INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine Universität Düsseldorf

Analyse des Kundenverhaltens durch Mustererkennung in Logfiles

Markus Stroh

Bachelorarbeit

Beginn der Arbeit:	18. Februar 2020
Abgabe der Arbeit:	18. Mai 2020
Gutachter:	Prof. Dr. Stefan Conrad
	Prof. Dr. Stefan Conrad

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 18. Mai 2020

Markus Stroh

Zusammenfassung

Hier kommt eine ca. einseitige Zusammenfassung der Arbeit rein.

Inhaltsverzeichnis

1 Einleitung

In dem ersten Kapitel der Arbeit wollen wir die Problematik und die Aufgabenstellung definieren. Dazu wird zunächst die Firma Computer-Communications Networks (CoCoNet) vorgestellt. Nachdem die Arbeit dadurch in den Firmenkontext gebracht wurde, wollen wir konkret die Fragestellung der Arbeit definieren und deren Aufbau der beschreiben.

1.1 Firmenprofil CoCoNet GmbH

Die Firma CoCoNet entwickelt digitale Banking-Lösungen für Firmenkunden. In dieser Arbeit beschäftigen wir uns mit einem bestimmten CoCoNet Produkt, das Multiversa International Finance Portal (IFP). Das IFP ist eine Server-Basierte Anwendung, auf die User per Internet-Browser zugreifen können. Nachdem sich ein User in das IFP eingeloggt hat, wird er zunächst zum Dashboard weitergeleitet. Das Dashboard ist in gewisser Weise die Startseite des IFPs, bestehend aus einer Navigationsleiste und Dashboardwidgets. Von hier aus haben User die Möglichkeit durch das IFP zu navigieren, um z.B. eine Überweisung zu tätigen. (CoCoNet, 2020)
Für den weiteren Verlauf der Arbeit wollen wir nicht den kompletten Funktionsumfang des IFPs beleuchten, sondern uns auf das Dashboard mit seinen Widgets fokussieren.
Ein Widget ist eine Einheit, die auf dem Dashboard abgelegt werden kann und eine bestimmte Funktionalität hat. Betrachten wir als Beispiel das *Open Payments Widget*. Wie der Name schon vermuten lässt, zeigt das Widget Zahlungen an, die noch offen sind, also authorisiert werden müssen. Klickt man nun eine offene Zahlung an, wird man auf die entsprechende Seite weitergeleitet, die mit den Zahlungsinformationen der offenen Zahlung ausgefüllt ist.

1.2 Problemstellung (Aufgabestellung?) (anforderungen an das system)

In dieser Arbeit geht es darum, das Userverhalten in Hinblick auf die Nutzung der Dashboardwidgets im IFP zu analysieren. Da der Begriff Userverhalten noch recht breit gefächert ist, sollen konkret die folgenden Fragen beantwortet werden:

1. Wie oft werden Widgets in einem bestimmten Zeitraum benutzt?

2. Lässt sich ein bestimmter Workflow durch die Nutzung der Widgets erkennen?

Diese Fragen wollen wir durch die Analyse von Logfiles beantworten.
Zunächst müssen wir aber definieren, was es bedeutet, ein Widgets zu benutzen. Da das Dashboard bzw. die Widgets das Erste sind, was man nach einem Login in das IFP sieht, könnte man meinen, dass das pure Vorhandensein auf dem Dashboard eine Nutzung schon einschließt. Betrachtet man z.B. das Widget in [ABB-SCREENSHOT-WIDGET] stellt man fest, dass das Widget schon einiges an Informationen liefert. Allerdings kann

man an dieser Stelle nur spekulieren, ob und in welchem Ausmaß der User diese Information wahrnimmt. Da also ein pures Vorhandensein des Widgets auf dem Dashboard als Nutzung nicht hinreichend ist, zählt erst ein Klick auf das Widget als Nutzung.
Wenn nun ein Widget angeklickt wird, wird der Nutzer in der Regel auf eine neue Seite weitergeleitet, die durch das Widget z.B. schon vorgefiltert ist. Da diese Weiterleitung ein HTTP-Request an den Server ist, wird sie geloggt und kann in den Logfiles erkannt werden.
Also sind die Anforderungen an das zu entwickelnde System, dass die Nutzung eines Widgets erkannt wird und aus diesen Informationen Verhaltensmuster der User bzgl. der eben erwähnten Fragen erkannt werden.
Ein weiterer zu berücksichtigender Aspekt ist, dass das System nutzerfreundlich sein soll, damit es auch Menschen ohne besondere Vorkenntnisse benutzen können.

1.3 Aufbau der Arbeit

Im folgenden Verlauf der Arbeit wird zunächst auf die theoretischen Grundlagen der Mustererkennung eingegangen. Dazu wird die Syntax der vorliegenden Logfile Einträge und anschließend die Phasen der Mustererkennung, die die Einträge durchlaufen, beschrieben. In dem darauf folgenden Kapitel wird beschrieben, wie die theoretischen Grundlagen technisch umgesetzt werden. Danach wird das entwickelte System getestet und die Ergebnisse ausgewertet. Zum Schluss wird die Arbeit durch ein Fazit zusammengefasst und zukünftige Einsatzmöglichkeiten erörtert.

2 Grundlagen

In diesem Kapitel beschäftigen wir uns mit den theoretischen Grundlagen der Methoden, die benutzt wurden, um die Kundendaten zu analysieren. Dafür wollen wir zunächst die Syntax der Logfiles beschreiben.

