumption of overfitting.

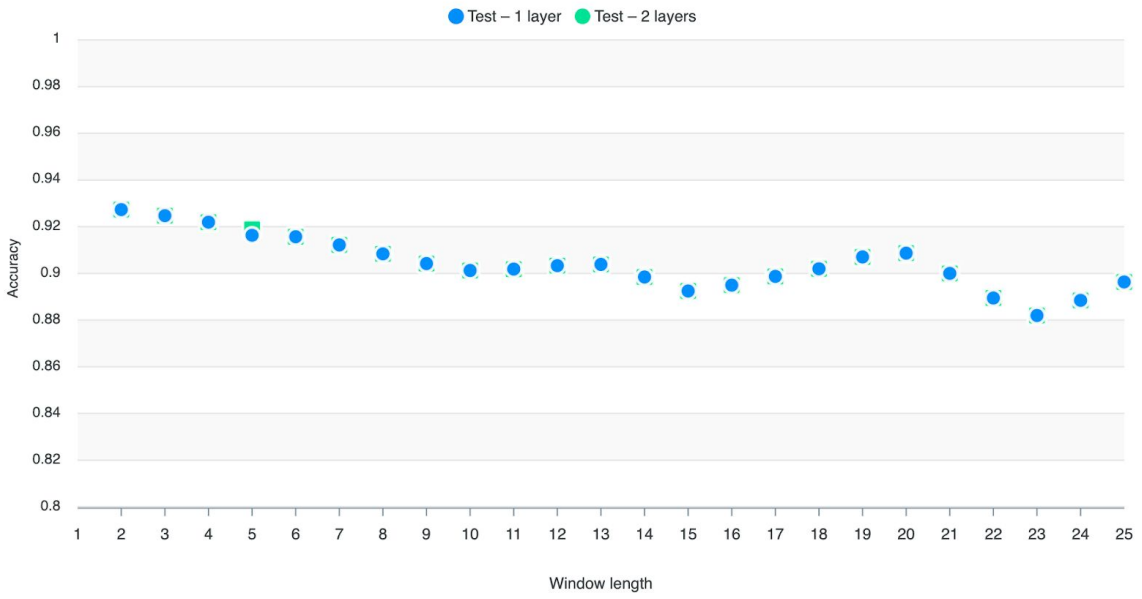

Figure 5.7 – Accuracy for different window lengths with one and two LSTM layers

Last hyperparameters for which we will test accuracies are activation functions. There are two main activation functions in our network: one for LSTM layer and one for the dense layer. Some most popular activation functions in LSTM networks are hyperbolic tangent and sigmoid. For the activation function in the dense layer, we will also try the linear function. In figure 5.8 are accuracies for different combinations of activations functions. Because we have two functions for the LSTM layer and three for the dense layer, in total we will have six combinations.

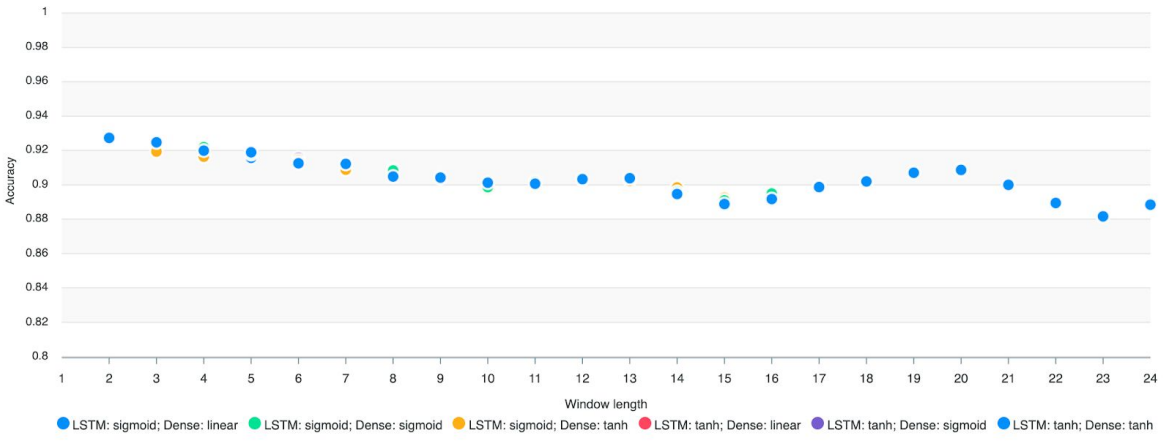

Figure 5.8 – Accuracy for different activation functions on dense and LSTM layers

Almost all accuracy scores for all six different activation functions combinations show similar results. This is something that happened before when we were optimizing depth of LSTM layer. Indifferent results show that our network has far more capacity than it is required. Achieving greater accuracy is problematic in these situations because there is a lot of noise in our training data.

