Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

CAI Centre for Artificial Intelligence

Bachelorarbeit (Informatik)

Game-Entwicklung: Schweizer Dialekte raten

Autoren	Philippe Giavarini Markus Pfefferli
Hauptbetreuung	Prof. Dr. Mark Cieliebak
Nebenbetreuung	Manuela Hürlimann

Datum	07.06.2024

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Erklärung betreffend das selbstständige Verfassen einer Bachelorarbeit an der School of Engineering

Erklärung betreffend das selbstständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)
Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:	**Name Studierende:**
Basel, 07.06.2024	Philippe Giavarini
Zürich, 07.06.2024	Markus Pfefferli

Abstract

This bachelor thesis describes the process of planning and implementing a web application game to guess the origins of Swiss German dialects. Dialects hold significant meaning in the daily lives of Swiss people and are an integral part of their national identity.
The web application is structured into two parts: The frontend is powered by Next.js, a framework for the JavaScript library React, and a backend which works with Python, helped by the FastAPI framework. The audio files of the different dialects are provided by the STT4SG-350 collection. The collection of data, to answer the question of how well different dialects can be recognized, is also part of this thesis. It can be determined that distinctive dialects, such as the Walliser dialect, are easier to recognize regardless of the origin of the person making the guess. Gender does not play a role in the ability to recognize dialects. What did matter, however, was the origin of the guesser: It was easier for them to guess their own dialects or dialects close by.

Zusammenfassung

Diese Bachelorarbeit beschreibt den Prozess der Planung und Implementierung einer Webanwendung um die Herkunft Schweizerdeutscher Dialekte auf spielerische Weise zu erraten. Dialekte haben eine grosse Bedeutung im täglichen Leben der Schweizer Bevölkerung und sind ein wesentlicher Bestandteil ihrer nationalen Identität.
Die hier beschriebene Webanwendung ist in zwei Teile gegliedert: Das Frontend wird von Next.js betrieben, einem Framework für die JavaScript-Bibliothek React, während das Backend mit Python, unterstützt vom FastAPI-Framework, geschrieben ist. Die Audiodateien der verschiedenen Dialekte stammen aus dem Korpus STT4SG-350. Um die Frage zu beantworten, wie gut verschiedene Dialekte erkannt werden können, ist die Datensammlung ebenfalls Teil dieser Arbeit. Es kann festgestellt werden, dass markante Dialekte, wie der Walliser Dialekt, unabhängig von der Herkunft der ratenden Person leichter zu erkennen sind. Das Geschlecht spielt keine Rolle bei der Fähigkeit, Dialekte zu erkennen. Was jedoch eine Rolle spielt, ist die Herkunft der ratenden Person: Es war für sie einfacher, ihren eigenen Dialekte oder räumlich nahegelegene Dialekte zu erraten.

Danksagung

Wir möchten uns herzlich bei Prof. Dr. Mark Cieliebak und Manuela Hürlimann bedanken für Möglichkeit unsere Bachelorarbeit in der Natural Language Processing Group der ZHAW zu schreiben. Vielen Dank für die gute Unterstützung während der Arbeit und dem Schaffen einer angenehmen und produktiven Atmosphäre.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Bedeutung der schweizerdeutschen Dialekte

Dialekte haben in der Schweiz eine sehr grosse Bedeutung. Standarddeutsch, oder schweizerisch, Schriftdeutsch oder Hochdeutsch, findet primär in gedruckter Form oder im formellen Rahmen Anwendung. Die kontextabhängige Nutzung von Hochsprache oder Dialekt wird als Diglossie [1]. Ändert sich die Sprachform in Abhängigkeit der Kommunikationsform, also mündlich und schriftlich, wird es als mediale Diglossie [2] bezeichnet. Heutzutage wird jedoch auch die schriftliche digitale Kommunikation (z.B. WhatsApp) häufig, insbesondere von den jüngeren Generationen [3], in Schweizerdeutsch verfasst. Der Einfluss des Dialekts nimmt also generell nicht ab.

Am gesprochenen Dialekt kann man die Herkunft eines Deutschschweizers erkennen. Einige Dialekte sind dabei prägnanter als andere, aber die Einordnung der ungefähren Herkunft ist bei den meisten Personen möglich. Wie genau dies möglich ist, soll im Rahmen dieser Bachelorarbeit untersucht werden. Zuvor muss jedoch eine geeignete Webapplikation konzipiert und programmiert werden, damit ein Sammeln von Daten einfach und spielerisch möglich wird.

Es sei hier erwähnt, dass im Rahmen dieser Arbeit OpenAIs ChatGPT 4.0 bzw. 4o als Programmierhilfe (Code generieren, debuggen) zum Einsatz kam. Ebenfalls wurde es für die Verbesserung oder Generierung von einzelnen Sätzen oder Textabschnitten verwendet.

1.2 Apps und Webseiten für die Dialektbestimmung

Durch diese Unterscheidbarkeit der Dialekte wurde und wird immer wieder versucht schweizerdeutsche Dialekte anhand der Aussprache bestimmter Wörter den entsprechenden Herkunftsorten zuzuordnen. Eine der ersten solchen Internetseiten dürfte «Das Chochichästli-Orakel»[4] sein, das immer noch abrufbar ist. Hier kann bei zehn ausgewählten schriftdeutschen Worten die eigene schweizerdeutsche Aussprache ausgewählt werden. Anhand dieser Auswahl wird dann mittels einer Heatmap die wahrscheinlichste Herkunft angezeigt.

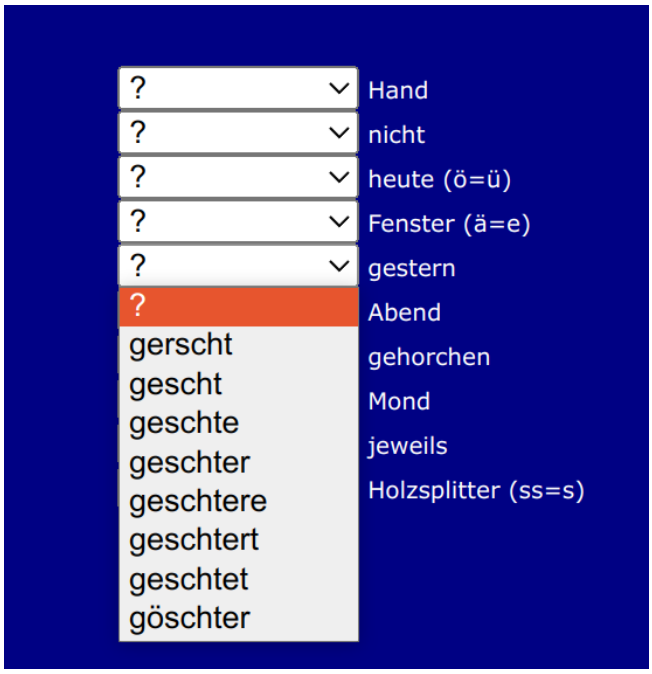

Abbildung 1.1: Ausschnitt aus der Webseite des Chochichästli-Orakels [4]. Über ein Dropdown-Menü wird der zum eigenen Dialekt passende Schweizerdeutsche Ausdruck ausgewählt.

Nach dem gleichen Prinzip funktioniert der Dialekt-Test des Tages-Anzeigers [5]. Er fragt die doppelte Anzahl Wörter ab und ist zeitgemässer gestaltet.
Gerade ganz aktuell wurde eine überarbeitete Version des Dialekt-Tests [6] veröffentlicht. Bei dieser neuen Version ist es zusätzlich möglich Dialekte aus dem ganzen deutschsprachigen Raum zu bestimmen und die Wörter der Schweizer Dialekte lassen sich auch anhören.

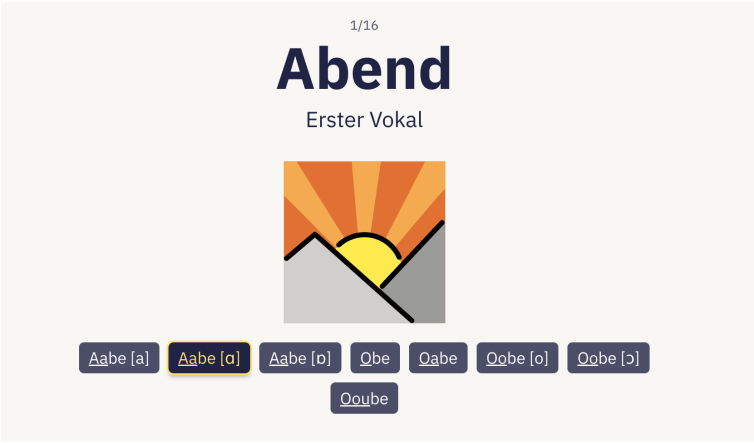

Abbildung 1.2: Ausschnitt eines Screenshots aus dem neuen Dialekt-Test des Tages-Anzeigers [6]

Als iOS-App gibt es von Adrian Leeman, der auch den Dialekt-Test des Tages-Anzeigers mitentwickelt hat, die «Dialäkt Äpp»[7]. Sie ermöglicht auch den eigenen Dialekt zu verorten, den eigenen Dialekt aufzunehmen und die Dialekte anderer Ortschaften anzuhören.

Die Webseite «sprachatlas.ch - Der Sprachatlas der deutschen Schweiz» [8], basiert auf dem gleichnamigen Atlas (häufig in Kurzform *SDS*): «Der achtbändige Atlas dokumentiert auf über 1500 Sprachkarten die alemannischen Mundarten der deutschen Schweiz, einschliesslich der Walserdialekte Norditaliens. Grundlage für die Karten bilden mündliche Befragungen aus den Jahren 1939 bis 1958. Diese Website macht die Kartenbände sowie das Originalmaterial (Fragebuch und Antwortmaterial) digital zugänglich.»

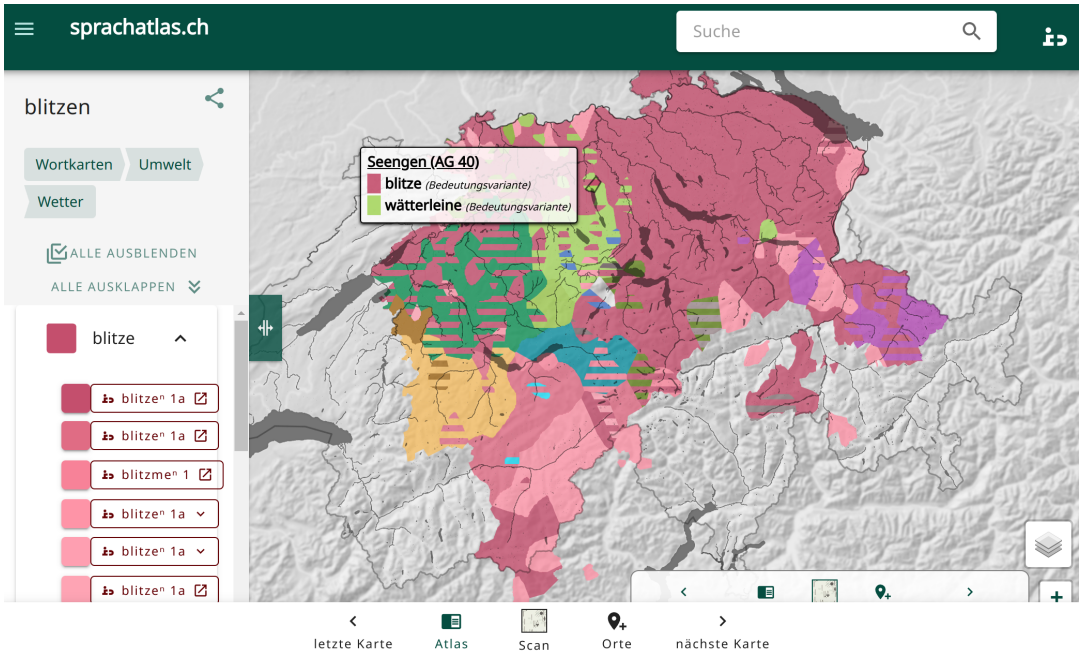

Abbildung 1.3: Ausschnitt aus der Webseite sprachatlas.ch [8]
. Auf Karten wird die geographische Verbreitung der verschiedenen Schweizerdeutschen Dialektausdrücke dargestellt.

1.3 Entwicklung einer eigenen Applikation

Die im Rahmen dieser Bachelorarbeit entwickelte Webapplikation hat einen ähnlichen Vorgänger, der jedoch nicht mehr online zur Verfügung steht. Da die Codebasis veraltet ist, wurde entschieden, die Applikation von Grund auf neu zu entwickeln. Dafür mussten verschiedene Grundsatzentscheide getroffen werden und Minimalanforderungen definiert werden. Dies geschah noch vor der Entscheidung für die verwendeten Technologien.

1.4 Anforderungen an eine eigene Webapplikation

Die Hauptanforderung an die neu zu entwickelnde Webapplikation ist die einfache Bedienbarkeit. Die Benutzeroberfläche soll übersichtlich gestaltet sein und es muss auch bei erstmaliger Benutzung klar sein, wie die Bedienung funktioniert. Mit dem später an anderer Stelle erläuterten Technologieentscheid, stellt sich auch die Frage, ob eine Mobile-Applikation oder eine Desktop-Applikation im Vordergrund stehen soll. Das heisst, soll eine Desktop-Webapplikation entwickelt werden, die auch auf kleineren Touchscreen-Bildschirmen bedienbar bleibt oder wird der umgekehrte Weg

eingeschlagen, also eine Mobile-Applikation, die zur Not auch auf einem Desktop-Bildschirm funktioniert. Für dieses Projekt wurde die erste Variante gewählt, auch vor dem Hintergrund, dass eine Karte der Schweiz auf einem Bildschirm im standardmässigen Querformat besser zur Geltung kommt und sich einfacher bedienen lässt.

Da die Karte zur Auswahl des Dialektherkunftsortes im Mittelpunkt steht, ist die Auswahl der verwendeten Technologie bzw. des Plugins besonders wichtig. Es stand schon relativ bald fest, dass hier Leaflet zum Einsatz kommen sollte. Dies wäre jedoch für das Flutter-Entwicklungskit nicht zur Verfügung gestanden.

Nachfolgend die verschiedenen definierten funktionellen Anforderungen und Screenshots dieser in der fertigen Applikation:

1.5 Seite «Registration»

1.5.1 Registration

Ein Nutzer der Applikation muss sich registrieren können. Nur so können wir wiederkehrende Besucher der Seite zweifelsfrei identifizieren und ihre Eingaben auswerten. Ein Spielen soll erst nach erfolgter Registration möglich sein.

1.5.2 Zusätzliche Angaben bei der Registration

Bei der Registration soll es möglich sein, zusätzliche Angaben zur Person abzufragen, die bei einer späteren Auswertung nützlich sein könnten. Einige davon sollen obligatorisch sein (z.B. E-Mail-Adresse) andere hingegen fakultativ (z.B. Altersgruppe).

1.5.3 Annahme der Nutzungs- und Datenschutzbedingungen

Bei der Registration muss es möglich sein, die Nutzungs- und Datenschutzbedingungen zu lesen, denen für eine erfolgreiche Registration zwingend zugestimmt werden muss.

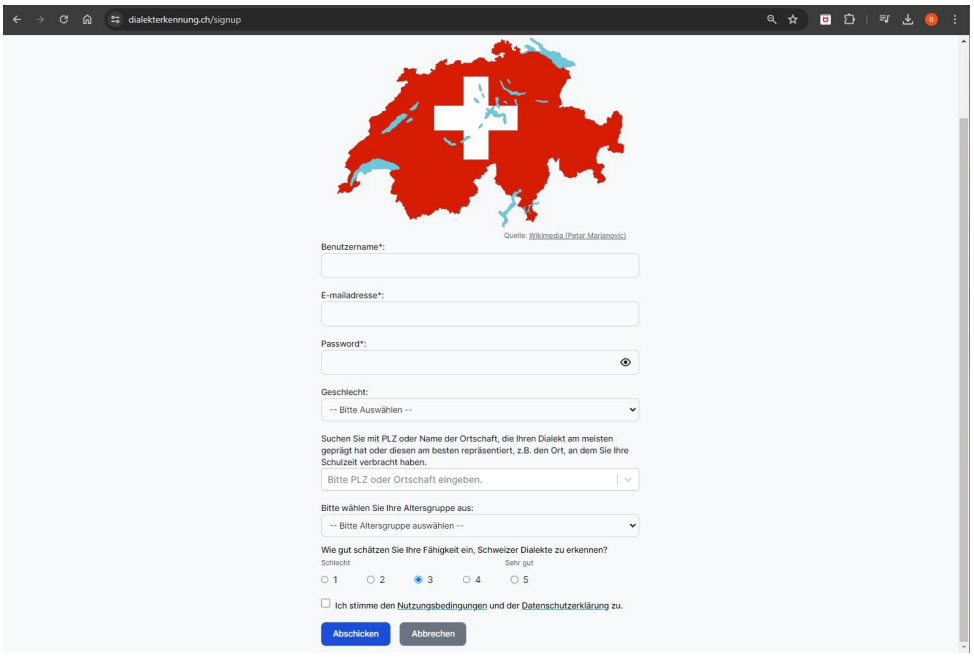

Abbildung 1.4: Screnshot: Seite «Registration»

1.6 Seite «Dialekte raten»

1.6.1 Zoombare Schweizerkarte

Der Herkunftsort des gehörten Dialekts soll auf einer verschiebbaren und zoombaren Schweizerkarte markiert werden können. Es soll nur die Schweiz selber sichtbar sein. Die Karte soll eine ansprechende Darstellung aufweisen und die wichtigsten Städte müssen auch in der Gesamtansicht sichtbar sein. Die Kantonsgrenzen müssen ebenfalls sichtbar sein.

1.6.2 Anzeige der momentan gewählten Ortschaft

Wird der Marker für die Dialektherkunft auf der Karte verschoben, so soll auch unterhalb der Karte immer angezeigt werden, in welcher Ortschaft er sich gerade befindet.

1.6.3 Suche per Textfeld

Die Anzeige des momentanen Markerstandortes ist ein Textfeld, in dem auch alle anderen Schweizer Ortschaften gesucht und ausgewählt werden können. Wird dies gemacht, so verschiebt sich der Marker automatisch an den richtigen Standort.

1.6.4 Audioplayer

Der Audioplayer soll sich harmonisch in das Gesamtbild der Seite einfügen und einfach zu bedienen sein. Er soll über einen Play-/Pause-Button verfügen. Die Gesamtspieldauer der Aufnahme sowie der verstrichenen Zeit sollte angezeigt werden. Ausserdem verfügt der Audioplayer über einen Lautstärkeregler.

1.6.5 Text des gesprochenen Satzes

Ober- oder unterhalb des Audioplayers ist der momentan gesprochene Satz als Text ablesbar.

1.6.6 Weitere Aufnahmen des gleichen Sprechers hören

Beim Audioplayer existiert ein Knopf, mit dem es möglich ist, bis zu vier weitere Aufnahmen des gleichen Sprechers anzuhören.

1.6.7 Vorschlag abgeben

Ist der Spieler mit der momentanen Position des Markers zufrieden, so kann er seine Wahl mithilfe eines auffälligen Buttons bestätigen.

1.6.8 Effektiver Dialektherkunftsort anzeigen

Als Auflösung soll dem Spieler der effektive Dialektherkunftsort ebenfalls auf der Karte angezeigt werden.

1.6.9 Distanz der Abweichung

Dem Spieler wird der Name des gesuchten Ortes, die Abweichung in Kilometer, sowie die gewählte Ortschaft angezeigt.

1.6.10 Weiterer Dialekt raten

Möchte der Spieler nach dem Anzeigen der Auflösung weiter spielen, so gelangt er über einen Button zur nächsten Spielrunde.

1.6.11 Menu

Es soll ein Menu existieren, um einfach zwischen allfälligen anderen Seiten, wie z.B. dem Projektbeschrieb, gewechselt werden kann.

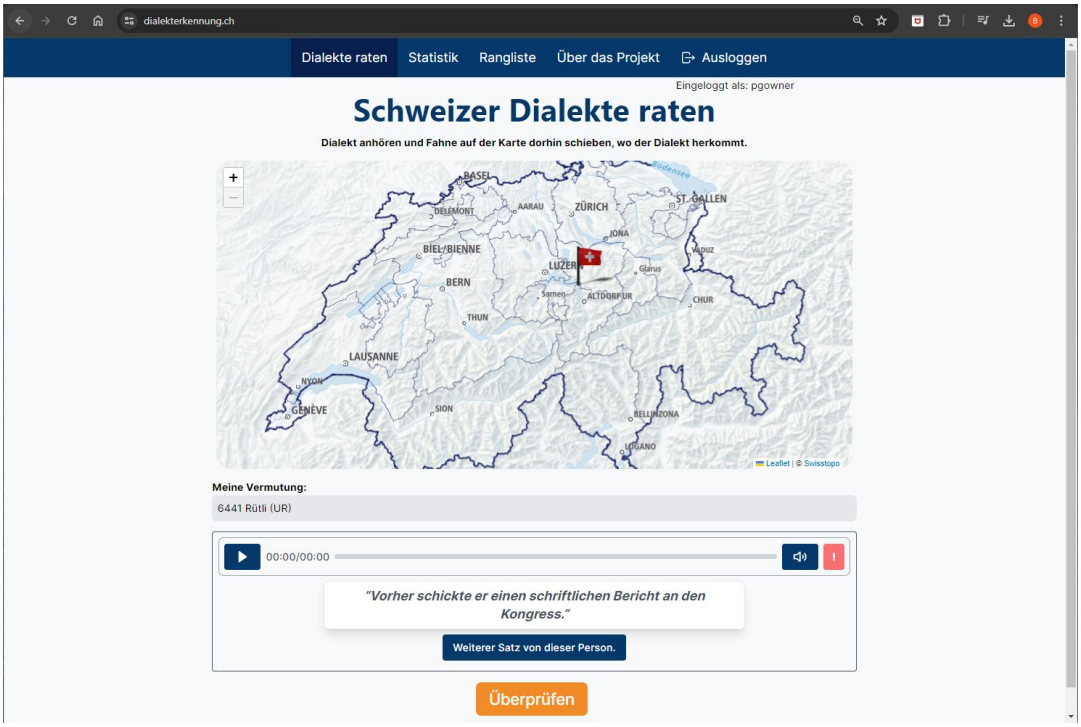

Abbildung 1.5: Screenshot: Seite «Dialekte raten»

1.7 Seite «Statistik»

1.7.1 Bereits geratene Dialekte anzeigen

Alle geratenen Dialekte sollen mit dem erzielten Resultat aufgelistet werden können.

1.7.2 Dialekte nochmals anhören

Die gespielten Dialekte lassen sich auch nochmals anhören.

1.7.3 Anzeige von verschiedenen zusätzlichen Benutzerdaten

Es soll der eigene Dialekt, der bei der Registrierung ausgewählt wurde, angezeigt werden. Zusätzlich soll der eigene Rang, die Anzahl der geratenen Dialekte, die durchschnittliche Abweichung über alle Rateversuche und der beste Versuch angezeigt werden.

Abbildung 1.6: Screenshot: Seite «Statistik»

1.8 Seite «Rangliste»

1.8.1 Anforderungen Rangliste

In der Rangliste wird der Nutzer anhand seiner gespielten Leistung eingeordnet. Die Bewertungsgrundlage ist dabei die durchschnittliche Abweichung von der geratenen zur effektiven Ortschaft. Da dem Nutzer zufällige Dialekte abgespielt werden, ergibt eine Mindestanzahl gespielte Versuche Sinn, nach denen er in der Rangliste erscheint, da sonst Glück eine grosse Rolle spielt.

1.8.2 Gestaltung

Die Rangliste soll einklappbar sein, so dass die ersten Ränge immer angezeigt werden, aber nicht alle Zwischenränge. Die Ränge direkt vor und nach dem Rang des Nutzers sind dann wieder sichtbar.

1.8.3 Zusätzliche Informationen

Der eigene Rang soll ersichtlich sein, sowie die durchschnittliche erspielte Abweichung.

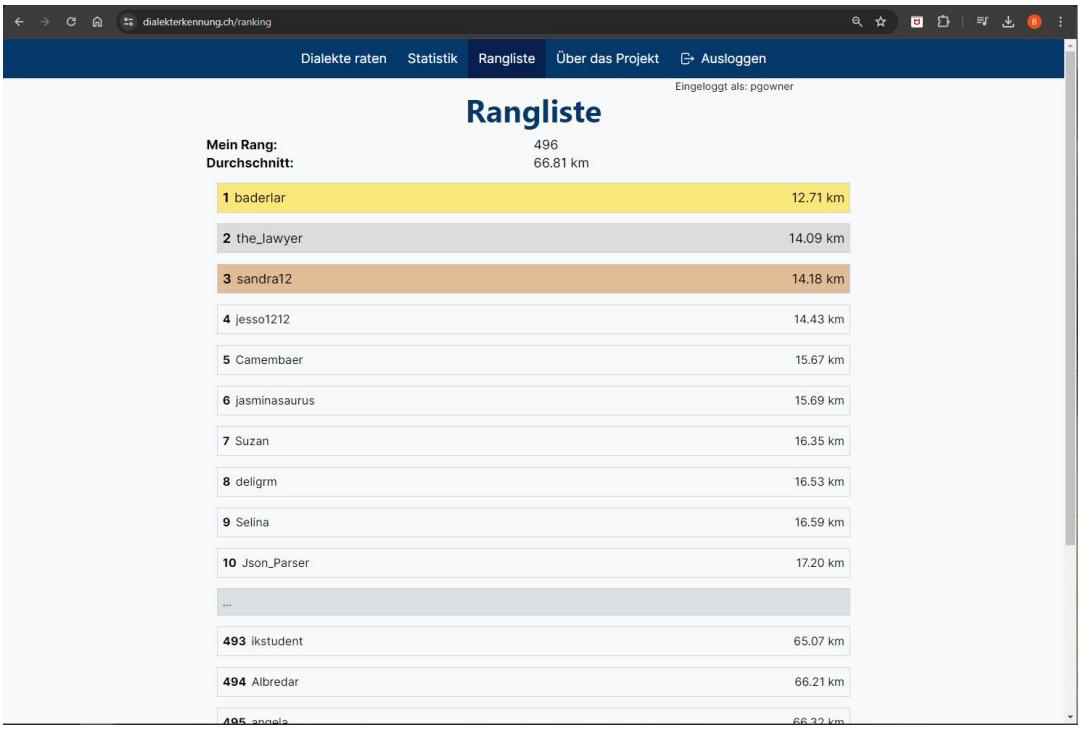

Abbildung 1.7: Screenshot: Seite «Rangliste»

1.9 Seite «Über das Projekt»

Auf dieser Seite soll der Projektrahmen erklärt werden. Hier soll beschrieben werden, wie das Projekt entstanden ist und auch woher die Sprachdateien stammen.

Zusätzlich soll auch eine Kontakt-E-Mail-Adresse vorhanden sein.