2.1 Syntax der Logeinträge

Das IFP generiert bei seiner Benutzung mehrere verschiedene Logfiles. Je nach Art des Logfiles können hier verschiedene Informationen gespeichert werden wie z.B. Zeitstempel, UserIDs, Exception Messages usw. Da für diese Arbeit aber nur die sog. „Sessionlogs“ relevant sind, soll auch nur deren Syntax beschrieben werden.
Die Einträge in den Sessionlogs spiegeln im Großen und Ganzen die Serveraktivität wider: Es werden ein- und ausgehende HTTP Requests festgehalten. Allerdings ist es nicht ausreichend nur eine Liste von Requests zu speichern. Sie müssen noch in einen gewissen Kontext gebracht werden. Deshalb werden zusätzlich zu den Requests auch Daten wie Zeitstempel, UserID, Parameter u.s.w. geloggt. Durch diese Informationen kann man nun nachvollziehen, welcher User zu welchem Zeitpunkt Daten gesendet bzw. angefragt hat. Neben der UserID wird auch die SessionID mit geloggt. Sobald sich ein User erfolgreich in das IFP einloggt, wird eine SessionID generiert. Diese SessionID ist vom Zeitpunkt des Logins bis zum Logout gültig. Möchte sich derselbe User nach dem Logout direkt noch einmal einloggen, wird eine neue SessionID generiert. Die folgende Abbildung zeigt die Requests, die beim Login Prozess geloggt werden: (ABB:LOGin-SESSIONLOG) (wahrscheinlicher besser im appendix)
Man erkennt schnell, dass man die Einträge in den Logfiles grob in die folgenden Felder unterteilen kann:

```
[Timestamp] [Loglevel] [SessionID] [CustomerID] [UserID] [TaskID] [HTTP Request]
```

Listing 1: Felder in Sessionlogs

Da nun die Syntax der Einträge erkannt wurde, kann die Mustererkennung beginnen.

2.2 Mustererkennung

Generell kann man in vielen verschiedenen Medien Muster erkennen, wie z.B. Bilddateien, Musikdateien oder eben in Textdateien. Dabei kommt es im Wesentlichen darauf an, welche Muster man erkennen möchte. So könnte man einem Programm beibringen, bestimmte Fischarten auf Fotos zu erkennen (Duda et al., 2001). Ein allgemeiner Ansatz, das Problem der Mustererkennung zu lösen, ist es, das Problem in folgende Teilprobleme aufzuteilen (Zitat anderes buch):

1. Segmentiereung
2. Feature Extraction

3. Klassifizierung

An dieser Stelle wollen wir drauf hinweisen, dass diese Begriffe eher konzeptionell zu verstehen sind, anstatt wörtlich. Im Folgenden wollen wir dieses generelle Vorgehen auf die Einträge der Logfiles anwenden.

2.2.1 Segmentierung (bestimmung der relevanten felder)

In dem Schritt der Segmentierung wollen wir feststellen, welche Informationen aus den Logfiles für die Fragestellung relevant sind. Greifen wir dafür zunächst einmal die Definition aus der Einleitung auf, was es bedeutet ein Widget zu benutzen. Wie bereits erwähnt reicht ein pures Anzeigen von Informationen nicht aus; das Widget muss auch angeklickt werden. Ein Klick auf ein Widget hat i.d.R. immer denselben Effekt: man wird auf eine (evtl. vorgefilterte) Seite weitergeleitet. Das heißt, man sendet einen HTTP Request an den Server und der Server sendet eine Antwort. Also ist es offensichtlich, dass wir für unsere Analyse nur die HTTP Requests betrachten wollen und die Antworten des Servers irrelevant sind. Demnach sind wir nur an Einträgen interessiert, die in dem Feld [HTTP Request] aus Listing 1 mit *Incoming Request* anfangen. Dadurch haben wir die Menge an Einträgen, die wir betrachten wollen, um ca. die Hälfte reduziert.
Betrachten wir nun den Rest des Feldes genauer anhand eines Beispieleintrags, in dem ein Widget benutzt wurde. Insbesondere die URL ist für uns von Interesse:

```
/MULTIVERSA–IFP/lightning/ecm/liquidity/banks/liquidity\_banks.
    jsf?selectedView=ecm.liquidity.banks.view.all\&viewType=0\&
    widget=LiquidityByBanksWidgetContent\&banks=
    RpBLVhy85c5u4nGKeISH0A\&conversationContext=3\&\_ns\_=
    f63a91bc–9ab1–472d–bcb0–b6771837ffaa3\&\_nc\_=
```

Listing 2: Beispiel URL

Der Name des Widgets ist in der URL erkennbar, wie man hier rot markiert sehen kann. Neben dem HTTP Request sind diese weiteren Felder von Relevanz:

- Timestamp
- SessionID
- UserID

2.2.2 Feature Extraction (transformation der daten)

Nachdem alle irrelevanten Daten in der Segmentierung eliminiert wurden, ist der nächste Schritt die feature extraction. Dieser Begriff könnte irreführend wirken, da in dieser Arbeit keine features im wörtlichen Sinn extrahiert werden, vielmehr werden die Daten in eine bestimmte Struktur transfomiert. Diese Struktur besteht aus drei Mengen, die Menge an Usern, die Menge an SessionIDs und die Menge an Widgets. Diese Struktur wird

von nun an *Session Entity* genannt. Eine Session Entity hat eine eindeutige UserID, eine eindeutige SessionID und eine Menge an Widgets, die in dieser Session benutzt wurden. Mit anderen Worten beinhaltet eine Session Entity die Informationen, welche Widgets in einer Session benutzt wurden. Abbildung X soll diesen Zusammenhang verdeutlichen. Zusätzlich zu der Information, welche Widgets benutzt wurden, wird noch festgehalten, zu welchem Zeitpunkt bzw. in welcher Reihenfolge sie benutzt wurden. Sowohl die Uhrzeit als auch die UserID werden im weiteren Verlauf der Arbeit keine tragende Rolle mehr spielen. Allerdings macht es Sinn diese Daten mit zu speichern, da sie für zukünftige Anwendungsfälle relevant werden können. In Kapitel (ENDE) wird dies genauer diskutiert.

