SA Machine Learning für Bilderkennung

Release Abgabe-1-g4099bf9

Diluxion Marku **Lukas Dätwyler**

23.12.2021

Inhaltsverzeichnis

Kürzel SA

Beschreibung SA Projekt

Autoren Diluxion Marku, Lukas Dätwyler (E-Mail an alle)

Version Abgabe-1-g4099bf9

KAPITEL 1

Aufgabestellung

1.1 Beteiligte Personen

- Teilnehmende: Lukas Dätwyler und Diluxion Marku
- Partner: Trihow AG, Christoph Schneider und Loana Albisser
- Betreuer: Prof. Frank Koch

1.2 Problembeschrieb seitens Industriepartner

In Workshops werden Fotos von Whiteboards aufgenommen, die mit handgeschriebenen Post-it Zetteln bestückt sind. Diese Fotos sollen in weiterverarbeitbare Grafiken überführt werden.

1.3 Konkrete Aufgabenstellung

Grundlage dieser Aufgabenstellung ist der für das HS21 gültige Leitfaden für Bachelor- und Studienarbeiten. Ziel der Arbeit ist ein Prototyp, der die Post-it mit Google AutoML oder Google Cloud Vision API via Objekterkennung und Texterkennung auswertet, sodass für jedes Post-it ein synthetisch erstelltes Bild mit Metadaten in der Cloud gespeichert werden kann. Die App zum Fotografieren der Post-it existiert bereits und wird vom Industriepartner Trihow zur Verfügung gestellt. Trihow stellt weiter Trainings- und Testdaten sowie den Technologie-Stack zur Verfügung.

Mit den Daten wird ein Modell trainiert, welches in einem Bild Post-it erkennen und segmentieren kann. Für die erkannten Post-it wird dann mittels eines von Google vortrainierten Modells die Texterkennung durchgeführt. Anhand der Informationen aus der Segmentierung und Texterkennung wird ein synthetisches Bild erstellt, in welchem die Inhalte des erkannten Post-it und deren Text abgebildet sind. Das synthetische Bild sowie die Metadaten mit äquivalenten Informationen (Post-it Koordinaten und Textinhalte) werden anschliessend in der Cloud abgelegt. Die Umsetzung erfolgt als Prototyp in TypeScript unter Verwendung der Cloud Umgebung Firebase / Firestore.

1.4 Funktionale Anforderungen

- Aufbau und Trainieren der Modelle mit Google AutoML oder Google Cloud Vision API
- Erkennung von Post-it Zettel in Fotos
- Generieren von strukturierten Metadaten in Firestore
- Generieren synthetisches Bild PNG/SVG in Firebase
- Prototyp in TypeScript unter Verwendung von Google AutoML und Firestore (CloudDB).

KAPITEL 2

Anforderungsspezifikationen

2.1 Einführung

2.1.1 Zweck

Die Anforderungsspezifikation beschreibt die konkreten Anforderungen, die wir an unsere Softwarelösung stellen.

2.1.2 Gültigkeitsbereich

Die Anforderungsspezifikationen sind für das Herbstsemester 2021 gültig, was der Dauer dieses Projekts entspricht.

2.1.3 Referenzen

Nicht gebräuchliche Begriffe und Abkürzungen werden im Glossar aufgelistet.

2.2 Allgemeine Beschreibung

2.2.1 Produkt Funktion

Das Produkt beinhaltet eine Softwarelösung mit einer Machine-Learning-Komponente, welche durch Bild- und Texterkennung Post-it Zettel auf den hochgeladenen Bildern erkennt und daraus eine digitale Version des Post-it erstellt und im Firestore speichert.

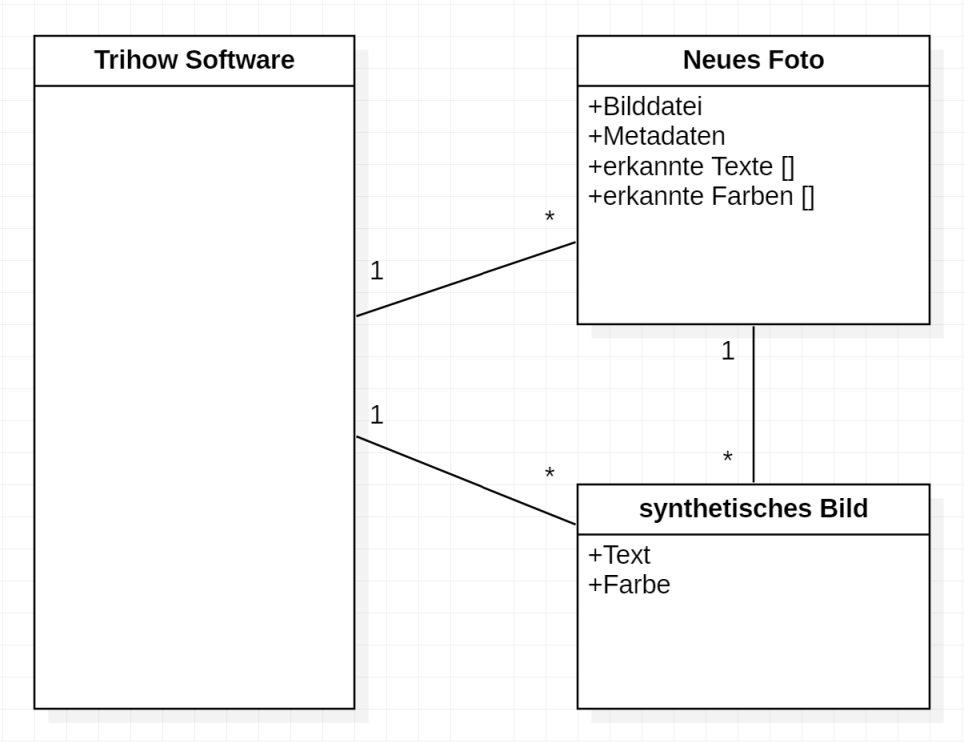
2.2.2 Einschränkungen

Da wir die bestehende Software von Trihow verwenden, betrifft unsere Lösung nur das Backend. Das Frontend werden wir nicht bearbeiten. Ausserdem ändern wir nichts an der verwendeten Serverstruktur. Somit besteht für das Projekt nur der Use Case, Bilder hochzuladen, welche dann analysiert werden. Die restliche CRUD-Funktionalität ist durch den bereits verwendeten Server gegeben.

2.2.3 Abhängigkeiten

Das Produkt hängt davon ab, ob wir schlussendlich die Cloud Vision API verwenden oder das Google AutoML benutzen, um unser eigenes Modell zu trainieren.

2.3 Strukturdiagramm



2.4 Use Cases

2.4.1 Aktoren & Stakeholder

- Eingeloggter und in eine Session beigetretener User
  - Der User kann in der Session Bilder von Post-it Zetteln hochladen, aus welchen digitale Versionen erstellt werden. *Bemerkung*: Verschiedenen Rollen wie z. B. Owner und Moderator wurden nicht beachtet, da die Komponente unabhängig vom User, jedes Bild auf Post-it Zettel überprüft.

2.4.2 Beschreibungen (Brief)

1. **Bild hochladen**: Der User lädt in einer Session ein Bild von einem Post-it Zettel hoch.

2.4.3 Beschreibungen (Fully Dressed)

Use Case 1: Bild hochladen

- **Primary Actor**: eingeloggter und in eine Session beigetretener User
- **Scope**: beigetretene Session

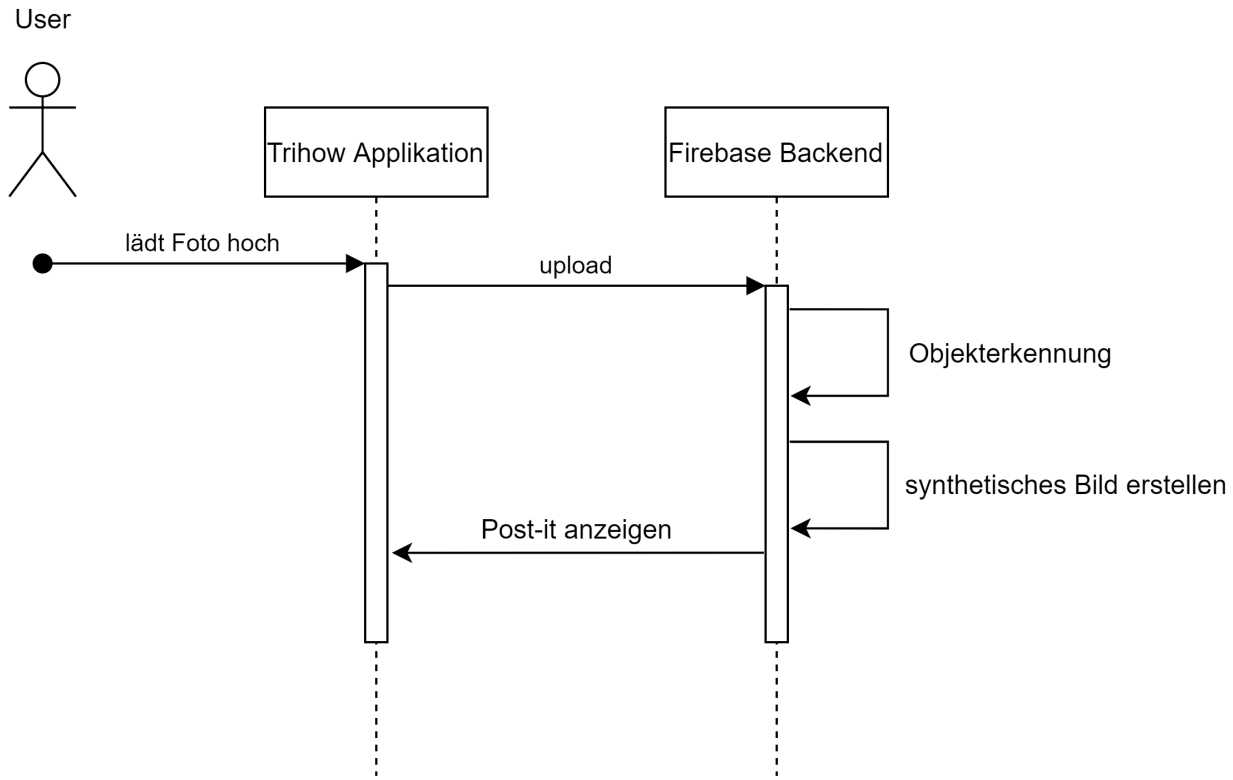
Main success scenario:

1. Der User kann ein Bild hochladen.
2. Das Post-it auf dem Bild wird erkannt.
3. Die Texte auf dem Post-it werden erkannt.
4. Die Metadaten und die Informationen aus den Texten werden in Firestore gespeichert.
5. Layout Konfigurationen des zu erzeugenden Bildes werden festgelegt.
6. Das synthetische Bild des Post-its wird erstellt und als PNG in der Cloud gespeichert.

Extensions:

- 2a. Form und Farbe des Post-its werden erkannt.
- 3a. Einfache Skizzenerkennung.
- 5a. Layout Konfigurationen selbst festlegen.
- 6a. Das synthetisch erstellte Post-it Bild als SVG in der Cloud speichern.

Sequenzdiagramm



KAPITEL 3

Projektplan

3.1 Einführung

3.1.1 Zweck

Dieses Dokument beschreibt den Projektplan für die SA „Machine Learning für Bilderkennung und Synthese von Post-it Fotos“.

3.1.2 Referenzen

Nicht gebräuchliche Begriffe und Abkürzungen werden im Glossar des Projekts aufgelistet.

- Trihow AG
- Trello
- Clockify
- TypeScript
- TSDoc
- TypeDoc
- Firebase
- Postman

3.2 Projekt Übersicht

In Zusammenarbeit mit der Firma Trihow soll ein Prototyp für die maschinelle Erkennung von Post-it Zetteln erstellt werden, welcher direkt in die vorhandene App integriert wird. Die erkannten Post-its sollen als digitale Versionen synthetisiert und zur Verfügung gestellt werden.

3.2.1 Zweck und Ziel

Das Ziel ist eine lauffähige Erweiterung der Software von Trihow, welche den Text und die Farbe eines Post-it Zettels erkennt und aus diesen Informationen ein synthetisches Bild erstellt. Zweck der Arbeit ist des Weiteren eine Weiterbildung der Teilnehmenden in Bezug auf maschinelle Bilderkennung.

3.2.2 Lieferumfang

- Eine lauffähige Version der Trihow Software, welche das Ziel der Arbeit erfüllt.
- Der komplette Source-Code bzgl. der Post-it Erkennung und Bild-Synthese.
- Die erstellte Dokumentation des Projekts.

3.2.3 Annahmen und Einschränkungen

Das Projekt baut auf der bestehenden Trihow Software auf. Den Teilnehmenden wird der Zugriff auf diese, sowie auf die gesamte weitere Infrastruktur gestattet.

