Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

SEMANTISCHE KORRELATION VON EIGENSCHAFTEN IN ANDROID-ANWENDUNGEN

Version 1.0 vom 01.10.2019

Johannes Feichtner – johannes.feichtner@a-sit.at

Bei der Analyse von Mobilanwendungen stehen neben einer angenäherten Fassung des originalen Programmcodes oft auch Funktionsbeschreibungen, ein Berechtigungsmodell, UI-Elemente, etc. zur Verfügung. Eine Korrelation dieser Daten mit dem Quellcode würde die sicherheitsorientierte Analyse von Mobilanwendung wesentlich unterstützen, indem kontextuelle Informationen beigesteuert werden könnten, die Aufschluss über den Einsatzzweck von Codefragmenten geben. Eine Verarbeitung, Auswertung und Integration dieser Metadaten in den Analyseprozess ist aufgrund der unterschiedlichen Datentypen zumeist aber nur bedingt möglich.

Im Zuge dieses Projekts sollen unter Zuhilfenahme neuester Technologien aus dem Bereich des maschinellen Lernens Wege erörtert werden, um Programmteile in ihrer Funktionalität miteinander zu verbinden. Durch den Fokus auf relevante Eigenschaften im Programmcode sollen funktional ausschlaggebende Elemente identifiziert und als Funktionsbeschreibung herangezogen werden. Das Ziel ist es, dadurch in natürlicher Sprache Rückschlüsse auf die Funktionalität von Code ziehen zu können. Ein anschließender Vergleich der hergeleiteten Beschreibungen mit der von Herstellern bereitgestellten Beschreibung einer App ermöglicht es, funktionale Diskrepanzen, unvollständige oder inakkurate Aussagen zu identifizieren.

Inhaltsverzeichnis

© A-SIT, Partner für Sicherheitsfragen

1. Einleitung

Stetig wachsende Speicher- und Leistungsreserven moderner Mobilgeräte ermöglichen es Herstellern von Apps, funktional zunehmend komplexere und inhaltlich umfangreichere Anwendungen zu entwickeln. Dass Android-Anwendungen heutzutage mitunter mehrere hundert MB groß sind, ist nicht nur auf die großzügige Einbindung von Grafiken, aufwändig gestalteten UI-Elementen oder eigens angepassten Klangeffekten zurückzuführen. Um die Entwicklung von Apps möglichst effizient zu gestalten, wird sehr oft auf öffentlich bereitgestellte Programmbibliotheken von Drittherstellern zurückgegriffen. Im Regelfall bilden diese Bibliotheken nicht nur eine gewisse Funktionalität ab, sondern stellen diverse Funktionsbausteine bereit, die Hersteller in Apps einbinden können. Die sicherheitsorientierte Analyse von Anwendungen wird dadurch jedoch signifikant erschwert, da aus einer Außenperspektive nur mehr schwer erkennbar ist, *welche Funktionalität* einzelne Codeteile abbilden.

Mit der intensiven Weiterentwicklung neuronaler Netze wurden in den letzten Jahren wesentliche Fortschritte erzielt, um aus einer Unmenge von Daten funktional ausschlaggebende zu identifizieren, sie zusammenzufassen und zu klassifizieren. Eine Anwendung dieser Werkzeuge und Techniken auf den Code von Android-Anwendungen kann dabei helfen, dem steigenden Umfang von Apps zu begegnen, Muster darin zu erkennen und Schlussfolgerungen zu ziehen.

Ein vergleichsweise trivialer Ansatz um die Funktionalität von Codeteilen zu klassifizieren, wäre die Festlegung starrer Regeln und Muster, nach denen Code erkannt werden kann. Angesichts einer Unmenge an API-Methoden des Android-Betriebssystems, Methoden aus Drittbibliotheken und einer daraus resultierenden Vielfalt, wie Hersteller eine gewisse Funktionalität auf unterschiedliche Weise implementieren können, erscheint dieser Ansatz nicht mehr zweckmäßig. Moderne Methoden des maschinellen Lernens, insbesondere im Kontext des Deep Learnings sind darauf ausgelegt, die Erkennung von Mustern so selbstständig vorzunehmen, dass eine Identifikation der Funktionalität von Apps trotzdem möglich sein sollte. Eine **bislang nicht gelöste Herausforderung** dabei ist jedoch, Funktionalität nicht nur möglichst eindeutig zu erkennen, sondern sie auch in natürlicher Sprache zu „beschreiben“. Ist dies das angestrebte Ziel, behilft man sich bei Aufgaben des Machine Learning typischerweise eine „supervised classification“, d.h. ein Datensatz wird manuell analysiert, einzelnen Klassen zugeteilt und damit ein Modell trainiert. Die „Beschreibung“ auf die hintrainiert wird, entspricht dabei in der Regel der manuell festgelegten Klasse. Angesichts der in der Praxis gegebenen Vielfalt an Apps, unterschiedlichen Ausrichtungen und ein somit a priori unvollständiges Modell einer „supervised classification“ verlangt nach anderen Herangehensweisen.

Die akkurate Beschreibung der Funktionalität von Anwendungen spielt auch eine wesentliche Rolle, wenn in Google’s Play Store nach Anwendungen gesucht wird. Neben Screenshots, Reviews und ggf. weiterer Metadaten trägt die Qualität der Beschreibung von Apps wesentlich dazu bei, ob Anwendungen installiert werden oder nicht. Zurzeit gibt es jedoch kein System, das die Beschreibungen von Apps in irgendeiner Form mit dem Code korrelieren würde. Die Freitextform, in der App-Beschreibungen bereitgestellt werden, sowie nichtexistierende Vorschriften bezüglich des Inhalts oder der Länge, führen in der Praxis dazu, dass die Implementierung von Apps nicht zwangsläufig etwas mit dem zu tun haben muss, was Hersteller in den Beschreibungen versprechen. Eine möglichst präzise Beschreibung würde jedoch BenutzerInnen bereits vor der Installation ein klareres Bild davon verschaffen, welche Fähigkeiten Anwendungen mitbringen. Vor allem (in der Praxis kaum anzutreffende) Hinweise, wo und wie Berechtigungen verwendet werden, könnten zu einem erhöhten Sicherheitsbewusstsein („*privacy awareness*“) bei BenutzerInnen beitragen.