2.2.3 Klassifizierung

Nun da unsere Daten transformiert wurden können wir beginnen, die Daten zu klassifizieren. Da wir in Abschnitt 1.2 verschiedene Fragen beantworten wollen, müssen wir dafür die Klassifizierung für diese beiden gesondert betrachten.

Diagramm

Um die Frage zu klären, wie sich die Widgetnutzung pro Tag verhält, wollen wir ein Linienhistogramm erstellen. Das hat den Vorteil, dass nicht nur die Nutzung an sich dargestellt wird, sondern auch ein Verlauf, bzw. ein Trend in der Nutzung der Widgets. Zusätzlich wird in einem Tortendiagramm die anteilige Widgetnutzung dargestellt werden.

Assoziationsregeln

Die Suche nach Assoziationsregeln wird auch oft als Warenkorbanalyse bezeichnet. Dabei geht es darum, in einer Datenbank Beziehungen zwischen den Daten zu finden. Bevor wir in die Theorie hinter den Assoziationsregeln eintauchen, wollen wir mit Hilfe der folgenden Tabelle die wichtigsten Begriffe vorstellen. Zum besseren Verständnis zeigen wir zu den Begriffen Analogien zu unserem Anwendungsfall und eines Online Shops.

Allgemein	Online Shop	IFP
Transaktion	Ein einzelner Warenkorb	Eine einzelne SessionID
Menge aller Transaktionen D	Menge aller Warenkörbe D	Menge aller SessionIDs D
Item	ein Produkt	ein Widget bzw. eine Widgetnutzung
Menge aller Items I	Menge aller Produkte I	Menge aller Widgets I

Tabelle 1: Begriffe zu Assoziationsregeln

Nun wollen wir definieren, was eine Assoziationsregel ist. Betrachten wir dafür die Widgets w_1 und w_2. Eine Assoziationsregel wird mit

$$w_1 \rightarrow w_2$$

notiert und sagt aus, dass wenn w_1 benutzt (also angeklickt) wurde, w_2 auch benutzt wurde. An eine Assoziationsregel sind zwei Werte geknüpft, die beschreiben, wie „gut“ eine Regel ist: der *Support* und die *Konfidenz*. Diese Werte wollen wir mit Hilfe eines kleinen Beispiels definieren:

SessionID	Benutzte Widgets
1	w_1, w_2, w_4, w_5
2	w_1, w_3
3	w_2, w_5
4	w_1, w_2
5	w_2, w_3, w_5

Tabelle 2: Beispiel Datenbank zu Assoziationsregeln

Wir wollen zunächst die Begriffe aus Tabelle 1 auf dieses Beispiel anwenden:

- Tabelle 2 ist unsere Datenbank
- Die Menge aller Items $I = \{w_1, w_2, w_3\}$. Eine Menge $X \subseteq I$ wird auch *Itemset* genannt. Also wäre z.B. $\{w_1, w_3\} \subseteq I$ ein Itemset.
- Die Menge aller SessionIDs $D = \{1, 2, 3, 4\}$. T_i mit $i \in D$ und $T_i \subseteq I$. Demnach gilt z.B. $T_1 = \{w_1, w_2, w_3\}$

Der Support einer Menge wird definiert als:

$$\text{support(X)} = \frac{\text{Anzahl der Transaktionen, die X enthalten}}{\text{Anzahl aller Transaktionen}},\ \text{X} \subseteq I$$

Demnach gilt z.B. für $X_1 = \{w_1, w_3\}$, $X_2 = \{w_2\}$ und $X_3 = \{w_1\}$:

$$\text{support}(X_1) = \frac{1}{5} = 0.2$$

$$\text{support}(X_2) = \frac{4}{5} = 0.8$$

$$\text{support}(X_3) = \frac{3}{5} = 0.6$$

Also beschreibt der Support eines Itemsets die relative Häufigkeit des Itemsets im Bezug auf die Anzahl der Transaktionen in der Datenbank. Des Weiteren gilt für X,Y $\subseteq$ I:

$$\text{support}(X \rightarrow Y) = \text{support}(X \cup Y)$$

Als nächstes wollen wir die Konfidenz definieren. Die Konfidenz einer Assoziationsregel $X \rightarrow Y$ definiert als:

$$\text{konfidenz}(X \rightarrow Y) = \frac{\text{support}(X \rightarrow Y)}{\text{support}(X)}$$

Auf unser Beispiel bezogen wollen wir nun die Konfidenz für die Regel $\{w_1\} \rightarrow \{w_2\}$ berechnen. Es gilt:

$$\begin{aligned}\text{konfidenz}(\{w_1\} \rightarrow \{w_3\}) &= \frac{\text{support}(\{w_1\} \rightarrow \{w_3\})}{\text{support}(\{w_1\})} \\ &= \frac{\text{support}(\{w_1\} \cup \{w_3\})}{\text{support}(\{w_1\})} \\ &= \frac{0.4}{0.6} = 0.67\end{aligned}$$