3.3 Management-Abläufe

3.3.1 Kostenvoranschlag

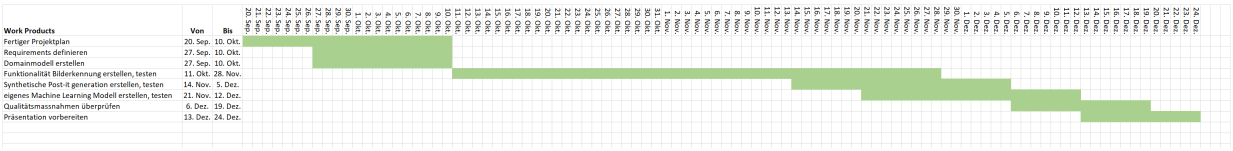
Beide Teilnehmer leisten je ungefähr 240 Arbeitsstunden. Es wird wöchentlich an der SA gearbeitet.

3.3.2 Zeitliche Planung

Für das Management der einzelnen Tasks wird *Trello* verwendet.

Die Zeiterfassung erfolgt mit *Clockify* (das in *Trello* integriert werden kann).

Phasen / Iterationen



Bezeichnung der einzelnen Phasen

1. *Inception:* In der initialen Besprechung wurde der Scope des Projekts definiert.
2. *Elaboration:* Wir setzen uns mit der bestehenden Infrastruktur und den verwendeten Tools auseinander. Um den Scope genauer zu definieren, wird ein Prototyp mit zwei unterschiedlichen Ansätzen erstellt. Einer mit der Vision API und ein zweiter mit dem Google AutoML. Am Schluss der Elaboration-Phase werden die beiden Ansätze verglichen und mit Trihow entschieden, welcher weiterverfolgt wird.
3. *Construction:* Der gewählte Ansatz wird fertig entwickelt und um weitere Funktionalität erweitert.
4. *Transition:* Die Software wird von Trihow entgegengenommen.

Bezeichnung der einzelnen Iterationen

Die einzelnen SCRUM Iterationen dauern jeweils 4 Wochen. Dazwischen finden Sprint Planning Treffen mit Trihow statt. Zusätzlich werden wöchentliche Treffen mit Frank Koch abgehalten.

Meilensteine

Bezeichnung der einzelnen Meilensteine

Datum	Meilenstein	Arbeitsprodukt
10. 10. 2021	Projektplan fertig	Projektplan
31. 10. 2021	Lauffähiger Prototyp ML	Source Code
5. 12. 2021	Synthetische Post-it Generation	Source Code
24. 12. 2021	Projektabgabe	Source Code + Dokumentation

1. Projektplan fertig

Der Projektplan ist fertiggestellt, sodass er mit Loana Albisser von Trihow besprochen werden kann. Darin enthalten sind ebenfalls Use Cases und die verwendete Architektur.

2. Lauffähiger Prototyp ML

Ein Prototyp, welcher mit Google Vision verbunden ist, und so Texterkennung und Objekterkennung auf einem Bild durchführen kann. Er muss noch nicht in das bestehende Trihow System eingebaut sein.

3. Synthetische Post-it Generation

Aus dem Post-it Objekt wird automatisch ein synthetisches Bild mit dem erkannten Text und der richtigen Farbe erzeugt.

4. Projektabgabe

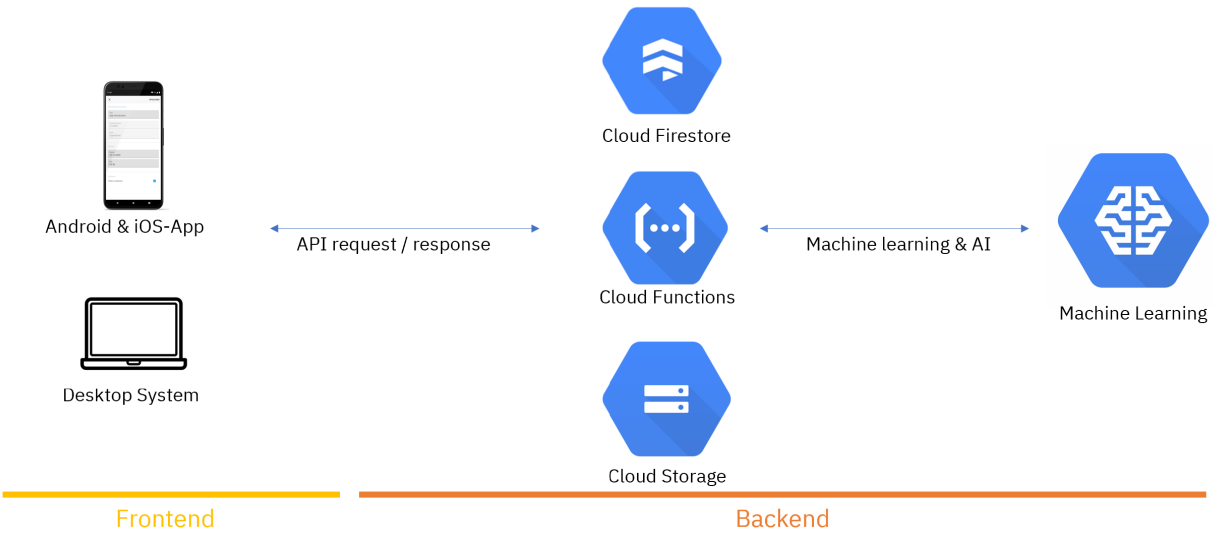
Das gesamte Projekt wird inklusive Source Code und Dokumentation an Trihow und an die Betreuungsperson abgegeben.

3.3.3 Besprechungen

Wöchentlich wird das Projekt mit der Betreuungsperson, Frank Koch, besprochen. Dabei wird überprüft, ob das Projekt zeitlich und bezüglich Scope im Rahmen der Planung liegt. Ausserdem werden allfällige technische Fragen geklärt.

3.4 Infrastruktur

3.4.1 Systemübersicht



Trihow stellt das lauffähige System gemäss Skizze, exklusive des Machine Learning Teils, zur Verfügung. Der Auftrag der Semesterarbeit ist es, einen Prototypen diesen Teils zu entwickeln und implementieren.

Das Projekt besteht aus mehreren cloud functions, welche in der Sprache *Typescript* geschrieben werden. Dabei handelt es sich um Funktionen, die auf einem *Firebase Server* gespeichert sind und durch unterschiedliche Triggers ausgeführt werden können (z. B. Wenn ein neues Bild hochgeladen wird oder mit einem HTTP Aufruf).

Für die Entwicklung werden weitere Tools wie der Firebase Emulator und Postman genutzt.

3.4.2 trihow-research Firebase Server

Vom Firebase Server sind für uns vor allem drei Reiter interessant.

Firestore Database

Dies ist eine NoSQL Datenbank, auf welcher in JSON-Dokumenten die Metadaten, sowie die von der Bilderkennung erstellten Informationen zu jedem Foto gespeichert sind.

Firebase Storage

Dies ist ein normales Filesystem mit Buckets, in welchen die hochgeladenen Bilder gespeichert werden. Um für Testzwecke die Autorisierung zu umgehen, haben wir einen zweiten Bucket erstellt. Dieser ist öffentlich zugänglich.

Cloud Functions

Hier werden die Typescript Files hochgeladen und ausgeführt. In einer Liste können alle aktiven cloud functions, sowie der zugehörige Trigger angezeigt werden. Die Bilderkennung wird innerhalb der *storageUploadFinalized* Funktion aufgerufen. Diese startet, wenn ein neues Element im zugehörigen Bucket im Firebase Storage erkannt wird.

Ausserdem können die Konsolenlogs hier gelesen werden. Während der Entwicklung ist dies ein wichtiges Tool, da ansonsten wenig Interaktion oder gar Debugging auf Firebase möglich ist.

Einzelne cloud functions können mit dem folgenden Command auf den Firebase Server deployed werden:

```
firebase deploy --only functions:<Name der cloud function>
```

Das Deployment dauert allerdings jeweils ungefähr drei Minuten, was die Entwicklungszeit verlängert.

3.4.3 Firebase Emulator

Mit dem Firebase Emulator können cloud functions lokal ausgeführt und getestet werden. Gestartet wird der Emulator vom functions-Ordner aus mit dem bereits definierten Node Befehl:

```
npm run serve
```

Nun können die Funktionen, welche im index.ts file als cloud functions definiert sind, per http Aufruf ausgeführt werden.

Es gibt allerdings eine grosse Einschränkung für die Funktionen, welche so ausgeführt werden können. Zugriffe auf die Cloud Vision API können nicht vom Emulator aus geschehen. Diese können nur online auf einem Firebase Server starten, da Resourcen von Google direkt benötigt werden. Somit konnte der Emulator leider für den Grossteil des Projekts nicht benutzt werden.

3.4.4 Postman

Postman ermöglicht das Erstellen von komplexen http Aufrufen. Relevant ist vor allem die Autorisierung, wofür uns Trihow eine Vorlage gesendet hat. Den Trigger für die *storageUploadFinalized* können wir damit zwar nicht ausführen, aber es ist möglich, die Bilderkennung von einer separaten, http getriggerten cloud function aus zu starten.

3.5 Qualitätsmassnahmen

Zeitraum	Massnahme	Ziel der Massnahme
Während gesamtem Projekt	Coding Guidelines einhalten	lesbarer Code, welcher mit dem bestehenden übereinstimmt
Ab Meilenstein 2	Integration Tests	Die Güte der Handschrift- und Objekterkennung wird überprüft
Vor Projektende	Acceptance Tests	Mit Trihow überprüfen, ob die Anforderungen erfüllt wurden

3.5.1 Dokumentation

Der Code wird gemäss dem *TSDoc* Standart dokumentiert. Daraus wird schliesslich mittels *TypeDoc* eine leserliche Dokumentation generiert.

3.5.2 Code Style Guidelines

Die Standard Code Style Guidelines von React / Typescript werden eingehalten.

3.5.3 Code Reviews

Der erstellte Code wird jeweils von den beiden Teilnehmenden überprüft. Wenn ein Feature Branch fertig ist, wird der Code vor dem Zurückmergen auf den Research Branch von Loana Albisser überprüft.

3.5.4 Testen

Unit Testing

Automatisierte Unittests werden keine geschrieben, da das Testing mit Firebase relativ kompiliziert ist und auch fast keine bestehenden Unittests als Referenzen vorhanden sind. Dies wurde auch so von Trihow bestätigt. Eine weitere Bemerkung seitens Trihow war, bei vorhandener Zeit sich eher auf die besprochenen Erweiterungen zu konzentrieren als das Unittesting. Aus diesen Gründen werden ausschliesslich die nachfolgenden manuellen Tests durchgeführt.

Integrationstest

Die Integrationstests werden manuell ausgeführt. Mit dem Hochladen von Post-it Fotos wird das gesamte erstellte Projekt durchlaufen und somit auch überprüft.

Acceptance Tests

Der Test wird mit einer breiten Sammlung von Post-it Bildern, welche von Trihow zur Verfügung gestellt werden, durchgeführt. Dabei wird die Korrektheit der gelesenen Texte und der Farbe getestet.

Für die Lösung mit AutoML ist es wichtig, dass die Bilder für den Test nicht schon in der Sammlung des Learnings waren. Deshalb werden eigene Bilder für den Acceptance Test erstellt.

Systemtest

Da die bestehende Trihow Software nur um ein Modul erweitert wird, sind keine separaten Systemtests nötig.

KAPITEL 4

Softwareimplementierung

4.1 Einführung

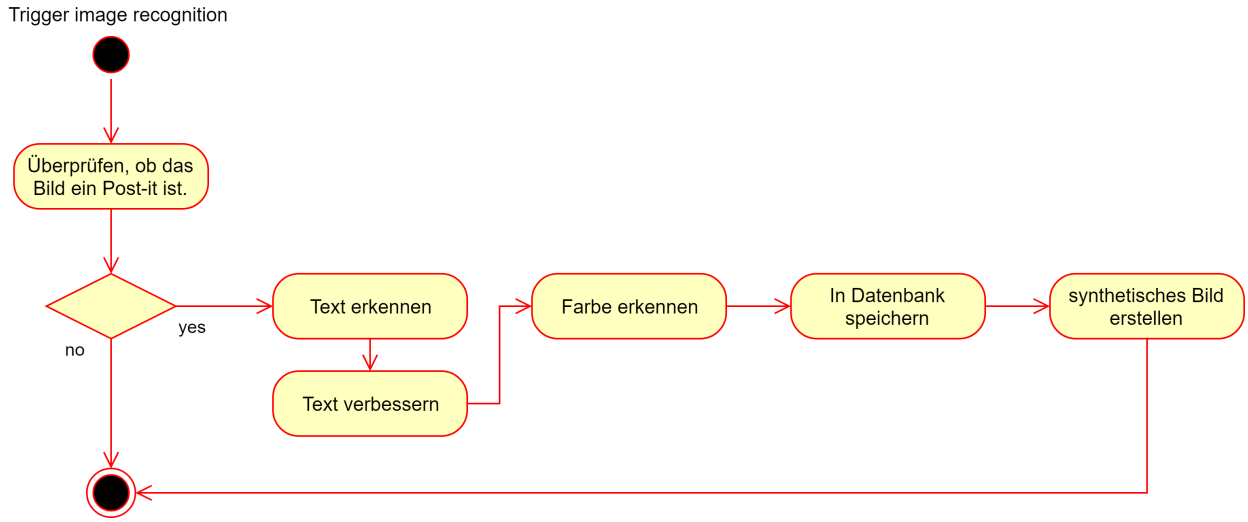
4.1.1 Zweck

In diesem Abschnitt werden die wichtigsten Schritte der Implementierung genauer erläutert.

