Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Smoking Cessation Programme App User Data Analysis and Visualisation

Bachelor Thesis

Study program: Open Informatics
Field of study: Computer and Information Science
Supervisor: Ing. Jindřich Prokop

Michal Šípek

Prague 2020

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Šípek Michal** Personal ID number: **474724**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Smoking Cessation Programme App User Data Analysis and Visualisation

Bachelor’s thesis title in Czech:

Analýza a vizualizace uživatelských dat mobilní aplikace pro odvyknutí kouření

Guidelines:

1) Familiarize yourself with text data processing and classification.
2) Study the existing user data and identify the interesting data fields.
3) Define the classes of the chosen free text inputs.
4) Create a classifier for the chosen text inputs.
5) Implement useful visualizations of the data statistics.
6) Implement filtering of the statistics based on user profiles (gender, registration date, device segment, etc.)

Bibliography / sources:

[1] Qiaozhu Mei, Xuehua Shen, and ChengXiang Zhai. Automatic labeling of multinomial topic models. New York, 2007; In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD ’07), 490–499.
[2] Hingmire, Swapnil & Chougule, Sandeep & Palshikar, Girish & Chakraborti, Sutanu. Document Classification by Topic Labeling. Dublin, 2013; International Conference on Information Retrieval (SIGIR 2013), 877 – 880.
[3] Magatti, Davide & Calegari, Silvia & Ciucci, Davide & Stella, Fabio. Automatic Labeling Of Topics. Pisa, 2009; ISDA 2009 - 9th International Conference on Intelligent Systems Design and Applications.
[4] Gourru, Antoine & Velcin, Julien & Roche, Mathieu & Gravier, Christophe & Poncelet, Pascal. United We Stand: Using Multiple Strategies for Topic Labeling. Paris, 2018. In book: Natural Language Processing and Information Systems, 352-363.

Name and workplace of bachelor’s thesis supervisor:

Ing. Jindřich Prokop, Analysis and Interpretation of Biomedical Data, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Jindřich Prokop	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

DECLARATION

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date _____ _____

Signature

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Ing. Jindřich Prokop, for his guidance and tips throughout this project.

Abstrakt

Expanze na trhu chytrých telefonů umožňuje kuřákům začít odvykat kouření s pomocí jejich mobilních zařízení. Mobilní aplikace mohou uživateli pomoct projít celým procesem odvykání. Tato práce se zaměřuje na analýzu a vizualizaci uživatelských dat z jedné takové aplikace.

Uživatelská data byla rozdělena na dvě hlavní skupiny - data jako jsou demografické údaje či postup aplikací, u kterých lze hodnoty automaticky interpretovat, a nestrukturovaná textová data. První skupina dat byla následně vizualizována pomocí vlastního rozšíření Django administrace, které umožňuje autorizovaným uživatelům filtrovat a dále zpracovávat uživatelská data a volit požadovaný typ grafu pro danou vizualizaci.

Analýza nestrukturovaných textových dat se zaměřila na návrh klasifikátoru s pomocí algoritmu K-Nearest Neighbors, který by byl schopen přiřadit více označení k jednotlivým hlavním důvodům uživatelů pro odvykání. Pro tento účel byla definována sada 7 možných označení: peníze, zdraví, vztahy, těhotenství, osobní hodnoty a estetika. Klasifikátor dosáhl 84% přesnosti. Klasifikátor byl následně použit na množinu 1822 uživatelských odpovědí. Z klasifikace bylo odhadnuto, že 60,3 % uživatelů se rozhodlo skončit ze zdravotních důvodů, 37,5 % z finančních důvodů, 18,6 % kvůli jejich životním cílům a/nebo osobním hodnotám, 15,6 % kvůli rodině či přátelům, 10,3 % kvůli dopadu na jejich psychiku, 1,3 % kvůli těhotenství a 0,4 % z estetických důvodů.

Klíčová slova: odvyknutí kouření, vizualizace dat, analýza textu, multi-label klasifikace textu

Překlad názvu: Analýza a vizualizace uživatelských dat mobilní aplikace pro odvyknutí kouření

Abstract

With the expansion of the smartphone market, smokers have the ability to enroll in a virtual smoking cessation programme on their phones. Such mobile applications can help users with the whole smoking cessation process. This thesis focuses on analyzing and visualizing user data from one such application.

User data was split into two main groups - data such as demographics and application progress, for which the values can be automatically interpreted, and free text input. The first group was then visualized through a custom extension of Django's administration interface that gives authorized users the ability to filter and process user data, as well as select the desired chart type.

Free text input analysis focused on designing a K-Nearest Neighbors classifier that would assign multiple labels to responses users entered as their main reasons for quitting. A set of 7 possible labels was defined: money, health, relationships, pregnancy, personal values, and aesthetics. The classifier was able to achieve 84% accuracy. Using this classifier on a dataset of 1822 user responses, it was estimated that 60.3% of users were quitting for health-related reasons, 37.5% for financial reasons, 18.6% because of their life goals and/or personal values, 15.6% because of their family/friends, 10.3% for their state of mind, 1.3% because of pregnancy, and 0.4% for aesthetic reasons.

Keywords: smoking cessation, data visualization, free text input analysis, multi-label text classification

List of Figures

List of Tables

Contents

1 Introduction

Tobacco use is currently one of the biggest public health threats according to the World Health Organization, killing over 8 million people globally each year [1]. To raise awareness of its negative effects on health, countries are implementing warnings printed on tobacco products, usually in the form of pictures.

While being aware of the health risks could reduce the number of new smokers, it may not be as effective for active smokers who are struggling with an addiction. The technological expansion on the smartphone market, however, introduces a new way of dealing with such addiction. People can simply download an application and enroll in a virtual smoking cessation programme. One of the advantages of this solution is that the application can provide help whenever the user needs.

This thesis aims to analyze and visualize data from one of these mobile applications for smoking cessation.

1.1 The Application for Smoking Cessation

The mobile application for smoking cessation analyzed in this thesis is available for both Android and iOS devices. It provides users with a two-month interactive therapy. During each day of the therapy, the user goes through a dialog with a virtual therapist. This serves as a way for them to personalize the whole experience, as well as reflect on their achievements throughout the therapy.

The whole therapy is split into the following main phases:

1. **The EE Phase** is a preparation phase with the main focus being to engage the user, elicit change talk and evoke motivation to make positive changes through motivational interviewing.

2. **The EQ Phase** is a quitting phase that starts usually 10 days after starting the first session of the EE Phase.

3. **The FU Phase** is a follow-up phase that starts once the user confirms quit attempts. It is also the final phase of the treatment.

Apart from the aforementioned phases, there is also **the SUR Phase** (consisting of questionnaires at 6 weeks and 6 months after quitting) and **the FIN Phase** (signifying that the user's contact information can be deleted and their data anonymized).

1.2 Terminology

In this thesis, the reader may encounter words that could seem ambiguous or unclear. To avoid possible confusion, here is a list of definitions that may be specific to the application:

- **PHASE** describes a specific portion of the cessation programme with a specific goal, such as the EE Phase with the goal of preparing the user for quitting smoking.

- **SESSION** represents a collection of pages, conditions, and actions that is meant to be viewed/interacted with during a single day and is a building block for phases. These include sessions such as EE01 (the first session of the EE phase) and EE07 (the session where users define what common situations pose a high risk of inducing smoking cravings).