Abbildung 1.8: Screenshot: Seite «Über das Projekt»

1.10 Welche Distanz soll verwendet werden?

Für das Spiel wird die Luftdistanz in Kilometern zwischen geratenem Ort und effektivem Dialektherkunftsort verwendet. Erste Überlegungen gingen jedoch dahin, dass es kaum möglich sein würde, den Herkunftsort eines Dialektes exakt zu bestimmen und daher ein linearer Verlauf der Bewertungsskala keinen Sinn ergeben würde. Also im Sinne von: Wer näher als eine bestimmte Distanz raten würde, hatte wohl einfach Glück. Dass Luftdistanz verwendet werden würde, anstelle z.B. von Fahrdistanz oder -dauer, stand relativ schnell fest, da Berge zwar ein Hindernis darstellen können, aber mit der heutigen Mobilität Uri z.B. von Glarus auch nicht mehr so stark getrennt ist, wie es vielleicht in früheren Jahrhunderten der Fall gewesen sein mag. Sprich, die meisten Schweizer sind schon mit den verschiedensten Dialekten in Kontakt gekommen, z.B. schon nur über Medien, auch wenn sie abgelegen wohnen würden. Das Problem bei der Verwendung einer Abstufung der Bewertung (z.B. volle Punktzahl für Abweichung < 10 km) liegt darin, dass die Abstufungen je nachdem, woher der geratene Dialekt stammt, eine ganz andere Bedeutung haben. Im Umfeld von kleinen Kantonen sind diese viel einfacher zu erreichen, als zum Beispiel im Kanton Bern.

Kapitel 2

Schweizer Dialekte

2.1 Allgemeine Theorie

Schweizerdeutsch ist der Oberbegriff für die verschiedenen alemannischen Dialekte, die in der Deutschschweiz gesprochen werden. Ein Dialekt ist dabei eine lokale oder regionale Sprachvarietät, also keine eigenständige Sprache. Um zwischen Dialekt und eigener Sprache zu unterscheiden, ist der sprachliche Abstand, also die Unterschiede zu einer Schriftsprache, nicht entscheidend. Es braucht vielmehr auch einen politischen Willen einen Dialekt zu einer Standardsprache weiterzuentwickeln [9]. Als Beispiel für eine solche Entwicklung wird häufig Luxemburgisch genannt. Für die Schweiz ist dies jedoch kein realistisches Szenario. Es müsste eine neue Schriftsprache entwickelt werden und sich dabei gleichzeitig auf eine «richtige» Variante geeinigt werden, was angesichts von Lokalpatriotismus und «Kantönligeist» schon von Anfang an zum Scheitern verurteilt ist.

Schweizerdeutsch steht im Gegensatz zum Schweizer Hochdeutsch, einer schweizerischen Variante des Standarddeutsch. Schweizer Hochdeutsch wird primär für die schriftliche Kommunikationsform verwendet wird oder falls ein Gesprächsteilnehmer kein Dialekt versteht.
Diese Unterscheidung zwischen gewählter Sprachform in Abhängigkeit von der Kommunikationsform wird, wie schon in der Einleitung erwähnt, als mediale Diglossie bezeichnet. Schweizer Dialekte unterscheiden sich unterschiedlich stark voneinander, aber sie sind grundsätzlich für jeden Schweizerdeutschsprechenden verständlich. Dialekt wird in der Deutschschweiz von der gesamten Bevölkerung gesprochen und nicht als Soziolekt [10] nur von bestimmten Bevölkerungsgruppen oder in bestimmten Situationen. Die immer noch grosse Bedeutung des Schweizerdeutschen wird unter anderem auf seine Rolle zu Zeiten der Gründung des modernen Bundesstaates 1848 zurückgeführt. Das Schweizerdeutsche diente als patriotisches Kennzeichen und gleichzeitig als Abgrenzung gegenüber Deutschland. In den 1930er und 1940er Jahren, im Rahmen der «geistigen Landesverteidigung», nahm das Schweizerdeutsche als Zeichen nationaler Identifikation ebenfalls eine sehr wichtige Stellung ein [11].

2.2 Wie viele schweizerdeutsche Dialekte gibt es?

Um zu beantworten, wie genau eine Dialektherkunftsbestimmung überhaupt möglich ist, wird man sich die Frage stellen, wie viele unterschiedliche schweizerdeutsche Dialekte es überhaupt gibt.

Schweizerdeutsche Dialekte werden, mit Ausnahme des Samnauner Dialekts, in Nieder-, Mittel-, Hoch- und Höchstalemannisch eingeteilt. In Samnaun spricht man den nachbarlichen Tiroler Dialekt, also Bairisch [12]. Bairisch zählt, wie auch die alemannischen Dialekte, zu den oberdeutschen Dialekten.

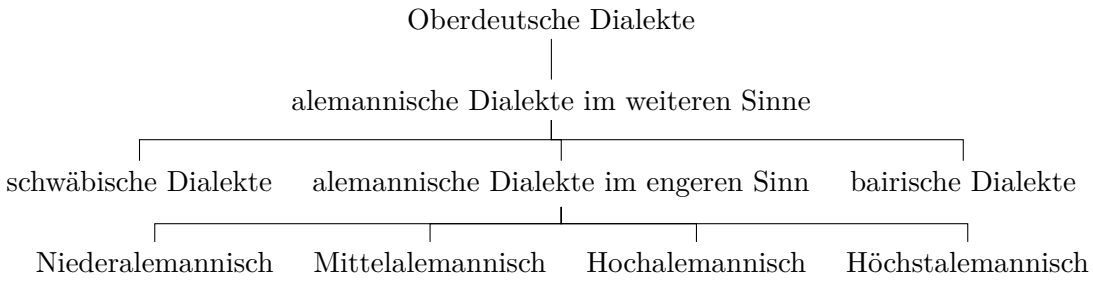

Abbildung 2.1: Gliederung der oberdeutschen Dialekte [13]

Die Zuteilung zu den vier verschiedenen alemannischen Dialektgruppen in der Schweiz erfolgt laut dem Schweizerischen Idiotikon [14] wie folgt: Die Mundart der Stadt Basel wird zum Niederalemannischen gezählt, mittelalemannische Dialekte finden sich in der äussersten Nordostschweiz, die Mundarten des Juras und Mittellandes zählen zum Hochalemannischen und die Mundarten des Alpenraums zum Höchstalemannischen. Die Unterteilung von Nieder- zu Höchstalemannisch ist gleichbedeutend mit der geografischen Lage, also von Norden her immer weiter in den Süden.

Eine genaue Antwort zur Frage der Anzahl schweizerdeutscher Dialekte gibt es nicht, da es auch eine Definitionssache ist, wie gross der sprachliche Unterschied sein muss, um überhaupt als separater Dialekt zu gelten. Klar ist, dass eine Aufteilung nur in die vier alemannischen Gruppen zu grob ist. Auch eine kantonale Aufteilung wird zwar häufig angewendet, ist aber vor allem in den Grenzgebieten der einzelnen Kantone nicht sinnvoll, da die Übergänge fliessend sind.

2.3 Das Korpus STT4SG-350

Ein Korpus ist laut Duden [15] «eine Sammlung einer begrenzten Anzahl von Texten, Äusserungen o. Ä. als Grundlage für sprachwissenschaftliche Untersuchungen». STT4SG-350 steht dabei für Speech-to-Text for Swiss German - 350 hours, also 350 Stunden Audioaufnahmen von gesprochenem Schweizerdeutsch, die das Training von Modellen ermöglichen, die helfen gesprochene Sprache in Text umzuwandeln. Das Korpus beinhaltet neben den Audioaufnahmen selbst auch tsv-Dateien (tsv: tab-separated values), die zusätzliche Informationen bereitstellen. Jede Audioaufnahme ist ein gesprochener Satz. Die tsv-Datei identifiziert u.a. den jeweiligen Sprechenden, gibt an, woher der Dialekt des Sprechenden stammt, wie der gesprochene Satz ausgeschrieben lautet und liefert noch weitere Informationen.

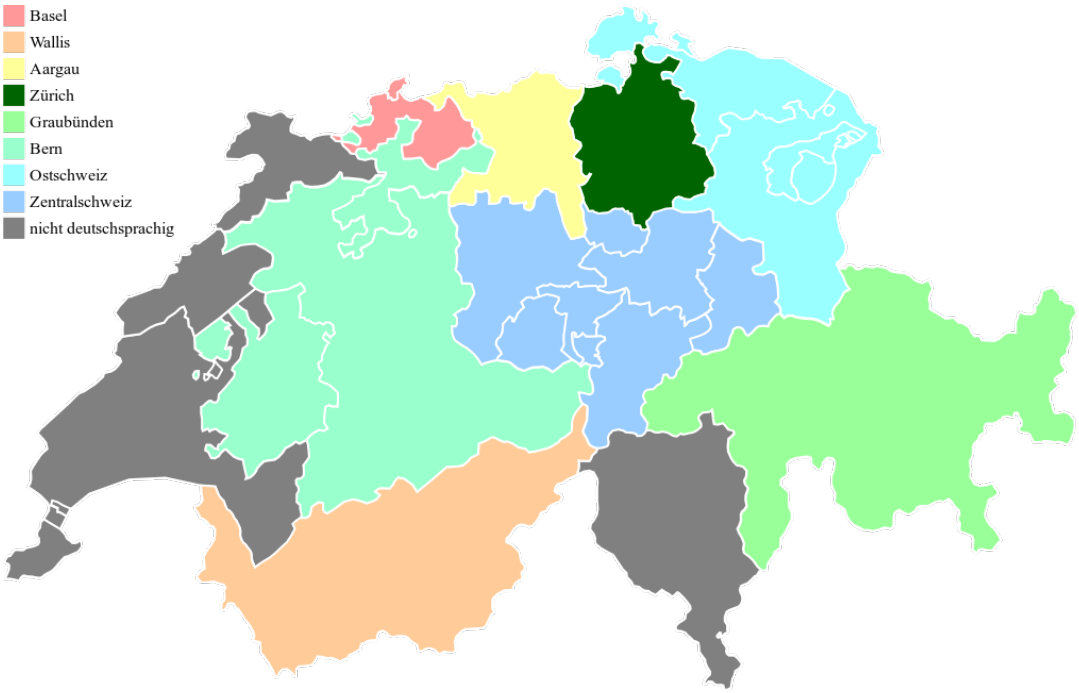

Abbildung 2.2: Dialektregionen der Schweiz, wie sie im Korpus STT4SG-350 [16] verwendet werden. Die Region Aargau wird dabei aufgeteilt.[1]

Abbildung 2.2 zeigt die im Paper [16] des Korpus verwendeten Dialektregionen. Diese werden in der Datenauswertung später ebenfalls verwendet. Aargau ist dabei keine eigene Dialektregion, sondern wird auf die Regionen Basel, Bern, Zentralschweiz und Zürich aufgeteilt.

Der Datensatz des Korpus ist in verschiedene Teile gegliedert, da es für Machine-Learning-Modelle zusammengestellt wurde. Für dieses Projekt wird der train-Teil verwendet (genaue Bezeichnung: *train_all.tsv*). Er beinhaltet fast 200000 Aufnahmen. Es sind Sprecher:innen aus allen Deutschschweizer Kantonen vertreten, mit Ausnahme von Appenzell Ausserrhoden, Solothurn und Nidwalden. Interessanterweise enthält der Test-Datensatz, der aber hier nicht verwendet wird, auch einige wenige Beispiele aus Solothurn. Die Verteilung auf die Kantone (siehe Abb. 2.3) sowie auf die Dialektregionen (siehe Abb. 2.4) bezieht sich auf die Anzahl der verwendeten Samples während dieses Projektes. Ein Sample (hier auch Dialektbeispiel genannt) entspricht dabei der Aufnahme eines gesprochenen Satzes.

[1] Quelle SVG-Datei: `https://upload.wikimedia.org/wikipedia/commons/c/c3/Schweiz_Tabak_Abgabealter_nach_Kanton.svg`

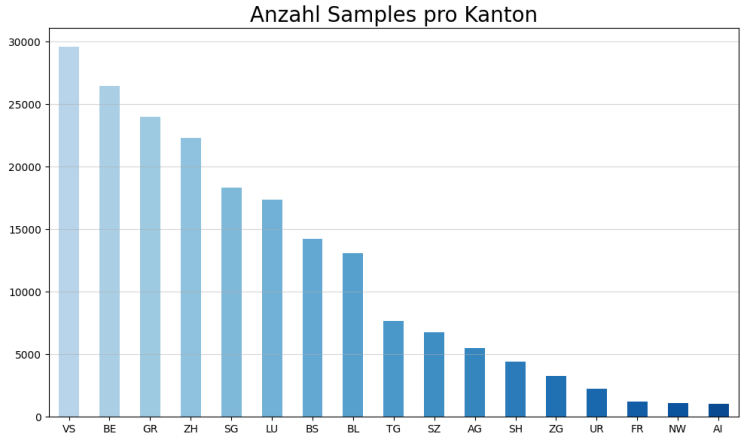

Abbildung 2.3: Anzahl Samples pro Kanton

Hier ist ersichtlich, dass die Aufteilung auf die Kantone sehr unausgeglichen ist. Wie in Abbildung 2.4 ersichtlich, beinhalten die Dialektregionen Wallis und Graubünden nur diese Kantone, während bei den restlichen Dialektregionen die Dialekte mehrerer Kantone zusammengefasst werden. Dies führt dazu, dass die Verteilung der Dialektbeispiele auf Ebene der Dialektregionen ausgeglichen ist.

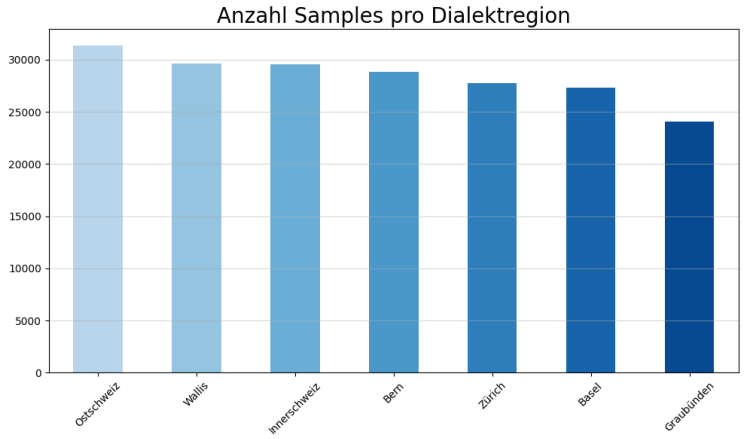

Abbildung 2.4: Anzahl Samples pro Dialektregion

Auf Ortschaftsebene enthält das Trainingsset 162 verschiedene Postleitzahlen. Die Dialektbeispiele stammen von 218 verschiedenen Sprecher:innen. Die genaue Verteilung der Sprecher:innen ist in Abbildung 2.5 ersichtlich. Die Zahl dürfte ausreichend gross sein, so dass die Spielenden nicht in absehbarer Zeit anhand der Stimme lernen, den richtigen Dialektherkunftsort zu erkennen, wie es als Befürchtung in einem Feedback während der Datensammlungsphase geäussert wurde.

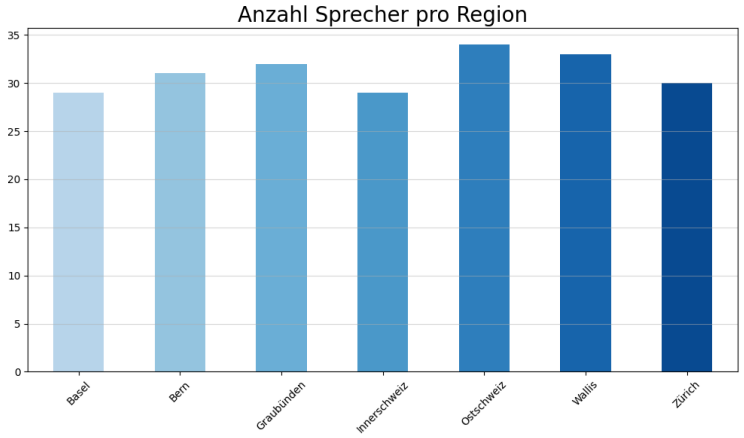

Abbildung 2.5: Anzahl Sprecher pro Dialektregion

2.4 Der Sonderfall Aargau

Wie im vorherigen Kapitel schon erwähnt, ist der Kanton Aargau ein Sonderfall, indem er als Kanton nicht nur einem Dialektgebiet zugeordnet wird, sondern gleich vier verschiedenen. Dass dies der Fall ist, hat historische Gründe und geht darauf zurück, dass das Kantonsgebiet in vier Gebiete aufgeteilt war. Die Webseite des Projekts Hunziker2020 [17] des Schweizerischen Idiotikons beschreibt diese Aufteilung wie folgt: «.. der Südwesten gehörte Bern, der Südosten den Innerschweizer Orten und Zürich, der Nordosten gehörte allen acht alten Orten und der Nordwesten gar als Teil von Vorderösterreich der habsburgischen Krone (bis 1802)». Die acht alten Orte sind dabei: Uri, Schwyz, Unterwalden, Luzern, Zürich, Zug, Bern und Glarus [18].

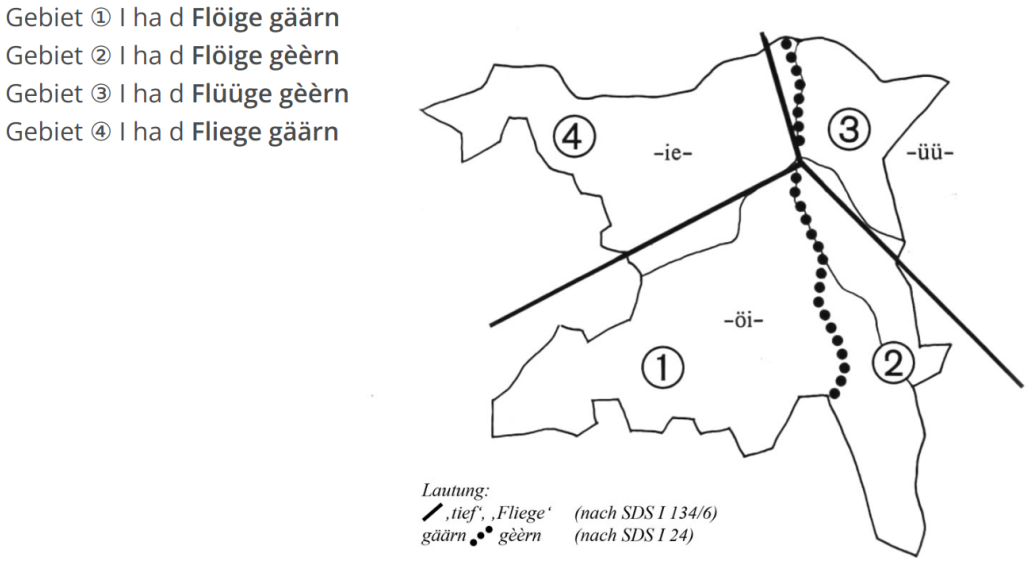

Abbildung 2.6: Aufteilung der Dialektgebiete im Kanton Aargau anhand der Aussprache des Satzes «Ich habe Fliegen gern.» im jeweiligen Dialekt. Abbildung ebenfalls von Hunziker2020 [17]

Das Problem dabei ist, dass sich diese Dialektgrenzen laut der gleichen Quelle in der Zwischenzeit verschoben haben. Diese Aufteilung wird aber immer noch an verschiedenen Stellen so verwendet.
Im Rahmen dieses Projektes mussten die geratenen Ortschaften den Dialektgebieten zugewiesen werden. Ausser beim Kanton Aargau stellt das natürlich kein Problem dar, da jede Ortschaft einem Kanton zugewiesen werden kann und der Kanton wiederum dem entsprechenden Dialektgebiet. Beim Kanton Aargau ist dies nun natürlich aus den bekannten Gründen schwierig und es wurde, die vermutlich falsche Annahme getroffen, dass Abb. 2.6 den momentanen Stand der Dialektgebiete im Kanton Aargau abbildet. Leider wurde dies zu spät bemerkt, so dass die Grafiken nicht erneut erstellt werden konnten. Eine wirklich gute Quelle für die aktuelle Aufteilung konnte auch noch nicht gefunden werden. Die interaktive Grafik des «Kleinen Sprachatlas» [19] zum Wort Bonbon deutet an, dass die Berner und Zürcher Dialektgebiete an Bedeutung gewonnen haben.

Die Aargauer Dialektgebiete wurden wie folgt zugeteilt:
zu Basel: Bezirke Rheinfelden und Laufenburg
zu Bern: Bezirke Zofingen, Kulm, Aarau, Lenzburg und Brugg
zu Zürich: Bezirke Zurzach und Baden
zur Zentralschweiz: Bezirke Muri und Bremgarten

Kapitel 3

Datensammlung und -auswertung

3.1 Die erste Phase der Datensammlung

Nach der Fertigstellung des Spiels bestand der nächste Schritt darin, möglichst viele Spielende zu finden, die wiederum möglichst viele Dialektbeispiele versuchen dem Herkunftsort zuzuordnen. In einer ersten Phase wurde das Spiel unter Verwandten und Bekannten verteilt, was einigen Aufwand erforderte und eher geringen Ertrag erbrachte. Dieser langsamer Start war zwar gut um allfällige Bugs zu finden und zu beheben, aber zeigte auf, dass es nicht ganz einfach sein würde genügend Spielende zu finden.

Im nächsten Schritt sollten per E-Mailverteiler für Forschungsumfragen die Mitstudierenden der ZHAW angeschrieben werden. Dieser Verteiler umfasst circa 1000 Studierende. Um die Anzahl Nutzer unseres Spiels zu erhöhen, musste ein Anreiz geschaffen werden, mitzuspielen. Es wurde entschieden, sechs Gutscheine für einen beliebten Schweizer Versandhändler, als extrinsischen Motivationsfaktor einzusetzen. Dabei war es wichtig, dass nicht nur einfach mitgespielt wird, sondern auch versucht wird ein möglichst gutes Resultat zu erzielen. Um dies zu erreichen, wurden drei Gutscheine für die ersten drei Ranglistenplätze vergeben. Die anderen drei Gutscheine wurden unter dem ersten Drittel der Ranglistenplätze verlost. Zusätzlich mussten mindestens 50 Dialekte geraten werden, um für die Vergabe der Gutscheine infrage zu kommen.

Am kommunizierten Stichdatum wurden die drei Gutscheine für die Erstplatzierten vergeben und gleichzeitig die Frist für die Vergabe der restlichen Gutscheine um zwei Tage, inklusive Auffahrtsfeiertag, verlängert, da festgestellt wurde, dass viele Mitspielende nicht die erforderlichen 50 Versuche aufwiesen. Die Verlängerung führte zu einigen zusätzlichen Gewinnkandidaten und damit einhergehend zusätzlichen Rateversuchen.

3.2 Optimierung der Rangliste

Damit nicht nur Glück eine Rolle spielt, erfolgt der Eintrag in der Rangliste erst nach einer Mindestanzahl geratener Dialekte. Diese Zahl wurde willkürlich auf zehn Dialekte festgelegt. Nachdem die ersten Spieler Einträge in der Rangliste hatten, wurde festgestellt, dass die meisten Spieler bei gutem Resultat nach diesen zehn Runden aufhören weiterzuspielen. Um möglichst viele geratene Dialekte zu erhalten, wäre es

allerdings vorteilhaft, wenn pro Person möglichst viele Runden gespielt würden. Beim ZHAW-internen Wettbewerb wurde dies forciert, indem mindestens 50 Versuche verlangt wurden, um an der Preisverleihung teilnehmen zu können. Vor dem geplanten Versand der E-Mail an die Teilnehmer der Dialektsammlung, wurde nun überlegt, ob eine Anpassung der Mindestanzahl sinnvoll sein könnte. Da nun schon über 400 Einträge in der Rangliste bestanden, lag es nahe, zu schauen, wie viele Versuche notwendig sind, bis sich die durchschnittliche Distanz pro geratenem Dialekt einpendelt. Dafür wurde der laufende Mittelwert berechnet und mit dem Mittelwert des vorherigen Versuchs verglichen. Liegt die Differenz der beiden Mittelwerte unter einem vorgängig definierten Betrag, so erhöht sich der Zähler um eins, anderenfalls wird er wieder auf null gesetzt. Dieser Zähler hat wiederum einen Maximalwert. Sobald dieser erreicht wird, wird die Berechnung abgebrochen und die notwendige Anzahl Versuche gespeichert. Dies wird nun mit allen Spielern gemacht, um im Anschluss den Mittelwert aller Versuche zu berechnen.

$\boldsymbol{n}$: n-ter geratener Dialekt
$\boldsymbol{m}$: maximaler Distanzunterschied zwischen aktuellem und vorherigem Mittelwert (z.B. 5 Kilometer), damit Bedingung als erfüllt gilt.
$\boldsymbol{z}$: zählt, in wie vielen nacheinander folgenden Versuchen die Bedingung m erfüllt wurde.
$\boldsymbol{z_{max}}$: ist $z = z_{max}$ wird abgebrochen und n gespeichert.

$$\bar{x}_{n-1} = \frac{1}{n-1}\sum_{i=1}^{n-1} x_i \quad \bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$z = \begin{cases} z+1 & \text{falls } \bar{x}_n - \bar{x}_{n-1} < m \\ 0 & \text{anderenfalls} \end{cases}$$

$$n = \begin{cases} n+1 & \text{falls } z < z_{max} \\ \textit{Abbruch} & \text{anderenfalls} \end{cases}$$

m	z_{max}	n
5	3	7,46
3	3	9,54
3	5	12,0
5	5	9,75

Tabelle 3.1: Anzahl benötigter Rateversuche n, damit $\bar{x}_n - \bar{x}_{n-1} <$ m z_{max}-mal aufeinanderfolgend erreicht wird.

3.3 Die zweite Phase der Datensammlung

Als Abschluss der aktiven Spielphase wurden 496 Mitwirkende der Dialektsammlung angeschrieben. Hierbei wurde mit einer relativ grossen Beteiligung gerechnet, da durch die vorherige Teilnahme beim Dialekte sammeln von einem grundsätzlichen Interesse für das Thema ausgegangen werden kann. Dies zeigte sich auch daran, dass verschiedene Fragen und Anmerkungen per E-Mail eingetroffen sind. Nach Abschluss dieser Phase sind nun 514 Teilnehmende mit einem Eintrag in der Rangliste präsent.

3.4 Probleme der Selbstdeklaration des eigenen Dialektes

Die Sprachdateien, die für dieses Projekt verwendet wurden, haben eine Angabe der Dialektherkunft in Form einer Postleitzahl. Die grundlegenden Daten sind also vorgegeben. Während der zweiten Spielphase, also während aktiv möglichst viele Daten in Form von geratenen Dialekten gesammelt wurden, erreichte uns eine Rückmeldung, in der der Herkunftsort von einzelnen Dialekten angezweifelt wurde. Eine Frage, die gestellt wurde, war, ob die Dialektherkunft nicht dem Wohnort des Sprechenden entsprechen würde und dass ein Hinweis darauf hilfreich wäre, falls dies der Fall sein sollte. Da der Herkunftsort des Dialektes von den einzelnen Sprechern selber deklariert wurde, sind einzelne Abweichungen zu den effektiven Herkunftsorten sehr gut denkbar, falls diese in dieser exakten Form überhaupt existieren. Die konkrete Frage, die bei der Erfassung der Audiodateien gestellte wurde, lautete: «Geben Sie die Ortschaft an, die Ihren Dialekt am meisten geprägt hat oder diesen am besten repräsentiert, z.B. den Ort, an dem Sie Ihre Schulzeit verbracht haben.»