4.1.2 Referenzen

- Google Vision API
- Google AutoML
- Cloud Functions
- Cloud Firestore
- Cloud Storage
- Google Vision API response object
- nspell package
- dictionary-de-ch
- temp package
- convert-svg-to-jpeg package

4.2 Schritte der Bilderkennung



Die Schritte der Bilderkennung werden im Folgenden einzeln erklärt.

4.2.1 Überprüfung, ob das Bild ein Post-it ist

Schnittstellen

- *Input:* Der Link auf das gespeicherte File im Bucket.
- *Output:* Ein Boolean, der aussagt, ob es als Post-it erkannt wurde.

Ausserdem werden während dem Entscheidungsprozess die erkannten Labels mit dem dazugehörigen Score in der Datenbank gespeichert.

Externe Klassen

- google-cloud/storage
- google-cloud/vision
- path
- os

Ablauf

Für die Überprüfung des Bildes gibt es zwei Wege. Es können die erkannten Labels der Vision API verwendet werden oder mithilfe von AutoML ein eigenes Machine Learning Modell erstellt werden, welches diese Unterscheidung vornimmt.

Vision API

Mithilfe der autoID-Methode wird ein temporärer Pfad erstellt auf welchen anschliessend das Bild geladen wird. Somit können die Methoden der Vision API ausgeführt werden, ohne auf die Zugriffsberechtigungen im ursprünglichen Bucket zu achten.

```
const tempPath = path.join(os.tmpdir(), autoID(element.length - 1))
await bucket.file(element).download({ destination: tempPath })
```

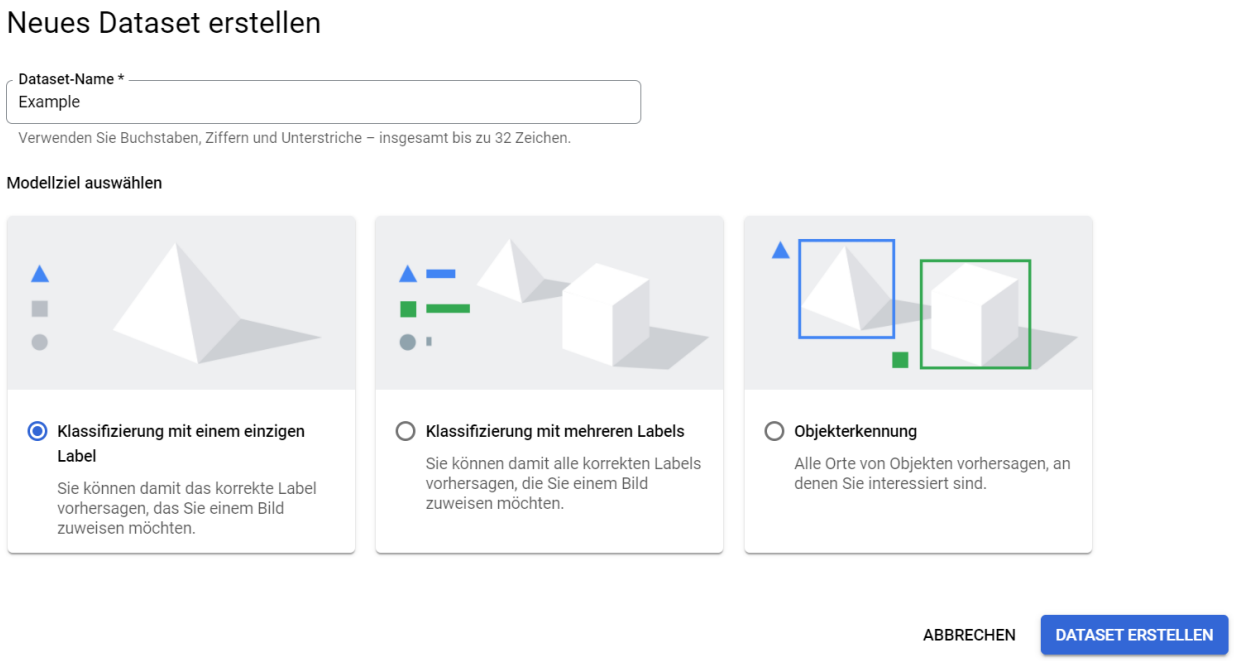
Die Anwendung der Methode zur label detection ist trivial:

```
const [result] = await client.labelDetection(originalTempPath)
const labels = result.labelAnnotations
```

Im resultierenden Array sind die zehn erkannten Labels mit dem höchsten score aufgeführt. Um die Toleranz möglichst hoch zu halten, wurde mit Trihow entschieden, nicht nur Bilder zuzulassen, bei welchen das Label „Post-it note“ erkannt wurde, sondern auch jene, die sowohl „Rectangle“ als auch „Handwriting“ enthalten.

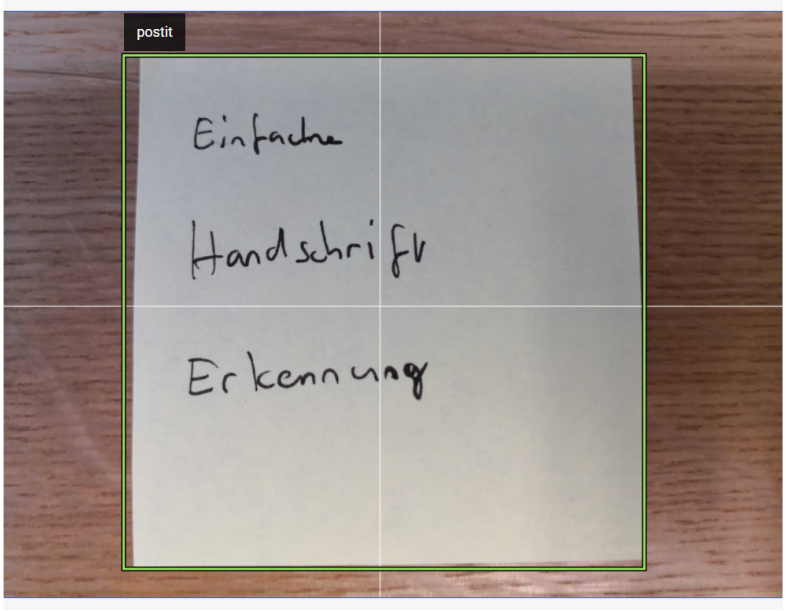
AutoML

Bei AutoML gibt es insgesamt drei verschiedene Möglichkeiten, ein eigenes Machine-Learning Modell zu trainieren. Jede der Varianten kann sowohl über das GUI als auch über die verwendete Programmiersprache und der respektiven AutoML Library bedient werden. Da die Modellvarianten fürs Erste nur für Testzwecke eingesetzt werden sollen, wird für die Bedienung hauptsächlich das GUI benutzt. Aus Neugier wurde trotzdem versucht, die Implementierung in TypeScript zu verwirklichen, aber es wurde schnell klar, dass der Zeitaufwand sich nicht lohnt.



Da ein hochgeladenes Bild ausschliesslich das Post-it enthalten sollte, wurde die Klassifizierung mit mehreren Labels nicht weiter in Betracht gezogen.

Angefangen wurde mit der Objekterkennung. Dazu wurde das Modell mit 25 Post-it Bildern trainiert, wobei auf jedem Bild die genaue Position des Post-it Zettels mit einem Label „postit“ markiert wurde.



Leider musste festgestellt werden, dass für Post-it Erkennung die Objekterkennung nicht geeignet ist. Die Trainings- und Testresultate waren sehr schlecht. Tatsächlich waren die Trainingsresultate sogar so schlecht und willkürlich, dass nicht mal die Auswertung einen Sinn ergab.

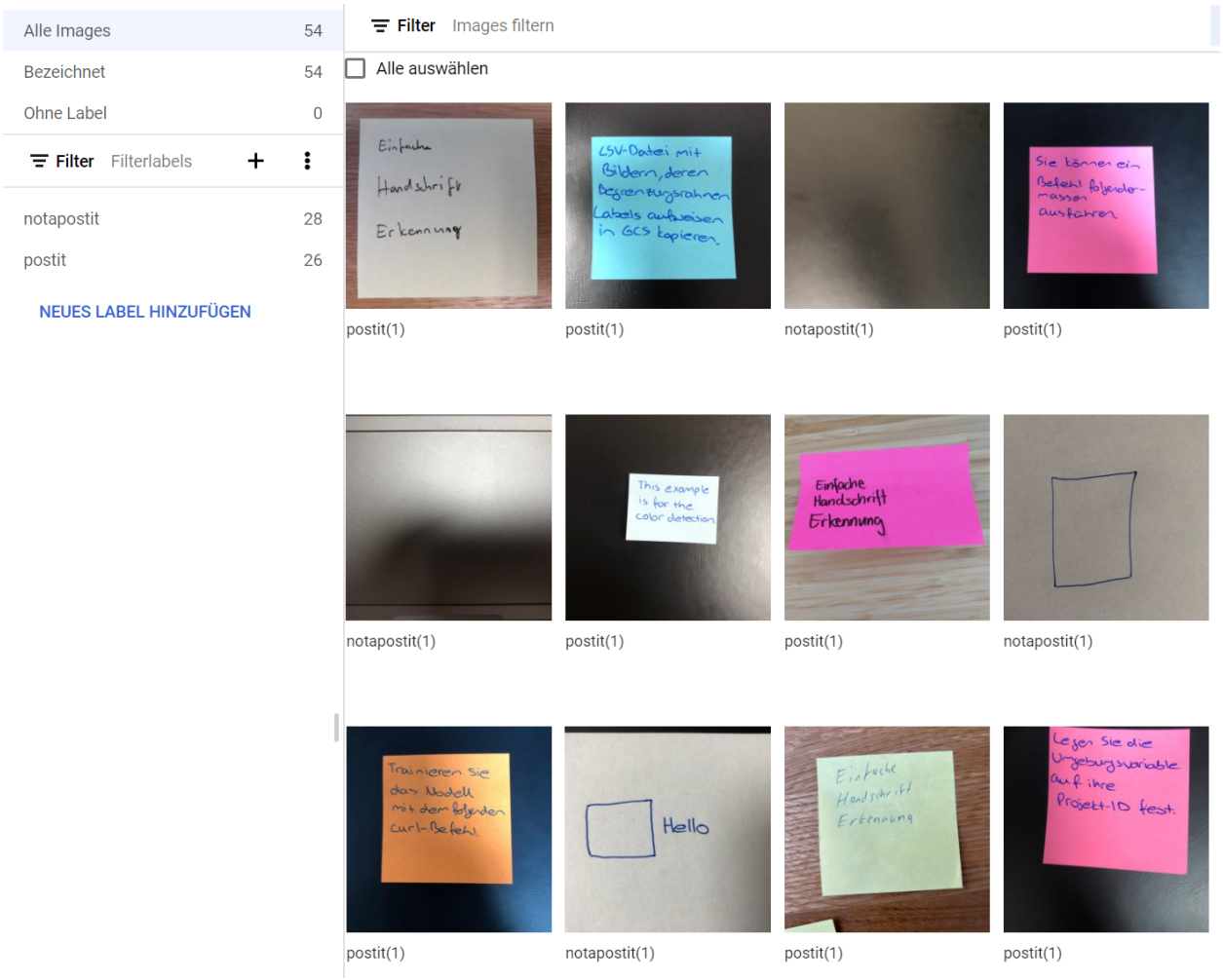
Das nachfolgende Bild zeigt die Trainingsresultate dieser Modellvariante.

- *Precision:* Modelle mit hoher Precision erzeugen weniger falsch positive Ergebnisse.
- *Recall:* Modelle mit hohem Recall erzeugen weniger falsch negative Ergebnisse.

Images gesamt	20
Testelemente	5
Objekte gesamt	5
Durchschnittliches Objekt-zu-Bild-Verhältnis	1
Precision	2,05 %
Recall	60 %

Es ist möglich, dass sich die Resultate mit mehr Trainingsbildern verbessern würden, aber die Vermutung liegt nahe, dass die Objekterkennung Probleme mit Bildern von „zweidimensionalen“ Objekten hat. Leider kann diese These nicht mit den vorhandenen Ressourcen überprüft werden.