Aus Perspektive einer sicherheitsorientierten Analyse von Apps sind Informationen darüber, warum auf gewisse Sensoren, Geräteeigenschaften und Userdaten zugegriffen wird, essentiell, um ein eventuelles Sicherheitsrisiko fundiert bewerten zu können. Ohne diese kontextuellen Informationen über den Verwendungszweck von Mobilanwendungen können objektive Probleme zwar aufgezeigt werden, eine anschließende Einschätzung der darauf resultierenden sicherheitsgefährdenden Konsequenzen ist jedoch nur bedingt möglich. Wenn es also gelingt, die im Code von Apps abgebildete Funktionalität automatisiert zu abstrahieren und zu beschreiben, kann dies zu einer wesentlichen Qualitätssteigerung bei Beschreibungen von Anwendungen führen und nicht zuletzt die Verhaltensanalyse von Apps durch Einbezug des jeweiligen Verwendungszwecks verbessern.

2. Kontextuelle Verhaltensindikatoren in Android-Anwendungen

Um von Mobilanwendungen automatisiert eine Beschreibung herzuleiten, muss zunächst eine fundamentale Frage beantwortet werden: *Welche Eigenschaften einer App beschreiben ihr Verhalten?* BenutzerInnen können diese Frage intuitiv beantworten, indem die Anwendung installiert und ausprobiert wird. App-Hersteller würden für vergleichbare Rückschlüsse hingegen auf den Quellcode einer Anwendung verweisen. Beide Perspektiven beeinflussen die im weiteren vorgestellte Herangehensweise für eine automatisierte Verhaltensmodellierung.

Das **Ziel dieses Projekts** ist es, für die Kernfunktionalität einer Android-App ausschlaggebende Eigenschaften aus Apps zu extrahieren, daraus Rückschlüsse in natürlicher Sprache zu ziehen und, angesichts der Komplexität und Größe heutiger Anwendungen, die rechnerisch benötigte Komplexität in Grenzen zu halten.

In den nachfolgenden Abschnitten wird ein System vorgeschlagen, das der genannten Zielsetzung Rechnung trägt und vom Programmcode Rückschlüsse auf die implementierte Funktionalität zieht. Zunächst werden die Maßnahmen beschrieben, die gesetzt werden müssen, um für das Verhalten von Apps semantisch relevante Eigenschaften aus Apps zu extrahieren. Als „semantische Informationsquellen“ wird dabei auf folgende App-Eigenschaften zurückgegriffen:

1) **Bezeichner von App-Ressourcen („resource identifiers“).**
   Um von Programmcode aus auf beigepackte Ressourcen wie UI-Elemente, Grafiken, oder mehrsprachig hinterlegten Definitionen zuzugreifen, sieht Android die Verwendung von sog. „Resource IDs“ vor, die entsprechende Elemente eindeutig referenzieren. Bei der Entwicklung von Apps legen Hersteller die zu verwendenden Bezeichner selbst fest. Wenngleich ihnen dabei Gestaltungsfreiheit geboten wird, entspricht der gewählte Bezeichner von Ressourcen in der Praxis in der Regel dem tatsächlichen Inhalt.

2) **Werte von Ressourcen („string constants“).**
   Android-Apps verwalten UI-Elemente und Sprachvariablen in Ressourcen, die von zuvor angeführten Bezeichnern referenziert werden. Auch die eigentlichen Inhalte, auf die hin verwiesen wird, liefern semantisch wichtige Indikatoren über den Zweck von Anwendungen. Enthalten UI-Elemente beispielsweise Zeichenketten wie „Neue Überweisung“, „Daueraufträge“ oder Umsätze, ließe sich in Kombination mit dem Programmcode einer App schlussfolgern, dass es sich um eine Anwendung für Finanztransaktionen handeln dürfte. Basierend auf dem Programmcode wäre diese kontextuelle Einordnung nur dann möglich, wenn Muster im Code so spezifisch wären, dass eine anderweitige Verwendung nicht infrage käme. Aufgrund der von Android vorgegebenen Struktur von Anwendungen, die gewisse Kernkomponenten („Activities“, „Services“, „Listener-Callback“-Strukturen) erfordert, ließe sich eine derartige Zuordnung nur sehr unpräzise vornehmen.

3) **Aufgerufene Methoden der Android-API.**
   Durch die verbreitete Verwendung von Programmbibliotheken von Drittherstellern und der breiten Funktionsvielfalt, die heutige Apps abbilden, ist es sehr herausfordernd, den Zweck eines jeden verwendeten Codefragments genau zu bestimmen. Wir postulieren daher, dass das Verhalten von Apps nicht (nur) im Zusammenspiel einzelner Codefragmente bestimmt wird, sondern vor allem in ihrer Interaktion mit dem Betriebssystem und BenutzerInnen. Für die Ableitung einer funktionalen Beschreibung fokussieren wir uns daher im Weiteren auf Aufrufe von APIs des Android-Frameworks. Mit ihrem modus operandi regeln sie Zugriffe auf sensible Userdaten sowie Sensoren und stecken die Funktionalität von Apps daher auch im Hinblick auf sicherheitsrelevante Aspekte und das Berechtigungsmodell klar ab.