Die Konfidenz beschreibt also wie hoch die Wahrscheinlichkeit ist, dass wenn w_1 benutzt wurde, w_3 auch benutzt wurde. (Beierle und Kern-Isberner, 2019)

Nachdem wir nun alle relevanten Begriffe bzgl. der Assoziationsregeln definiert haben, wollen wir genauer erörtern, wie Regeln gefunden werden sollen. Dazu rufen wir uns noch einmal ins Gedächnis, welche Information wir uns aus den Regeln erhoffen: wir wollen Workflows erkennen. Nehmen wir mal an, das Itemset $\{w_1, w_3\}$ käme in einer Datenbank mit 1000 Einträgen nur ein einziges Mal vor. Es ist offentsichtlich, dass man bei dieser Regel nicht von einem Workflow sprechen kann. Von daher suchen wir Itemsets, die *häufig* vorkommen. Diese Häufigkeit wird durch den *minsupport* festgelegt. Ein Itemset gilt als häufig, wenn gilt:

$$\text{support}(X) \geq \text{minsupport}$$

Analog wollen wir auch nicht alle Regeln, die wir aus den häufigen Itemsets generieren, betrachten. Vielmehr wollen wir die Regeln finden, die die *minkonfidenz* erfüllen. Daraus folgt, dass wir unsere Aufgabe in zwei Teilprobleme aufteilen können:

1. Finde alle Itemsets die häufig sind, also den minsupport erfüllen. Ein naiver Ansatz wäre, die Potenzmenge $\mathcal{P}(I)$ zu berechnen und zu prüfen, welche der Teilmengen den minsupport erfüllen. Allerdings hätte dieses Vorgehen eine Laufzeit von $\mathcal{O}(2^n)$ wobei n die Anzahl der Widgets ist. Deshalb wird der sog. *Apriori Algorithmus* verwendet, der die Laufzeit stark einschränkt, indem die zu betrachtenden Itemsets

eingeschränkt werden.

2. Finde alle Assoziationsregeln, die aus den häufigen Itemsets generiert werden können und die die minkonfidenz erfüllen. Wenn Itemset X häufig ist, ist auch jede Teilmenge A von X häufig und ergibt die Regel A $\rightarrow$ X $-$ A. Für diese Regeln muss die Konfidenz geprüft werden.

Bevor wir nun die Algorithmen vorstellen, die zu Lösung der Teilprobleme benutzt wurden, sei noch angemerkt, dass der minsupport und die minkonfidenz Werte sind, die vom Benutzter festgelegt sind.

Apriori Algorithmus

Der Apriori Algorithmus wurde von Agrawal et al. (1993) entwickelt und wird benutzt, um möglichst effizient häufige Itemsets zu finden. Dabei wird folgendes Lemma ausgenutzt:

Lemma 1. *Sei X ein Itemset. Wenn X kein häufiges Itemset ist, ist auch keine Obermenge von X häufig. (Agrawal et al., 1993)*

Ausgangspunkt der Kandidatengenerierung ist, dass man häufige Itemsets mit k Elementen betrachtet. Aus diesen Itemsets sollen nun Itemsets generiert werden, die $k+1$ Elemente beinhalten. Um das zu erreichen, werden zwei Itemsets vereinigt, die $k-1$ gleiche Elemente haben. Angenommen, wir hätten die häufigen Itemsets $X = \{w_1, w_2, w_3\}$ und $Y = \{w_1, w_2, w_4\}$. Beide Mengen haben $k = 3$ Elemente. Um aus diesen Menge eine $k + 1 = 4$ - elementige Menge zu generieren, müssen sie $k - 1 = 2$ gleiche Elemente haben. Da das der Fall ist, erhalten wir als Ergebnis die Vereinigung beider Mengen mit $X \cup Y = \{w_1, w_2, w_3, w_4\}$. Dieser Schritt wird *join* genannt.
Nach dem join muss noch geprüft werden, ob der generierte Kandidat auch Lemma 1 erfüllt. Dies geschieht beim *pruning* indem geprüft wird, ob jede $k - 1$ - elementige Teilmenge des generierten Kandidaten ist. Denn wenn ein eine Teilmenge des generierten Kandidaten nicht häufig ist, kann dieser auch nicht häufig sein und muss nicht weiter betrachtet werden. In Anhang Teil A wird die Kandidatengenerierung mit dem Beispiel aus Tabelle 2 vorgerechnet.

Assoziationsregeln finden

Aus den häufigen Itemsets, die der Apriori Algorithmus gefunden hat, wollen wir nun Assoziationsregeln finden. Ähnlich wie bei den häufigen Itemsets wollen wir nur Regeln finden, die die minkonfidenz erfüllen. Dies erreichen wir, indem wir aus einem häufigen Itemset X zunächst Regeln bilden, die die Form

$$Y \rightarrow X - Y, Y \subset X$$

bilden. Ähnlich wie bei der Kandidatengenerierung nutzen wir wieder eine Teilmengenbeziehung aus: Wenn die Regel $Y \rightarrow X - Y$ die minkonfidenz nicht erfüllt, erfüllt auch keine Teilmenge $Y' \subset Y$ mit der Regel $Y' \rightarrow X - Y'$ die minkonfidenz.

3 Umsetzung

Nachdem wir in Kapitel 2 die theoretischen Grundlagen für unser Vorhaben erläutert haben, wollen wir in diesem Kapitel beschreiben, wie die Grundlagen umgesetzt wurden.