Ein mögliches Problem dabei ist, dass dies durchaus der den gesprochenen Dialekt prägende Ort sein kann, aber durch Umzug ein anderes Gebiet können sich Dialekteigenschaften mit der Zeit abschwächen und es kann ein Gemisch aus verschiedenen Dialekten entstehen. Daher wäre es interessant zu wissen, wie lange die Sprechenden jeweils wo wohnhaft waren bzw. momentan gerade wohnen. Es ist jedoch anzunehmen, dass nicht jeder diese persönlichen Informationen teilen möchte. Für eine zukünftige Erfassung von neuen Dialektsammlungen wäre sie aber durchaus interessant. Eventuell in einer abgeschwächten Form, bei der nur die Kantone und Zeitintervalle der Aufenthaltsdauer angegeben werden (z.B. ZH: 6-10 Jahre; AG: 11-15 Jahre).

Eine direkte Auswertung neu dazugewonnen Daten wäre sehr aufwendig, aber bei einzelnen auffälligen Dialektbeispielen könnte sie Antworten liefern, ob der gesprochene Dialekt repräsentativ für die gewählte Ortschaft ist oder eventuell durch längere Aufenthalte in anderen Gebieten beeinflusst wurde.

Eine Möglichkeit mit den bestehenden Daten untypische Dialektbeispiele zu eliminieren, bestünde darin, dass sie von den Spielenden markiert werden können. Das Spiel verfügt bereits über eine Möglichkeit Audioaufnahmen zu melden, die bestimmte Mängel aufweisen und könnte hier um den Punkt «Dialekt untypisch für diese Ortschaft» ergänzt werden.

3.5 Welche durchschnittliche Abweichung ist ein gutes Ergebnis?

Sieht man die durchschnittliche Abweichung nach ein paar gespielten Dialekten, fragt man sich unweigerlich, ob das nun ein gutes Ergebnis ist. Wie viele Kilometer Abweichung zum effektiven Dialektherkunftsort ist gut und wann könnte man sogar mit zufälligem Raten bessere Ergebnisse erzielen?
Ein erstes Indiz ist sicher die eigene Position in der Rangliste, aber eventuell sind die anderen Mitspieler einfach extrem gut im Dialekte erkennen oder hatten vielleicht schlicht mehr Glück, falls sie nur gerade zehn Dialekte geraten haben.

Um die Frage der zufälligen Rateversuche zu klären, wurde eine des Bundesamtes für Statistik eine Karte[1] des Bundesamtes für Statistik verwendet, die die offiziellen

[1] https://www.atlas.bfs.admin.ch/maps/13/de/17138_17137_235_227/26599.html

Landessprachen den jeweiligen Regionen zuweist. Da alle rätoromanischen Gebiete ebenfalls Schweizerdeutsch sprechen, wurden diese Gebiete zusammengefasst und die entsprechende Fläche auf der Karte einheitlich rot eingefärbt. Seen wurden dabei ebenfalls eingefärbt, da es sonst im nächsten Schritt zu Problemen kommen könnte. Mittels Python wurde dieses Gebiet nun in ein Polygon verwandelt. Im nächsten Schritt wurde der Mittelpunkt des Polygons bestimmt. Der Mittelpunkt liegt ungefähr in der Mitte des westlichen Ufers des Urnersees. Nun wurden im deutschsprachigen Gebiet zufällig 10000 Punkte gesetzt. Von jedem dieser Punkten wurde die Distanz zum Polygonmittelpunkt erfasst und der Durchschnitt über alle Punkte berechnet. Daraus ergab sich eine Distanz von 68,81 Kilometern.
Dies ist also die durchschnittliche Abweichung, die man bei zufälligem Raten erreichen würde unter der Annahme, dass man das deutschsprachige Gebiet erkennt, was mittels sichtbarer Ortschaftsnamen auf der Karte einigermassen genau machbar ist, und entsprechend nur dort rät.

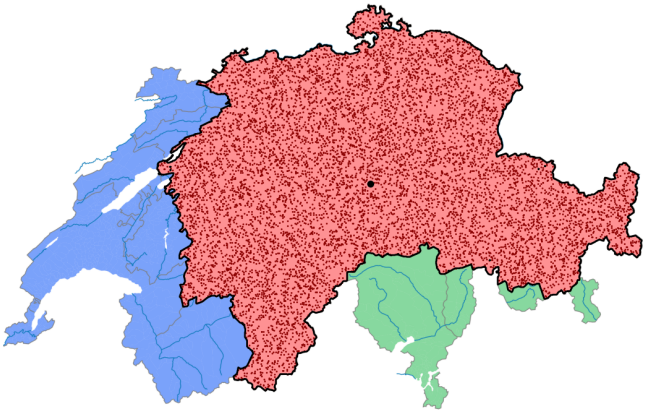

Abbildung 3.1: Deutschsprachige Gebiete (rot) mit 10000 zufälligen Punkten (dunkelrot) und Mittelpunkt (schwarzer Punkt)

In einem zweiten Versuch wurden die Herkunftskoordinaten von jedem Dialektbeispiel genommen und die Distanz zum Fahnenursprungspunkt berechnet, d.h., falls man raten würde, ohne das Fähnchen überhaupt zu bewegen. Das Fähnchen befindet sich am Anfang jedes Rateversuchs in der Ortschaft Rütli, wo sich auch die Rütliwiese befindet, auf der, der Legende nach, der Rütlischwur stattgefunden haben soll. Hier ergibt sich eine minimal grössere Abweichung von 70,90 Kilometern.

Für ein gutes Ergebnis muss die durchschnittliche Abweichung also sehr deutlich unter 70 Kilometern liegen.

3.6 Auswertung der Spieldaten

Insgesamt haben 735 Ratende 19149 Vermutungen abgegeben. Bis zur Abgabe einer Vermutung konnten bis zu fünf Sätze desselben Sprechenden angehört werden. 45707 Sätze wurden tatsächlich angehört, d.h. es wurden im Schnitt 2,39 Sätze angehört, bevor der Entscheid für einen Herkunftsort des Dialektes fiel.

3.6.1 Bereinigung der Daten

Die durchschnittliche geratene Distanz pro Spieler:in ist wie folgt verteilt:

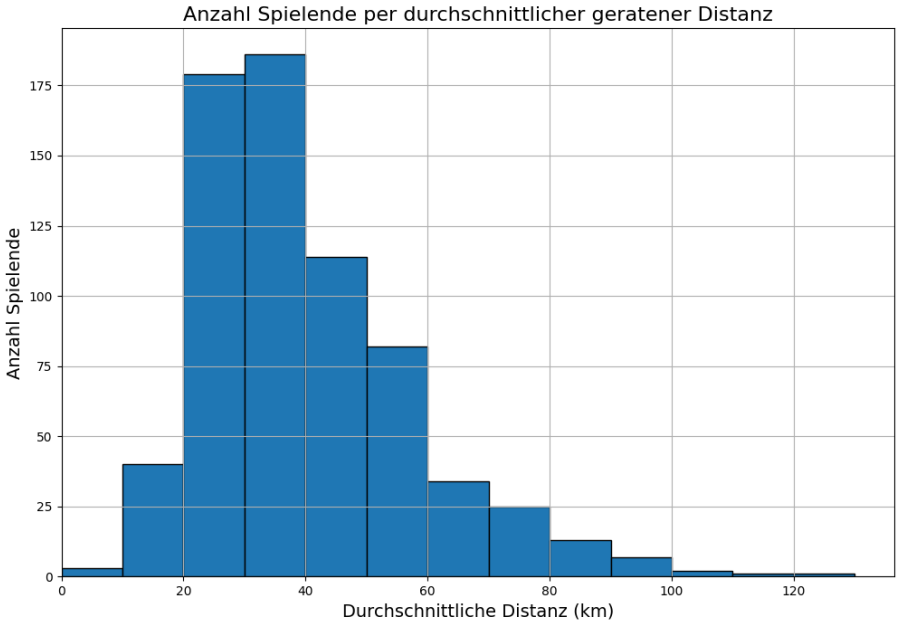

Abbildung 3.2: Durchschnittliche geratene Distanz aller Spielenden

Wie man hier gut sieht, gibt es einige Spielende, die eine sehr grosse durchschnittliche Distanz aufweisen. Das heisst, entweder wurde nur planlos geklickt oder extrem wenige, oder sogar nur einzelne, Dialekte geraten. Es wurde daher entschieden, die Versuche aller Spielenden mit einer durchschnittlichen Ratedistanz von über 70 Kilometern nicht mehr zu berücksichtigen. Die 70 Kilometer wurden unter der Berücksichtigung von zwei Aspekten ausgewählt: Die berechnete Distanz für zufälliges Raten im vorherigen Kapitel und dass die Verteilung so mehr in Richtung Normalverteilung geht. Insgesamt wurden die Resultate von 50 Spielenden gelöscht. Einige davon wiesen nur einen einzigen Dialektrateversuch auf. Zusätzlich wurde bemerkt, dass 92 Versuche mit geratenem Ort Rütli gespeichert wurde. Die allermeisten davon wurden sicherlich durch versehentliches Drücken von «Überprüfen» abgegeben. Nach stichprobenartiger Sichtung der Einträge wurde festgestellt, dass auch die Einträge mit kleiner Distanzabweichung versehentlich getätigt wurden, z.B. insgesamt nur ein Dialekt geraten. Daraufhin wurden alle Rütli-Einträge gelöscht. Insgesamt verblieben so noch 18575 Rateversuche.

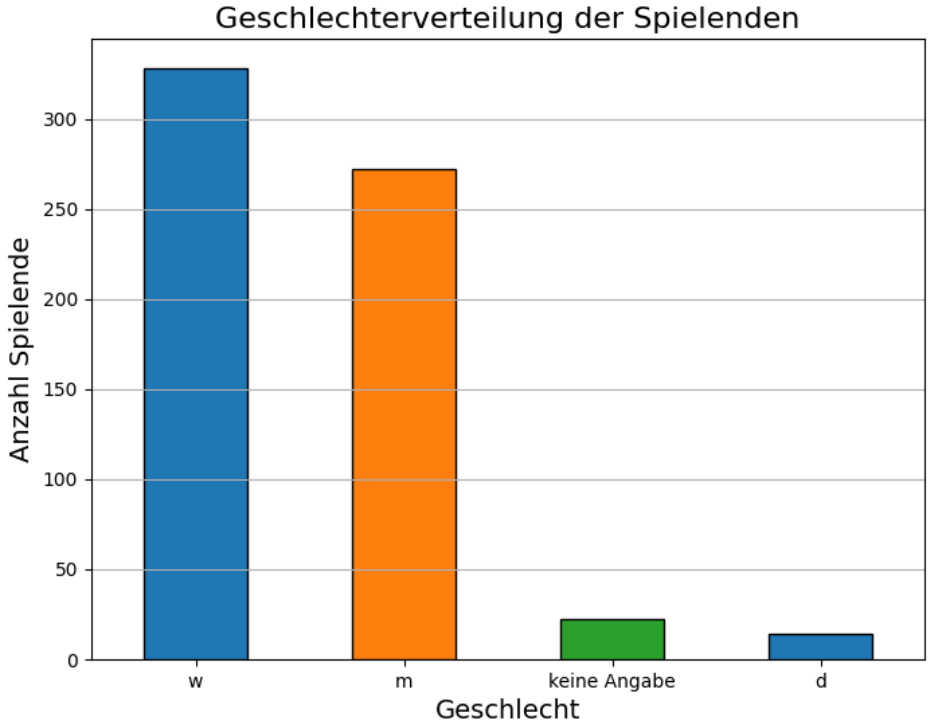

Abbildung 3.3: Geschlechter der Ratenden

Unter den Ratenden waren ungefähr 50 mehr Frauen als Männer vertreten.

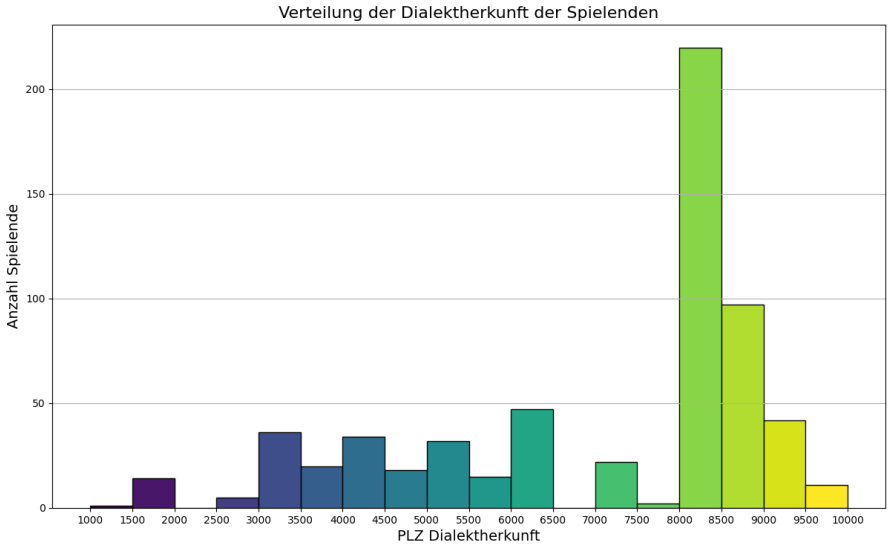

Abbildung 3.4: Postleitzahlen der Dialektherkunftsorte der Ratenden.

In Abb. 3.4 ist ersichtlich, dass die Mehrheit der Spielenden einen Dialekt aus der Region Kanton Zürich/Thurgau sprechen.

Auch wenn, wie in Kap. 3.4 beschrieben, die Unterscheidung von Dialekten anhand der Zuordnung zu den Kantonen nicht zwingend die beste Lösung ist, so bietet sie sich für eine erste Auswertung doch an.

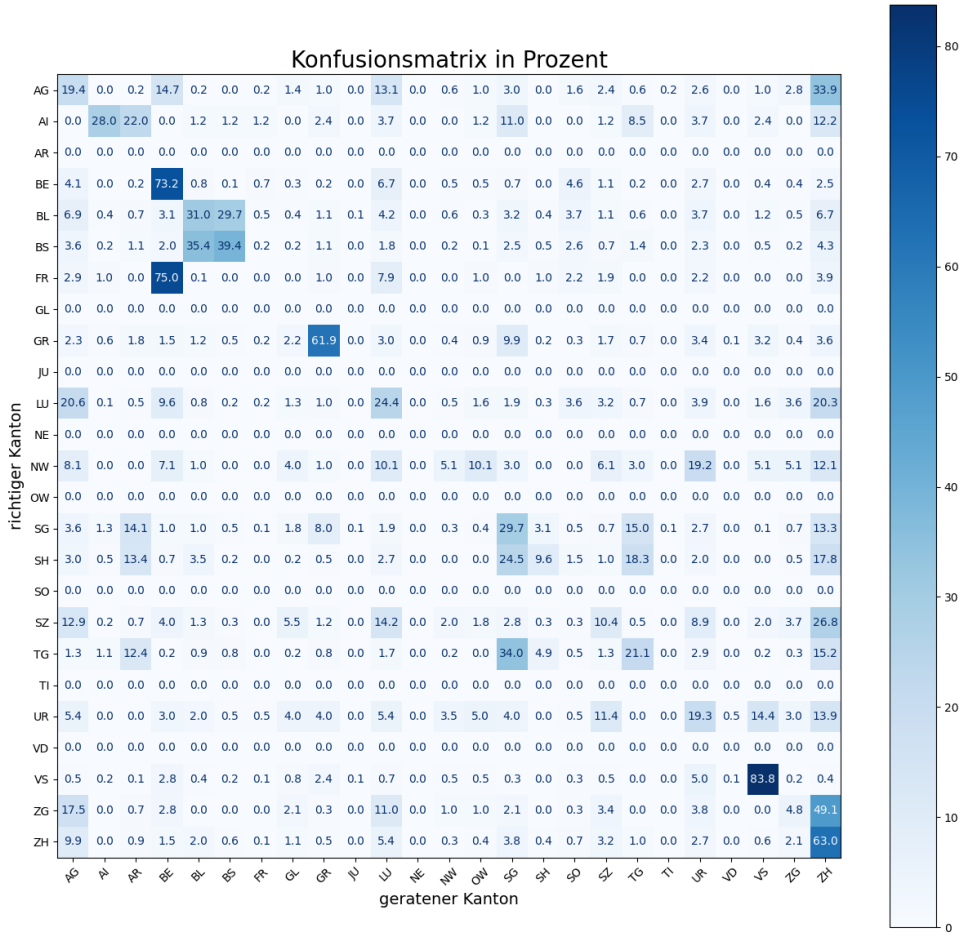

Abbildung 3.5: Konfusionsmatrix aller Kantone. Lesebeispiel: In 73,2% aller Fälle wurden die Dialektbeispiele des Kantons Bern richtig erkannt. Zeilen mit lauter Einträgen 0.0 haben keine eigenen Dialektbeispiele.

In der Konfusionsmatrix sieht man auf einen Blick, welches der richtige Herkunftskanton ist auf der y-Achse und welche Kantone effektiv geraten wurden auf der x-Achse.

Die Hypothese vor Beginn der Datensammlung war, dass Kantone mit besonders prägnanten Dialekten einfacher zu raten wären als Kantone, die relativ ähnliche Dialekte aufweisen. Überträgt man das auf die Dialekttheorie von Kap. 2.2 bedeutet das, dass die niederalemannischen (beide Basel) und höchstalemannischen Dialekte (Mundarten des Alpenraums) einfacher zu erraten sind, als die hochalemannischen Dialekte des Mittellandes.

Auf den Walliser Dialekt scheint das zuzutreffen. Er wird in 83,8 % aller Fälle korrekt erkannt. Für die Fälle, bei denen nicht Wallis gewählt wurde, wurde am meisten auf Uri (5,0 %) und GR (2,4 %) getippt.

Ebenfalls gut erkannt wurde der Berner Dialekt. In 73,2 % wurde hier richtig geraten.

Besonders interessant ist, dass der Kanton Freiburg gar nicht erkannt wurde (0,0 %). Aber 74,1 % aller Dialektbeispiele des Freiburger Dialekts wurden dem Kanton Bern zugeordnet.

Mit 61,9 % wurde der Bündner Dialekt noch richtig erkannt. Am meisten falsche Antworten erhielt dabei der Kanton St. Gallen. Ihm wurden 9,9 % der Bündner Beispiele zugeordnet.

Mit 63,0 % richtigen Antworten liegt Zürich sogar noch vor Graubünden. Die meisten falschen Antworten erhielt hier der Nachbarkanton Aargau (9,9 %).
Aargau selbst wiederum wurde in nur gerade 19,4 % aller Versuche richtig erkannt. 33,9 % der Ratenden tippten stattdessen auf Zürich. Dahinter folgte Bern mit 14,7 % und Luzern mit 13,1 %.

Luzern wiederum wurde am meisten mit dem Aargau (20,6 %) und Zürich (20,3 %) verwechselt. Immerhin 24,4 % aller Versuche wurden richtig getippt.

Ein schwieriger Fall ist Zug, das in nur gerade 4,8 % aller Fälle richtig geraten wurde. Fast die Hälfte (49,1 %) aller Ratenden entschieden sich stattdessen für Zürich. Danach folgt noch Aargau mit 17,5 % und Luzern mit 11,0 %.

Der Urner Dialekt wurde in 19,3 % aller Fälle richtig erkannt. Die meisten falschen Antworten entfielen auf den Kanton Zürich (13,9 %) und die Nachbarkantone Wallis (14,4 %) und Schwyz (11,4 %).

Schwyz wurde am meisten mit Zürich (26,8 %), Luzern (14,2 %) und AG (12,9 %) verwechselt. In 10,4 % aller Versuche gelang die richtige Zuordnung.

Nidwalden wurde viel mit Uri (19,2 %), Zürich (12,1 %), Luzern und Obwalden (je 10,1 %) verwechselt. In 5 % aller Fälle lagen die Ratenden richtig.

Die beiden Basel konnten nicht auseinandergehalten werden und erreichten in beiden Fällen zwischen 29,7 und 39,4 % der Stimmen. Basel-Stadt wurde dabei häufiger richtig geraten. Verwechselt wurden sie am meisten, ausser mit dem jeweiligen anderen Halbkanton, mit Zürich und Aargau (zwischen 3,6 % und 6,9 %).

Schaffhausen, Thurgau und St. Gallen haben eine sehr ähnliche Aufteilung der geratenen Kantone. Am meisten wurde St. Gallen geraten (29,7 % bei St. Gallen, 24,5 % bei Schaffhausen, 34 % bei Thurgau), dann Thurgau (SG: 15 %, SH: 18,3 %, TG: 21,1 %) und Zürich (SG: 13,3 %, SH: 17,8 %, TG: 15,2 %).

Appenzell Innerrhoden wurde in 28 % der Fälle richtig geraten. In 22 % der Fälle wurde stattdessen auf das Ausserrhoden getippt. Die meisten anderen falschen Tipps teilten sich auf zwischen Zürich (12,2 %) und St. Gallen (11,0 %).

3.6.2 Selbsteinschätzung Dialekterkennungsfähigkeit

Bei der Registration wurde nach der selbst eingeschätzten Dialekterkennungsfähigkeit gefragt. Es konnte dabei ein Wert von eins bis fünf gewählt werden. Standardmässig war er auf drei voreingestellt. Um zu testen, wie gut dieser Wert mit den effektiven Spielresultaten korreliert, wurde dieser mit den durchschnittlich erzielten Distanzen verglichen.

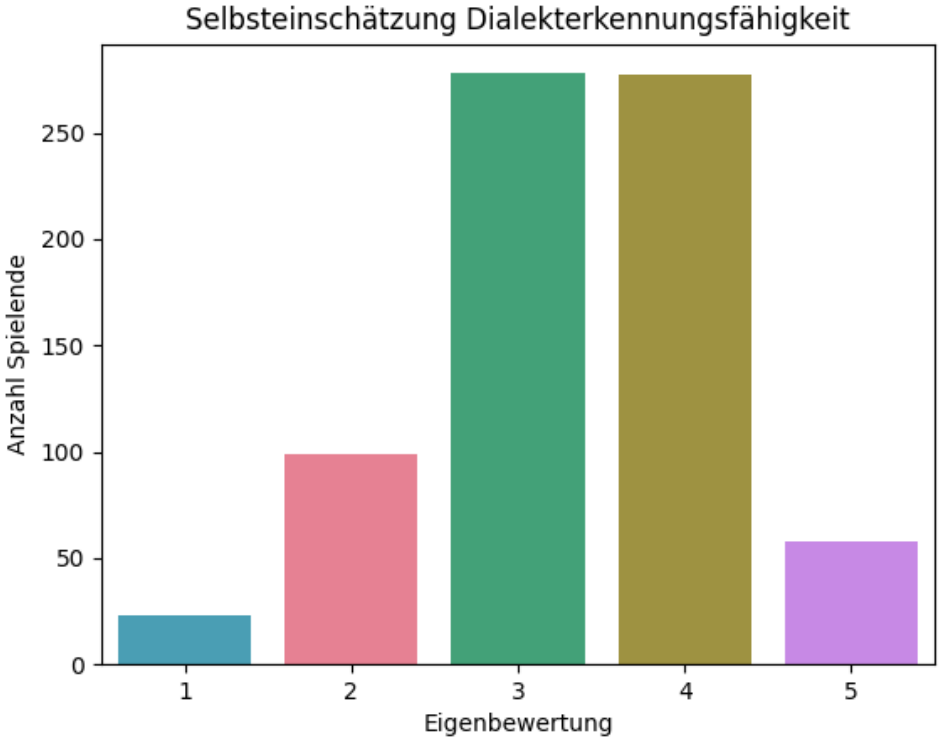

Abbildung 3.6: Eigenbewertung aller registrierter Nutzer

Die Hypothese bestand darin, dass es eine negative Korrelation zwischen Dialekterkennungsfähigkeit und durchschnittlicher Distanz gibt. Dafür wurden mittels Python-Skript die durchschnittliche erzielte Distanz jedes Spielenden berechnet und dann mit dem von ihnen selber eingegebenen Wert verglichen. Es wurde ein Korrelationskoeffizient von gerundet -0,24 erhalten, was eine schwache negative Korrelation bedeutet. Der p-Wert liegt bei gerundet $1,35 \times 10^{-9}$, was zeigt, dass das Ergebnis statistisch signifikant ist. Es besteht also ein, wenn auch schwacher, Zusammenhang zwischen Eigenbeurteilung und erzieltem Resultat.

3.6.3 Gibt es einen Geschlechtsunterschied beim Dialekte erkennen?

Ebenfalls interessant ist die Frage, ob Frauen oder Männer besser im Dialekte erkennen sind. Dafür wurden die durchschnittlichen Distanzen für jedes Geschlecht berechnet sowie die durchschnittlich gespielten Versuche.

Geschlecht	Anzahl Spielende	$\overline{\text{Distanz}}$	$\overline{\text{Anzahl Versuche}}$
w	328	34,56	30,0
m	272	33,77	28,3
d	14	30,00	33,3
keine Angabe	22	31,42	26,1

Tabelle 3.2: Durchschnittliche geratene Distanz und Anzahl Versuche pro Geschlecht

Hier zeigt sich, dass die Männer-Gruppe geringfügig besser geraten haben als die Frauen. Die Frauen haben dafür im Schnitt einen Versuch mehr gespielt. Die Gruppe Divers hat am meisten Versuche gespielt, erzielten aber einen schlechteren Durchschnitt, wobei die geringe Anzahl Spielende natürlich einen grossen Einfluss hat und die Ergebnisse dadurch nicht direkt verglichen werden können.

Altersgruppe	Anzahl Spielende	$\overline{\text{Distanz}}$	$\overline{\text{Anzahl Versuche}}$
U18	6	36,87	19,5
18-25	260	32,32	29,8
26-35	188	32,32	32,2
36-45	47	31,28	23,6
46-55	36	28,59	30,6
56-65	35	38,25	20,8
66-75	13	33,23	32.6
76-85	5	38,87	29,0
86+	2	40,50	27,5
k. A.	44	34,97	24,7

Tabelle 3.3: Durchschnittliche geratene Distanz und Anzahl Versuche pro Altersgruppe

3.6.4 Gibt es einen Altersunterschied beim Dialekte erkennen?

In der Tabelle 3.3 ist eine Tendenz sichtbar, dass die sechs jüngsten Spielenden in den Alterskategorien U18 nicht ganz mit den vier folgenden älteren Kategorien mithalten können. Allerdings haben sich auch durchschnittlich am wenigsten Versuche gespielt. Die Ergebnisse werden mit ansteigendem Alter laufend besser, bzw. bleiben gleich gut, bis die Kategorie 56-65 wieder ein schlechteres Ergebnis erzielt. Diese haben aber auch im Schnitt zwei bis elf Dialekte weniger geraten als die jüngeren Kategorien. Mit der Kategorie der 66- bis 75-Jährigen wird die durchschnittlich geratene Distanz wieder kleiner, also besser. Danach wächst sie in den zwei ältesten Kategorien wieder an. Allerdings ist hier der Stichprobenumfang mit fünf und zwei Spielenden auch deutlich zu klein.