Auf Basis dieser drei heterogenen Datentypen wird im Weiteren ein neuronales Netzwerk trainiert, das aus einer semantischen Kombination dieser Features Wörter in natürlicher Sprache ableitet, die das Verhalten von Apps charakterisieren. Um zu verstehen, wie das neuronale Netzwerk entscheidet und welche der gewählten Eigenschaften in Apps tatsächlich ausschlaggebend sind für eine Beschreibung, wird ein Algorithmus eingesetzt, der Entscheidungen des neuronalen Netzwerks nachvollziehbar macht.

3. Ableitung einer Anwendungsbeschreibung

Basierend auf den im vorigen Abschnitt beschriebenen semantischen Eigenschaften von Android-Apps wird in weiterer Folge ein Zusammenspiel von drei neuronalen Netzen vorgeschlagen, die diese Informationen miteinander verbinden und Schlussfolgerungen ermöglichen. Durch den heterogenen Aufbau der verwendeten Daten (Zeichenketten/Wörter/Methodenaufrufe) muss zunächst eine Vorverarbeitung geschehen. Dafür werden die Relationen der Daten untereinander zunächst mithilfe von drei getrennten, neuronalen Netzen abgebildet und die Ausgaben („Outputs“) der drei Netze schließlich im Gesamten verbunden und evaluiert.

3.1. Gesamtübersicht

Abbildung 1 fasst die Trainingsetappen der neuronalen Netzwerke zusammen. Im ersten Schritt werden Archive von Android-Applikationen im APK-Format vorbereitet („Preprocessing“). Hierbei werden die im vorherigen Abschnitt angeführten semantischen Eigenschaften extrahiert und mithilfe von TF-IDF[1] gewichtet. Analog dazu werden die Beschreibungstexte von Anwendungen als TF-IDF-Vektoren [1] dargestellt. Für jedes dieser Inputfeatures wird in weiterer Folge ein separates neuronales Netzwerk trainiert, bei dem alle Neuronen miteinander verbunden sind („dense neural network“). Der Trainingsprozess wird über einige Epochen fortgesetzt und erst abgebrochen, wenn die Trainingsperformance stagniert. Dieses Vorgehen ist notwendig, um zu verhindern, dass sich das Modell zu sehr an die Trainingsdaten anpasst und sie sozusagen auswendig lernt („Overfitting“).

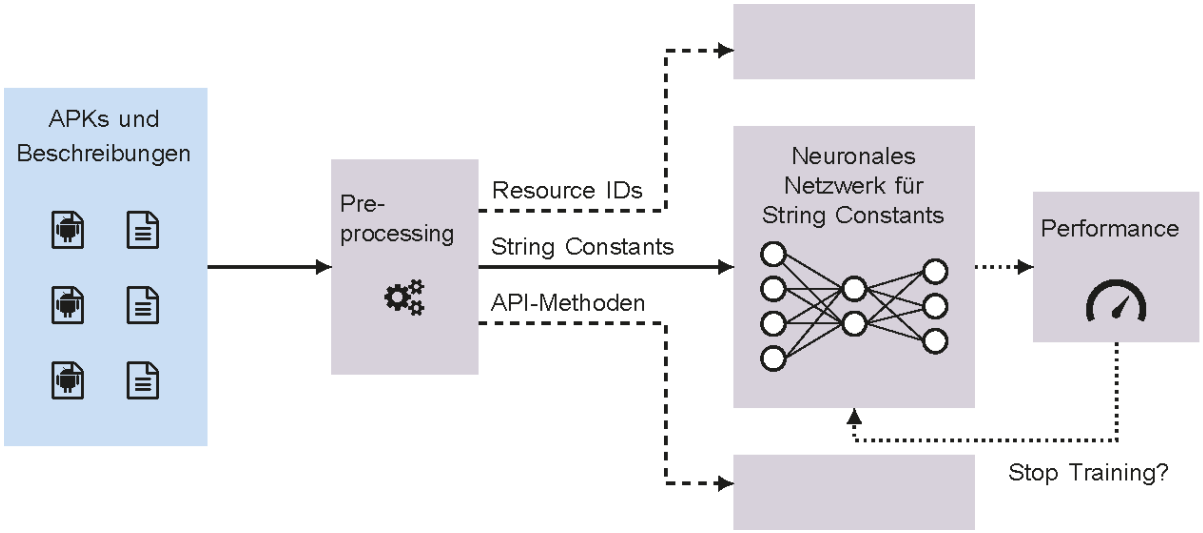

Abbildung 1. Training von drei neuronalen Netzwerken

In der „Prediction“-Phase, dargestellt in Abbildung 2, werden die Modelle mit Test-Datensätzen konfrontiert, die im Training nicht berücksichtigt bzw. gelernt wurden. Die hierfür verwendeten Apps erhalten das gleiche „Pre-Processing“ wie die Apps in der Trainingsphase: Zuvor erstellte TF-IDF-Modelle werden eingesetzt, um „resource identifiers“, „string constants“ und aufgerufene Methoden der Android-API in eine Vektorrepräsentation zu transformieren und den Vektor-Output des Modells wieder zurück zu lesbaren Wörter zu wandeln. Abhängig vom bereitgestellten Input gibt jedes der drei „dense“-Netzwerke schließlich Tokens zurück, die entsprechend ihrer „Wichtigkeit“ absteigend sortiert sind. Die Relevanz bzw. Wichtigkeit der einzelnen Tokens wird zusätzlich über Dezimalwerte zwischen 0 und 1 ausgedrückt, die für jeden vorhergesagten Token zurückgegeben werden.