- **PAGE** is a viewable content unit used for building sessions. It can contain texts, input fields, images, etc. An example of a page would be EE01.1 which is the first page of the EE01 session where the user is greeted and asked their name.
- **VARIABLE** is, in the context of user data, synonymous with a data field. It represents a specific property of a user, such as their age, current phase and phase step, responses to various questions, etc.

2 User Data Analysis

Throughout the smoking cessation programme, various types of user data are being collected, including, but not limited to, demographics, application progress, and smoking habits and aspects. Due to this diversity, however, it was necessary to perform an analysis of a large enough subset of data fields in order to build a general framework for data visualizations, as there may exist limitations to specific types of data.

2.1 Categorization of Selected Data Fields

To identify potential limitations and/or specific requirements, it was necessary to focus on how the data is processed when being visualized. First, the data has to be retrieved. Next, there needs to be the ability to filter entries to avoid having to manually process data of all users. Then, it must be possible to aggregate the entries based on common attributes. Finally, it is useful to have the option to post-process the results.

Given this pipeline, only the filtering step does not perform any transformation of the original data, it only removes entries. Both aggregation and post-processing imply transformation and thus need to be examined further. It could be argued that even the fetching step does not transform any data. Upon further examination, however, use cases can be found in which it can be beneficial to allow the fetching step to transform the contents of a data field (see 2.1.4).

The analysis identified the following traits that have impact on the aforementioned steps: **data type**, **dimensionality**, **meaning**, **parameters**.

2.1.1 Data Type

Above all, the data representation, or data type, has to be taken into consideration - either *numerical* or *textual*. The reason for this distinction stems from the various ways the data can be aggregated. Given a set of users' ages, for instance, it might be of interest to find the minimum, maximum, average, and other statistical properties of the dataset to better understand the customer base.

On the other hand, given the set of users' names, the same aggregation may not performed. In case of finding the minimum and maximum of the set, it could be based on lexicographical order but such results are most likely of no use.

Finally, not all data fields must always have a value. This is most noticeable on fields such as user's list of high risk situations which is defined several days into the cessation programme. On the subject of lists, it should be said that a user entry with multiple values returned by a single field, i.e. a list of values, can be thought of as having multiple entries of the same user with the individual values of that field in each entry. Also, if a data field should contain an object (i.e. a collection of any of the previously mentioned data types) it should be split into its components.

2.1.2 Dimensionality

In most cases, a user's data field contains a single value. When a list of values is encountered, it may be split into individual values as described earlier. However, this is only possible if no connection among the values exists, i.e. they are independent of each other, and thus no information is lost by splitting them.

On the other hand, there may be cases in which there is a connection among the values and they must be treated as a single entity, an n-tuple, rather than a collection. One example could be tracking progress of users that registered on a specific date. This would imply a *2*-tuple of registration date and current step that loses its original meaning once split.

To generalize the idea, an n-tuple for any natural value of n should be anticipated as input for the aggregation step.

2.1.3 Meaning

The next aspect is purely code-related and stems from the fact that in most use cases, it is beneficial to avoid repetition by introducing symbolic values in place of the real, full values.

One example of this is user's sex. It can be represented by a single number instead of a string of characters. Not only is this more memory efficient but it also allows for easier internationalization of the application, since a single symbolic value can represent multiple versions of the same text in different languages.

Since the symbolic values are not easy to interpret by a human, it is required to add value translation from symbolic to explicit values as part of the post-processing step where applicable.

2.1.4 Parameters

When trying to understand how a user interacts with the application, it is useful to, for instance, see how much time they spend going through a specific session, or on a single page. This insight might be useful for optimizing content length of individual pages since it is possible that a user might skip reading a page if it seems too long.

Given that there are many individual pages and sessions, it would not be practical to create a separate data field for each and every page and session. This problem, however, can be solved if parameters are allowed to be passed to data fields. That way, there is only one entry point for retrieving similar data which also helps with code readability.

2.2 Data Fields Selected for Visualization

While the aim of this thesis was to develop a general framework for visualizing any data field available, only a small portion was selected to be the default set of fields. Those default fields would be pre-configured and thus ready-to-use.

Firstly, there are fields that are almost purely business-oriented and can be used to measure the application's success. The following fields have been selected for this purpose:

- **In-app rating** [CZ: *Hodnocení*]: a star-based rating system available to the user after using the app for a certain period of time (from 0 to 5 stars),
- **Registration date** [CZ: *Datum registrace*]: a timestamp representing the exact moment a user joined the cessation programme,
- **EE step given registration date** [CZ: *Dosažený krok EE od registrace*]: a custom metric for visualizing users' progress with respect to the date on which they joined the cessation programme,

- **Paid or ended in trial** [CZ: *Zaplatili vs. skončili v trial*]: a binary indicator of whether the user decided to stop using the app after trial period or paid for the full version.

The next set of fields aims to better understand users' progress throughout the cessation programme:

- **Phase** [CZ: *Fáze*]: one of predefined values representing their current phase of the cessation programme,
- **EE step** [CZ: *Krok EE fáze*]: a numeric value corresponding to user's current progress in the EE Phase (empty if not reached yet or already finished),
- **Relapse** [CZ: *Počet kouření v FU*]: the number of times a user has reported smoking in the FU phase,
- **Session time** [CZ: *Čas na sezení*]: the amount of time a user has spent in a given session.

The previously mentioned *EE step given registration date* would also fall into this category but its original purpose, when deciding on what fields to visualize, was to see the effectiveness of promotional campaigns.

The final set of data fields provides a better insight into the users' personal situation, their addiction, and their goals. For all of the following fields, the user is asked to select a subset of a predefined set of possible values:

- **Desires** [CZ: *Touhy / Cíle*]: goals the user desires to achieve by quitting smoking (e.g. save money, achieve inner peace of mind, better one's physical performance),
- **Topics** [CZ: *Témata k probrání*]: topics the user wishes to learn about during the course of the cessation programme (e.g. How should I stop smoking without gaining weight? How will I cope with stress without cigarettes?),
- **High-risk situations** [CZ: *Rizikové situace*]: situations that pose a high risk of smoking (e.g. after a meal, on vacation, when feeling sad),
- **Strategies** [CZ: *Strategie*]: actions that the user chooses to take when having cigarette cravings (use nicotine substitutes, wait, get support from their friends, family, etc.),
- **Values** [CZ: *Hodnoty*]: personality traits, life views, objects, etc. that the user values (e.g. courage, family, enjoying life).

3 Free Text Input Analysis

As a part of the cessation programme, the users are asked to answer questions where selecting from a list of predefined answers may not be the most appropriate solution. This might be because too many options would be required to accommodate the whole spectrum of users and the overall experience might seem less personalized. Allowing for free text input solves this issue but, at the same time, makes it more difficult to analyze users' responses automatically, as it requires implementing natural language processing capabilities.

3.1 Data Fields

All of the available free text inputs can be split into two basic groups: *text-related* and *smoking-related*. The *text-related* fields are of no use in the context of this thesis, as they include data such as user's name declension (namely the first and the fifth case). From the *smoking-related* group, the following three free text input fields were identified as potentially interesting for visualizations:

- main reason for quitting (code name *ReaQui*),
- smoking urge plan (code name *SmoUrPlan*),
- high-risk situation plan (code name *HRSitPlan*).

However, it should be noted that those are not the only free text input fields available.