Auf eine weitere Aufteilung auf Geschlechter wurde verzichtet, da die Kategorien dann noch weniger Spielende aufweisen würden und dies statistisch nicht mehr sinnvoll wäre.

3.6.5 Auswertung nach Dialektregionen

Die in Abbildung 2.2 gezeigte Aufteilung in Dialektregionen findet hier wieder Verwendung. Anstatt nur die Trefferquote für die einzelnen Kantone zu betrachten, wird dasselbe nun mit Dialektregionen gemacht.

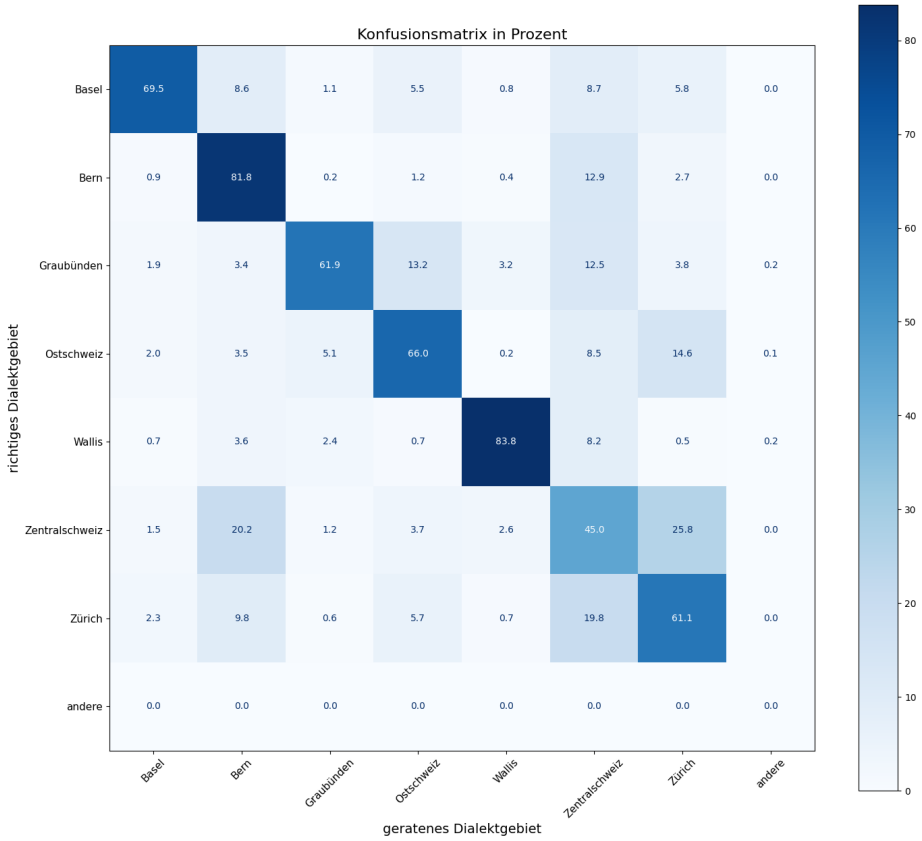

Abbildung 3.7: Konfusionsmatrix aller Dialektregionen in Prozent

Man sieht in dieser Abbildung schön die ausgeprägte Diagonale, was heisst, dass jeweils die Mehrheit der Rateversuche das richtige Dialektgebiet erwischt haben. Die Zentralschweiz bereitet Mühe und wird häufig mit Zürich und Bern verwechselt. Eventuell ist hier der Luzerner Dialekt dem Kanton Aargau zugeordnet worden und dabei sind dessen Berner und Zürcher Dialektgebiete ausgewählt worden. Es kann aber natürlich auch sein, dass direkt der Kanton Zürich gewählt wurde, da beide Dialekte im Allgemeinen nicht besonders prägnant sind. Schaut man sich das auf Kantonsebene an, sieht man das meistens AG und ZH ausgewählt wurde anstatt LU und BE wurde am Drittmeisten gewählt.
Wallis ist, nicht unerwartet, gut erkannt worden, ebenso wie Bern. Für Bern fielen jetzt natürlich die potenziellen Verwechslungsmöglichkeiten auf Kantonsebene mit Solothurn oder Freiburg weg. Das Ostschweizer Dialektgebiet profitiert davon, dass die Verwechslungen hier vor allem innerhalb der Kantone untereinander stattgefunden haben, was jetzt ebenfalls wegfällt. Zürich bekommt hier immer noch einige Prozent an falschen Vermutungen ab. Graubünden wird weniger gut als Bern oder Wallis erkannt und wird mit der Ostschweiz und Zentralschweiz verwechselt. Die Ostschweizer Prozente sind primär dem Kanton St. Gallen zuzuordnen. In Basel wurde ja primär BS und BL verwechselt, was jetzt auch wegfällt.
Die «andere» Bezeichnung enthält Kantone, die nicht in den betrachteten Dialektgebiete liegen, also nicht deutschsprachig sind.

3.6.6 Auswertung nach Dialektherkunftsorten der Ratenden

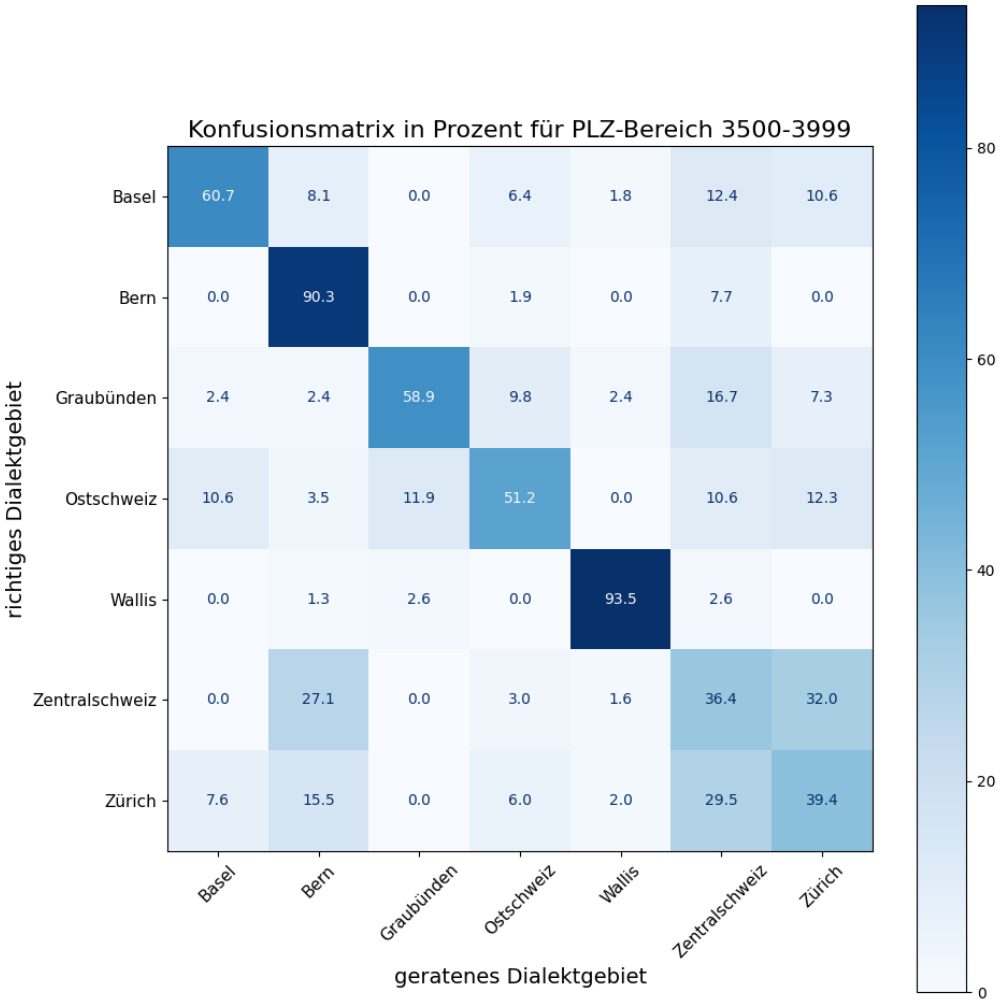

Abbildung 3.8: Konfusionsmatrix in Prozent für den PLZ-Bereich 3500-3999 (Ortschaften in den Kantonen Bern und Wallis)

Die Ratenden der in Abb. 3.8 dargestellten Konfusionsmatrix haben einen Dialekt mit Herkunftsort in den Kantonen Bern und Wallis. Man sieht hier schön, dass sie mit diesen beiden Dialektregionen wenig Probleme hatten. Basel wird nicht ganz so gut erkannt. Am meisten Mühe bereiteten die Gebiete Zentralschweiz und Zürich, die gegenseitig verwechselt wurden. Die Ostschweizer Dialektgebiete werden auch unterdurchschnittlich erkannt und fast gleichmässig mit Basel, Graubünden, Zentralschweiz und Zürich verwechselt.

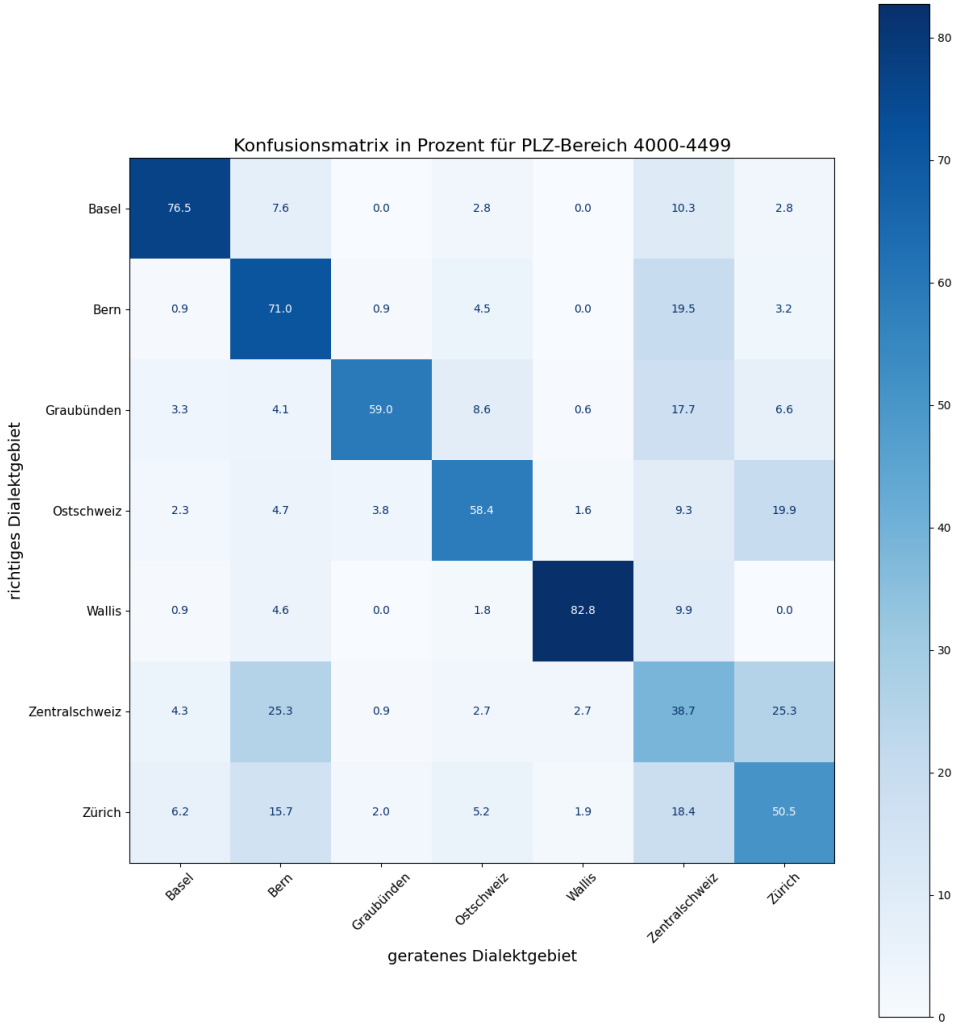

Abbildung 3.9: Konfusionsmatrix in Prozent für den PLZ-Bereich 4000-4499 (Ortschaften in den Kantonen Basel-Stadt, Basel-Landschaft, Solothurn und Aargau)

Dieser PLZ-Bereich umfasst Ortschaften in den Kantonen Basel-Stadt, Basel-Landschaft, Solothurn und Aargau. Ausser Wallis, das fast immer gut erkannt wird, werden auch die Dialektregionen Basel und Bern gut erkannt. Wie schon beim vorherigen PLZ-Bereich sind auch hier die Gebiete Zürich und Zentralschweiz nicht gut erkannt worden bzw. miteinander verwechselt worden.

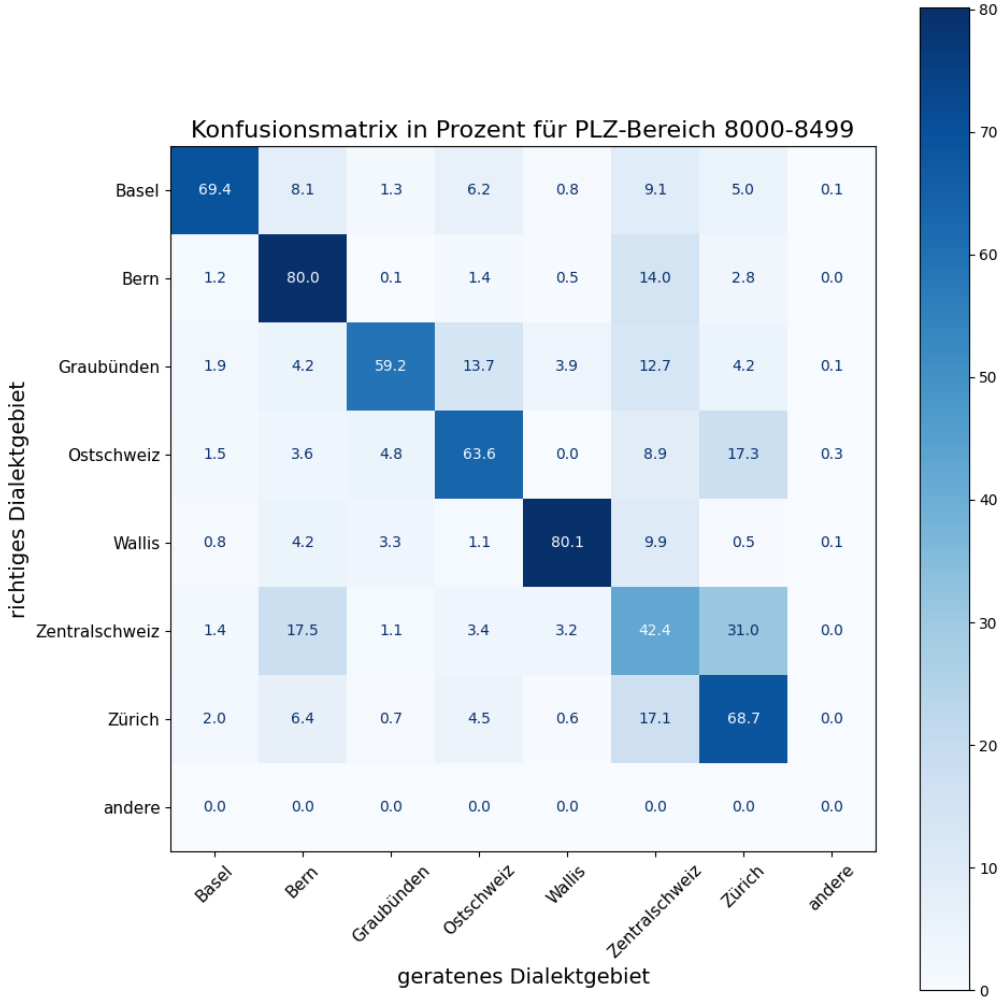

Abbildung 3.10: Konfusionsmatrix in Prozent für den PLZ-Bereich 8000-8499 (Ortschaften im Kanton Zürich)

Der hier dargestellte PLZ-Bereich umfasst Ratende mit Dialekten aus der Region Zürich und Winterthur. Zürich wird hier im Vergleich gut erkannt. Falls nicht richtig geraten wurde, entfielen die Vermutungen auf das Dialektgebiet Luzern. Auch die Dialektgebiete Bern und Wallis wurden gut erkannt, aber weniger gut, als das die Ratenden des PLZ-Bereichs 3500-3999 vermochten, die diese Dialekte selber sprechen.

Weitere Konfusionsmatrizen von anderen PLZ-Bereichen sind im Anhang bei 5.3 zu finden.

3.7 Fazit und Ausblick

Grundsätzlich kann die Entwicklung dieses Dialektratespiels (inoffizieller interner Name: dialect guesser) als Erfolg angesehen werden. Das Ziel eine Web-Applikation zu entwickeln, die einfach zu bedienen ist und Spass macht gespielt zu werden, wurde aus der Sicht der Autoren erreicht. Auch die Rückmeldungen waren bis jetzt durchwegs positiv. Die Feuertaufe mit vielen gleichzeitigen Nutzern und bis zu vier geratenen Dialekten pro Sekunde im Rahmen der Datensammlung an der ZHAW hat die Applikation schadlos überstanden. Gerade das Datensammeln bereitete anfangs einige Kopfschmerzen, da es schwierig war, genügend potenzielle Spielende zu motivieren sich zu registrieren und ein paar Runden zu spielen. Die Idee mit der Gewinnmöglichkeit von Gutscheinen, an die die Bedingung geknüpft war, bestimmte Leistungen zu erreichen, erwies sich als sehr erfolgreich. Sozusagen vom Homo economicus zum Homo ludens, auch wenn natürlich immer noch mindestens partiell ökonomisch motiviert.

Die Wahl der verwendeten Technologien stellte eine erste schwierige Entscheidung dar. Im Nachhinein betrachtet kann gesagt werden, dass sie ein Erreichen der gesteckten Ziele möglich gemacht haben und darum schon mal nicht grundsätzlich falsch sein können. Der Ansatz Front- und Backend zu trennen hat gut funktioniert und ermöglicht es andere Frontendtechnologien einzusetzen, sollte in Zukunft die Notwendigkeit dazu bestehen. Python mit FastAPI hat als Backend überzeugt durch einfache Bedienung, gute Performance und automatische Dokumentation der verfügbaren Schnittstellen. Next.js bietet eine Vielzahl von Funktionen, die im Rahmen dieses Projektes nicht ansatzweise alle eingesetzt werden konnten. Einzig eine direkt integriert Authentifikation fehlt bislang. Die vielen Funktionen und der Paradigmenwechsel von Page Router zu App Router machten das Arbeiten mit Next.js 14 nicht immer ganz einfach. Gerade weil Fullstack-Entwicklung schon sonst genügend Herausforderungen bietet. Als Beispiel sei die temporäre Schwierigkeit genannt, die Audiodateien auf iPhones zum Abspielen zu bewegen, während es mit allen anderen Geräten problemlos funktionierte.

Es sind verschiedenste Weiterentwicklungsmöglichkeiten denkbar, von denen hier ein paar Ideen festgehalten werden sollen. Ein Beispiel wären verschiedene Nutzerrollen, so dass nach dem Einloggen durch Anklicken eines Nutzers in der Rangliste die verschiedenen geratenen Dialekte angezeigt werden können. Anstatt nur Dialekte zu raten, wäre eine Funktion denkbar, bei der dieselben Sätze in mehreren Dialekten angehört werden könnten. Weiter gäbe es Möglichkeiten bei Anzeige der Auflösung die der anderen Nutzer ebenfalls anzuzeigen.

Das Auswerten der gesammelten Daten bot seine eigenen Herausforderungen. Gerade der Entscheid, welche Daten überhaupt für eine Auswertung verwendet werden sollten, sorgte für Diskussionen. Einerseits war das Ziel möglichst viele der gesammelten Daten auch zu nutzen, andererseits waren einige Spielende im Vergleich zu den anderen so schlecht, dass man annehmen musste, dass sie entweder extrem ziellos oder sogar absichtlich falsch geraten haben. Anders sind vermutete Dialektherkunftsorte in den Kantonen Tessin oder Genf eigentlich nicht zu erklären.

Wir konnten feststellen, dass eine kleine negative Korrelation zwischen der Selbsteinschätzung und den erzielten durchschnittlichen geratenen Distanzen besteht. Das heisst, wer sich selber als gut im Dialekterkennen einschätzt, ist es häufig auch.

Das Geschlecht der Ratenden hatte jedoch keinen signifikanten Einfluss auf den Rateerfolg. Einen geringen Einfluss hatte das Alter. Hier spielte aber vermutlich auch die stark unterschiedliche Anzahl Spielende und unterschiedliche Anzahl Versuche eine grössere Rolle.

Die Herkunft des Ratenden hatte einen grösseren Einfluss, da die eigene oder nahegelegene Dialektregionen besser erkannt wurden.

Dass die prägnanten Dialekte wie der Walliser Dialekt gut erkannt wurden, war weniger überraschend. Interessant war zu beobachten, dass die Erkennungsrate auch abhängig vom Dialekt des Ratenden ist.

Bei der Auswertung auf der Kantonsebene war interessant zu beobachten, wie die Kantone innerhalb der Dialektgebiete verwechselt wurden, besonders bei TG/SG/SH und BL/BS war dies besonders ausgeprägt. Interessant wäre zu wissen, ob die Zuteilung einfacher würde, wenn z.B. ein Basel-Landschaft- und Basel-Stadt-Dialektbeispiel jeweils im direkten Vergleich angehört werden könnte. Eventuell trifft aber hier auch das Problem der Dialektdurchmischung auf, wie es in Kap. 3.4 angesprochen wurde und die Beispiele sind nicht so eindeutig nur Stadt oder Landschaft zuzuordnen.

Generell hätte der Teil der Auswertung noch einiges an Potenzial zu bieten, der in dieser Arbeit am Schluss wegen Zeitmangel nicht vollständig ausgeschöpft werden konnte. Insbesondere das «Aargau-Problem» (siehe Kap. 2.4) müsste nochmals betrachtet werden.

Kapitel 4

Technische Dokumentation

Da der Fokus dieser Bachelorarbeit auch auf der Entwicklung der Web-Applikation lag, soll hier genauer auf den Aufbau und die verwendeten Komponenten eingegangen werden. Da die Applikation im Rahmen von zukünftigen Projekt- oder Bachelorarbeiten weiterentwickelt werden könnte, ist eine gute Dokumentation wichtig.

4.1 Verwendete Technologien

In diesem Kapitel werden die wesentlichen Technologien vorgestellt, die bei der Entwicklung unserer Web-Applikation zum Einsatz gekommen sind. Diese Technologien werden in vier Kategorien eingeteilt: Frontend, Backend, Datenbank und Reverse-Proxy. Im Bereich Frontend erläutern wir die Nutzung von Next.js, einem React-Framework, sowie Leaflet für interaktive Karten und die Verwendung von VectorTiles. Das Backend wird hauptsächlich mit Python und FastAPI entwickelt, unterstützt von SQLAlchemy für die Datenbankinteraktion und Docker für die Containerisierung. Zusätzlich setzen wir S3Container für die Speicherung der Audiodateien, Nominatim und Swisstopo für das Geocoding und Nginx als Reverse-Proxy vor dem Webserver ein. Jede Technologie wird detailliert beschrieben, wobei kurz auf ihre jeweiligen Vor- und Nachteile eingegangen wird. Zudem wird begründet, warum diese Technologien ausgewählt wurden und welche Alternativen evaluiert wurden.

4.1.1 Frontend

Beim Frontend handelt es sich um die Benutzeroberfläche, die für die Web-Applikation verwendet worden ist. Hier werden die verwendeten Tools und Frameworks beschrieben und aus welchem Grund diese verwendet werden. Der Fokus liegt dabei auf Next.js und dem Herzstück des Projektes, die interaktive Karte Leaflet.

Next.js

Next.js ist eine Webframework, das die React JavaScript-Bibliothek um zusätzliche Funktionen erweitert, um sowohl die Performance als auch die Benutzerfreundlichkeit für Entwickler zu verbessern. Durch serverseitiges Rendering sorgt es für schnelle Ladezeiten. Gleichzeitig ermöglicht Next.js eine nahtlose Integration mit verschiedenen Datenquellen und APIs. Diese ermöglicht es hoch interaktive und reaktionsschnelle Webanwendungen zu erstellen, die auf allen Geräten reibungslos laufen. [20]

Beworbene Vorteile von Next.js:

- Performance: Statische Seitengenerierung verbessert die Ladezeiten.
- Einfaches Routing: Dateibasiertes Routing erleichtert die Verwaltung von Routen.
- Flexibilität: Unterstützt sowohl statische als auch dynamische Websites.

Bekannte Nachteile von Next.js:

- Komplexität: Kann für einfache Projekte überdimensioniert sein.
- Build-Zeiten: Grosse Projekte können lange Build-Zeiten haben.
- Serveranforderungen: Serverseitiges Rendering erfordert mehr Serverressourcen.

Die Entscheidung fiel auf Next.js, da es eine niedrige Einstiegshürde aufweist, schon in der 14. Version verfügbar ist, was die Chance auf kontinuierliche Weiterentwicklung erhöht und auch von vielen grossen Konzernen (Nike, Spotify, Sonos u.a.) eingesetzt wird. Features wie das serverseitige Rendering, das die Ladezeiten für Nutzende verringert, sind dabei nur ein Bonus und haben nicht zur Entscheidung beigetragen. Ein zusätzlicher Grund war jedoch, dass das Projekt als reine Webapplikation implementiert werden sollte, so dass dieses auf jedem Gerät mit Browser lauffähig ist.

Das Entwicklungskit Flutter von Google, das auch zur Diskussion stand, setzt den Fokus mehr auf mobile Applikationen für die jeweiligen Plattformen und nutzt Dart als Programmiersprache, für die keine Vorkenntnisse bestanden.

Leaflet

Leaflet[21] ist eine Open-Source-JavaScript-Bibliothek, die es ermöglicht, mobile-freundliche, interaktive Karten in Webanwendungen einzubinden. Sie zeichnet sich durch ihre Einfachheit, Benutzerfreundlichkeit und Performance aus und bietet eine Vielzahl von Funktionen zur Kartenvisualisierung und -interaktion. Dank einer aktiven Entwicklergemeinschaft gibt es zahlreiche Plugins und Erweiterungen, die zusätzliche Funktionalitäten bieten.

Beworbene Vorteile von Leaflet:

- Einfach zu verwenden: Klare und intuitive API, die schnelle Integration ermöglicht.
- Leichtgewichtig: Geringe Dateigrösse sorgt für schnelle Ladezeiten.
- Mobilfreundlich: Optimiert für Touch-Bedienung und mobile Geräte.
- Flexibel: Unterstützt eine Vielzahl von Kartentypen und -quellen.