Nun zur Variante der Klassifizierung mit einem einzigen Label. Wie sich herausgestellt hat, ist diese Variante am besten geeignet für die Post-it Erkennung. Für dieses Training wurden 54 Bilder verwendet.



Dabei wurden 26 davon mit dem Label „postit“ und 28 davon mit dem Label „notapostit“ markiert. Zu beachten ist hier die Verwendung der „Negativ“-Bilder und Labels. Durch diese Unterscheidung der Bilder wird das Modell um einiges robuster und kann dadurch mit hoher Sicherheit die Vorhersage für ein Bild treffen.

Das nachfolgende Bild zeigt die Trainingsresultate dieser Modellvariante.

Images gesamt	46
Testelemente	8
Precision	100 %
Recall	100 %

Jedes Bild, das mit diesem Modell getestet wurde, konnte mit 99% Sicherheit entweder als Post-it erkannt oder abgelehnt werden.

Entscheidung

Bei einem Testdurchlauf mit 38 Fotos von typischen Post-its erkannten beide Varianten alle Fotos korrekt und mit hoher Sicherheit. Aufgrund der erhöhten Kosten der AutoML Variante, ohne grosse Leistungssteigerung, hat sich Trihow für die Variante mit der Vision API entschieden.

4.2.2 Texterkennung

Schnittstellen

- *Input:* Der Link auf das gespeicherte File am temporären Ort
- *Output:* Ein String mit dem erkannten Text und den zugehörigen Metadaten

Zur Nachvollzierbarkeit werden die Text Annotations komplett auch in der Datenbank gespeichert.

externe Klassen

- google-cloud/vision

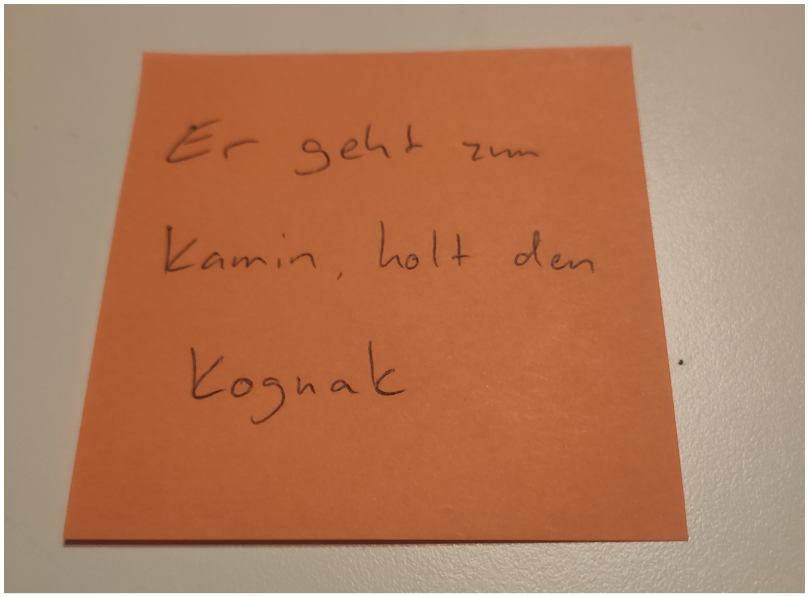
Ablauf

Die Texterkennung mithilfe der Google Vision API kann auf zwei verschiedene Arten durchgeführt werden.

- **TEXT_DETECTION** extrahiert kleine Mengen Text aus einem Bild.
- **DOCUMENT_TEXT_DETECTION** ist optimiert, um grössere Textmengen und auch einzelne Paragraphen zu extrahieren.

Bei einem Test mit 38 Test Post-its hat die DOCUMENT_TEXT_DETECTION Variante deutlich besser abgeschnitten, weshalb diese gewählt wurde.

Allerdings ist die Texterkennung auch mit dieser Variante noch alles andere als perfekt. So wurde aus dem folgenden Bild der nachstehende Text gelesen:



```
Er geht
+
Kamin, holt den
Kognak
```

Dass zusätzliche Zeilenumbrüche eingefügt wurden, kommt bei rund der Hälfte der getesteten Post-its vor. Um den Grund dafür zu finden, muss erst verstanden werden, wie ein Text mit der Vision API strukturiert wird.

```
const [result] = await client.documentTextDetection(filePath)
```

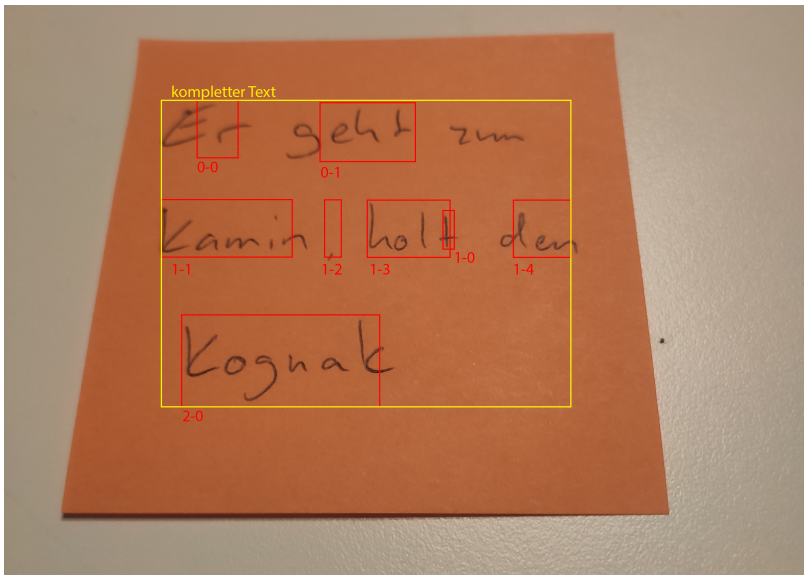
Für die Texterkennung ist das **fullTextAnnotation** Objekt interessant, welches im Resultat enthalten ist. Nebst dem zusammengesetzten Text sind darin auch die ursprünglichen Elemente stark verschachtelt aufgeführt.

pages -> blocks -> paragraphs -> words -> symbols

Dies sieht für den ersten Buchstaben „E“ im Text folgendermassen aus. Für den besseren Überblick wurden die meisten, für das Verständnis uninteressanten Objekte weggelassen.

```
{
    "pages": [{
        "blocks": [{
            "paragraphs": [{
                "words": [{
                    "symbols": [{
                        "text": "E",
                        "boundingBox": {"vertizes": [{"x":1185,"y":2276}, ...]}
                    }, ...],
                    "boundingBox": {"vertizes": [{"x":1185,"y":2276}, ...]}
                }, ...],
                "boundingBox": {"vertizes": [{"x":1185,"y":2276}, ...]}
            }, ...],
            "boundingBox": {"vertizes": [{"x":1185,"y":2276}, ...]}
        }, ...],
    }, ...],
    "text": "Er geht\n+\nKamin, holt den\nKognak\n"
}
```

Zu allen Blöcken, Paragraphen, Wörtern und Zeichen gibt es also ein mit vier Punkten definiertes Polygon. Diese Koordinaten der Wörter wurden nachfolgend auf dem Bild gekennzeichnet.



Als erstes ist erkennbar, dass die Rechtecke nicht immer sinnvoll scheinen. So ist das „g“ im Rechteck um „geht“ nicht enthalten, es wird aber trotzdem erkannt. Ebenfalls wird klar, dass das „zum“ am rechten oberen Rand gar nicht erkannt wurde, was sich durch die recht unleserliche Schrift erklären lässt. Spannend ist aber vor allem, dass jetzt erkennbar ist, woher das „+“ im Resultat kommt. Auf der Zeile 2 wurde der Buchstabe „t“ als eigenes Wort interpretiert und an den Anfang der Zeile geschrieben. Warum dies geschieht, lässt sich nicht erklären.

Durch diese Aufzeichnung der Koordinaten zeigt sich, dass sich das Resultat aus der Texterkennung verbessern lässt, wenn die Koordinaten der einzelnen Wörter überprüft und sie so in die richtige Reihenfolge gebracht werden. Ausserdem sollte es möglich sein zu erkennen, wenn ein Buchstabe eines Wortes als separates Wort interpretiert wird. Somit können diese falsch erkannten Elemente gelöscht werden.

4.2.3 Textverbesserung

Schnittstellen

- *Input:* Die von der vision API erkannten Textdaten.
- *Output:* Ein String mit dem verbesserten Text.

Zur Nachvollziehbarkeit werden die Ergebnisse der einzelnen Schritte ebenfalls in der Datenbank gespeichert.

externe Klassen

- google-cloud/vision
- nspell
- dictionary-en-us
- dictionary-de-ch
- dictionary-fr
- dictionary-it
- dictionary-es

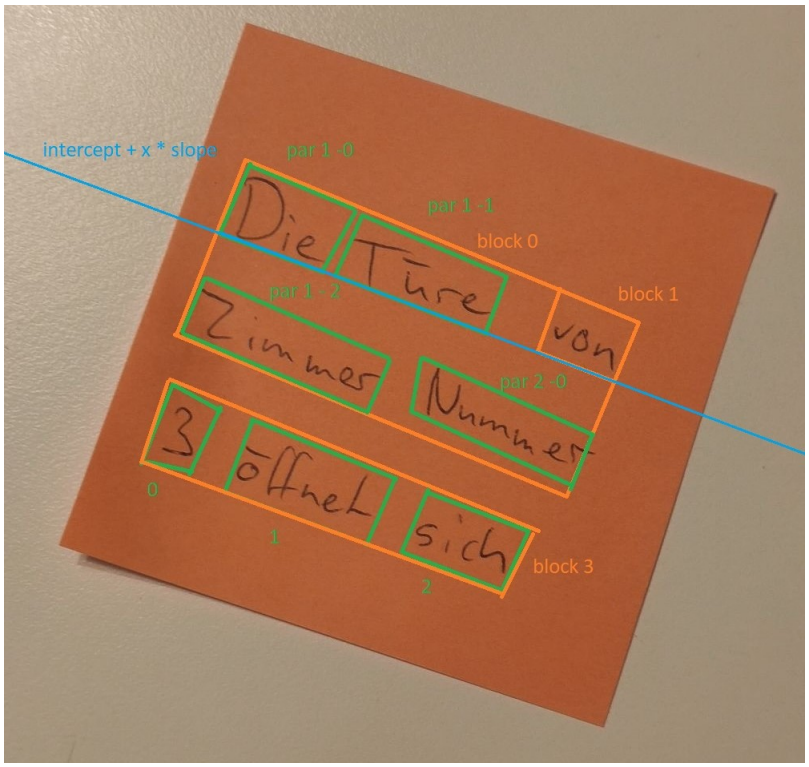
Ordering

Die Idee des Ordering ist es, in einer Schleife jeweils die oberste Zeile zu entfernen, diese nach der horizontalen Reihenfolge zu sortieren und anschliessend als String zu speichern. Um einer Endlosschleife bei fehlerhaftem Laufzeitverhalten vorzubeugen, wurde eine maxRows-Konstante eingefügt.

```
let text: Array<string> = []
let counter = 0
const maxRows = 20
while (words.length > 0 && counter < maxRows) {
    counter++
    const result = this.removeTopRow(words)
    words = result.otherWords
    this.orderRow(result.upmostLine)
    for (let word of result.upmostLine) {
        text.push(word.text)
    }
}
```

Herauszufinden, welche Wörter auf der gleichen Zeile sind, gestaltet sich etwas komplizierter, da davon ausgegangen werden muss, dass der Text im Bild nicht perfekt horizontal ist. Die Korrektur sollte auch funktionieren, wenn das Bild des Post-its schräg aufgenommen wurde.

