Final Dissertation

Natural Language Understanding and app architecture for the Dicio free software assistant

Fabio Giovanazzi

Contents

Abstract

Users primarily use voice assistants for basic tasks like controlling lights, setting timers, or checking the news. Though this simplicity limits monetization opportunities for the industry, it also hints that computation to understand user input could be conducted entirely offline. This approach could significantly enhance user privacy, improve availability in areas with unstable internet access, potentially reduce response time, and pave the way for free software alternatives to thrive without reliance on cloud services.

In 2019 we created Dicio, a free software assistant that runs on-device and therefore has the aforementioned advantages. We structured the codebase to allow the community to contribute by adding and translating new skills, and the app now supports 9 skills and 9 languages.

In order to interpret user requests, we created an ad-hoc Natural Language Understanding system which compares the user input with sentences from a reference set whose meaning is known. This setup met the requirements in performance and in ease of creating community translations, but was not flexible enough to support complex matching scenarios, and would fail on some tricky user requests.

In this thesis we present a new NLU system that can handle even the tricky requests. It can do so thanks to a scoring function to use during matching that takes into consideration these four statistics: the number of matched words in the user sentence and in the reference sentence, and the total number of words in the user sentence and in the reference sentence. The actual scoring function we chose is linear in the four parameters, because we could prove that a non-linear function would not behave well linguistically, and would not allow for a greedy matching algorithm.

The proposed NLU system is also more flexible than the previous because it makes it possible to add new reference constructs that can efficiently match pieces of the user sentence, returning a score and possibly extracting data in the process. The previous NLU could also do these operations, but without the possibility of defining new constructs easily, and without a well-defined and efficient way to calculate the score of newly added constructs.

In this thesis we also describe the migration of the codebase from the traditional Android UI development approach, which was imperative and required writing an XML file controlled by Java (similar to HTML+JS), to the Jetpack Compose declarative UI toolkit. While in the process we also rewrote the code in Kotlin, embraced the Model-View-ViewModel pattern for UI, adopted the Hilt library for dependency injection and made the code more modern in general.

1 Introduction

A decade ago the industry had high hopes on voice assistants being the next streamlined computing interface. However, it turned out that users primarily use them for basic tasks like controlling lights, setting timers, or checking the news. This simplicity significantly limits monetization opportunities for the industry, and various famous assistants were shut down [1] or lost billions of dollars. [2] At the same time, not needing to understand complex queries hints that computation to understand user input could be conducted entirely offline. This approach could significantly enhance user privacy, improve availability in areas with unstable internet access, potentially reduce response time and pave the way for free software alternatives to thrive without reliance on cloud services. The absence of free software alternatives was the main reason why we started developing a new assistant for Android in 2019, and published it on F-Droid and Play Store under the name "Dicio". The app allows users to interact via voice with 9 skills, performs both Speech To Text (STT) and Text To Speech (TTS) completely offline, and is translated by the community in 9 languages.

The Dicio assistant employs an ad-hoc Natural Language Understanding (NLU) system which compares the **user input** with sentences from a reference set whose meaning is known. The system was initially designed to be fast and to make it simple for the community to provide translations of the reference sentences. As the application grew, however, a third unfulfilled requirement started becoming apparent: extensibility. Adding new ways to match pieces of sentence required too much effort or was downright impossible, due to the API design. Moreover, the system could not interpret well some specific sentences, and there was no clear way to solve this issue.

Another problem which slowed down development was the architecture of the application code, which did not follow good practices everywhere, and still relied on libraries, programming languages and tools wwhich were old or had a few shortcomings. For example, in the last few years Google pushed for the adoption of Kotlin over Java, and introduced a new declarative toolkit for UI with significant advantages over the traditional Android imperative approach to UI.

This thesis addresses the problems by:

- creating APIs that are easy to implement and maintain, through which skills give a score to the user input and generate graphical and speech output (chapter 3)
- rethinking the scoring strategy used in the comparison of two sentences, to make it more general and better at interpreting some nasty cases (chapter 4)
- developing an efficient and extensible algorithm to find the best match between two sentences based on the aforementioned scoring strategy (chapter 5)
- converting the code to Kotlin, migrating to new libraries especially for UI (see fig. 1.1), and adopting a modern architecture design (chapter 6)

Chapter 4 contains a mathematical proof justifying our choice to only consider linear scoring functions. Non-linear functions would not fulfill some basic properties that a scoring function is supposed to have based on linguistical common sense.

To choose the parameters for the linear function, we developed an evaluation framework allowing to test out various possibilities. We also used it to check non-linear functions, which only in some

[1] The Verge: Microsoft shuts down Cortana app on Windows 11, https://www.theverge.com/2023/8/11/23828311/microsoft-shuts-down-cortana-windows-11

[2] Ars Technica: Amazon Alexa is a "colossal failure," on pace to lose $10 billion this year, https://arstechnica.com/gadgets/2022/11/amazon-alexa-is-a-colossal-failure-on-pace-to-lose-10-billion-this-year/

cases were as good as linear functions. In order to build the evaluation dataset, we picked many of the tricky examples that the algorithm in the previous version of the Dicio app did not interpret well.

To track improvements in the performance of the algorithm from chapter 5, we developed some benchmarks, explained in section 5.5. We also proved the final complexity to be optimal, because it is the same as the alleged best possible complexity of the Levenshtein distance algorithm [4].

In addition to the core parts of the thesis, the Background chapter (2) introduces the Dicio project and some useful concepts to keep in mind, and the Conclusion chapter (7) lists related and future work and finally draws conclusions.

The links to the code we produced in this thesis and other relevant resources are listed here:

- The Dicio application is available in the *dicio-android* repository, which also hosts the implementation of the NLU algorithm in the `skill` module, and the plugin that generates Kotlin code for sentences based on `app/src/main/sentences/*.yaml` files in the `sentences-compiler-plugin` module: https://github.com/Stypox/dicio-android
- The framework that evaluates the scoring functions, along with the dataset of sentence comparisons, is available in the *dicio-evaluation* repository: https://github.com/Stypox/dicio-evaluation
- The parser of the regex-like sentence language is in the *dicio-sentences-compiler* repository: https://github.com/Stypox/dicio-sentences-compiler
- The parser and formatter that is used by some Dicio skills to extract numbers, dates and durations from spoken text is available in the *dicio-numbers* repository (not covered by this thesis): https://github.com/Stypox/dicio-numbers
- Dicio on F-Droid: https://f-droid.org/packages/org.stypox.dicio/
- Dicio on Play Store: https://play.google.com/store/apps/details?id=org.stypox.dicio

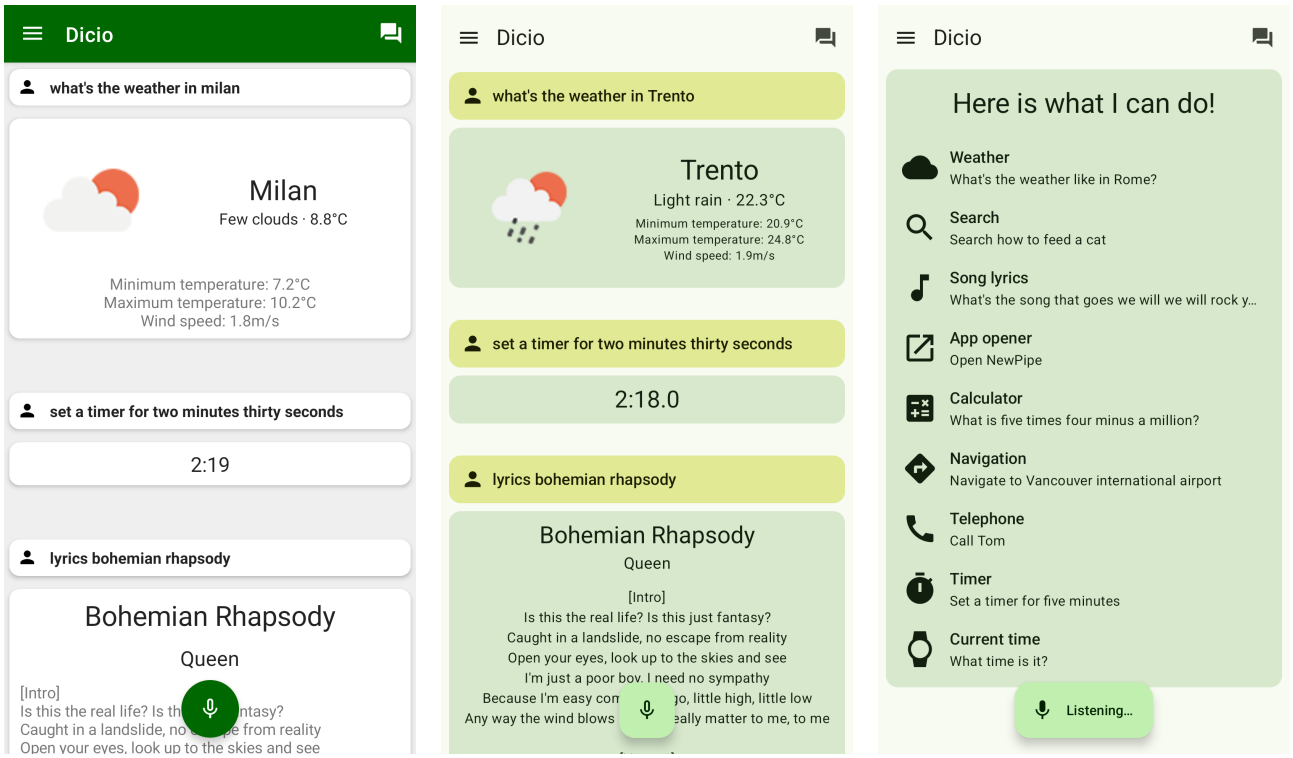

Figure 1.1: On the left, some example interactions of a user with Dicio, in the old look. In the center, the same home screen, but in the new Material3 UI. On the right, the greeting message shown on startup, with the list of available skills.

2 Background

This section explains some basic concepts about what an assistant is (section 2.1) and provides some information on the state of the Dicio application before we started working on this thesis (section 2.2).

2.1 What is an assistant?

Personal assistants are usually made of multiple skills. Each *skill* is assigned the task of understanding a specific set of queries from the user, and to perform actions or load data based on the received queries. The kind of output provided by a skill varies depending on the skill itself and on the kind of assistant. For example, in voice-only assistants the skills might respond to queries just via synthesized voice, while in an assistant with a UI the skills will probably also show some widget on the screen. The widgets can be more or less interactive: for example, a widget showing weather forecasts will be static, while a timer widget will show a countdown and allow stopping the timer.

The user usually has the ability to send text input to an assistant via a Speech To Text (STT) engine. Most assistants also feature a wakeword recognition service, that listens for a wakeup word (e.g. "Hey Google") in the background and start the assistant whenever they hear it. Upon receiving some input the assistant has to choose a skill that can process it, i.e. it has to *classify* the input. The chosen skill is then mandated with processing the input and generating output. For example, if the user says "What's the weather like", the assistant should understand that the user is asking for weather, and trigger the weather skill.

Most of the times it is not enough to just *classify* the input, but the assistant should also be able to extract *entities* from it. For example, if the user asks "What's the weather in Trento", the assistant should extract "Trento" as the place to show weather for. Extracted entities, also called *capturing groups* or *captures* in this thesis, might not be just text, but also numbers, dates, durations, and more.