Each of these data fields is introduced during a different step of *the EE Phase* - in the same order as they are listed here. Since any two steps are at least one day apart and users may stop using the application after a certain amount of time, the later the field is introduced, the higher the chance of it being empty. The following table shows the absolute frequency of each data field as well as the response rate given the number of users that have started the appropriate phase step:

Code Name	**Step**	**Number of users**	**Number of responses**	**Response rate**
ReaQui	EE02	1956	1877	95.96%
SmoUrPlan	EE04	1419	1302	91.75%
HRSitPlan	EE07	594	569	95.79%

Table 1: Overview of free text input fields answer frequencies (April 2020).

Given the topic and structure diversity among all three of these fields and the relatively low number of responses, only the first field (*ReaQui*) was selected for visualization. Also, understanding the main reasons for quitting provides an insight that could be used for better targeting new users, e.g. through specially tailored ads.

3.2 Classification Labels

The goal of the classifier is to assign appropriate labels to users' answers. While the description of the *ReaQui* field states it should be the main reason, we should not assume the users would only enter a single value. Moreover, even a single reason could be assigned more than one label. Due to this fact, it should be treated as a multi-label classification problem.

In order to assign labels to strings of text, a set of all available/expected labels has to be defined. In a 2013 study on smokers' reasons to quit [2], the reseachers identified the following groups of reasons: *present health*, *future health*, *pregnancy or child birth*, *imposition by partner/family*, *recommendation by physician*, *economic cost of cigarettes*, *other reasons*. Based on their results and a preliminary analysis of users' responses in the application, the following set of labels was devised:

- **Money**: cigarette prices, one's income, plans to save money, financial situation, etc.
- **Health**: illnesses, diseases, physical performance issues, etc.
- **Relationships**: family, partner, friends, colleagues, etc.
- **State of mind**: expression of feelings, description of mental health, etc.
- **Pregnancy**: state of pregnancy, childbirth, newborn baby, etc.
- **Values**: life values and goals, life style, one's philosophy and desires, etc.
- **Aesthetics**: cigarette smoke odor, wrinkles, yellow teeth, etc.

3.3 User Response Analysis

Before selecting a classification method, it might be beneficial to first analyze at least a subset of the currently available responses. The following list summarizes points of interest that resulted from the preliminary data analysis (*NOTE: approximate values*):

- average response length is 41 characters,
- average word count per response is 8 words,
- most frequent word count per response is 3,
- 64% of answers contain accented / local characters (such as í, ý),
- 3% of answers contain duplicate characters in a sequence.

It should be noted that these statistics were performed on a slightly pre-processed dataset to avoid negative effects of punctuation and whitespaces on letter/word counts. Since most of the userbase, at the time of writing this thesis, is from the Czech Republic and Slovakia, only Czech/Slovak answers shall be considered for the analysis further into this text. This is important when performing vectorization of the sentences.

Based on these statistics, we can assume that the answers are rather short in terms of word count and may or may not contain local characters. Also, the probability of a sequential character repetition occurring in user's response (such as typing '*moooc*' instead of '*moc*' - EN: *very much*) is in the context of this thesis almost negligible and, in specific cases, it might actually be the correct spelling of a word. Thus the real percentage may be even smaller. This might otherwise affect potential vocabulary look-up if required. Another aspect, however, that could have the same negative effect as character repetition is grammar - both in terms of grammatical errors and usage of prefixes/postfixes that a pre-trained vocabulary may not include.

3.4 Vectorization

Vectorization is an important step when processing text data. It is the process of transforming a string of characters (in this case) into a vector of numbers. There are various ways of achieving this goal.

One possible approach is to prepare a vocabulary where each word corresponds to a dimension of the resulting vector. This way, a sentence vector is simply a sum of word vectors where each component signifies the corresponding word's frequency. That, however, affects the vector's magnitude which may not be desirable, e.g. having the same word 5 times in a row might have the same meaning as having it in a sentence just once. In such cases, each vector component can be limited to being only 0 or 1.

On the other hand, each word need not be represented by a single binary indicator. A popular approach is to represent each word in a distributed manner [3]. This way, words similar in meaning tend to be close to each other. On top of that, it can be potentially improved by representing each word as a set of n-grams, i.e. subwords of fixed length. This might be especially useful for languages such as Czech where declension is common and extensive. For that reason, the FastText[1] library and its pre-trained model[2] for the Czech language shall be used for the purpose of this thesis to minimize the effect of incorrect spelling and/or grammatical errors that can occur when typing on a mobile device.

3.5 Classification Method

There are various vector-based classification methods, including Support Vector Machines, K-means, and K-Nearest Neighbors. When selecting the appropriate method, it should be taken into consideration that the currently available dataset is rather small (fewer than 2000 samples) and it requires the ability to assign multiple labels at once.

In their paper, Gulisong and Kouzani compared various multi-label classification methods [4]. Based on their findings, the K-Nearest Neighbors algorithm was selected for the purpose of this classification task, as it is able to perform well even on smaller datasets.

The main logic of the K-Nearest Neighbors algorithm in general is to first save a set of labelled vectors. To classify a new vector, the algorithm finds a set of K closest vectors from the training dataset with respect to a predefined metric. Based on their labels, the classifier then has to decide which of these labels to assign to the new vector. Two different approaches of assigning labels shall be compared in this thesis - *naive* and *statistical*. The *naive approach* disregards any additional knowledge about the training dataset while the *statistical approach* uses maximum a posteriori principal to estimate label probability based on the dataset.

Let us consider the following notation:

- $k \in \mathbb{N}$ ... number of nearest neighbors to be evaluated,
- $\boldsymbol{k} \in \{0, 1, \ldots, m\}^k$ ... set of indices (rows) of the k nearest neighbors,
- $m \in \mathbb{N}$ ... number of known samples,
- $n \in \mathbb{N}$ ... number of known labels,
- $d \in \mathbb{N}$ ... sentence vector dimensionality,

[1] `https://fasttext.cc/`
[2] `https://fasttext.cc/docs/en/crawl-vectors.html`

- $X \in \mathbb{R}^{m \times d}$ ... training set of m sentence vectors,
- $Y \in \{0,1\}^{m \times n}$ ... training set of m label assignments,
- $\boldsymbol{y} \in \{0,1\}^{n}$ ... label assignment vector,
- $\epsilon \in [0.5, 1)$ ... activation threshold for the *naive approach*,
- H_0^i ... event of tested vector having i-th label not assigned,
- H_1^i ... event of tested vector having i-th label assigned,
- E_c^i ... event of i-th label occurring exactly c times among the tested vectors's nearest neighbors,

where 0 denotes *not assigned* and 1 denotes *assigned* in terms of label assignment.

The naive approach splits the classification problem into individual, independent label evaluations. In order to assign a specific label to a vector, its relative frequency among the nearest neighbors must be greater than or equal to a pre-defined threshold. The threshold can be then subject to optimization on a given dataset to minimize the number of false positives and false negatives. This rule can be re-written as follows:

$$y_i = \begin{cases} 0 & \text{if } \frac{1}{k} \sum_{j \in \boldsymbol{k}} Y_{j,i} < \epsilon \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

where y_i is the i-th component of the label assignment vector $\boldsymbol{y}$ and $Y_{j,i}$ is the i-th label assignment indicator of the j-th training sample.