Dafür werden die Wörter in einem ersten Schritt nach der Y-Komponente der oberen, linken Ecke sortiert. Das oberste Wort steht somit an erster Stelle. Von diesem Wort wird die lineare mathematische Funktion, welche durch die beiden unteren Ecken des Wortes gehen gefunden. *Im folgenden blau aufgezeichnet.*



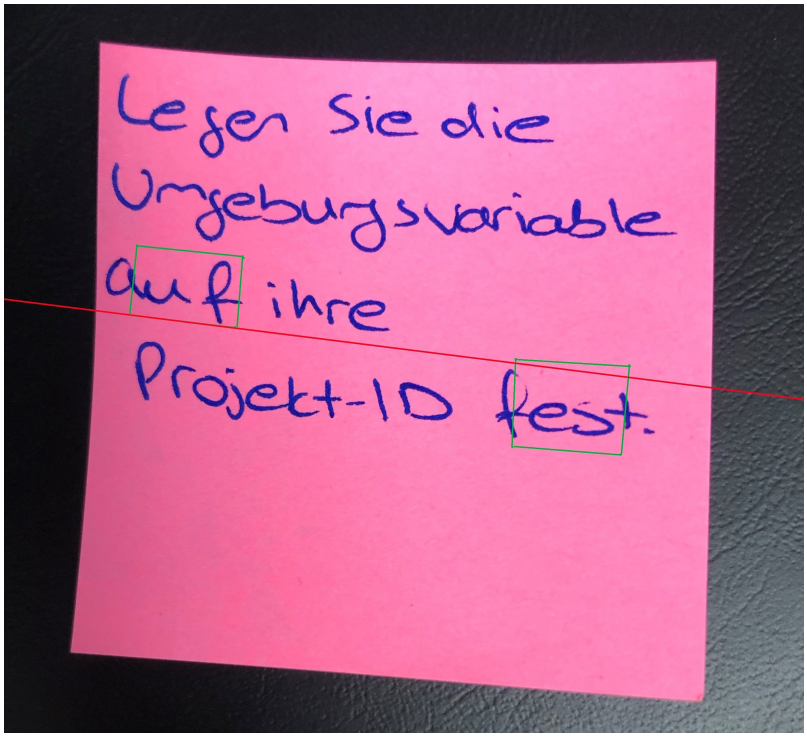
Alle Wörter, bei denen mindestens eine der oberen beiden Ecken oberhalb dieser Linie liegen, zählen zur ersten Zeile. Diese können danach direkt nach ihrem X-Wert sortiert werden.

Test

Die Qualität dieser Funktion wurde mittels des *Herr der Ringe Tests* (Kapitel 5.3.1) geprüft. Das Resultat ist wie folgt:

- *Ordering initial richtig:* 17
- *Ordering verbessert:* 7
- *Ordering verschlechtert:* 1

Diese Resultate sehen vielversprechend aus. Bei Post-its, welche von anderen Personen geschrieben wurden, funktionierte die Funktion allerdings nicht mehr gleich gut. In mehreren Fällen wurde das Ergebnis gar verschlechtert.



In diesem Beispiel wurde das Wort „fest“ vor das Wort „Projekt-ID“ verschoben. Die gezeichnete Linie zeigt, dass es fälschlicherweise zur oberen Zeile gezählt wurde.

Schlussfolgerung

Wenn die Zeilen zu nahe aufeinander sind, versagt das Ordering. Um das Ergebnis zu verbessern, wurde die „isInLine“ Funktion angepasst. Dafür wurde ein Threshold (Grenzwert) Parameter eingefügt. Dieser bestimmt, um wie viele Pixel die oberen Koordinaten eines Wortes oberhalb der Linie sein müssen. Da die Bilder eine unterschiedliche Auflösung haben können, ist dieser Wert sinnvollerweise relativ und von der Höhe eines Wortes abhängig. Diese Variante wurde mit der halben Höhe des ersten Wortes *(threshold = (y2 - y1) / 2)* als Grenzwert getestet.

Das Resultat mit dem *Herr der Ringe Test* ist wie erwartet schlechter als zuvor:

- *Ordering initial richtig:* 17
- *Ordering verbessert:* 5
- *Ordering verschlechtert:* 3

Allerdings wurde das vorherig fehlerhafte Beispiel dieses Mal richtig geordnet.

Ein anderer Ansatz besteht darin, nicht die beiden oberen Ecken eines Wortes, sondern die vertikalen Mittelpunkte rechts und links des Wortes mit der linearen Funktion zu vergleichen.

((y0 + y3) / 2 < linearFunction ODER (y1 + y2) / 2 < linearFunction)

Zur Erinnerung: Da die Koordinaten vom Punkt oben links gezählt werden, müssen die Werte kleiner als die lineare Funktion sein.

Spelling

In einem grossen Teil der gelesenen Post-its wurde mindestens ein Wort nicht korrekt erkannt. Es fehlten Buchstaben oder falsche Buchstaben wurden eingefügt. Um diesem Problem entgegenzuwirken, wurde eine spellCheck-Funktion implementiert.

Kern dieser Funktion sind das Node Modul *„nspell“*, sowie verschiedene *„dictionaries“*, welche die benötigten Daten beinhalten. Diese Module wurden gewählt, da sie bereits im Projekt enthalten waren und somit keine neuen Abhängigkeiten hinzugefügt werden mussten.

In einem ersten Schritt wird der *dictionary* mit der korrekten Sprache gewählt. Dafür liefert die Vision API einen *language code* gemäss ISO-639-1. Anschliessend werden alle Wörter einzeln von *nspell* überprüft und falls nötig durch einen Vorschlag ersetzt.

```
const nspell = require('nspell')

return new Promise((resolve, reject) => {
    dictionary(function (err: any, dict: any) {
        const spell = nspell(dict)
        for (let [index, word] of clonedWords.entries()) {
            if (SpellCheck.isAWord(word) && !spell.correct(word)) {
                const suggestions = spell.suggest(word)
                if (suggestions[0]) {
                    clonedWords[index] = suggestions[0]
                }
            }
        }
        resolve(clonedWords)
    })
})
```

Der Ablauf muss in einem Promise geschehen, da der *dictionary* seine Daten zuerst lädt und erst anschliessend die übergebene Funktion als Callback ausführt.

Test

Es wurden die selben Testdaten wie beim Ordering verwendet, um die Funktion zu überprüfen. Das Resultat war jedoch ernüchternd:

- Falsche Wörter: 38
- Verbesserte Wörter: 5
- Verschlechterte Wörter: 13

Schlussfolgerung

Dieses Ergebnis zeigt deutlich, dass die gewählte Strategie oft unbrauchbar ist. Um die Texte mit einer Rechtschreibeprüfung zu verbessern, müssen sie wohl komplett in ein System mit *natural language processing* eingelesen werden. Ein solches System könnte ebenfalls überprüfen, welche Wörter in den Kontext passen würden. Nichtsdestotrotz wird der korrigierte Text in die Firestore Db gespeichert. Damit kann er bei der Bildgenerierung als zusätzliche Variante gewählt werden.

4.3 Schritte der Bildgenerierung

4.3.1 Bild-Synthese

Nun kommt der Teil, in dem die bisher gesammelten Daten genutzt werden, um ein digitales Post-it im JPEG-Bildformat zu generieren. Die ursprüngliche Aufgabe war die Generierung des Bildes im PNG-Bildformat, aber Trihow hat sich später umentschieden.

Schnittstellen

- *Input:* Der im Post-it erkannte Text und Farbe.
- *Output:* Der temporäre Pfad zum generierten Post-it im JPEG-Bildformat.

externe Klassen

- temp
- convert-svg-to-jpeg

SVG-Generierung

Durch SVG-Strings kann ein digitales Post-it generiert werden, welches vollkommen editierbar ist. Dazu werden lediglich die notwendigen Parameter der createSvg-Funktion übergeben. Als Rückgabewert gibt diese Funktion den ganzen SVG-String zurück.

Im Grunde beinhaltet die Implementierung dieser Funktion nur die korrekte Verwendung der eingegebenen Parameter innerhalb des SVG-Strings. Die konkrete Definition des Strings war aber nicht so trivial. Es ist nicht immer klar wann welche Tags verwendet werden sollen und es gibt sehr viele Möglichkeiten für die Visualisierung.

Alte Version

```
let svgPrefix = `
    <svg xmlns="http://www.w3.org/2000/svg" width="500" height="500">
        <g xmlns="http://www.w3.org/2000/svg">
            <rect width='500' height='500' x='0' y='0' opacity="undefined"
                stroke="#000" fill="rgb(${color.red},${color.green},${color.blue})"/>
            <text x="50" y="50" font-size="28">`
                let i = 4
                while (i < splittedText.length) {
                    svgPrefix += `<tspan  dy="2em" x="50">${
                        splittedText.slice(i - 4, i + 1).join(' ')}</tspan>`
```

(Fortsetzung auf der nächsten Seite)

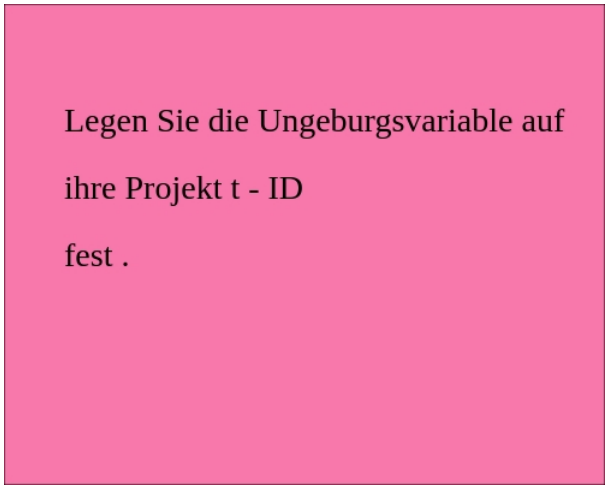
(Fortsetzung der vorherigen Seite)

```
                    i += 5
                }

                svgPrefix += `<tspan  dy="2em" x="50">${
                    splittedText.slice(i - 4, splittedText.length).join(' ')}</tspan>`

return svgPrefix + `</text></g></svg>`
```

Die Dimension des SVG-Bildes wurde fix definiert, da die echten Post-its sich auch nicht massiv in der Grösse unterscheiden. Es wäre aber auch denkbar, verschiedene Dimensionen anzubieten, was relativ leicht zu implementieren wäre.

Die Form des digitalen Post-its wurde einfachheitshalber auf ein Quadrat festgelegt. Die Farbe des zu generierenden Post-its wird mit den Ergebnissen der Farberkennung definiert, was der echten Farbe sehr nahe kommt.

Wie man oben sehen kann, wurden in der ursprünglichen Version dieser Funktion manuelle „Line-Breaks“ hinzugefügt, um den Text auf mehreren Zeilen anzuzeigen, wie es in einem echten Post-it der Fall wäre. Zu diesem Zeitpunkt konnte noch keine Möglichkeit gefunden werden, den Text automatisch auf mehrere Zeilen zu verteilen. Aus diesem Grund wird hier nach jeweils fünf Wörtern ein Line-Break hinzugefügt, um den Text auf das ganze Bild zu verteilen.



Da diese Lösung unbefriedigend war, wurde lange nach einer besseren Lösung recherchiert, um das Text-Wrapping zu automatisieren. Schlussendlich gab es doch eine Möglichkeit dies zu verwirklichen.

Aktuelle Version

```
const width = 500
const height = width
const padding = 50

const svg = `
    <svg xmlns="http://www.w3.org/2000/svg" width="${width}" height="${height}">
        <rect width='${width}' height='${height}' x='0' y='0' opacity="undefined"
            stroke="#000" fill="rgb(${color.red},${color.green},${color.blue})"/>
        <foreignObject x="${padding}" y="${padding}"
            width="${width - (padding * 2)}" height="${height - (padding * 2)}" font-
↪size="36">
            <text xmlns="http://www.w3.org/1999/xhtml">
                ${text}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            </text>
        </foreignObject>
    </svg>`

return svg
```

Die aktuelle Version der SVG-Generierung unterscheidet sich, wie oben erwähnt, vor allem im automatischen Text-Wrapping. Da es in der SVG-Spezifikation selbst keine direkte Möglichkeit gibt, dies z. B. mit dem Tag `<text>` zu verwirklichen, wurde das Tag `<foreignObject>` verwendet. Damit wird hier signalisiert, dass der folgende Text als ein (X)HTML-Element betrachtet werden soll, was wiederum das automatische Text-Wrapping zulässt.

Zudem wurde die Möglichkeit hinzugefügt, die Dimension und das Padding anzupassen. Mit Padding wird hier der Abstand vom Text zum Rand bezeichnet.

Wie man sehen kann, ist diese Version der Implementierung nicht nur verständlicher, sondern auch dynamisch erweiterbar.

Test

Der SVG-String selbst konnte mit einem Previewer direkt innerhalb der Entwicklungsumgebung betrachtet und bearbeitet werden. Dadurch musste nur sichergestellt werden, dass der korrekte Text und die korrekte Farbe dargestellt werden, was manuell erledigt wurde.

Schlussfolgerung

Die Verwendung von SVG-Strings zur Bild-Synthese war für dieses Projekt auf jeden Fall die richtige Wahl, da mit wenig Code das Post-it Bild generiert werden konnte. Zudem ist ein SVG-String so erweiterbar, dass auch komplexe Formen und Objekte dargestellt werden können.