3.1 Grundlagen zum Elastic Stack

Um die Fragen aus der Problemstellung zu beantworten, wurde der Elastic Stack (ELK) benutzt. Der ELK besteht aus den Software Produkten Elasticsearch, Logstash und Kibana der Firma Elastic N.V. Zusätzlich wurde noch Filebeat von der gleichen Firma benutzt. Die Funktionen der einzelnen Produkte werden in den nachfolgenden Unterkapiteln weiter erläutert. Die folgende Abbildung stellt den Workflow des Systems dar:

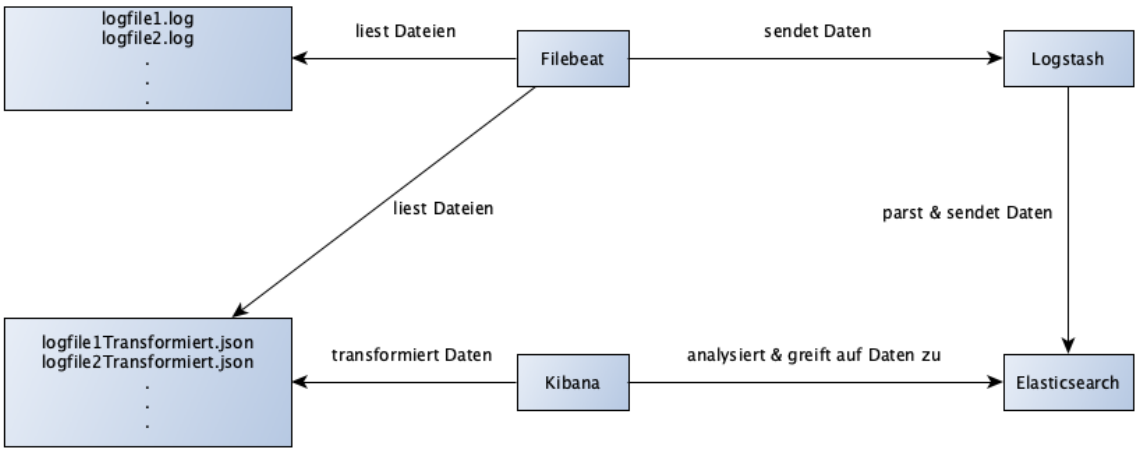

Abbildung 1: ELK Workflow

3.1.1 Filebeat

Mit Filebeat können Dateien zeilenweise gelesen werden. In der Konfigurationsdatei von Filebeat gibt man an, wo und nach welchen Dateien gesucht werden soll. In unserem Fall waren es .log und .json Dateien. Außerdem kann man ein multiline pattern angeben, falls ein Eintrag mehrzeilig sein sollte. Die eingelesenen Dateien sendet Filebeat an Logstash weiter. Dabei merkt sich Filebeat, welche Dateien schon gelesen wurden und welche Zeilen in diesen Dateien gelesen wurden. Das bedeutet ebenfalls, wenn eine komplett neue Datei oder eine neue Zeile in einer alten Datei auftaucht, merkt Filebeat das, liest es und sendet die Daten wieder an Logstash. Damit ist die Echtzeitanalyse der Daten gewährleistet.

3.1.2 Logstash

Die Aufgabe von Logstash ist Daten von Filebeat zu empfangen und zu verarbeiten. Die empfangenen Daten befinden sich in einem rohen Zustand und werden mithilfe von mehreren Filter-Plugins verarbeitet.(Elastic, 2020a) Nachdem die Daten verarbeitet wurden, sendet Logstash sie weiter an Elasticsearch.

Grok Filter Plugin

Für die Logdateien wurde das Grok Filter-Plugin verwendet. In dem Grok-Filter gibt man einen regulären Ausdruck an, mit dem ein Eintrag in den Logfile gematcht werden soll. Dabei kann man auch direkt festlegen, wie die Daten gespeichert werden sollen, falls gematcht wurde. Allgemein ist die Syntax dabei %{REGEXP:Feld}. Neben einigen regulären Ausdrücken, die im Grok-Filter bereits implementiert sind (wie z.B. *LOGLEVEL, TIMESTAMP_ISO8601, SPACE, URIPATH*, uvm.), kann man auch eigene reguläre Ausdrücke definieren. Wie schon in Kapitel 2.2.1 beschrieben, sollen die Daten segmentiert werden, was mit dem Grok-Filter realisiert werden kann. So kann man z.B. duch richtiges Platzieren von *Incoming Request* in dem regulären Ausdruck erreichen, dass *Outgoing Responses* nicht gematched und dadurch auch nicht weiter beachtet werden. Außerdem ist es möglich Daten matchen, aber keinem Feld zuweisen, wodurch sie auch nicht gespeichert werden.
Weiterhin hat man die Möglichkeit, die Felder, die durch das Matching mit dem regulären Ausdruck mit Daten gefüllt werden, weiter zu bearbeiten. Dafür können wiederum verschiedene Filter benutzt werden. So hat man die Möglichkeit Felder, die Logstash automatisch anlegt, zu bearbeiten. Ein praktisches Beispiel hierfür ist der *date*-Filter: Logstash legt automatisch ein Feld für einen Zeitstempel an, der auf den Moment eingestellt ist, an dem Logstash den eingehenden Eintrag verarbeitet. Mit dem date-Filter hat man die Möglichkeit dieses Datum zu manipulieren. In unserem Fall wurde nicht der aktuelle Zeitstempel benutzt, die Einträge zu speichern, sondern der Zeitstempel aus dem Logeintrag. Das hat den Vorteil, dass man nicht zwei verschiedene Zeitstempel speichern muss.