The statistical approach, like the naive approach, splits the classification into individual label evaluation. Instead of relying only on the nearest neighbors, however, it classifies based on the conditional probability of specific label being assigned given its frequency among the neighbors and tries to maximize it [5]. Using the Bayesian rule, it can be re-written as:

$$y_i = \underset{b \in \{0,1\}}{\operatorname{argmax}} P(H_b^i) P(E_c^i | H_b^i), \tag{2}$$

where y_i is the i-th component of the label assignment vector $\boldsymbol{y}$ and:

$$c = \sum_{j \in \boldsymbol{k}} Y_{j,i}$$

4 Solution Architecture

All of the users' data, as well as the content users can see in the mobile application, is handled by a single website. Authorized users can then view stored data and update the content through the website's administration interface. Given this solution, it was decided that the visualizations should be implemented as a part of the administration interface.

4.1 Requirements

In order to build such extension, a set of functional requirements had to be collected. The following list contains all of the main functional requirements:

F1. The solution supports visualization of the selected user data fields (see 2.2).

F2. Visualized user data can be filtered based on user profile (sex, group, registration date, current phase, etc.).

F3. Visualized data can be exported to CSV.

F4. Multiple user data visualizations can be filtered at the same time (global filters).

F5. Visualization supports multiple chart types (table, pie chart, column chart, line chart).

F6. The application supports visualization of statistics from Google's Play Console and Apple's App Store.

F7. The application supports visualizing selected KPIs.

F8. Data filtering supports regular expressions.

4.2 Backend

The project is built on top of *Django* which is a Python web framework for rapid development[3]. It organizes the whole project into smaller packages called *applications* - the solution developed as a part of this thesis is located in the *stats* application. Such application can access the database through *models*, i.e. special classes directly mapped to a table. It can also map URLs to specific pages (called *views*). In this solution, all of the views are very similar to each other. They are almost empty HTML templates that get populated with data using JavaScript and therefore will not be described further.

The core functionality required on the backend is an API that can fetch, filter, aggregate, and post-process data which the user wants to visualize. To achieve this, a URL endpoint accessible to authorized users, which performs the aforementioned list of actions and returns the desired data, had to be created.

With the addition of **F6** and **F7**, it became necessary to add custom models to store and represent reports from both Play Console and App Store as well as application-specific KPI reports. Since such data is generated by third parties, the solution also needed the ability to periodically synchronize data with remote servers.

[3] `https://www.djangoproject.com/`

4.2.1 API

To obtain visualization data for the frontend, authorized users have access to an API endpoint with the following URL format:

/admin/stats/api/{SRC}/{VAR}/{FLT}/{AGG}/{PRM}

Explanation of the endpoints parameters can be seen in Table 2 bellow.

Parameter	Definition	Example	Reference
SRC	model from which the data should be retrieved	users.models.User	4.2.2
VAR	model's visualizable variable to be processed	phase	4.2.3
FLT	string-encoded set of filters	__none__	4.2.4
AGG	aggregation type used on retrieved data	freq	4.2.5
PRM	optional: URL-encoded parameters passed to visualizable methods	page_id=300	4.2.6

Table 2: Explanation of API endpoint parameters.

Each API call first evaluates validity of the passed parameters, in the order of their definition, and terminates on the first error it encounters. For debugging purposes, the error messages were designed to be verbose. For instance, if the data source validation fails, it offers the closest possible match as a suggestion. More details on the error messages can be found in their respective sections (see *Reference* in Table 2).

Once the initial validation passes, the API executes two database queries for the given data source. The first query is without the filters applied while the second query applies requested filters - unless no filters were requested in which case only one query is executed. Both filtered and unfiltered data then gets aggregated.

As mentioned in 2.1.3, not all values saved in the database are intended to be directly visualized but require a form of transformation from symbolic to explicit values. Once the data is aggregated, all symbolic values get translated prior to sending the API response.

4.2.2 Data Sources

Originally, all of the visualized data was directly linked to specific users (e.g. age or current phase). Later into the development, however, it became obvious that, in certain cases, the ability to pull data from models other than the User model would be necessary. Such cases would include the previously mentioned reports from Play Console and App Store from which we can get the number of application downloads for their respective platform. In the context of this thesis, a *data source* represents specific model that has been added to the API's configuration. All of the available data sources then form a subset of all the models created as a part of the whole project.

As of writing this thesis, there are exactly 5 registered data sources:

- **users.models.User**: each instance represents a user that is registered in the application,

- **stats.models.GoogleInstallsReport**: each instance represents a single installations report from the Play Console (i.e. statistics for one day),
- **stats.models.AppleSalesReport**: each instance represents a single daily sales report from the App Store,
- **stats.models.KPIReport**: each instance contains values required to calculate KPIs for given day (i.e. number of downloads, number of newly registered users, etc.),
- **payments.models.Payment**: each instance represents a single payment that can be paired to a user.

To avoid possible name conflicts in the future, full class identifier was chosen over class name only.

In order for a model to be among the registered data sources, it has to be added to the dictionary of allowed sources in the *DataSource* class. Each source entry requires the key to be the full identifier of the model. This entry then has 2 required attributes - *model* and *alias*. The model must be assigned a reference to the actual class. The alias must be a string - its sole purpose is to allow assigning additional, user-friendly name to the model which can then be utilized on frontend.

4.2.3 Visualizable Variables

In 2.1, it was described how the format of the visualized data may vary from variable to variable. Since it was established the format affects multiple steps of the data retrieval, there are essentially two ways of resolving this issue from the code's perspective - analyze the data and determine the correct properties automatically or allow the programmer to configure each variable on their own.

Let us consider the first option, i.e. no configuration required on programmer's side. The data type and dimensionality can both be easily determined from the data and any additional parameters could be detected by looking at the method's definition programatically. The issue is with values' meaning. Without any additional knowledge, it is not possible to reliably estimate whether the values should be translated or not. And even if it were possible, there would still need to be a way to determine which value mapping to use. This could be remediated if the methods returned already translated values but that would require re-writing large portion of the codebase and could potentially affect the performance of aggregation. Also, it limits any additional configuration, such as user-friendly names.

On the other hand, if all of the configuration is done by the programmer, no ambiguity arising from format prediction is possible. Not only that but the programmer can also define additional details about the variable that could potentially improve code readability, as well as improve user experience on the frontend by displaying additional information. For these reasons, the manual configuration approach was selected.

Variable configuration is done through a newly defined *visualizable* decorator. It is those methods decorated as visualizable that can be accessed via the API, i.e. variables cannot be visualized unless decorated appropriately. The full parameter list can be seen in Table 3 bellow. The visualizable decorator can be used on both regular methods and methods already decorated as *property*. The former is treated as a parametric visualization while the latter cannot take any parameters. In order to retrieve the visualization data, it is required to enter the method's name as the variable in an API request (e.g. *phase*, *sex*, *time_on_page*, etc.).

Parameter	Description	Format	Default	Legacy
alias	user-friendly name	string	None	no
dict	name of value translation mapping	string	None	no
values	inline translation mapping	dictionary	None	no
dimen	dimensionality	integer	1	no
numeric	should expect number?	boolean	False	yes
dependencies	names of variable dependencies	list	[ ]	no
labels	label for each axis	tuple	(”, ’četnost’)	no
ignore_zero	ignore the value 0?	boolean	False	no
format	expected format for each dimension	string/tuple	’text’	no
params	expected parameters and their values	dictionary	{}	no

Table 3: Description of *visualizable* decorator's parameters (*legacy* signifies whether the parameter is used only by the old API).

4.2.4 Data Filtering

While only the decorated methods can be visualized, each instance of the data source can be filtered based on any of its fields. Not only that, but it allows the same capabalities as Django's object filtering, i.e. can filter even by fields of other objects paired by foreign keys. For cases where there is no need for filtering, the filter processor[4] has a special keyword registered: *__none__*. In any other case, the *filter string* must either be a rule following the Django's syntax or a concatenation of such rules separated by the vertical line symbol (|).