JPEG-Generierung

In der createJpeg-Funktion wird die oben erwähnte createSvg-Funktion aufgerufen und der erhaltene SVG-String weiterverarbeitet. Mit Hilfe des Moduls „temp“ wird der String dann als eine SVG-Datei in einem temporären Verzeichnis gespeichert. Danach wird mit der Funktion convertFile aus dem Modul „convert-svg-to-jpeg“ die SVG-Datei in ein JPEG-Image konvertiert und im gleichen temporären Verzeichnis gespeichert. Schlussendlich wird der Pfad zum JPEG-Image als Rückgabewert der Funktion zurückgegeben.

```
const svg = this.createSvg(text, color)

// Automatically track and cleanup files at exit
temp.track()
let stream = temp.createWriteStream()
// stream.path contains the temporary file path for the stream
stream.write(svg)
stream.end()

return await convertFile(stream.path)
```

Test

Auch hier musste manuell sichergestellt werden, dass der Text und die Farbe korrekt dargestellt werden. Zusätzlich musste noch kontrolliert werden, ob es sich hierbei auch wirklich um eine JPEG-Datei handelt.

Schlussfolgerung

Die Generierung und Umwandlung einer SVG-Datei in eine JPEG-Datei konnte schlussendlich mit den erwähnten Hilfs-Modulen leichter gelöst werden als zunächst am Anfang gedacht. Allerdings wurde sehr viel Zeit für die Recherche aufgewendet.

4.4 Daten auf Firebase hochladen

Mit der createAndUploadJpeg-Funktion wird, wie der Name schon hindeutet, das digitale JPEG-Image des Post-its erzeugt und auf Firebase hochgeladen. Im Anschluss werden zusätzlich die Metadaten hochgeladen. Nachfolgend wird die createAndUploadJpeg-Funktion in zwei Abschnitten näher erläutert.

Zu beachten: In der Funktion wird der gleiche Code für drei verschiedene Versionen des Textes (initial, ordered, spellChecked) verwendet. Aber in den Abschnitten unten wird jeweils die Implementierung von nur einer Version besprochen.

4.4.1 Schnittstellen

- *Input:* Der Pfad zum Originalbild im Storage, der temporäre Pfad zum Benutzen des Bildes, die Referenz auf das Dokument im Firestore und die respektive Upoload-ID.

4.4.2 externe Klassen

- google-cloud/storage
- google-cloud/vision

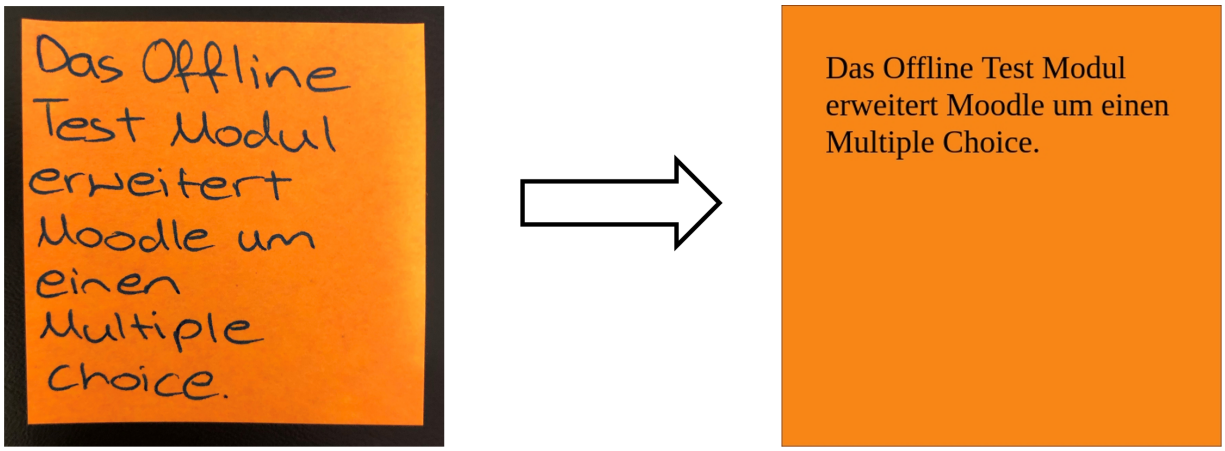
4.4.3 Storage

In diesem Abschnitt wird der erste Teil der createAndUploadJpeg-Funktion besprochen. Nämlich, wie das generierte Bild in der Firebase Storage im entsprechendem Bucket abgespeichert werden kann.

```
const bucket = gcs.bucket('gs://trihow-research.appspot.com')

const updated_document = await this.getDocumentFromDb(uploadID)
const colors = await TestColorRecognition.testColorRecognition(tempPath)

const tempJpegPathInitial = await CreateImage.createJpeg(
    updated_document.assetsImageRecognition['11_initial'],
    colors[0].color,
)

const pathForJpeg = objectPath.substring(
            0,
            objectPath.length -
                (path.parse(objectPath).name + path.parse(objectPath).ext)
                    .length,
        )
const destination1 = pathForJpeg + 'initial' + '.jpeg'

await bucket.upload(tempJpegPathInitial, {
            public: false,
            destination: destination1,
        })
```

Als erstes wird der Bucket verlinkt, in dem das Bild später gespeichert werden soll. Danach werden die benötigten Daten für die Bild-Synthese abgefragt und der weiter oben erwähnten createJpeg-Funktion übergegeben. Diese gibt einen temporären Pfad zum generierten JPEG-Bild des Post-its zurück, welcher für den Upload benötigt wird. Der richtige Speicherort im Bucket, wohin das Bild dann hochgeladen werden soll, wird einfach vom Pfad des Originalbildes extrahiert. Zum Schluss muss nur noch der vollständige Pfad inklusive des Namens und der „.jpeg“-Endung des Bildes definiert und zusammen mit dem Pfad zum erstellten temporären Bild der upload-Funktion dem Bucket übergeben werden.

Ergebnis einer erfolgreichen Post-it Erkennung und JPEG-Bild Generierung:

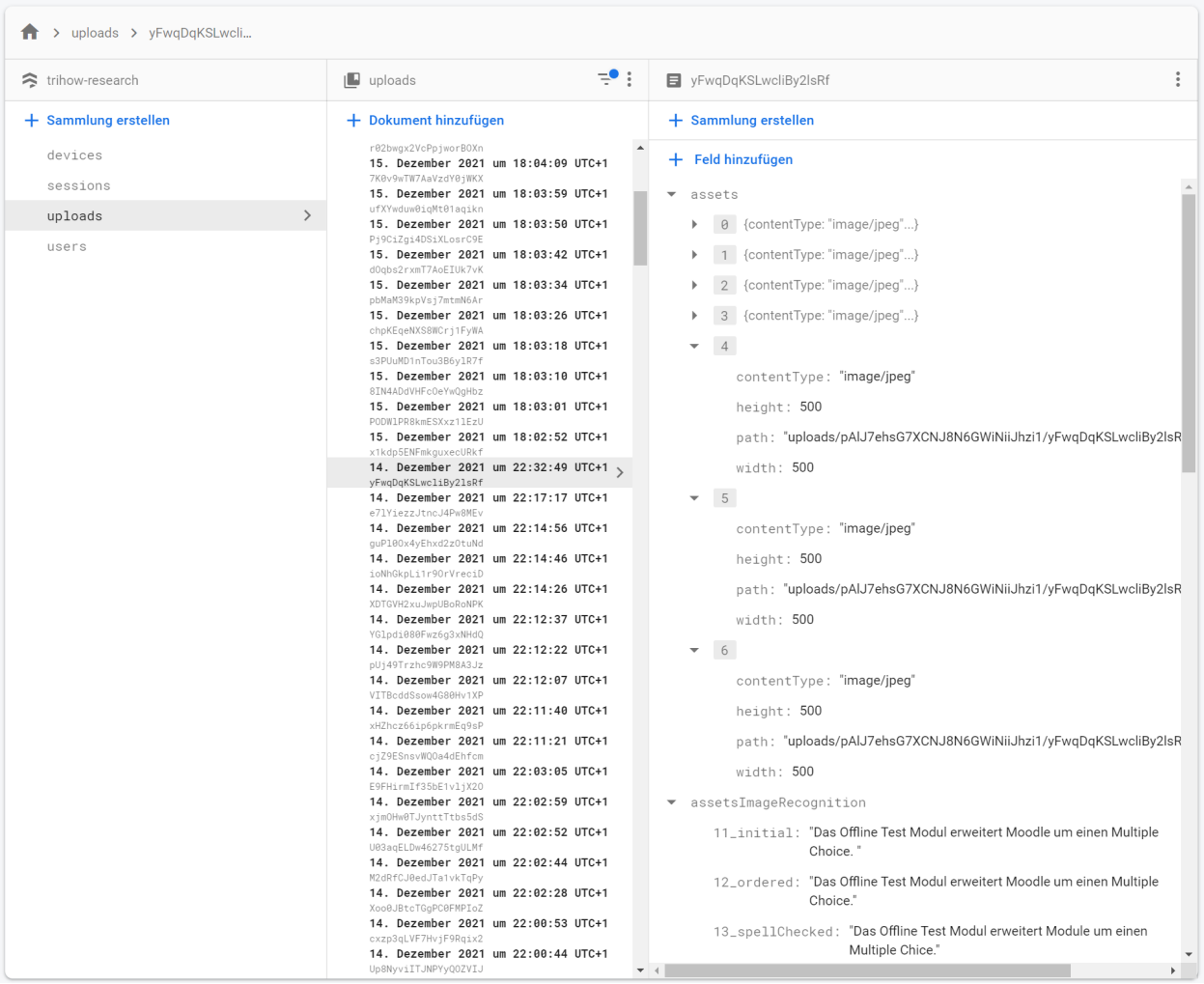


4.4.4 Firestore Database

In diesem Abschnitt wird der zweite Teil der createAndUploadJpeg-Funktion vorgestellt. Es wird gezeigt, wie die Metadaten des generierten Bildes in die Firestore Database hochgeladen werden.

```
await documentReference.update({
            assets: admin.firestore.FieldValue.arrayUnion(
                {
                    contentType: 'image/jpeg',
                    path: destination1,
                    width: 500,
                    height: 500,
                }
            )
        })
```

Da diese Funktion die Referenz auf das Dokument des Originalbildes im Firestore bekommen hat, kann direkt die update-Funktion darauf aufgerufen werden. Dazu müssen lediglich die zu aktualisierenden bzw. neu hinzuzufügenden Felder mit den entsprechenden Werten übergeben werden. Da das Feld „assets“ im Dokument schon als ein Array existiert, muss diese spezielle arrayUnion-Funktion benutzt werden, um die existierenden Daten des Arrays nicht komplett zu überschreiben.



4.4.5 Test

Es musste manuell sichergestellt werden, dass die generierten JPEG-Bilder am richtigen Ort im Storage gespeichert werden und dass die Metadaten tatsächlich dem Dokument zugewiesen, wie auch korrekt dargestellt werden.

4.4.6 Schlussfolgerung

Auch wenn es am Anfang Schwierigkeiten in der Verwendung der Firebase-Umgebung gab, konnten schlussendlich alle Aufgaben wie gewünscht erledigt werden. Es hat eine Weile gedauert bis die entsprechenden Funktionen / Klassen gefunden und verstanden wurden, aber danach verlief die Arbeit mit Firebase problemlos.

KAPITEL 5

Integrationstest

5.1 Einführung

5.1.1 Zweck

Der Zweck dieses Abschnitts ist es, die durchgeführten Integrationstests genau zu beschreiben und somit die gewonnenen Resultate nachvollziehbar zu machen.

5.1.2 Referenzen

- Google Vision API
- Cloud Functions
- Cloud Firestore
- Cloud Storage
- Postman

5.2 Methoden

Bei der Entwicklung war es häufig nötig, die gesamte Pipeline der Trihow Software zu testen. Die Resultate dieser Tests konnten direkt auf die Firestore Db geschrieben werden. Alternativ wurden Konsolenlogs verwendet, um den Ablauf im kleineren Rahmen aufzuzeigen. Dies war insbesondere nötig, da bei der Entwicklung auf Firebase Komfortfunktionen wie Debugging wegfielen.

5.2.1 Manueller Upload der Bilder

Zu Beginn wurden die Bilder von Post-its manuell mithilfe der Trihow Research App auf den Firebase Server geladen. Bei jedem Upload wurde automatisch die gesamte Pipeline durchlaufen und die gesammelten Logs sowie die erstellten Datenbankeinträge konnten über das Browserinterface gelesen werden.

Der Vorteil dieser Methode war, dass dafür keine Systemkenntnisse nötig waren. Allerdings dauerte ein Test auch relativ lange, da ein Smartphone, sowie ein Rechner verwendet werden mussten. Ausserdem musste bei jedem Test ein Bild auf Firebase geladen werden, was eine gewisse Zeit beanspruchte und unsinnigerweise Speicher auf dem Server verbrauchte.

5.2.2 Die TestMlPerformance-Klasse

Aus dem Vorgängerprojekt war bereits ein File vorhanden, welches verschiedene Integrationstests beinhaltete. Diese konnten zwar nicht direkt verwendet werden, boten sich aber als Vorlage an, um eigene Tests zu schreiben. Um solche Funktionen auszuführen, müssen sie im index.ts File als Cloud Function mit https-trigger definiert werden. Danach können sie über ein Tool wie Postman ausgeführt werden.

