University
of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2018 Open / ~~Confidential~~
Author: Mats Jonassen	[illegible] (signature of author)
Programme coordinator: Vinay Jayarama Setty Supervisor(s): Vinay Jayarama Setty	
Title of Master's Thesis: Adaptive Selection and Delivery of Rich Media Notifications to Mobile Users	
Credits: 30 ECTS	
Keywords: Media notifications • Random Forest Classifier • Resource Optimization • Knapsack Problems	Number of pages: 66 + supplemental material/other: - Code included in PDF Stavanger, June 15, 2018

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Adaptive Selection and Delivery of Rich Media Notifications to Mobile Users

Master's Thesis in Computer Science
by

Mats Jonassen

Internal Supervisor

Vinay Jayarama Setty

June 15, 2018

Abstract

The ongoing increase in cellular network coverage is steadily increasing the availability of individuals all around the world. This availability enables notification solutions to achieve their goals of announcing new content with great success. Recently notification systems have evolved to include media content. In cases where this content is of substantial quality and frequency, it may induce large resource consumption.

We aim to limit the total resource consumption of media notifications while preserving the user experience. We achieve this by implementing established techniques for measuring the quality of content. We enable the utilization of these techniques by implementing a working systems tackling the practical issues of notification generation, device communication and notification scheduling. We test the system using Spotify as our notification provider and compare our results to the standard FIFO approach of notifications. Using these tests we find that correctly prioritizing content can massively increase a users utilization of notification content.

Acknowledgements

I would like to thank Vinay Jayarama Setty for his guidance, supervision, and ideas over the course of this project. He has provided a bastion in times of trouble and without his help this thesis would have never been. I am also incredibly grateful for the support the University of Stavanger gives its students. Finally I would extend a thank you to my fellow students for supporting each other as we collectively lost our minds.

Contents

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
URL	**U**niform **R**esource **L**ocator
FIFO	**F**irst **I**n **F**irst **O**ut
JSON	**J**avaScript **O**bject **N**otation
CSV	**C**omma-**S**eparated **V**alues
DDOS	**D**istributed **D**enial **O**f **S**ervice

Chapter 1

Introduction

1.1 Motivation

In the last 15 years global mobile cellular subscriptions have increased by over six billion [1]. From 2011 to 2015 3G coverage rose from 45% to an estimated 69%, with urban population coverage reaching an estimated 89%.

Numbers from Gartner [2] put Android and iOS operating systems in 99.8% of mobile devices sold in the first quarter of 2017. Both of these operating systems support media notifications, allowing media to be inserted directly into their notifications. These can contain images, audio, and video, all displayed on the home screen.

These notifications provide excellent tools for social media services to appeal to their userbase. With this surge in available consumers, a large battle between these service providers has ensued. The constant fight for more users and their attention has incentivised the providers to produce high-quality content, increasing market shares and advertisement revenues.

The mobile devices, that represent the targets of this content have limitations in available resources: batteries have a set amount of ampere-hours, and cellular data use is limited by data plans. A user receiving a large amount of media notifications may experience a rapid drain in available device resources. This may be due to being a member of multiple notification services or being a member of remarkably aggressive providers.

The ultimate consequence is inefficient notifications that may leave users frustrated due to the high resource consumption of their devices, and possibly feeling overwhelmed or even hounded by these services.

1.2 Problem Definition

The goal of this thesis is to explore and implement a resource-aware system to maximize users utility of media notifications for mobile devices under resource constraints.

The constant change in the hardware of smartphones leads to similar changes in both resources available and in resources consumed. For data, these resources are also affected by the user's data plan. Thus, for us to approach the goal stated above we need to develop a technique capable of collecting the resource constraints, and these constraints have to appear satisfactory for all users even with their completely unique hardware and use patterns.

Due to the nature of resource limitations, we will be unable to transfer an unlimited amount of notifications. The maximization of user utility is convoluted as the degree of user satisfaction has to be estimated ahead of delivery, and choices have to made on what notifications should obtain priority.

To ensure the goals of the implementation are met we should generate media notifications from a realistic provider and test the system from notification generation to delivery.

1.3 Example

A mobile device may be disabled for a period of time. A user may be on a flight and required to turn their phone off. This can lead to notifications accumulating throughout the day. Upon activation of the device, the relevant user may be flooded by notifications in an attempt at briefing them of new content. In cases of large catch-ups, and disconnected consistent options such as WiFi and battery power, users can experience large battery consumption and heavily reduced cellular data speeds.

A user can apply our solution as their notification framework. Activation of the device would thereafter lead to priority notifications arriving first, and fewer notifications in total allowing users to combat notification flooding with limited use, and save their resources for times of better resource availability.

1.4 Contributions

We realize our objectives by designing a solution utilizing a machine learning model to calculate a utility score for the individual notifications. With the aid of user-defined data constraints, we model notification selection as a 0–1 knapsack problem that is solved with a greedy algorithm.

This design is implemented and tested using Spotify as our notification provider. We explain how to generate, score estimate, and transmit the notifications. We continue by defining the requirements to implement other notification services and explain the control structure linking these services to the final transmission of notifications.

We also show our results for experiments surrounding resources used and compare them to more current implementations.

1.5 Outline

This thesis is structured as follows.

Chapter 1 Introduces the problems and objectives of this thesis.

Chapter 2 Exposes relevant background information, introducing the reader to related theories.

chapter 3 Presents the proposed solution design

chapter 4 Explains the core components of producing a working system.

chapter 5 Presents performance tests and results.

chapter 6 Concludes the thesis and suggests future expansion.

Chapter 2

Background and Related Work

2.1 Related Work

Uddin, Setty, Zhao, Vitenberg, and Venkatasubramanian [3] suggest in their paper "Richnote: Adaptive selection and delivery of rich media notifications to mobile users", a theoretical analysis and solution suggestion to a system notably similar to the system we aim to implement. The paper does not cover all of the practical limitations encountered during the implementation. It does, however, suggest issues that need resolving before attempting perfect implementation. It also presents the notion of presentation levels, subdividing each notification into differing qualities, with different value metrics.

Similarly, the work of Do, Zhao, Wang, Hsu and Venkatasubramanian [4] lays a basic foundation for controlling notification resource expenditure. Do et al. [4] suggests a broker/proxy system collecting all notifications, and subsequently only spending resources on the most relevant ones.

2.2 Mobile Notifications

Notifications for smartphones have been developed and improved upon since the first modern smartphone, the iPhone, from 2007. The goal of notifications is to transport information from a server to a mobile device. The target of a notification may be a single device or a group of devices. Classically there are two core techniques used to achieve such a system: push notifications and pull notifications.

In a pull notification system, the device polls the server for new information, often at set intervals or using some manner of backoff algorithm. This has the advantage of reducing the resource overhead related to maintaining an active connection. The central drawback to the pull notification is that it is not aware of whether or not the server can provide any new information. A pull request has to be attempted and may not yield any new information and be considered a waste. This could be helped by increasing the time interval between each server poll, thus, increasing the probability of a new notification. However, in a pull notification system, the time delta between each poll also dictates the average waiting time for a notification. If a notification was to arrive shortly after a polling request the device would have to wait its entire time delta to retry, and finally obtain the media notification. For these reasons balancing the delta in a poll system is a difficult task that heavily defines the resulting performance of the notification system.

A push notification system allows the server to dictate when the notifications are sent and received at the mobile device. This removes the disadvantage of waiting for updates and allows time-sensitive updates like messaging and news alerts to be collected as soon as they are available. This is done by leaving an open connection. Such a connection can consume a non-trivial amount of power. If a number of different notification services all maintain separate connections for the purpose of push notifications the idle battery consumption of the device will drastically increase.

To combat this issue most mobile device providers implement their own notification systems allowing multiple notification providers to use the same connection to push notifications to the device. This results in a single, better-utilized connection. The efficiency of this push system, its performance, and the simplicity in using these libraries have all lead to this being the prevailing solution. These systems do however sometimes apply their own limitations such as a max rate of notifications and limit on greatest data amount per notification.

In recent years operating system providers have allowed for more detailed notifications. To further increase developers ability to engage with their users most of these providers now allow media notification. Media and metadata are directly embedded into notifications, displaying images, video, and playing audio. Media notifications also commonly support interactive elements such as media controls.

2.3 Publisher/Subscriber

Publisher/Subscriber, often denoted Pub/Sub, is a system paradigm. In this paradigm, a large number of subscribers may subscribe to content they are interested in. This content is produced by publishers, who publish their content to a central server. This service then distributes it out to the subscribers in an asynchronous manner. This technique allows stronger separation of the two services as it is not required of them to be aware of each other. The publishers publish content to an event service, which in turn distributes it to the subscribers. The specifications of the event service are very loose in the paradigm allowing for highly distributed and scalable solutions. This partially explains the surge in Pub/Sub systems with large services [5].