The filtering rule has the following general format:

$$< \text{target field} > \_\_ < \text{comparison operator} >= < \text{specific value} >$$

where target field is the identifier of a non-relational field (e.g. *date_joined* for users). To take advantage of the relational capabilities, the user can chain fields, separating them with two underscores (__), until the desired non-relational field is reached (e.g. *access_code__business__pk* for users). The comparison operator can be any of the operators allowed by Django[5], such as *eq* for value equality or *gte* for greater than or equal to given value.

It should be noted that using multiple filters at the same time behaves as logical AND, i.e. all conditions must be met in order for the data source instance to be visualized. No other logical operator is currently supported.

4.2.5 Data Aggregation

Aggregation can be of great help when trying to understand large sets of data. Depending on the task, specific aggregation types can be more useful than others. Table 4 summarizes all of the implemented types.

The main reason for (un)limited dimensionality of the aggregated data is that, in the early stages of development, all of the data was only one-dimensional, i.e. a single value per

[4] *FilterProcessor*: class responsible for parsing filter strings

[5] `https://docs.djangoproject.com/en/3.0/topics/db/queries/`

Code Name	Title	Description	Max. dimension
freq, exact	Frequency	counts frequency of each unique tuple	*any*
stats	Statistical	finds minimum, maximum, and mode of a numerical dataset	1
histo	Histogram	builds histogram data using the Freedman Diaconis Estimator[6]	1
none	None	does not aggregate	*any*

Table 4: Overview of all available aggregation types with their descriptions. NOTE: The *exact* code name is for backwards-compatibility only, as it was used with the old frontend / backend.

model instance. Later into the process, however, the ability to calculate frequency of two-dimensional data, i.e. 2-tuples, became necessary. In order to avoid another functionality update in the future, the code was generalized to allow for any n-dimensional case. There is currently no use for such functionality, though.

4.2.6 Parameters

With the introduction of variables such as *time per page*, it was required to allow passing additional parameters to the models' methods to avoid code repetition.

Given a parametric variable to process, the API determines what parameters to expect (based on the visualizable configuration) and checks them against the parameters received from the API call. The *parameter string* follows a format similar to filter string - multiple parameters are separated by the vertical line symbol (|) and are subject to following general format:

$$< \text{parameter name} >=< \text{value} >$$

Such string is then parsed and only if all expected parameters are found does the API proceed with the request.

4.3 Frontend

The main goal of the frontend part was to provide a user-friendly graphical interface for the API. To achieve this, it was decided that the interface (= the application) would consist of multiple pages where each page would serve as a container for multiple independent visualizations. Since the structure of each page is the same, we shall focus only on the visualizations, as the pages themselves are used for organizational purposes rather than for functionality.

To build the frontend, various technologies were utilized on top of the three common web-oriented languages - HTML, CSS, and JavaScript. Firstly, it is the *Vue* framework - a progressive JavaScript framework for building user interfaces. Vue was chosen for its incremental adoptability and its gentle learning curve[7]. It also encourages better code organization and reusability through single file components.

This, however, required the addition of a compiler, as such components are not in a format known to the browser. A popular module bundler *Webpack* was used for that. Given

[6] abc `https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.histogram.html`
[7] `https://vuejs.org/v2/guide/`

a configuration file, it allows for processing, compiling, and bundling various types of source files. Although, Webpack itself only understands JavaScript and JSON files[8]. Additional functionality, such as compiling Vue templates, is done through third-party plugins and loaders that can be added to the pipeline.

To better organize style sheet snippets, the CSS extension language *SASS* was used. It adds features such as variables and nested declarations[9] on top of the CSS language. Like Vue templates, SASS requires a compiler which can be easily added to the Webpack workflow by using appropriate third-party plugins.

Finally, the *Google Charts* JavaScript library was used for the graphical representations themselves. Google Charts make it easy to transform tables into different types of charts, including pie charts, column charts, and line charts.

4.3.1 Vue Components

Decomposing larger layouts into components can help organizing code, as well as help avoiding repetition. The main idea was to have a general container for multiple visualizations where each visualization should have a title, the actual visualized data, a way of configuring the visualization (e.g. filters and aggregation), and buttons for additional actions, such as export or refresh. This rough idea could then be used to start designing components that shall be described in the following paragraphs. The final result can be seen in Figure 1.

VisualizerPage is the main, root component for all visualizations. Its main purpose is to load, save, and update page configuration. Such configuration defines what visualizations should be present on given page, in what order they should be displayed and how many visualizations can be shown next to each other (i.e. number of columns in a grid layout). It then persists the configuration through browser's *localStorage* instance and thus the user can continue working even after leaving the page. It should be stressed that the configuration is stored in user's browser and is in no way synchronized with the server. Therefore, it is available on that specific device only. Another one of its responsibilities is to load Google Charts. This is necessary because its terms of use do not allow for local hosting of the library[10] - otherwise it could have been bundled with the rest of the frontend code using Webpack and would have been loaded at the same time as the component. And finally, VisualizerPage allows the user to add more visualizations.

Visualizer is the key component of the application. It represents a single, independent visualization that is directly connected to the API endpoint. Each Visualizer instance receives the last saved configuration from its parent VisualizerPage which contains all of the following fields: title of the visualization, endpoint parameters (see subsubsection 4.2.1), preferred chart type, and whether to display the filtered or unfiltered data. Each Visualizer consists of its header, chart area, action panel, and configurator.

Header is a container for the vizualization's title and the filter toggle. Apart from passing forward the event of filter toggle switching to Visualizer, its role is to allow user to drag and drop the visualization from its current position to another available slot which, in this case, is the space occupied by any other visualization. This way, the user can personalize their experience without having to remove visualizations and then adding them again in the desired order.

[8] `https://webpack.js.org/concepts/`
[9] `https://sass-lang.com/guide`
[10] `https://developers.google.com/chart/interactive/faq`

Chart is a custom Google Charts wrapper that adds error/warning messaging capabilities. It is through the Chart component that the visualization data export happens. Based on the current state of the filter toggle it can either display all data or only the filtered entries. As both of these dataset are sent back with each request, it does not need additional API call and thus allows for quick switching between filtered and unfiltered data.

ActionPanel is where all action buttons are placed. Those include the refresh button which initiates a new API request, the configuration button which opens the configurator, buttons for both exportable types, and the delete button which removes the Visualizer from page's configuration.

Configurator is the last component of a Visualizer instance that can be seen in Figure 2. This component is responsible for building configuration object, i.e. JavaScript object, for API calls. For convenience, it consists of 3 subcomponents. The first subcomponent, *Configurator_general*, is where the user can enter their own title of the visualization, preferred chart type, information level (used for either showing or hiding additional data, namely column value percentages), and aggregation type. Next, *Configurator_data*, is the component that controls what data source is used as well as the visualized variable. In case the selected variable is parametric, it also offers the parameter selection. Finally, *Configurator_filters* is responsible for building and managing applied filters.

4.4 Older Solution

Initially, only requirements **F1** to **F5** were identified when designing the solution. This allowed for the development of a more user-oriented (as in *the User model*) solution. In such scenario, it was easy to implement global filtering, as required by **F4**, since all visualizations were connected to the same model with the same available fields.