Wie in Abbildung 2 dargestellt, kann der Output des Netzes zur Vorhersage von Ressourcenwerten beispielsweise aus den Wörtern „sms“, „messenger“ und „friends“ bestehen, wobei der nebenstehende Dezimalwert die Relevanz der Vorhersagen ausdrückt. Mithilfe eines Algorithmus zur Erklärung neuronaler Netze namens SHAP [2] lässt sich für die Output-Tokens herausfinden, welche gegebenen Input-Features für die Vorhersage relevant sind. Jedes der drei Modelle retourniert in der „Prediction“-Phase also eine Liste von Schlüsselwörtern, die das Verhalten einer Android-App auf Basis der verwendeten Ressourcen und aufgerufenen API-Methoden beschreiben.

[1] https://de.wikipedia.org/wiki/Tf-idf-Ma%C3%9F

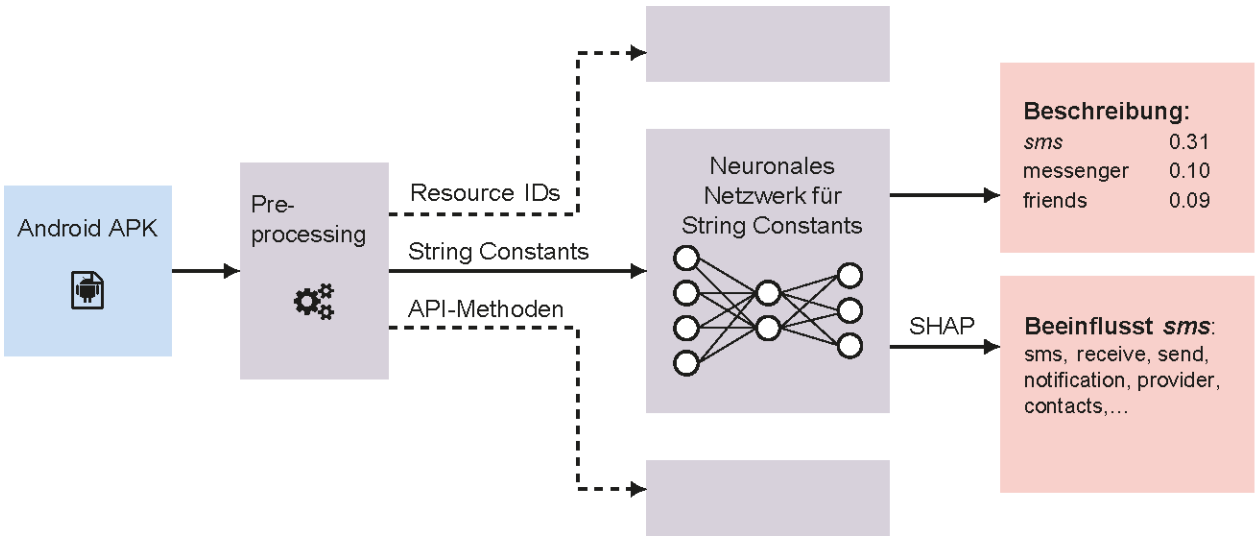

Abbildung 2. Prediction-Phase der drei neuralen Netze.

3.2. Pre-Processing

Um Bezeichner und Werte von Ressourcen mithilfe eines neuronalen Netzwerks verarbeiten zu können, muss zunächst eine geeignete Vektorrepräsentation gefunden werden.

3.2.1. Bezeichner und Werte von Ressourcen

Bezeichner von Ressourcen sind separat abgelegte String-Werte, die aus der Programmlogik heraus referenziert werden, um auf Ressourcen zuzugreifen. Im Gegensatz zu Quellcode sind sie normalerweise nicht obfusziert, sondern so hinterlegt, wie App-Hersteller sie bei der Entwicklung festlegen. Die nachstehende Tabelle zeigt exemplarisch Bezeichner von Ressourcen, wie sie in der Praxis vorkommen. In der Regel tragen diese Bezeichner „sprechende“ Namen und bestehen aus alphanumerischen Zeichen und Unterstrichen.

Beispiel-Bezeichner
pay_btn
select_image_dialog
confirmRemove
welcome_message_1
start_quiz_headline

Eine empirische Analyse mehrerer hundert realer Anwendungen hat aufgezeigt, dass diese Bezeichner in kondensierter Form Aussagen über Aktionen und UI-Elemente ermöglichen. Die Herausforderung ist folglich, diese Werte sinnvoll zu zerlegen („tokenizen“), um semantische Relationen zu erfassen. Ohne Zerlegung ließe sich beispielsweise nicht feststellen, dass die fiktiven Bezeichner „select_image_dialog“ und „select_video_dialog“ ähnliche Aktionen implizieren, die sich nur in „image“ und „video“ unterscheiden. Durch eine Zerlegung zu „select“, „video“, „image“, „dialog“ und Verknüpfung der Wörter untereinander als sog. „n-grams“ ist eine sinnvollere Abbildung möglich.

Android ermöglicht es, dass UI-Elemente in Apps in unterschiedlichen Sprachen angezeigt werden. Hersteller müssen dazu separate Sprachdefinitionen für sämtliche Elemente hinterlegen, die bei der Ausführung über „Resource Identifiers“ referenziert werden. Diese Mehrsprachigkeit ist für die in diesem Projekt angestrebte Ableitung einer App-Beschreibung jedoch hinderlich, da der Output des Netzwerks nur in einer Sprache abgefasst und nicht Wörter verschiedenster Sprachen enthalten soll. Um alle Wörter aller vorkommenden Sprachen sinnvoll in Zusammenhang setzen zu können, bräuchte der Trainingsprozess zudem einen semantisch ausgewogenen Korpus über alle Apps. Für das weitere Vorgehen fokussieren wir uns daher ausschließlich auf englische Wörter und identifizieren sie anhand ihres Vorkommens in Wörterbüchern.