Der generierte https-request muss allerdings um einige Umgebungsvariablen erweitert werden. Ansonsten schlägt die Autorisierung fehl. Dafür wurde von Trihow ein JSON File mit den folgenden Values zur Verfügung gestellt:

```
"values": [
    {
        "key": "project_id",
        "value": "trihow-research",
        "enabled": true
    },
    {
        "key": "auth_email",
        "value": "{YOUR_EMAIL}",
        "enabled": true
    },
    {
        "key": "auth_password",
        "value": "{YOUR_PASSWORD}",
        "enabled": true
    },
    {
        "key": "id_token",
        "value": "{YOUR_TOKEN}",
        "enabled": true
    },
    {
        "key": "auth_uid",
        "value": "{YOUR_UID}",
        "enabled": true
    }
]
```

testImageRecognition

Die statische TestImageRecognition Methode wird genutzt, um alle Schritte der Bilderkennung auszuführen. Dafür wird in der Methode der Pfad zum Bild definiert, mit dem diese Schritte ausgeführt werden sollen. Wichtig ist, dass dieses Bild bereits einmal mit der Trihow Research App hochgeladen wurde. Somit wird garantiert, dass auch in der Firestore Db ein Eintrag für das Bild vorhanden ist.

```
public static async testImageRecognition() {
    const bucket = gcs.bucket('gs://trihow-research.appspot.com')
    const [metadata] = await bucket
        .file('uploads/snXlZDUHXThGpsIHXPNbwY0NvjM2/rSYPjIduQ1bsV9Z1a3hG/BSshXc8c0.jpg
↪')
        .getMetadata()
    await ImageRecognition.imageRecognition(metadata)
}
```

testImageRecognitionMultiple

Um die Bilderkennung wiederholt auf mehreren Fotos auszuführen, wurde die testImageRecognitionMultiple-Methode geschrieben. Diese sucht sich alle Bilder, welche zwischen zwei definierten Zeitpunkten aufgenommen wurden und führt die Bilderkennung auf diesen Bildern aus.

Implementiert wurde dies mithilfe der Firestore Collection Funktionen. Das heisst allerdings, dass wieder nur Bilder verwendet werden, welche bereits mit der App hochgeladen wurden und somit einen Eintrag in der Firestore Db besitzen.

```
public static async testImageRecognitionMultiple() {
        const bucket = gcs.bucket('gs://trihow-research.appspot.com')
        const date1 = new Date('2021-11-29T00:00:00')
        const date2 = new Date('2021-11-30T00:00:00')
        const documents = await admin
            .firestore()
            .collection('uploads')
            .where('startTime', '>', date1)
            .where('startTime', '<', date2)
            .get()

        for (let doc of documents.docs) {
            if (doc.data().assets[0]) {
                const [metadata] = await bucket
                    .file(doc.data().assets[0].path)
                    .getMetadata()
                console.log(
                    'TEST_IMAGE_RECO: for image: ' +
                        doc.data().assets[0].path +
                        ' started',
                )
                await ImageRecognition.imageRecognition(metadata)
            }
        }
    }
```

5.3 Durchführungen

5.3.1 Herr der Ringe Test

Vorbereitung

Der Einfachheit halber wurden für diesen Test die ersten fünf Absätze des ersten Kapitels von *Herr der Ringe: Die zwei Türme* auf 25 Post-it Zetteln verteilt abgeschrieben. Jeder Zettel wurde anschliessend einzeln fotografiert. Dabei wurde darauf geachtet, dass die Bilder nicht gerade von oben, sondern auch schief aufgenommen wurden.

Nachdem diese 25 Bilder mit der Trihow Research App hochgeladen wurden, musste die testImageRecognitionMultiple-Methode angepasst werden. Die Variablen date1 und date2 wurden so erstellt, dass sie das Aufnahmedatum der Bilder umschlossen. Da der Source-Code somit verändert wurde, musste die neue Version mit dem Befehl:

```
firebase deploy --only functions:testImageRecognitionMultiple
```

auf den Firebase Server geladen werden.

Durchführung

Anschliessend konnte mithilfe von Postman ein https-request mit der vorher genannten Umgebung an die Adresse: „https://us-central1-{{project_id}}.cloudfunctions.net/testImageRecognitionMultiple“ gesendet werden.

Die Ergebnisse waren als Einträge in den Dokumenten der Firestore Db ersichtlich. Bei allen verwendeten Bildern wurden die Parameter „11_initial“, „12_ordered“ und „13_spellChecked“ hinzugefügt. Dort stand der jeweilige Text, als String formatiert. Von Hand musste nun überprüft werden, ob das Ordering und die erkannten Wörter mit dem Ausgangstext übereinstimmten.

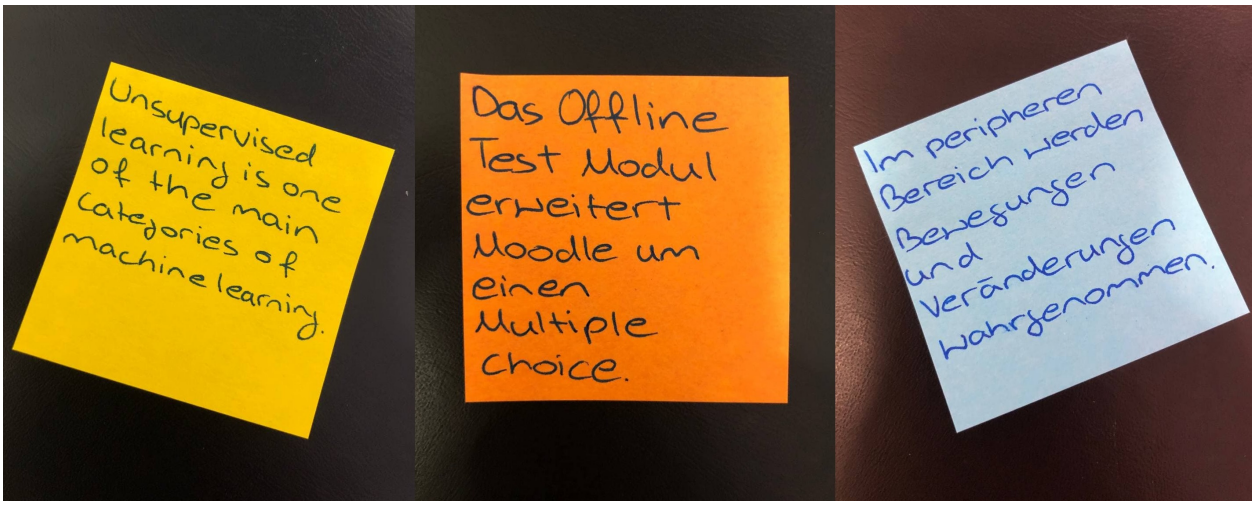
Ergebnis

Dieser Test wurde mehrmals durchgeführt, um einzelne Parameter im Bereich Ordering zu optimieren. Die Resultate sind jeweils in den Abschnitten des Kapitels 4.2.3 Textverbesserung aufgezeichnet.

5.3.2 Acceptance Test

Vorbereitung

Als Acceptance Test wurden von vier verschiedenen Personen jeweils 10 Post-its geschrieben und anschliessend fotografiert. Von den 10 Fotos pro Person wurden jeweils drei von schräg links, vier gerade und drei von schräg rechts aufgenommen.



Durchführung

Die 40 Fotos wurden mithilfe der Trihow Research App auf Firebase geladen. Bei jedem Bild wurde die Bilderkennungspipeline automatisch ausgeführt. Die Resultate wurden auf der Firestore Db in den drei Parametern: „11_initial“, „12_ordered“ und „13_spellChecked“ gespeichert. Wie beim *Herr der Ringe Test* musste das Resultat manuell überprüft werden.

Ergebnis

Die resultierenden Strings wurden auf folgende Eigenschaften überprüft, welche anschliessend aufsummiert wurden.

- Reihenfolge vorher richtig: 24
- Reihenfolge verbessert: 13
- Reihenfolge verschlechtert: 1
- Richtige Wörter: 382
- Fehlende Wörter: 0
- Zusätzliche Wörter: 14
- Falsche Wörter: 40
- Verbesserte Wörter: 9
- Verschlechterte Wörter: 11

Das Ergebnis zeigt, dass die Parameter des Orderings sinnvoll gewählt wurden. Damit hatte sich die Menge an Post-its mit korrekter Reihenfolge von **63%** auf **93%** erhöht. Die Rechtschreibeprüfung ist hingegen nach wie vor sehr unzuverlässig. Allerdings wurden bereits von der Vision API **91%** der Wörter korrekt erkannt.

Erkannte Probleme

Bei der Auswertung des Acceptance Tests wurden einige Probleme erkannt, welche in einer Weiterentwicklung der Software behoben werden sollten:

- Kleine Zahlen werden von der Rechtschreibeprüfung als Wörter interpretiert und somit durch ein willkürliches Wort ersetzt. Dies kann einfach verhindert werden, indem Zahlen vor der Rechtschreibeprüfung herausgefiltert werden.
- Bei der Erstellung des Strings aus den Wörtern werden auf beiden Seiten der Klammern jeweils Leerzeichen gesetzt. Dasselbe bei Anführungszeichen. Bei den Klammern dürfte diese Korrektur simpel sein, bei den Anführungszeichen muss hingegen herausgefunden werden, ob es sich um das erste oder das zweite Anführungszeichen handelt.
- Wörter, welche mit einem Bindestrich auf zwei Zeilen geschrieben wurden, werden als zwei einzelne Wörter gespeichert. Der Bindestrich fällt dabei weg. In dem erkannten Satz, welcher direkt von der Vision API gesendet wird, ist allerdings ein Bindestrich vorhanden. Es sollte also möglich sein, diese Informationen aus dem TextAnnotations-Objekt zu extrahieren.
- 14-mal wurde ein zusätzliches Wort eingefügt. Meistens ist dies ein Buchstabe, welcher von der Vision API zwei Mal erkannt wurde. Die Koordinaten dieses Buchstabens sollten sich mit jenen des Wortes, in dem er ein zweites Mal steht, überschneiden. Somit sollte es möglich sein, solche Fälle zu erkennen und herauszufiltern.

KAPITEL 6

Schlussbericht

6.1 Ablauf

Das Projekt war in vier Sprints unterteilt, an deren Ende jeweils ein Arbeitspaket abgeschlossen sein sollte. Dafür wurden die folgenden Meilensteine festgelegt:

1. Projektplan fertig (10. Oktober 2021):

Vier Wochen nach Start des Projekts sollte der Projektplan fertig sein. Dies beinhaltete insbesondere die Use Cases, sowie die verwendete Architektur.

Da die Architektur bereits vorgegeben war und es nur einen Use Case gab, entstand dadurch wenig Aufwand. Mehr Zeit wurde deshalb für das Verständnis der Problemstellung und der verwendeten Komponenten aufgewendet.

2. Lauffähiger Prototyp ML (31. Oktober 2021):

Geplant war, zu diesem Zeitpunkt einen Prototypen fertiggestellt zu haben, der erkennt, ob es sich bei einem Bild um ein Post-it handelt, und von diesem den Text und die Farbe ausliest.

Im erhaltenen Source-Code war bereits die Implementierung einer früheren Studienarbeit mit ähnlichem Ziel vorhanden. Somit bestand der grösste Teil des Aufwandes darin, diese Implementation zu verstehen. Da keine gängigen Coding-Guidelines eingehalten worden waren, gestaltete sich dies eher umständlich. Trotzdem konnte zum angepeilten Zeitpunkt die gewünschte Funktionalität erreicht werden.

Von Trihow wurde gewünscht, verschiedene Lösungsstrategien zu testen. Aus diesem Grund, und da es schwierig war zu zweit am selben Prototypen zu arbeiten, begann die Arbeit an einem eigenen Machine Learning Modell auch schon in diesem Sprint. Das Ziel war, bis zum folgenden Sprint Review am 16. November zwei Prototypen zur Bilderkennung implementiert zu haben. So konnte an diesem Treffen die Lösung gewählt werden, welche weiterverfolgt werden sollte. Dieses Ziel wurde auch erreicht und mit Trihow wurde entschieden, an der Lösung mit der Google Cloud Vision API weiterzuarbeiten.

3. Synthetische Post-it Generation (5. Dezember 2021):

Beim zweitletzten Meilenstein sollte die Generierung eines synthetischen Post-its fertiggestellt sein. Vorgängig hatte man sich darauf geeinigt, in einem ersten Schritt eine SVG-Datei zu erstellen.

Während die Erstellung des Strings, welcher die SVG-Datei beschreibt, relativ problemlos funktionierte, gab es einige Probleme, eine neue Datei auf den Firebase Storage Bucket zu laden. Dies wurde temporär gelöst, indem der String als Parameter in die Firestore Db eingefügt wurde. Da der Prototyp mit einem eigenen Machine Learning Modell bereits im letzten Sprint realisiert worden war, gab es in dieser Iteration noch Zeit, um die Texterkennung zu verbessern. So wurde an der Reihenfolge der Wörter, sowie an der Rechtschreibeprüfung gearbeitet.

4. Projektabgabe (24. Dezember 2021):

In der letzten Iteration der SA wurde die synthetische Bilderstellung überarbeitet, so dass die Bilder neu als JPEG und direkt in einem Firebase Storage Bucket gespeichert wurden. Ausserdem wurden Daten für den Acceptance Test erstellt und dieser durchgeführt und ausgewertet. Schlussendlich musste die Dokumentation noch fertiggestellt werden.

6.2 Zielerreichung

Die folgenden Projektziele wurden zum Beginn in Auftrag gestellt:

- *Aufbau und Trainieren der Modelle mit Google AutoML oder Google Cloud Vision API:* Die Objekt-, Farb- und Texterkennung wurde mittels Google Cloud Vision API umgesetzt. Das Trainieren eines eigenen Modells wurde zwar auch umgesetzt, jedoch nicht weiter verwendet.
- *Erkennung von Post-it Zettel in Fotos:* Dies wurde mittels der Google Cloud Vision API umgesetzt.
- *Generieren von strukturierten Metadaten in Firestore:* Der erkannte Text wurde in mehreren Schritten verbessert und danach auf der Firestore Db gespeichert. Ebenfalls gespeichert wurden die erkannte Farbe und die Metadaten des generierten JPEG-Bildes.
- *Generieren synthetisches Bild PNG/SVG in Firebase:* Dieses Ziel wurde während des Projekts verändert, so dass neu ein JPEG-File im Firebase Storage Bucket gespeichert werden sollte. Dies wurde umgesetzt.
- *Prototyp in TypeScript unter Verwendung von Google's Machine Learning Lösungen und Firestore (CloudDB):* Die Lösung wurde komplett in TypeScript mit den gängigen Style Guidelines implementiert.

Somit konnten alle Ziele erreicht werden.

6.3 Persönliche Erfahrungen

6.3.1 *Diluxion Marku*

Anfangs gab es viele Schwierigkeiten, wie das Verstehen der bereits existierenden Lösung, aber auch der Umgang mit der Firebase Infrastruktur. Sehr viel Zeit wurde allein darin investiert, um alles zum „Laufen" zu bringen. Da ich vor dieser Arbeit weder mit Firebase noch mit Cloud Vision etwas zu tun hatte, war es eine grosse Umgewöhnung, dass nun die ganze Softwarelösung auf Servern läuft und nicht lokal auf dem Engineering PC. Nichtsdestotrotz war diese Arbeit sehr spannend und ermöglichte mir, die Erlernung neuer Skillsets und das Verständnis für serverbasierte und verteilte Softwaresysteme. Es war eine tolle Erfahrung, ein Feature für eine bereits im Markt existierende Softwarelösung mit einer Firma zusammen weiterzuentwickeln.

6.3.2 *Lukas Dätwyler*

Für mich war es eine spannende und neue Erfahrung, einen Teil in einem viel grösseren Projekt umzusetzen. Dadurch wurde es nötig, den bestehenden Code ohne gross verhandene Dokumentation zu verstehen, damit der eigene Teil an der richigen Stelle eingefügt werden konnte. Im eigenen Teil war sowohl das Nutzen von grossen Packages wie der Cloud Vision API, als auch die Erarbeitung eigener Lösungen, wie z.B. das Ordering sehr spannend.

KAPITEL 7

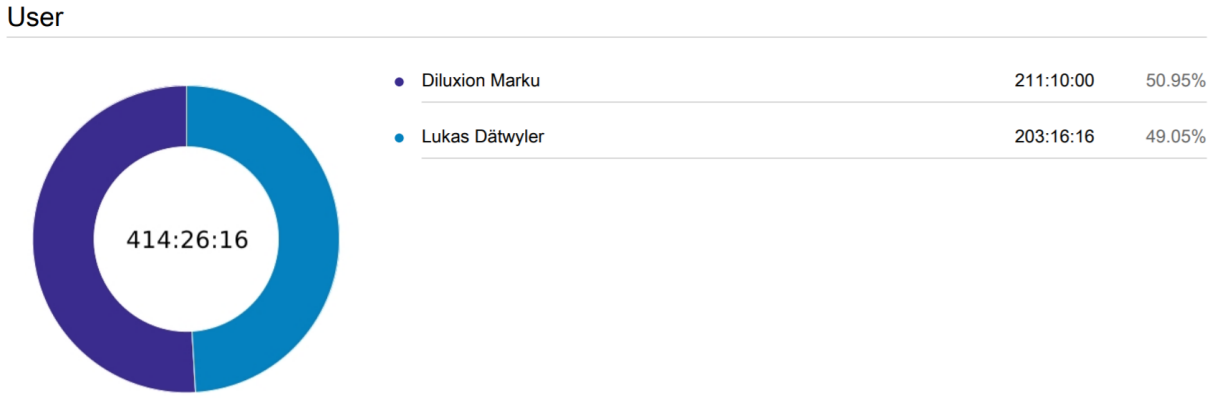
Time Report

7.1 Zweck

Dieser Time Report dient zur Auswertung des Zeitmanagements der Studierenden, während der Dauer des Projekts.

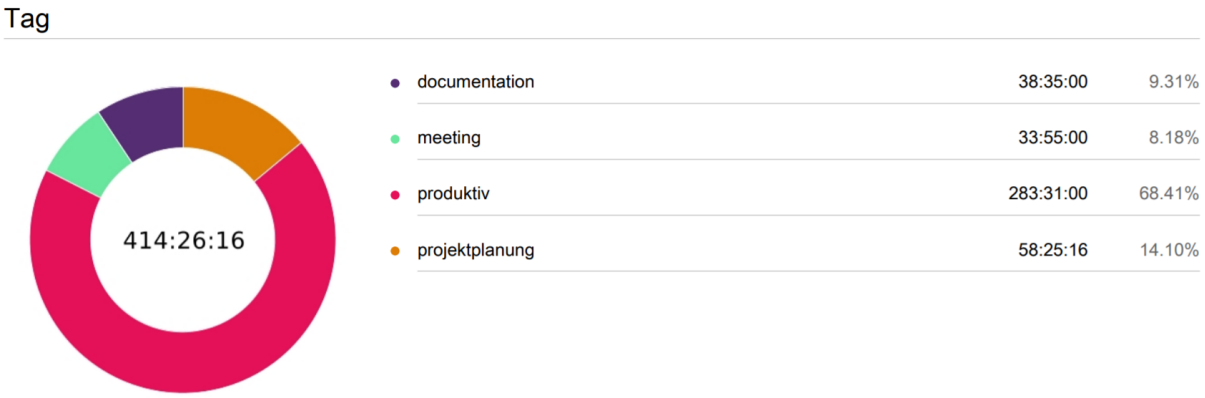
7.2 Auswertung der Zeit Pro Teammitglied

Das Projekt wurde von beiden Studierenden ungefähr im gleichen Anteil bearbeitet. Wie man sieht, wurden die 240 Stunden pro Student nicht ganz erreicht. Erwähnenswert ist aber, dass die geleistete Zeit, vor allem am Anfang des Projekts, nicht konsequent eingetragen wurde. Aus diesem Grund fehlen natürlich noch diese Stunden in der Auswertung. Da die Zeiteintragung schliesslich nicht automatisiert werden konnte, war dies ein mühsames Unterfangen. Zudem fehlen auch noch einige Stunden, die für die abschliessenden Arbeiten an der Dokumentation aufgewendet wurden.



7.3 Auswertung der Zeit Pro Label

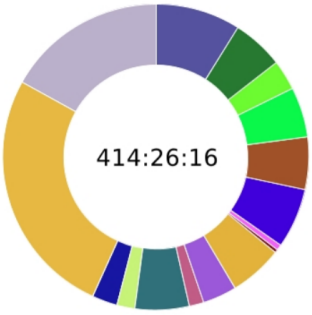
Wie man sehen kann, wurde die meiste Zeit auch tatsächlich für die Implementierung investiert. Mit grossem Abstand kommt dann die Projektplanung. Abgesehen von den Meetings, wurde die wenigste Zeit für die Dokumentation investiert. Nicht aufgeführt ist dabei die Dokumentation, welche bereits während der Implementierung geschrieben wurde.



7.4 Auswertung der Zeit Pro Arbeitspaket

Wie man der Auswertung entnehmen kann, wurde für die Erstellung des jeweiligen Prototyps, sowie der Implementierung der Rechtschreibprüfung und der Bildsynthese die meiste Zeit investiert.

Description



Bildsynthese	70:00:00	16.89%
laufender Prototyp für Google Vision API und ML Kit für die Entscheidungsfindung	109:21:00	26.39%
einlesen bisherige Lösung	10:45:00	2.59%
Sprint Review	08:00:00	1.93%
Projektplan erstellen	23:45:16	5.73%
Firebase Emulator aufsetzen	06:10:00	1.49%
Neue Session auf Firebase Server erstellen	15:15:00	3.68%
Validation aktuelle Implementation	23:15:00	5.61%
Meeting mit Christoph	01:15:00	0.30%
SA Initialmeeting	02:30:00	0.60%
Research	26:15:00	6.33%
Softwarearchitektur dokumentieren	22:55:00	5.53%
Weekly Meeting	22:10:00	5.35%
Dokumentation fertigstellen	13:00:00	3.14%
Projekt aufsetzen	22:30:00	5.43%
Rechtschreibprüfung (NLP)	37:20:00	9.01%

7.5 Bearbeitete Arbeitspakete Pro Teammitglied

Lukas Dätwyler	203:16:16
Projektplan erstellen	13:45:16
Rechtschreibprüfung (NLP)	37:20:00
Firebase Emulator aufsetzen	06:10:00
einlesen bisherige Lösung	10:45:00
Projekt aufsetzen	10:00:00
Softwarearchitektur dokumentieren	22:55:00
Dokumentation fertigstellen	13:00:00
Weekly Meeting	08:00:00
Sprint Review	05:00:00
laufender Prototyp für Google Vision API und ML Kit für die Entscheidungsfindung	61:36:00
Research	07:15:00
SA Initialmeeting	01:15:00
Validation aktuelle Implementation	06:15:00

Diluxion Marku	211:10:00
Bildsynthese	70:00:00
Weekly Meeting	14:10:00
laufender Prototyp für Google Vision API und ML Kit für die Entscheidungsfindung	47:45:00
Sprint Review	03:00:00
Neue Session auf Firebase Server erstellen	15:15:00
SA Initialmeeting	01:15:00
Projekt aufsetzen	12:30:00
Projektplan erstellen	10:00:00
Meeting mit Christoph	01:15:00
Validation aktuelle Implementation	17:00:00
Research	19:00:00

KAPITEL 8

Glossar

SA Abkürzung für Studienarbeit.

Google Cloud Vision API Interface, um die automatische Bilderkennung von Google Cloud zu nutzen. (https://cloud.google.com/vision)

Google AutoML Tool, um eigene Machine Learning Modelle zu bauen. (https://cloud.google.com/automl)

SVG Vektorbasiertes Grafikformat. Abkürzung für Scalable Vector Graphics.

Firebase Entwicklungsplatform von Google für Mobile- und Webanwendungen. (https://firebase.google.com/)

Firestore Dokumentbasierte Datenbank, welche von Firebase genutzt wird. (https://firebase.google.com/docs/firestore)

Firebase Storage Bucketbasiertes Filesystem, um grössere Dateien abzulegen. (https://firebase.google.com/docs/storage)

Cloud Functions Funktionen, welche auf Firebase gespeichert sind und per Trigger ausgeführt werden. (https://firebase.google.com/docs/functions)

Postman Software, mit welcher https-Anfragen erstellt und ausgeführt werden können. (https://www.postman.com/)

nspell Npm package, welches eine Rechtschreibeprüfung mittels eines dictionaries durchführt. (https://github.com/wooorm/nspell)

dictionaries Npm packages, welche die gültigen Wörter, sowie einige Metainformationen einer Sprache beinhalten. (https://github.com/wooorm/dictionaries/tree/main/dictionaries/de-CH)

temp Npm package für die Erstellung von temporären Verzeichnissen. (https://www.npmjs.com/package/temp)

convert-svg-to-jpeg Npm package für die Konvertierung von SVG Files zu JPEG Bildern. (https://www.npmjs.com/package/convert-svg-to-jpeg)

Typescript Erweiterung von Javascript, welche Typen und einen Compiler einführt. (https://www.typescriptlang.org/)

TsDoc Standart für das Schreiben von Dokumentation in Typescript Projekten. (https://tsdoc.org/)

Typedoc Tool, um automatische Dokumentation aus Typescript Code zu generieren. (https://typedoc.org/)

Stichwortverzeichnis