Bekannte Nachteile von Leaflet:

- Skalierbarkeit: Kann bei sehr grossen Datensätzen und umfangreiche Karten Leistungseinbussen haben.
- Kompatibilität: Ältere Browser werden teilweise nicht unterstützt.
- Weniger umfassend als Alternativen: Konkurrenzprodukte wie Google Maps bieten umfangreichere APIs und Funktionen.

Die Entscheidung fiel auf Leaflet, weil sich Karten von verschiedenen Anbietern (z.B. OpenStreetMap oder swisstopo) einbinden lassen. Unterstützte Technologien sind unter anderem Vector Tiles, Web Map Tiling Services (WMTS) und GeoJSON. Leaflet erfordert im Gegensatz zu Konkurrenten wie Google Maps keine Authentifizierung, da nur Kartenmaterial externer Anbieter verwendet wird. Verwenden diese keine Authentifizierung, fällt diese also komplett weg. Ein weiterer Vorteil ist die Optimierung für mobile Geräte, was einen reibungslosen Einsatz auf verschiedenen Plattformen gewährleistet.

Lucia Auth

Lucia Auth [22] ist eine Authentifizierungsbibliothek, die die Komplexität der Session-Verwaltung vereinfacht. Sie arbeitet mit einer Datenbank zusammen, um eine API bereitzustellen, die einfach zu nutzen, verständlich und erweiterbar ist.

Beworbene Vorteile von Lucia:

- Gute Dokumentation: Lucia hat eine gute Dokumentation, wie diese mit verschiedenen Technologien und Datenbanken eingebunden wird.
- Einfache Konfigurationen: Entfernt die Komplexität von Authentifizierung und Session-Erstellung.

Bekannte Nachteile von Lucia:

- Datenbankabhängigkeit: Es wird eine Datenbank benötigt, um die Sessions zu verwalten.

Bei der Entscheidung wurden NextAuth und Lucia Auth in Betracht gezogen. Der Grund, dass Lucia Auth ausgewählt worden ist der, dass Lucia Auth bei Next.js Projekten eine gute Dokumentation hat, alle notwendigen Features beinhaltet und die Authentifizierung bei NextAuth sehr einfach mit externer Authentifizierung implementieren werden kann, eine eigene Implementierung jedoch schwieriger ist.

4.1.2 Backend

Beim Backend handelt es sich um den Teil der Web-Applikation, welche sich um die serverseitigen Aufgaben kümmert. Hier wird darauf eingegangen, wie die Kommunikation mit der Datenbank erfolgt, Daten gespeichert und zur Verfügung gestellt werden.

FastAPI

FastAPI [23] ist ein modernes, schnelles (High-Performance) Web-Framework für Python, das auf Standard Python Annotations basiert. Es wird verwendet, um APIs zu erstellen, und bietet automatisierte Dokumentation durch die Nutzung von OpenAPI und JSON-Schema. FastAPI ist bekannt für seine hohe Leistung, vergleichbar mit Node.js und Go, und ermöglicht eine schnelle Entwicklung dank einfacher und intuitiver Syntax.

Beworbene Vorteile von FastAPI:

- Hohe Leistung: Dank der Nutzung von unterschiedlichen Tools bietet FastAPI eine Geschwindigkeit, die mit den schnellsten Frameworks vergleichbar ist.
- Automatische Generierung von Dokumentation: FastAPI generiert automatisch interaktive API-Dokumentationen mithilfe von Swagger UI und ReDoc.
- Asynchrone Unterstützung: FastAPI unterstützt nativ asynchrone Programmierung, was zu einer besseren Skalierbarkeit führt.

Bekannte Nachteile von FastAPI:

- Weniger Ressourcen und Tutorials: Aufgrund der Neuheit gibt es weniger Lernmaterialien und Ressourcen.
- Komplexität bei grossen Projekten: Bei sehr grossen Projekten kann die Verwaltung der Abhängigkeiten und Konfiguration komplex werden.

Die Entscheidung fiel auf FastAPI, da diese sehr schnell und einfach mit Python implementiert werden kann. Die REST APIs, die erstellt werden, werden automatisch mithilfe der Swagger UI dokumentiert und die Schemas und Modelle sind somit sehr übersichtlich.

SQLAlchemy

SQLAlchemy [24] ist ein SQL-Toolkit und eine Objekt-Relational Mapping (ORM) Bibliothek für Python. Es ermöglicht Entwicklern, Datenbankabfragen und -operationen in Python-Code zu schreiben, ohne direkt SQL schreiben zu müssen. SQLAlchemy unterstützt viele Datenbanktypen und bietet eine leistungsfähige Abstraktionsschicht, die den Umgang mit Datenbanken vereinfacht.

Beworbene Vorteile von SQLAlchemy:

- Sicherheit: Verringert das Risiko von SQL-Injektion-Angriffen durch automatische Eingabevalidierung und -sicherung.
- Flexibilität: Unterstützt mehrere Datenbanksysteme und kann leicht zwischen ihnen wechseln.
- Leistungsfähig: Ermöglicht komplexe Abfragen und Transaktionen.

Bekannte Nachteile von SQLAlchemy:

- Performance-Overhead: Die Abstraktionsschicht kann zu Performance-Einbussen führen.
- Einrichtungsaufwand: Erfordert zusätzliche Konfiguration und Einrichtung.
- Abhängigkeit: Projekte können stark von SQLAlchemy abhängig werden, was die Flexibilität einschränken kann.

Die Entscheidung fiel auf SQLAlchemy, weil es durch automatische Eingabevalidierung Sicherheit bietet vor SQL-Injektion. Ein weiterer Vorteil ist es, dass selbst keine aufwändigen SQL-Queries geschrieben werden müssen und Funktionalitäten von Python, während den Abfragen, verwendet werden können. Somit können schwierigere Aufgaben einfacher umgesetzt werden.

Docker

Docker [25] ist eine Open-Source-Plattform zur Automatisierung der Bereitstellung von Anwendungen in Containern, die isolierte Umgebungen bereitstellen, um Software und ihre Abhängigkeiten konsistent über verschiedene Entwicklungs- und Produktionsumgebungen hinweg zu betreiben. Container sind leichtgewichtig und portabel, was die Entwicklung, das Testen und die Bereitstellung von Anwendungen vereinfacht. Docker ermöglicht die schnelle Skalierung und Verwaltung von Anwendungen.

Beworbene Vorteile von Docker:

- Portabilität: Docker-Container laufen unabhängig von der Infrastruktur.
- Isolierung: Anwendungen und ihre Abhängigkeiten sind in Containern isoliert.
- Effizienz: Container sind leichtgewichtig und verbrauchen weniger Ressourcen als virtuelle Maschinen.
- Skalierbarkeit: Einfaches Hochskalieren von Anwendungen und Diensten.

Bekannte Nachteile von Docker:

- Komplexität: Verwaltung und Orchestrierung vieler Container können komplex sein.
- Sicherheitsrisiken: Container teilen sich denselben Kernel, was potenzielle Sicherheitslücken eröffnet.
- Persistenz: Datenpersistenz in Containern kann herausfordernd sein.
- Inkompatibilitäten: Nicht alle Anwendungen sind für die Ausführung in Containern geeignet.

Die Entscheidung fiel auf Docker-Container, weil diese schnell und einfach auf anderen Servern eingerichtet werden können, ohne grosse Änderungen vorzunehmen. Ein Vorteil ist, dass Container, die über Docker compose gestartet werden oder sich im gleichen Docker-Netzwerk befinden, nicht über die IP, sondern über die Containernamen miteinander kommunizieren können. Docker stellt ausserdem viele Docker-Container im eigenen Docker Hub zur Verfügung, so dass diese nicht eigenständig erstellt werden müssen.

S3-Speicher

S3 steht für Simple Storage Service, ein cloudbasierter Filehostingdienst (Object Storage), welcher von Amazon entwickelt wurde, aber auch von Drittanbietern angeboten wird. Die Abrechnung erfolgt hier anhand der effektiven Nutzung (Speicherplatz, Zugriffe). Eine externe Speicherung der Daten kam für unser Projekt nicht infrage, aufgrund von Datenschutz und zusätzlichen Kosten und so wurde der S3Proxy von Andrew Gaul ausgewählt. So bleiben die Daten lokal auf dem Server gespeichert und können per API abgefragt werden.

Der S3Proxy [26] ist ein Open-Source-Tool, das es ermöglicht, verschiedene storage buckets über eine S3-kompatible API zu nutzen. Es fungiert als Vermittler, der Anfragen, die an eine S3-Schnittstelle gesendet werden, an verschiedene unterstützte Speicherziele weiterleitet, darunter lokale Dateisysteme, OpenStack Swift, Google Cloud Storage und andere.

Beworbene Vorteile von S3Proxy:

- Interoperabilität: Ermöglicht die Nutzung verschiedener Speicherlösungen mit einer einheitlichen S3-API.
- Flexibilität: Unterstützung für zahlreiche Backend-Speicherlösungen, sowohl Cloudbasierte als auch lokale.
- Open Source: Kostenlos und anpassbar, mit einer aktiven Community und Weiterentwicklung.

Bekannte Nachteile von S3Proxy:

- Komplexität: Erfordert technisches Wissen und Aufwand für die Konfiguration und Wartung.
- Fehlende Features: Nicht alle S3-Funktionen und -Eigenschaften werden möglicherweise vollständig unterstützt.
- Community Support: Abhängigkeit von Community-Support, der weniger umfangreich sein kann als bei kommerziellen Lösungen.

Die Gründe, weshalb der S3Proxy ausgewählt worden ist: Es sind keine Lizenzen notwendig, die Daten sind lokal gespeichert und können per API abgefragt werden. Durch den S3Proxy besteht die Möglichkeit den S3-Service in der Zukunft einfach auswechseln zu können, für den Fall, dass ein externer Dienst verwendet werden soll anstelle des lokalen Dateisystems. Auch der S3Proxy steht als Image im Docker Hub zur Verfügung.

Nominatim

Nominatim ist ein Open-Source-Geokodierungsdienst, der auf den Daten von OpenStreetMap (OSM) basiert.[27] Es wandelt Adressen und Ortsnamen in geografische Koordinaten um und umgekehrt. Dies wird als Geokodierung (geocoding) und inverse Geokodierung (reverse geocoding) bezeichnet. Nominatim wird häufig in geografischen Informationssystemen (GIS) und verschiedenen Anwendungen verwendet, die Ortsdaten benötigen.

Beworbene Vorteile von Nominatim:

- Kostenlos und Open-Source: Nominatim ist kostenlos und der Quellcode ist offen, was die Anpassung und Integration erleichtert.
- Aktuelle Daten: Es verwendet die ständig aktualisierten Daten von OpenStreetMap, was zu aktuellen und präzisen Ergebnissen führt.
- Integration: Leicht in andere Anwendungen und Systeme integrierbar, da es über eine API verfügt.

Bekannte Nachteile von Nominatim:

- Datenqualität: Die Genauigkeit hängt von der Qualität und Aktualität der OpenStreetMap-Daten ab, die regionsabhängig variieren kann.
- Geschwindigkeitsbeschränkungen: Öffentliche Instanzen von Nominatim haben Abfrageratenbeschränkungen, die die Nutzung einschränken können.
- Ressourcenintensiv: Der Betrieb einer eigenen Nominatim-Instanz erfordert erhebliche Serverressourcen, insbesondere für sehr grosse Datenbanken.

Die Entscheidung für das Reverse Geocoding fiel auf Nominatim, weil geplant wurde alle Daten entweder lokal oder über eine API zur Verfügung zu stellen. Da bei jeglichen APIs die Nutzung stark limitiert ist oder Lizenzen benötigt, wurde entschieden Nominatim zu verwenden. Auch die öffentlich zugängliche API von Nominatim selbst untersteht strengen Regeln, die kontinuierliche Abfragen, wie in diesem Fall benötigt, verbieten. Darum wurde entschieden, Nominatim lokal zu hosten. Es steht ebenfalls bereits im Docker Hub als Container zur Verfügung.

4.1.3 Datenbank

Für die Speicherung von Daten, die vom Front- und Backend verwendet werden, wird je eine Datenbank eingesetzt. Nachfolgend soll kurz die Auswahl der Datenbanktechnologie erläutert werden.

SQLite[28] ist eine kompakte, serverlose SQL-Datenbank-Engine, die in eine Anwendung eingebettet werden kann. Im Gegensatz zu anderen Datenbanksystemen, die als separate Serversoftware installiert und konfiguriert werden müssen, speichert SQLite die gesamte Datenbank in einer einzigen Datei und benötigt keine Konfiguration oder Verwaltung eines separaten Servers.

Der Grund, weshalb SQLite ausgewählt worden ist, ist, dass die Datenbank plattformunabhängig implementiert werden kann und keine weiteren Abhängigkeiten benötigt. SQLite kann sehr einfach überall eingebunden werden, ohne grosse Konfigurationen und die Daten können direkt in der Datenbankdatei gespeichert werden. Auch Auswertungen der gespeicherten Daten sind unkompliziert, da nicht zuerst ein Datenbankexport vorgenommen werden muss, sondern direkt die Datenbankdatei kopiert werden kann. SQLite ist vor allem im Lesezugriff schnell und kann die in diesem Projekt auftretenden parallelen Schreibzugriffe problemlos bewältigen.

Würden deutlich mehr parallele Zugriffe auftreten, könnte auf PostgreSQL umgestellt werden. Dank der Verwendung eines ORM (SQLAlchemy) im Backend wäre dies ohne grosse Änderungen am Code möglich. Im Frontend wären die Anpassungen grösser, da dort better-sqlite3 eingesetzt wird. Im Frontend finden weniger Datenbankzugriffe statt und diese sind auch weniger komplex und damit weniger ressourcenintensiv, so dass diese Anpassungen sehr viel später notwendig würden.

PostgreSQL wird ausserdem standardmässig von Nominatim verwendet, um die OpenStreetMap-Daten zu speichern.

4.1.4 Reverse-Proxy

Nginx wird als Reverse-Proxy eingesetzt. Der Proxy-Server leitet eingehende Anfragen auf Port 80 resp. 443 weiter auf den standardmässig von Next.js verwendeten Port 3000 weiter. Das SSL-Zertifikat wird ebenfalls von Nginx bereitgestellt und dabei wird auch sichergestellt, dass alle http-Anfragen automatisch auf https umgeleitet werden. Da Nginx auch ein sehr schneller Webserver ist, würde die Möglichkeit bestehen, die statischen Webseiten der Applikation direkt darüber ausliefern zu lassen. Die dynamischen Seiten würden weiterhin vom Next.js-Server bereitgestellt werden. Dies ist momentan nicht implementiert, da der Next.js-Server den Anforderungen auch so gewachsen war. Das Vorschalten von Nginx bietet zusätzliche Sicherheit und könnte bei Bedarf auch gegen DoS-Attacken schützen, indem für die gesamte URL oder Unterseiten, z.B. die Loginseite, rate limiter eingesetzt werden können, die nur eine bestimmte Gesamtanzahl Zugriffe pro Zeiteinheit zulassen.

4.2 Architektur

In diesem Kapitel wird beschrieben, wie die Struktur des Projektes aufgebaut ist und was für Vorteile dies bringt. Dabei wird auf die Trennung zwischen Front- und Backend eingegangen und wie die selbst erstellten Datenbanken strukturiert sind.

4.2.1 Trennung Front- und Backend

Das Ziel ist es, dass das Frontend und Backend unabhängig voneinander sind. Eine der Gründe, der dafür spricht, ist es, dass das Backend unabhängig vom Frontend verwendet werden kann und somit mit einer anderen oder mehreren Frontends kompatibel ist. Das Frontend ist ebenfalls vom Backend unabhängig, jedoch kann keine Runde gespielt werden, solange keine Daten und Informationen von einer Datenbank, wie sie sich im Backend befindet, abgefragt werden, da keine Daten lokal abgespeichert werden sollen.

Ein weiterer Vorteil ist, dass durch die Trennung eine Modularität entsteht, so dass im Falle einer Erweiterung oder sonstigen Modifikation bei einem der beiden Teile, keine Änderungen beim anderen Teil vorgenommen werden müssen, solange die Schnittstellen nicht verändert werden.

4.2.2 Verwendung von Docker

Wie zuvor erwähnt, haben fiel die Wahl auf Docker, weil es schnell und plattformunabhängig implementiert werden kann.

Es wurde entschieden, ein Docker-Netzwerk aufzubauen. Der Grund dafür ist, dass die Container, welche sich im Netzwerk befinden, so einfach über den Namen des jeweiligen Containers ansprechbar sind, anstelle von IP-Adressen oder Domänen-Namen. Dasselbe könnte mit einer einzigen Docker Compose-Datei erreicht werden. Jedoch müsste die Docker Compose-Datei dafür sowohl Front- als auch Backendkomponenten enthalten, was dem Prinzip der Trennung dieser widerspricht. Indem sich alles in einem Netzwerk befindet, können einfach weiter Docker-Container hinzugefügt werden, ohne Änderungen an den bisherigen vorzunehmen.

Docker hat einen grossen Nachteil, dass alle Daten nur auf die virtuelle Maschine kopiert werden und nur mit diesen Daten gearbeitet wird. Das Problem dabei ist, dass diese nicht so einfach abgefragt werden können. Zusätzlich werden grosse Mengen an Daten dupliziert, welches viel Platz benötigen. Aus diesen Gründen werden die Datenbanken und Audiodateien nur gemountet, so dass diese direkt von der physikalischen Maschine übernommen werden, anstelle von der Kopie, die im virtuellen Container liegt.

In Abbildung 4.1, ist ersichtlich, wie das Frontend, das sich in einem Docker-Container befindet und mithilfe von Next.js, Leaflet und Lucia entwickelt worden ist, sich in die Gesamtarchitektur einfügt. Für die Lucia Authentifikation wird noch eine SQLite Datenbank gemountet. Für das Backend wird mithilfe von FastAPI und SQLAlchemy eine REST API erstellt, welche in einem weiteren Docker-Container Zugriff auf die gemountete SQLite-Datenbank hat. Der S3Poxy wird anhand von einem Docker-Image erstellt, dass bereits auf Docker Hub existiert. Dabei werden die Audiodateien gemountet. Nominatim wird genau wie S3Proxy anhand eines Docker-Images implementiert, der Unterschied hierbei ist, dass die PostgreSQL-Datenbank nicht gemountet wird, sondern nur auf der virtuellen Maschine gespeichert wird.

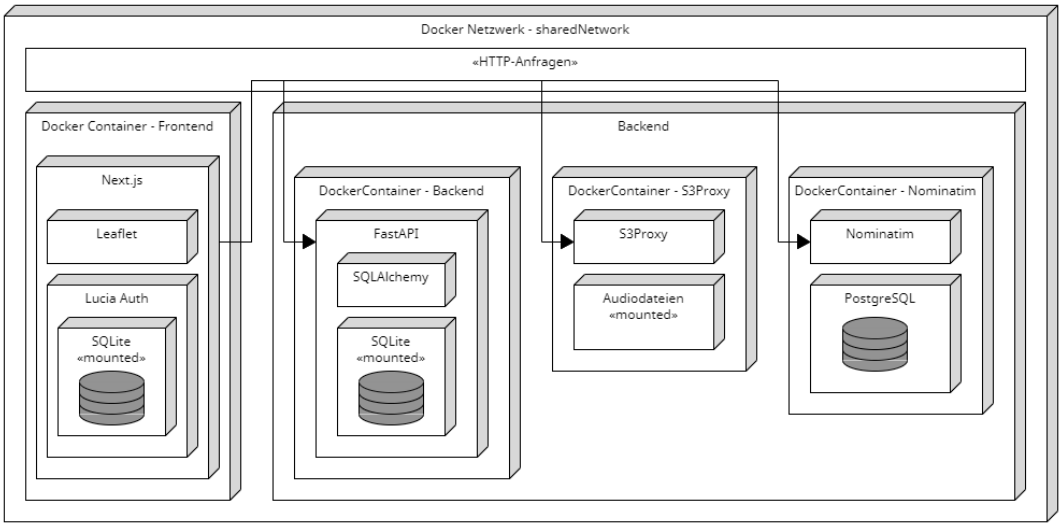

Abbildung 4.1: Architektur des Projektes

Bei diesem Projekt kommuniziert das Frontend nur mit den Backend-Endpunkten. Dafür muss beim Frontend nur http://Container-Name:Interner Port verwendet werden.

4.2.3 Client- und serverseitige API-Anfragen im Frontend

In Next.js-Projekten werden die Seiten meistens serverseitig geladen. Gewisse Komponenten müssen jedoch clientseitig geladen werden, was Schwierigkeiten mit den API-Anfragen mit sich bringt. Der Grund dafür ist, dass die Docker-Container clientseitig nicht angesprochen werden können, da sich das ganze Docker-Netzwerk natürlich nur auf dem Server befindet. Wird die API-Abfrage nun auf dem Gerät des Nutzers selbst versucht, findet dieses den Container nicht und verursacht den Fehler *InternalServerError (500)*.

Für diese API-Anfragen, die clientseitig getätigt werden, muss eine interne API-Abfrage erstellt werden, welche als Proxy fungiert. Diese Funktion wird serverseitig erstellt, damit sie mit den anderen Container im Docker-Netzwerk kommunizieren kann.

4.2.4 Datenbank

In diesem Kapitel wird erklärt, wie die Tabellen der selbst erstellten Datenbanken strukturiert sind und welche Informationen abgespeichert werden.

Lucia Datenbank

Es wurde entschieden, dass die Daten für das Login, wegen der sensiblen Daten, die dort gespeichert werden, in einer separaten Datenbank abgespeichert werden. Der Grund dafür liegt, bei der FastAPI nur gewisse Informationen gebraucht werden und E-Mail, Passwörter und Session-Informationen nicht benötigt werden. Die Tabellen von Abbildung 4.2 werden im Detail beschrieben.

Bei der Tabelle *user*, werden die Benutzerinformationen abgespeichert. Für die Authentifizierung der Benutzerdaten wird nur der Benutzername und das Passwort benötigt. Dabei ist zu beachten, dass das Passwort mit Argon2 gehasht wird. Ebenfalls benötigt wird die E-Mail-Adresse für das Zurücksetzen des Passwortes. Die restlichen gesammelten Informationen werden für die spätere Auswertung der Daten gespeichert. Zuerst die Postleitzahl, die aussagt, welche Ortschaft den Dialekt am meisten geprägt hat. Als zweite Angabe wird die Altersgruppe definiert, welcher die Person angehört. Drittens folgt eine ebenfalls freiwillige Angabe des Geschlechts und zuletzt kann eine Selbsteinschätzung, mit einem Wert zwischen 1 und 5, die eigene Dialekterkennungsfähigkeit von Schweizer Dialekten angegeben werden.

In der nächsten Tabelle *session* wird eine Session-ID erstellt, welche ein auch ein Ablaufdatum enthält. Hier wird die Benutzer-ID als Fremdschlüssel verwendet. Für den Fall, dass die Session noch nicht abgelaufen ist, wird der Benutzer direkt eingeloggt. Anderenfalls wird der Benutzer zur Loginseite weitergeleitet.

Bei der letzten Tabelle *password-reset-token* wird ein temporärer Hash-Token erstellt, welcher für das Ändern des Passwortes verwendet wird. Dieser Token wird an die E-Mail gesendet, die in den Benutzerinformationen abgespeichert ist. Dieser Token läuft nach einer Stunde ab.

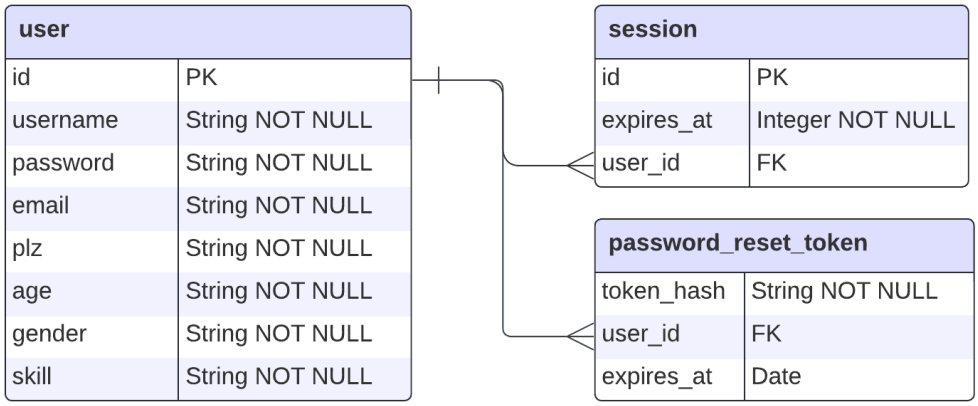

Abbildung 4.2: Lucia Datenbankmodell

FastAPI-Datenbankmodell

Hier handelt es sich um die Daten, die gesammelt werden oder vom STT4SG-350 Korpus importiert werden. Die Tabellen von Abbildung 4.3 werden hier im Detail beschrieben.

Bei der Tabelle *Utterance* handelt es sich um Informationen, welche aus dem Korpus STT4SG-350 stammen. Diese Daten liegen in Form einer TSV-Datei vor, von welcher die nötigen Informationen zu den Audioaufnahmen in die eigene Tabelle importiert werden. Es ist zu beachten, dass die Audiodateien in einem Ordner gespeichert sind, welcher mit der ID des Sprechers (*clientID*) gekennzeichnet ist und sich darin wiederum mehrere Audioaufnahmen befinden, welche mit der ID des Satzes gekennzeichnet sind.
Bei *clientID* handelt es sich um die ID des Sprechers, die vom Korpus importiert wird.
Bei *clipPath* handelt es sich um den relativen Pfad, unter welchem die Datei abgespeichert ist. Ebenfalls importiert vom Korpus.
Bei *sentence* und *sentence-source* handelt es sich um den Satz, welcher in der Audiodatei gesprochen wird, sowie aus welcher Quelle dieser stammt. Diese Informationen wurden ebenfalls aus dem Korpus importiert.
Bei *age*, *gender* und *zipcode* handelt es sich um die Informationen des Sprechers der Audioaufnahme. Die Postleitzahl gibt an, woher der gesprochene Dialekt stammt. Diese Daten wurden ebenfalls importiert.
Bei *town*, *canton* und *coord-lat/lng* handelt es sich um Informationen, welche anhand der Postleitzahl mithilfe einer Tabelle von Swisstopo hinzugefügt worden sind. Es sind die Namen der Ortschaft und des dazugehörigen Kantons und die Koordinaten dieser Ortschaft.
Zuletzt ist *reported* ein Counter, welcher angibt, wie oft eine Audioaufnahme aufgrund der Qualität gemeldet worden ist.

Bei der Tabelle *Guess* handelt es sich um den Ort, der für die Dialektherkunft geraten wurde.
Die ID des Sprechers, welche aus *Utterance* übernommen wurde, wird in *clientID* gespeichert.

Bei *plz-guess*, *canton* und *town* handelt es sich um die Postleitzahl, Kanton und Ortschaftsname, der geratenen Ortschaft.
In *coordinatesX* und *coordinatesY* werden die Koordinaten in Längen- und Breitengrad abgespeichert, welche geraten worden sind.
Bei *distance* handelt es sich um die Luftdistanz zwischen dem geratenen Punkt und dem effektiven Herkunftsort in Kilometern.
Bei *createdAt* handelt es sich um das Datum, an dem geraten worden ist.
Bei *username* und *plz-guesser* handelt es sich um den Benutzernamen und die Postleitzahl, welche von den Benutzerinfomationen übernommen worden sind.
corpus gibt an, aus welchem Korpus die Quelle stammt. Diese Spalte wurde hinzugefügt, weil anfänglich geplant wurde, mehr als nur einen Korpus zu verwenden.