Um Bezeichner und Werte von Android-Apps für die Verarbeitung mit dem neuronalen Netz aufzubereiten, werden daher alle Unterstriche verworfen und die vorkommenden Wörter als in Kontext stehende Sequenz verarbeitet. URLs, HTML-Code, Namen von Java-Packages werden anhand ihrer Syntax erkannt und verworfen. Das Endresultat einer Verarbeitung von Bezeichnern und Werten von Android-Ressourcen besteht schließlich aus einer Liste englischsprachiger Wörter, die in Wörterbüchern vorkommen und nur aus alphanumerischen Zeichen bestehen.

3.2.2. Methoden der Android-API

Um herauszufinden, welche Android-APIs von Anwendungen verwendet werden, gilt es festzustellen, wo und in welchem Kontext ein Aufruf stattfindet. Eine Inspektion des „Call Graphs“ von Apps ermöglicht es herauszufinden, wo entsprechende Aufrufe stattfinden, wie oft sie passieren und in welchem Kontext bzw. welcher Methode dies stattfindet.

Abbildung 3 zeigt die Vorgehensweise, um die Verwendung von API-Aufrufe festzustellen und ihre Häufigkeit zu zählen. Zunächst wird das Archiv einer Android-Anwendung dekompiliert und ein Call Graph auf Basis statischer, expliziter Aufrufe gebildet. Dynamische, zur Laufzeit aufgelöste Methodenaufrufe können folglich nicht berücksichtigt werden. Um dennoch einen möglichst akkuraten Call Graph zu erhalten, fügen wir weitere Kanten hinzu, die Vererbungen anzeigen. Darüber hinaus werden implizite Aufrufe erfasst (sog. „Listener-Callback-Methoden“), indem auf vorgefertigte Listen[2] zurückgegriffen wird, in denen derartige Aufrufe und ihre Verweise erfasst sind. Um ein entsprechendes Mapping für die aktuelle Version des Android-Frameworks zu generieren, wird das Werkzeug EdgeMiner [3] von Cao et al. eingesetzt. Da Android-Apps keine vordefinierten Einsprungspunkte kennen, werden als Ausgangspunkt der Modellierung des Call Graphs die im *AndroidManifest.xml* einer jeden App definierten *Activities* verwendet. Durch diesen Ansatz lässt sich sicherstellen, dass die Erhebung verwendeter API-Methoden wirklich nur jene berücksichtigt, die von Anwendungen auch tatsächlich aufgerufen werden und nicht als „toter Code“ enthalten sind.

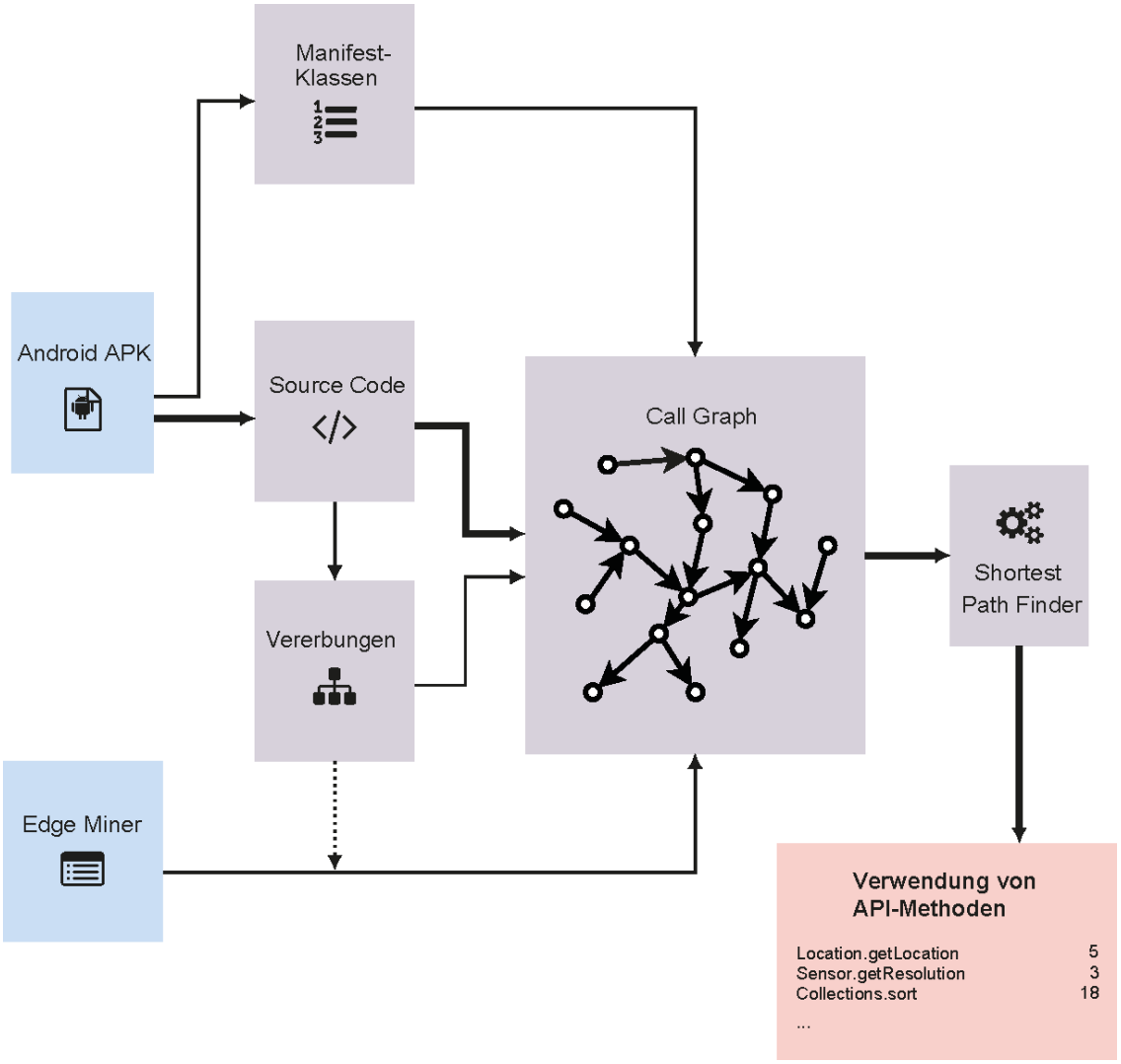

Abbildung 3. Strategie zur Auflösung verwendeter Android API-Calls.