Ruby Filter Plugin

Das Ruby Filter Plugin wird benutzt, um transformierte Daten in Elasticsearch zu speichern. An dieser Stelle sei angemerkt, dass Elasticsearch zwar eine experimentelle *transform* Funktion besitzt, diese aber für unseren Anwendungsfall (noch) nicht passend implementiert ist (Elastic, 2020b). Deshalb wurde ein Ruby Skript entwickelt, das diese Funktion ergänzt.
Das entwickete Skript liest aus JSON Dateien die Session Entities ein, die in Kapitel 2.2.2 vorgestellt wurden. Wie genau diese Daten aussehen bzw. wie sie zustande kommen, wird in Kapitel 3.1.4 genauer erläutert. Zum besseren Verständnis, wie das Ruby Skript arbeitet, wird an dieser Stelle daher die Datenstruktur abstrakt beschrieben. Im Prinzip sind die Daten in drei Ebenen aufgeteilt:

1. UserIDs
2. SessionIDs
3. Widgets

Dabei kann man jeweils eine Ebene als ein Array betrachten. So besteht die oberste Ebene aus einem Array, das mit UserIDs gefüllt ist. Zu der UserID wird zusätzlich ein Array gespeichert, das aus SessionIDs besteht, die zu der UserID gehören. Zu diesen SessionIDs wird schließlich ebenfalls ein Array verknüpft, in dem festgehalten wird, welche Widgets in der Session genutzt wurden und wie oft. Nun durchläuft das Skript also die beschriebenen Array Ebenen und speichert die gegebenen Daten in passende Felder. Die Daten werden schließlich an Elasticsearch gesendet und dort in einem passenden Index gespeichert.
Erwähnenswert ist noch, dass wir die transformierten Daten allerdings auf zwei Arten verarbeitet werden mussten. Die Gründe dafür werden in Kapitel 3.1.4 genauer erläutert.

3.1.3 Elasticsearch

Nachdem die von Filebeat gelesenen Daten von Logstash geparst wurden, werden die Daten in Elasticsearch gespeichert. Dabei werden die Logeinträge und die transformierten Daten separat indiziert. Die Daten, die an Elasticsearch gesendet werden, können automatisch gemapped (also einem Datentyp zugeordnet) werden. Das heißt, Elasticsearch erkennt, ob z.B. ein Datum oder eine Zahl gespeichert wurde. Des Weiteren hat man aber auch die Möglichkeit, ein eigenes Mapping zu definieren. Nachdem aber ein Mapping in dem Index gespeichert wurde, kann es nicht mehr ohne weiteres geändert werden.
Für das weitere Verständnis ist es notwendig zu beschreiben, wie unsere Daten in Elasticsearch gespeichert werden. Um aber nicht unnötig ausschweifend zu werden, beschränken wir uns nur auf die Aspekte, die für uns relevant sind.
Allgemein werden Daten in Elasticsearch in Indizes gespeichert. Diese Indizes sind eine Sammlung von Dokumenten, in denen die gespeicherten Daten stehen. Die Daten sind als Key-Value-Paare gespeichert.

3.1.4 Kibana

Kibana ist ein Tool, um die Daten, die in Elasticsearch gespeichert sind, zu visualisieren bzw. zu analysieren. So kann man bspw. mit wenigen Einstellungen Visualisierungen der in Elasticsearch gespeicherten Daten erstellen, welche man wiederum zu einem Dashboard zusammen fassen kann. Ferner bietet Kibana die Möglichkeit, die Daten auch manuell zu durchsuchen. Sollte Kibana eine gewünschte Visualisierung oder ein gewünschtes Analysetool nicht nativ enthalten, hat man die Möglichkeit es um die gewünschten Funktionen zu erweitern.
Für unsere Anwendung ist eben dieser Fall eingetreten, da man nicht ohne weiteres Assoziationsregeln mit Kibana finden kann.
Bevor wir darauf eingehen, wie das Custom Plugin und die Visualisierungen eingesetzt wurden, wollen wir den Hinweis aus Kapitel 3.1.2 aufgreifen und erläutern, warum es

notwendig war, die transformierten Daten in separaten Indizes zu speichern. Wie schon erwähnt wird zu einer SessionID ein Array gespeichert, das Informationen zu den benutzten Widgets enthält. In Elasticsearch kann man Arrays unter dem Datentyp „nested “ speichern. Allerdings ist es in den Visualisierungen, die für unsere Zwecke relevant sind, nicht möglich, Daten in einer verschachtelten Datenstruktur zu lesen bzw. zu durchsuchen. Also müssen die Einträge zu den transformierten Daten in einzelnen Dokumenten in Elasticsearch gespeichert werden.
Für die Suche nach Assoziationsregeln ist diese Datenstruktur aber nicht geeignet. Möchte man z.B. nach den Sessions suchen, in denen zwei bestimmte Widgets benutzt wurden, schlägt die Suche fehl. Die Begründung dafür ist, dass in den Dokumenten nur ein Widgetname steht, aber Elasticsearch nach zwei suchen würde. Deshalb müssen die Session Entities sowohl in der Array Variante gespeichert werden, als auch als Dokumente, die nur einen Widgetnamen beinhalten.

Custom Plugin

Um einen erleichterten Einstieg in die Entwicklung eines Custom Plugins zu ermöglichen, bietet Kibana die Möglichkeit, das Grundgerüst für ein Custom Plugin generieren zu lassen. Das generierte Plugin liefert direkt die nötige Server-Client-Architektur, um den Elasticsearch-Server anzusprechen. Die grafische Oberfläche, die der Nutzer sieht, ist dabei als Client zu verstehen. Von hier aus werden HTTP Requests an einen Node.js Server gesendet. Dieser kann nun die gesendeten Daten (bspw. eine Suchanfrage) vorbereiten, an den Elasticsearch-Server senden, die Antwort verarbeiten und an den Client weiterleiten. Außerdem ist der Node.js Server auch in der Lage, Python Code ausführen zu lassen. Die folgende Abbildung soll diese Beziehung verdeutlichen.