Our system is comparable to that of a pub/sub system in its aggregation of data, however while a pub/sub system focuses on spreading information from a single provider to multiple users, our system will collect data from multiple sources and present it to a single subscriber, as we do not know ahead of time what users are interested in what data.

2.4 Optimizing Notification Content

The dominating approach in research [3],[4] to notification optimization is broker/proxy like systems similar to Do et al. [4]. The approach can be summarized in the following steps:

1. Intercept and accumulate all notifications destined to the device.
2. Evaluate all notifications.
3. Decide what notifications meet the threshold for delivery.
4. Deliver the notifications from step three, withhold notifications that do not.

Approaches in this style often differ in how to evaluate and how to decide threshold. There are also differences in what to do with remaining notifications. The general consensus is however that the most efficient technique to save resources over time in regards to notifications is to prevent insignificant notifications from transmission in over-constrained systems.

2.5 Content Utility

The estimation of content utility is a difficult and highly subjective issue. Estimating users satisfaction with regards to content has been researched and tested and is widely used for recommendation of content. Most companies implementing recommendation systems consider the specifics of the implementation to be of great value and thus attempts to keep details private. Common recommendation techniques include machine learning using social relations and interactions [6][7], collaborative filtering, and machine learning based on content and user history analysis.

The predominant technique used with machine learning is using models estimating the probability of notifications being used. Uddin et al. [3] suggests using a random forest classifier with class probabilities for utility score, while Do et al. [4] suggests a logistic regression classifier, to build a prediction model.

2.6 Saving Resources

Saving resources in notification system is mainly achieved by smart selection of what content to notify of [4], with a core objective of only delivering poor notifications when resources are in abundances such as available WiFi or connected power chargers. These techniques require heavy investments into notification importance classification.

Previous research also suggests variable quality and size in notification [3]. This achieves lower resource consumption of notifications when possible, and suggests maximizing quality when it presents the greatest reward. This requires notifications to be available with different levels of quality.

2.7 Scheduling

The scheduling of the notifications and updates both influence the performance of the notifications. If notifications are not scheduled and pushed on availability the service will work very much like a FIFO notification service. During continuous periods of online connectivity, notifications will be sent as soon as they become available. The system will also lose its ability to compare notifications that are outbound, as there will not be multiple notifications outbound simultaneously. waiting too long between each round of calculations will, in turn, lead to large wait times for notifications, hampering their use for time-sensitive items like instant messaging.

Uddin et al. [3] suggests grouping notifications into batches and maximizing notification performance per batch. A batching system improves performance by abiding by the constraints per "round" and carry over remaining constraint to the next. With further analysis, this can be used to find a user optimal round interval.

2.8 Knapsack Problems

Knapsack problems are a set of problems commonly defined as: given a set of items with a value and a weight, maximize the sum of values while keeping the total weight under a max weight. There are multiple knapsack problems all adding different additional rules to the base problem, e.g. the fractional knapsack problem allows items to be divided into any fraction, and the unbounded knapsack problem allows for multiple copies of the same item. The most common knapsack problem is the 0–1 knapsack problem where there is only one of each item and it has to be taken in full or abandoned in full.

The 0–1 knapsack problem is defined as:

$$\text{Maximize:} \quad \sum_{i=1}^{n} s_i x_i \tag{2.1}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n} c_i x_i \leqslant C \quad \text{and} \quad x_i \in \{0,1\} \tag{2.2}$$

All knapsack problems with complexity similar to the 0–1 knapsack problem or greater are NP-hard. Thus there exist many heuristic solutions that, in practice, outperform optimal solutions as the constraint space grows or the amount of simultaneous knapsack solutions increase.

Chapter 3

Solution Design

The aim of the solution is to design a tool for efficient notifications. The solution should support the preferences of the users and enable them to conform to the limitations set by the many mobile internet providers. In general, a user should receive a notification when it arrives, and not have to wait for some extended time. Partially because notifications are sometimes expected and operating with a long notification interval may cause users to question the availability of the service. Users may also disappear for a period of time. This can cause notifications to stack up as they are unable to be delivered. In turn this may cause the resource consumption of the notifications to exceed the user-defined constraints for the time interval. To resolve these situations it is vital to provide a notification selection algorithm that can supply the most notification value under the available constraints.

We propose a system consisting of separate modules enabling easy problem detection, problem separation, and allows us to easily alter the implementations of the different modules.

3.1 System Requirements

The generated notifications have to cover most applications and sizes. By keeping the requirements for notifications simple we allow for the implementation of multiple social media services without restricting the notification content.

To maximize notification utility with received notifications we need to able to estimate whether it will be activated by the user. We define activation of content as the user pressing the notification, loading it in full and engaging with it. Engaging with content may be looking at an image, looking at a video or listening to audio. This estimate of whether a notification will be activated is henceforth referred to as utility score. Estimating this utility score has to be done before sending the notification. The goal of a notification considered as being activated by a user.

To ensure that the solution adheres to the data constraints entails a measure of data use per notification, as well as solution to construct these constraints. The measure of data usage of a notification has to be estimated ahead of transmission. This prevents the notification from applying its resource consumption during utility score calculation and allows the server system to analyze the and simulate the transmission of notifications.

With the two measures data cost and utility score, the system will be able to control the amount of utility sent under a defined data constraint and maximize this value for optimal notifications.

It is assumed infinite resources for the server system, due to our focus on efficient mobile device resources. In turn, this allows the server application to analyze the notification elements without limitations.

3.2 Scoring

The variations of social media elements differ greatly and their appeal fluctuates with varying relative amounts. We, with the aim to facilitate comparison, introduce utility score. Utility score serves as a measure of the value of a notification. utility score is defined as the probability, between 0 and 1, of a user utilizing the notification. This implies that two notifications with a relatively low utility score may combine to the same score as a better notification that provides a significantly larger probability of being activated.

Example: We have three notifications with (utility score, cost): (0.6, 50), (0.5, 45) and (0.9, 95). The system and its schedulers would assume the first two notifications would provide a total of 1.1 utility score with a cost of 95. If the data constraint is lower than 190 these two would be the selected notifications.

However, if we label the three notifications with A, B, C respectively, the probability of a user activating at least one of the two selected notifications is notifications is:

$$\mathbb{P}(A \cup B) = 1 - \mathbb{P}(\bar{A} \cap \bar{B}) = 1 - 0.4 * 0.5 = 0.8$$

Hence selecting notification C would have provided a better probability of notification activation. Selecting both notifications does, however, bring the possibility of a user activating multiple notifications. Our model works with expected values and is only accurate as long as we consider the activation of two notifications twice as good as the activation of a single notification. This may not be accurate for users who might prefer very good notification content over two decent elements.

The use of 0–1 scoring allow relatively easy comparison of notifications from multiple providers. This solution is naive and not safe from mischievous providers who may artificially increase their utility score to receive a higher priority for their notifications.

3.3 Gathering User Constraints

Cellular data is controlled at the highest level by cellular providers. These commonly provide subscription plans with a fixed data capacity or charge per data unit transmitted. Capacities are controlled in long-term periods and for limited data plans, these limits are usually reset monthly. The data available in these plans range from a few megabytes to unlimited, hence it is difficult to infer constraints that satisfy multiple users. If a user exceeds the limit set by their cellular providers they may lose access to cellular data or experience a heavy reduction in network speeds. Unlimited users may, with wrong constraints for their use, receive large and troublesome invoices. Risking these drawbacks gives users incentive to limit their data use to some maximum bound. We, therefore, let the user define their own desired data constraints.

Battery power in smartphones is a short-term resource compared to cellular data. Power outlets and USB charging ports are installed in many public areas from restaurants and cafés to airports and planes. New mid to high-end phones usually provide around 3000 mAh of power [8][9][10] resulting in around 11 to 14 hours of cellular data communication. This leads to devices running out of power frequently, and a large variety of effective battery life that is highly dependent on individual users, and day to day events. Android provides insights into app-specific network information but does not extend this support for battery information. While there are tools to help measure the current battery level, attempts at measuring battery consumption are distorted by background processes, and thus often inaccurate.

This results in battery power being difficult to implement as a constraint, as the device provides no guarantee that the power will be there on transmission. Transmission of cellular data is also very closely related to power consumption and these two factors usually follow each other quite closely. It is therefore likely that if we were to implement a power constraint for selection most of our notifications would scale equally in the two cost dimensions, and the selection module would in practice run the same process with the lowest of the two constraints as the critical factor. With the difficulty being so high and the probable result being so inefficient, we decide that a battery power constraint will not be implemented for notification selection.

As discussed above battery power is another constraint in which a user may detect a performance difference, but due to the problems highlighted it is difficult to consider in the selection process. However due to negative emotions commonly being related to running out of power we allow an implemented mobile application to present a stop flag for low power. This enables a user to preserve their remaining battery level.