Die Tabelle *GuessUtterances* ist ein Knotenpunkt zwischen Utterance und Guess. Bei jedem Raten werden fünf Audioaufnahmen zur Verfügung gestellt und müssen auch so gespeichert werden. Somit werden fünf Einträge erstellt. Die ID der Guess-Tabelle ist bei allen fünf immer die gleiche und bei der Utterance-Tabelle immer eine andere. '*listenedTo*' zeigt an, ob die zur Verfügung gestellte Audioaufnahme angehört worden ist oder nicht.

Die Tabelle *UserInfo* enthält die Informationen, welche bei der Registrierung abgespeichert wurden. Für mehr Informationen siehe Tabelle *user* der Lucia Datenbank.

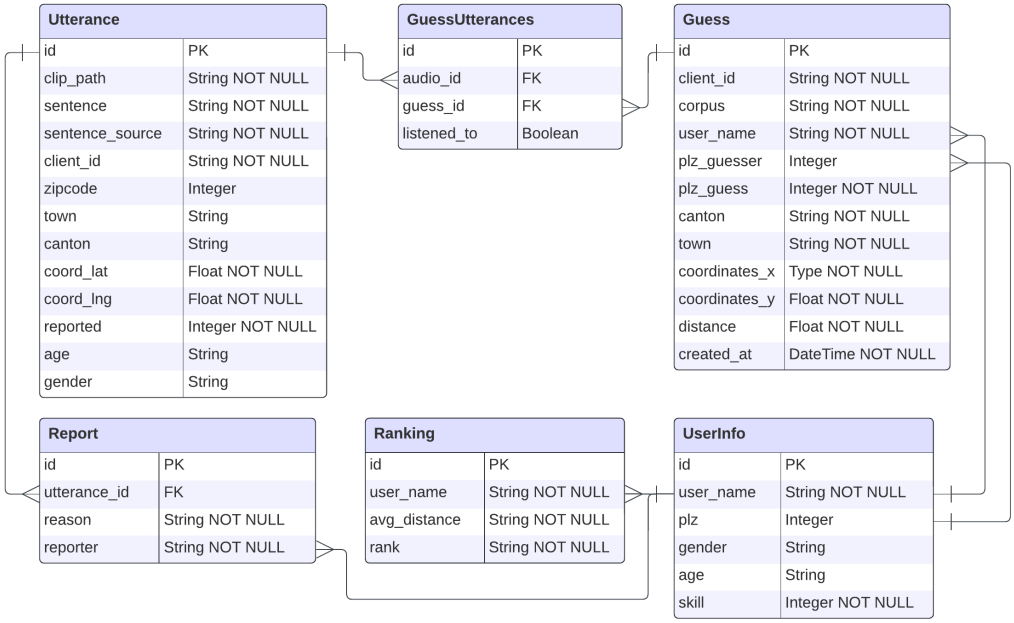

Abbildung 4.3: FastAPI-Datenbankmodell

Bei *Ranking* handelt es sich um die Tabelle, in welcher der Rang des Benutzers aufgelistet wird. Anhand des Benutzernamen werden allen Einträgen in der Tabelle *Guess* genommen und damit die eigene durchschnittliche Distanz berechnet. Danach wird die gesamte Rangliste aktualisiert. Dies geschieht nach jedem Versuch eines Benutzers.
Hat ein Nutzer weniger als zehn Versuche gespielt, wird ihm kein Rang zugeteilt, sondern er wird als 'unranked' gekennzeichnet.

In der Tabelle *Report* wird der Grund für die Meldung gespeichert. Dafür wird die ID der Audiodatei angegeben und noch eine kurze Beschreibung hinzugefügt. Zusätzlich wird noch festgehalten, wer diese Meldung abgesetzt hat.

4.3 Implementation

In diesem Kapitel wird erklärt, wie ein typischer Ablauf eines Versuches aussieht, wie die Funktionen implementiert worden sind und wie gewisse davon optimiert worden sind.

4.3.1 Ablauf eines Versuches

Der Kern des Frontends besteht darin, dass schweizerdeutsche Sätze auf einem Audio-Player abgespielt werden können und auf einer interaktiven Karte ein Ort geraten werden kann, um den Dialekt zu schätzen, woher dieser stammt.

Wie im Sequenzdiagramm (siehe Abbildung 4.4) dargestellt, wird hier ein einfacher Ablauf eines Versuches beschrieben.

Als erstes, wenn die Seite neu geladen wird oder ein neuer Versuch durchgeführt wird, werden fünf Datensätze eines Sprechers von der SQLite-Datenbank des Backendes abgefragt, mittels *http://backend:8000/api/utterance/same_client/5*.

Als zweites, anhand von diesen Daten, wird der Positionsmarker für die Lösung aktualisiert. Ebenfalls anhand dieser Daten wird der relative Pfad der Audio-Datei angegeben, wo diese im S3-Bucket gespeichert ist. Mit *http://s3proxy:80/audio-bucket/{relative_path}* wird dieses aus dem S3-Bucket direkt im Audio-Player gestreamt.

Als drittes, muss der Spieler sich die Aufnahme anhören und sich Gedanken machen, woher dieser Dialekt kommt. Wenn die Entscheidung auf einen Dialekt gefallen ist, kann der Positionsmarker des geratenen Dialektes verschoben werden. Dabei werden mit den Koordinaten des Markers mittels *http://nominatim:8080/reverse?lat={lat} &lon={lng}&zoom=13* die Ortschafts-Informationen (PLZ, Gemeinde und Kanton) abgefragt. Der Zoom bei dieser Abfrage limitiert die Abfrage (siehe Kapitel 4.3.3). Anhand von diesen Informationen werden die Daten vom Marker aktualisiert.

Als letztes bestätigt der Spieler den Versuch, sobald dieser mit der Auswahl zufrieden ist. Dabei wird die Luftdistanz zwischen den beiden Marker ausgerechnet und zusammen mit den Daten von beiden Marker, inklusive den Benutzerinformationen mittels der POST-Methode, auf die SQLite-Datenbank vom Backend gesendet: *http://backend:*
8000/api/add_guess. Sobald diese Informationen auf dem Backend angekommen sind, wird die Rangliste aktualisiert. Dabei werden alle Versuche vom Spieler genommen und die durchschnittliche Distanz ausgerechnet. Danach wird die Rangliste anhand der neuen Distanz aktualisiert.

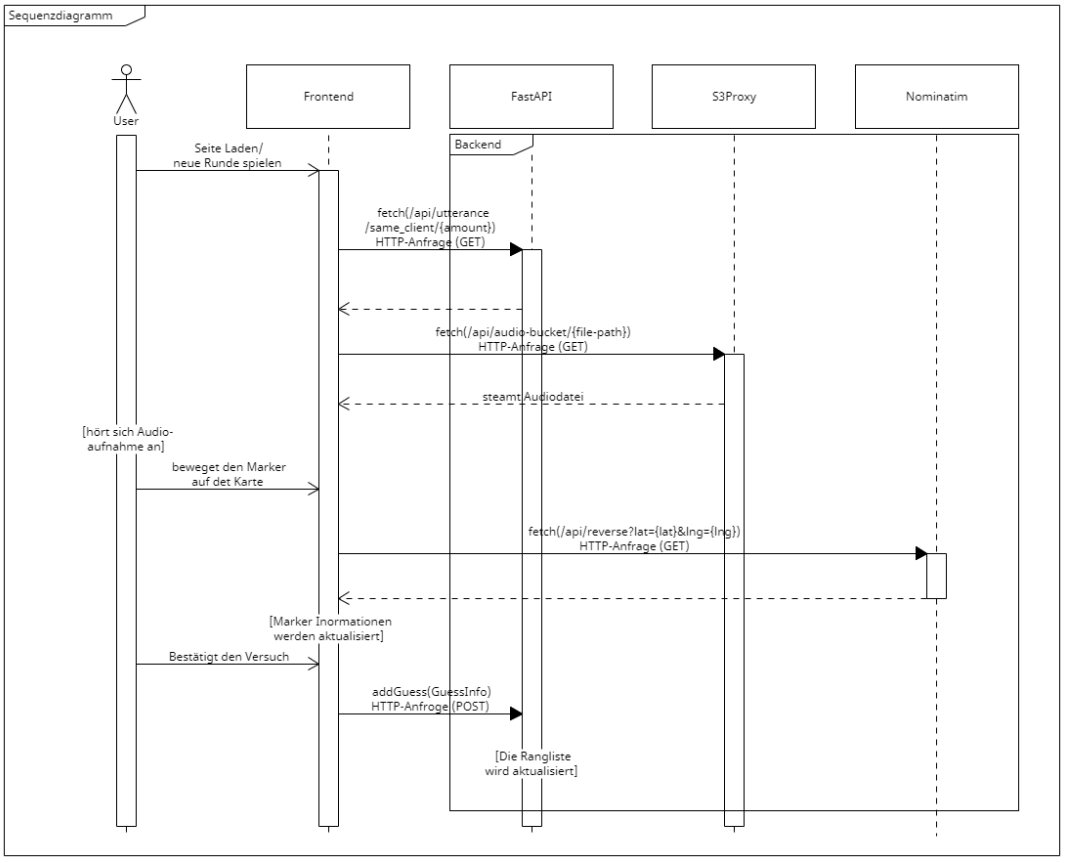

Abbildung 4.4: Ablauf eines Versuches, anhand eines Sequenzdiagramms

4.3.2 Implementation der Features

Nachfolgend werden die Features und deren Umsetzung genauer erläutert.

Karte

Für die Anzeige der Schweizer Landkarte wurde das kostenlose Open-Source-Tool Leaflet verwendet, und für die Karte werden VectorTiles von Swisstopo genutzt. Leaflet ermöglicht es, Positionsmarker, Informationen in Pop-ups und Formen wie Polygone über die Layer der Karte zu legen. Die Karte wird in der Komponente *./components/Map/Map.tsx* initialisiert. Leaflet lässt sich in Next.js sehr einfach integrieren, indem der *MapContainer* im JSX eingebunden wird. Innerhalb dieses Containers muss die *VectorTileLayer*-Karte eingebunden werden, um die Karte darzustellen, sowie zwei Positionsmarker. Der erste Marker dient dazu, den Dialekt zu erraten, während der zweite Marker die Lösung anzeigen soll.

Swisstopo wurde ausgewählt, da diese Daten vom Schweizerischen Bundesamt für Landestopografie stammen. Da es sich bei diesem Projekt, um eines mit im Vergleich wenigen Nutzern handelt, können die Nutzungsbedingungen einfach eingehalten werden. «Die Einbindung der Geodienste in Webapplikationen mit im Schnitt 10'000 Nutzern pro Tag oder Desktopanwendungen entspricht einer Fair-Use-Nutzung.»[29]

Wie in Abbildung 4.5 dargestellt, war ursprünglich geplant, eine Bitmap von Swisstopo zu verwenden, jedoch gab es mehrere Mängel. Erstens konnten die Kantonsgrenzen nicht gut dargestellt werden. Zusätzlich wurden viele grössere Städte nicht angezeigt, sodass die Standorte auf der Karte nicht gut ersichtlich waren.
Mit der *VectorTilesLayer* musste nur eine *styles.json*-Datei hinzugefügt werden, die die Layer-Informationen enthält und eine bessere Darstellung ermöglicht. Wie diese Karte dargestellt wird in Abbildung 4.6 dargestellt.

Abbildung 4.5: Einbindung Karte von Swisstopo mit *Bitmap*

Abbildung 4.6: Einbindung Karte von Swisstopo mit *Vector Tiles*

Der Marker für den Versuch kann einfach auf der Karte verschoben werden. Dabei werden beim Backend die Reverse-Geocoding-Daten über *http://nominatim:8080/reverse* mit Längen- und Breitengrad abgerufen, was die Ortschaftsinformationen zurückgibt. Sollten ausnahmsweise keine Informationen dieses Standortes zur Verfügung stehen, wird der Marker zu seinem letzten gültigen Standort zurückversetzt. Dasselbe gilt, wenn er ausserhalb des Polygons der Schweizer Grenze gesetzt wird. Dieses Polygon wird mittels einer GeoJSON-Datei vorgegeben.

Die Daten beider Marker werden bei *./components/Map/Map.tsx* unter den *useState* *markerData* und *markerTargetData* gespeichert. Diese enthalten die Koordinaten und die Informationen des Standortes (PLZ, Ortsname und Kanton).

Suchfeld

Das schriftliche Anzeigefeld für die Vermutung der Dialektherkunft wurde als normales Suchfeld implementiert. Wird eine Postleitzahl oder eine Gemeinde angegeben, werden mehrere Vorschläge in einem Dropdown-Menü angegeben (siehe Abbildung 4.7), welche ausgewählt werden können. Sobald eine dieser Standorte ausgewählt worden ist, werden die Informationen des Markers *markerDate* in *./Map/Map.tsx* aktualisiert und der Marker selbst wird an den Standort der ausgewählten Ortschaft verschoben.
Die Informationen, welche im Dropdown-Menü erscheinen, werden aus einer JSON-Datei geladen, die sich unter *./lib* befindet. Wird keine der vorgegebenen, maximal fünf, Ortschaften aus dem Dropdown-Menü ausgewählt, ist die Eingabe ungültig und die Position des Markers ändert sich nicht.
Wird der Marker auf der Karte verschoben, so wird im Suchfeld automatisch der neue Standort angegeben. Zur Start eines neuen Versuches wird der Marker auf die Ortschaft Rütli (siehe Kapitel 3.5) verschoben und im Suchfeld wird Rütli angezeigt.

Abbildung 4.7: Das Suchfeld mit einem Dropdown-Menü

Lösung

Wird die Seite neu geladen oder ein neuer Versuch gestartet wird, werden die Daten der Dialektherkunft aus der Datenbank abgerufen, mit *http://backend:8000/api/utterance/same_client/5*. Die *markerTargetData* werden anhand von diesen Daten aktualisiert.

Wenn ein Versuch durchgeführt wird, erscheint eine weiterer Positionsmarker auf der Karte (siehe Abbildung 4.9). Währenddessen werden die Informationen beider Marker genommen, die in *./components/Map/Map.tsx* gespeichert werden. Anhand von diesen Koordinaten wird die Distanz in Kilometer mit der Funktion *calculateDistance()* berechnet. Bei *Map.tsx* wird das Eingabefeld ausgeblendet und die Lösung eingeblendet, wenn ein Versuch bestätigt worden ist. Die Lösung (siehe Abbildung 4.8) gibt an, was die Antwort des Versuches war und was die effektive Antwort ist. Die Luftdistanz zwischen diesen beiden Punkten wird dort in Kilometer ebenfalls noch angezeigt.

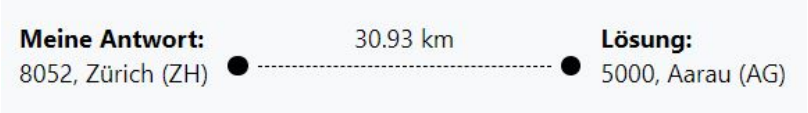

Abbildung 4.8: Darstellung der Lösung als Text

Abbildung 4.9: Darstellung der Lösung auf der Karte

Audio-Player

Beim Audio-Player handelt es sich um eine Komponente bei *./components/Audioplayer/Audioplayer.tsx*, welcher unter der Karte bei *Map.tsx* hinzugefügt wird (siehe Abbildung 4.10.

Wird die Seite neu geladen oder ein neuer Versuch gestartet wird, werden die Daten der Dialektherkunft aus der Datenbank abgerufen, mit *http://backend:8000/api/utterance/same_client/5*. Diese Liste wird als Parameter weitergegeben zu *Audioplayer.tsx*, da diese die Informationen enthalten, wo im S3-Bucket der Audio-Clip, relativ abgespeichert ist. Der relative Pfad enthält die *client_id* und die *sentence_id*, welche vom S3-Bucket mit http://s3proxy:80/audio-bucket/{cliend_id}/{sentence_id} abgerufen wird.
Ebenfalls aus den Daten des Parameters entnommen, wird der Satz, der gesprochen wird, auch schriftlich angegeben, als visuelle Unterstützung für den gesprochenen Satz. Da es sich bei dem Parameter um eine Liste handelt, werden diese Informationen mit einem Index gelesen. Wenn ein anderer Satz des gleichen Sprechers gespielt werden soll, wird mit *playNext()* der Index auf den nächsten Eintrag der Liste gelegt, welcher erneut die Daten aus dem S3-Bucket abruft.
Um zu sehen, ob die Audioaufnahmen angehört worden sind, wird bei der Bestätigung des Versuches eine Liste von Indexen zurückgegeben, welche angibt, bei welchen Indexen der Clip abgespielt worden ist.

Abbildung 4.10: Der Audio-Player

Versuch zur Datenbank hinzufügen

Wird ein Versuch bestätigt, werden die Informationen der beiden Marker bei *markerData* und *markerTargetDate* genommen und die Luftdistanz dieser beiden Punkte wird ausgerechnet (siehe Kapitel 4.3.2 Lösung). Diese Daten werden dann in ein JSON-Format gebracht, anhand der *Guess*-Tabelle bei Kapitel 4.2.4. Dieses wird mit einer POST-Methode mittels *http://backend:8000/api/add_guess* auf der SQLite Datenbank gespeichert.
Dabei wird die ID des Versuches zurückgegeben, damit zusätzlich noch die IDs der Daten des Sprechers mit der ID des Versuches verknüpft werden kann. Dafür wird eine weitere POST-Methode ausgeführt: *http://backend:8000/api/guess/utterance*. Diese Methode wird für jede ID des Sprechers durchgeführt, wo referenziert wird, um welche ID es sich beim Versuch handelt und welche Clips angehört worden sind (siehe Kapitel 4.3.2 Audio-Player).

Rangliste

Bei der Rangliste handelt es sich um eine Backend-Funktion, die im Hintergrund nach jedem Versuch ausgeführt wird. Dabei werden alle Einträge von den Versuchen eines Spielers genommen und aus diesen wird die durchschnittliche Distanz berechnet. Danach werden alle Einträge der Rangliste nach der Distanz sortiert und der Rang wird anhand des Indexes zugeteilt. Dabei ist zu beachten, dass falls jemand weniger als zehn Versuche gespielt hat, sein Rang als *unranked* eingestuft wird, da er sich nicht für die Rangliste qualifiziert. Für eine genauere Beschreibung, siehe Kapitel 3.2.
Bei der Rangliste beim Frontend wird nur die Liste von der *Ranking*-Tabelle, mittels *http://backend:8000/api/ranking/all*, abgerufen. Die Einträge ohne Rang werden aus der Darstellung entfernt.

Authentifizierung

Für die Authentifizierung wurde Lucia Auth verwendet, wobei jeweils beim Einloggen anhand der ID des Benutzernamens eine Session erstellt wird. In *db.ts* wird die Verbindung zur Lucia-Datenbank erstellt und das Modell der Datenbank (siehe Kapitel 4.2.4) definiert und in *auth.ts* wird die Session erstellt, verwaltet und validiert.
Unter jeglichen *page.tsx*-Dateien der App wird eine Validierung hinzugefügt, dass bei bestehender Session kein Einloggen mehr notwendig ist. Sollte diese Session jedoch abgelaufen sein oder noch keine existieren, wird automatisch auf die Login-Seite weitergeleitet.

4.3.3 Leistungsoptimierung

Während dem Testen hatten gewisse HTTP-Abfragen eine längere Ladezeit, was den Spielablauf verzögerte und ein schlechtes Spielerlebnis mit sich brachte. Daraufhin wurden diese Abfragen lokalisiert und so optimiert, dass der Spielablauf flüssiger lief.

FastAPI

Die API-Abfragen, die mithilfe von FastAPI und SQLAlchemy erstellt wurden, hatten zunächst einige Perfomanceschwierigkeiten, was zu längeren Ladezeiten führte.

Damit in der Tabelle *Utterance* zufällige Informationen zu den Audioaufnahmen eines Sprechers zurückgegeben werden können, musste zunächst nach dem Sprecher gefiltert und anschliessend fünf zufällige Einträge zurückgegeben werden.
Um mit SQLAlchemy eine zufällige Antwort zu erhalten, muss die Funktion *order_by(func.random())* verwendet werden, was zu einer Laufzeit von $O(n \cdot log\ n)$ ergibt. Bei vielen Einträgen in der Datenbank dauert diese Funktion zu lange.
Die Abfrage wurde so optimiert, dass zunächst alle IDs eines Sprechers abgefragt und diese dann mit der Funktion *random.sample(population, k)* zufällig ausgewählt werden. k steht für die Anzahl der zurückzugebenden IDs und *population* für die ursprüngliche Liste der IDs. Danach wird die Liste nur nach den IDs gefiltert, was effektiv zu einer Laufzeit von $O(3 \cdot n)$ führt.

Grundsätzlich wurden alle GET-Methoden pro Tabelle so entwickelt, dass sie nur eine Laufzeit von $O(x \cdot n)$ haben, wobei x maximal drei betragen darf. Das führt dazu, dass die Daten schnell beim Frontend ankommen.
Für die POST-Methoden wird normalerweise eine Laufzeit von $O(1)$ bei SQLAlchemy benötigt. Allerdings müssen mehrere Methoden Werte aus anderen Tabellen verwenden, um bestimmte Einträge zu erstellen.
Ein Beispiel ist die Rangliste, bei der zunächst alle geratenen Versuche aus der Tabelle *Guess* anhand des Benutzernamens gefiltert und der Durchschnitt der Luftdistanz berechnet wird. Danach muss die gesamte Tabelle *Ranking* überarbeitet werden, indem sie nach der Distanz sortiert und anhand des Indexes der Rang zugewiesen wird. Somit hat die Aktualisierung der Rangliste eine Laufzeit von grösser $O(n \cdot log\ n)$. Diese Funktionen werden jedoch immer als Hintergrundfunktionen des Servers ausgeführt, sodass sie keine direkten Auswirkungen auf das Spiel haben.

Nominatim

Bei Nominatim wurde festgestellt, dass die Reverse-Geocoding-Abfrage mittels Längen- und Breitengrad länger dauert als erwartet. Somit dauert es immer ein paar Sekunden, bis der aktuelle Standort des Markers angegeben wird. Dies war eine der grössten Belastungen auf dem Server. Der Grund dafür ist, dass die Informationen bei Nominatim auf mehreren Ebenen gespeichert und abgefragt werden können. Je tiefer die Ebene, desto mehr Daten müssen abgefragt werden. Auf den ersten Ebenen handelt es sich nur um allgemeine Informationen, wie das Land und die Kantone. Auf den tiefsten Ebenen geht es werden Strassen und Hausnummern gespeichert.

Für das Projekt ist die kleinste Ebene, die benötigt wird, die der Gemeinden. Daher konnte bei der API-Abfrage *http://nominatim:8080/reverse?lat={lat}&lon={lng}* eine Begrenzung der Ebenen hinzugefügt werden: *&zoom=13*, welches nur die Ortschaften auf Gemeindeebene ausgibt. Die Ebenen über 13 machen mehr als zwei Drittel der Datenbank aus und werden aufgrund der Begrenzung nicht mehr abgefragt.

4.4 Deployment

In diesem Kapitel wird Schritt für Schritt beschrieben, wie das Projekt in die Live-Umgebung eingebunden wird (siehe Abbildung 4.11).

Die Konfigurationen werden anhand eines Ubuntu 22 Betriebssystems dargestellt.

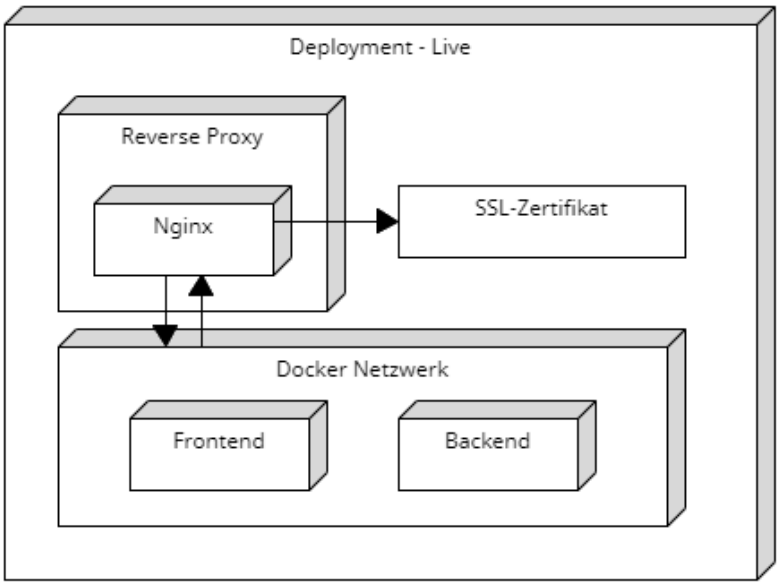

Abbildung 4.11: Struktur auf dem Server nach dem Deployment

4.4.1 Vorbereitung

Zuerst müssen die beiden Verzeichnisse vom Front- und Backend auf dem Server geladen werden. Dafür sollten diese am besten von einem GitHub-Verzeichnis heruntergeladen werden.

Es ist zu anzumerken, dass gewisse Dateien, wegen ihrer sensiblen Natur oder grossem Speicherplatzbedarf, nicht im GitHub-Verzeichnis gespeichert sind. Diese Dateien müssen noch separat hinzugefügt werden.

Beim Backend müssen alle Audiodateien, die verwendet werden sollen, über FTP auf den Server geladen werden. Dabei ist zu beachten, dass diese im richtigen Verzeichnis gespeichert werden. Diese müssen unter *backend-game/audio/clips* gespeichert werden. Dabei müssen die Audiodateien unter in den Ordner, welcher nach der ID des Sprechers benannt worden ist.

Zusätzlich muss noch die SQLite Datenbank-Datei textitbackend-game/db_api hinzugefügt werden.

```
.
└── backend-game
    ├── docker-compose.yml
    ├── db_api
    |   ├── ...
    |   ├── backend.db
    |   └── Dockerfile
    └── audio
        └── clips
            ├── ...
            └── [client_id]
                ├── ...
                └── [sentence_id].[mp3|flac]
```

Beim Frontend fehlt ebenfalls die SQLite Datenbank-Datei. Ebenfalls fehlt noch eine lokale *.env.local*-Datei, welche sensible Daten enthält, wie die SMTP Konfigurationen und Passwörter vom externen E-Mail-Dienst. Diese beiden Dateien müssen im gleichen Verzeichnis hinzugefügt werden, wie die Docker Compose-Datei.

```
.
└── dialect-guesser
    ├── docker-compose.yml
    ├── Dockerfile
    ├── main.db
    ├── .env.local
    ├── ...
    └── app
        ├── ...
        └── page.tsx
```

4.4.2 Docker Netzwerk

Bevor die Docker Container gestartet werden können, muss noch ein Docker Netzwerk auf dem Gerät erstellt werden. Der Name des Netzwerkes muss definiert werden. In diesem Projekt wurde das Netzwerk *sharedNetwork* benannt. Zum Erstellen des Netzwerkes muss der folgende Befehl eingegeben werden:

```
1 sudo docker network create NETWORK_NANME
```

Sobald das Netzwerk erstellt worden ist, muss zum Verzeichnis des Frontends navigiert werden. Nun wird folgender Befehl ausgeführt, um das Frontend zu starten:

```
1 sudo docker-compose up --build -d
```

Als Nächstes wird zum Verzeichnis des Backends navigiert. Hier wird folgender Befehl ausgeführt werden, um das Backend zu starten:

```
1 sudo docker-compose up --build -d
```

–build muss nur beim ersten Mal oder bei Änderungen ausgeführt werden, damit das Frontend neu gebuildet wird.
-d wird mitgegeben, damit der Docker Container als Hintergrundprozess läuft.