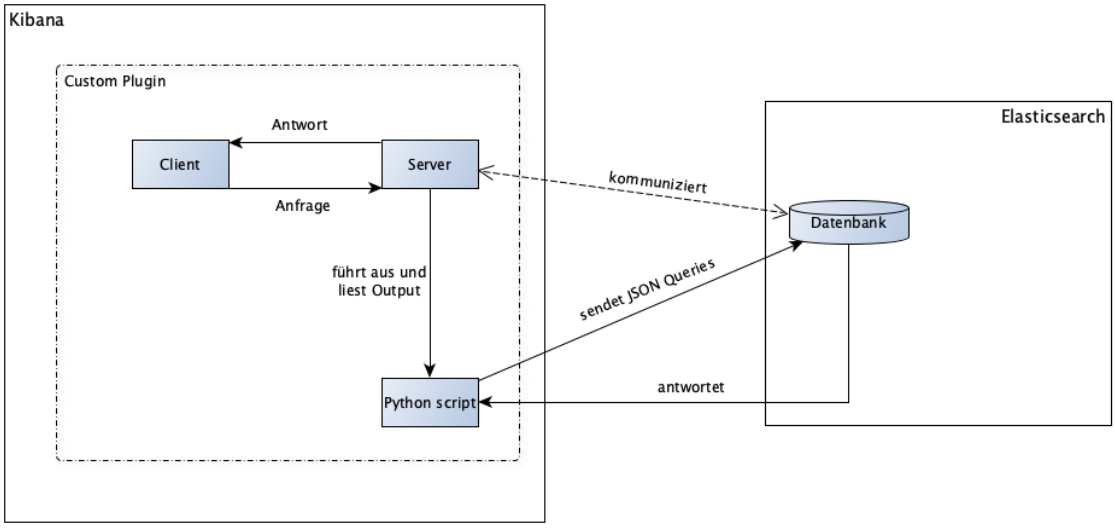

Abbildung 2: Architektur des Custom Plugins

Visualisierungen

In dieser Arbeit wurden zwei Visualisierungen benutzt, die in Kibana bereits integriert sind: die *Pie* und *Line* Visualisierungen. Erstere wurde benutzt um festzustellen, zu welchen Anteilen die zufälligen Widgeteinträge generiert wurden.
Die *Line* Visualisierung wurde benutzt, um die erste Frage aus Kapitel 1.2 zu beantworten. Die Visualisierung zeigt in einem zweidimensionalen Diagramm die Benutzung der Widgets über einen bestimmten Zeitraum, wobei dieser Zeitraum auf der x-Achse abgebildet wird. Die y-Achse gibt die Anzahl an Widget Nutzungen an. Das bedeutet ein Punkt in dem Diagramm zeigt an, wie oft ein Widget an einem bestimmten Tag benutzt wurde. Schließlich verbindet die Visualisierung die Punkte miteinander und so erhält man eine Linie, die der Widgetnutzung in einem bestimmten Zeitraum entspricht. Abbildung 3 zeigt wie die Visualisierung beispielsweise aussehen könnte.

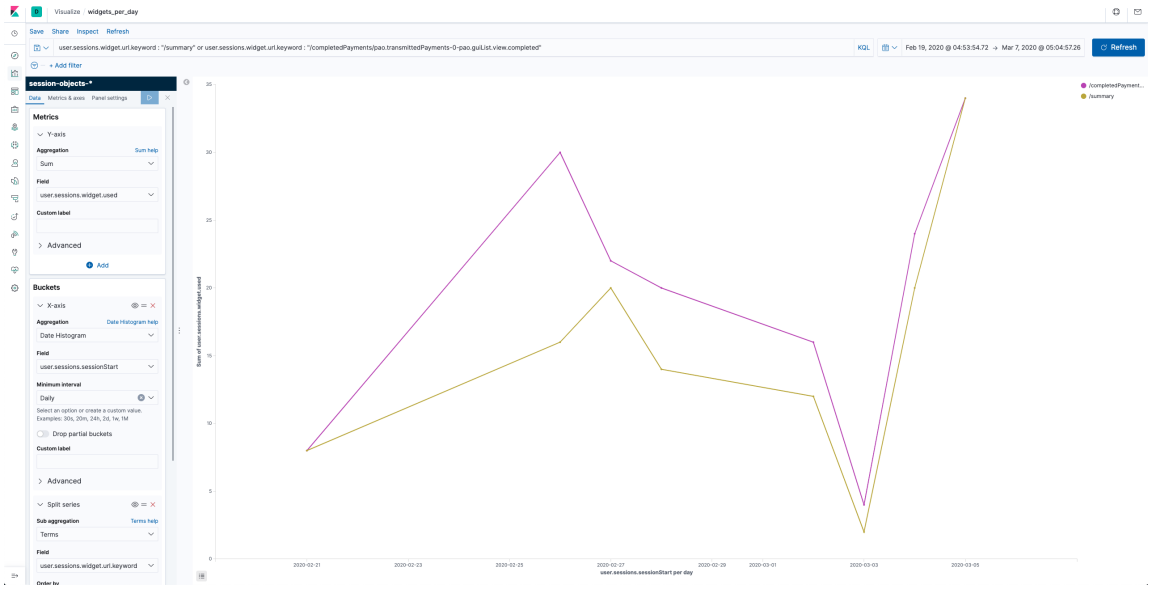

Abbildung 3: Lines Visualisation mit zwei Widgets

4 Auswertung

In diesem Kaptiel wird beschrieben, wie das entwickelte System eingesetzt bzw. getestet wurde.