3.4 Estimating Resource Costs

Mobile devices use many different resources to receive a notification. Cellular data and battery power are used to download data, and processing and memory are used to present it. With the resource consumption being our limiting factor for transmitted and activated notifications, an accurate estimate measurement for resource consumption will allow us to accomplish better results by enabling confident approach of constraints.

Cellular data, battery power, processing power, and memory are all resources used in the reception and presentation of a notification. Modern mobile devices are often sold with gigahertz processors and gigabytes of ram, making the processing and memory requirements trivial when compared to the scale of data constraints and battery constraints. Transmission of over large open-air distances requires large amounts of energy. Displaying image notifications activates large, often high definition, screens and playing music may power large headphones or Bluetooth communication. For these reasons, we will consider cellular data and battery power as our constraints of interest.

The data cost of a notification is measured using the size of the notification. This does not account for the overhead per transmission, which we have little control over due to its dependence on the network protocol. If the data contents of a notification are known we can also measure the data size of the content. This gives us a very accurate estimation of data content per notification. Data costs will be measured in the number of bytes. Sometimes it may be most practical to only send the meta-data on its own, e.g. to allow device isolated user security. In such cases, data should still be estimated based on the total that will be used to deliver the end notification. This allows the selection algorithm to remain accurate and prevents the data usage from surmounting the user-defined constraint.

3.5 System Overview

With these requirements and tools, we propose a system comprised of three modules. Separating the responsibilities into three parts: Generating notification, selecting notifications to deliver, and sending notifications. Figure 3.1 displays an overview flow of notifications through the modules.

Utility scoring and Data cost estimation are naturally notification provider specific. This is because generating these values can be done very differently depending on the type of notification provider. The details of these constructions are therefore not expanded beyond what is clarified in Sections 3.2 and 3.4. Chapter 4 contains more details on how this is done for a single notification provider.

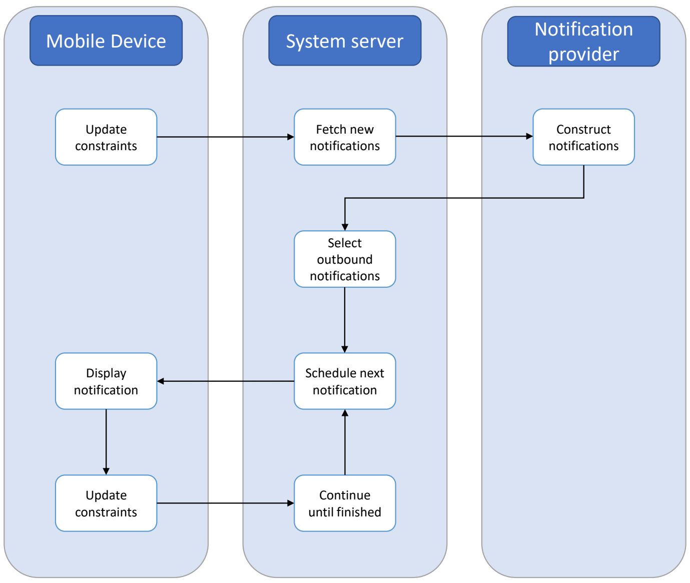

Figure 3.1: The flow from a user preparing to receive notifications, to all current notifications sent. The mobile device, at exponential backoff intervals, updates its the systems available resources. The server requests new notifications from the notification provider on behalf of a user. Upon reception of these notifications the server initializes the process of transmission by selecting the best notifications in the set. These notifications are continuously transmitted until either: A notification is not acknowledged, or all notifications are sent.

3.6 Selection

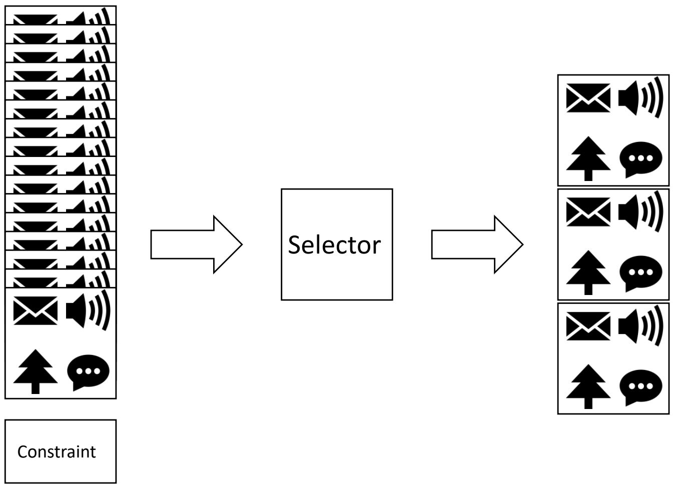

Figure 3.2: The selector receives a set of notifications and a constraint. Using these parameters it selects the notifications that maximize utility score while abiding by the constraint.

Selecting what notifications to transmit is one of the core segments of maximizing user satisfaction. The information available to us is the properties defined in Section 3.1 and the data constraint defined by the user. By comparing user satisfaction to utility score we define our objective as: maximize the total utility score of a set of notifications given a data consumption constraint. The selector does this through picking a subset of notifications as displayed in Figure 3.2

Sending a notification multiple time has no increased value, if a user would like to reuse notification content the device should be responsible for storing the content. When selecting a notification we should select it in full. This removes the requirement for notifications to support partial transmission, which is hard to enforce when we do not know the exact content on the notifications. For example, a video notification could support transmitting only its audio or image components. However, for some notifications, this could be crucial and so this requirement appears unforgiving. Maximizing utility score by picking full items under constraints can be modelled as a 0–1 knapsack problem.

Given n notifications we denote s_i and c_i as the utility score and cost of notification i respectively. The total data constraint is denoted C. The knapsack problem is defined as:

$$\text{Maximize:} \quad \sum_{i=1}^{n} s_i x_i \tag{3.1}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n} c_i x_i \leqslant C \quad \text{and} \quad x_i \in \{0, 1\} \tag{3.2}$$

The 0–1 knapsack problem is a well known, and well researched NP-complete problem [11]. Multiple dynamic programming algorithms have been proposed for its solution, solving the problem in pseudo-polynomial time [11]. The problem can also be approximated by greedy algorithms in $O(n)$ time, often providing a heuristic solution given relatively small notification costs.

3.6.1 Analysis

Greedy algorithm

In Section 3.6 we claim that the greedy algorithm for the 0–1 knapsack problem is a heuristic solution when the relative size of notifications is small. Below we use the fractional knapsack problem and its easier to calculate an optimal solution to how the relative size of the notifications, measured as the data cost of the highest data cost divided by the total capacity, dictates the worst case error of the greedy solution to the 0–1 knapsack problem.

The fractional knapsack problem has a greedy solution ([12], pp. 426), that is produced by adding the elements that provide the highest utility score per cost. When the remaining capacity of the knapsack is not sufficient to grab the next element, the algorithm fills the remaining capacity with a fraction of the next element. The fractional knapsack problem is defined as:

$$\text{Maximize:} \quad \sum_{i=1}^{n} s_i x_i \tag{3.3}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n} c_i x_i \leqslant C \quad \text{and} \quad x_i \in [0,1] \tag{3.4}$$

Clearly the 0–1 knapsack problem has stricter constraints than the fractional knapsack problem, thus any solution that satisfies the constraints of the 0–1 knapsack problem is also valid under the constraints of the fractional knapsack problem.

Lemma 3.1. Given the same parameters and elements, the set of possible solutions to the 0–1 knapsack problem is a subset of the solutions to the fractional knapsack problem.

Defining the set of solutions that satisfy the requirements of the fractional knapsack problem as P_f, and the set of solutions that satisfy the requirements of the 0–1 fractional knapsack problem P_b. Lemma 3.1 gives

$$P_b \subset P_f \tag{3.5}$$

Corollary 3.1. Given the same parameters and elements, an optimal solution to a fractional knapsack problem will be better than or equal to an optimal solution to the 0–1 knapsack problem.

The greedy solutions to both of these problems act similarly until the cost of the next element to be added exceeds the available capacity in the knapsack. This next element that does not fit in the greedy solution to the 0–1 knapsack problem is hereafter referred to as x.

In a worst case scenario, the difference between the cost of x, c_x and the capacity remaining, c_r is infinitely small. Thus $c_x \approx c_r$ while also $c_x > c_r$. The 0–1 knapsack problem would not be able to add x and, in the worst case would not add any further elements leading to the solution Q. A fractional problem with the same elements and constraints would add $\frac{c_r}{c_x} * s_x$ where s_x is the utility score of x. As $c_x \approx c_r \Rightarrow \frac{c_r}{c_x} \approx 1$ the optimal solution to the fractional knapsack problem would be $Q + s_x$. We define the optimal solution to the 0–1 knapsack problem as Q_b. Due to corollary 1 we have $Q_b \leqslant Q + s_x$ thus the difference may not exceed s_x.

$$Q_b - Q \leqslant s_x \tag{3.6}$$

Lemma 3.2. Given the same parameters and elements, the difference between the optimal solution and the greedy solution to the 0–1 knapsack problem may not exceed the utility score of the next greedy element that does not fit.