Wenn das Backend zum ersten Mal gebuildet wird, kann dies eine gewisse Zeit (∼30 Minuten) in Anspruch nehmen, da Nominatim noch alle Daten von OpenStreetMap in die neu erstellte PostgreSQL-Datenbank importieren muss.

Ob die Docker Container laufen, kann mit folgendem Befehl kontrolliert werden:

```
sudo docker ps
```

Falls sich noch keine Daten in der Tabelle *Utterance* befinden, kann mittels *http://backend:8000/api/utterance/import_tsv* die *tsv*-Datei vom STT4SG-350 Korpus importiert werden. Dabei handelt es sich um rund 200'000 Datensätze, was eine gewisse Zeit(∼20 Minuten) in Anspruch nimmt.

4.4.3 Reverse Proxy

Für den Reverse Proxy und das SSL-Zertifikat werden *Nginx* und *Certbot* verwendet. Zuerst müssen die beiden Tools mit den folgenden Befehlen importiert werden:

```
sudo apt install nginx

sudo snap install --classic certbot
sudo ln -s /snap/bin/certbot /usr/bin/certbot
```

Der letzte Befehl ist wichtig, dass der *certbot* Befehl ausgeführt werden kann.

Der folgende Befehl muss ausgeführt werden, dass die Zertifikate erstellt werden:

```
sudo certbot certonly --nginx -d DOMAIN_NAME -d www.
   DOMAIN_NAME
```

certonly steht dafür, dass nur die Zertifikate erstellt werden und diese nicht selbstständig in Nginx eingebunden werden.
–nginx steht dafür, dass die Zertifikate nach Nginx-Format erstellte werden sollen.
-d DOMAIN_NAME gibt an, für welche Domäne das Zertifikat erstellt wird. *DOMAIN_NAME* muss durch den richtigen Domänennamen ersetzt werden.

Die Zertifikate werden unter */etc/letsencrypt* abgespeichert.

Die Konfigurationen von Nginx wird bei *etc/nginx/sites-available/default* vorgenommen. Hier muss die Umleitung von der Domäne auf das lokal laufende Projekt konfiguriert werden.
Dafür müssen zwei Ports konfiguriert werden. Der Port 80, über den die HTTP-Anfragen laufen und den Port 443 für die HTTPS-Anfragen.
Beim Port 443 muss der *proxy_pass* definiert werden, welcher über die IP des Docker Netzwerks den Frontend-Container anspricht. Zusätzlich müssen noch die erstellten Dateien vom Certbot angegeben werden.
Beim Port 80 wird eine Umleitung angegeben, so dass im Falle eines Aufrufs der Domäne per HTTP, automatisch auf HTTPS weitergeleitet wird.

Am Ende sollte die Konfigurationsdatei[1] von Nginx wie folgt aussehen:

[1] Basiert auf: https://gist.github.com/iam-hussain/2ecdb934a7362e979e3aa5a92b181153

```
# etc/nginx/sites-available/default

# HTTPS (Port 443)
server {
    server_name www.DOMAIN_NAME DOMAIN_NAME;

    location / {
        # reverse proxy for next server
        # port 3000 is frontend
        proxy_pass http://DOCKER_NETWORK_IP:3000;

        # reverse proxy settings
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    listen [::]:443 ssl ipv6only=on;
    listen 443 ssl default_server;

    # Certbot erstellte Dateien.
    ssl_certificate /etc/letsencrypt/live/DOMAIN_NAME/
   fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/DOMAIN_NAME/
   privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
}

# HTTP (Port 80)
server {
    listen 80;
    listen [::]:80;
    server_name www.DOMAIN_NAME DOMAIN_NAME;
    return 301 https://$host$request_uri;
}
```

Mit dem folgenden Befehl wird Nginx auf dem Server gestartet und ist dann über die Domäne erreichbar.

```
sudo nginx -t
sudo systemctl restart nginx
```

Kapitel 5

Anhang

5.1 Verwendung von generativer KI

Als Programmierhilfe (Code generieren, debuggen) kam OpenAIs ChatGPT 4.0 bzw 4o zum Einsatz. Ebenfalls wurde es für die Verbesserung oder Generierung von einzelnen Sätzen oder Textabschnitten verwendet.

5.2 Installation der Funktionen

In diesem Kapitel wird erläutert, wie die wichtigsten Komponenten installiert wurden. Zunächst wird beschrieben, wie die REST-API im Backend entwickelt oder eingebunden wurde. Anschliessend wird erklärt, wie das Frontend eingerichtet wurde, einschliesslich der wichtigsten Funktionalitäten dieses.

5.2.1 Backend

Es wird Schritt für Schritt erklärt, wie die verschiedenen APIs implementiert und konfiguriert wurden. Zuerst wird die eigene REST-API mit FastAPI und SQLAlchemy beschrieben. Danach wird erläutert, wie die beiden Docker-Container in Docker Compose eingebunden wurden.

REST API

Anforderungen:

Bei der REST-API handelt es sich um ein Python-Skript, welches die API-Schnittstelle aufbaut. Dafür muss zuvor Python auf dem Gerät installiert sein. Zusätzlich müssen die folgenden Python-Pakete installiert werden: fastapi, gunicorn, uvicorn, sqlalchemy und matplotlib.

```
pip install fastapi
pip install gunicorn
pip install uvicorn
pip install sqlalchemy
pip install matplotlib
```

FastAPI

FastAPI ist sehr schnell einzurichten. Zuerst muss eine main.py-Datei erstellt und mit dem folgenden Code gefüllt werden. Dabei ist zu beachten, dass die CORS-Einstellungen für den Port des Frontends und auf sich selbst angepasst werden müssen.

```
from fastapi import FastAPI

app = FastAPI()

origins = [
    "http://localhost",
    "http://localhost:3000"
]
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE"],
    allow_headers=["*"]
)
```

Somit ist FastAPI erfolgreich eingerichtet worden. Nun können API-Methoden sehr einfach in der main.py-Datei hinzugefügt werden. Es muss nur angegeben werden um welche HTTP-Methode es sich handelt und welches JSON-Ojekt zurückgegeben werden soll. Folgend ein Beispiel aus der FastAPI- Dokumentation:

```
#HTTP-Methode (get, post, put oder delete)
@app.get("/")
async def root():
    #Das JSON-Objekt das zurueckgegeben wird.
    return {"message": "Hello World"}
```

Sobald die ersten API-Methoden erstellt worden sind, kann mit Uvicorn die main.py-Datei gestartet werden. Dafür muss im CLI auf dasselbe Verzeichnis gehen, wo die main.py-Datei sich befindet und folgenden Befehl ausführen.

```
uvicorn main:app --reload
```

Dieser Befehl wird während der Entwicklung des Projektes verwendet. Der Grund dafür ist es, dass jegliche Erfolge und Fehlermeldungen in der Konsole ausgegeben werden. Ebenfalls wird *–reload* verwendet, um bei jeglichen Änderungen im Code die API neu zu starten.

Sollte der Befehl erfolgreich ausgeführt worden sein, kann die API über http://127.0.0.1:8000 erreicht werden. Für eine genauere Dokumentation der API können mittels Aufruf von http://127.0.0.1:8000/docs alle API-Methoden angezeigt werden.

Sobald der Code auf einem produktiven Server eingesetzt wird, sollte ein anderer Befehl ausgeführt werden:

```
gunicorn main:app -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000 --timeout 120
```

Mit *-w 4* wird definiert, wie viele Anfragen zur gleichen Zeit bearbeitet werden können. *-w uvicorn.workers.UvicornWorker* definiert, welche Arbeiterklasse verwendet werden soll. *–bind 0.0.0.0:8000* definiert, dass die API über die IP und den Port 8000 des Servers angesprochen werden kann. Der letzte Option *–timeout 120* definiert, dass jegliche Prozesse, die nach 120 Sekunden nicht abgeschlossen sind, terminiert werden.

SQLAlchemy

Für SQLAlchemy müssen noch drei weitere Dateien erstellt werden. Zuerst muss eine Verbindung zur Datenbank aufgebaut werden mit der *dateabase.py*-Datei. Zusätzlich müssen noch die Modelle und Schemas für die Datenbank definiert werden in *models.py* und *schemas.py*.

Für die Datenbank wird SQLite verwendet. Sollte eine Datenbank mit vorherigen Daten verfügbar sein, sollte diese ebenfalls noch im Verzeichnis eingefügt werden. Sonst wird vom Skript selbst eine neue Datenbank anhand der vorhandenen Modelle erstellt.

Am Ende sollte das Verzeichnis so aussehen:

```
.
└── db_api
    ├── database.py
    ├── main.py
    ├── models.py
    ├── schemas.py
    └── backend.db
```

Zuerst wird eine SQLAlchemy-Engine erstellt. Dort wird die URL von der Datenbank benötigt. Dabei ist zu beachten, dass die Verbindung-Argumente nur von SQLite benötigt werden. *Das ist, dass nicht die gleiche Verbindung für mehrere Anfragen verwendet wird.*

Als zweitens muss eine lokale Session von der Datenbank gestartet werden. Dafür binden wir die Engine aus dem ersten Schritt für die Session.

Als letzter Schritt muss noch eine Base erstellt werden, welche später die Modelle und Schemas für die Datenbank erstellt.

```
#database.py

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Erster Schritt
DATABASE_URL = "sqlite:///./backend.db"
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})

```

```
# Zweiter Schritt
SessionLocal = sessionmaker(autocommit=False, autoflush=
    False, bind=engine)

# Dritter Schritt
Base = declarative_base()
```

Das Datenbankmodell muss nun eingebunden werden. Dafür muss zuerst die Base die zuvor erstellt worden ist, importiert werden, damit diese das Modell der Datenbank erstellen kann.

Nun muss für jede Tabelle der Datenbank eine Klasse definiert werden. Als Parameter wird die Base angegeben. Der Inhalt dieser Klasse besteht aus dem Namen der Tabelle und den Spalten, die benötigt werden. Für jede Spalte wird der Datentyp definiert werden und eine enthält zwingend den Primärschlüssel.

```
#models.py

from sqlalchemy import Column, Integer, String, Float,
    DateTime, func, Boolean

from database import Base

class Utterance(Base):
    __tablename__ = 'utterances'
    id = Column(Integer, primary_key=True, index=True)
    clip_path = Column(String)
    sentence = Column(String)
    sentence_source = Column(String)
    client_id = Column(String)
    zipcode = Column(Integer, nullable=True)
    town = Column(String, nullable=True)
    canton = Column(String, nullable=True)
    coord_lat = Column(Float)
    coord_lng = Column(Float)
    reported = Column(Integer)
    age = Column(String, nullable=True)
    gender = Column(String, nullable=True)

...
```

Als nächster Schritt wird für die jeglichen POST- und PUT-Methoden ein Pydantic-Modell erstellt, in dem das Schema bzw. die verlangte Datenstruktur festgehalten wird.

Dabei ist zu beachten, dass manche Spalten der Modelle nicht eingebunden werden, da diese automatisch erzeugt werden.

```python
#schemas.py

from pydantic import BaseModel

class UtteranceSchema(BaseModel):
    clip_path: str
    sentence: str
    sentence_source: str
    client_id: str
    zipcode: int
    town: str
    canton: str
    coord_lat: float
    coord_lng: float
    reported: int
    age: str
    gender: str

...
```

Als letzter Schritt müssen noch in der main.py-Datei Änderungen vorgenommen werden, damit die Datenbank beim Start der API erstellt wird, falls sie noch nicht vorhanden ist.

Dazu muss noch eine Funktion geschrieben werden, die eine Session mit der Datenbank erstellt.

```python
#main.py

...
from database import engine, SessionLocal, Base

Base.metadata.create_all(bind=engine)

# Code from FastAPI
...

def get_db():
    try:
        db = SessionLocal()
        yield db
    finally:
        db.close()
```

Nun können mit SQLAlchemy jegliche API-Methoden geschrieben werden.

Dockerize REST API

Im Verzeichnis, wo die main.py-Datei gespeichert wird, muss eine Dockerfile erstellte werden, dass zuerst das Verzeichnis des Conainers angibt. Danach werde alle Daten vom lokalen Verzeichnis in das neue Verzeichnis kopiert. Im nächsten Schritt werden alle Python-Pakte die definiert sind installiert. Zum Schluss wird der Port 8000 freigegeben und der Befehl, der beim start ausgeführt wird definiert.

```
FROM python:3.9

# Set the working directory in the container
WORKDIR /backend

# Copy the current directory contents into the container
   at /usr/src/app
COPY . ./backend

# Install any needed packages specified in requirements.
   txt
RUN pip install --no-cache-dir fastapi gunicorn uvicorn
   sqlalchemy matplotlib

# Make port 8000 available to the world outside this
   container
EXPOSE 8000

# Run main.py when the container launches
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--reload
   "]
#CMD ["gunicorn", "main:app", "-w", "4", "-k","uvicorn.
   workers.UvicornWorker", "--bind", "0.0.0.0:8000", "--
   timeout", "120"]
```

Da mit weiteren Docker Container gearbeitet werden, soll das Dockerfile nicht direkt ausgeführt, sondern in einer Docker Compose-Datei integriertet werden.

Hier kann der Container benannt werden. Es muss definiert werden, wo sich die Dockerfile befindet, relativ von der Docker Compose-Datei. Erneut muss angegeben werden über welchen Port dieser Container angesprochen wird.

Damit die Datenbank nicht nur im Docker Container sondern auch im lokalen Dateisystem aktualisiert wird, muss die SQLite-Datei noch gemountet werden.

```
version: '3'
services:
  backend:
    container_name: backend
    build: ./db_api
    ports:
      - "8000:8000"
    volumes:
      - ./db_api:/backend
    environment:
      - DATABASE_URL=sqlite:///./db_api/backend.db
```

Am Ende sollte das Verzeichnis so aussehen:

```
.
├── docker-compose.yml
└── db_api
    ├── Dockerfile
    ├── database.py
    ├── main.py
    ├── models.py
    ├── schemas.py
    └── backend.db
```

Die Docker Compose Datei kann einfach mit dem folgenden Befehl ausgeführt werden:

```
docker-compose up --build
```

–build ist notwendig, wenn der Container zum ersten Mal ausgeführt wird.

S3Proxy

Für die Speicherung der Audiodateien wird ein S3Proxy verwendet, welcher die Daten, die lokal abgespeichert sind, über eine API abrufen kann. Dafür wird Andrew-Gauls/S3Proxy verwendet, welcher über den Docker Hub zur verfügung gestellt wird. Somit muss nur ein Docker Compose-Datei erstellt werden mit den nötigen Konfigurationen.

In der Docker Compose-Datei muss angegeben werden, welche Image vom Docker Hub verwendet wird und wie der Container benennt wird. Folgend muss angegeben werden, über welchen Port der Container angesprochen werden kann.

Bei den restlichen Konfigurationen handelt es sich, um die CORS-Berechtigungen und um was für ein Speicher es sich handelt.

Die CORS-Berechtigungen werden beschränkt, so dass die Audiodateien nur abgefragt werden können und nicht neue hinzugefügt oder bestehende gelöscht werden können.

Bei den Cloud-Konfigurationen handelt es sich um ein Dateisystem, durch welches keine Identifizierung und Referenzen benötigt werden und kann einfach als ïgnored"gekennzeichnet werden.

Als letztes muss nur noch angegeben werden wo auf der lokalen Maschine die Dateien zu finden sind und wo diese im Docker-Container gemountet werden.

```
version: '3'
services:
  ... # REST API
  s3proxy:
    image: andrewgaul/s3proxy
    container_name: s3proxy
    expose:
      - '80'
    ports:
      - '8080:80'
    environment:
      S3PROXY_AUTHORIZATION: none
      S3PROXY_ENDPOINT: http://0.0.0.0:80
      S3PROXY_CORS_ALLOW_ORIGINS: http://localhost:8080/
   audio-bucket/
      S3PROXY_CORS_ALLOW_METHODS: GET
      S3PROXY_CORS_ALLOW_HEADERS: Accept Content-Type
      S3PROXY_CORS_ALLOW_CREDENTIAL: "true"
      JCLOUDS_PROVIDER: filesystem
      JCLOUDS_IDENTITY: "ignored"
      JCLOUDS_CREDENTIAL: "ignored"
      JCLOUDS_FILESYSTEM_ROOT_DIRECTORY: "/data"
    volumes:
      - ./audio/clips:/data/audio-bucket
```

Es ist wichtig, dass die Audiodateien dafür am richtigen Ort gespeichert werden.

Die Daten, welche wir aus der Dialektsammlung verwenden, werden zuerst nach den Sprechern (clientID) gruppiert und jeder Sprecher hat im Normalfall mehrere Sätze (sentenceID) aufgenommen.

```
.
├── docker-compose.yml
└── audio
    └── clips
        ├── ...
        └── [client_id]
            ├── ...
            └── [sentence_id].[mp3|flac]
```

Nun sollten die Dateien über http://127.0.0.1:8080/audiobucket/[clientID]/[sentenceID].[mp3|flac] abrufbar sein.

Die Docker Compose Datei kann einfach mit dem folgenden Befehl ausgeführt werden:

```
docker-compose up --build
```

–build ist notwendig, wenn der Container zum ersten mal ausgeführt wird.

Nominatim

Damit Nominatim verwendet werden kann müssen die Geocoding-Daten zuerst in einer Datenbank abgespeichert werden. Dafür gibt es wieder eine Image von Docker Hub, welcher verwendet werden kann.

Anders als bei REST API wird ein PostgreSQL Datenbank verwendet. Dafür muss bei Docker die Datenbank unter den Volumes definiert werden. Sobald der Docker Image angegeben ist und und dem Container einem namen gegeben worden ist, kann defniert werden woher die Daten genommen werden. In diesem Fall nehmen wir die neusten OpenStreetMap-Daten von der Schweiz.

Mit *shm-size* wird angegeben wie viel Speicher für den Import von den Dateien zur Verfügung gestellt wird.

Der Container wird über einen Port freigegeben.

```
version: '3'
services:
  ... #S3Proxy und REST API
  nominatim:
    image: mediagis/nominatim:4.3
    container_name: nominatim
    environment:
      - PBF_URL=https://download.geofabrik.de/europe/switzerland-latest.osm.pbf
      - REPLICATION_URL=https://download.geofabrik.de/europe/switzerland-updates/
      - IMPORT_WIKIPEDIA=false
      - NOMINATIM_PASSWORD=kih%&!!(OHBo87975!rOGILZG
    volumes:
      - nominatim-data:/var/lib/postgresql/14/main
    ports:
      - "8400:8080"
    shm_size: 1g

volumes:
  nominatim-data:
```

Die Docker Compose Datei kann einfach mit dem folgenden Befehl ausgeführt werden:

```
docker-compose up --build
```

–build ist notwendig, wenn der Container zum ersten mal ausgeführt wird.

Der Import der Dateien kann dauert einige Zeit. Sobald die Dateien heruntergeladen sind und der Container läuft, kann über http://127.0.0.1:8400 die Geocoding API verwendet werden.

5.2.2 Frontend

Es wird Schritt für Schritt erklärt, wie das Frontend-Framework implementiert worden ist und die Karte und Authentifizierung eingebunden worden sind.

Next.js

Um ein Next.js Applikation zu entwickeln muss zuerst *npm* und *npx* auf einem Gerät installiert sein.
Danach muss folgender Befehl eingegeben werden, um ein neues Next.js-Applikation zu erstellen:

```
npx create-next-app@latest PROJECT-NAME
```

PROJECT-NAME kann frei gewählt werden, muss aber klein geschrieben werden.

Nachdem diese Befehl ausgeführt worden, werden nach ein paar Konfigurationen abgefragt, bei welchen immer der Default-Wert ausgewählt worden ist.
Dies führt dazu, dass das Projekt mit TypeScript anstelle von JavaScript geschrieben wird und Tailwind CSS automatisch eingebunden wird.

Ohne weitere Konfigurationen vorzunehmen, kann nun Next.js mit folgenden Befehl für die Entwicklung ausgeführt werden:

```
npm run dev
```

Für die Live-Umgebung, sollte das Projekt zuerst gebildet werden und dann kann es normal gestartet werden, mit folgenden Befehlen:

```
npm build
npm run
```

Damit Komponenten benutzt werden können, muss ein Ordner erstellt werden, der *components* heisst. Dort drin können werden JSX-Dateien hinzugefügt, welche in anderen JSX-Dateien einfach aufgerufen werden können.

```
<COMPONENT-NAME example={param1}/>
```

Leaflet

Damit Leaflet eingebunden werden kann, muss zuvor es über die node module importiert werden. Folgende Module müssen importiert werden: *leaflet*, *react-leaflet* und *react-leaflet-vector-tile-layer*. Um die Karte bei Leaflet einzubinden, muss eine neue React-Komponente erstellt werden, mit dem Namen *Map.tsx*
Zuest werden die Node-Module von importiert, inklusive *useState* und *useEffect* von React. Weil es sich um eine Client-Komponente handelt, muss ganz oben noch *üse client"* hinzugefügt werden.

```
"use client"

import "leaflet/dist/leaflet.css"
import { MapContainer} from "react-leaflet";
import { useEffect, useState } from "react";
import L, { LatLngBounds } from "leaflet";
import VectorTileLayer from "react-leaflet-vector-tile-
   layer";
```

Danach werden für die Marker der Karte Icons erstellt. Ein Marker für das Raten des Dialektes und ein Marker für die Lösung. Dabei wird das Bild, der Schatten und die Grösse definiert. Zusätzlich wird angegeben, wo sich das Zentrum des Icons befindet.

```
let flagIcon = L.icon({
    iconUrl: "marker-icon-2x.png",
    iconRetinaUrl: "marker-icon-2x.png",
    iconAnchor: [3, 60],
    iconSize: [40, 60],
    shadowUrl: "marker-shadow.png",
    shadowSize: [60, 60],
    shadowAnchor: [3, 60]
})

let targetIcon = L.icon({
    iconUrl: "marker-icon_target.png",
    iconRetinaUrl: "marker-icon_target.png",
    iconAnchor: [12, 41],
    popupAnchor: [10, -44],
    iconSize: [25, 41],
});
```

Damit die Komponente aufgerufen werden kann, muss diese erstellt werden und auch für andere als Export zur Verfügung stehen.

```
const Map = () => {
    ...
    return(...)
}
export default Map;
```

Innerhalb der Komponente, müssen die beiden Marker gewisse Informationen besitzen. Dafür werden *useStates* verwendet, so dass diese Daten gespeichert werden.

Der Marker für das Raten darf sich nur während des Ratens bewegen und nicht während der Kontrolle der Lösung.
Der Marker der Lösung soll erst nach dem Raten ersichtlich sein.
Für beide Marker werden noch die Koordinaten und Informationen der Ortschaft abgespeichert.

```
const [draggable, setMarkerDraggable] = useState(true);
const [markerVisible, setMarkerVisible] = useState<boolean>(false);
const [markerData, setMarkerData] = useState<MarkerData>({
    coordinates: [center.lat, center.lng],
    location: {
         plz: 6441,
        town: "Ruetli",
        canton: "UR"
    }
});
const [markerTargetData, setMarkerTargetData] = useState<MarkerData>({
    coordinates: [center.lat, center.lng],
    location: {
        plz: 0,
        town: "",
        canton: ""
    }
});
```

Die Karte selbst wird als *MapContainer* hinzugefügt. Damit jedoch eine Karte dargestellt wird, muss noch ein *VectorTileLayer* hinzugefügt werden.

Zusätzlich werden die beiden Marker als Komponenten hinzugefügt. Sobald die Informationen der Marker aktualisiert werden, wird über die *useEffect*-Funktion automatisch deren Koordinaten auch aktualisiert.

```
return (
    <>
        <MapContainer
            className="map"
            center={[46.8, 8.289]}
            zoom={8}
            minZoom={8}
            scrollWheelZoom={true}
            maxBounds={switzerlandBounds}
        >
            <VectorTileLayer
                attribution='&copy; <a href="https://www.swisstopo.ch/">Swisstopo</a>'
                styleUrl="style.json"
            />
            <DraggableMarker draggable={draggable} markerData={markerData} setMarkerInformation={setMarkerInformation} flagIcon={flagIcon} />
                {markerVisible && (
                <TargetMarker markerData={markerTargetData} flagIcon={targetIcon} />
            )}
        </MapContainer>
```

```
    </>
)
```

Lucia Auth

Bevor Lucia implementiert werden kann, müssen die node-module *@lucia-auth/adapter-sqlite*, *better-sqlite* und *lucia* installiert werden.

Zuerst muss die Datenbank und das Modell dazu erstellt werden. Es wird eine Datei *db.ts* erstellt und eine verbindung mit der SQLite Datenbank-Datei aufgebaut.

```
import sqlite from "better-sqlite3";

export const db = sqlite("main.db");
```

In dieser Datei werden auch noch die Modelle im Code definiert. Diese Modelle werden genauer im Kapitel 4.2.4 erklärt.

```
...

db.exec(‘CREATE TABLE IF NOT EXISTS password_reset_token (
    token_hash TEXT NOT NULL UNIQUE,
    user_id TEXT NOT NULL,
    expires_at DATE
)‘);
```

Zuerst muss ein Adapter für die Datenbank erstellt werden, danach werden die Session-Cookie-Optionen definiert. Weil grundsätzlich HTTPS verwendet wird, sind diese Attribute sicher.

Zusätzlich wird noch eine Funktion geschrieben, welche anhand des Benutzernamens die Session überprüft und zurückliefert, ob diese schon abgelaufen ist.
Dabei ist zu beachten, dass beim Login eine neue Session erstellt wird und beim Logout die alte gelöscht wird.