This solution was, however, quite limited. The design choices would make it difficult to extend the API in terms of possible data sources, filtering fields and several more aspects. Also, the user could experience worse performance under certain conditions due to the specifics of the implementation. For that reason, the new version (on both frontend and backend) was created to overcome these issues. The only problem with the new solution was that it had not been decided how to handle global filtering - since it allows users to add any number of visualizations connected to potentially all available data sources, each one having different fields for filtering.

Given that most of the desired visualizations would still be user-oriented, where the ability to filter multiple visualizations at once is very useful, it was decided to let both versions coexist. This way, the user experience stays the same until the preferred approach is selected at which point the old version can be fully replaced by the new version.

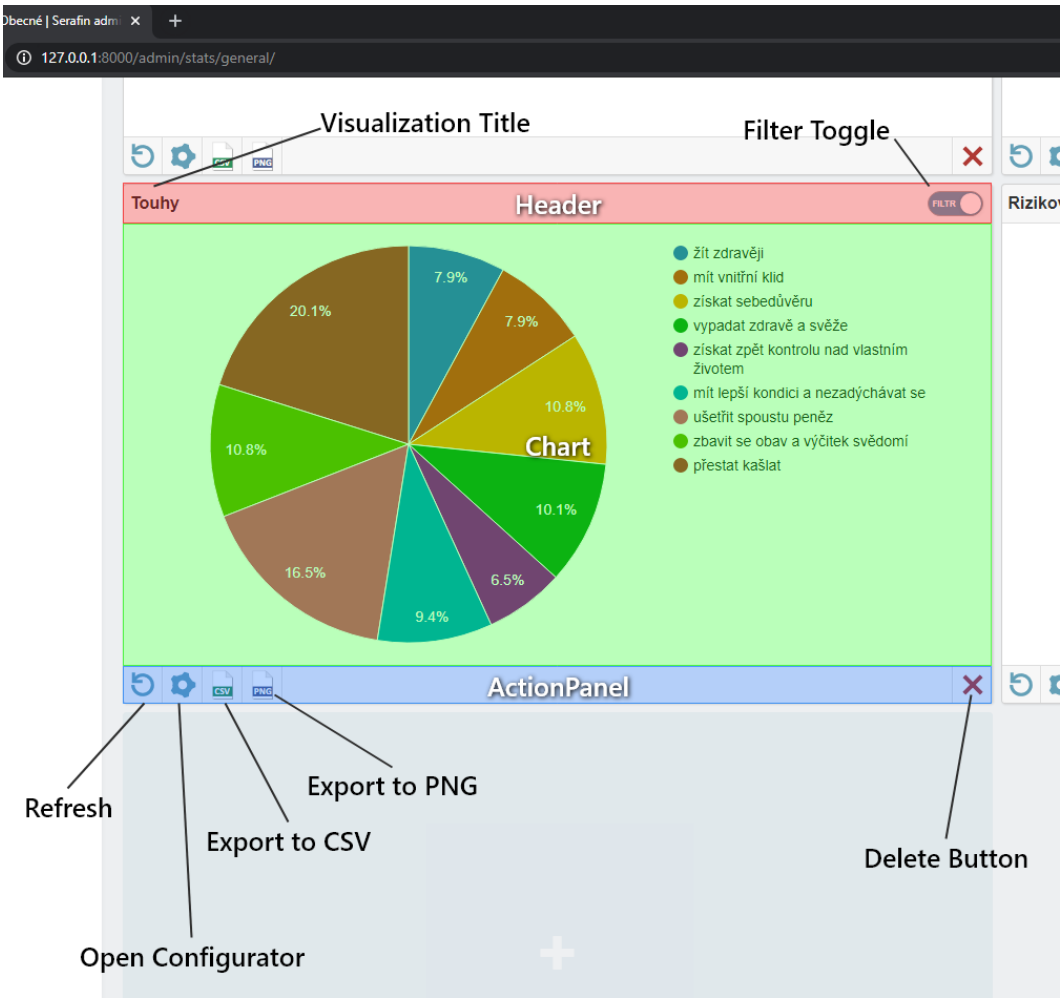

Figure 1: Decomposition of the *Visualizer* component - excluding the configurator.

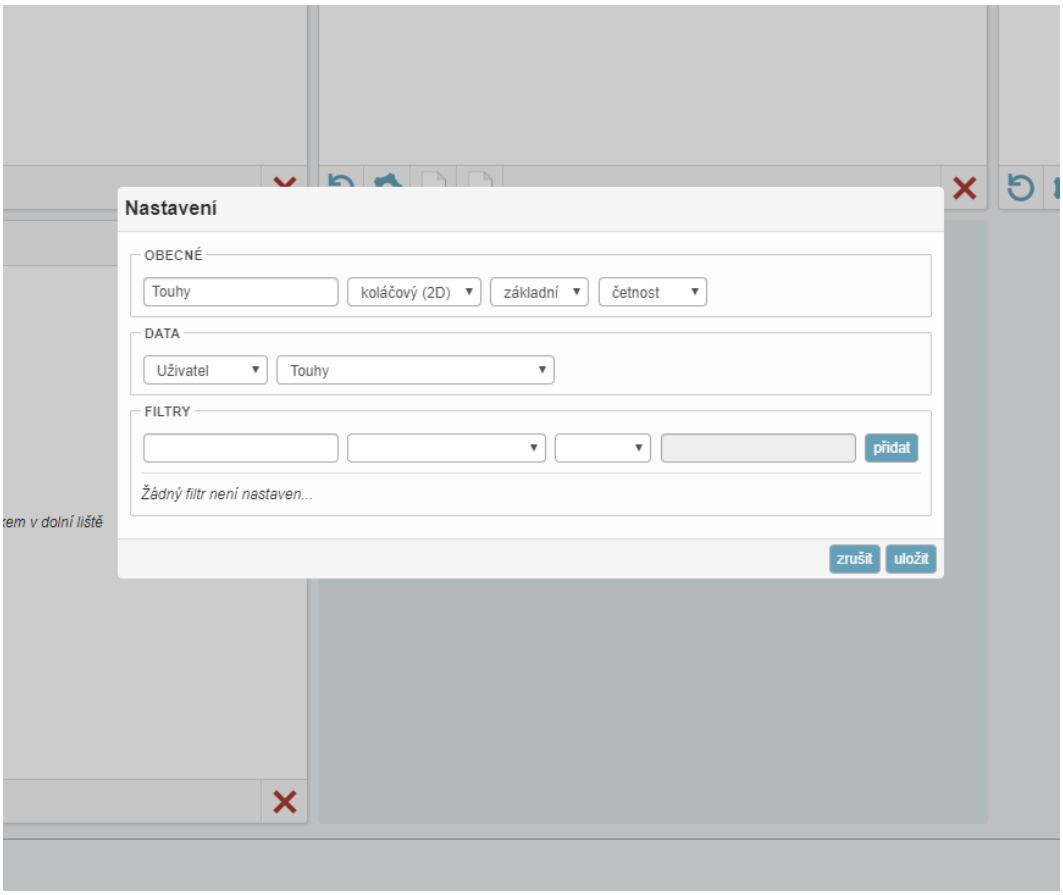

Figure 2: Preview of the *Configurator* component.

5 Text Classification

Classifying textual data can be, and in this case is, a computationally demanding process. For vectorization, the FastText library was utilized with a pre-trained model that is approximately 7 GB in file size. Therefore, it should be stressed that it would not be feasible to integrate the classifier into the website, as the whole model needs to be loaded into memory. To overcome this issue, both testing and the final classification were done locally.