5.4 – **Training and evaluation**

For hyperparameter optimization, we used the same infrastructure so we can get reliable results. It is important to make the training process independent from outer impacts, which can affect our accuracy scores. We used a virtual machine deployed on Google Cloud Platform. This virtual machine was used only to train our model using different hyperparameter combinations. This environment is well encapsulated, what we can see in some results for hyperparameter optimizations where accuracy scores were the same with different hyperparameter values.

Training in the virtual machine on GCP is done using CPU, which is not optimized for training neural networks which require a lot of matrix manipulation and multiplication. For that reason, training using GPU would be a better choice. Google Cloud Platform also allows us usage of tensor processing units or TPU for short. Those units are specially built for training neural networks as they have hardware support for matrix operations. Problem with TPUs is that they are not very good for running much training with different network configuration. Each training starts with an optimization process where the model is optimized for running on TCU. That optimization allows us running training up to ten times faster than usual, but it requires some overhead time for model optimization. TCU is very good for continuous training on same network architecture which needs to be optimized once and then trained many times. This is a perfect scenario for using our model in production: we optimize our model once we are happy with our

hyperparameters, and from that point on we keep the model constant, with multiple training which is much faster thanks to TCUs.

In our process of optimizing hyperparameters, there were two phases: training and evaluation phase. Training phase also consists of the validation phase, but we will not consider it as a separate phase for now. In figure 5.9 is shown dependency between execution time and sliding window length for training and evaluation. Execution time can also be a success metric, just like accuracy, and in our case, it is an important one. Our system required speed in making a prediction as it will need to provide results in almost no time. It is impractical and insufficient if response for time entry predictions comes with a few seconds delay. Luckily all optimizations we did suggest that small sliding window lengths give really good results, and those window lengths require less time for computation.

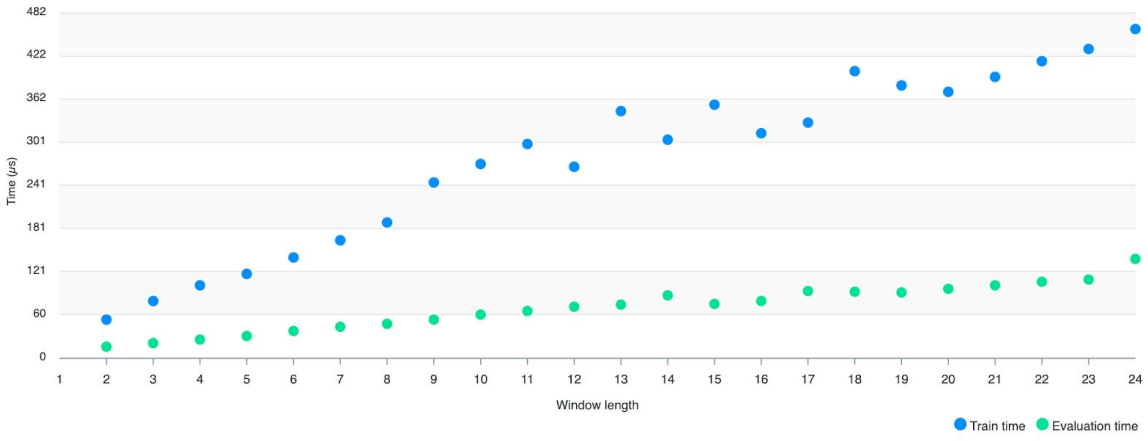

Figure 5.9 – Execution time for different sliding window lengths

6 – **Connected system**

Once we are happy with the neural network, we can start planning how all components will work together in our system. We described the architecture of our system in section 2, but here we will focus on the inner workings of our new microservice. We developed two fundamentally different algorithms and both are a good fit for our service.

The first algorithm, one using heuristic search, is fast in both training and evaluation process but does not provide quite good accuracy scores. The second algorithm, using a recurrent neural network, gives better accuracy scores but lacks the speed. In the final structure, we could use both algorithms to create a hybrid version. Another way to speed up the process is the usage of the cache. Some patterns occur more often than others and results for that patterns could be saved in the cache so they are instantly available. Cache key could be easily calculated as a string containing zeros and ones representing the pattern as shown in sections 3.5 and 4.2.

Our system will consist of 3 steps: pre-evaluation, evaluation, and post–evaluation step. Each step is responsible for one part of the data flow and generating the final response. All three steps are independent and can be easily replaced with a different and better implementation as long as input and output to and from the step we keep the same.

6.1 – **Pre-evaluation**

The first step of data flow in our complete system is pre-evaluation step and its task, once received a request, is to get required data for evaluation, parse and preprocess it, so it is ready for the evaluation step. We will cover flow in case of evaluation, as training will be done manually. Our system will listen for HTTPS

requests for a prediction. Those requests will consist of only one query parameter: date, as we want to have an ability to generate predictions for past and future to test our system. If the date is not provided, today's date will be used. Next, our system will make an additional HTTPS request to API server to receive time entries in the last couple of days. Length of how many days are needed is the same as the length of the sliding window. The response of that request will contain data about time entries. As API serves all date using JSON response structured in JSON API standard, we first need to serialize that data so we can use it. After serialization arises the preprocessing step described in section 3. Also from the original time entries, all projects are extracted, which will be used for the evaluation step. For each project group, we start a separate evaluation process to determine if we should show suggestion for that project or not.

6.2 – **Evaluation**

Input for the evaluation step is a pattern, list of zeros and ones representing the existence of the time entry for a certain day in the time range. The first action in this step is to check if the input pattern exists in the cache memory. If it exists we instantly return a cached value, otherwise, we start with the evaluation using one of our algorithms. Which algorithm to use is based on configuration and the power of the server running our microservice. A hybrid approach would be to use the recurrent neural network as long as the load on the server is not high. In case the server receives a lot of requests in a small amount of time, the system could automatically switch to using the heuristic search algorithm. This approach could be also used in the case when there is a problem with the neural network.

Another approach is to always calculate predictions using the heuristic search algorithm, but in the background start calculating prediction using neural network and its result save to the cache. This way if a pattern is common it will exist in the cache storage which has values from the neural network. If the cache does not have a given pattern, it will fallback to a fast, but not so accurate method.

6.3 – **Post-evaluation**

After evaluation steps for all projects are done, their results need to be processed and sent back as a response to the original request. Responses from evaluations are filtered and mapped so that we get only a list of projects for which response was one, and not zero. Those filtered projects are actual predictions for time entries. Just like API service, APP service, which sent an initial request for prediction, expects a response in the form of JSON and structured according to JSON API standard. To return complete time entry, except project, we also need a relationship to the person for which this time entry is bound to and date. The date is equal to the received date parameter in the initial request, and the person is the same as the currently authenticated user.

6.4 – **Caching**

As in any other server, caching plays a big role in maintaining speed. Cache serves as a bypass for slow, but deterministic processes. Those processes have to be deterministic and give the same result for the same input parameters, elsewhere it could return wrong data. That is why many different API provider services, including ours, do not want to have nondeterministic and time-sensitive data in its service. In case the same person makes a request with the same requested date and created time entries, our service should provide the same result all the times.

Scope of cache is also important. If we place it right at the start of pre-evaluation step we would cache incoming requests and responses. But, if we cache only evaluation step, then only patterns would be saved. Both scopes make sense, but with caching whole requests we would often have "cache miss" — the situation where our cache key is too specific so we rarely find a match in the cache memory and we do the evaluation quite often. To mitigate that problem we

cache only evaluation step where multiple different requests will be reduced to the same patterns.

Conclusion

In this project, we have covered a lot of different areas. Our main motivation was to improve profitability reports for agencies or design and architecture studios by improving the user experience of time tracking. User experience improvement was done by introducing suggested time entries to users when they track time. Those suggested time entries should be relevant to users so they can actually improve user experience.

We went through a couple of tools which provide time tracking and profitability monitoring for the company. Productive seemed like a great fit and also developers from Productive offered us help in the development of our system by providing training data so we could have real data for measuring the accuracy of our algorithms.

Before training or evaluating data, we needed to preprocess raw data so our algorithms could easily understand inputs. That preprocessing was done in six steps, and at the end, we got patterns in the form of a list containing binary values. The length of the list was a hyperparameter called sliding window length.

Once our data was processed and ready we could start testing different algorithms. First one was based on a heuristic search using preprocessed historical data. It was looking for a neighborhood of a given pattern and tried to find one in the training dataset. This algorithm was very fast, but it lacked accuracy and in some situations, it could not provide a result. The second algorithm was based on a recurrent neural network. We investigated how those networks work and how modifying different hyperparameters impacts accuracy in the test dataset. That algorithm provided results with better accuracy, but it had a greater number of hyperparameter to optimize. After optimization some of then we realized that even the quite simple structure is too powerful for our case. Our data had a lot of

noise which was normal in our context, but bad for accuracy scores of our algorithms.

Once algorithms were done, we tried to connect all microservices together with our microservice. Our microservice works in 3 steps: pre-evaluation, evaluation, and post-evaluation steps. Each step is independent on other steps and could easily be replaced with better implementation as long input and output structure is the same. In the evaluation step, we offered a couple of solutions on how to combine both algorithms in an efficient structure. We also investigated the impact of caching in this system.

We are quite satisfied with the final result and further improvements could be done in the pre-evaluation step where we could cache time entries from the API service, in the heuristic search algorithm where we could introduce weights to different patterns.

Sources

[1] Microsoft Excel – https://products.office.com/en/excel, April 2019

[2] Toggl – https://toggl.com/features/, April 2019

[3] Harvest – https://www.getharvest.com/, April 2019

[4] 10,000ft – https://www.10000ft.com/time-tracking, April 2019

[5] Productive – https://www.productive.io/, April 2019

[6] Productive time tracking feature – https://www.productive.io/tour/time-tracking/, April 2019

[7] JSON API specification – https://jsonapi.org/, April 2019

[8] Flask framework – http://flask.pocoo.org/, April 2019

[9] Classification: Accuracy – https://developers.google.com/machine-learning/crash-course/classification/accuracy, April 2019

[10] Barbara Črgar – February 2019 – Struggling to Get Creatives to Track Time? Here Are 4 Foolproof Tips – https://www.productive.io/blog/struggling-to-get-creatives-to-track-time-here-are-4-foolproof-tips/, April 2019

[11] MIT Introduction to Deep Learning – February 2019 – https://medium.com/tensorflow/mit-introduction-to-deep-learning-4a6f8dde1f0c?linkId=64189766, May 2019

[12] Recurrent Neural Networks – https://www.tensorflow.org/tutorials/sequences/recurrent, May 2019

[13] Christopher Olah – August 2015 – Understanding LSTM Networks – https://colah.github.io/posts/2015-08-Understanding-LSTMs/, May 2019

[14] Recurrent neural network – https://en.wikipedia.org/wiki/Recurrent_neural_network, May 2019

[15] Keras loss functions – https://keras.io/losses/, May 2019

[16] StackExchange – Why does keras binary_crossentropy loss function return wrong values? – https://stats.stackexchange.com/questions/303229/why-does-keras-binary-crossentropy-loss-function-return-wrong-values, May 2019

[17] Sam Witteveen – January 2019 – Keras on TPUs in Colab – https://medium.com/tensorflow/tf-keras-on-tpus-on-colab-674367932aa0, May 2019

[18] LSTM Binary classification – GitHub Gist – https://gist.github.com/urigoren/b7cd138903fe86ec027e715d493451b4, May 2019

[19] Caching Overview – https://aws.amazon.com/caching/

Predikcija vremenskih zapisa u alatu za praćenje vremena

Cilj ovog rada je bio dizajnirati i testirati novi mikroservis za postojeći alat za praćenje vremena koji će pružati predikcije za vremensko praćenje. Motivacija je bila poboljšanjem iskustva vremenskog praćenja ujedno i poboljšati kvalitetu vremenskih zabilješki i izvještajima o profitabilnosti. Odlučili smo koristiti alat Productive jer je kreiran s ciljem da poboljša profitabilnost tvrtke. Kreirali smo dva algoritma, jedan koji koristi heurističko pretraživanje i drugi koji koristi povratnu neuronsku mrežu. Oba algoritma imaju svoje pozitivne i negativne strane, i za najbolje rezultate hibrid oba algoritma bi se mogao koristiti.

Ključne riječi: Praćenje vremena, predikcija uzoraka, neuronska mreža, pretraživanje heuristikom, alat Productive.

Time Entry Prediction for a Time Tracking Software

The goal of this thesis was to design and test new microservice for an existing time-tracking tool which will provide suggestions for time tracking. Our motivation was that by improving time-tracking experience we could get sounder time entries and reliable profitability reports. We decided to use Productive as our time-tracking tool as it is built around making companies more profitable. We created two different algorithms, one using a heuristic search, and another with a recurrent neural network. Both algorithms had positive and negative sides, and for best performance, a hybrid method could be used.

Keywords: Time tracking, pattern prediction, neural network, heuristic search, Productive tool.