Information Technology and Management Science
ISSN 2255-9094 (online)
2021, vol. 24, pp. 8–14
https://doi.org/10.7250/itms-2021-0002
https://itms-journals.rtu.lv

1862
RIGA TECHNICAL UNIVERSITY

The Practice of Implementing ML Service into an Internet Business Application

Pavels Osipovs*
Riga Technical University, Riga, Latvia

***Abstract* – Currently, there are a large number of articles describing the theoretical aspects of development in the field of machine learning. However, the experience of their practical application in real systems is described much less often. Basically, authors describe the efficiency, accuracy, and other performance metrics of the resulting solution, but everything stops at the prototype stage. At the same time, how the trained model will behave not on test data, but in real conditions, can be very different from the indicators obtained at the development stage. This article describes the experience of the implementation and real use of a classification service based on machine learning techniques.**

***Keywords* – Machine learning, machine learning for business, REST service, text classification, WEB API.**

I. Introduction

Scientific articles on the development of new, or improvement of already existing methods of machine learning are constantly published in journals and collections of articles released as a result of conferences. However, against the backdrop of a large amount of academic research, the number of articles describing the practice of using machine learning to solve real-world problems is not so large.

This article describes the experience of using the classification model in a real working business system. The task was to create an internal service that would return a set of relevant categories based on some text description. At the beginning of the project, there was a database with a large number of texts and related to each category. As part of the task, the initial import and preparation of data for training were made, the classification model was trained, the effectiveness of several types of classification models was tested, a micro-service was created and its load testing was performed.

II. General Architecture

Python was chosen as the main programming language for the system being created. Currently, it is one of the most popular tools for creating systems in the field of machine learning [1]. The language has a large number of libraries, both directly intended for solving problems of training classification models, and for solving related problems, such as importing data, cleaning it, lemmitization [1]. The main libraries used in the project:

- *csv* – for generating files in CSV format;
- *zipfile* – for unpacking and packing archived files;
- *shutil* – for recursive work with the file system;
- *pymysql* – for working with a MySQL database;
- *pickle* – for serializing binary models to files;
- *pandas* – for convenient presentation of data from a CSV file in memory;
- *sklearn.** – for importing various classification models [2].

An interesting feature of the language is the creation of virtual environments (*virtualenv*), which allows you to isolate the software libraries used in the project from their system-wide versions of the underlying operating system – the host.

Most modern projects use version control systems both during development and for distribution. Systems such as GIT or Mercurial [3] allow you to conveniently organize the project development process among a large number of developers, provide decentralized storage of all source codes, and maintain an advanced revision history. The created system also uses a closed GIT repository to store all project files.

Modern versions of the command shell (console), in addition to just text input / output, provide various options for outputting information. For example, for operations that take a long time to complete, it is convenient to use an interactive indicator of the percentage of operations performed. It is also convenient to provide the ability to customize script parameters based on user input. For example, the following simple Python code allows the user to confirm or reject a question from the script:

```
def user_yes_no_query(question):
    '''
    CLI-based question for user input
    :param question:
    :return:
    '''
    sys.stdout.write('%s [y/n]\n' % question)
    while True:
        try:
            return strtobool(input().lower())
        except ValueError:
            sys.stdout.write('Please respond with \'y\' or \'n\'.\n')
```

* Corresponding author. E-mail: pavels.osipovs@gmail.com

Then the following code will display a request to delete the directory with old data, write it to the *delete_old_data* variable (Fig. 1):

```
delete_old_data = user_yes_no_query('Delete old generated data folder?')
```

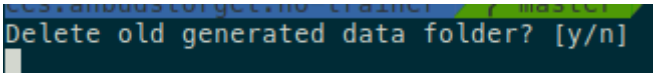

Fig. 1. Request when executing a script in the console.

To train a high-quality model, there must be a lot of consistent information in the database. In this case, about 600 000 records were used, with several categories for each of them. The relationship between texts and categories in the database is shown in Fig. 2.

Initially, texts and categories are stored in different database tables, but during the import process they are combined into one common CSV file.

The final process of obtaining a classification model from records in the database is shown in Fig. 3.

The process consists of the following steps:

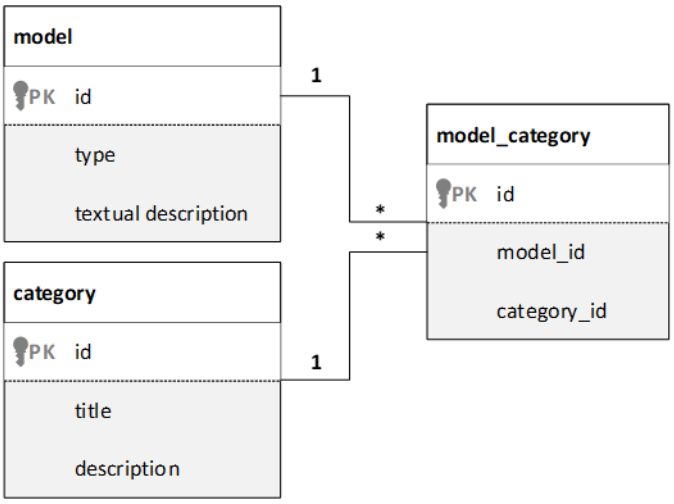

Fig. 2. Simple relation scheme in database.

1. Getting records from the database formatting and saving them as a CSV file.
2. Training the model on data from a CSV file, and serializing the resulting model into a PICKE file.

In the process of formatting the text, it is possible to clear it of unnecessary characters (numbers, punctuation marks, etc.), as well as to bring it to normal form (lemmatization). Depending on the characteristics of the texts used for training, a model trained on a normalized text can show both better and worse results relative to a model trained on a non-normalized text.

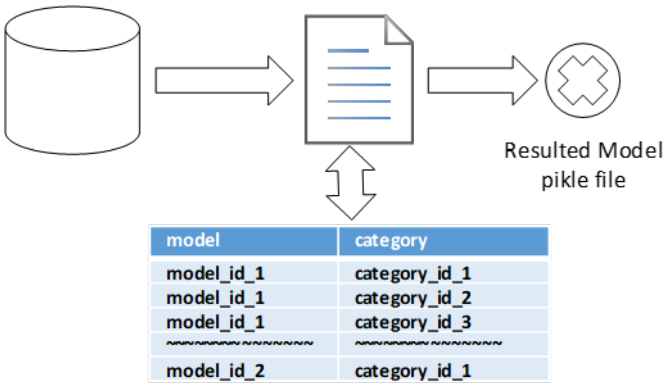

Fig. 3. Common model education process.

III. Training and Model Selection

Usually, when training a model, the base dataset is divided into two parts: *training* and *test*, then the first is used for training, and the second is used to assess the quality of the built model. In manual mode, splitting a test sample into parts is not always convenient, so the **sklearn** package has a special method for this: *train_test_split*(), which automatically splits the sample into the required parts. An example of using this method is given below:

```
from sklearn.model_selection import train_test_split
x = np.arange(1, 15).reshape(22, 2)
y = np.array([1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

The division into parts happens every time in a random order. As a result, the method returns four variables:

x_train: The training part of the first sequence (x);
x_test: The test part of the first sequence (x);
y_train: The training part of the second sequence (y);
y_test: The test part of the second sequence (y).

When creating a machine learning model, the most important issue is to evaluate the effectiveness of the trained classifier. One of the simplest and effective ways to obtain this estimate is to compare the classification accuracy of the test sample created by the model with the real categories marked in the test sample. For this purpose, it is possible to use the algorithm shown in Fig. 4, or the same algorithm in the format of code:

```
y_score = clf.predict(test_x)
n_right = 0
for i in range(len(y_score)):
    if y_score[i] == test_y[i]:
        n_right += 1
print("Accuracy: %.2f%%" % ((n_right / float(len(test_y)) * 100)))
```

Here, the *clf* variable contains the object of the classifier model under test.

As a result, after executing this code, the console will display the percentage of correctly classified records, which can be considered a measure of the accuracy of the resulting classifier.

Since the various models from the **scikit-learn** package

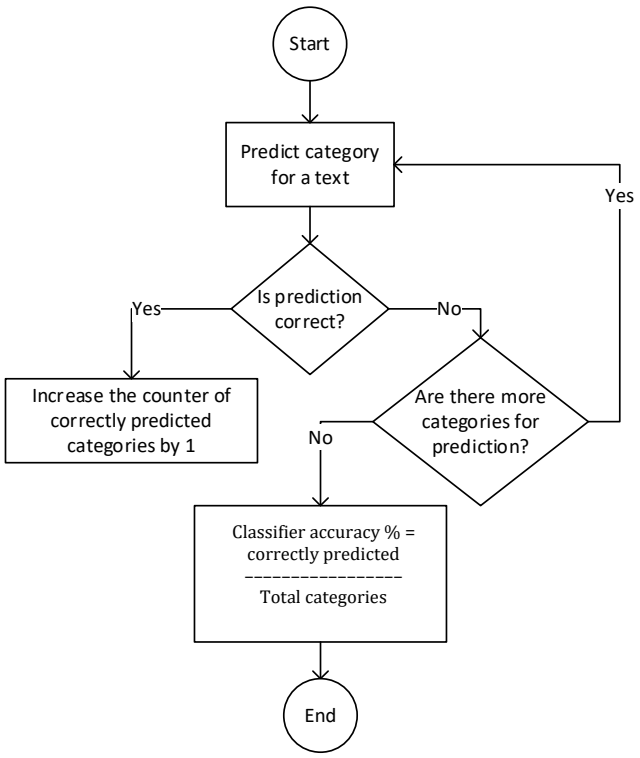

Fig. 4. Classification accuracy calculation.

inherit a common interface, they all have a *model.predict*() method, which allows you to evaluate the quality of different models using the same code.In the future, the presence of a common interface for working with different models can provide interesting opportunities for automatic selection of the best model. To do this, you only need to prepare a list of all models that are suitable for the current task + an array of possible parameters for each. Additionally, for each of the possible parameters, you need to indicate its type (nominal, discrete, ordinal, numerals, etc.). For different data types, you must also specify either a set of possible values or a range + the step size used. As a result, the program will automatically apply all algorithms to the training data, and test the effectiveness of the resulting model on a test sample. The efficiency of the parameters can be selected using genetic algorithms, then initially several random sets of possible parameter values are formed, and then this population evolves, thereby improving the quality of the used set of parameters. As a result, only the best algorithm with the best found set of parameter values will be used for classification. Such a method can require significant time and resources, but the final model can show good performance indicators, and against this background, the cost of finding it may not be significant.

The process of training the model itself is relatively simple, and consists of the following stages:

1. The required library is imported.
2. Parameters of training of the selected model are set.
3. The training code is called, into which the training and test samples are transferred.

Examples of training different models are given below.

Naive Bayes

Naïve Bayes classifiers [4] are often used for text classification because of their speed and good accuracy in some of cases.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

$$P(B) = \sum_{Y} P(B|A)P(A).$$

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(train_x, train_y)
```

Classification tree [5]

It is a powerful and popular text classification method. It shows especially good results in the case of the consistency of the training samples set. Simple decision tree structure is presented in Fig. 5.

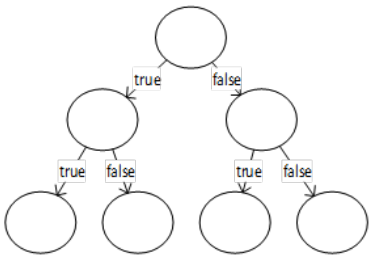

Fig. 5. Simple decision tree.

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0).fit(train_x,
train_y)
```

Logistic regression

Logistic regression [6] (Fig. 6) is one of the most common classification algorithms in the field of Natural Language Processing (NLP).

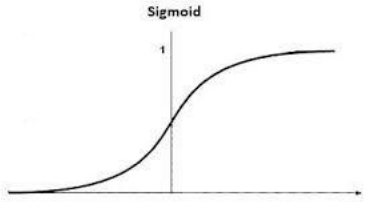

Fig. 6. Simple logistic regression function.

```
from sklearn.linear_model import LogisticRegressionCV
```

```
clf=LogisticRegressionCV(cv=5, random_state=0,
multi_class='multinomial').fit(train_x, train_y)
```

K-nearest neighbors

K-nearest neighbors [7] – it's a popular approach to the classification of texts, when the categories correspond to the text based on the closest Euclidian distance (Fig. 7).

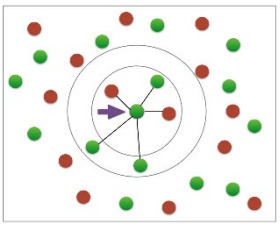

Fig. 7. K-nearest neighbors approach.

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3).fit(train_x,
train_y)
```

Linear classifier with SGD training

Stochastic Gradient Descent [8] (Fig. 8) classification sometimes shows good results [9] for texts classification.

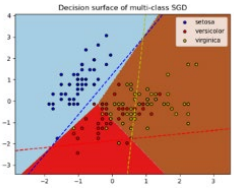

Fig. 8. SGD Linear classifier.

```
from sklearn import linear_model
clf = linear_model.SGDClassifier(max_iter=1000, tol=1e-3,
loss='log').fit(train_x, train_y)
```

SVM classifier variant

Support Vector Machines [10] (Fig. 9) – another popular approach for text classification with Machine Learning.

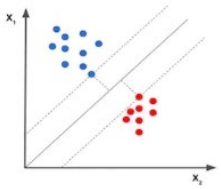

Fig. 9. SVM approach to classification.

```
from sklearn.svm import SVC
clf = SVC(kernel='linear', verbose=1).fit(train_x,
train_y)
```

One-vs-the-rest (OvR) multiclass strategy

OvR [11] is a popular heuristic method for using binary classification algorithms for the multi-class classification purposes. Common approach for OvR classification is presented in Fig. 10.

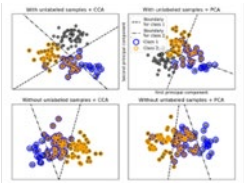

Fig. 10. OvR classification approach.

```
from sklearn.multiclass import OneVsRestClassifier
clf = OneVsRestClassifier(LogisticRegression(C=1,
dual=False, penalty='l1', solver='liblinear',
verbose=0)).fit(train_x,train_y)
```

Depending on the type of model and training settings, the time required to build the final classifier can vary greatly. With 600 000 records, some of the models take 15 minutes to train and some take hours. At the same time, it is impossible to assess the quality of the model without spending time on its training. Therefore, testing the influence of various parameters on the final quality of the classification can be quite time-consuming.

The size of the resulting PICKLE file with the model is not large for most models. As part of the work carried out, for various classifiers it varied from 0.5 to 2.5 Mb.

IV. Creating Micro-Service

After the most effective model has been obtained, the stage of creating a micro-service begins, which will allow using it within the framework of a real service. For this purpose, an approach based on a combination of Supervisor [12] + Nginx [13] = JSON API [14] technologies is used.

The structure of the created solution is shown in Fig. 11. A pool of processes managed by *Supervisor* is created in the system.

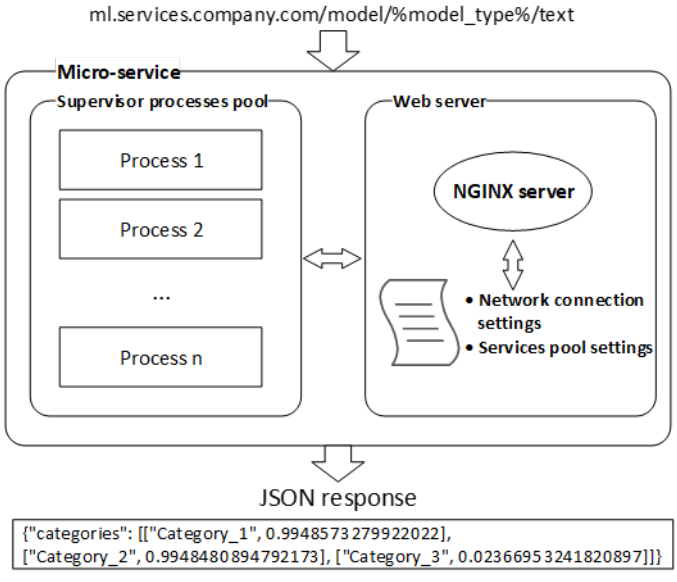

Fig. 11. Common structure of WEB server created.

Each of the processes is available for communication using its unique access port. The configuration file for *Supervisor*, which provides automatic operation of four processes, is given below:

```
[program:atml]
numprocs = 4
numprocs_start = 1
process_name = atml_%(process_num)s
logfile=/var/log/supervisor/atml.log

; Pass TCP port numbers. Path to virtualenv Python3 interpreter used

command=/home/vagrant/ml.services.company.com/ml_server/bin/python3 /home/vagrant/ml.services.company.com/ml_server/ml_server/server.py --port=808%(process_num)s

user=vagrant
autostart=true
autorestart=true
```

Here, both the number of supported processes and the settings for communication with each of them are set. Also, ports of access to each of the created processes are automatically allocated.

Next, the WEB Server Nginx is configured, which serves as an intermediary between requests from the network and *Supervisor*. Also, it balances the load between all processes available in the pool.

An example configuration file used to configure Nginx is shown below:

```
upstream atml {
    server 127.0.0.2:8081 fail_timeout=0;
    server 127.0.0.2:8082 fail_timeout=0;
    server 127.0.0.2:8083 fail_timeout=0;
    server 127.0.0.2:8084 fail_timeout=0;
}

server {
    listen 80;
    server_name atml;
    charset utf-8;
    location / {
      proxy_set_header Host $http_host;
      proxy_redirect off;
      proxy_buffering off;
      proxy_pass http://atml;
    }
    error_log  /var/log/nginx/atml-error.log error;
}
```

Here, for each of the pool of processes available in *Supervisor*, the interaction parameters are written to send the incoming request to one of the free processes, as well as return the result.

The initial configuration algorithm is shown in Fig. 12. In the form of a program code, it is shown below.

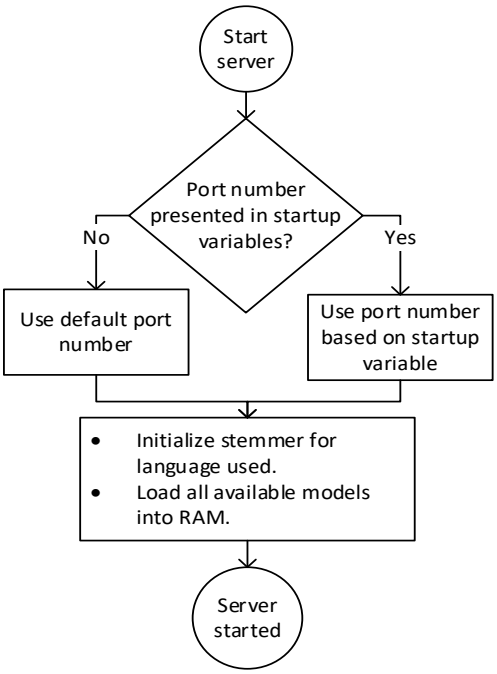

Fig. 12. ML server initialization steps.

```
#!/usr/bin/env python3

import pickle
import json
import glob
import ntpath
import sys

from aiohttp import web
import argparse

import nltk
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize

server_host = '127.0.0.2'
server_port = 8089  # Default value

# Get port assigned by supervisor
parser = argparse.ArgumentParser(description="Get port assigned by supervisor")
parser.add_argument('--port')
args = parser.parse_args()
if args.port:
    server_port = args.port
```

The very process of using the previously trained models is also described directly in the server script. The block diagram is shown in Fig. 13.

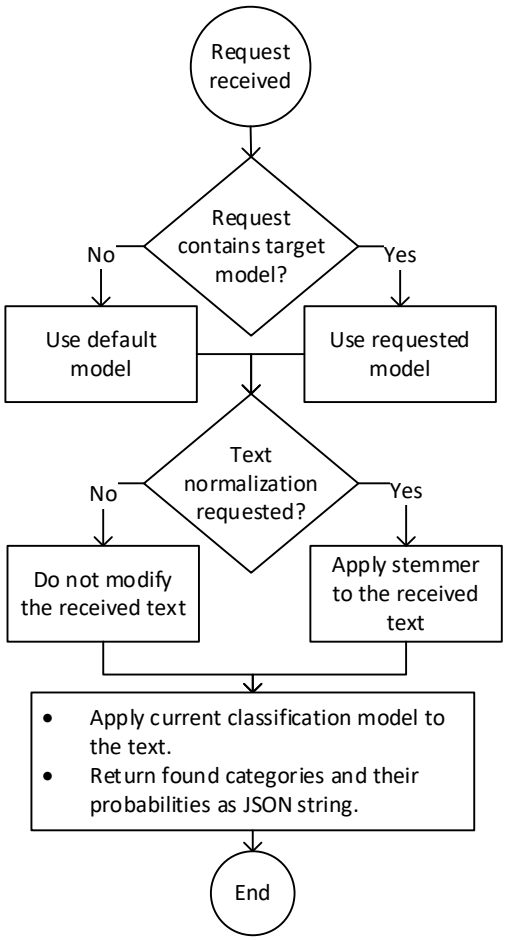

Fig. 13. Request received after model application.

The result of the script is passed back to Nginx, which then sends it to the requesting device. As a result, the created service becomes available for interaction from the internal network or the Internet. Depending on the expected load, the number of processes can be significantly increased. The speed of the server is also important. Currently, a pool of 32 processes provides classification on a public service with a response rate of 250 ms. per request. Until the number of simultaneous requests to the server is exceeded, the processing time for each request will be constant. When queues appear, the service response time will noticeably increase. Therefore, it is important to initially correctly estimate the maximum planned load and set the optimal number of processes in the pool.

Directly the response time of the service can be obtained using the standard Linux utility CURL:

```
curl -o /dev/null -s -w 'Total: %{time_total}s\n'
https://ml.services.company.com/model/?text=lorem%20ipsum
```

For more detailed testing of a service under load, the ApacheBench utility [15] is often used, which allows evaluating the response time of a service under conditions of a different number of parallel requests. The utility accepts three main parameters as input:

-n: the number of requests to send;

-t: a duration in seconds after which ab will stop sending requests;

-c: the number of concurrent requests to make.

Then an example of a request for testing a service for response speed with 100 requests in parallel 10 requests simultaneously will be implemented as follows:

```
ab -n 100 -c 10
https://ml.services.company.com/model/?text=lorem%20ipsum
```

At the end of the work, the utility displays a summary table with the results (Table I), which contains the minimum, average and longest time it took to receive a response from the service.

TABLE I

SERVICE RESPONSE TIME CHECKING RESULTS

Connection Times (ms)	min	mean	[+/– sd]	median	max
Connect	92	94	2.3	93	104
Processing	397	731	194	677	1206
Waiting	396	727	194	677	1206
Total	490	824	195	771	1307

If the service is accessible on the Internet, it is important to ensure its security. This topic is no longer related to the topic of this article, and for services working in the Supervisor + Nginx bundle, there is a detailed description of how to ensure resistance to hacking. Let us just say that it is important to pay attention to such concepts as ***requests throttling*** and ***failtoban***.

Researchers often perform an automatic analysis of vulnerabilities using ***pentest services*** [16]. This type of service makes it possible to assess the presence of typical vulnerabilities in various services. When creating a publicly available service, the programmer is not always able to provide protection against all known and unknown vulnerabilities at the time of writing the code. Some of the vulnerabilities are closed by updating both the operating system itself and the libraries used in its creation, but they cannot protect against human errors. Therefore, there are both free and paid professional tools for automatic and semi-automatic testing of services for a wide variety of vulnerabilities. Usually, when using them, the checking service generates a large number of requests to the service using various POST, GET, PUT parameters, various header keys are used, as well as direct requests to system configuration files (also log files), which may contain important data, not intended for public access. In the event of any atypical response from the service under test, the verification system signals a potentially found vulnerability.

V. Conclusion and Future Work

In conclusion, we can say that the system described in the article does not include a large number of implementation details, but at a general level it allows you to describe the structure of an ML-based application actually working in business, and this is the aim of this article.