The worst case would also assume that this element x would be the largest element in the collection, as that would be the element that carries most utility score overall. This would maximize s_x.

We define s_a as the sum of all the scores added to the knapsack in the 0–1 knapsack problem, and c_a as the sum of the costs added to the same knapsack. As the greedy algorithm prioritizes elements with the highest $\frac{score}{cost}$ we have that

$$\frac{s_a}{c_a} > \frac{s_x}{c_x}$$

which leads to

$$\frac{c_x}{c_a} > \frac{s_x}{s_a} \tag{3.7}$$

We define the total capacity of the knapsack as c_t. As c_a is the sum of all the added elements we see that $c_t = c_a + c_r$, thus $c_a = c_t - c_r$. We mentioned above the worst case $c_x \approx c_r$, and with this substitute in Equation 3.7 to:

$$\frac{c_x}{c_t - c_x} > \frac{s_x}{s_a} \tag{3.8}$$

Reducing the relative cost of the elements in the collection we have from Equation 3.8:

$$\lim_{\frac{c_x}{c_t} \to 0^+} \frac{c_x}{c_t - c_x} = 0 \Rightarrow \lim_{\frac{c_x}{c_t} \to 0^+} \frac{s_x}{s_a} = 0$$

Ergo using Lemma 3.2 as we reduce the relative cost of the largest element in the collection we also reduce the worst case relative difference between the optimal solution and the greedy solution.

3.7 Scheduling

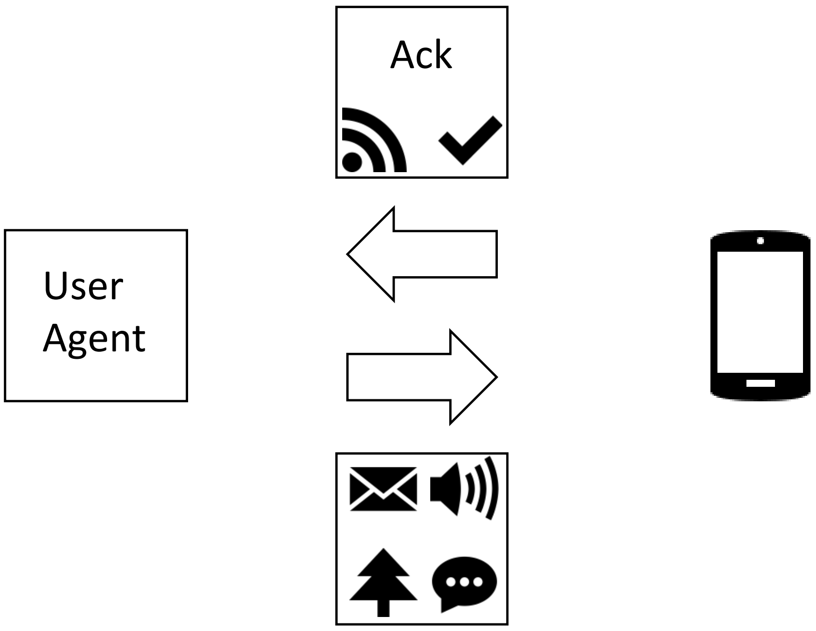

Figure 3.3: The user agent transmits a notification to a device. Upon success the device responds to the service with an acknowledgement.

The scheduling of notifications also affects efficiency. Batching notifications allows for fewer messages, and prioritizing high-value messages increases system performance during abrupt state changes such as disconnects. It also defines the frequency of updates. affecting how the users interact with their notifications. This can affect how users view the responsiveness of our system and interact their notifications.

As the user may define a power constraint that is not considered in the selection module it is possible that the receiving device meets the power constraint before it is able to deliver the complete selected optimal. This can sometimes lead to situations where the ordering of the notifications plays a decisive role in the total notification utility score collected by the device.

However because of the issues revealed in Section 3.3 it is not certain that a mobile device will have the required battery power. We can assume larger notifications will require more power to transmit. Hence, we send the notification in a greedily sorted order, sending the most data efficient notifications first. As the notifications to be sent has already been decided and will be transmitted, this will deliver the most power efficient notifications first providing maximized throughput with the knowledge we have.

The system uses acknowledgement messages to know if the recipient successfully received the data and if the user is still alive to receive a new message. This interaction is shown in Figure 3.3

Chapter 4

Implementation

This chapter serves to present and clarify the specifics of the entire solution as needed to solve the related practical issues. Its purpose is to explain the modules and their purposes to such detail that an entity could implement it using the programming tools and services of their choice. It does therefore not present code snippets as we fear it would be too limiting for implementation in potential other tools. All code used to form our solution is however attached to this thesis if specific answers are requested.

The implementation of our service is achieved as an extension of the design in chapter 3. By implementing its notification, selection, and scheduling techniques we can achieve the base of the service. We use Spotify as our media notification provider and add Spotify API, Spotify data, Spotify utility scoring, and a user agent as separate modules to produce the modular design as displayed in Figure 4.1.

By generalizing as many of the modules as possible the proposed solution becomes more adaptable and able to provide functionality for multiple social media platforms. The implementations of the selector, scheduler and code for transmission, therefore, accept the generic kind of notification. The Spotify module is Spotify specific and would be specialized for each new provider added.

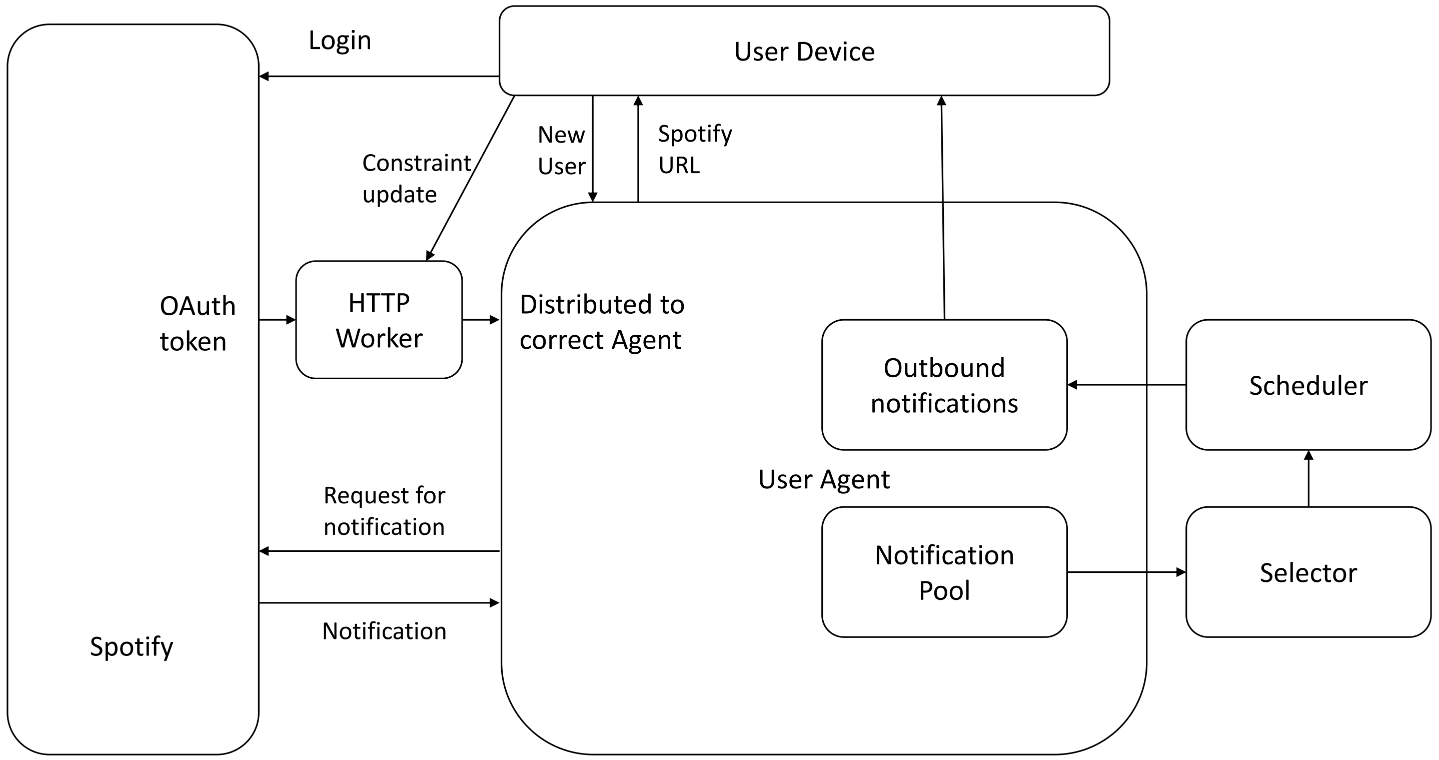

Figure 4.1: The structure of our modular design. The User Agent is the main body maintaining the user's state. Upon initialization, it returns a login URL the device. When the user logs in using this URL the service delivers an access token to our service. The server, at set intervals, polls the notification provider for new notifications. Upon discovery of items, all the outbound notifications are delivered to the selector. Through selection and scheduling, we proceed to deliver the calculated optimal notifications. Unsent items are returned to the pool.