[2] http://yinzhicao.org/EdgeMiner/

3.2.3. App-Beschreibungen

Die angestrebte Ausgabe des zu entwerfenden Machine Learning-Modells ist eine Beschreibung in natürlicher Sprache. Dazu ist es notwendig, den vom Hersteller bereitgestellten Beschreibungstext einer Anwendung zusammen mit den aus der App extrahierten semantischen Features zu trainieren. Um die Wichtigkeit einzelner Wörter in Beschreibungstexten zu gewichten, verwenden wir das TF-IDF-Verfahren. Da es, wie eingangs erwähnt, keine Formvorschriften für Beschreibungstexte gibt, kann es passieren, dass Texte sehr kurz sind und für unsere Zwecke dementsprechend wenig hilfreich wären. Wir maximieren daher den Informationsgehalt von Beschreibungen, indem wir das TF-IDF-Verfahren nicht nur auf einzelne Wörter anwenden, sondern auch kurze Phrasen, wie z.B. take photo“ oder „SD Card“, damit erfassen.

Für die Trainingsphase des Netzwerks werden zudem sämtliche Stoppwörter wie etwa Präpositionen, Artikel und Satzzeichen entfernt. Damit das Netzwerk semantisch identische Wörter wie „photo“ und die Pluralform „photos“ nicht als unabhängig voneinander betrachtet, wird auf alle Wörter „Stemming“[3] angewandt. Dadurch werden Wörter auf ihren Wortstamm reduziert. Dies führt in der Praxis dazu, dass die resultierenden Wörter für Menschen oft nur mehr schwer interpretierbar sind. Eine Ausgabe von „gestemmten“ Wörtern würde also dem eingangs gesetzten Ziel widersprechen, lesbare Beschreibungsfragmente auszugeben.

Als Abhilfe propagieren wir daher einen Greedy-Algorithmus, der originale Wörter aus „gestemmten“ wiederherstellt. Dazu merken wir im Zuge des Stemmings von Wörtern das Originalwort und zählen die Häufigkeit des Vorkommens. Die Idee dahinter ist es, dass Wörter die öfter in Beschreibungen verwendet werden, auch tendenziell jene sind, die durch Stemming auf ihren Wortstamm verkürzt wurden. Wie in Tabelle exemplarisch dargestellt, kann der nach Stemming erhaltene Token etwa „locat“ lauten und im Original sowohl für „location“, „located“, „locating“ oder „locale“ stehen. Indem im Zuge der Trainingsphase mit aufgezeichnet wird, welche Wörter wie oft vorkommen, lässt sich über die Häufigkeitsverteilung auf das wahrscheinlichste Wort im Original schließen. Naturgemäß muss diese Übereinstimmung nicht zwangsläufig dem originalen Wort entsprechen und kann mitunter sogar falsch sein. Für die Herleitung von Beschreibungstexten ist der Output „location“ jedoch viel intuitiver und aussagekräftiger als der Wortstamm „locat“.

„gestemmtes“ Wort	**Wort im Original**	**Häufigkeit**
locat	**location**	**890**
	located	418
	locating	356
	locale	208
effect	**effects**	**286**
	effect	109
	effecting	75

Tabelle 2. Beispiel einer häufigkeitsbasierten Rückwandlung von gestemmten Wörtern.

Jede Beschreibung einer Android-App wird schließlich als Liste von Tokens repräsentiert, die in ein TF-IDF-Modell überführt werden. Mit der zuvor genannten Erweiterung auf Phrasen bzw. Wortkombinationen enthält dieses Modell neben einzelnen Tokens auch 2-grams und 3-grams. Da Stemming auch auf n-grams angewandt wird, kann so auch mit höherer Verlässlichkeit beim Rückführen gestemmter Wörter auf originale. Die ursprüngliche Wortkombination „English locale“ beispielsweise führt somit auch beim Invertieren von Stemming wieder zur gleichen Phrase und nicht etwa zum tendenziell weniger häufig vorkommenden 2-gram „English location“.

3.3. Netzwerkarchitektur

Nach Vorbereitung der unterschiedlichen Features, mit denen ein neuronales Netzwerk trainiert werden kann, sowie der App-Beschreibung als Output, kann die eigentliche Netzwerkarchitektur konstruiert werden. Wie in Abschnitt 3.1 konzeptionell dargelegt, erstellen wir drei ähnliche Modelle, die Tokens einer App-Beschreibung ausgeben und als Input jeweils die aus Apps extrahierten, semantischen Eigenschaften verwenden.

[3] https://de.wikipedia.org/wiki/Stemming

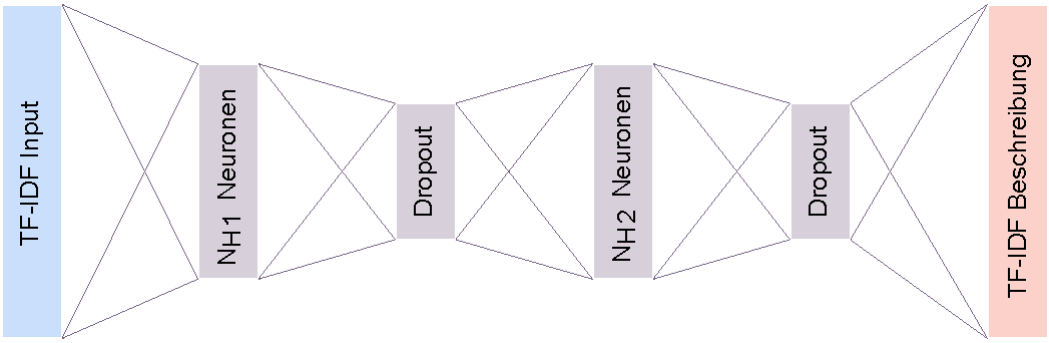

Abbildung 4. Architektur eines Dense Neural Networks.