Zusätzlich muss Lucia noch als Modul deklariert werden.

```
import { Lucia } from "lucia";
import { BetterSqlite3Adapter } from "@lucia-auth/adapter-
   sqlite";
import { db } from "./db";
...
import type { DatabaseUser } from "@/interfaces/db_user";

const adapter = new BetterSqlite3Adapter(db, {
    user: "user",
    session: "session"
});

export const lucia = new Lucia(adapter, {
    sessionCookie: {
        attributes: {
        secure: process.env.NODE_ENV === "production"
```

```
    },
        ...
    }
});

export const validateRequest = cache(...);

// must have! IMPORTANT!
declare module "lucia" {
    interface Register {
    Lucia: typeof lucia;
    DatabaseUserAttributes: Omit<DatabaseUser, "id">;
    }
}

});
```

Nun kann bei jeder Seite des Frontends die Session überprüft werden. Sollte keine aktive Session gefunden werden, soll direkt auf die Login-Seite weitergeleitet werden.

```
import { validateRequest } from "@/lib/auth";
import { redirect } from "next/navigation";

export default async function Page() {
    const { user } = await validateRequest();
    if (!user) {
        return redirect("/login");
    }

    return (...)
}
```

Passwort zurücksetzen

Um ein Passwort zurücksetzen zu können, muss ein Token erstellt werden. So wird sichergestellt, dass es sich auch wirklich um den legitimen Eigentümer dieses Accounts handelt.
Von Lucia Auth wird dieser Hash-Token erstellt und mittels von *nodemailer*, einem Node-Modul, wird ein E-Mail an die Adresse geschickt, die bei der Registration angegeben werden musste.
Bei Nodemailer muss angegeben werden, um welchen *EMAIL_SERVER_HOST* es sich handelt und über welchen *Port* dieser kommuniziert. Am besten sollte ein SMTP-Port ausgewählt werden. Zusätzlich muss noch ein *Benutzernamen* und *Passwort* angegeben werden, um sich beim Host zu authentifizieren.
Nachdem diese Angaben gemacht worden sind, kann ein neues E-Mail definiert werden. Dabei wird die Absender- sowie Empfänger-E-Mailadresse angegeben. Als Inhalt des Mails wird ein Link geschickt, der zu einem Formular führt, bei welchem das Passwort anhand des Tokens zurückgesetzt werden kann.

Dockerize Frontend

Damit das Frontend ebenfalls über Docker gestartet werden kann, muss ebenfalls ein Dockerfile erstellt werden, das alle Daten von Projekt auf einem virtuellen Docker Container kopiert, einen Port öffnet und die nötigen Befehle ausführt.
Dabei müssen alle Node-Module importiert werden und das Projekt neu gebuildet werden. Nach diesen Befehlen muss das Projekt mit einem Befehl gestartet werden.

```
FROM node:18

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
RUN npm run build
CMD npm start
```

Damit das Frontend auch in das Docker Netzwerk eingebunden werden kann, muss das Dockerfile einer Docker-Compose-Datei hinzugefügt werden.
Dabei wird dem Container einen Namen gegeben, und über welchen Port dieser intern und extern angesprochen werden kann. Zusätzlich wird die Datenbank für die Authentifizierung gemountet.

Der Name des Netzwerkes wird noch angegeben werden und definiert, dass es sich um ein externes Netzwerk handelt.

Es wird ausserdem angegeben, dass der Container eigenständig neu starten soll, falls dieser aus irgendwelchen Gründen seine Ausführung beenden sollte.

```
version: '3.8'
services:
  app:
    build: .
    ports:
      - "3000:3000"
    container_name: frontend
    volumes:
      - ./main.db:/app/main.db
    networks:
      - shared-network
    restart: always
networks:
  shared-network:
    external: true
```

Die Docker-Compose-Datei kann einfach mit dem folgenden Befehl ausgeführt werden:

```
docker-compose up --build
```

–build ist notwendig, falls der Container zum ersten Mal ausgeführt wird.

Nun sollte das Frontend über http://127.0.0.1:3000 erreichbar sein.

5.2.3 Struktur der lokalen Umgebungs-Datei

Während der Beschreibung des Deployments, wird vom *.evn.local*-Datei gesprochen (siehe Kap. 4.4.1). Hier werden viele sensible Daten abgespeichert, welche im Projekt verwendet werden.
Hier werden die Verbindungsinfromationen zum externen E-Mail-Dienst gespeichert, welche benötigt werden, um eine Verbindung zum Server aufzubauen und schlussendlich eine Mail zu senden.
Ebenfalls wird hier der geheime Schlüssel definiert, welche für manche Ent- und Verschlüsselungen zuständig sind. Diese Daten werden sehr wahrscheinlich nicht so oft geändert.
Jedoch die benötigten URL müssen je nach Methode anders deklariert werden. Zum Beispiel, wenn am Projekt gearbeitet wird, muss es nicht mit einer Domäne angesprochen werden, sondern auf dem lokalen Gerät. Somit kann der URL einfach vor dem Start gewechselt werden. Ebenfalls kann definiert werden, ob die Docker Container vom Backend wie beim Deployment die Docker Container innerhalb des Docker Netzwerkes anspricht oder diese von ausserhalb angesprchen werden. Der Vorteil diese von aussen anzusprechen ist es, dass das Frontend nicht im Docker Netzwerk ist während der Entwicklung.

```
1  // E-mail service
2  EMAIL_SERVER_USER="username"
3  EMAIL_SERVER_PASSWORD="password"
4  EMAIL_SERVER_HOST="server-host"
5  EMAIL_SERVER_PORT=465
6  EMAIL_FROM="email-address"
7
8  // Secret key
9  SECRET_KEY="private key"
10
11 // Domain
12 DEFAULT_URL="https://dialekterkennung.ch"
13 #DEFAULT_URL="http://localhost:3000"
14
15 // Production
16 BACKEND_URL="http://backend:8000"
17 S3PROXY_URL="http://s3proxy:80"
18 NOMINAtIM_URL="http://nominatim:8080"
19
20 // Development
21 #BACKEND_URL="http://localhost:8000"
22 #S3PROXY_URL="http://localhost:8080"
23 #NOMINAtIM_URL="http://localhost:8400"
```

5.3 Restliche Konfusionsmatrizen

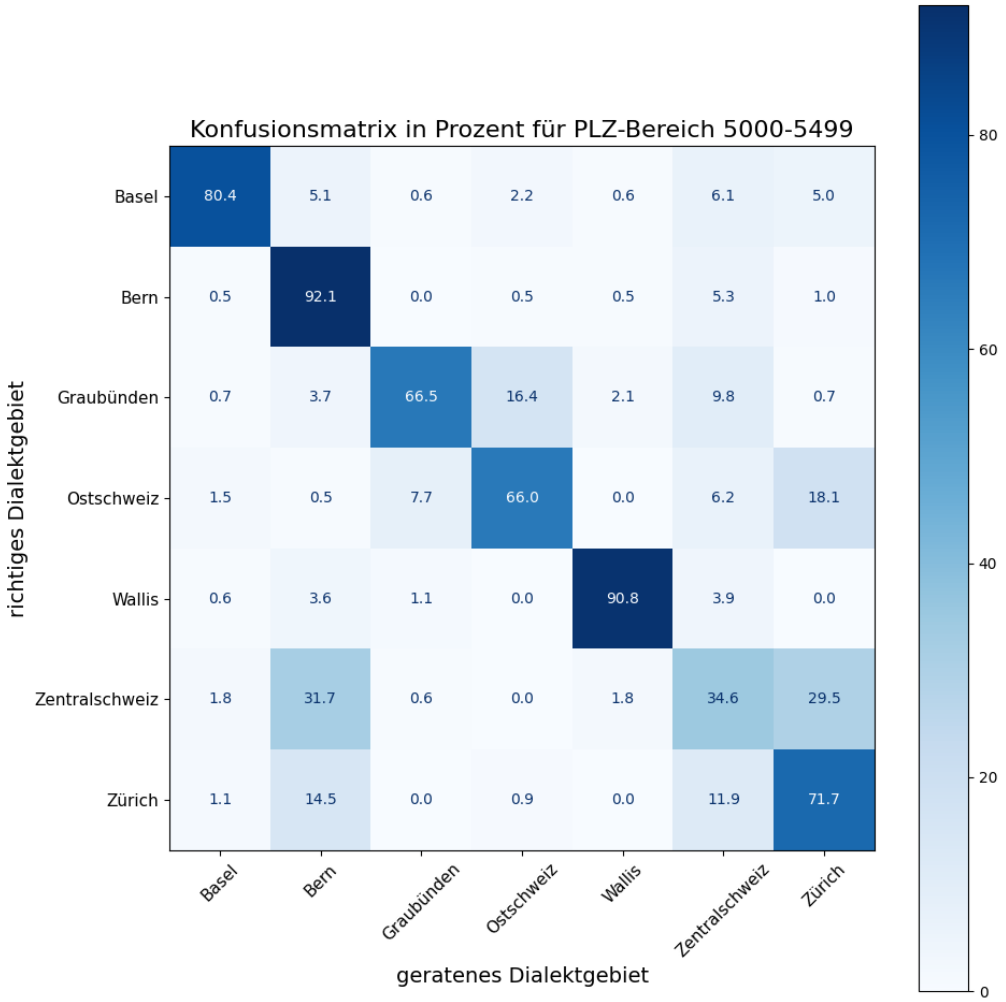

Abbildung 5.1: Konfusionsmatrix in Prozent für den PLZ-Bereich 5000-5499

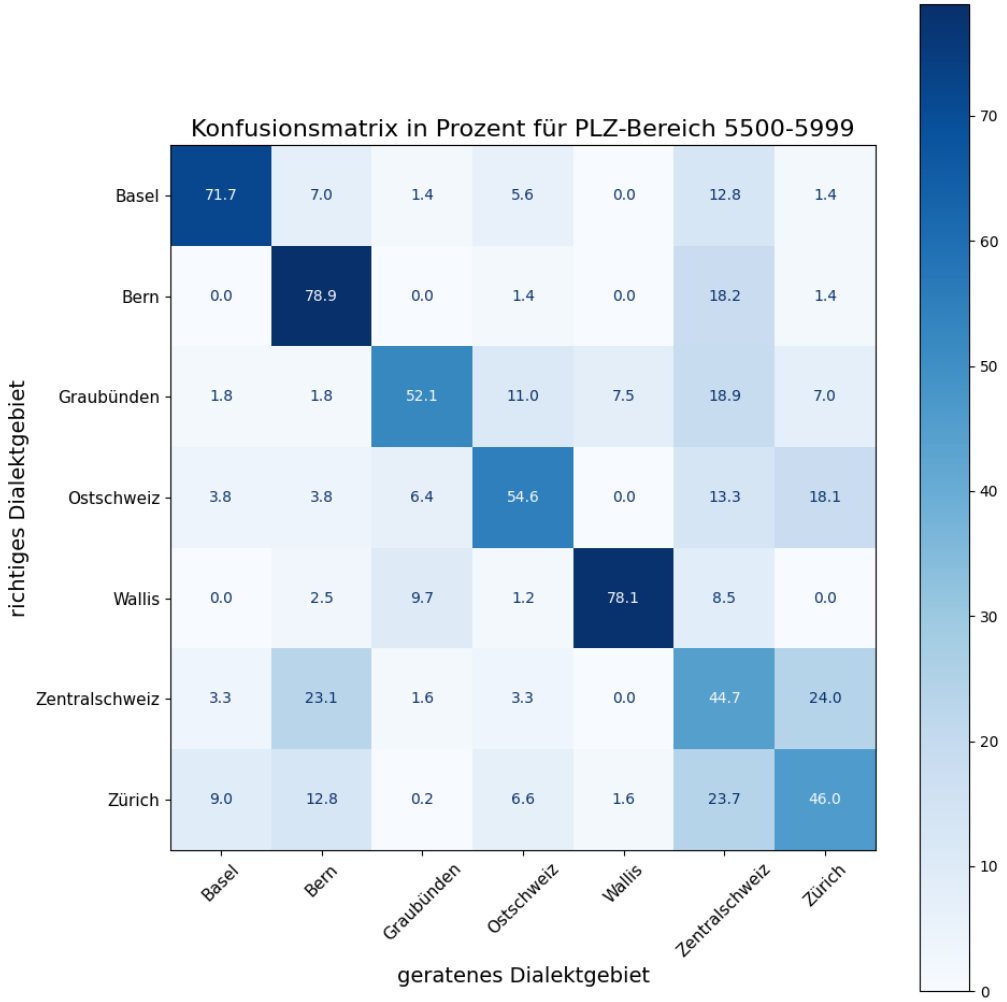

Abbildung 5.2: Konfusionsmatrix in Prozent für den PLZ-Bereich 5500-5999

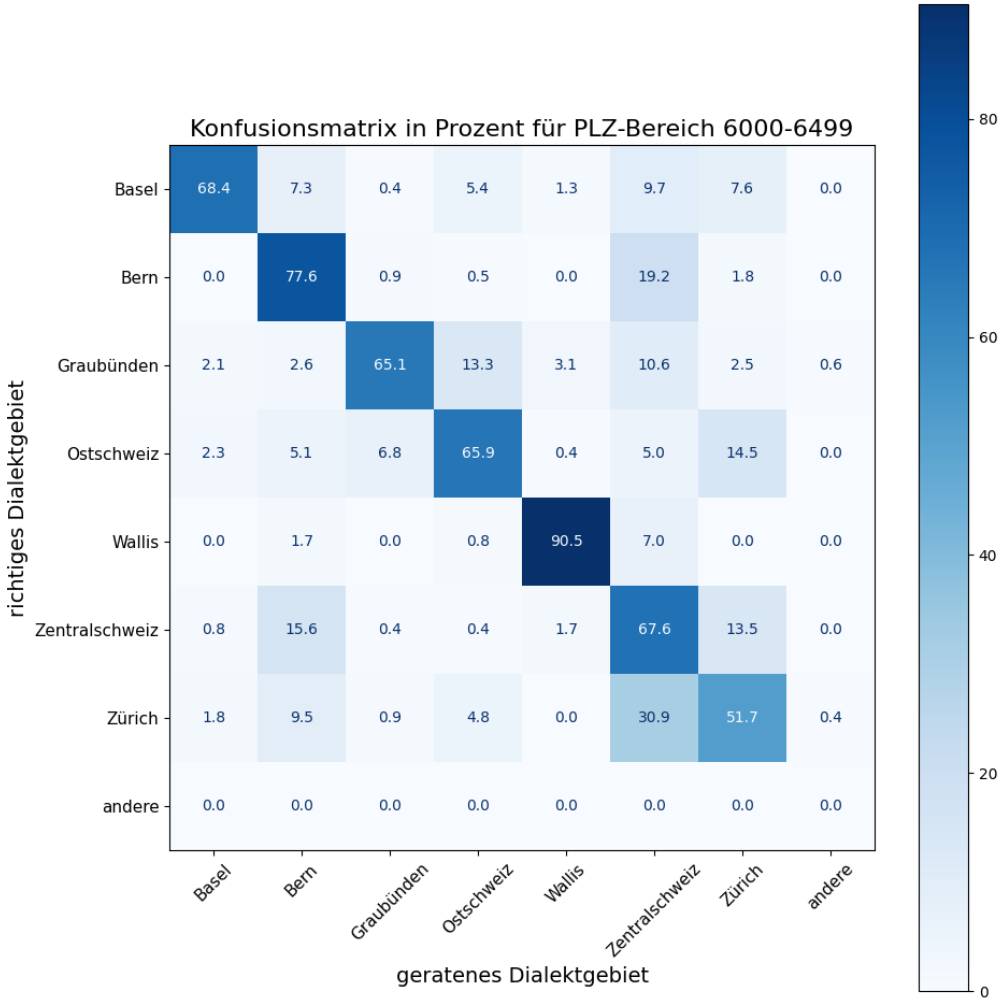

Abbildung 5.3: Konfusionsmatrix in Prozent für den PLZ-Bereich 6000-6499

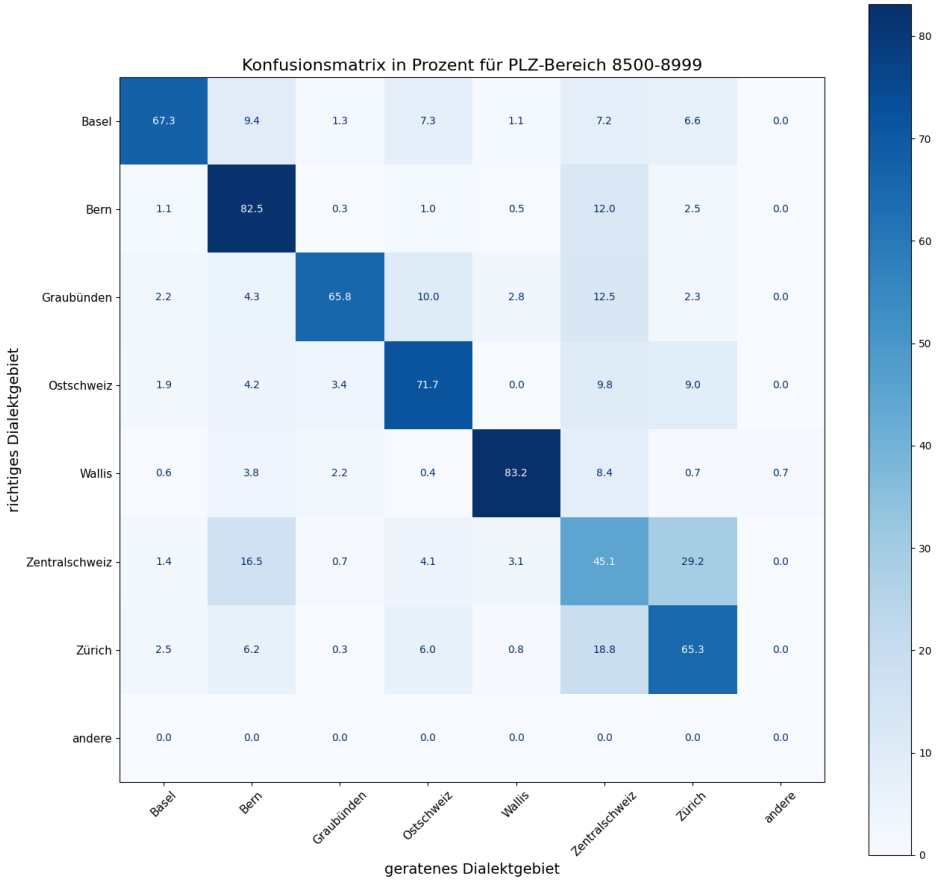

Abbildung 5.4: Konfusionsmatrix in Prozent für den PLZ-Bereich 8500-8999

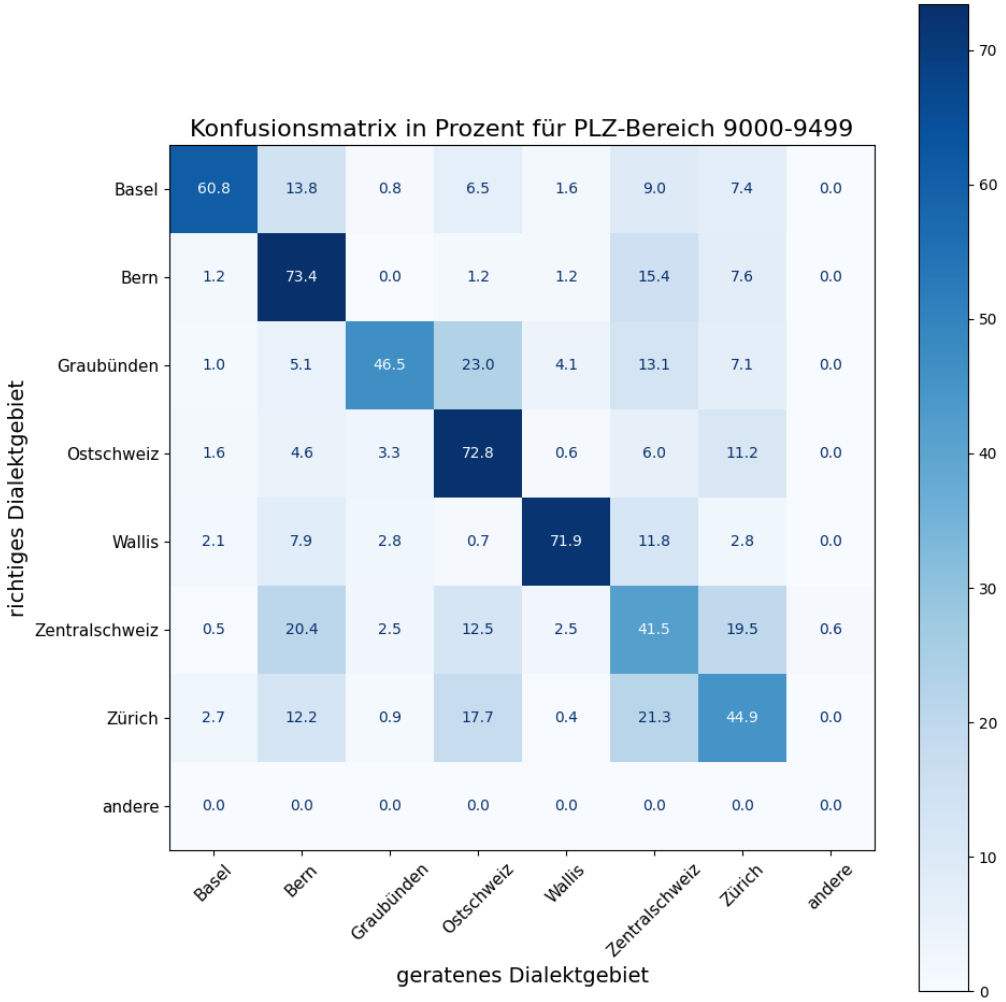

Abbildung 5.5: Konfusionsmatrix in Prozent für den PLZ-Bereich 9000-9499

Abbildungsverzeichnis

Literatur

[1] Charles A. Ferguson. *Sociolinguistic Perspectives: Papers on Language in Society, 1959-1994 (Oxford Studies in Sociolinguistics)*. 1996.

[2] Gottfried Kolde. *Sprachkontakte in gemischtsprachigen Städten: vergleichende Untersuchungen über Voraussetzungen und Formen sprachlicher Interaktion verschiedensprachiger Jugendlicher in den Schweizer Städten Biel/Bienne und Fribourg/Freiburg i. Ue.* Zeitschrift für Dialektologie und Linguistik. Wiesbaden: Steiner, 1981.

[3] Christa Dürscheid; Franc Wagner; Sarah Brommer. *Wie Jugendliche schreiben: Schreibkompetenz und neue Medien*. Linguistik – Impulse & Tendenzen; 41. De Gruyter, 2010.

[4] Dominik Heeb. *Das Chochichästli-Orakel*. URL: http://dialects.from.ch/ (besucht am 25. 03. 2024).

[5] Dariush Mehdiaraghi und Marc Brupbacher. *Unser Dialekt-Test weiss, woher Sie stammen*. Juli 2023. URL: https://www.tagesanzeiger.ch/unser-dialekt-test-weiss-woher-sie-stammen-132564027495 (besucht am 27. 05. 2024).

[6] Dariush Mehdiaraghi und Marc Brupbacher. *Dialekt-Test: Wir wissen, woher Sie stammen - jetzt noch genauer*. März 2024. URL: https://www.tagesanzeiger.ch/dialekt-test-wir-wissen-woher-sie-stammen-jetzt-noch-genauer-205060715129 (besucht am 04. 06. 2024).

[7] Adrian Leemann. *Dialäkt Äpp im App Store*. URL: https://apps.apple.com/ch/app/dial%C3%A4kt-%C3%A4pp/id606559705.

[8] o.A. *sprachatlas.ch - Der Sprachatlas der deutschen Schweiz*. URL: http://sprachatlas.ch (besucht am 24. 05. 2024).

[9] Elvira Glaser. *Ist das Schweizerdeutsche eine eigene Sprache?* URL: https://www.linguistik.uzh.ch/de/easyling/faq/kolmer-schweizerdeutsch.html (besucht am 12. 05. 2024).

[10] Christa Dürscheid. „Soziolekt (Lexikonartikel)“. In: *Kernbegriffe der Sprachdidaktik Deutsch : Ein Handbuch*. Hrsg. von Björn Rothstein und Claudia Müller. Baltmannsweiler: Schneider Verlag Hohengehren, 2013, S. 381–383.

[11] Emanuel Ruoss. *Zur Konsolidierung der Deutschschweizer Diglossie im 19. Jahrhundert*. Berlin, Boston: De Gruyter, 2019.

[12] o.A. *«Samnauner Dialekt»*. URL: https://www.samnaun.ch/de/samnauner-dialekt (besucht am 30. 05. 2024).

[13] o.A. *Sprachlandschaften - Dialektlandschaften*. URL: https://www.adfontes.uzh.ch/tutorium/die-deutsche-sprache-in-den-quellen/deutsche-sprachlandschaften-und-schreibsprachen/sprachlandschaften-dialektlandschaften (besucht am 27. 05. 2024).

[14] o.A. *Schweizerdeutsch*. URL: https://www.idiotikon.ch/schweizerdeutsch-info.

[15] o.A. *Korpus > Rechtschreibung, Bedeutung, Definition, Herkunft | Duden*. URL: https://www.duden.de/rechtschreibung/Korpus_Sammlung (besucht am 04. 06. 2024).

[16] Michel Plüss u. a. „STT4SG-350: A Speech Corpus for All Swiss German Dialect Regions“. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Hrsg. von Anna Rogers, Jordan Boyd-Graber und Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, Juli 2023, S. 1763–1772.

[17] o.A. *Hunziker2020 Aargauer Wörterbruch - Aargauer Dialektlandschaft*. URL: https://hls-dhs-dss.ch/de/articles/026413/2012-02-08/ (besucht am 07. 06. 2024).

[18] Andreas Würgler. *Historische Lexikon der Schweiz - Eidgenossenschaft*. Aug. 2012. URL: https://hls-dhs-dss.ch/de/articles/026413/2012-02-08/ (besucht am 07. 06. 2024).

[19] o.A. *Vorherrschendes Dialektwort für "Bonbon"*. URL: https://www.kleinersprachatlas.ch/interaktive_karten/F16_Bonbon/bonbon.html (besucht am 07. 06. 2024).

[20] o.A. *Next.js – Präzise, Innovativ, Zuverlässig*. URL: https://www.netnode.ch/nextjs (besucht am 02. 03. 2024).

[21] Petro Salgado. *Leaflet – bessere Alternative zu Google Maps*. Newsletter. URL: https://www.bergx2.de/en/news/leaflet-bessere-alternative-zu-google-maps/.

[22] o.A. *Lucia documentation*. URL: https://lucia-auth.com/ (besucht am 25. 03. 2024).

[23] tiangolo. *FastAPI*. URL: https://fastapi.tiangolo.com/ (besucht am 05. 03. 2024).

[24] o.A. *SQLAlchemy - The Database Toolkit for Python*. URL: https://www.sqlalchemy.org/ (besucht am 05. 03. 2024).

[25] o.A. *Docker Docs*. URL: https://docs.docker.com/get-docker/ (besucht am 15. 03. 2024).

[26] andrewgaul. *gaul/s3proxy*. GitHub. Mai 2024. URL: https://github.com/gaul/s3proxy (besucht am 17. 05. 2024).

[27] o.A. *Open-source geocoding with OpenStreetMap data*. URL: https://nominatim.org/ (besucht am 27. 05. 2024).

[28] o.A. *SQLite Home Page*. URL: https://www.sqlite.org/index.html (besucht am 05. 03. 2024).

[29] o.A. *Allgemeine Nutzungsbedingungen und Betriebsbestimmungen der Bundes Geodaten-Infrastruktur BGDI*. Dez. 2021. URL: https://www.geo.admin.ch/de/allgemeine-nutzungsbedingungen-bgdi (besucht am 10. 03. 2024).