5.1 Dataset

To implement the K-NN classifier, a set of already labeled data needs to be prepared. Such dataset can then be vectorized and loaded into the classifier's memory for later evaluation of new sentence vectors. For this purpose, a subset of the already saved answers was manually labeled. The size of the dataset is 600 entries out of roughly 1800 in total. Answers that were in a language other than Czech or Slovak were removed for the purpose of this classifier.

5.2 Hyperparameters

In general, a K-NN classifier only has one hyperparameter that can be optimized - the value of K. In subsection 3.5, however, two different approaches to the classification were introduced: naive and statistical. In the former case, a threshold was defined that could potentially be optimized as well. The latter did not introduce any additional parameter.

For the optimization, value ranges of the hyperparameters had to be defined. Experiments for this thesis used the following sets of values:

- $k \in \{x \in \mathbb{N} \mid 1 \leq x \leq 40\}$,
- $\epsilon \in \{\emptyset, 0.5, 0.6, 0.7, 0.8\}$,

where the threshold $\epsilon = \emptyset$ signifies that the statistical approach should be utilized instead of the naive approach.

In subsection 3.5, it was mentioned that the K-Nearest Neighbors algorithm uses a predefined metric to identify neighbors. To evaluate distance between vectors $\boldsymbol{a}, \boldsymbol{b}$ in this case, the cosine distance was utilized, i.e.:

$$d(\boldsymbol{a}, \boldsymbol{b}) = 1 - \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\| \, \|\boldsymbol{b}\|}$$

5.3 Classifier Accuracy

A key piece of information about the classifier is an estimate of how well it might perform on a previously unseen dataset. To calculate this value, we need to define how to get such estimate for the labeled dataset and how we score a single evaluation.

Let us focus on the evaluation scoring first. From subsection 3.2, there are exactly 7 possible labels where each one is either assigned or not assigned. Each sentence evaluation is then represented as a 7-dimensional assignment vector with its components being either 0 (= not assigned) or 1 (= assigned). Then the accuracy can be defined as:

$$S_{\mathcal{T}} = 1 - \frac{1}{|\mathcal{T}|} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}} \frac{1}{n} \left\| \mathrm{KNN}(\boldsymbol{x}) - \boldsymbol{y} \right\|_1 \tag{3}$$

where $S_{\mathcal{T}}$ is the accuracy function for given dataset $\mathcal{T} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), \dots (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$, m is the number of samples in dataset, $(\boldsymbol{x}, \boldsymbol{y})$ is a pair of a sentence vector $\boldsymbol{x}$ and its correct assignment vector $\boldsymbol{y}$, KNN($\boldsymbol{x}$) is the assignment vector generated by the classifier, and n is the dimension of the vectors, i.e. the number of known labels. This approach assumes that both false positives and false negatives are equally undesirable.

To evaluate accuracy, a set of labeled data needs to be available. One possible method is to split the dataset into two subsets depending on the desired ratio of training to testing data, e.g. 4:1. However, the available dataset is quite small and thus a possibly more appropriate approach is *k-fold cross-validation*. Its logic can be summarized as follows:

1. shuffle the dataset,
2. split the dataset into k chunks,
3. select one chunk for testing (one that has not been selected yet) and use the rest for training,
4. train and test the classifier (saving its score for later),
5. go back to (3) unless all chunks have been used as testing datasets,
6. average the scores to get the final accuracy.

This way, the accuracy evaluation is less prone to bias in training / testing dataset which could otherwise result in an unreliable score (e.g. too optimistic).

5.4 Classifier Evaluation

A series of classifier evaluations were performed to find the optimal hyperparameters. Based on the sets defined in subsection 5.2, all possible combinations were generated and then, using 10-fold cross-validation, scored.

Firstly, the naive approach generally achieved accuracy of 80% or higher as can be seen in Figure 3, with the only exception being threshold $\epsilon = 0.8$. And, for most of the K values, the optimal threshold was $\epsilon = 0.6$ while also being the most stable (meaning incrementing the K value by 1 does not change the classifier's accuracy significantly).

Secondly, the statistical approach was able to achieve similar results to the naive approach with $\epsilon = 0.6$. Comparison of their results can be seen in Figure 4. The experiment has shown that the maximum score is approximately **84.286%** and is achievable for $K = 23$ with the statistical approach . Therefore, this configuration was selected for the final classification of the whole dataset.

5.5 Classification Results

Using the best configuration found during hyperparameter optimization, the classifier was given the full labeled dataset for reference and the classification was performed on the full dataset of 1822 responses. The results can be seen in Figure 5. Relative frequencies of individual labels from most frequent to least frequent can be seen in Table 5 bellow.

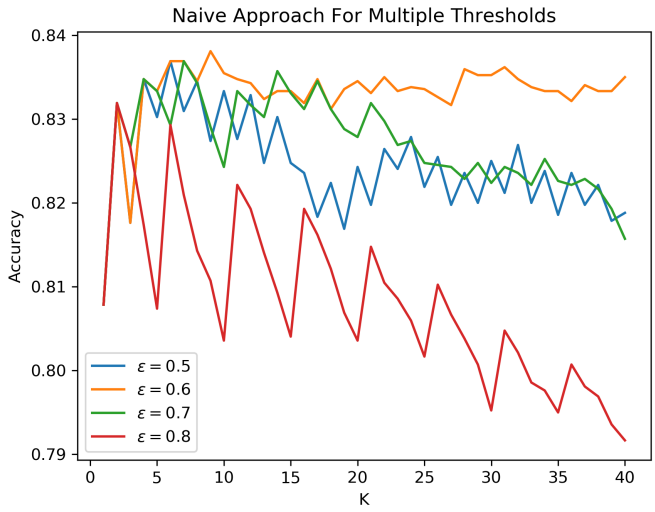

Figure 3: Naive approach performance over multiple values of K.

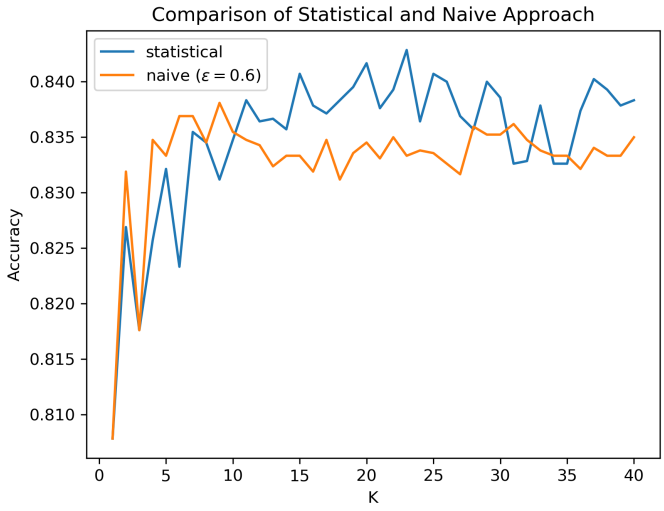

Figure 4: Comparison of the statistical approach and the best performing naive approach.

Label	Relative Frequency
health	60.318%
money	37.541%
values	18.551%
relationships	15.642%
state of mind	10.263%
pregnancy	1.262%
aesthetics	0.439%

Table 5: Relative frequencies of labels among data from the ReaQui variable.

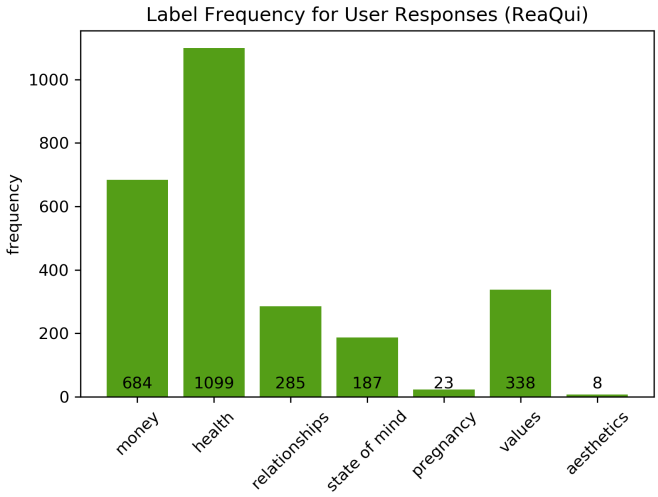

Figure 5: Frequency of individual labels assigned to users' main reason(s) for quitting smoking.

5.6 Exploring the Results

The K-Nearest Neighbors algorithm has proven its ability to achieve good results even for smaller datasets. To further check the validity and the nature of the classifier's weaknesses, a sample of 20 responses was randomly selected (5 of which were part of the classifier's knowledge). Using the accuracy function (see Equation 3), an overall score of approximately **87.143%** was achieved. The most frequent classification error was not assigning the *relationships* label (in 5 out of 20 cases). Similar issue was with the *state of mind* label which was falsely not assigned 3 out of 20 times.

These finding may suggest that the *relationships* label is defined too broadly given its representation in the labeled dataset. Broader definition could then lead to a higher chance of label occurrence being dependent on another label which, in turn, may introduce a bias to the known dataset. Let us consider a scenario where there are many known vectors assigned only label A. Then, we pass a sentence to the classifier where we expect a combination of labels A and B. However, label B may be overlooked simply because there are more examples

of label A than label B (assuming the vector representing the new sentence is close enough to an average of any two known vectors assigned only label A and only label B).

Both the *pregnancy* and *aesthetics* labels are worth examining further due to their very low frequency among user responses. If we assume that the average user is similar to an average participant of the 2011 study [2], we should expect a much higher relative frequency of the *pregnancy* label. Although, this might be a very strong assumption given the different time frame of the experiment, as well as potentially very different sociodemographic characteristics. Nonetheless, frequencies of both aforementioned labels emphasize the core problem of the naive approach, i.e. given the value of $K = 23$, it could easily encounter a situation where under-represented label can never be assigned. This issue, however, is mitigated by the use of the statistical approach which takes into account the number of label instances in the whole dataset.

6 Conclusion

The aim of this thesis was to analyze and process user data from a smoking cessation programme mobile application. The first part of the analysis focuses on data such as user's cessation programme progress or their age where the possible values are well defined, i.e. are set by the application automatically, are selected by the user from a predefined set of options, or are strictly limited to numerical values. This type of data is then categorized by its data type (number or text), dimensionality (size of the returned n-tuple), meaning (symbolic or explicit), and parameters (no parameters or a list of parameters), as each of the categories may limit the way the data can be processed.

The second part of the analysis focuses on a more complex problem - free-text input analysis. During certain steps of the cessation programme, users are allowed to enter any textual value they want. During the first steps, the user is asked to enter their main reason for quitting. This field was selected as the main area of interest in terms of text analysis, as it provides a unique and important insight and it also offers the largest dataset. The goal was to build a classifier capable of assigning multiple labels to each user response. Using findings from a 2011 study [2] combined with preliminary analysis of the already entered user responses, a set of 7 labels was defined. To perform the classification, the K-Nearest Neighbors algorithm was selected for its ability of achieving good results even for smaller datasets. Two different approaches of using the algorithm for multi-label classification were proposed - naive and statistical. The naive approach only analyzes the labels assigned to the nearest neighbors while the statistical approach takes advantage of prior knowledge about whole dataset as well, taking into account probability of individual labels occurring under different circumstances.

After the analysis, an extension of Django's administration interface was described. This solutions consisted of an API for retrieving, filtering, and preprocessing user data and an appropriate frontend to visualize it. The API's architecture was built directly on top of the knowledge acquired from the first part of the analysis, utilizing a 4-step procedure of locating data (the source and the field), applying requested filters, aggregating according to selected rule, and translating from symbolic to explicit values when necessary. It was then extended to allow for non-user data, such as reports from third-party websites (Google's Play Console and Apple's App Store). The proposed frontend solution consisted of multiple pages with the same structure, allowing the user to organize visualizations into groups. Each page is then described as a collection of independent visualizations where each visualization is a wrapper for the API and the Google Charts library.

The final section was dedicated to the implementation and results of the K-Nearest Neighbors classifier. Both proposed approaches were compared and discussed, showing the advantages and slightly better accuracy of the statistical approach when compared to an optimized version of the naive approach. The classifier with optimized hyperparameters, i.e. statistical approach for $K = 23$, was then used on analyzing 1822 responses given by the users.

References

[1] World Health Organization. *Fact Sheets - Tobacco*. [online]. URL: https://www.who.int/news-room/fact-sheets/detail/tobacco (visited on 2020-05-21).

[2] Silvano GALLUS et al. "Why do smokers quit?" In: *European Journal of Cancer Prevention* 22.1 (2013), pp. 96–101. ISSN: 0959-8278. DOI: 10.1097/CEJ.0b013e3283552da8. URL: http://journals.lww.com/00008469-201301000-00014 (visited on 2020-04-25).

[3] Tomas MIKOLOV et al. "Efficient estimation of word representations in vector space". In: (2013). arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781v3 (visited on 2020-04-25).

[4] Gulisong NASIERDING and Abbas Z. KOUZANI. "Comparative evaluation of multi-label classification methods". In: *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery* (2012), pp. 679–683. DOI: 10.1109/FSKD.2012.6234347. URL: http://ieeexplore.ieee.org/document/6234347/ (visited on 2020-04-25).

[5] Min-Ling ZHANG and Zhi-Hua ZHOU. "ML-KNN: A lazy learning approach to multi-label learning". In: *Pattern Recognition* 40.7 (2007), pp. 2038–2048. ISSN: 00313203. DOI: 10.1016/j.patcog.2006.12.019. URL: https://linkinghub.elsevier.com/retrieve/pii/S0031320307000027 (visited on 2020-04-25).

[6] Apple Inc. *Summary Sales Report*. [online]. 2020. URL: https://help.apple.com/app-store-connect/#/dev15f9508ca (visited on 2020-03-26).

[7] Google Inc. *Download export monthly reports*. [online]. 2020. URL: https://support.google.com/googleplay/android-developer/answer/6135870 (visited on 2020-03-26).

[8] Bing Liu et al. "Text classification by labeling words". In: *AAAI*. Vol. 4. 2004, pp. 425–430. URL: https://www.cs.uic.edu/~liub/publications/aaai04-labelWords.pdf (visited on 2020-03-19).

[9] Django Software Foundation. *Django 1.8 Documentation*. [online]. URL: https://docs.djangoproject.com/en/1.8/.

[10] JS Foundation. *Webpack 5 Documentation*. [online]. URL: https://webpack.js.org/concepts/.

[11] Facebook Inc. *fastText Documentation*. [online]. URL: https://fasttext.cc/docs/en/support.html (visited on 2019-04-10).

[12] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *arXiv preprint arXiv:1607.04606* (2016). URL: https://arxiv.org/abs/1607.04606 (visited on 2020-04-25).