Abbildung 4 zeigt die Architektur eines „Dense Neural Networks“, das für alle drei Input-Typen verwendet wird. Die „Bag of Words“-Darstellung der TF-IDF-Vektoren legt die Position der Wörter in der Inputmatrix fest. Der Vektor eines gewissen Wortes wird also immer der gleichen Vektorzelle zugewiesen. Diese Eigenschaft ermöglicht die Verwendung einer sog. „dense“ Netzwerkstruktur, die im Gegensatz zu „Convolutional Neural Networks“ oder LSTMs die Position und Sequenz von Wörtern während des Trainings nicht verändert oder abstrahiert. Anstatt nur Layers der Reihe nach miteinander zu verbinden, ist bei „dichten“ Verbindungen jeder Reihe mit jedem nachfolgenden direkt verbunden [4]. Wenngleich He et al. diesen Ansatz eigentlich für Aufgaben im Bereich „Computer Vision“ entworfen haben, hat eine Untersuchung von Ruder et al. [5] aufgezeigt, dass sich die Methodik ebenso für den NLP-Bereich eignet.

Zwischen den „dense layers“ des Netzes wird ein Dropout-Layer [6] zur Regularisierung verwendet. Nachdem der Output eines jeden Neurons eine Gleitkommazahl ist, kann eine Regressionsanalyse vorgenommen werden, bei der für den letzten Layer („Output“) eine lineare Aktivierungsfunktion, sowie die mittlere quadratische Abweichung („Mean Squared Error“) als Loss Function eingesetzt werden können. Weitere Hyperparameter, mit denen das Netzwerk trainiert wird, sind in Tabelle 3 zusammengefasst.

Optimizer	Adam
Batchgröße	32
Initialisierung	Uniform
Loss-Funktion	Mean-Squared Error
Aktivierungsfunktion bei hidden layers	ReLU
Aktivierungsfunktion final	Linear
Early Stopping	6 Epochen

Tabelle 3. Gewählte Hyperparameter des Dense Neural Networks.

Die Größe des Inputs und Outputs hängt von der Wörterbuchgröße der TF-IDF-Modelle ab. Die Grenzen werden dabei durch die Häufigkeit des Vorkommens einzelner Tokens als Resource Identifiers oder bei Aufrufen von API-Methoden eingeschränkt. Praktisch spielt vor allem die untere Grenze eine wesentliche Rolle: Ist sie zu hoch, verlieren wir Informationen, die das Netzwerk für einen möglichst präzisen Output bräuchte. Ist sie zu niedrig, erinnert sich das Netzwerk zu „leichtfertig“ an Tokens, die nur selten vorkommen und vergleichsweise „irrelevant“ sind.

Die Auswahl geeigneter Hyperparameter spielt daher eine essentielle Rolle, um Laufzeit und Präzision zu steuern. Um geeignete Werte zu finden, wurden im Rahmen dieses Projekts mehrere Netze mit einem bis drei hidden layers trainiert, die jeweils 1000 bis 15000 Neuronen umfassten. Mit der Dropout-Rate wurde in einem Bereich von 0 bis 40% experimentiert, wobei 30% als bekanntermaßen, praxistaugliche Referenz gelten [7]. Bei der Suche nach passenden Hyperparametern wurde jeweils auf die Wörterbuchgröße und Performance der Modelle Acht gelegt. Die minimale Auftrittshäufigkeit von Wörtern wurde dabei mit 2% festgelegt, die maximale über alle Apps mit 20%. Das heißt in der Praxis, dass ein Token in mindestens 2% aller Apps vorkommen muss, jedoch in maximal 20% vorkommen darf, um im Wörterbuch berücksichtigt zu werden. Dadurch verhindern wir sowohl ein Auswendiglernen der Trainingsdaten („Overfitting“) als auch eine zu starke Konzentration auf selten vorkommende, tendenziell unwichtige Wörter.

4. Evaluierung

Die nachfolgende Evaluierung verfolgt in erster Linie das Ziel, die Performance des entworfenen Netzwerks im Hinblick auf Anwendung mit realen Android-Apps zu überprüfen. Mithilfe eines Algorithmus zur Erklärung der Entscheidungen neuronaler Netze soll ein Einblick gewonnen werden, auf Basis welcher Features Vorhersagen getroffen werden und wie direkt oder indirekt diese z.B. an das Vorkommen gewisser Input-Wörter gebunden sind.

4.1. Datensatz

Um von gegebenen Archiven von Android Apps auf die Elemente von Beschreibungen schließen zu können, werden für das Training sowohl Anwendungen als auch zugehörige Beschreibungen benötigt. Da ein Crawlen des Google PlayStore technischen Beschränkungen unterworfen ist, verwenden wir für die nachfolgende Analyse die Sammlung des PlayDrone-Projekts [8].

18.000 gesammelte Apps mit Beschreibungen werden unterteilt in ein Trainings-, Validierungs- und Testset. Die Aufteilung erfolgt nach dem Zufallsprinzip, wobei 1000 Apps für Testzwecke ausgewählt werden und aus den verbleibenden 80% für das Training und 20% für Validierung eingesetzt werden. Darüber hinaus wird für alle Samples sichergestellt, dass die resultierenden TF-IDF-Vektoren für Input und Output jeweils Werte über null beinhalten. Dadurch lässt sich von vorneherein verhindern, dass Samples verwendet werden, die keinen erkennbaren Informationsgehalt aufweisen. Ein Beispiel dafür sind etwa Cross-Platform-Apps, bei denen die Implementierung nicht als Dalvik Bytecode vorliegt, sondern in Form einer JavaScript-Webanwendung realisiert ist.