In some cases simple request-response interactions between the user and the assistant are not enough, for example because the skill might want to ask the user for more information before being able to fulfill the original request, or because the user might refer to information from a previous interaction. Therefore assistants usually support some form of longer conversations. For example, if the user says "Set a timer", the assistant might respond with "How much should the timer last?" and wait for the user to say "Five minutes".

2.2 The previous state of the Dicio application

The Dicio project was started in 2019, and even before we begun working on this thesis, the application could already fulfill most requirements for a personal assistant: it had a Natural Language Understanding unit that could efficiently classify user input, extract entities to a limited extent and handle basic multi-turn conversations; it ran the STT, the TTS and the NLU locally; it featured a nice-looking UI for showing graphical output for skills. Moreover, it had 9 skills (weather, search, lyrics, app opener, calculator, navigation, telephone, timer, current time) translated in 9 languages (Czech, English, Frech, German, Greek, Italian, Russian, Slovenian and Spanish) by the community.

The Dicio application was originally created to provide a free and open source alternative to proprietary assistants on Android, because there was no app like that on the most famous store for free software apps, F-Droid [8]. The app was designed to allow the community to easily add and translate skills. Now the app is published on F-Droid and on Play Store.

The NLU in Dicio worked by comparing the user input to a set of reference sentences for each skill, written by translators in a compact maneer. The NLU could only extract text entities, and it was the job of skills to extract numbers, dates and durations, e.g. using the *dicio-numbers* library. [1]

[1] https://github.com/Stypox/dicio-numbers

3 Skill architecture

This chapter describes the architecture of a core part of the Dicio application, namely the skill evaluator, whose purpose is to handle skill metadata (section 3.1), build skills (section 3.2), trigger the correct skill based on user input (section 3.3), and keep track of the conversation state (section 3.4).

3.1 `SkillInfo` holds metadata and builds skills

A *skill* is like a mini-app that is able to interpret a specific kind of user query and answer the query with speech and graphical output. Just like apps have an icon and a name, skills also have some metadata that describes them. Each skill in Dicio has a corresponding `SkillInfo` object with the following metadata fields:

- `id` is a unique identifier among all skills, e.g. "weather"
- `name` is the localized name of the skill, e.g. "Weather"
- `sentenceExample` is a localized example of a query that the skill can answer, e.g. "What's the weather?"
- `icon` is an icon representing the skill, e.g. sun and clouds representing weather
- `neededPermissions` is a list of Andoid permissions that the skill requires in order to work, for example the telephone skill needs the permission to read contacts and place phone calls

`SkillInfo` objects are also in charge of building `Skill` objects, via the `build()` method. However, a skill might not available on a particular device or in a particular language (e.g. if its reference sentences have not been translated), therefore the `isAvailable()` method also exists to tell whether the skill is buildable or not. Both the `build()` and the `isAvailable()` methods take a `SkillContext` object as parameter, just like other skill functions. This allows them to access information and resources about the app and the environment, such as the Android context, the current locale or the localized number parser and formatter.

Sometimes skills want to expose some settings: for example, the weather skill allows choosing a default place to use when the user query does not specifically mention a city. This is made possible by the `renderSettings` field, which is `null` if the skill has no settings to expose, and otherwise is a lambda that can render UI to let the user change the settings. Persisting settings to disk is left to the skill.

For further customization, the user is given the option to enable or disable individual skills.

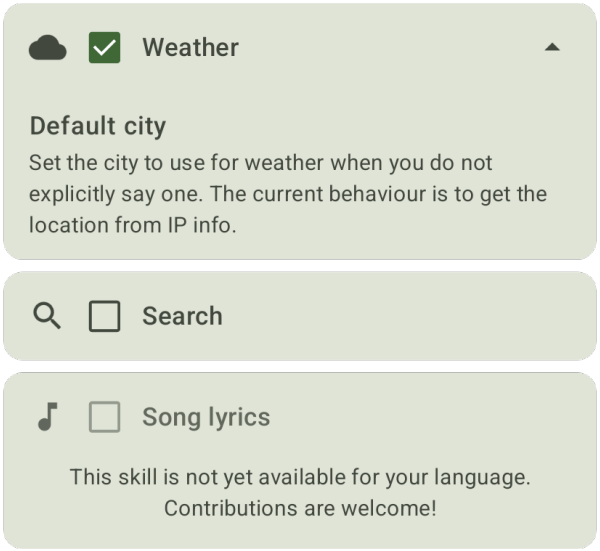

Figure 3.1: Three skills in the settings: weather is enabled and exposes the "Default city" setting, search is disabled, lyrics is not available

3.2 The `Skill` interface

Classes inheriting the `Skill` interface are meant to process user input, provide a score indicating how well they match, and generate speech and graphical output in case they end up being the best matching skill.

The `Skill` interface has a generic type parameter, `InputData`, indicating the data that the input-processing phase (i.e. `score()`) extracts from the input, and that the output-generating phase (i.e. `generateOutput()`) can use later.

The `score()` function takes the user input string as a parameter and returns an object of type `InputData`, i.e. the data extracted from the input, and a `Score` object, with information on how well the skill matches the input. Each `Skill` also has a `specificity` field that can be low, medium or high. It is an additional hint to the skill ranker, indicating how specific the skill is in recognizing input. If a skill with a high specificity yields a high score, it will be preferred over a skill with a lower specificity and roughly the same score. This is because if the user input matched both a highly-specific skill and a lower specificity one, then what the user most likely wanted was the highly-specific skill. For example, if the user says "what is the weather", the weather skill should be chosen, even if the search skill interprets it with a high score as searching for "the weather".

Each skill is free to implement the `score()` method as it wants, which allows for a lot of flexibility. However, most skills in the Dicio application extend from `StandardRecognizerSkill` described in chapter 5.

The `generateOutput()` function takes an `InputData` object as a parameter, and returns a `Skill Output` object, which contains localized speech output, a function to build the graphical output UI, and information on how to continue the conversation. `generateOutput()` is a `suspend` function, which means it can be run in an asynchronous maneer, and so it can perform network requests or other slow operations from a background thread.

Both `score()` and `generateOutput()` also take a `SkillContext` as parameter, to be able to access system information and resources or perform actions.

3.3 How the best skill is chosen

As seen in section 3.2, each skill exposes a `score()` function that, given the user input string, calculates a score representing how well the skill can understand that input. The `Score` type is generic, in order to accomodate for different scoring strategies. However, any two `Score` objects always need to be comparable (otherwise how would it be possible to determine the best one?), so they implement these functions:

- `scoreIn01Range()`, which calculates a score number in the $[0, 1]$ range (higher is better). Comparing two scores using the output of this function is lossy, because the function cannot be linear (since non-constant linear functions are never constrained to a range), and thus does not respect theorem 4.4.2 and property 4.4.1.

- `isBetterThan()`, which compares two `Score` objects in a lossless way if they are compatible with each other, and otherwise falls back to comparing based on `scoreIn01Range()`.

Whenever an input device captures some input from the user (e.g. the user talks to the Speech To Text model), the assistant needs to decide which skill should be asked to generate output. In the Dicio application this task is assigned to the skill ranker. The skill ranker is constructed with a list of `Skill` objects, which it then separates in three classes based on the specificity of each skill. Whenever an input needs to be processed, the skill ranker matches the input against each skill and keeps only the one with the highest score in each class according to `isBetterThan()`.

Finally, in order to choose among the three resulting skills, we developed some heuristics that use the three values in range $[0, 1]$ returned by `scoreIn01Range()`. The thresholds shown in table 3.1 are checked in the order given by the arrows, and whenever the skill with the specificity indicated in the column header has a score higher than the threshold, that skill is chosen. If this process terminates without any skill being chosen, it is highly likely that no skill can fulfill the user request, and therefore a fallback skill is chosen instead.

Low	Medium	High
	0.85	
0.9	→	0.8
0.9	→	0.8 → 0.7

Table 3.1: Skill ranker thresholds

The implementation of `scoreIn01Range()` for a `StandardScore` (see section 5.3) uses the four parameters from definition 4.3.1. It calculates the harmonic mean of the **user** *score* $\mathbf{u_m}/\mathbf{u_w}$ and of the reference *score* r_m/r_w. The harmonic mean was chosen so that if one of the two terms is significantly lower than the other, the mean tends towards the lower one.

$$\frac{2}{\frac{\mathbf{u_w}}{\mathbf{u_m}} + \frac{r_w}{r_m}}$$

3.4 Displaying the output and continuing the conversation

If the score of a skill turns out to be the best, `Skill.generateOutput()` is called to generate speech and graphical output for the user, as seen in section 3.2. The returned object inherits `SkillOutput`, contains all of the data that can be used to generate the output, and is supposed to be serializable. This allows saving the data to disk, and restoring it on the next app start to show a list of previous interactions to the user. These three `SkillOutput` interface methods use the data for various purposes:

- `getSpeechOutput()` returns a localized string that will be spoken via the configured Text To Speech service.
- `GraphicalOutput()` builds the UI that will be shown in a box on the home screen. The UI can be interactive and can act as a widget: for example the timer skill shows the ongoing countdown.
- `getNextSkills()` returns a list of skills that could continue the current conversation. If this list is non-empty, the next time the user asks something to the assistant, these skills will be considered before all other skills, and if any of these skills understands the user input well enough, the conversation continues. For example, if the user says "Call Mom", the assistant may answer with "Should I call mom?" and this method would return a skill that can understand a yes/no response.

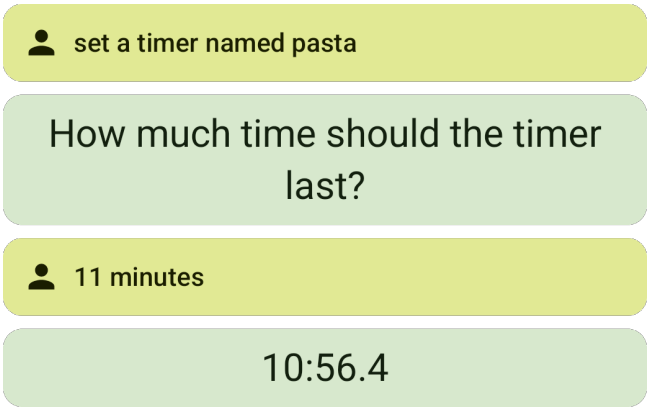

Figure 3.2: An example of multiturn conversation involving the timer skill, where the user says the name of the timer but forgets the duration, so the assistant asks it and then starts the timer

4 Scoring strategy

This chapter addresses the problem of comparing an **input from the user** to a list of reference sentences to determine the best-matching one, and thus understand the user intent (see section 4.1 for definitions). This is done by reasoning about existing metrics (section 4.2), picking the parameters that play a role when comparing sentences (section 4.3), analyzing the properties of a function that can choose the best among two sets of such parameters (section 4.4), and finally choosing one such function (sections 4.5 and 4.6).

4.1 Definitions

The **input from the user** is colored in **orange** and represents a query from the user that needs to be recognized to extract the user intent. It is often the case that *entities* need to be extracted from queries, too, such as places, dates, durations, numbers, pieces of text, etc. For example, if the user asks "What's the weather in Trento", an assistant should not only extract "weather" as the main intent, but also "Trento" as the place.

Reference sentences are colored in blue and have a unique intent associated to them. A reference sentence is meant to be matched against user queries so as to generate a similarity score. The reference sentence that scores highest against a specific user query determines the intent associated to the query, and possibly also extracts entities and other data.

4.2 What about Levenshtein?

When scoring how well a **user sentence** matches against a reference sentence, we basically want to calculate some form of *string distance* between the two sentences.