4.1 Datengenerierung

Die zur Analyse genutzten Daten sind keine tatsächlichen Kundendaten, sondern wurden zum Zweck dieser Abschlussarbeit generiert. Das war notwendig, da in der aktuellen Version des IFPs die Widgetnamen nicht festgehalten werden und man in der aktuellen Implementierung nicht alle Widgets anhand der Logdateien erkennen kann. Selbst bei einer schnellen Auslieferung wäre der Zeitraum zu klein, um genügend Kundendaten zu sammeln.
Mithilfe eines Python Skripts wurden Logfiles für einen Zeitraum von 30 Tagen generiert. Die generierten Logfiles entsprechen einer verkürzten Version der normalen Logfiles: es wurden nur *Incoming Requests* und Trennlinien, wie sie auch in den echten Logfiles vorkommen, in die generierten Dateien geschrieben. Zu Beginn der Entwicklung des Systems wurde der Logstash Filter mit Logfiles entworfen, in denen alle Einträge standen, also auch *Outgoing Responses*. Zu diesem Zeitpunkt hat der Filter erfolgreich die Daten nach unseren Wünschen aus Kapitel 2.2.1 segmentiert. Deshalb war es für die Generierung der Daten ausreichend nur noch *Incoming Requests* zu betrachten, die eine Widgetnutzung repräsentieren. Insgesamt wurden 15 verschiedene Widgets betrachtet.
Um zu prüfen, ob das entwickelte System nach unseren Wünschen funktioniert, müssen die Logfiles zwei Bedingungen erfüllen:

1. Die Einträge in den Logfiles müssen zufällig ausgewählt werden. Die Idee hinter dieser Bedingung ist, dass die Daten in den Logfiles unbekannt sind.

2. Eine Kombination an Widgets muss in verschiedenen Abständen wiederholt in den Logfiles vorkommen. Diese Kombination stellt einen bestimmten Workflow dar, der durch die Suche nach Assoziationsregeln erkannt werden soll. Die einzige Information, die wir zu diesem Punkt verwenden möchten ist die Anzahl an Sessions und die Anzahl an Sessions, die diesen Workflow beinhalten. Durch diese Werte lässt sich der minsupport für den Workflow bestimmen und dient als Hilfe zur Erkennung des Workflows.

Unter berücksichtigung dieser Bedingungen wurde das Skript entwickelt. Eine Ausführung des Python Skripts entspricht einer Session. In dieser Session werden die Widgets, die „benutzten“ werden, zufällig ausgewäht. Das wurde erreicht, indem die beispielhaften *Incoming Requests* als Strings in einem Array gespeichert werden. Pythons RANDOM Funktion liefert eine zufällige Zahl in den Array Grenzen, die bestimmt, welche Zeile geschrieben wird. Zusätzlich hat man die Möglichkeit bei der Ausführung des Skripts eine Reihe an Zahlen zu übergeben, die den Workflow darstellen. Schließlich kann über einen weiteren Parameter eine Zahl übergeben werden, die angibt, welches Datum ge-

loggt wird. Dieser Wert ist standartmäßig auf 0 gesetzt, was dem aktuellen Datum entspricht. Falls dieser Parameter übergeben wird, wird die Zahl zum aktuellen Datum addiert.
Zusätzlich zu dem Python Skript wurde ein Bash Skript entwickelt, das die gesamte Datengenerierung automatisiert. Das Skript simuliert für einen fest gelegten Zeitraum (in unserem Fall 30 Tage) eine zufällige Anzahl an Sessions pro Tag. Außerdem soll ca. jede dritte Session den festen Workflow beinhalten. Welche Widgets benutzt werden wird durch drei zufällige Zahlen bestimmt. Am Ende gibt das Skript auf der Kommandozeile aus, wie viele Sessions insgesamt geschrieben wurden und wie viele davon den Workflow beinhalten.

4.1.1 Durchführung

5 Zusammenfassung

5.1 Fazit

5.2 Zukünftige Anwendungsmöglichkeiten

A Anhang

Zusatzteil 1

Dies ist ein Anhang.

Literatur

Rakesh Agrawal, Tomasz Imieliundefinedski und Arun Swami (Juni 1993). „Mining Association Rules between Sets of Items in Large Databases“. In: *SIGMOD Rec.* 22.2, S. 207–216.

Christoph Beierle und Gabriele Kern-Isberner (2019). „Maschinelles Lernen“. In: *Methoden wissensbasierter Systeme: Grundlagen, Algorithmen, Anwendungen*. Springer Fachmedien Wiesbaden, S. 99–160.

CoCoNet (2020). *(CoCoNet)*. URL: https://www.coconet.de/en/corporate-banking-solutions/payment-cash-management-solution/ (besucht am 05. 05. 2020).

Richard O. Duda, Peter E. Hart und David G. Stork (2001). *Pattern Classification*. John Wiley & Sons, Inc.

Elastic (2020a). *(Logstash filter plugins)*. URL: https://www.elastic.co/guide/en/logstash/current/filter-plugins.html (besucht am 18. 02. 2020).

Elastic (2020b). *Elasticsearch*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/transforms.html (besucht am 07. 04. 2020).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings