DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Notification Timing for a Proactive Virtual Dietary Advisor

Oguz Gültepe

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Notification Timing for a Proactive Virtual Dietary Advisor

Timing von Benachrichtigungen für einen proaktiven virtuellen Ernährungsberater

Author:	Oguz Gültepe
Supervisor:	PD Dr. rer. nat. Georg Groh
Advisor:	Monika Wintergerst
Submission Date:	15.07.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2019 Oguz Gültepe

Acknowledgments

I would like to thank everybody who who helped me on this thesis:

My supervisor Dr. Georg Groh, whose ideas shaped this thesis to a great degree.
My advisor Monika Wintergerst, who guided me through the thesis process.
My parents, who provided me with great support throughout my education.
My brother, who was the greatest influence on my choice of studying computer science.

And finally, my friends, who kept me sane during my years at TUM.

Abstract

We want to build a system that can accurately predict when a user is likely to eat. In order to accomplish this, we first build a chatbot that let's users log their eating patterns. Next we integrate this chatbot into Telegram and share the access link with some volunteers in order to test it. The chatbot collects a total of 150 entries in a duration of roughly 2 weeks. Most of these entries are from 3 users. We train seperate models on the eating patterns of these 3 users and use these models to predict when the user is going to eat again. When we test the best performing models of each user on unseen data, we get R^2 scores of 0.675, 0.869 and 0.895.

Contents

1 Introduction

1.1 Why did we build this system?

Why did we build this system?
Why did we need it?
The answer is actually rather straight forward:
Because we tend to forget about our convictions when we are hungry.
We all set dietary goals, but only a few of us follow through with them.
This is why we are impressed when we meet people who have followed through with their goals.
Because following through with your goals is actually hard.
But what if there was a system that could remind you of your goals just before you ate?
A system that could learn your dietary patterns and intevene when you need it to?
We have built this system as an answer to this question, and this thesis documents the building process.

1.2 Goal and Structure of the Thesis

The goal of this thesis is to explain the process of building and evaluating a system that learns a user's dietary patterns. We first build a chatbot that enables users to log their eating patterns. Next, we build a machine learning pipeline that takes a user's dietary data and uses it to train predictive models. Using these models we predict when the user is going to eat again. We start sending the user messages at the predicted times, and ask for feedback on the timing. The system fine tunes the timing based on this feedback.

After building the system, we ask some volunteers to use it. The volunteers use the system for some time and afterwards, we take a look at how the system performed.

In order to build this system, we first need to be familiar with some natural language processing and machine learning concepts. We explain these concepts in chapter 2. Afterwards, in chapter 3, we show how we build the system based on this information. Next, at Chapter 4, we take a look at the data collected by our system and the models trained on this data. At Chapter 5, we discuss the collected data, the user feedback on

the chatbot and the trained models. Finally, at Chapter 6, we mention some possible improvements on this work.

2 Theoretical Background

2.1 Natural Language Processing

Natural language processing (NLP) is a subfield of computer science which focuses on, as implied by the name, processing of human (natural) languages. NLP enables us to make use of copious knowledge that is expressed in natural language. (Russell and Norvig 2010b)

In this section we will be taking a look at some NLP concepts that are relevant to us. Most of the knowledge provided here is based on (Jurafsky and Martin n.d.[d]). We will start with regular expressions, powerful tools able to catch text patterns. Then we will be looking at information extraction from natural language. Finally, we will finish this chapter on dialog systems and chatbot.

2.1.1 Regular Expressions

A regular expression is formally defined as an algebraic notation that represents a specific set of strings. However, this representation is not explicit. Regular expressions denote textual patterns which are able to produce implicit sets. These patterns are useful for searching in text. A regular expression search function finds every instance of text in the corpus that belongs to the pattern's implicit set. These instances are said to be 'matched' by the pattern.

Regular expressions are supported in every computer language, word processor and text procesing tool. However, there may be differeces in how they treat certain expressions. As the knowledge provided here is based on (Jurafsky and Martin n.d.[c]), we will be treating expressions as they are shown there. Another thing to keep in mind is that this is not a comprehensive guide for regular expression. We will be looking at basics and some more complex operators we need for our specific problem.

Basic Regular Expression Patterns

Simplest regular expressions are sequences of characters. For example `snack` matches any string that contains the sequnce 'snack'. Here are some examples that would be matched by this regular expression:

"I just had a small snack."
"His username was 123snacko."
As we see, this regular expression only searches for the sequence 'snack'. Whether the found sequence actually is a word or not is irrelevant.

The period `.` is treated as a special character: the wildcard. It matches any character, so for example `r.n` matches:
"I ran 5k today."
"You better run."
Another special charcter is the question mark: `?` Adding a question mark after an element makes the element optional. For example, `hours?` matches:
"The hour arm of the clock was missing."
"I have been waiting for you for hours."
Special characters can also be used as regular characters. We just need to put a backslash `\` before the special character. For example, `Inc\.` matches:
"Monsters Inc. was a great movie."

Table 2.1: Regular Expression Basics

Regular Expression	Match	Example
`duck`	'duck'	duck
`r.ck`	'r' and 'ck' with any character in between	rock, rack
`minutes?`	'minute' or 'minutes'	minute,minutes
`\?`	'?'	How?

Square Brackets, Range and Negation

It is important to note that regular expressions are case sensetive, so `meal` matches the first example, but not the second:
"We had a tasty meal."
"Meal is ready."
This problem is solved by the use of square brackets:`[]`. Square brackets signify a disjunction between the characters inside. So `[Mm]` means either 'M' or 'm' and `[Mm]eal` matches both examples:
"We had a tasty meal."
"Meal is ready."
Square brackets can be used for regular expression that match any digit `[1234567890]` or any letter `[abcdefghijklmnopqrstuvwxyz]`. However, for such common disjunctions we can also use the dash - operator. Dash operator denotes a range: `[0-9]` matches

any digit between '0' and '9'. `[a-z]` and `[A-Z]` match any lowercase letter and any uppercase letter, respectively.

Another use of square brackets is negation. When a caret `^` is the first character inside a bracket, any character other than the ones inside the brackets is matched. `[^ab]` matches any character that is not 'a' or 'b'. `[^0-9]` matches any character that is not a digit.

It is important to note that when used outside of these contexts, both dash - and caret ^ have different meanings. Outside of the brackets, dash operator is treated as a character. When the caret occurs in the brackets after the first character, it is also treated as a character. For example, `0-9` matches the first sentence but not the second:
"Please enter a value between 0-9."
"This course is worth 8 ECTS credits."

There are some shorthands for commonly used ranges or disjunctions. `\d` is equivalent to `[0-9]` and matches any digit. `\s` matches any whitespace.

Table 2.2: Use of Square Brackets

Regular Expression	Match	Example
`[Ss]nack`	'Snack' or 'snack'	Snack,snack
`[0-9]`	Any digit	5 missed calls!
`[^abc]`	Any character that is not 'a','b' or 'c'	carb
`[3^2]`	'3','^', or '2'	n^1 equals n
`\d`	Any digit	5 missed calls!
`\s`	Any whitespace	5_missed_calls!

Disjunction and Anchors

Now, with the help of square brackets, we are capable of basic disjunction between characters. But what if we need a pattern that matches either 'meal' or 'snack'? `[mealsnack]` would not work as square brackets denote character level disjunction. What we need here is the pipe symbol `|`, the disjunction operator. `snack|meal` matches both of the following examples:
"I need a snack."
"When was your last meal"
Next, we have the anchors: special characters that can specify locations in text. For example, the caret operator ^ when used outside of brackets specify the start of a line. The dollar sign $ specifies the end of a line. So `^I|\?$` matches any sentence that either starts with I or ends with a question mark:

"I thought we were gonna eat together."
"Are you available for lunch ?
We have two more anchors that we can make use of: `\b` to match word boundries and `\B` to match non-boundries. For example, `\bsnack\b` matches the first sentence, but not the second:
"I just had a small snack."
"His username was 123snacko."
To use these anchors, it is important to understand what a word means. In terms of regular expressions, a word is defined as any sequence of digits, letters or underscores. So `\bap\b` would match 'A$ap'. As $ is neither a digit or a letter or underscore, it is treated as a word boundry.

Table 2.3: Disjunction and Anchors

Regular Expression	Match	Example
`eat|ate`	'eat' or 'ate'	I just ate.
`^\.`	Any character at the start of a line	I just ate.
`\.$`	Any character at the end of a line	I just ate.
`\b59\b`	'59' between word boundries	It costs $59.99.
`\B59\B`	'59' between non-boundries	It costs $2599.

Kleene Operators, Grouping and Precedence

There are cases where we need to match repetetive patterns. For example, we might want to match any 'hey' with an arbitary amount of 'y's. To do this, we can use the Kleene Star `*`. Kleene star means "0 or more occurrences of the preceeding element". This means `y*` matches '','y','yy' and any other number of 'y' characters. So in order to match any 'hey' with an arbitary amount of 'y's, we would use `heyy*`. This use, however, is so common that we have another operator for it: the Kleene Plus `+`. Kleene Plus means "1 or more occurrences of the preceeding element". So `heyy*` and `hey+` are functionally equal.

But what if we need to match a phrase such as as 'hahaha', which consists of an arbitary number of 'ha's? `ha+` would not work, because the Kleene operator works on the preceeding element. So how can we set 'ha' as the preceeding element?

This is where grouping comes into play. In regular expressions, grouping is done by wrapping a phrase in parantheses. From then on, the phrase inside the parantheses is treated as a singular entity by operators outside the parantheses. `(ha)+` matches any 'ha', 'haha' and any number of repetitions of the phrase 'ha'.

Grouping also comes in handy when using the disjunction operator. For example, let's assume we need a regular expression that matches 'hour', 'minute', 'hours' and 'minutes'. We ca not use `hour|minutes?`, because the disjunction is between `hour` and `minutes?` This is because the `?` is evaluated first, then the sequences and only then the disjunction. Instead, we can use `(hours|minute)s?`. This expression works since the parantheses are evaluated before all the other operators.

The collection of rules determining which operator is evaluated first is called the operator precedence hierarchy. An operator x is said to have precedence over another operator y, if x is to be evaluated before y. Regular expression operator precedence hierarchy is as follows, from highest precedence to lowest precedence:

Table 2.4: Regular Expression Operator Precedence Hierarchy

Parantheses	`()`
Counters	`* + ?`
Sequences and anchors	`seq ^I \.$`
Disjunction	`|`

Table 2.5: Regular Expression Kleene Operators and Grouping

Regular Expression	Match	Example
`b*`	0 or more repetitions of 'b'	aaabbb
`a+`	1 or more repetitions of 'a'	aaabbb
`(ab)+`	1 or more repetitions of 'ab'	aababb

Capture Groups

Let's assume we have a list of names and we want to make sure all the names occur only once in the list. The following regular expression accomplishes that:
`\b([A-Z][a-z]*)\b.*\b\1\b`
When we use parantheses for grouping, the text that matches the expression inside is stored in the memory. This stored text is referred to as a capture group. Capture groups are numbered from left to right and can be referenced by the use of a backslash, followed by the group number.

When we need to group expressions but do not want to store the matches in the memory, we can use non-capturing groups. Adding `?:` after the opening paranthesis makes a group non capturing.

2.1.2 Information Extraction

We have mentioned that massive amount of information is expressed in natural languages. The process of converting this information into a structured data format is called information extraction. In this subsection, we will be taking a look at information extraction techniques that are relevant to us.

We will start with extracting temporal expressions from text. We will continue with normalizing the extracted temporal expressions. Finally, we will end this subsection on template filling.

Knowledge provided in this subsection is based on (Jurafsky and Martin n.d.[b]).

Extracting and Normalizing Temporal Expressions

Temporal expressions are parts of text that refer to durations or points in time. Here, we will be focusing less on durations and more on points in time. Temporal expressions such as "17th of July" and "19.30" that directly refer to point in time are called absolute temporal expression. Expressions such as "last week" and "2 hours ago" that refer to points in time in relation to some other point are called relative temporal expressions.

In order to extract temporal information, we first need to find spans of text that contain temporal expressions. This task can be accomplished with the help of lexical triggers. Lexical triggers are textual phrases that imply the existence of a temporal expression. For example; yesterday, past, next or hour. It is possible to build automatas that search and find lexical triggers in text. Additional layers of automatas can be built over these in order to find the spans of temporal expressions based on the lexical triggers.

After extraction, temporal expressions need to be normalized. This process includes mapping the expressions to points in time, be it a calendar date, a time of the day or both. This is fairly simple for absolute time expressions, as they refer to points in time directly. Relative temporal expressions, however, refer to points in time in relation to the document's temporal anchor. A document's temporal anchor is the point in time in which the documents is set. This can for example be the date an article was published or the time a text message was sent. Based on the temporal anchor, relative temporal expressions can be mapped to points in time using temporal arithmetic.

Finally, the points in time, which the temporal expressions were mapped to, must be saved in a standardized way. The standards for saving temporal expressions can be seen in (*Date and time – Representations for information interchange – Part 1: Basic rules* 2019)

Template Filling

Many times we have to deal with texts describing common and stereotypical events. These events usually follow certain patterns or have certain participants with certain roles. Our knowledge of these common events may come in handy in information extraction. For example, when we hear the following:
"Would you like to join me for dinner at L'Osteria tomorrow at 8pm?"
Even if we have never heard of L'Osteria, we can assume that it is a restaurant. This is because inviting somebody for a dinner is a common event which usually includes an inviter, invitee, a restaurant and a time. In a simple way, such common events can be represented as templates with fixed slots to fill. A template for a dinner invitation would look something like Table 2.6

Table 2.6: Example Template For a Dinner Invitation

Key	Value
Inviter	You
Invitee	Me
Restaurant	L'Osteria
Time	Tomorrow 8pm

Such templates make it easy to infer details that are not explicitly mentioned, such as the fact that L'Osteria is a restaurant. The task of template filling refers to finding the events that fit particular templates and filling the associated templates with information extracted from the text.

2.1.3 Dialog Systems

In this subsection, we will be talking about dialog systems. Dialog systems are programs use natural language to communicate with the user. In this subsection, we will first go through the difference between task-oriented dialog agents and chatbots. Next we will be taking a look at modern task-oriented dialog systems: frame based dialog agents. Finally, we will end this subsection on control structures for such dialog agents. The knowledge provided in this subsection is based on (Jurafsky and Martin n.d.[a]).

Task-Oriented Dialog Agents vs Chatbots

The words chatbot and dialog agents are often used interchangably by many people. However, in the context of NLP, this is a mistake. In NLP, chatbots refer to a distinct

class of dialog agents, the other class being task-oriented dialog agents. So what is the difference between the two classes? Let's start with task-oriented dialog agents.

Task-oriented dialog agents are dialog systems build to -as the name suggest- serve specific tasks. These tasks can include for example finding restaurants, booking hotels or sending messages. Popular digital assistants such as Alexa and Siri fall under this category. Chatbots, on the other hand, are dialog agents without specific goals, built to mimic human conversation.

Here we will be focusing on task based dialog agents rather than chatbots.

Frame Based Dialog Agents

What is a frame based dialog agent? In order to answer this question, we first need to understand some concepts.

Domains are specific areas of knowledge or activity such as computer science or fishing. Domains are modelled by abstract constructs called domain ontologies. A domain ontology defines concepts and relations that belong to a domain.

We build frames based on domain ontologies. A frame is a collection of slots that accept predefined types of values. The slots of a frame defines the information that the systems needs for the task. Dialog agents that are based on such frames are called frame based dialog agents.

Table 2.7 shows an example frame for a task-oriented dialog agent designed to make reservations at a restaurant:

Table 2.7: Example Frame for a Task-Oriented Dialog Agent

Slot	Type
Restaurant	String
Number of People	Integer
Reservation Date	Date
Reservation Time	Time

Control Structure for Frame Based Dialog

A task-oriented dialog agent needs to gather appopriate data to accomplish the task at hand. In order to achieve this, dialog systems implement control structures that guide the conversation. For a frame base dialog agent, the control structure is built around the frame. These control structures are usually finite-state automatas that are hand-designed for the task.

Let's consider the example frame shown at Table 2.7. First, for each slot, we come up with an associated question that elicits an answer to fill the slot. Table 2.8 shows these questions for our example frame.

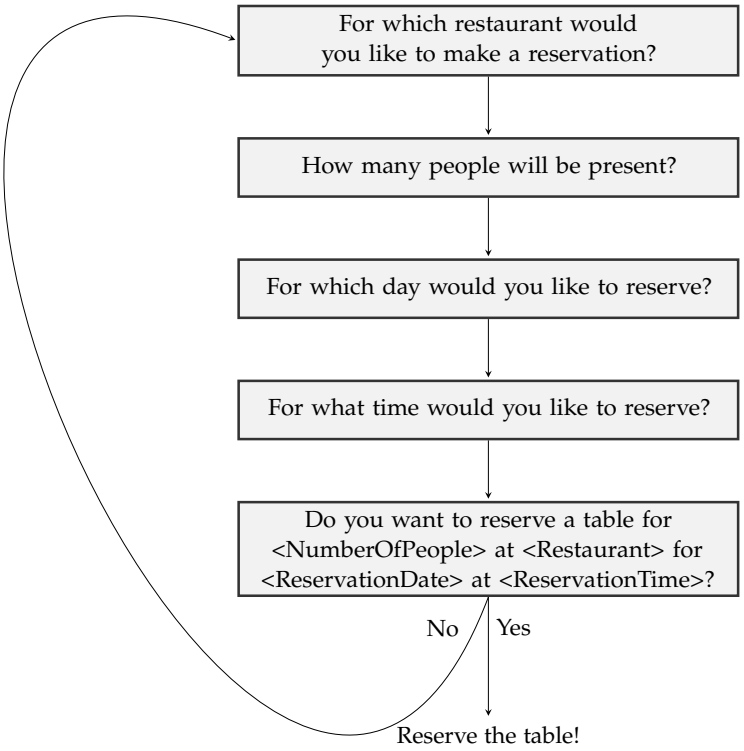
Table 2.8: Questions for the Example Frame

Slot	Question
Restaurant	For which restaurant would you like to make a reservation?
Number of People	How many people will be present?
Reservation Date	For which day would you like to reserve?
Reservation Time	For what time would you like to reserve?

Based on these questions, we build a finites-state automata to guide the conversation. In this automata, questions are the states and user's answers are the transitions. Take a look at Figure 2.1 for an example finite-state automata that is built based on the questions at Table 2.8.

In our example, the conversation is controlled entirely by the automata. The user does not have much option other than answering the questions. This means that the user has no initiative; no control over the conversation. Systems where the user has no initiative such as the one shown at Figure 2.1 are called system-initiatives. The major advantage of system-initiative architecture is the fact that the system always knows what question is being answered. This alleviates the need to match answers with slots. It also allows for fine tuning the answer processing for the expected type of answer. However, such systems offer the user no flexibility. Consider the following input from the user:
"I would like reserve a table at L'Osteria for tomorrow"
This sentence already includes the information needed to fill two slots: Restaurant and Reservation Date. It is possible to augment a dialgo system with the ability to recognize such inputs, allowing for multiple slots to be filled at once. Such a system would pose the user questions for the empty slots, and simply skip the questions for the filled ones. These systems are called mixed-initiatives, as the user also holds some inititive to control the conversation.

Figure 2.1: Example Finite-State Automata as Control-Structure



2.2 Machine Learning

Machine Learning is the science (and art) of programming computers so they can learn from data. (Géron 2017d) In this section, we will take a look at some machine learning concepts that are relevant to this thesis. We will start with basics of ML, explaining some basic terminology. Afterwards we will take a brief look at the models used in the thesis. Finally, we will end the section on some additional relevant concepts. Machine learning is a broad subject with many subcategories. Here we will be focusing on supervised learning; learning from labeled data.

2.2.1 Basics of Machine Learning

How does a program learn from labeled data? Assume we have a dataset consisting of N input-output pairs such as the following:

$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

For any of these pairs, we call the input value x_i the predictor and the y_i the labels.(Géron 2017d) Supervised learning is based on the assumption that there is function dependency between the predictors and the labels. This functional dependency can be modeled as $f(x_i) = y_i$, where f is called the true function. The goal of the supervised learning is to construct a function h -called hypothesis- that approximates the true function f. In general, we split supervised learning instances into two groups: classification and regression. Classification refers to the instances where the labels are selected from a finite set of values (such as cat, dog or cow). Regression on the other hand refers to the instances where the labels are numeric values (such as 0, 10 or 42). (Russell and Norvig 2010a)

We usually construct hypotheses by training models on the data. But what does it mean to train a model on the data? To explain this we need to understand what a loss function is: A loss (or cost) function is a function that maps models to numeric values. Specifically, a loss function tells us how much our model fails to explain some data. In this context, training means tuning the adaptive parameters of a model in order to minimize the loss function. (Bishop 2006)

We are of course interested in models that are capable of explaining more than just the training data. We want to see if the model generalizes well; if it can accurately predict labels for previously unseen inputs. Becuase of this we test the trained models on heldout data. Usually, we split our data into three distinct sets: training set, validation set, and test set. We train our models on the training set, compare them using the validation set and report the performance of the final model on the test set. It is important that the test set is held out during the training phase. Otherwise it is not possible to see if the model generalizes well or just memorizes the data.(Bishop 2006)

2.2.2 Relevant Models

Let's take a brief look at some popular machine learning models.

Linear Regression

A linear regression model is a linear function of the input variabes. It corresponds to a weighted sum of the input features plus a constant called the bias term.(Géron 2017f) For example, a linear regression model for a dataset with two input features would look like this:

$f(x_i) = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} = y_i$

For more information on linear regression models, please refer to (Géron 2017f).

Ridge Regression

Ridge regression is very similar to linear regression. The only difference is that ridge regression adds a regularization term $a \sum_{i=1}^{n} \theta^2$ to the cost function. This penalizes large weight in order to prevent overfitting, a phenomenon where a model memorizes the data instead of generalizing over it.(Géron 2017f)

a is called the regularization strength and determines how much the large weights get penalized. Regularization strength is a hyperparameter, a parameter that is set before the learning process and is not optimized over the data.(Géron 2017f)

More information about ridge regression can be found at (Géron 2017f)

Support Vector Machines

Support Vector Machines construct hyperplanes that act as decision boundries between classes. What makes these hyperplanes special is that they are positioned to be as far away as possible from the closest data points. The closest data points therefore determine where the hyperplane lays, and are called the support vectors. (Géron 2017e)

This is all better understood with some visualisation. Please take a look at Figure 2.2. It is easy to see in this picture that the hyperplane is positioned to maximize the margin between the support vectors. It is also important to realize that the data points that are not support vectors do not affect the hyperplane in any way. (Géron 2017e)

Support vector machines can also be used for regression. This is also done by constructing a hyperplane. However, the objective is reversed. This hyperplane is positioned to include as many data points in the margin as possible. The size of the margin is determined by hyperparameter ϵ. (Géron 2017e)

For the mathematical background and more information on support vector machines, please refer to (Géron 2017e).

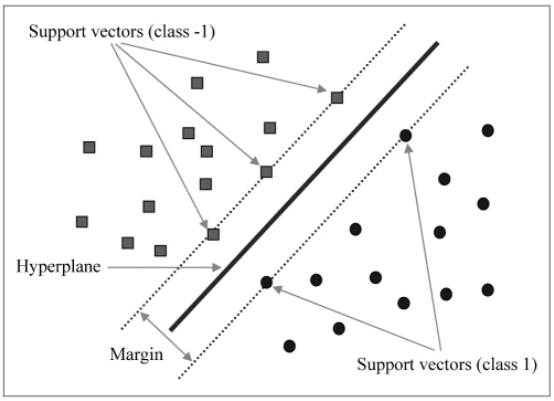


Figure 2.2: A Support Vector Machine (Chen, Hsiao, Huang, et al. 2009)

Decision Trees

Decision trees are a form of graph based models.(Bishop 2006) More specifically, decision trees are trees that contain basic conditional statements on their non-leaf nodes. For any given feature vector, a decision tree runs tests these conditions to reach a decision. These tests start at the root of the tree. Depending on the outcome of the test, a child node is selected. This process continues iteratively until a leaf node is reached, where a decision is stored. Decision in this context is the label for the feature vector.(Russell and Norvig 2010a)

For the mathematical background and more information on decision trees, please refer to (Géron 2017a).

Ensemble Learning

Instead of training a singular model, we can train many weak models and aggregate the results. Such aggregations of models are called ensembles, and the learning process is called ensemble learning. One simple yet particularly powerful example is the random forest: A random forest is an ensemble of decision trees, each trained on different random subsets of the training set. A random forest makes predictions by aggregating the predictions of each individual tree. For regression, the predictions are averaged. For classification, the most predicted class is the final prediction.(Géron 2017c)

More information on ensemble learning can be found at (Géron 2017c)

2.2.3 More Machine Learning

Finally, let's go through some more advanced machine learning concepts that will be relevant in the next chapter.

Data Preprocessing

It is often the case that we have to work on imperfect data. Some values might be missing for some instances, data might not be uniform or features might not be helpful as they are. Collection of transformations we apply to the data before feeding it into to our models is called preprocessing.(Géron 2017b) This preprocessing includes:

Cleaning the data to make sure it is uniform.

Extracting features from the existing input variables.

Scaling the features to make sure all the features have similar scales.

The reasoning behind these transformations and more information on preprocessing can be found in (Géron 2017b).

Cross Validation

Cross validation is a technique used in model selection. The data is first divided into N parts. Afterwards the model is trained on $N-1$ parts and validated on the remaining part. This process is repeated N times, each with a different part as the validation set. Finally, the best performing model is selected.(Bishop 2006)

Hyperparameter Optimization

Hyperparameter optimization refers to the process of fine tuning the model hyperparameters to improve performance. This is usually done automatically using grid-search.(Géron 2017b)

R^2 Score

R^2 Score (pronounced R squared) or the coefficent of determination is statistical metric. In most simple terms, R^2 score shows how much of the variance in the target values can be explained by a statistical model. It can be used to evaluate models, where a higher score indicates a better fit on data. Best possible R^2 score is 1.0. R^2 score can be negative too, indicating that the model is arbitrarily worse.(*Scikit-Learn User Guide, 3.3. Model evaluation: Quantifying the quality of predictions* n.d.)

2.3 Nutrition

2.3.1 Dietary Assesment

Food and Agriculture Organization of United Nations defines dietary assesment as the following:

"Dietary assessment is an evaluation of food and nutrient intake and dietary pattern of an individual or individuals in the household or population group over time." (Food and Agriculture Organization of the United Nations 2018)

There are many different dietary assesment methods available, but we will be focusing on integration of innovative technologies to improve dietary assesment. We will specifically be focusing on mobile-based technologies. Mobile-based technologies allow users to log their dietary-intake in real time using a smartphone or a tablet. This allows for higher quality control of the data.(Food and Agriculture Organization of the United Nations 2018)

2.3.2 Dietary Goals

We couldn't find any resource that shows a collection of dietary goals that are collectively agreed upon by nutrition scientist. However, popular dietary health applications such as MyFitnessPal and Fitbit implement three main dietary goals:

Lose weight

Maintain weight

Gain weight

After consulting S. Holzmann and B. Kaiser (personal communication, May 16, 2019), we have determined that this set of goals is also suitable for our task.

3 Methodology

Our goal here is to build a system that is capable of accurately predicting when the user is going to eat. This can be achieved by building predictive models and training them on user data. But, in order to this, we first need to collect data on user's eating patterns. This process is called dietary assesment. There are many methods for dietary assesment. Here, we will be focusing on a mobile-based solution: We will build a dialog agent that allows for instant logging of a user's dietary intake. Afterwards, using the gathered data, we will build a machine learning pipeline that predicts the time of the next entry.

3.1 Dialog Agent

3.1.1 Design

Chatbot Architecture

We will be building a task-oriented dialog agent. The task is to understand the eating patterns of the user and make future predictions. Let's take a look at our task from a frame-based perspective. What information do we need from the user in order to accomplish this task? There are many answers to this question with varying complexity. Keeping the scope of the thesis in mind, we contruct a simple frame shown in Table 3.1

Table 3.1: Frame for Logging Dietary Intake

Slot	Type	Explanation
User Id	int	A user's identifying integer
Meal Time	Time	Time of the meal
Meal Date	Date	Date of the meal
Meal Type	int	Type of the meal, 0 for Snack and 1 for Full Meal

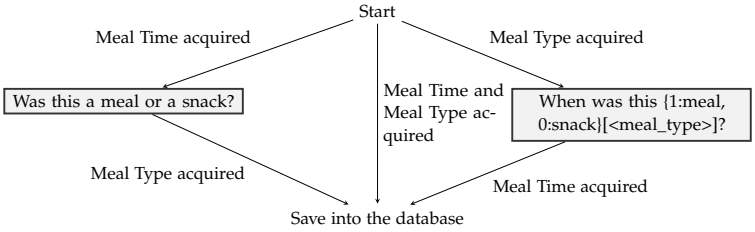
Now we need to come up with a question for every slot in the frame. Table 3.2 shows these questions. We don't assign questions to User Id and Meal Date slots, as they are meant to be infered from the context.

Table 3.2: Questions for Logging Dietary Intake

Slot	Questions
User Id	-
Meal Time	When was this meal/snack?
Meal Date	-
Meal Type	Was this a meal or a snack?

Based on these questions, we build a simple control structure shown in Figure 3.1. This control structure follows a mixed initiative approach, allowing the user to deliver the information in any sequence.

Figure 3.1: Control Structure for Meal Logging



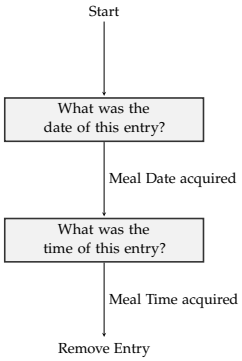
But what if a user makes a faulty entry and wants to remove it? We need to include a system that let's users remove entries. Knowing the date and time of an entry is enough to remove it, so we construct the frame shown in Table 3.3.

Table 3.3: Frame and Questions for Removing Entries

Slot	Type	Questions
User Id	int	-
Meal Date	Date	What was the date of this entry?
Meal Time	Time	What was the time of this entry?

Based on Table 3.3, we construct a control structure for entry removal, shown in Figure 3.2.

Figure 3.2: Control Structure for Removing Entries



Before logging any meals, however, we need to know some things about the user. For example, user's timezone. It is not possible to obtain accurate time information without knowing the user's time zone. For the overall system, we also need to know the user's dietary goal. Both of these informations must be acquired at first contact with the user. Based on this, we build the frame and questions seen in Table 3.4 for user registration. We derive the timezone from user's location. The dietary goal is represented as an integer, corresponding to:

0: Lose Weight
1: Mainting Weight
2: Gain Weight

Table 3.4: Frame for User Registration

Slot	Type	Question
User Id	int	-
Timezone	text	Please share your location with me so I can find your timezone.
Goal	int	Please select your goal.

Based on Table 3.4, we construct the control structure shown in Figure 3.3

We also contruct two more control structres, for changing timezone and goal settings, shown in Figure 3.4

We also need to send messages at predicted times and ask for feedback on the prediction. The control structure for sending messages and asking for feedback is shown in Figure 3.5.

Bringing all these control structures together, we end up with a final structure describing the entire system, shown in Figure 3.6

Figure 3.3: Control Structure for User Registration

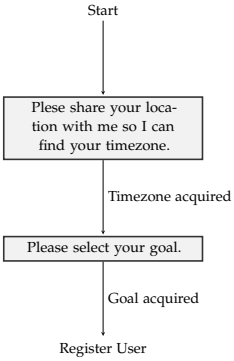


Figure 3.4: Control Structures for Changing Timezone and Goal

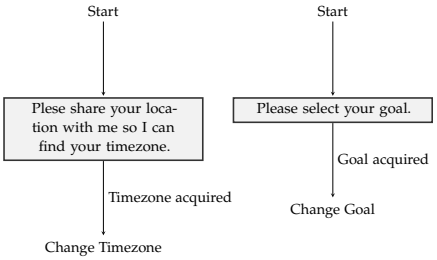


Figure 3.5: Control Structure for Sending Messages and Asking for Feedback

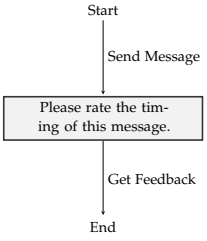
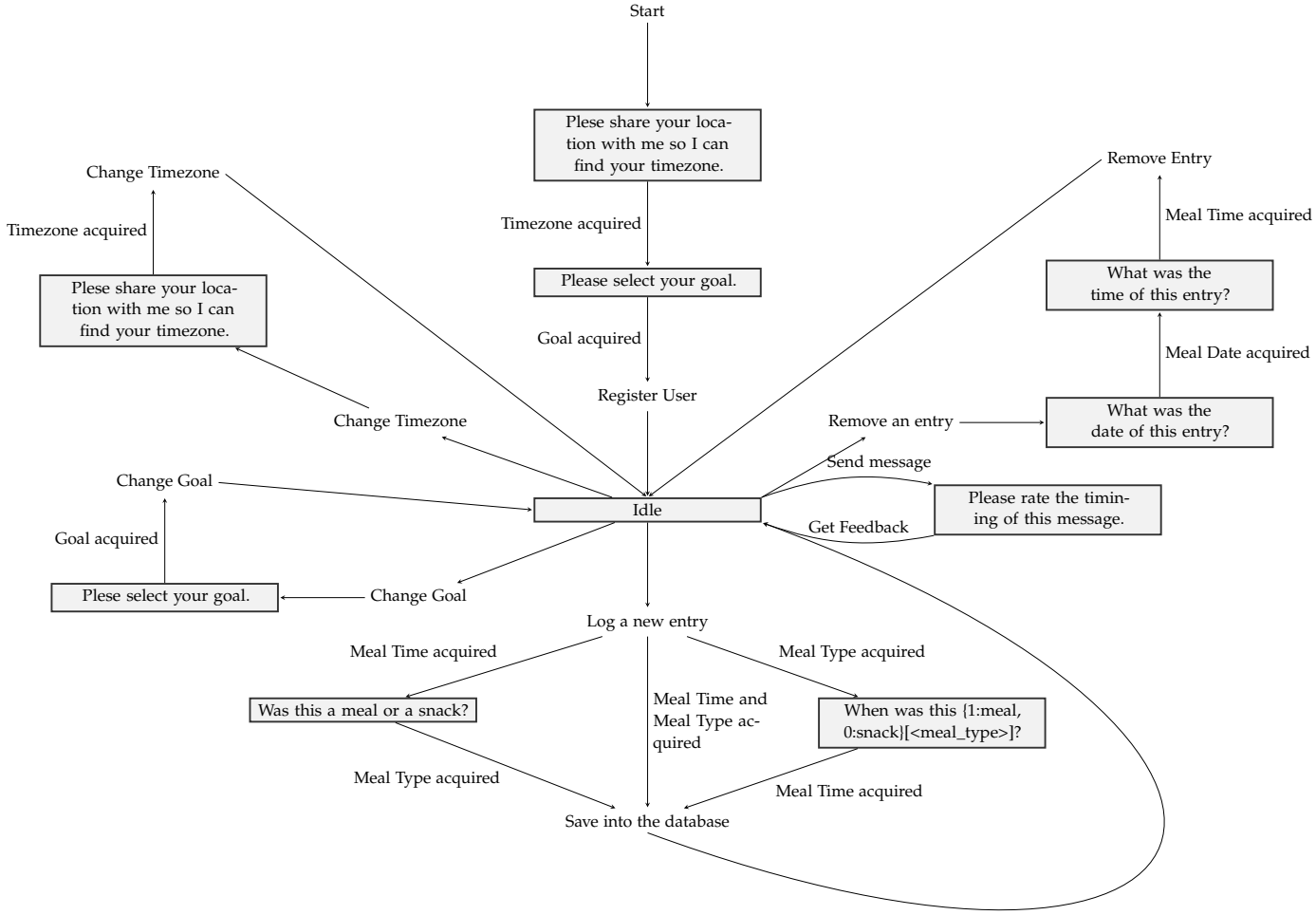


Figure 3.6: Control Structure for the Entire System



Information Extraction

We need to extract relevant information from the user's messages in order to fill the slots. This mainly concerns extracting time and type of an entry. For other information, we will be taking advantage of Telegram Chatbot API's various functionalities, which will be explained in the next subsection. We will be performing information extraction primarily with the help of regular expressions. All the mentioned regular expressions need the X flag set, allowing for whitespace within the expression. For extracting the type of the entry (i.e. meal or snack) we will be using the regular expression shown in Figure 3.7.

```
type_pattern = r'''\b
(?:
(?P<meal>meal|breakfast|lunch|dinner)|
(?P<snack>snack|small|little|few)
)
\b'''
```

Figure 3.7: Regular Expression for Extracting Entry Type

This regular expression matches a word from a set of words between word boundries. If the matched word is meal, breakfast, lunch or dinner; the extracted type is set to meal. If the matched word is snack, small, little or few; the extracted type is set to snack.

```
r'''\b
(?:at\s*)?
(?:
(?P<hours>
[01]?\d|2[0-3])
(?:[:\.\s]
(?P<minutes>
[0-5]?\d))?
(?P<AmPm>\s*[ap]m)?
|(?P<hours_alt>[01]\d|2[0-3])(?P<minutes_alt>[0-5]\d))
\b'''
```

Figure 3.8: Regular Expression for Extracting Absolute Temporal Expressions

For extracting the time of the entry, we use two different regular expressions: one for

```
r'''\b
(?P<hours>
(?:half(?:\s*\b(?:a|an|)\b\s*)?|
\ba\b|\ban\b|
1?[0-9]|2[0-4])?
\s*
(?:h|hour)s?)?
\s*
(?P<minutes>
(?:\ba\b|\ban\b|
[0-9]{1,2})
\s*
(?:m|min|minute)s?\b)?
\s*
(?:ago|before|prior)
\b'''
```

Figure 3.9: Regular Expression for Extracting Relative Temporal Expressions

absolute temporal expressions shown in Figure 3.8, and another for relative temporal expressions shown in Figure 3.9. From here on, in order to avoid confusion between regular expressions and temporal expressions, we will be referring to the regular expressions as extractors.

Absolute time extractor matches any phrase that has a hour part; a single or double digit number between 0 and 23. It also optionally matches a minutes part; a single or double digit number between 0 and 59. The hours and minutes parts need to be seperated by a delimiter; either a colon, a period or a space. After the hours and minutes parts, if there is a phrase signifying 12 hour format (am-pm), the phrase is also matched. Alternatively, the extractor also matches military time; a four digit representation where the first two digits signigy hours and last two digits signify minutes. (e.g 2000)

Relative time extractor matches phrases followed by a trigger word. This word can be ago, before or prior. Preceeding the trigger word, hours part, minutes part or both can be matched. Hours part is a single or double digit number between 0 and 24, followed by a word signifying hours (h, hours etc.). Minutes part is a single or double digit number between 0 and 99, followed by a word singifying minutes (min, minutes etc.). Additionally, the extractor also matches the word 'half' if it preceeds the hours part. After the extraction, relative time is normalized based on the user's timezone.

When we get a message from the user, we first run the relative time extractor on the text. To understand why, consider the phrase 'I ate 5 minutes ago'. While this is clearly a relative temporal expression, it is also matched by absolute temporal expression. However, relative time extractor involves trigger words. This means that if the relative time extractor matches a phrase, it is safe to assume that the phrase is a relative temporal expression.

We have a final regular expression shown in Figure 3.10 We use this expresssion to determine if a user is trying to log an entry.

```
r'''\b
(eat|ate|eaten|food)
\b'''
```

Figure 3.10: Regular Expression for User Intent

Similar to the type expression, this expression matches a word from a set of words between word boundries. The words that can be matched are; eat, ate, eaten and food. We use this expression only if both time and type expressions fail to match anything. This is because if the previous expressions match a phrase, then the intent is already established.

3.1.2 Implementation

Instead of developing a stand-alone application, we build our dialog agent as a Telegram chatbot. This makes the development process significantly easier. Instead of focusing on application design, we can fully focus on our dialog agent. We build our dialog agent using Python 3 and python-telegram-bot library, which provides a python interface for Telegram Bot API.

In the previous subsection, we have mentioned that we would be using some Telegram Bot API functionalities in order to make information extraction easier. We will start this subsection by explaining these functionalites. Next we will move on to the control structure of a Telegram chatbot. Finally, we will briefly mention some other libraries that are also used in our implementation.

python-telegram-bot Library

Telegram Bot API

There are some Telegram Bot functionalities that we make use of in order to make information extraction easier. First of those are the chatbot commands. Telegram

Bot API allows user to perform actions via commands. A command is a string of alphanumerical characters and underscores, all of which is preceeded by a '/' character. For example, '/help', '/set_nickname' or '/command3' or all viable commands. In our system, we use commands to start any operation other than logging an entry. We implement a total of four commands:

/start command is automatically called when a user messages our chatbot for the first time. This command initiates the registration sequence shown in Figure 3.3. Registered user's can not call this command.

/change_timezone command initiates the timezone changing sequence shown in Figure 3.4. This command can be called at anytime after the registration.

/change_goal command initiates the goal changing sequence also shown in Figure 3.4. This command can also be called at anytime after the registration.

/remove_entry command initiates the entry removal sequence shown in Figure 3.2. This command can also be called at anytime after the registration.

We also make use of Inline-Keyboard functionality to decrease information extraction effort. Inline-Keyboards are collections of custom buttons, each with their own text and callback values. For any situation where a user needs to choose from a finite set of alternatives, we send an Inline-Keyboard. This significantly limits the need for information extraction, as the user directly tells us what we need to know. We use Inline-Keyboards for:

Selecting goals We send a Inline-Keyboard with three buttons. Each button displays a goal from our list of dietary goals. When the user presses on a button, we receive an integer value corresponding to the selected goal.

Picking up a date for entry removal We send an interactive calendar built as an Inline-Keyboard. When a user selects a date, we receive a datetime object corresponding to the date selected. We use `telegramcalendar.py` script in order to create these interactive calendars. `telegramcalendar.py` script is written by Github user unmonoqueteclea. The repository contatining the script can be seen at:

`https://github.com/python-telegram-bot/python-telegram-bot`

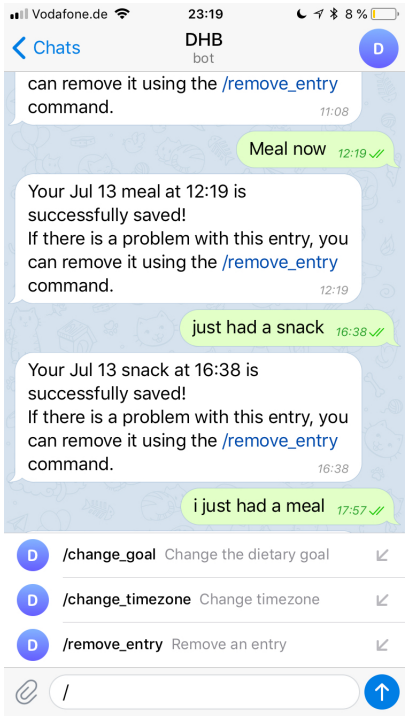


Figure 3.11: Screenshot Showing Available Commands for the Chatbot

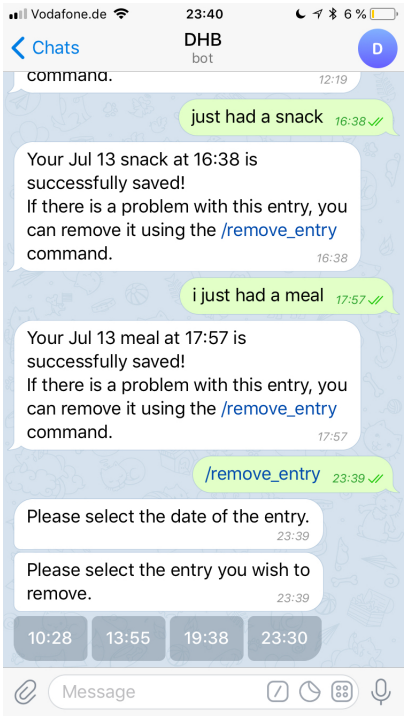


Figure 3.12: Screenshot Showing an Inline Keyboard

Picking up a time for entry removal After the date selection, we send the user an Inline-Keyboard with buttons containing the entry times for that date. When the user picks a time, we get the string representation of the time selected.

Getting Feedback on Notification Timing When we send a message at a predicted time, we also send a Inline-Keyboard for feeback. This keyboard contains five buttons: Too Early, Early, On Time, Late and Too Late. When the user presses a button, we get an integer corresponding to the button pressed.

One last thing to mention is how we get the user's timezone information. Telegram allows users to send their locations. Instead of performing any information extraction, we ask the user for their location. When we receive the location, we use it to find user timezone.

Control Structure

Telegram chatbots are mainly controlled by Updater and Dispatcher objects. Updater object receieves updates from Telegram and passes them to the Dispatcher object. Dispatcher object, in turn, passes the update to the suitable Handler objects.

Each Handler object accepts a predefined set of updates, and passes these updates to the associated functions, which then process the update. Telegram Bot API offers a wide variety of Handler objects. In our implementation, we will be using:

CallbackQueryHandler handles updates containing callback queries, which are results of user interaction with Inline-Keyboard.

CommandHandler handles updates that are results of chatbot commands. Each CommandHandler handles a specific command.

MessageHandler handler updates that are results of telegram messages. It is possible to filter messages based on type. (text, image etc.)

ConversationHandler is an aggregate of multiple handlers. ConversationHandler objects are designed like finite state automatas where handlers are the states. For every ConversationHandler, there are three sets of handlers. The first is a list of entry points, determining what starts the conversation. The second is a dictionary of states and handlers, determining which handler acts at which state. The last is list of fallback handlers. These handlers are called when no other handler in a given state is capable of handling an update.

It is clear to see that our control structures can easily be translated into ConversationHandler objects. We simply create handlers for every state in our control structures, and tie them together using conversation handlers. We implement a total of 5 conversation handlers:

Registration This conversation handler has the /start command handler as its entry point. It contain two states, one that asks for the user's timezone and the other that asks for the user's goal. The handler that corresponds to the first state is a message handler, filtering for messages containing locations. The handler that corresponds to the second state is callback query handler, intented to process the user's goal selection. There is also another message handler that accepts any message as a fallback. This handler reminds the user that the registration needs to be completed in order to continue.

Changing Timezone This conversation handler has the /change_timezone command handler as its entry point. It contains only one state, which coressponds to a message handler that filter for messages containing locations. The fallback handler for this one accepts any message and cancels the operation, returning the chatbot to the idle state. This handler cancels the current operation and returns the chatbot to the idle state.

Changing the Goal This conversation handler has the /change_goal command handler as its entry point. It also only contains one state, a callback query handler intended to process user's goal selection. This conversation handler also has a fallback handler that cancels the operation.

Removing an Entry This conversation handler has the /remove_entry command handler as its entry point. It has two states, both of which correspond to callback query handlers. The first one is for processing date selection, and the second one is for processing time selection. This conversation handler also has a fallback handler that cancels the operation.

Meal Logging This conversation handler is unique in the way that it has a message handler as an entry point. It has three states, all of which are message handlers that filter for text messages. The first one is the same as the entry point. The second one is intended to extract type, and the thirds one is intened to extract time. When a user sends a text message, the initial handler tries to extract the time and the type of the entry. If it extracts both, the entry is saved into the database If it extracts only one, it transitions into other state based on the missing information. If it extracts none, it runs the regular expression shown in Figure 3.10 in order to determine if the user is trying

to log an entry. If the pattern matches a phrase, the handler tries to get the information required. If this pattern also matches nothing, the chatbot returns to the idle state. This conversation handler also has a fallback handler that cancels the operation.

In addition to the conversation handlers, we implement a callback query handler to handle feedback on predictions.

Other Libraries

We use other libraries to implement some additional functionality not offered by python-telegram-bot library.

For scheduling messages to be sent at predicted times, we use the 'schedule' library.

For deriving timezones from locations, we use the 'timezonefinder' library.

Finally, for converting timezone information stored as text into timezone objects, we us the 'pytz' library.

3.2 Machine Learning

In this section, we will go through the machine learning component of our system. Much like any machine learning pipelin, we will start with data preprocessig. Afterwards, we will take a look at the models used. Finally, we will end this section on model seleciton and prediction.

We use pandas (McKinney 2010–) and numpy (Oliphant 2006–) libraries for data manupilation. For creating, training and testing our models, we use the scikit-learn (Pedregosa, Varoquaux, Gramfort, et al. 2011) library.

3.2.1 Preprocesseing

We will be going through some of the standard steps of preprocessing.

Cleaning the Data

Data neads to be cleaned before being fed into the model. We have to make sure that there are no rows with missing values, and that each column has the same data type for every row.

After inspecting our data, we see that there are indeed no missing values. However, we realize that the meal time values aren't saved uniformly. The current system stores meal time in 'hours:minutes: format. However, there are some entries from the early stages of the system, where the meal time is stored as 'hours:minutes:seconds.microseconds+timezone'.

We solve this problem by applying the function shown in Figure 3.13 to meal time values.

```
lambda x:x[:5]
```

Figure 3.13: Cropping Function for Time Values

This function simply returns the substring containing the first 5 characters of the input string. Values that are 5 characters long ('hours:minutes' values) aren't affected. For any other value, the return value corresponds to the 'hours:minutes' part, rendering the data uniform.

After making sure that our data is uniform, we sort our values based on meal date and meal time for further processing.

Feature Extraction

We need to craft more expressive features from our input values. We also need to craft a target value (label) that is dependent on our features. After some brainstorming we decide on the features shown in Table 3.5 and the target value shown in Table 3.6.

Table 3.5: Features

Feature	Explanation
Day of the Week	Day of the week as a one-hot encoded feature value
Meals so far	Number of meals the user had on the particular date
Snacks so far	Number of snack s the user had on the particular date
Last Entry Type	Type of the last entry, 0 for snack and 1 for meal
Last Entry Time	Time of the last entry, represented in minutes

Table 3.6: Target

Target	Explanation
Time Until Next Entry	Time until next entry, represented in minutes

We convert our base input values to feature vectors using the code shown in Figure 3.14.

```
def feature_template(data):
    new_data = np.zeros((data.shape[0]-1,12))
    current_datetime = ''
    meal_count = 0
    snack_count = 0
    counter = 0
    meal = 0
    first = True
    for row in data:
        if first:
            current_datetime = datetime.strptime(row[1]+' '+row[2],
                                                 '%Y-%m-%d %H:%M')
            meal = row[3]
            first = False
            continue

        new_datetime = datetime.strptime(row[1]+' '+row[2],
                                         '%Y-%m-%d %H:%M')
        new_data[counter][int(current_datetime.strftime('%w'))] = 1
        new_data[counter][7] = meal_count
        new_data[counter][8] = snack_count
        new_data[counter][9] = meal
        new_data[counter][10] = int(current_datetime.strftime('%H')) * 60
        new_data[counter][10] += int(current_datetime.strftime('%M'))
        diff = new_datetime - current_datetime
        new_data[counter][11] = diff.days*1440 + diff.seconds/60
        if current_datetime.date() == new_datetime.date():
            if meal:
                meal_count += 1
            else:
                snack_count += 1
        else:
            meal_count = 0
            snack_count = 0

        current_datetime = new_datetime
        meal = row[3]
        counter += 1
```

Figure 3.14: Function for Feature Extraction

```
last_entry = np.zeros((11,1))
last_entry[int(current_datetime.strftime('%w'))] = 1
last_entry[7] = meal_count
last_entry[8] = snack_count
last_entry[9] = meal
last_entry[10] = int(current_datetime.strftime('%H')) * 60
last_entry[10] += int(current_datetime.strftime('%M'))

previous_entries = pd.DataFrame(new_data, columns=
                                ['Sunday','Monday',
                                'Tuesday','Wednesday',
                                'Thursday','Friday',
                                'Saturday','Meals␣So␣Far',
                                'Snacks␣So␣Far','Meal␣or␣Snack',
                                'Time␣of␣Entry','Next␣Entry'])

return previous_entries, last_entry
```

Figure 3.14: Function for Feature Extraction (continued)

This function takes a numpy array containing the user data and returns two objects. The first object is pandas data frame, containing the feauture vectors and target values for every entry except the last one. We use this data frame to train and test our models. The second object is a numpy array, contatining the feature vector for the the last entry. This array is meant to be used after model selection, in order to predict the time of the next entry for the user.

Training and Test Set Seperation

We need to seperate our data into training and test sets before the next step. We do this with the help of `train_test_split()` function from scikit-learn library. This function accepts an array of feature vectors (X), an array of target values (y), and a float value specfiying the size of the test set. It returns four arrays; X_train, X_test, y_train and y_test. We set our test size as 0.1, meaining 10% of the data is used for testing.

Feature and Target Scaling

After feature extraction, we end up with a feature vector. This feature vector mostly consists of small values; usually zeros or ones, and single digit numbers for two features. However, one feature, namely the time of entry, has a range between 0 and 1440. Most machine learning models perform better when the features are similaryly scaled. Because of this, we need to rescale our features.

We use `MinMaxScaler()` object from scikit-learn library for this task. This object transforms features by scaling each feature to a given range, where the default range is betwen 0 and 1. It works like this: First, we fit the scaler to the data using the `fit()` function. Then we call `transform()` function on the data to scale our features.

So why did we need to seperate our data before this step? Because fitting the scaler to the data means that the scaler learns the data. Good data science practice dictates that the test set must contain unseen data. As a result we fit the scaler to our training data, and then use it to transform both the training and the test data. We also have significantly larger target values (mostly between 0 and 1440). This might cause instability during the training phase. Because of this, we repeat the scaling process on the target values using another scaler.

3.2.2 Models

Now that our data is preprocessed, we can feed it into our models. We have very little data to work with. This means that we need to use simple, lightweight models to prevent overfitting.

We use a total of 4 base models, all of which we create using scikit-learn library. Additionally we use 4 ensemble learners based on these 4 models. All of the models and their corresponding scikit-learn objects can be seen in Table 3.7 `SVR()` `kernel` parameter is selected to keep the model linear. Other model parameters shown are selected based on trial and errork

Table 3.7: Models

Model	Scikit-learn Object
Linear Regression	`LinearRegression()`
Ridge Regression	`Ridge()`
Support Vector Machine Regression	`SVR(kernel='linear')`
Decision Tree Regression	`DecisionTreeRegressor()`

Ensemble of Linear Regressors	`BaggingRegressor(`
	`base_estimator=`
	`LinearRegression(),`
	`max_samples=1.0,`
	`max_features=0.7)`

Ensemble of Ridge Regressors	`BaggingRegressor(`
	`base_estimator=Ridge(),`
	`max_samples=1.0,`
	`max_features=0.7)`

Ensemble of SVM Regressors	`BaggingRegressor(`
	`base_estimator=`
	`SVR(kernel='linear'),`
	`max_samples=1.0,`
	`max_features=0.7)`

Random Forest Regressor	`RandomForestRegressor(` `max_features=0.7)`

We also perform 10 fold cross validation on every base model, and hyper parameter optimization on two base models using `GridSearchCV()` from scikit-learn library. We optimize the `'alpha'` parameter for the ridge regression between 0.5 and 1.0, and the `'C'` parameter for SVM regresssion between 0.1, 0.5 and 1.0.

3.2.3 Model Selection and Prediction

After preparing our data and model, we come to the model selection phase. We compare our models using the R^2 scores. The first thing we do is checking if there is an

already stored model in the database. If there is, we set this model as the best model and the model score as the best score. If there is no stored model, we set the best model to `None` and the best score to 0.

Next, we create, train and test the previously mentioned eight models for a number of iterations. This number is set to 100 if the best score is 0, signiying that this is the first time that we train models for the user. Otherwise, the number is set to 10. During an iteration, if we find a model performing better than the best model, we set this model as the best model and its score as the best score. After all the iterations are complete, we check the best model. If it is still set to `None`, this means that all the models scored less than 0, most likely due to insufficent data. In this case we make no prediction. Otherwise, we use the best performing model to predict the time until next entry. After the prediction, we save the best performing model into our database.

3.2.4 Feedback Value

Eating is not an instantenous activity. When users report meal times, they can report any point in time that is within the duration of the eating activity. We use a feedback value to account for this. When we send a user a notification, we ask for a feedback on the timing. We save the user's feedback as an int value and use it the next time we need to make a prediction. This feedback value denotes delay, and can be positive or negative. When we make a prediction, we simply add this feedback value to the result. We calculate the time of the next entry based on this final value.

3.2.5 Chatbot Integration

When a user logs a new entry, we call a function that contains the entire machine learning pipeline. This process doesn't take long due to small amount of data available and the simplicity of the models. We schedule a message at the precited time, and the system goes back to the idle state.

3.3 Testing the System

We give access links to volunteers consisting of university students in order to test the system. The volunteers are asked to use the system for 15 days, and then give feedback on the experience.

4 Results

4.1 Data

A total of 6 users clicked the access link and registered in the system. The users have collectively logged a total of 150 entries by 14/07/2019. Distribution of these entries among users can be seen in Figure 4.1.

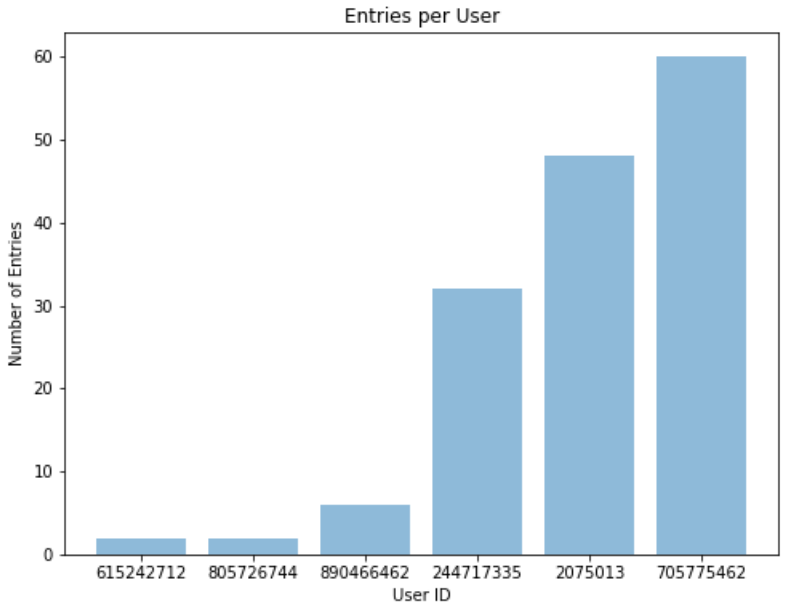


Figure 4.1: Distribution of Entries per User

After removing the users with less than 10 entries, we are left with 3 users. Eating patterns of these 3 users can be seen in Figure 4.2.

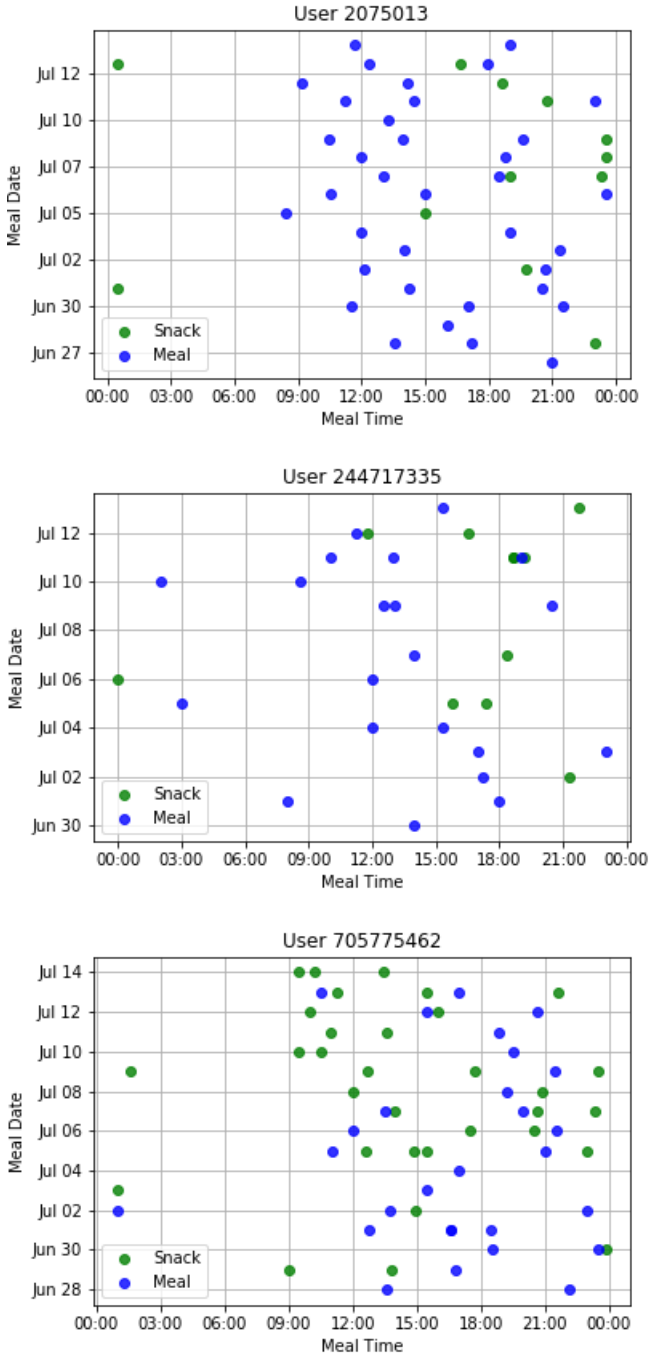


Figure 4.2: Eating Patterns of the Most Active Three Users

4.2 Models

Best scoring models and the corresponding R^2 scores for the users with more than 10 entries can be seen in Figure 4.3.

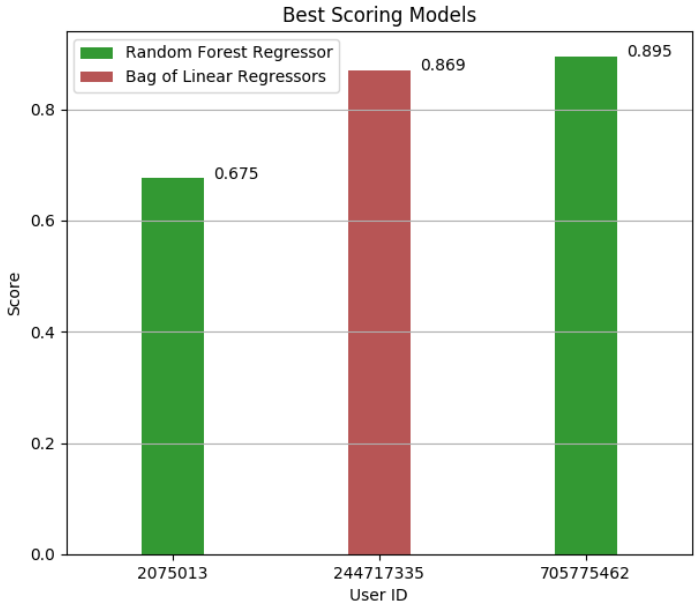


Figure 4.3: Best Scoring Models and the Corrosponding R^2 Scores

5 Discussion

5.1 Data

It is clear to see that dietary patterns vary drastically not just between users, but also between days of a user. This high variance might be due to the user demographic which consists of university students. Whatever the reason, this lack of easily discernible patterns naturally makes pattern recognition harder.

Another obstacle that makes pattern recognition harder is the apperant lack of data. There are only 3 users with more than 10 entries, which means that we can train and test models only for these 3 users.

5.2 Dialog Agent

This lack of data for the other 3 users can be attributed to a common feedback about the dialog agent: Users often forget to log what eat. This common feedback points to a need for a reminder system, which could be implemented as future work.

Multiple users stated that they usually logged all their meals at the end of the day collectively. This behaviour obviously leaves most of the predictive functionality obsolete, as the prediction occurs at the time of logging. One possible solution to this problem is to schedule all the messages for the day at the start of the day, and then optimize the timing as the user makes entries. This functionality could also work as a reminder, solving two problems at once.

One of the users suggested adding a functionality to display all logged entries, in a similar fashion to Figure 4.2. She argued that this could help people see their eating patterns and try to change them for the better.

5.3 Machine Learning

Overall, users stated that the system would somewhat accurately predict when they would get hungry.

Machine learning models achieved suprisingly high R^2 scores; over 0.85 for two users and 0.67 for another. Ensemble learning methods outperformed basic models

as expected; Best performing models were Random Forests for two users, and Bag of Linear Regressors for the other one. The fact that the best performing models vary between users is an indicator that there isn't a single best model for our task.

There is another interesting point to mention here: User 705775462 had the highest entry frequency, followed by user 2075013, followed by user 244717335. However, while the predictive model for user 705775462 held the highest score, it is closely followed by the predictive model for user 244717335. This seems to indicate that predictibility and eating frequency are not directly related.

6 Conclusion

We have built a system that faciliates communcation with the user and makes somewhat accurate predictions. However, there is a lot of room for improvement. In the next section, we will be going over some possible future work on this topic.

6.1 Future Work

6.1.1 Solving the Initial Lack of Data

One of the biggest problems concerning our system is the initial lack of data. This lack of data prevents predictive functionality until the user logs enough entries. However, there might be some possible solutions to this problem.

One possible solution would be to ask the users do describe their eating habits at registration. This information could be used for initial values for time prediction, which could be optimized over time as the user logs more entries.

Another possibility is to use a clustering algorithm in order to categorize users' eating patterns. During the registration phase, the new users would answer a series of questions about their eating patterns. Users would be categorized based on their answers. After the categorization, randomly sampled data from the other users in the same category could be used to train an initial predictive model for the user.

It is also possible to combine these two approaches by categorizing the users based on eating patterns, but still consulting the user about concrete meal times.

These systems could be evaluated by comparing how well the initial predictive model approximates the late stage predictive model trained on the actual user data.

6.1.2 Reminders

One missing component of out dialog agent that is much desired is the ability to send reminders. While sending a reminder is easy, timing and frequency of reminders is not as trivial. One possible approach would be to base the reminders on the predictions. If the user doesn't log anything for a certain time after a prediction, a reminder could be sent. But what 'certain time' is appopriate for this task?

What about the frequency? While sending too few reminders is obviously not optimal, sending too many reminders could easily agitate the users.

These questions could be answered by asking the user for feedback on the reminders. The user could rate the appopriateness of the reminder, and the reminder component could be iteratively optimized based on these ratings.

The resulting component could be evaluated based on the number of reminders that recieved good ratings.

6.1.3 Better Notification Timing

We focus on sending users notifications when they are about to eat. However, this is not the only time that people make food related choices. Consider grocery shopping for example, which dictates what the user eats at home for the next days.

It could be possible to build a system that could notify the user whenever a food related choice is about to be made. For example, the user could notify the system before certain events, such as shopping, cooking or going to a restaurant. A predictive model could be trained that processes user location and time to determine if such an event is about to occur. Such a model could be evaluated based on recall; number of accurately predicted food related events divided by the total number of food related events.

6.1.4 Detailed Meal Logging

When logging a new entry, we only differentiate between meals and snacks. One improvement over this would be to let the user log exactly what they eat. The system would need to calculate the approximate nutritient intake for every entry, and make recommendations on what to eat based on this information and the user goal.

List of Figures

List of Tables

Bibliography

Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.

Chen, S.-T., Y.-H. Hsiao, Y.-L. Huang, S.-J. Kuo, H.-S. Tseng, H.-K. Wu, and D.-R. Chen (Aug. 2009). "Comparative Analysis of Logistic Regression, Support Vector Machine and Artificial Neural Network for the Differential Diagnosis of Benign and Malignant Solid Breast Tumors by the Use of Three-Dimensional Power Doppler Imaging." In: *Korean journal of radiology : official journal of the Korean Radiological Society* 10, pp. 464–71. DOI: 10.3348/kjr.2009.10.5.464.

Food and Agriculture Organization of the United Nations (2018). *Dietary Assessment: A resource guide to method selection and application in low resource settings*.

Géron, A. (2017a). "Decision Trees." In: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc. Chap. 6.

– (2017b). "End-to-End Machine Learning Project." In: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc. Chap. 2.

– (2017c). "Ensemble Learning and Random Forests." In: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc. Chap. 7.

– (2017d). *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc.

– (2017e). "Support Vector Machines." In: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc. Chap. 5.

– (2017f). "Training Models." In: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc. Chap. 4.

Date and time – Representations for information interchange – Part 1: Basic rules (Feb. 2019). Standard. Geneva, CH: International Organization for Standardization.

Jurafsky, D. and J. H. Martin (n.d.[a]). "Dialog Systems and Chatbots." In: *Speech and Language Processing*. 3rd ed. Chap. 24. In preparation.

– (n.d.[b]). "Information Extraction." In: *Speech and Language Processing*. 3rd ed. Chap. 18. In preparation.

– (n.d.[c]). "Regular Expressions, Text Normalization, and Edit Distance." In: *Speech and Language Processing*. 3rd ed. Chap. 2. In preparation.

– (n.d.[d]). *Speech and Language Processing*. 3rd ed. In preparation.

McKinney, W. (2010–). *Data Structures for Statistical Computing in Python*.

Oliphant, T. (2006–). *NumPy: A guide to NumPy*.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Russell, S. J. and P. Norvig (2010a). "Learning From Examples." In: *Artificial Intelligene: A Modern Approach*. 3rd ed. Pearson. Chap. 18.

– (2010b). "Natural Language Processing." In: *Artificial Intelligene: A Modern Approach*. 3rd ed. Pearson. Chap. 22.

Scikit-Learn User Guide, 3.3. Model evaluation: Quantifying the quality of predictions (n.d.).