4.1 Implementing a Generic Notification

We aim to implement a generic notification structure supporting the requirements of many different media notification providers. The notifications must work to appease the design explained in Chapter 3, to satisfy the demands of our notification service. With this in mind, we define notifications as an abstract class, specifying that all inheritance members must have: a utility score, a data cost, and a `pushnotification` defining the notification content to be pushed to the device. The `pushnotification` only contains data that is relevant to the mobile device and is the data that is pushed to the device. The `pushnotification` should store this data as text and abide the rules set by Firebase.[1] We also require that each notification is built with a unique resource identifier(URI). This carries no weight in how the system delivers notifications but is used for practical reasons.

To support notifications from multiple providers and to support internal functions for the notifications we set no other requirements for them. These standard items are the ones that are used by the other modules of our service, enabling seamless traversal through the system.

4.2 Deriving User Constraints

User information is handled by agents. By creating user agents with basic information we can ease the notification process for both parties. It removes the requirement for users to log in for services with every batch and reduces the number of notifications that have to be created. This is because notifications that are not selected are stored for the next set. User agents are responsible for storing all data related to a unique user. By running their own threads they look for new notifications destined for users and maintain these users' current constraints. The details of user agents are covered in Section 4.4

The solution receives semi-frequent updates for the user's device, with the longest period being hourly. These updates contain all the important content for the user agent. The mobile device is also expected to send a constraint update message with every received notification. This works as the scheduling acknowledgement, while also keeping the system accurate and up to date on new parameters. The device

[1] https://firebase.google.com/docs/reference/fcm/rest/v1/projects.messages

application can control data usage by delivering this constraint. By updating the service on what resources are available at the time, we increase the accuracy of the constraint numbers. This is due to the mobile device estimations for data use being more accurate than the estimation provided by the service. These updates also facilitate adaptation to alterations in the user's preferences. This has the benefit of giving the device application control over partitioning the capacity constraints, e.g. if a user wants to set a limit per month, the mobile device can divide this constraint into days or weeks to ensure even spread of performance over time. As these updates are used so frequently to increase accuracy and acknowledge successful deliveries of notifications, we consider this the simpler way to implement long-term constraints. The strongest alternative is providing the service with an upper limit and how often this limit is reset. This alternative can suffer from time drift and inaccurate measures over time.

We use the battery power as a hard cap on when the device is available to deliver updates. The resource updates resulting from sent notifications and from hourly updates are used as signs of device status. When a user agent detects its device has become unavailable, it too will become inactive until the connected device shows activity. This prevents the service from attempting to push to the device.

This is important because continuous pushing to a device that does not acknowledge could still drain battery life if the service is disabled by the power capacity. Also the agent would continue to use server resources to poll for notifications. While the server does not have limited resources itself it is limited by Spotify's DDOS prevention, which only allows a set amount of requests per second, thus sleeping agents could increase througput for active agents.

4.3 Selection

The selection module is responsible for selecting the final notifications destined for the device. Its intent is to maximize the notification utility score under a constraint. The instance of the module is initialized with the full of notifications intended for a user. After running its algorithm it returns the subset of notifications that it has found to return optimal value.

As mention in Section 3.6 we solve selection through solving the 0–1 knapsack problem. We implement the solution over two different algorithms for performance comparison.

The greedy algorithm for solving the 0–1 knapsack problem is not guaranteed to find the optimal solution. It is however incredibly resource efficient, running at $O(n \log n)$ For the greedy algorithm, we calculate each notifications efficiency by $\frac{UtilityScore}{Byte}$. These elements are sorted and we add the highest efficiency items first. In the case where the next element is too large to fit under the constraint, we continue to the next element and to the same until we have cycled through all the elements.

The dynamic algorithm for solving the 0–1 knapsack problem requires a large amount of memory and time, as it runs at a complexity of $O(nw)$. The technique is centred around solving for the optimal solution for each possible constraint, for each item, working from 0 to the actual constraint. Each time we add an item the optimal constraint for that amount of items at that constraint will be either the optimal for that constraint without the item or, adding the utility score of the item and the optimal for the remaining constraint.

4.4 User Agent

The user agent is the main storage structure for user unique data. It stores contact tokens, updates notifications on behalf of users and is responsible for all manners of user-specific operations. It distributes the user information to instances of the other modules and uses their results to an improved experience to its designated individual. Because of all the control structure, it plays a very important part in implementation and functionality of our service. It is however not very advanced in its complexity and thus is not scrutinized in this thesis.

User agents are initialized by delivering a Firebase token to the service. The system replies with login URLs used to log in for the different supported notification services. For Spotify, this URL specifies the service, the callback URL, and an identifier state variable to separate users. The agents serve notifications to the users based on what services are logged in.

4.5 Spotify

To produce a realistic proof of concept, we implement the provider-specific modules of the system with Spotify as our notification provider. Spotify is a music streaming service with over 170 million active users.[13] Spotify is heavily invested in a social listening experience. Users can share their experiences with their friends and follow each other's activities. Spotify has also combined solutions with Facebook and Instagram creating a social music sharing experience. With its massive user pool and immense artist library, Spotify is equipped with tools for generating millions of notifications originating from a plethora of artists.

In an attempt to branch out to developers and designers Spotify maintains a large solution library, giving access to tools ranging from user operations to web integration. These tools help developers produce custom solutions, like our notification system. For our implementation, we use Spotify's web API. [2] This gives us access to metadata for Spotify's public library of music, and access to various user information. This will be our source for notification generation. The Spotify API uses OAuth2 [3] for user login and to provide services with user data.

4.5.1 Generating Notifications

When a user logs in with their Spotify credentials, the service starts checking regularly for new content from the users followed artists. The act of "Following" is a function in, and made by, Spotify. It allows users to use their three main sources of content: artists, users, and playlists with greater effect. By "following" such a content source the users make a statement that they are interested in the related music. By using this for our notification generation we guarantee that the content created is content that the user has interest in. We use the artist follow list for simplicity. When a user logs in to our service through Spotify, access to this list is requested. This list forms the source of our notification generation.

The notification content of interest to the user is generated by collecting relevant metadata; artist, song title, album name, and a URL to download a preview. These are all sent to the device. Spotify does not provide direct access to their raw media data. Because of this, we use the available music preview as the main media of our

[2] https://developer.spotify.com/documentation/web-api/
[3] https://oauth.net/2/

notification. Because of the size of such a preview, the audio file itself is not sent, and the device is expected to use the URL to download the file.

Scoring of Spotify's Notifications

Spotify's items are scored per track. The tracks are utility scored on a set of feature that provides insight into the general reception of the track. The set of features that are the base of a tracks utility score follows: the hour of the day, track popularity, album popularity, artist popularity, track duration, and artist followers.

If an album is released, we utilize the notification as if it was a single track. Aiming to provide a user with an introduction to an album that they can afterwards seek on their own. Such a media notification would not carry the entire media to be played, but advertise an albums release and short introduction to its style.

The data laying the foundation for the model was collected by analyzing Spotify notifications that were hovered over, to only account notifications that were noticed by the user. Notifications that were ignored were counted as negatives, while notifications that were activated counted as positives. If the hovered over requirement was not present the dataset could contain a large number of negatives from users who never considered the notifications in the first place. We would have no indication of a users preference of such notifications and the data would be skewed. Disclosure: the dataset utilized is the same as collected in 2015 for the RichNote paper [3].

The data is used to train a random forest classifier in Python3.6. This model is implemented using the machine-learning library "Scikit-learn"[4]. The classification models from Scikit-learn support model training, predicting class a set of features, and provide the probability of each class. As mentioned above one of our classes is whether a notification was activated. We can satisfy the requirement from Section 3.2 by using the probability feature in scikit-learn and deliver a probability that a notification will be activated.

[4]http://scikit-learn.org/stable/index.html

Data Cost of Spotify's Notifications

To satisfy the benefits of Section 3.4 we aim to accurately measure the data byte size of a notification. As mentioned above we do not send the media file itself. We are however required to download the item, so that we may get an exact measure of its size. Upon generating the notification containing all the metadata to be sent, we combine the size of the metadata and the size of the media preview to create the full data cost of the notification.