One first idea might be to use the Levenshtein string distance [13], which operates on single characters. That algorithm repeatedly performs the following three operations on the **user sentence** it is given as input, in order to turn it into the reference one: *deletion* (removing a character), *insertion* (adding a character), *substitution* (replacing a character with another one). Each of these three operations is assigned a weight (usually 1 for all of them), and the purpose of the algorithm is to minimize the sum of all the weights of the operations needed to turn the first sentence into the second.

While Levenshtein distance is a good metric to compare strings, it does not work well if used directly to compare two sentences, for multiple reasons:

- since it operates on single characters, longer words will result in a higher weight than shorter ones, which makes little linguistical sense
- a word in a sentence might end up matching its characters to multiple words in the other sentence, e.g. "tree story" and "restore" have a string distance of just 4, but it does not make sense for "restore" to be split in half
- given a fixed **user sentence**, the longer the reference sentence, the more likely it is to have a higher string distance even if more characters match, e.g. **"how are you"** has a lower distance to "who are you" (2) with respect to "how are you doing" (6)
- it does not allow for more complex behaviors in reference sentences, such as optional words, alternative words, capturing groups that extract *entities*, ...

The things we can get away from Levenshtein, though, are the allowed operations (*deletion*, *insertion* and *substitution*), and the fact that each of them is assigned a weight. Moreover, the optimal computational complexity of the Levenshtein algorithm is $\mathcal{O}(\mathbf{n}\mathsf{m})$ when using Dynamic Programming (unless the SETH conjecture is false [4]). **n** and m are, respectively, the lengths of the **user** and of

the reference sentences. Therefore, it would be good if the scoring algorithm we choose in the end is also $\mathcal{O}(\mathbf{nm})$.

4.3 Four relevant parameters

Definition 4.3.1 (Score parameters)**.** By drawing conclusions from the weak spots of Levenshtein, and from how it works, we came up with these four relevant parameters:

- $\mathbf{u_m}$, *user matched*, approximately counts the number of words in the **user sentence** that were matched correctly (i.e. were not skipped). Should roughly be increased by the same amount $\mathbf{u_w}$ is increased by, every time a word from the user sentence is consumed and matches.
- $\mathbf{u_w}$, *user weight*, approximately counts the total number of words in the **user sentence**. Should roughly be increased by 1 every time a word from the user sentence is consumed.
- $\mathsf{r_m}$, *reference matched*, approximately counts the number of words in the reference sentence that were matched correctly. Should roughly be increased by the same amount $\mathsf{r_w}$ is increased by, every time a word from the reference sentence is consumed and matches.
- $\mathsf{r_w}$, *reference weight*, approximately counts the total number of words in the reference sentence. Should roughly be increased by 1 every time a word from the reference sentence is consumed.

The terms "approximately" and "rougly" were used a lot in the above definitions, because:

- single special characters in the **user sentence** can be matched too, and not only words, although they would have a smaller weight
- some important words may be assigned a weight > 1, while less important words (e.g. "a", "the") may be assigned a weight < 1
- some parts of the reference sentence may not be really "words", e.g. capturing groups

Remark 4.3.2 (Parameter constraints)**.** All of these parameters are ≥ 0, and moreover $\mathbf{u_m} \leq \mathbf{u_w}$ and $\mathsf{r_m} \leq \mathsf{r_w}$ because it would not make sense to have more correctly matched words than there are in total.

Definition 4.3.3 (Set of scores)**.** Let us denote $S = \mathbb{R}^4$ as the set of all possible tuples $(\mathbf{u_m}, \mathbf{u_w}, \mathsf{r_m}, \mathsf{r_w})$, so that $s \in S$ is a score. Note that this definition also includes scores that do not abide by remark 4.3.2, but it is not important.

4.4 Properties of scoring functions

Once a **user sentence** has been matched against two reference sentences, or during the process of performing such matchings, we need a way to compare two scores and choose which one is the best. More formally, we need a *linear/total order relation* $\leq$ amongst elements of S. This is equivalent to choosing a function $f : S \to \mathbb{R}$ to map scores onto the real line, and then using the standard definition of $\leq$ on $\mathbb{R}$, since $|S| = |\mathbb{R}^4| = |\mathbb{R}|$.

Remark 4.4.1 (Order is preserved under sum)**.** One additional property we want this relation to have, is the following:

$$\forall s_1, s_2, s_3 \in S,\ s_1 \leq s_2 \iff s_1 + s_3 \leq s_2 + s_3$$

where $+$ is the component-wise sum. This, again, makes linguistical sense: if "what is the weather" matches better than "what is the meaning" against **"is the weather"**, then after prepending **"what"** to the user sentence (obtaining **"what is the weather"**, and obtaining $s_3 = (\mathbf{1}, \mathbf{1}, 1, 1)$ since "what" is also present in both reference sentences), "what is the weather" should still match better than "what is the meaning".

Theorem 4.4.2 (Scoring functions are $\approx$ linear)**.** *All functions $f : S \to \mathbb{R}, f \in C^1$ that respect property 4.4.1 are of the form $f(s) = g(h(s))$, where $g : \mathbb{R} \to \mathbb{R}, g \in C^1$ is strictly monotone and $h : S \to \mathbb{R}$ is linear.*

Remark 4.4.3 (Equality is preserved under sum)**.** To prove theorem 4.4.2 we can focus on picking $s_1, s_2 : f(s_1) = f(s_2) \iff f(s_1) \leq f(s_2) \wedge f(s_1) \geq f(s_2)$, and then if for a specific function it is possible to find an $s_3 : f(s_1+s_3) \neq f(s_2+s_3) \iff f(s_1+s_3) < f(s_2+s_3) \vee f(s_1+s_3) > f(s_2+s_3)$, then the condition 4.4.1 would not be satisfied. Therefore $s_1, s_2 : f(s_1) = f(s_2) \implies \forall s_3 \in S, f(s_1 + s_3) = f(s_2 + s_3)$.

Lemma 4.4.4 (Same value implies same gradient)**.** Let us look at two points $s_1, s_2 : f(s_1) = f(s_2)$. We can take the Taylor expansions of degree 1 of the function at the two points:

$$f(s_1 + \Delta s) = f(s_1) + \langle \nabla f(s_1), \Delta s \rangle + o(\|\Delta s\|)$$
$$f(s_2 + \Delta s) = f(s_2) + \langle \nabla f(s_2), \Delta s \rangle + o(\|\Delta s\|)$$

Let us pick $s_3 = \Delta s = h\nabla f(s_1)$ with $h \in \mathbb{R}_+$, then $f(s_1 + h\nabla f(s_1)) = f(s_2 + h\nabla f(s_1))$ because of remark 4.4.3, but $f(s_1) = f(s_2)$ so those terms cancel out.

$$f(s_1 + h\nabla f(s_1)) = f(s_2 + h\nabla f(s_1))$$
$$\cancel{f(s_1)} + \langle \nabla f(s_1), h\nabla f(s_1) \rangle + o(\|h\nabla f(s_1)\|) = \cancel{f(s_2)} + \langle \nabla f(s_2), h\nabla f(s_1) \rangle + o(\|h\nabla f(s_1)\|)$$
$$\cancel{h}\langle \nabla f(s_1), \nabla f(s_1) \rangle = \cancel{h}\langle \nabla f(s_2), \nabla f(s_1) \rangle + \cancel{h}o(\|\nabla f(s_1)\|)$$
$$\langle \nabla f(s_1), \nabla f(s_1) \rangle = \langle \nabla f(s_2), \nabla f(s_1) \rangle + o(\|\nabla f(s_1)\|)$$

We can ignore $o(\|\nabla f(s_1)\|)$ because the equation must always hold, including in the limit where $h \to 0$ (which, remember, is also when $s_3 \to 0$).

$$\langle \nabla f(s_1), \nabla f(s_1) \rangle = \langle \nabla f(s_2), \nabla f(s_1) \rangle$$

From the above equation it follows that $\nabla f(s_2) = 0 \implies \nabla f(s_1) = 0$. Now let us focus on the case where $\nabla f(s_1) \neq 0$ and $\nabla f(s_2) \neq 0$.

$$\|\nabla f(s_1)\|\cancel{^2} = \|\nabla f(s_2)\| \cancel{\|\nabla f(s_1)\|} \cos(\theta)$$
$$\|\nabla f(s_1)\| = \|\nabla f(s_2)\| \cos(\theta)$$
$$\text{where } \theta = \angle(\nabla f(s_2), \nabla f(s_1))$$

By doing the same operations but by choosing $s_3 = \Delta s = h\nabla f(s_2)$ this time, we obtain the other direction of $\nabla f(s_2) = 0 \iff \nabla f(s_1) = 0$. Otherwise, in case $\nabla f(s_1) \neq 0$ and $\nabla f(s_2) \neq 0$, we obtain the following system. Notice that $\cos(\theta) > 0$ because otherwise we would have something > 0 on the left and something ≤ 0 on the right, making equations impossible.

$$\begin{cases} \|\nabla f(s_1)\| = \|\nabla f(s_2)\| \cos(\theta) \\ \|\nabla f(s_2)\| = \|\nabla f(s_1)\| \cos(\theta) \end{cases}$$

$$\implies \cos^2(\theta) = 1 \iff \cos(\theta) = \pm 1$$
$$\text{but } \cos(\theta) > 0, \text{ so } \cos(\theta) = 1 \iff \theta = 2k\pi \text{ with} k \in \mathbb{Z}$$
$$\implies \|\nabla f(s_1)\| = \|\nabla f(s_2)\|$$

So $\nabla f(s_1)$ and $\nabla f(s_2)$ are aligned and have the same length, therefore they must be equal, and we have proven that $f(s_1) = f(s_2) \wedge f(s_1 + s_3) = f(s_2 + s_3) \forall s_3 \implies \nabla f(s_1) = \nabla f(s_2)$. ■

Let us ignore for a moment the level sets where $\nabla f = 0$. Since the gradient along a level set is always perpendicular to the level set itself, and all gradients in a level set are the same (and thus parallel) because of lemma 4.4.4, then each level set must be a flat hyperplane. The dimension of such hyperplane would be $\mathbb{R}^3$ (remember we are working in $\mathbb{R}^4$) due to the implicit function theorem by Dini and because $f \in C^1$ and $\nabla f \neq 0$.

Now, onto the level sets where $\nabla f = 0$.

Lemma 4.4.5 (Points with same score form straight lines)**.** Let there be two points $s_1, s_2 : f(s_1) = f(s_2)$ in a level set, and $s_m = (s_1 + s_2)/2$ is the midpoint. Let us assume for the sake of argument that $f(s_1) < f(s_m)$, and let us use remark 4.4.3:

$$f(s_1) < f(s_m)$$
$$f(s_1 + (s_m - s_1)) < f(s_m + (s_m - s_1))$$
$$f(s_m) < f(s_2)$$
$$\implies f(s_1) < f(s_m) < f(s_2)$$

But this contradicts $f(s_1) = f(s_2)$, therefore $f(s_1) = f(s_m) = f(s_2)$ after doing the same process in the other direction (i.e. with the assumption that $f(s_1) > f(s_m)$).

By repeatedly taking midpoints, we can build a dense set on the $\overline{s_1 s_2}$ segment. By continuity of f and by the fact that f is constant on the dense set, f must be constant on the whole $\overline{s_1 s_2}$ segment.