Despite the external simplicity, the described system has been successfully working for several years. Of the problems that have arisen during this time, we can only name the need to increase the number of processes in the Supervisor pool, when, due to the activation of an advertising campaign, an unexpectedly large number of requests began to enter the system.

Also, the system does not have an automated tool for retraining the model, and when updating it, you need to manually fill in the new model and restart both *Supervisor* and *Nginx*. However, the model is updated very rarely, and there is simply no need to automatically update the model.

The overall effectiveness of the created solution is assessed by the company's management, based on the financial benefits received from its implementation. Since the new service reduced the amount of manual work and increased the profit indicator, its implementation was recognized as completely successful.

In the future, it is planned to expand both the number of various automatic classifiers and improve their quality in order to further reduce manual work. It is planned to introduce automatic methods of retraining, quality testing and updating models when the system will be constantly improved.

References

[1] J. Plisson, N. Lavrac, and D. Mladenic, "A rule based approach to word lemmatization," *Proceedings of IS04*, Ljubljana, Slovenia, vol. 3, 2004.

[2] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[3] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, "Collaboration tools for global software engineering," *IEEE software*, vol. 27, no. 2, pp. 52–55, 2010. https://doi.org/10.1109/MS.2010.39

[4] "Naive Bayes classifier for multinomial models". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html. Accessed on: Sep. 12, 2021.

[5] "Decision tree classifier". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. Accessed on: Sep. 12, 2021.

[6] "Logistic regression CV (logit, MaxEnt) classifier". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html#sklearn.linear_model.LogisticRegressionCV. Accessed on: Sep. 12, 2021.

[7] "Classifier implementing the k-nearest neighbors vote". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. Accessed on: Sep. 12, 2021.

[8] "Linear classifiers (SVM, logistic regression, etc.) with SGD training". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier. Accessed on: Sep. 12, 2021.

[9] S. Diab, "Optimizing stochastic gradient descent in text classification based on fine-tuning hyper-parameters approach. A case study on automatic classification of global terrorist attacks," *International Journal of Computer Science and Information Security (IJCSIS)*, vol. 16, no. 12, pp. 155–160, Dec. 2018.

[10] "C-Support vector classification". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. Accessed on: Sep. 12, 2021.

[11] "One-vs-the-rest (OvR) multiclass strategy". [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html. Accessed on: Sep. 12, 2021.

[12] "Supervisor: A process control system". [Online]. Available: http://supervisord.org/. Accessed on: Sep. 12, 2021.

[13] C. Nedelcu, *Nginx HTTP Server*. Packt Publishing, 2013.

[14] H. Wenhui et al., "Study on REST API test model supporting web service integration," *2017 IEEE 3rd International Conference on Big Data Security on Cloud (bigdatasecurity), IEEE International Conference on High Performance and Smart Computing (hpsc), and IEEE International Conference on Intelligent Data and Security (ids)*, Beijing, China, May 2017, pp. 133–138. https://doi.org/10.1109/BigDataSecurity.2017.35

[15] D. Rahmel, "Testing a site with ApacheBench, JMeter, and Selenium," *Advanced Joomla!*. Apress, Berkeley, CA, 2013, pp. 211–247. https://doi.org/10.1007/978-1-4302-1629-2_9

[16] L. Richard et al., "Potassium: penetration testing as a service," Proceedings *of the Sixth ACM Symposium on Cloud Computing*, Aug. 2015, pp. 30–42. https://doi.org/10.1145/2806777.2806935

Pavels Osipovs, Dr. sc. ing., is a Leading Researcher at the Institute of Information Technology, Riga Technical University. He received his Doctoral degree from Riga Technical University, Riga, Latvia.
His research interests include web data mining, machine learning and knowledge extraction. Big part of research focuses on different aspects of user behavior modelling. One more new area of interest is to explore the use of Python programming language at all steps of scientific research, from initial idea brainstorming, through all steps, to final article preparation in text format.
E-mail: pavels.osipovs@gmail.com
ORCID iD: https://orcid.org/0000-0003-4027-3997