4.6 Device Interaction

For a device to receive and respond to a message from a server an active connection is required. The active connection would routinely send keep-alive packets and await messages from the server. This solution was commonly, before 2009, implemented on a service to service basis. This caused devices covering many services to maintain a large number of active connections with very low utility. In 2009 Apple launched the first push notification system for iOS called Apple Push Notification System.[5] This system provides an increased efficiency by utilizing a single open connection for all server messaging communication to the device. This provides a scalable solution where the cost of adding a new service is only the cost of the transmissions of that new service. Google provides a similar system through Firebase Cloud Messaging(FCM)[6] supporting both iOS and Android.

In our system, we use FCM for message delivery to the device. By sending data notifications to a Firebase app-specific URL, FCM controls that the notification is received at the correct device, and invokes the correct handlers in the corresponding app. The overall result is an efficient and convenient pipeline that delivers the notification data to the mobile devices. Firebase data notifications are limited to 4KB per notification. This data limit is circumvented by delivering a URL for the device to download media content above 4KB.

[5] https://developer.apple.com/notifications/
[6] https://firebase.google.com/products/cloud-messaging/

4.6.1 HttpWorker

The HTTPWorker is the module responsible for all inbound communication. It provides an HTTP endpoint that is known to all app instances and Spotify. The worker is responsible for the reception of both constraint updates and new service tokens. For these reasons it has knowledge of all active user agents and transmits both types of messages to the correct agent. The worker runs on its own thread to provide better uptime and faster responses. For constraint updates it supports JSON message structures as shown in Table 4.1

Table 4.1: Displays and explains the fields available in the messages received by the server. These messages are expected as POST requests.

Field	Values	Explanation
msg_type	"new_user", "update"	*Required* Designates the purpose of the message.
device_id	string	*Required* Firebase token, used to identify the origin of the message.
data_limit	integer	*Required* The current data limit in bytes.
battery_limit	integer from 0–100	*Optional* Battery percent at which to halt notifications.
battery_pct	integer from 0–100	*Optional* Current battery percent.
wifi	boolean	*Optional* Current device WiFi status

Chapter 5

Experiments and Evaluation

To test the performance of our system we implement a FIFO selector, selecting notifications by date. This is comparable to how a system would operate without selection optimization, where notifications are sent to the device as they arrive. We compare the utility score produced by the selectors over a range of constraints and compare the computational time used to produce these results.

5.1 Experimental Setup and Datasets

The hardware used for the computational server experiments are provided by Google Cloud Platform.[1] The machines run 4 virtual CPU cores and support up to 26GB of RAM. The service running on this platform is made in .NET core 2.0. We use a University of Stavanger campus machine to train and host our Random Forest model, as the memory requirements for the model is too great for the available Google Cloud machines.

The Spotify notification scorer used for the experiments is trained with the same as dataset specified in Section 4.5.1. The dataset was collected from 1st January 2015 to 7th January 2015 using Spotify's notification system at the time. The dataset contains features of the notifications sent to users and their responding actions. The actions are only stored for users who hovered over the notification, and documents whether the notifications were clicked or hovered over and subsequently unused. The dataset contains 2312953 entries.

[1]https://cloud.google.com/

To create notifications for testing purposes we log in with an account. Upon creating a new user agent we replace the current date with an artificially old date. This has the effect of producing notification for each new release since that date, for each artist the logged in user follows. This simulates a known user who has been unavailable since this date. For our tests the date in question is 01/01/2015, the tests were run 10/06/2018 producing over three years worth of notifications. We do this for two separate users. The generated notifications are utility scored and data cost measured to provide two data sets of fully functional notifications.

To produce a baseline we implement a FIFO-selector. The FIFO selector sorts the notification elements by their content release dates to produce its priority list. It will select the oldest notifications suitable for the data constraint. Similar to the greedy selector, if a next element will not fit under the current constraint it will halt instantly, but attempt the next element in the list until the list has been exhausted. The full implementation of these selectors may be found in Appendix A.

The two notification sets are fed to all three selectors with constraints ranging from 100KB to 20MB in steps of 200KB. The upper bound has been selected to ensure that the Google Cloud machine does not run out of RAM due to dynamic calculation's high memory cost. These tests produce 300 sets of selected notifications for each dataset. These values are stored in CSVs in the format specified in the list below.

1. Constraint
2. Total utility
3. Total data cost
4. Number of notifications in selection
5. Calculation time

The two result sets are aggregated by averaging the results for each constraint value. This results in three CSV files representing the results for each of the three selectors.

5.2 Results

We test our Random Forest model using five-fold cross validation in sci-kit and achieve 0.75 accuracy with less than 0.01 variance.

Figure 5.1 displays the time for each selector to make a complete selection measured in milliseconds, and how they evolve as the size of the constraints grow.

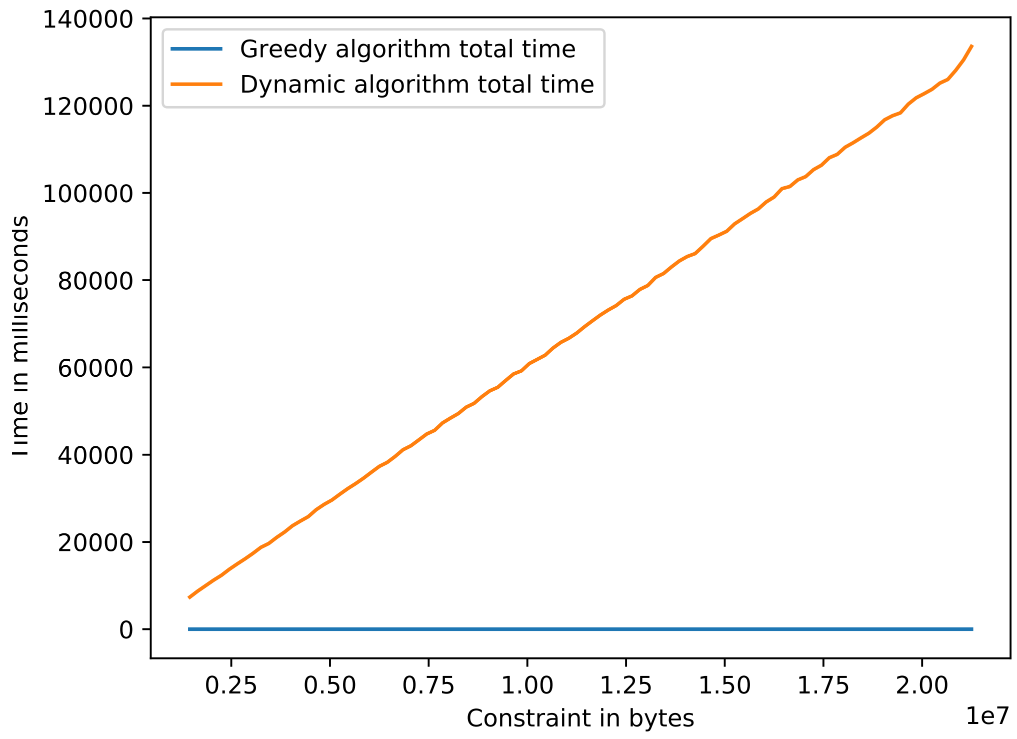

Figure 5.1: Displays the time in milliseconds for the selectors to complete their calculations

Figure 5.2 shows the increase in utility score as the sum of selected notifications for the constraint level and allows for easy comparison of the performance of our different selectors.

Figure 5.3 displays ratio of greedy utility score to dynamic utility score, and how this evolves as the constraint increases.

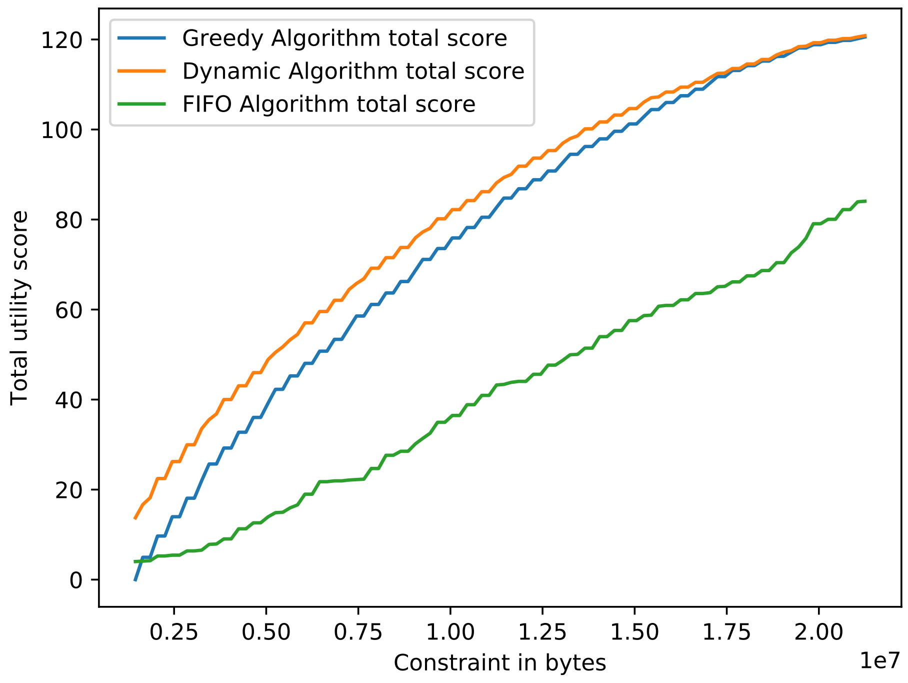

Figure 5.2: Total utility score accumulated over tested range. The x-axis displays the constraint

5.3 Analysis

Figure 5.1 shows the dynamic selector rapidly increase towards impractical times. With many users, the dynamic selector risks not being able to deliver the throughput required to serve a reasonable userbase and therefore does not provide the required scalability. This inadequacy is made more distinct by the greedy algorithms very stable performance.

From the tests, it appears that the utility score difference in dynamic and greedy selection is minute. Due to greedy selections far superior computational time, we conclude that it provides higher efficiency. We do not, however, have enough data to solidify this statement. For us to gather more data we would have required an increase in tested accounts. Because of this being costly and requiring these accounts to cover a large variety of followed artists we can estimate that our solution is quite poor at generating test data. However, if we were to conclude based on the data we have it appears that greedy selection is sufficiently effective at producing an advantage in utility score when compared to FIFO.

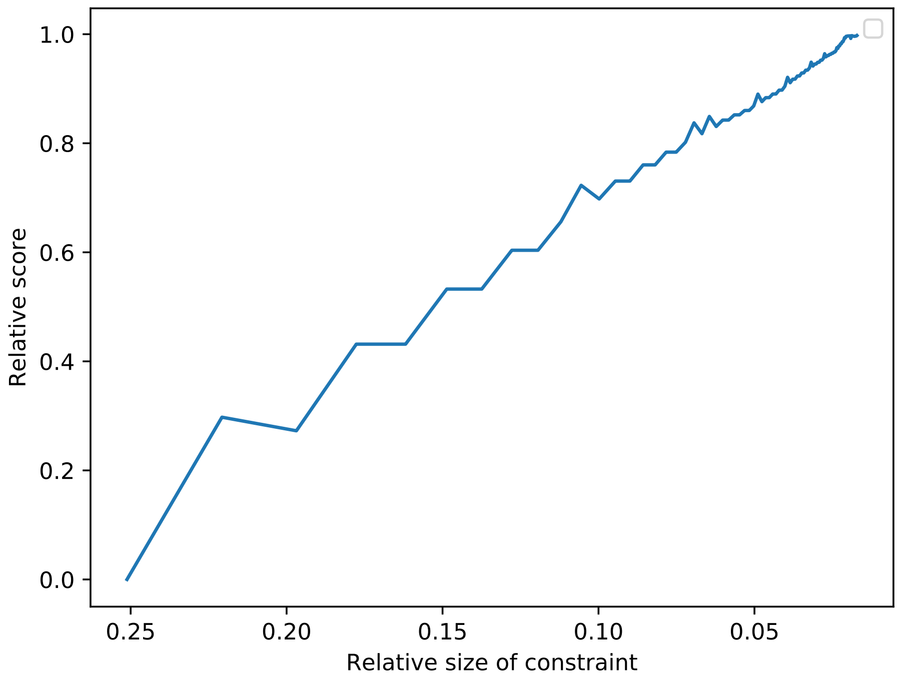

Figure 5.3: Displays the relative similarity of dynamic and greedy selection utility scores.

Chapter 6

Conclusion and Future Work

In this thesis, we set out to research and implement a system capable of actively engaging in the way users receive notifications. This system would have to reduce the total resource consumption of such notifications while maintaining a high degree of user satisfaction with the content.

We implement a working version of well-established techniques and problem solutions. Our solution relies on our users to provide the constraint limits they desire. Using these limits we prioritize high-value media notifications as resolved by our machine learning Random Forest model. This model is trained on a large amount of data providing good estimations of users priority of the notifications. As a result, we deliver media notifications to constrained devices that support a large probability of use.

We test our performance when the total set of notifications grows by comparing our system to common notification behaviour, and conclude that our solution has a high probability of resulting in increased notification activation while adhering to users desired constraints.

6.1 Future Work

6.1.1 User Specific Scoring

Music, like all other art forms, affect people differently. Some are very adamant about the music they like, and maybe more so about the music they dislike. For example, pop, enjoyed and danced to by millions, may cause disgust in others. Different users may, therefore, react differently to similar notifications. This can give listeners with negative relations towards mainstream music a feeling of dissatisfaction with the service.

To reduce this effect we only generate notifications from content that users follow. That way a user can maintain some control over what notifications they receive by unfollowing an artist with continuously disliked content that scores high for the general model.

Our scoring technique, explained in Section 4.5.1 has no user parameters and all users will calculate the same utility score for the same notification, given the same time of day. This results in two users with the exact same set of outbound notifications and same constraints receiving the same notifications

Recently there has been a lot of research into more dynamic machine learning modes that could enable our system to implement a unique model per user. This could help the service appeal to individual and produce better individual performance.

6.1.2 Presentation Levels

Uddin et al. [3] suggests that notifications are divided into multiple presentation layers, where each higher level layer has higher quality content and higher data cost, and the lowest level is to not select the notification. In such a system, a single presentation layer from each notification would be chosen by the selection algorithm.

This results in a multiple-choice knapsack problem. Defined as a knapsack problem with N classes, each class contains multiple items and we are tasked with selecting one item from each class to maximize the value. In this analogy, each notification is a class, and we are tasked with selecting a single presentation level from each class.

$$\text{Maximize:} \quad \sum_{i=1}^{|N|}\sum_{j=i}^{|N_i|} s_{ij}x_{ij} \tag{6.1}$$

$$\text{Subject to:} \quad \sum_{i=1}^{N}\sum_{j=1}^{N_i} c_{ij}x_{ij} \leqslant C \tag{6.2}$$

$$x_{ij} \in \{0,1\} \tag{6.3}$$

$$\sum_{x \in M} x = 1, \forall M \in N \tag{6.4}$$

If each notification only supports the two layers: to select an item in full or not to select the item at all, it becomes a 0–1 knapsack problem with only two x_{ij}. However, any notification supporting more than two layers would provide the greedy solution with an extra, smaller in cost option. Thus completing the greedy solution not by choosing the best notification but the best upgrade, would produce a better result due to its higher resolution. This is both because it reduces the relative size of the items to be selected and because where a 0–1 knapsack greedy solution would run out of space, it is possible that a multiple-choice knapsack may be able to fit more items. Because it deals with different elements, it is also plausible that some solutions would outperform the dynamic solution to the 0–1 knapsack problem.

Implementing such a system is not however without its limitations. According to Hifi, Michrafy, and Sbihi [14] the added complexity of the multiple-choice knapsack problem makes the optimal solution "...not suitable for most real-time decision-making applications..." hence we lose the ability to calculate this optimal solution.

A system with such an implementation would allow notification selection to perform better. However Spotify does not support multiple playback qualities, and the results of such an implementation would be notifications with only two presentation layers. The greedy selection algorithm would produce the same output as it does for single layer notifications.

Implementing media manipulation would allow us to realize such a system and increase performance. Research into how much of an advantage and how practical it is to implement would be an interesting way to advance the solution.

List of Figures

Appendix A

Selectors

Listing A.1: Greedy selector

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.IO;

namespace master.Selection
{
    class GreedySelector : Selector
    {
        private List<INotification> sortedArray;

        public GreedySelector(List<INotification> unsortedNotifications, long dataconstraint) :
        {
        }

        public override INotification[] getSelection()
        {
            orderNotifications();
            long totalData = 0;
            int i = 0;
            var retVal = new List<INotification>();
            while(totalData < dataConstraint && sortedArray.Count > 0)
            {
                var next = sortedArray[i];
                i++;
                if (i >= sortedArray.Count)
                {
                    break;
                }
                if (totalData + next.DataCost > dataConstraint)
                {
                    continue;
                }
            retVal.Add(next);
            totalData += next.DataCost;
        }
        /*
        foreach (var elem in retVal)
        {
            Console.WriteLine($"greedy selection: {elem.Uri}, {elem.Score}, {elem.DataCost}");
        }
        */
        double totalscore = retVal.Sum(elem => elem.Score);
        return retVal.ToArray();
    }

    public override INotification[] nextBatch()
    {
        if (sortedArray.Count <= 0)
        {
            throw new SelectionException();
        }
        var retVal = sortedArray[sortedArray.Count-1];
        sortedArray.RemoveAt(sortedArray.Count - 1);
        return new INotification[] { retVal};
    }

    public void orderNotifications()
    {
        var items = new List<INotification>(notifications);
        items.Sort(compareNotificationsData);
        this.sortedArray = items;
    }

    private int compareNotificationsData(INotification notX, INotification notY)
    {
        try
        {
            double totalX = notX.Score / notX.DataCost;
            double totalY = notY.Score / notY.DataCost;

            if (totalX > totalY)
            {
                return -1;
            }
            else if(totalX == totalY)
            {
                return 0;
            }
            return 1;
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.StackTrace);
        }
        return 0;
    }
    }
}
```

Listing A.2: Dynamic selector

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace master.Selection
{
    class DynamicSelector : Selector
    {
        private int[,] M;
        private bool[,] keep;
        private INotification[] relevantNotifications;

        public DynamicSelector(List<INotification> unsortedNotifications, long dataconstraint) :
        {
        }

        public override INotification[] getSelection()
        {
            DP();
            var retList = new List<INotification>();
            long k = dataConstraint;
            for (int i = relevantNotifications.Length - 1; i >= 0; i--)
            {
                if (keep[i, k])
                {
                    retList.Add(relevantNotifications[i]);
                    k -= relevantNotifications[i].DataCost;
                }
            }
            Console.WriteLine($"Capacity: {dataConstraint}, best score: {retList.Sum(elem => ele
            /*
            foreach (var elem in retList)
            {
                Console.WriteLine($"Dynamic selection: {elem.Uri}, {elem.Score}, {elem.DataCost}
            }
            */
            return retList.ToArray();
        }

        private void DP()
        {
            relevantNotifications = notifications.Where(notification => notification.DataCost <
            var nots = relevantNotifications;
            if (dataConstraint == 0 || nots.Length == 0)
            {
                return;
            }
            Array.Sort(nots, notificationComparer);
```

```
var old = new double[dataConstraint + 1];
var curr = new double[0];
keep = new bool[nots.Length, dataConstraint + 1];
/*
for (int j = 0; j <= dataConstraint; j++)
{
    M[0, j] = 0;
}
*/

for (int i = 0; i < nots.Length; i++)
{
    try
    {
        curr = new double[dataConstraint + 1];
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
    }
    //Console.WriteLine(GC.GetTotalMemory(true));
    try
    {
        for (int j = 0; j <= dataConstraint; j++)
        {
            // Special case to avoid index out of range
            if (i == 0)
            {
                if (nots[i].DataCost > j)
                {
                    curr[j] = 0;
                    keep[i, j] = false;
                }
                else
                {
                    curr[j] = nots[i].Score;
                    keep[i, j] = true;
                }
                continue;
            }
            if (nots[i].DataCost > j)
            {
                curr[j] = old[j];
                keep[i, j] = false;
            }
            else
            {
                double opt1 = old[j];
                double opt2 = old[j - nots[i].DataCost] + nots[i].Score;
                if (opt1 > opt2)
                {
                    curr[j] = opt1;
                    keep[i, j] = false;
                }
```

```
                        else
                        {
                            curr[j] = opt2;
                            keep[i, j] = true;
                        }
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                System.Environment.Exit(0);
            }
            old = curr;
        }
    }

    private int notificationComparer(INotification notX, INotification notY)
    {
        return notX.DataCost - notY.DataCost;
    }

    public override INotification[] nextBatch()
    {
        throw new NotImplementedException();
    }

  }
}
```

Listing A.3: FIFO selector

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.IO;
using Newtonsoft.Json.Linq;

namespace master.Selection
{
    class FIFOSelector : Selector
    {
        private List<INotification> sortedArray;
        private SpotifyData data;

        public FIFOSelector(List<INotification> unsortedNotifications, long dataconstraint, Spot
        {
            this.data = data;
        }

        public override INotification[] getSelection()
        {
            orderNotifications();
            long totalData = 0;
            int i = 0;
            var retVal = new List<INotification>();
            while (totalData < dataConstraint && sortedArray.Count > 0)
            {
                var next = sortedArray[i];
                i++;
                if (i >= sortedArray.Count)
                {
                    break;
                }
                if (totalData + next.DataCost > dataConstraint)
                {
                    continue;
                }
                retVal.Add(next);
                totalData += next.DataCost;
            }
            /*
            foreach (var elem in retVal)
            {
                Console.WriteLine($"greedy selection: {elem.Uri}, {elem.Score}, {elem.DataCost}");
            }
            */
            //double totalscore = retVal.Sum(elem => elem.Score);
            return retVal.ToArray();
        }

        public override INotification[] nextBatch()
        {
            if (sortedArray.Count <= 0)
            {
                throw new SelectionException();
            }
            var retVal = sortedArray[sortedArray.Count - 1];
            sortedArray.RemoveAt(sortedArray.Count - 1);
            return new INotification[] { retVal };
        }

        public void orderNotifications()
        {
            var items = new List<INotification>(notifications);
            items.Sort(compareNotificationsDate);
            this.sortedArray = items;
        }

        public DateTime GetReleaseDate(INotification not)
        {
            JObject notObj = data.GetSpotifyResource(not.Uri);
            string dateString = "";
            string precision = "";
            switch (not.Uri.Split(':')[1])
            {
                case "album":
```

```
                dateString = notObj["release_date"].ToString();
                precision = notObj["release_date_precision"].ToString();
                break;
            case "track":
                dateString = notObj["album"]["release_date"].ToString();
                precision = notObj["album"]["release_date_precision"].ToString();
                break;
        }

        int year = 1;
        int month = 1;
        int day = 1;
        switch (precision)
        {
            case "year":
                year = int.Parse(dateString);
                break;
            case "month":
                string[] monthSplit = dateString.Split("-");
                year = int.Parse(monthSplit[0]);
                month = int.Parse(monthSplit[1]);
                break;
            case "day":
                string[] daySplit = dateString.Split("-");
                year = int.Parse(daySplit[0]);
                month = int.Parse(daySplit[1]);
                day = int.Parse(daySplit[2]);
                break;
        }

        return new DateTime(year, month, day);
    }

    private int compareNotificationsDate(INotification notX, INotification notY)
    {
        try
        {
            DateTime dateX = GetReleaseDate(notX);
            DateTime dateY = GetReleaseDate(notY);

            if (dateX < dateY)
            {
                return -1;
            }
            else if (dateX == dateY)
            {
                return 0;
            }
            return 1;
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.StackTrace);
        }
```

```
            return 0;
        }

    }
}
```

Bibliography

[1] ICT Data and Statistics Division. Ict facts & figures 2015. PDF available online at https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf, 2017.

[2] gartner. Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017. *Gartner*, May 2017. URL https://www.gartner.com/newsroom/id/3725117.

[3] Md Yusuf Sarwar Uddin, Vinay Setty, Ye Zhao, Roman Vitenberg, and Nalini Venkatasubramanian. Richnote: Adaptive selection and delivery of rich media notifications to mobile users. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 159–168. IEEE, 2016.

[4] Ngoc Do, Ye Zhao, Shu-Ting Wang, Cheng-Hsin Hsu, and Nalini Venkatasubramanian. Optimizing offline access to social network content on mobile devices. In *INFOCOM, 2014 Proceedings IEEE*, pages 1950–1958. IEEE, 2014.

[5] Patrick TH. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. doi: http://doi.acm.org/10.1145/857076.857078.

[6] Ye Zhao, Ngoc Do, Shu-Ting Wang, Cheng-Hsin Hsu, and Nalini Venkatasubramanian. O 2 sm: Enabling efficient offline access to online social media and social networks. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 445–465. Springer, 2013.

[7] Dong Liu, Guangnan Ye, Ching-Ting Chen, Shuicheng Yan, and Shih-Fu Chang. Hybrid social media network, 2012.

[8] Samsung. Samsung mobile specification. https://www.samsung.com/uk/smartphones/, 2018. Online; accessed 4 June 2018.

[9] Apple. Apple iphone specification. https://www.apple.com/iphone/, 2018. Online; accessed 4 June 2018.

[10] Sony. Sony mobile specification. https://www.sonymobile.com/global-en/products/phones/, 2018. Online; accessed 4 June 2018.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, USA, 1979. ISBN 0-7167-1045-5.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, third edition, 2009. ISBN 978-0-262-53305-8.

[13] Spotify. Spotify company info. https://newsroom.spotify.com/companyinfo/, 2018. Online; accessed 14 June 2018.

[14] Mhand Hifi, Mustapha Michrafy, and Abdelkader Sbihi. Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *Journal of the Operational Research Society*, 55(12):1323–1332, 2004.