Now, if f is constant on a continuous segment, then it must be constant on the whole straight line that contains that segment. Otherwise it would easily be possible to choose s_1, s_2, s_3 along that direction that break remark 4.4.3: a possible example is to choose s_1 and s_2 belonging to the segment, and s_3 such that $s_1 + s_3 = s_2$ but $s_2 + s_3$ is outside of the segment. Therefore if $s_1, s_2 : f(s_1) = f(s_2)$, then every point on the straight line passing through s_1 and s_2 is in the level set. ■

Lemma 4.4.5 generalizes to more than two (non-aligned) points, which implies that level sets, and in particular those where $\nabla f = 0$, must be hyperplanes of dimension 0 (a single point), 1 (a line), 2 (a plane), 3 (a space), 4 (the degenerate case where f is constant on the whole $\mathbb{R}^4$, but in that case f would be linear). However, if there were level sets of dimension 0, 1 or 2, they would correspond to local maximums/minimums, around which it would be possible to find s_1, s_2, s_3 that break remark 4.4.1. Therefore the level sets where $\nabla f = 0$ must also be of dimension $\mathbb{R}^3$.

Since level sets cannot intersect with other level sets, all hyperplanes must be parallel to one another, because otherwise they would intersect. This also implies that all gradients of f at all points in S are parallel (when they are not zero), so f can change along only one direction (let us call it $\vec{d}$), while it would be constant along all other perpendicular directions. Let us call $g : \mathbb{R} \to \mathbb{R}, g \in C^1$ the function representing how f changes along $\vec{d}$. g must be strictly monotone, because otherwise it would be easy to find s_1, s_2, s_3 along $\vec{d}$ that break remark 4.4.3. Therefore g is invertible, and after removing the change along $\vec{d}$ from f by doing $g^{-1}(f(s))$ we must be left with a linear function in all directions. Therefore $f(s) = g(h(s))$, where $h : S \to \mathbb{R}$ is linear, finally proving theorem 4.4.2. ■

4.5 Choosing a scoring function

Theorem 4.4.2 proves that f is of the form $f(s) = g(h(s))$. However, we can notice that g is useless for our purposes, since it is stricly increasing and so does not change the order of elements. Therefore we can just choose f itself to be linear, i.e. $f(\mathbf{u_m}, \mathbf{u_w}, \mathsf{r_m}, \mathsf{r_w}) = \mathbf{a_{um}}\mathbf{u_m} + \mathbf{a_{uw}}\mathbf{u_w} + \mathsf{a_{rm}}\mathsf{r_m} + \mathsf{a_{rw}}\mathsf{r_w}$. There is no constant term because, again, it would be useless for our purposes.

An intuitive way to choose the parameters is (*number of matched words - number of non-matched words*) = (*number of matched words - (total number of words - number of matched words)*) = (2 * *number of matched words - total number of words*) $= 2\mathbf{u_m} - \mathbf{u_w} + 2\mathsf{r_m} - \mathsf{r_w}$. This makes sense because if a new word is added, the score is increased/decreased if the new word matches/does not match. Therefore the score is higher for longer sentences, but only if more words actually match, otherwise shorter sentences have higher score. It turns out that penalizing non-matched words yields slightly better results in practice, so in the end we chose $\mathbf{a_{uw}} = \mathsf{a_{rw}} = -1.1$.

We performed some evaluation tests, available in the *dicio-evaluation* repository, [1] to justify the choice of scoring function and of parameters based on some real-world data. We constructed a dataset consisting of pairs (**user sentence**, reference sentence) associated with a unique ID and a list of IDs of other entries that this data point is supposed to be better than. The dataset contains 54 items,

[1] https://github.com/Stypox/dicio-evaluation

which in total generate 40 `"betterThan"` score comparisons. [2] For example:

```
{
  "weather1": {
    "user": "what s the weather in Rome",
    "ref": "what s the weather in ..",
    "betterThan": ["weather2"]
  },
  "weather2": {
    "user": "what s the weather in Rome",
    "ref": "what s the weather at ..",
    "betterThan": [/* ... */]
  },
  /* ... */
}
```

Each score comparison (e.g. `"weather1"` vs `"weather2"`) is considered to be correct if the score of the first user-ref pair is higher than the score of the second one. The score of a user-ref pair is calculated with an algorithm which tries all possible ways to match words and pieces of sentences, and returns a list of all possible scores. The best score in that list is considered the actual score of the user-ref pair being considered. Since the number of possible scores would be huge, making the algorithm really slow, heavy pruning is done by always keeping only the best score for every pair of character ranges in the two sentences. These greedy choices are guaranteed to still lead to the optimal score thanks to the properties of linear functions.

Table 4.1 shows how many score comparisons turned out wrong, out of the 40 score comparisons present in the dataset. The evaluation was run for various combinations of parameters.

Name	**Parameters**				**Wrong** (out of 40)
	a_{um}	a_{uw}	a_{rm}	a_{rw}	
Linear, balanced, 2 vs -1	2	−1	2	−1	3
Linear, balanced, 2 vs -1.1	2	−1.1	2	−1.1	**2**
Linear, balanced, 2 vs -1.3	2	−1.3	2	−1.3	**2**
Linear, balanced, 2 vs -0.8	2	−0.8	2	−0.8	4
Linear, more ref	2	−1.1	3	−1.65	**2**
Linear, more user	3	−1.65	2	−1.1	3
Linear, only ref	0	0	2	−1	10
Linear, only user	2	−1	0	0	22

Table 4.1: Evaluation of linear scoring functions

As the table suggests, all combinations of parameters that take into account both user and reference words result in a really good score, getting wrong only 2 to 4 score comparisons. However, increasing $*_w$ from 1.0 to 0.8 gives a worse performance, while decreasing it to −1.1 or −1.3 gives a better performance. So it makes sense to pick −1.1 (since decreasing past −1.1 does not seem to make a difference). Giving more weight to reference words also does not seem to make a difference, while giving more weight to user words gives a worse performance. However, since the dataset is really small, these considerations are not based on enough data, and a bigger dataset would be needed to better sustain these claims.

4.6 Other scoring functions

In order to make sure that linear functions were actually a good choice with respect to other possible functions, we also evaluated some non-linear functions. Note that in order to make the algorithm fast, we had to employ heavy pruning here, too. However, as proven in theorem 4.4.2, for functions not of the form $f(s) = g(h(s))$, doing greedy choices might not lead to the optimal result since remark 4.4.1 does not hold.

[2] https://github.com/Stypox/dicio-evaluation/blob/main/evaluation/src/test/resources/comparisons.json

Fortunately, this fact does not seem to impact our evaluation too much, because:

- The components of s_3 are most of the times multiples of 1.0 (i.e. the weight of one word), so there is not much variability.
- We tried to run the algorithm without pruning on the shorter sentences (where it is fast enough), and got all of the same results as by running with pruning.
- For some of the functions, we looked for a triple s_1, s_2, s_3 where remark 4.4.1 is false using the `z3` theorem prover [7]. We only found some specific cases where the greedy algorithm would produce sub-optimal results. For example, for $f(s) = (\mathbf{u_m} + \mathsf{r_m})/\sqrt{\mathbf{u_w} + \mathsf{r_w}}$, these tuples break remark 4.4.1, becuase $f(s_1) = 0.5 > f(s_2) = 0$, but $f(s_1 + s_3) \approx 1.76 < f(s_2 + s_3) \approx 1.78$:

$$s_1 : (\mathbf{u_{w1}} + \mathsf{r_{w1}} = 4 \wedge \mathbf{u_{m1}} + \mathsf{r_{m1}} = 1)$$
$$s_2 : (\mathbf{u_{w2}} + \mathsf{r_{w2}} = 1 \wedge \mathbf{u_{m2}} + \mathsf{r_{m2}} = 0)$$
$$s_3 : (\mathbf{u_{w3}} + \mathsf{r_{w3}} = 4 \wedge \mathbf{u_{m3}} + \mathsf{r_{m3}} = 4)$$

Another thing that is worth pointing out, is the fact that for all of these functions, putting the weights $(*_w)$ under square root (or to the power of 0.9) yields a better result. This is because, with this adjustment, the score increases if the length of the **user** and reference sentences increases, with an equal ratio of matching words over total words.

Name	**Weight exponent**	**Formula for score**	**Wrong** (out of 40)
Ratio	0.5	$\frac{\mathbf{u_m} + \mathsf{r_m}}{\sqrt{\mathbf{u_w} + \mathsf{r_w}}}$	**4**
	0.9	$\frac{\mathbf{u_m} + \mathsf{r_m}}{(\mathbf{u_w} + \mathsf{r_w})^{0.9}}$	9
	1	$\frac{\mathbf{u_m} + \mathsf{r_m}}{\mathbf{u_w} + \mathsf{r_w}}$	8
Sum of ratios	0.5	$\frac{\mathbf{u_m}}{\sqrt{\mathbf{u_w}}} + \frac{\mathsf{r_m}}{\sqrt{\mathsf{r_w}}}$	**4**
	1	$\frac{\mathbf{u_m}}{\mathbf{u_w}} + \frac{\mathsf{r_m}}{\mathsf{r_w}}$	9
Product of ratios	0.5	$\frac{\mathbf{u_m}\mathsf{r_m}}{\sqrt{\mathbf{u_w}\mathsf{r_w}}}$	**4**
	1	$\frac{\mathbf{u_m}\mathsf{r_m}}{\mathbf{u_w}\mathsf{r_w}}$	7
Intersection over union	0.5	$\left(1 + \frac{\sqrt{\mathbf{u_w}}}{\mathbf{u_m}} + \frac{\sqrt{\mathsf{r_w}}}{\mathsf{r_m}}\right)^{-1}$	**5**
	1	$\left(1 + \frac{\mathbf{u_w}}{\mathbf{u_m}} + \frac{\mathsf{r_w}}{\mathsf{r_m}}\right)^{-1}$	8

Table 4.2: Evaluation of non-linear scoring functions

5 Sentence matching algorithm

This chapter discusses the implementation of the scoring strategy devised in chapter 4 into the `score()` method from the `Skill` interface introduced in section 3.2. In particular, this chapter describes how reference sentences are represented conveniently (section 5.1), how they are converted to Kotlin objects (section 5.2), how the algorithm uses that data to calculate scores (section 5.3), the complexity of such algorithm (section 5.4) and how we tracked the performance improvements that we made (section 5.5).

5.1 Convenient representation of reference sentence alternatives

Each skill is free to implement the `Skill.score()` from section 3.2 method as it wants, which allows for a lot of flexibility. However, most skills in the Dicio application extend from `StandardRecognizer Skill` and share the same *standard* implementation, based on the scoring strategy described in chapter 4. Therefore each skill that uses the standard recognizer has a list of localized reference sentences to compare the **user sentence** to.

The list of all reference sentences is not written somewhere in full, because there would be too many sentences, and it would also be impossible for translators to come up with such a long list. For example, the number of alternative sentences of all skills in the English language is 1721. [1] Therefore we developed a regex-like language that allows packing sentences together, available in the *dicio-sentences-compiler* repository. [2] So for example the translators can write `hello|hi how (are you doing?)|(is it going)` and it would match perfectly with these 6 **user sentences**: "hello how are you", "hello how are you doing", "hello how is it going", "hi how are you", "hi how are you doing", "hi how is it going". These are all of the constructs available in the regex-like language:

- **word** (e.g. `hello`). A simple word: it can contain unicode letters, and the matching will be case insensitive. Diacritics and accents will be ignored while matching, unless the word is wrapped in quotes. E.g. `hello` matches "hèllo", "héllò", ... while `"hello"` matches only the exact "hello".
- **word with variations** (e.g. `<e|g?>mail`). A word with possible variations. This construct is not so useful in the English language, but comes in handy in languages where words can have multiple declensions, such as Italian. A variations group, i.e. the piece included in the angle brackets `<>`, is a set of variations separated by `|`, and optionally followed by a `?` indicating the empty variation. A word can have multiple variation groups. E.g. `<e|g?>mail` matches "email", "gmail" and "mail".
- **or-red constructs** (e.g. `hello|hi`). Any of the or-red construct could match. E.g. `hello|hi| hey assistant` matches "hello assistant", "hi assistant" and "hey assistant".
- **optional construct** (e.g. `hello?`). This construct can be skipped during matching. E.g. `bye bye?` matches both "bye" and "bye bye".
- **composite parenthesized construct** (e.g. `(hello)`). This allows grouping constructs together, just as math parenthesis do. E.g. `how (are you doing?)|(is it going)` matches "how are you", "how are you doing" and "how is it going".
- **capturing group** (`.NAME.`). This tells the matching algorithm to match a variable-length list of any word to that part of the reference sentence. `NAME` is the name of the capturing group. E.g. `how are you .person.` matches "how are you Tom" with "Tom" in the "person" capturing group.

[1] Calculated with the `count` subcommand of dicio-sentences-compiler, passing as the input all of the `app/src/main/sentences/en/*.dslf` files of commit `396bc28fd47f496826c0041fd8004f4d0c9b3947` in dicio-android

[2] https://github.com/Stypox/dicio-sentences-compiler

5.2 From Yaml data to Kotlin code

The data pertaining to the standard recognizer, including translated sentences, is stored under the path `app/src/main/sentences/` in the Dicio application. Each skill that wishes to use the standard recognizer has an entry in the `skill_definitions.yml` file, which describes the kinds of sentences it can interpret, along with the capturing group names and types that each sentence may have. For example, this is the definition of the timer skill, which offers the ability to set, cancel and query timers.

```yaml
- id: timer
  specificity: high
  sentences:
    - id: set
      captures:
        - { id: duration, type: duration }
        - { id: name, type: string }
    - id: cancel
      captures: [ { id: name, type: string } ]
    - id: query
      captures: [ { id: name, type: string } ]
```

The actual reference sentences for each language are stored in a `LANGUAGE/SKILL_ID.yml` file, and they must abide by the definition of the skill given in `skill_definitions.yml`, so there cannot be additional sentence IDs or unknown capturing group names. For example the English sentences for the timer skill, a part of which is shown below, are in `en/timer.yml`:

```
set:
  - timer|(ping me in) .duration.
  - create a? time<r|rs?> (for|of .duration.)? (called .name.)?
cancel:
  - cancel|terminate the? time<r|rs?> (called .name.)?
query:
  - how long|(much time) is left? on the? time<r|rs?> (called .name.)?
```

Having a regex-like language to pack many sentences into one, and storing this data in Yaml files, makes it simple for the community to translate the sentences to other languages (Dicio is already available in 9 languages!). In order for this data to be accessible from the Kotlin code at runtime, we developed the sentences compiler plugin for the Gradle build system (read more in section 6.5), which parses the sentences in the Yaml files and generates Kotlin code.

The generated code also takes care of extracting the sentence ID and the capturing groups from a match between a reference and a user sentence (i.e. a `StandardScore` instance, described in section 5.3). In order to enforce type safety in how the extracted data is used, the plugin generates a *sealed* class for each skill, and then one subclass for each of the sentence IDs that the skill can have. Each sentence ID class then has a nullable field for each of the capturing groups that the sentence may have. The sealed class for a skill (and its subclasses) is generated based on the corresponding skill definition in `skill_definitions.yml`, and is meant to be used as the `InputData` type parameter when extending the `Skill` interface (see section 3.2).

```kotlin
// code generated from the timer definition in skill_definitions.yml:
sealed interface Timer {
    data class Set(val duration: Duration?, val name: String?) : Timer
    data class Cancel(val name: String?) : Timer
    data class Query(val name: String?) : Timer
}

// example usage in the timer skill implementation (Skill<Timer>):
fun generateOutput(ctx: SkillContext, inputData: Timer): SkillOutput {
    return when (inputData) {
        is Timer.Set -> setTimer(inputData.duration, inputData.name)
        is Timer.Query -> queryTimer(inputData.name)
        is Timer.Cancel -> cancelTimer(inputData.name)
    }
}
```

5.3 Implementation of the algorithm with Constructs

As explained in the previous section, the generated code for each skill contains a map between languages and lists of reference sentences. Each sentence is actually a construct, and as explained in the list in section 5.1, there are many construct types available, some of which recursively contain more constructs: `WordConstruct`, `OrConstruct`, `OptionalConstruct`, `CompositeConstruct`, `CapturingConstruct`.

Each of these classes inherits from the `Construct` interface, and thus implements the `matchToEnd()` method which takes care of matching and calculating scores. We designed the API of `matchToEnd()` to allow for the fastest algorithm, and to be flexible, since we want to support more construct types in the long run (for example, a `DurationCapturingConstruct`, which would extract a duration).

```
fun matchToEnd(memToEnd: Array<StandardScore>, helper: MatchHelper)
```

- The `helper` argument contains information about the **user sentence**, along with various precalculated tokenizations to speed up the matching process. For example, the starting indices of all words in the **user sentence**, or the cumulative **user weight** array $\mathbf{C_{uw}}$ such that $\mathbf{C_{uw}}[\mathbf{i}]$ is the weight of the characters in the $[0, i)$ range in the **user sentence**.

- The `memToEnd` argument is an array of `StandardScore` and is both an input and an output to the function. `StandardScore` implements the `Score` interface, and contains $(\mathbf{u_m}, \mathbf{u_w}, \mathsf{r_m}, \mathsf{r_w})$ and any data captured by capturing groups. When the function is called, the ith element of the `memToEnd` array represents the best score obtained by matching the **user sentence** from the ith character *until the end*, to *all* constructs that come *after* the current one. When the function returns, this array must have been updated to include the score gains or losses caused by additionally matching the current construct, in a bottom-up fashion.

Before starting to match a sentence, the `memToEnd` array is initialized with $\mathbf{u_m} = \mathsf{r_m} = \mathsf{r_w} = 0$ everywhere, and with the ith $\mathbf{u_w}$ set to $\mathbf{C_{uw}}[\text{end}] - \mathbf{C_{uw}}[\mathbf{i}]$, i.e. the cumulative **user weight** from the ith character of the **user sentence** until the end.

Remember that increasing $\mathbf{u_m}$ and $\mathsf{r_m}$ increases the score, while increasing $\mathsf{r_w}$ and $\mathbf{u_w}$ decreases it:

$$f(\mathbf{u_m}, \mathbf{u_w}, \mathsf{r_m}, \mathsf{r_w}) = 2\mathbf{u_m} - 1.1\mathbf{u_w} + 2\mathsf{r_m} - 1.1\mathsf{r_w}$$

This is how every construct implements `matchToEnd()`:

- `WordConstruct`(text, construct weight $\overline{\mathsf{w}}$) increases $\mathsf{r_w}$ for every element in `memToEnd` by $\overline{\mathsf{w}}$, temporarily lowering the score as there is one more unmatched reference word. It then checks if there is a word in the **user sentence** starting at any position i that matches text. For every match that is found, the following operations are performed on a copy of `memToEnd[i + word.length]` (i.e. the best score immediately after the end of the word) to reflect that this reference word can be matched: $\mathbf{u_m}$ and $\mathbf{u_w}$ are increased by $\mathbf{C_{uw}}[\mathbf{i} + \text{word.length}] - \mathbf{C_{uw}}[\mathbf{i}]$, and $\mathsf{r_m}$ is increased by $\overline{\mathsf{w}}$. Finally, `memToEnd[i]` is updated to the best score between its current value and the newly calculated score.

- `OrConstruct` calls `matchToEnd()` on all of its sub-constructs with *a copy* of `memToEnd`, and then updates the *original* `memToEnd` such that `memToEnd[i]` is the best score among all of the `memToEnd[i]` copies mutated by sub-constructs.

- `OptionalConstruct` only exists as part of an `OrConstruct` and does not contain any sub-construct, so its `matchToEnd()` leaves `memToEnd` untouched, since there are no gains nor losses from matching nothing. This makes it so that, if no other constructs in the parent `OrConstruct` match, the `memToEnd` resulting from the `OptionalConstruct` would turn out to be the best, since it would result in lower $\mathsf{r_w}$ and thus higher score.

- `CompositeConstruct` matches its sub-constructs one after the other from the last to the first on *the same* `memToEnd`. This construct in particular shows how the algorithm is bottom-up.

- `CapturingConstruct`(capturing group name, construct weight $\overline{w}$) increases r_w for every element in `memToEnd` by $\overline{w}$, temporarily lowering the score as there is one more unmatched reference capturing group. Then, for every starting index i and ending index $j > i$, these operations are performed on a copy of `memToEnd[j]` (i.e. the best score immediately after the end of the capturing group) to reflect that this capturing group can be matched: $\mathbf{u_m}$ and $\mathbf{u_w}$ are increased by $\mathbf{C_{uw}}[\mathbf{i} + \text{word.length}] - \mathbf{C_{uw}}[\mathbf{i}]$, r_m is increased by $\overline{w}$, and the string range $[i, j)$ is added to the captured data. Finally, `memToEnd[i]` is updated to the best score between its current value and all of the newly calculated scores for every j.

5.4 Complexity of the algorithm

Let us consider $\mathbf{n}$ as the length of the **user sentence**, and m as the total number of constructs in the reference sentence. All constructs mutate `memToEnd` (whose length is $\mathbf{n} + 1$) in $\mathcal{O}(\mathbf{n})$:

- `WordConstruct` does at most 1 comparison per character in the user sentence, and $\mathcal{O}(1)$ operations at every index i thanks to precalculated word indices, yielding a complexity of $\mathcal{O}(\mathbf{n})$.
- `CapturingConstruct` calculates the best possible ending index $j > i$ for a capturing group *starting* at index i in $\mathcal{O}(1)$ without the need to actually check all j, by processing the user input from the end to the beginning and keeping track of the best index where a capturing group can *end*. Thanks to this trick the complexity is also $\mathcal{O}(\mathbf{n})$.
- `OrConstruct` and `CompositeConstruct` just call `matchToEnd()` on their $\overline{m}$ sub-constructs, possibly cloning `memToEnd` in $\mathcal{O}(\mathbf{n})$ for at most $\overline{m}$ times. This adds a complexity of $\mathcal{O}(\mathbf{n})$ to each of the sub-constructs, but this does not matter because the calls to the `matchToEnd()` of the sub-constructs would also be $\mathcal{O}(\mathbf{n})$ each. Therefore the overall complexity would be $\mathcal{O}(\mathbf{n}\overline{m})$, but note that this complexity is distributed over $\overline{m}$ constructs.

The overall complexity of the algorithm is therefore $\mathcal{O}(\mathbf{n}m)$. This is the same complexity as the Levenshtein string distance algorithm described in section 4.2, and is optimal, unless the SETH conjecture is false [4]. To achieve a lower complexity, an approximated algorithm would be needed.

5.5 Incremental performance improvements

During the development of the optimal algorithm for matching a reference sentence to a **user sentence** we made improvements incrementally. The complexity went from exponential, to $\mathcal{O}(\mathbf{n}^3 m)$, to $\mathcal{O}(\mathbf{n}^2 m)$ (like the implementation in the previous NLU), and finally to $\mathcal{O}(\mathbf{n}m)$. At various steps we also applied optimizations that did not change the complexity, but just improved the constant factor, by analyzing execution flamegraphs and optimizing the code sections where the most time was spent.

Step 0 - $\Omega(\mathbf{n}^{2m})$ in the worst case. The interface of `Construct` initially used to take i and $j \geq i$ as input, and return the best score obtainable by matching only the interval $[i, j)$ in the **user sentence**. This top-down definition was easier to understand than the bottom-up `matchToEnd()`, at the cost of performance. The initial algorithm was exponential in the number of constructs because the `CompositeConstruct` would calculate every possible way to arrange its sub-constructs by choosing $[i, j)$ for the first sub-construct in $\approx \mathbf{n}^2$ steps, and then recursively recalculating the score for the $\overline{m} - 1$ remaining constructs, thus performing to $\mathbf{n}^{2\overline{m}}$ choices of indices.

Step 1 - $\Omega(\mathbf{n}^{2m})$ in the worst case. We optimized the performance by $\approx 30\%$ by storing capturing groups inside `Score` objects in a list instead of a hash map. Operations on a hash map are $\mathcal{O}(1)$, but with a high constant factor, and therefore switching to a list made a significant difference.

Step 2 - $\mathcal{O}(\mathbf{n}^3 m)$. After adding a cache of size $\mathbf{n}m$ to the recursive algorithm in the `CompositeConstruct`, the complexity became polynomial. Since there are $\approx n^2$ ways to choose i and j as the inputs for a `CompositeConstruct`, and that the size of the cache of capturing constructs is $\mathbf{n}m$, the overall complexity is $\mathcal{O}(\mathbf{n}^3 m)$.

Steps 3-6 - $\mathcal{O}(n^3m)$. We made a few optimizations, such as reducing the number of instantiated JVM objects (step 4) or precalculating where words can match (step 6).

Step 7 - $\mathcal{O}(n^2m)$. We noticed that the initial algorithm could not improve past the $\mathcal{O}(n^3m)$ complexity, so we came up with the `matchToEnd()` method taking `memToEnd` as parameter. After rewriting all `Constructs` to implement this new bottom-up interface, the complexity got down to $\mathcal{O}(n^2m)$, and even dropped to $\mathcal{O}(nm)$ for reference sentences without capturing groups.

Step 8 - $\mathcal{O}(nm)$. After applying the trick described at the bottom of section 5.3 to the `Capturing Construct` implementation, the complexity reached the optimum.

Figures 5.1 and 5.2 plot the time improvements with a fixed **user sentence** length, while Figures 5.3 and 5.4 plot the improvements in the number of **characters** processed per second. For the current time plots, there is no difference between steps 7 and 8, because the current time sentences do not contain capturing groups. The data is in the dicio-android repo under `skill/benchmarks/` and was obtained with `skill/src/test/.../PerformanceTest.kt`; plot with `skill/plot_benchmarks.py`.

The *inexact* algorithm in the old Dicio codebase was an optimized $\mathcal{O}(n^2m)$ top-down approach.

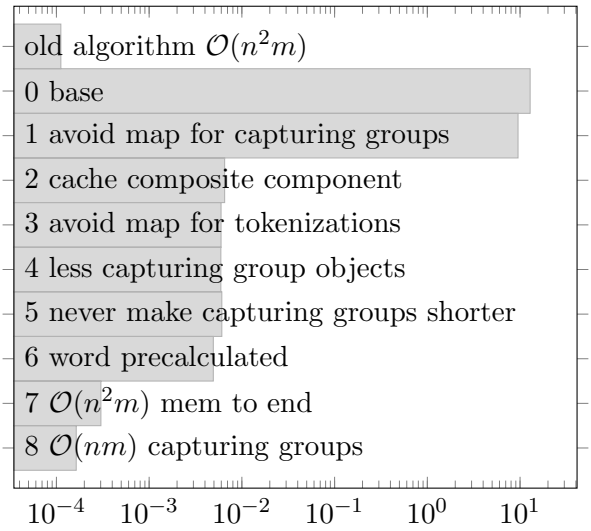

Figure 5.1: Seconds it took the **timer** reference sentences to match a 19-character **user sentence** (log scale, lower is better)

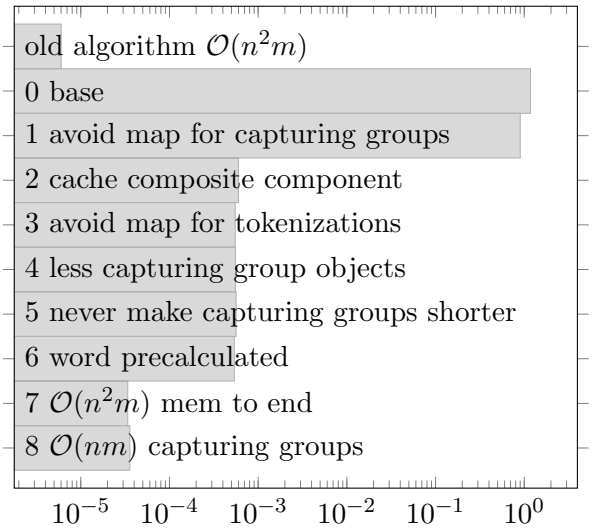

Figure 5.2: Seconds it took the **current time** reference sentences to match a 40-character **user sentence** (log scale, lower is better)

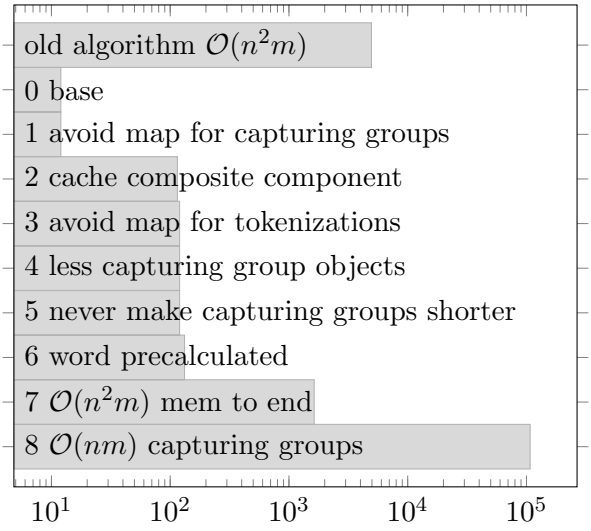

Figure 5.3: Max length of **user sentence** processed by the **timer** reference sentences in 1s (log scale, higher is better)

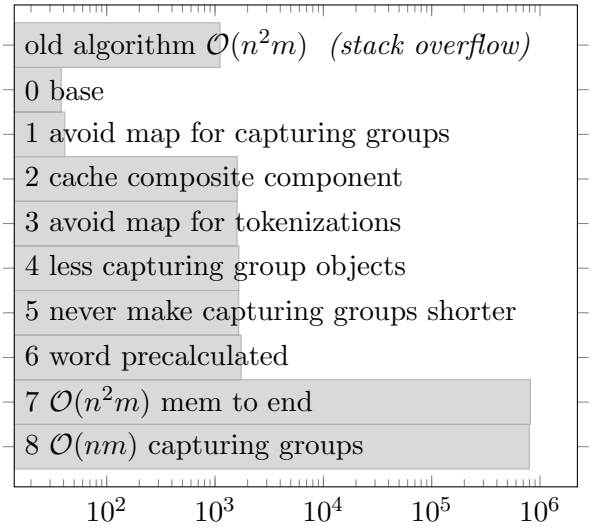

Figure 5.4: Max length of **user sentence** processed by the **current time** reference sentences in 1s (log scale, higher is better)

6 Modern Android code

This chapter illustrates the various steps that we took to migrate the Dicio Android application to new and modern technologies. The original codebase dated back to 2019 and did not respect good Android app design standards even from back then. The migration took just ≈ 80 hours to complete.

We made these changes: we converted the codebase from Java to Kotlin (section 6.1), we migrated the UI from the imperative Android Views to the declarative Jetpack Compose and picked up the MVVM architecture (section 6.2), we introduced dependency injection to better separate components and provide state to composables (section 6.3), we ditched SharedPreferences in favour of DataStore (section 6.4), we applied a few build system overhauls (section 6.5). The last section, 6.6, explains the implementation of a particularly complicated component: the Speech To Text button.

6.1 From Java to Kotlin

The only programming language available for Android development was Java up until 2017, when Google announced plans to also support Kotlin. [1] Kotlin really took off within the developer community only after 2019, when Google adopted a Kotlin-first approach. [2]

Since the first commit to the Dicio codebase dates back to 2019, and Kotlin was not widespread yet, back then we naturally chose Java as the programming language for the project. In more recent years, though, the advantages of Kotlin over Java have become apparent: [3]

- Kotlin avoids `NullPointerException`s since nullable types are embedded into the type system. For example `Type?` can take values of type `Type`, or the value `null`. To turn a `Type?` into a `Type` the programmer needs to explicitly assert non-nullity (e.g. `obj!!`) or use a safe call (e.g. `obj?.doSomething()`, which calls `doSomething()` only if `obj != null`). Instead Java has no built-in null safety, which often leads to crashes, although `@Nullable` and `@NonNull` annotations can help.

- Kotlin has a more powerful function system than Java, with inline lambdas, properties (which automatically create getters and setters), operator overloading, extension functions.

- Kotlin also has various shortands for creating singleton classes, data classes, union types.

- Kotlin compiles to JVM bytecode just like Java, and an effort was made to make Java and Kotlin almost fully interoperable, allowing to migrate apps one file at a time.

- Kotlin does not have checked exceptions, which is kind of a disadvantage, but speeds up development.

Therefore the first step in migrating Dicio to modern techologies was to convert all of the code from Java to Kotlin. Fortunately this could be done mostly automatically thanks to the tools provided by Android Studio, leaving only a few errors and broken references to solve.

Furthermore, the old Dicio codebase used a Reactive Programming framework named RxJava3 [15] to achieve concurrency. This was needed, for example, to analyze the user input and perform network requests without blocking the UI thread. During the migration phase it made sense to switch to Kotlin *coroutines* [12] instead, which are built into the language itself. Coroutines implement the `async/await` paradigm at the language level, making them more flexible and easy to reason about.

[1] Android Developers Blog: Celebrating 5 years of Kotlin on Android, https://android-developers.googleblog.com/2022/08/celebrating-5-years-of-kotlin-on-android.html

[2] Android Developers: Android's Kotlin-first approach, https://developer.android.com/kotlin/first

[3] Kotlin foundation: Kotlin comparison to Java, https://kotlinlang.org/docs/comparison-to-java.html

For example, here is some Java code which uses RxJava3. Its purpose is to run `skill.processInput()` in the background, and after it has finished call either `generateOutput()` or `onError()` on the main thread. (this code uses an old version of the `Skill` interface)

```
Single.fromCallable(() -> {
    skill.processInput();
    return skill;
})
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(this::generateOutput, this::onError);
```

Here is the equivalent Kotlin code that uses coroutines. It uses standard coding structures (e.g. `try-catch`), and makes it more explicit what is running on the UI thread.

```
val scope = CoroutineScope(Dispatchers.IO)
/* ... */
scope.launch {
    try {
        skill.processInput()
        activity.runOnUiThread { generateOutput(skill) }
    } catch (throwable: Throwable) {
        activity.runOnUiThread { onError(throwable) }
    }
}
```

6.2 From Android Views to Jetpack Compose

The traditional Android development employed an *imperative* approach to UI. The initial view tree to show on the screen was instantiated based on the structure stored in an XML file, and this tree was mutated throughout the app lifecycle via Java code. For example, to obtain a reference to a view on the screen, the Java function would have been `findViewById()`, in a similar fashion to `document.findElementById()` in HTML+JS. This setup, however, had many problems: it required developers to always work on multiple files, it made it difficult to reuse code, it was open to programmers using wrong view IDs causing crashes at runtime, and most importantly it required to keep the view tree in sync with the application state manually.

Therefore in 2019 Google introduced Jetpack Compose [10], a *declarative* toolkit for UI inspired by ReactJS and Flutter, and made it stable in 2021. In the declarative approach, the UI is a *pure function* of the application state (a so-called *`@Composable` function*), making the application state the single source of truth. Whenever the state changes, the function is called to obtain the new UI, i.e. a *recomposition* happens. While this process may seem really slow and wasteful, frameworks employ heavy optimizations to only recompose the parts of the UI corresponding to what changed in the state, making the performance on par with the imperative approach. `@Composable` functions are written entirely in Kotlin and within their body they just call other @Composable functions to build the UI as desired, allowing to easily split the code in many reusable components. The most basic components, like buttons or text areas, are usually provided by libraries, for example the Material Design 3 library.

Here is an example app with a counter that can be increased by pressing on a button (no styling is included to keep the code short):

```
<!-- This code would be inside res/layout/main_activity.xml -->
<LinearLayout>
    <Button
        android:id="@+id/button_id"
        android:text="Click here" />
    <TextView
        android:id="@+id/text_id" />
</LinearLayout>
```

```java
// This code would be inside MainActivity.onCreate()
setContent(R.layout.main_activity)
Button button = findViewById<Button>(R.id.button_id)
TextView text = findViewById<TextView>(R.id.text_id)
button.setOnClickListener(v -> {
    // assume counter is a field of the MainActivity class
    counter += 1;
    text.setText("Counter is: " + counter);
});
text.setText("Counter is: " + counter);
```

The equivalent Jetpack Compose code is much shorter and lies in only one file. Moreover, there is no manual call to `text.setText()` every time the counter changes, which the developer might forget to include.

```kotlin
@Composable
fun MainScreenWithState() {
    var counter by rememberSaveable { mutableIntStateOf(0) }
    MainScreen(counter, { counter += 1 })
}

@Composable
fun MainScreen(counter: Int, increaseCounter: () -> Unit) {
    Column {
        Button(onClick = increaseCounter) {
            Text("Click here")
        }
        Text("Counter is: " + counter)
    }
}
```

The Dicio app was originally written using the legacy Android Views and so had to be migrated to Jetpack Compose. This migration step was the one which took the longest, since it required rethinking how to handle state throughout the whole app, as the previous code had no separation between UI and business logic. For the rewritten code, however, we picked the MVVM (Model-View-Viewmodel) architecture, encouraged by the good `ViewModel` support provided by Jetpack Compose.

Here is an example of the same `MainScreen`, but controlled by a `ViewModel`.

```kotlin
class MainScreenViewModel : ViewModel() {
    private val _counterState = MutableStateFlow(0)
    val counterState: StateFlow<Int> = _counterState.asStateFlow()

    fun increaseCounter() {
        _counterState.update { counter ->
            return@update counter + 1
        }
    }
}

@Composable
fun MainScreenWithState() {
    val viewModel: MainScreenViewModel = viewModel()
    val counter by viewModel.counterState.collectAsState()
    MainScreen(counter, viewModel::increaseCounter) // see above
}
```

This design separates the logic that increases the counter from the UI, and makes future changes much easier (some examples of future changes may be to allow resetting the counter, to control the counter via a notification, ...). One notable thing is the use of `MutableStateFlow`, which makes it possible for the UI layer to listen to changes (via the `collectAsState()`), and also handles atomic `update` operations. Moreover, Kotlin flows support mapping, filtering and other operations on their items via coroutines, which would make it easy to e.g. create a second screen that only shows even counter values, while making sure that recompositions only happen when the even number changes.

Another advantage of Jetpack Compose is the convenient tooling available right from the Android Studio IDE, which shows previews (i.e. `@Composable` functions also annotated with `@Preview`) and can test the UI for common issues without even starting an Android emulator. Furthermore, the Compose toolkit is independent of the components implemented on top of it, making it simple to switch UI design. For the new UI of the Dicio assistant we adopted the components from the Material 3 library (see the difference with the previous Material 2 design in fig. 1.1).

6.3 Dependency Injection with Hilt

Dependency Injection (DI) is a design pattern used in object-oriented programming that allows an object to receive its dependencies from an external source rather than creating them itself. This promotes more modular code, eliminates boilerplate factory classes, and allows replacing real instances of services with fake ones at test time.

Dicio extensively uses the Hilt library [20] for dependency injection. Injectable classes are declared by adding the `@Inject` annotation to their constructor. Each injectable class (and a few other components, e.g. `ViewModel`s, activities and fragments) can in turn have some injectable fields that will be automatically built and provided by Hilt upon instantiation. It is possible to make an injectable class a singleton with the `@Singleton` annotation, which is often useful for app-wide services that need to be accessed from multiple places.

For example, the home screen uses a Hilt view model, which depends on `SkillHandler` and `SkillEvaluator`, which in turn uses `SkillHandler` too. Moreover `SkillEvaluator` is also used in `MainActivity`. And the graph of dependencies can get even worse than this, as the codebase grows, but Hilt keeps things tidy, as the following code shows:

```
@AndroidEntryPoint // <- Hilt needs an entry point to be able to fulfill @Injects
class MainActivity : BaseActivity() {
    // automatically obtains the singleton SkillEvaluator when MainActivity is setup
    @Inject lateinit var skillEvaluator: SkillEvaluator
}

@Composable
fun HomeScreen() {
    // automatically obtains a Hilt view model for use in a @Composable
    val viewModel: HomeScreenViewModel = hiltViewModel()
}

@HiltViewModel // <- makes sure the lifecycle of the view model is handled correctly
class HomeScreenViewModel @Inject constructor(
    val skillHandler: SkillHandler,
    val skillEvaluator: SkillEvaluator,
) { /* ... */ }

@Singleton // <- this is a singleton, so Hilt will instantiate this class only once
class SkillEvaluator @Inject constructor(
    private val skillHandler: SkillHandler,
) { /* ... */ }

@Singleton
class SkillHandler @Inject constructor() { /* ... */ }
```

6.4 From SharedPreferences to Protobuf-backed DataStore

`SharedPreferences` were the standard way to store user settings in Android apps up until recently, when Google started suggesting `DataStore` [9] instead.

`SharedPreferences` were basically just an XML file on disk with key-value pairs, where the keys were strings, and the values could be any Java built-in type plus string sets. The Android ecosystem provided ways to easily read and write those values, but there were a few shortcomings, such as blocking I/O, contrived setup to listen to changes, hardcoded string keys, and no built-in way to encode enum values.

`DataStore` solves these problems by providing an API based on Kotlin coroutines, that allows

getting values asynchronously and makes it simple to listen to changes. In order to make the stored values type-safe, and eliminate the need for hardcoded string keys, `DataStore` allows storing the data on disk as a `.pb` Protobuf-encoded file. Each Protobuf-encoded file corresponds to a `.proto` file processed at compilation time, which contains type and enum definintions and describes the binary representation of data on disk.

For example, this might be a `.proto` file for user settings in an app like Dicio:

```
message UserSettings {
    Theme theme = 1;
    bool show_speech_to_text_button = 2;
    map<string, bool> enabled_skills = 3;
}
enum Theme { THEME_LIGHT = 0; THEME_DARK = 1; /* ... */ }
```

Then, in the app code, the settings can be read by just accessing the `.data` field of a `DataStore` object. Listening to changes, e.g. inside a `@Composable`, boils down to manipulating the Kotlin flow:

```
val dataStore: DataStore<UserSettings> = DataStoreFactory.create(/* ... */)

@Composable
fun SttButtonIfEnabledInSettings() {
    val showSttButton = dataStore.data // this is a Kotlin flow
        .map { settings -> settings.showSpeechToTextButton }
        .collectAsState(initial = true) // value to use while settings are being loaded

    if (showSttButton) {
        Button(onClick = /* ... */) {
            Text("Speech to text button")
        }
    }
}
```

6.5 The Gradle build system

Android apps are setup and built using Gradle [11], a build system based on the JVM. Keeping Gradle build files up-to-date requires some maintenance, since very often there are new features that need to be adopted and old functionalities become deprecated.

For example, the recently introduced Version Catalogs provide a way to keep the versions of all dependencies in a single `.toml` file, which helps avoid version clashes and forgetting library updates. Therefore the dependencies of Dicio are now listed in a Version Catalog, for example:

```
# in gradle/libs.versions.toml
[versions]
datastore = "1.1.1"
[libraries]
androidx-datastore = { module = "androidx.datastore:datastore", version.ref = "datastore" }

// in app/build.gradle.kts
dependencies {
    implementation(libs.androidx.datastore)
}
```

A major overhaul of Gradle happened when the suggested language of build files changed from Groovy DSL to Kotlin DSL (where DSL stands for "Domain Specific Language"). Groovy DSL is a dynamically typed language with a rather lenient syntax, making build scripts easy to read but hard to maintain: for example, variables can be assigned without `=`, and functions can be called without `(...)`, making assignments and function calls easy to confuse. Kotlin DSL, on the other hand, is statically typed and uses the concise but strict Kotlin syntax rules. Therefore all Dicio build files were migrated from Groovy DSL (`.gradle` files) to Kotlin DSL (`.gradle.kts` files). For example, these snippets perform the same actions, but the first is in Groovy DSL and the second in Kotlin DSL:

```
minifyEnabled false
proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
```

```
isMinifyEnabled = false
proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"), "proguard-rules.pro")
```

In order to understand what the user says, Dicio has a list of possible sentences in various languages, as described in section 5.2. This list needs to be directly accessible from Kotlin code for maximum performance, but it would be cumbersome for community translators to translate Kotlin files directly. Therefore the Kotlin lists are generated at compile-time based on some `.yaml` files stored under `app/src/main/sentences`. This is done by a Gradle plugin we created, `sentences-compiler-plugin`, that properly handles build task dependencies and only triggers recompilation when something changes. The plugin uses the KotlinPoet library [19] to generate correct Kotlin code.

6.6 Speech To Text button

One of the most complicated components of the Dicio app is the Speech To Text (STT) button at the bottom of the home screen. It needs to handle various operations: downloading, unzipping and loading the STT model while showing progress to the user; listening to the user speech and providing that information to other app components; handling language changes; acquiring the microphone permission; reporting errors. The button icon and label need to reflect the current state of the STT input device, so the user knows what is going on and what to expect when clicking on the button. Clicks on the button need to be perform different actions based on the current state, e.g. initiating the model download or starting to listen.

To perform STT we chose Vosk [2], a project that provides small STT models that can be run locally with good results, along with the libraries needed to actually run the models. Vosk models need to be downloaded, unzipped and loaded before they can be used.

In order to handle the STT button complexity correctly, we designed the state machine shown in fig. 6.1, which we implemented for the Vosk STT input device. [4] Bold arrows indicate user clicks on the button, while normal arrows indicate automatic processes (e.g. the download finishes or the STT input device detects silence and stops listening). Error states are greyed out so that they create less confusion. The *NoMicrophonePermission* state is handled separately in the UI layer, and is thus not included here.

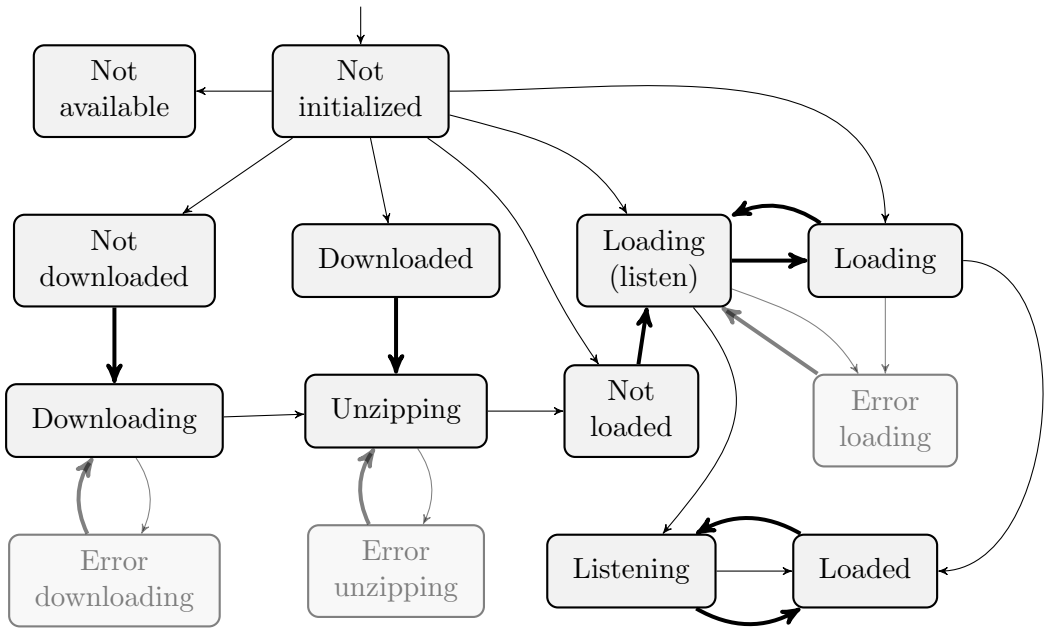

Figure 6.1: Vosk Speech To Text input device state machine

A few notes about the non-obvious state transitions:

[4] `app/src/main/kotlin/org/stypox/dicio/io/input/vosk/VoskInputDevice.kt`

- When the app is started, the STT input device is created with *NotInitialized* as the initial state. As soon as the app language is known, the state is set to:
  - *NotAvailable* if the STT input device does not support the current language
  - *NotDownloaded* if the previously downloaded model was not in the current language
  - *Downloaded* if a `.zip` is found on disk
  - *NotLoaded* or *Loading* if the model is found on disk (the actual state and the value of `Loading.thenStartListening` are chosen depending on the way the app was started and on user settings)
  - *NotDownloaded* otherwise
- The *Downloading* and *Unzipping* states hold information about the progress of the operation. When downloading finishes, unzipping starts directly without manual intervention.
- *Loading* comes in two possible states: with `thenStartListening` either true or false. When it is true (*Loading (listen)*), after loading has finished, the STT input device will immediately start listening. If the user clicks on the STT button while in the *Loading* state, `thenStartListening` will alternate between true and false.
- *Loaded* and *Listening* states own the loaded STT model themselves, so that even if the state changes in an unexpected way, the STT model is garbage collected along with the state object and does not create memory leaks.
- While in the *Listening* state, the STT input device reports user utterances and other events to a callback provided by the last caller of the `onClick` method (i.e. the method that the UI layer must call whenever the button is clicked). This is to allow using the STT input device from multiple places in the app at the same time, i.e. in the main screen as an input to the assistant, and in the Speech To Text service pop-up window.

Figure 6.2 shows what the button looks like for all of the states. On the first row: *NoMicrophonePermission*, *NotInitialized*, *NotAvailable*, *NotDownloaded*. On the second row: *Downloading*, *ErrorDownloading*, *Downloaded*, *Unzipping*. On the third row: *ErrorUnzipping*, *NotLoaded*, *Loading* with `thenStartListening` set to true, *Loading* with `thenStartListening` set to false, *ErrorLoading*, *Loaded*, *Listening*,

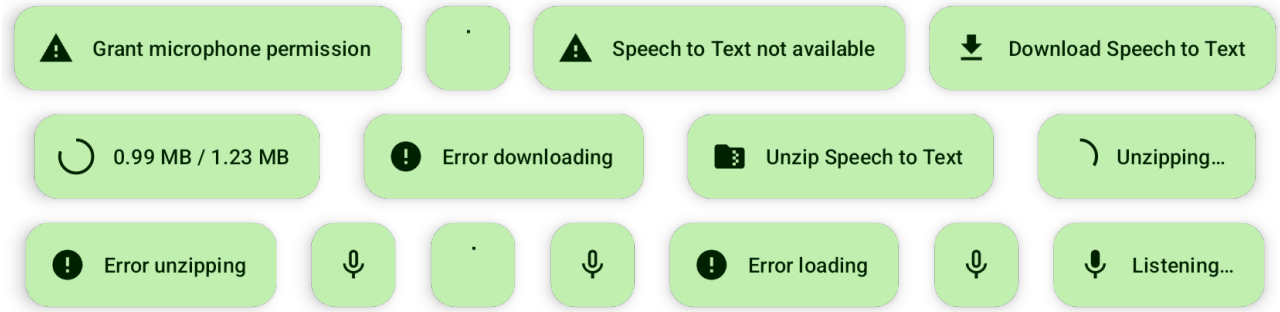

Figure 6.2: Speech To Text button previews

7 Conclusion

This chapter analyzes related work (section 7.1), lists future work that could be done to further improve Dicio (section 7.2) and finally draws conclusions about the thesis as a whole (section 7.3).

7.1 Related work

Commercial assistants, such as Google Assistant, Siri or Alexa, rely on online NLU systems that are far more complex and powerful than the one described in this thesis. They support a wider range of skills and integrate better in the system. However, they rely on cloud services, and started moving some computation offline only in recent years with many limitations and little privacy guarantees. [1]

Rhasspy [18] and Mycroft/OVOS [17] are two full-featured and community-developed free software assistants. Unfortunately, they are written in Python and are meant to be installed on a computer.

Some lightweight NLUs available online like *nlp.js* [3] and *Mycroft Padatious* [16] use small neural networks trained on reference sentences, but they lack rigorous assessment. On the other hand, *Snips* [6] has been evaluated on [5] and other datasets, and its performance is comparable to commercial proprietary cloud-based solutions. Other alternatives include *CoreNLP* [14], *NLTK*, *spaCy* and *rhasspy-nlu*. We created a new NLU from scratch for Dicio because many of these options are limited to only some natural languages, are difficult or slow to run on Android, or are not flexible enough.

Aas et al. [1] provide relevant insights into advanced on-device NLU techniques that might be worth exploring, in case even the new NLU system in Dicio falls short of expectations.

7.2 Future work

Moving forward, the dicio-numbers [2] library will provide `Construct` implementations that match numbers, dates and durations, to make it possible to e.g. distinguish "What's Trento" from "What's one plus one" and trigger the search and calculator skills, respectively.

Thanks to the flexibility of the new architecture for skills and for the Dicio app, it will be possible to save interactions to disk, add new reference sentences at runtime, match words fuzzily, and more. [3]

A more rigorous evaluation of the NLU with more data is essential for future work.

The recent developments in the field of LLMs showed that they can be small enough to run on phones and still provide decent performance. They could be integrated in Dicio to handle complex NLU cases, or for specific tasks such as summarization.

7.3 Conclusion

This thesis focused on resolving the two main problems of the Dicio free software assistant for Android: the outdated codebase, and the unextensibility and suboptimal performance of the Natural Language Understanding system. To address the first issue, we migrated to Kotlin and modern libraries, we adopted declarative UI and we improved the architecture of skills. To address the second, we rigorously devised a scoring function based on a mathematical proof and on an evaluation framework, and implemented an $\mathcal{O}(\mathbf{nm})$ algorithm to compare **user** and reference sentences.

In conclusion, we believe that Dicio, with its simple yet capable NLU, is becoming a valid on-device alternative to proprietary assistants, and it could thrive with contributions from the community.

[1] 9to5Mac: Hands-on: Here's what does and doesn't work with offline Siri in iOS 15: https://9to5mac.com/2021/10/13/how-offline-siri-works-iphone-in-ios-15/

[2] https://github.com/Stypox/dicio-numbers

[3] Dicio project roadmap, https://github.com/Stypox/dicio-android/issues/129

Bibliography

[1] Cecilia Aas, Hisham Abdelsalam, Irina Belousova, Shruti Bhargava, Jianpeng Cheng, Robert Daland, Joris Driesen, Federico Flego, Tristan Guigue, Anders Johannsen, Partha Lal, Jiarui Lu, Joel Ruben Antony Moniz, Nathan Perkins, Dhivya Piraviperumal, Stephen Pulman, Diarmuid Ó Séaghdha, David Q. Sun, John Torr, Marco Del Vecchio, Jay Wacker, Jason D. Williams, and Hong Yu. Intelligent assistant language understanding on device, 2023.

[2] Alphacephei. Vosk speech recognition toolkit. `https://alphacephei.com/vosk/`, 2019. Accessed: 2024-06-19.

[3] AXA Group Operations Spain S.A. nlp.js. `https://github.com/axa-group/nlp.js`, 2018. Accessed: 2024-06-25.

[4] Arturs Backurs and Piotr Indyk. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false). *arXiv e-prints*, page arXiv:1412.0348, November 2014.

[5] Daniel Braun, Adrian Hernandez-Mendez, Florian Matthes, and Manfred Langen. Evaluating natural language understanding services for conversational question answering systems. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 174–185, Saarbrücken, Germany, August 2017. Association for Computational Linguistics.

[6] Alice Coucke, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, et al. Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces. *arXiv preprint arXiv:1805.10190*, pages 12–16, 2018.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, Berlin, Heidelberg, March 2008.

[8] F-Droid Contributors. F-Droid - free and open source Android app repository. `https://f-droid.org/`, 2010. Accessed: 2024-06-25.

[9] Google. DataStore. `https://developer.android.com/topic/libraries/architecture/datastore`, 2012. Accessed: 2024-06-18.

[10] Google. Jetpack compose. `https://developer.android.com/develop/ui/compose`, 2019. Accessed: 2024-06-14.

[11] Gradle Inc. Gradle build tool. `https://gradle.org/`, 2008. Accessed: 2024-06-19.

[12] JetBrains. Kotlin coroutines. `https://kotlinlang.org/docs/coroutines-overview.html`, 2017. Accessed: 2024-06-14.

[13] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[15] Microsoft. ReactiveX RxJava: Reactive Extensions for the JVM. https://github.com/ReactiveX/RxJava, 2011. Accessed: 2024-06-14.

[16] Mycroft. Padatious nlu framework. https://github.com/MycroftAI/padatious, 2017. Accessed: 2024-06-25.

[17] OVOS. Open Voice OS assistant (continuation of Mycroft). https://www.openvoiceos.org/, 2020. Accessed: 2024-06-25.

[18] Rhasspy community. Rhasspy assistant. https://rhasspy.readthedocs.io/en/latest/, 2020. Accessed: 2024-06-25.

[19] Square. KotlinPoet. https://square.github.io/kotlinpoet/, 2012. Accessed: 2024-06-19.

[20] Square and Google. Dagger hilt. https://dagger.dev/hilt/, 2012. Accessed: 2024-06-18.