4.2. Erklärung von Vorhersagen

Die Ausgabe der trainierten neuronalen Netze besteht aus einer Liste von Wörtern, die in einer Beschreibung enthalten sein sollten. Durch die Zusammensetzung vieler unterschiedlicher Schichten und Abstraktion während des Lernprozesses, sind von neuronalen Netzen vorgenommene Entscheidungen in der Regel schlecht nachvollziehbar. Um diesem Umstand beizukommen, setzen wir auf SHAP [9], einem Algorithmus zur Erklärung von Entscheidungen von Modellen neuronaler Netze.

Gibt das Modell zur Vorhersage von „Resource Identifiers“ beispielsweise das Wort „dictionary“ aus, kann durch SHAP herausgefunden werden, welche Faktoren diese Vorhersage beeinflussen. Eine intuitiv gut erfassbare Relation wäre etwa zu Input-Tokens wie „search“, „word“ oder „translate“ gegeben. Wenn SHAP jedoch Input-Token aufzeigt, die logisch gesehen keinen Sinn machen, impliziert das, dass das Modell Korrelationen in ähnlichen Apps gelernt hat, jedoch nicht in Bezug auf ein konkretes Feature der App. In der Praxis passiert dies etwa, wenn Hersteller Codeteile in einer Vielzahl von Apps verwenden (etwa in der Form von Bibliotheken von Drittherstellern) und das gleiche Feature in allen oder vielen Beschreibungstexten auftritt. Bei der Arbeit mit einem Datensatz, der aus realen, potentiell unzureichenden App-Samples und Beschreibungstexten besteht, ist es daher essentiell zu verstehen, wie gewisse Vorhersagen zustande kommen.

Die Erklärung mittels SHAP passiert auf Basis einzelner Samples. Der Algorithmus wird also auf einzelne Vorhersagen angewandt und liefert daher individuelle Erklärungen pro Vorhersage. Die dabei zurückgegebenen Indikatoren bestehen aus sog. „Shapley values“, die für jedes Feature des Inputs und Outputs bestimmt werden und anzeigen, wie relevant gewisse Inputfeatures für die Aktivierung gewisser Outputneuronen sind. Zu diesem Zweck nimmt der Algorithmus mehrere untereinander ähnliche Inputs entgegen und nutzt die Unterschiede zwischen ihnen zur Berechnung der Relevanz für ein gewisses Endergebnis. Um diese Relevanz zu bestimmen, propagiert der Algorithmus Werte durch alle Layer des Netzwerks und summiert sie auf. Die Ausgabe ist eine Liste von „Shapley values“, die nach Sortierung anzeigt, welche Inputfeatures für die Vorhersage eines gewissen Wortes relevant waren.

Die möglichen Ausgaben des Netzwerks sind, wie in Abschnitt 3.3 beschrieben, durch das TF-IDF-Modell limitiert. Um für eine gegebene Android-App die sinnvollsten Wörter einer Beschreibung vorherzusagen, wenden wir SHAP auf alle vorhergesagten Tokens der drei Modelle an, sortieren sie nach angezeigter Wichtigkeit und erhalten dadurch eine Liste relevanter Inputfeatures.

4.3. Fallstudie: Vevo App

Die Illustration welchen Output die drei Netze zurückgeben, wird im Folgenden exemplarisch anhand von Vevo, einer App für Videostreaming, veranschaulicht.

Tabelle 4 listet Schlüsselwörter einer Beschreibung auf, die die drei Modelle für die gegenständliche App vorhersagen. Die Tabelle listet für jedes der drei Modelle die vom Netzwerk ausgegebenen Token auf und daneben eine Gewichtung, die die festgestellte Relevanz in Relation zu anderen Wörtern darstellt. Indem die TF-IDF-Vektoren auch n-grams abbilden, bestehen die Tokens sowohl aus einzelnen Wörtern, als auch aus Kombinationen mehrerer, etwa im Falle von „tv channels“. Ebenso erkennbar ist, dass Wörter wie etwa „video“, „watch“ und „content“ in mehreren Modellen vorhergesagt werden und dadurch insgesamt als wichtiger gesehen werden kann.

Output: Vorhersage / Beschreibung	**Input: Resource IDs**	**Input: Resource-Werte**	**Input: API-Calls**
video	0.295	0.132	0.131
tv show	0.261		
kids	0.227		
music	0.148		
watch	0.123	0.041	0.098
songs	0.081		
subscription	0.066		
tv channels		0.136	
movies		0.033	0.049
new		0.025	
content	0.064	0.024	
live		0.024	0.056
stories		0.024	
tv			0.094
news			0.062
sports			0.046
us			0.045

Tabelle 4. Vorhersage von Beschreibungstoken für die Videostreaming-App Vevo.

Eine gewichtete Darstellung der gleichen Ergebnisse findet sich in Abbildung 5. Die Word-Cloud fasst die am stärksten gewichteten Vorhersagen für die gegebene App in einer Grafik zusammen. Die Positionierung der Elemente erfolgt zufallsbedingt und spielt keine Rolle.

Abbildung 5. Word-Cloud mit gewichteten Vorhersagen.

Die Gegenüberstellung einzelner Wörter mit dem eigentlichen Zweck der Vevo-App zeigt eine hohe Übereinstimmung. Abgesehen vom Wort „video“, wird die Charakteristik einer Plattform zum Streaming und Tauschen von Videos in den Worten bzw. Kombinationen „tv show“, „tv channels“, „movies“, „subscription“, „music“, „live“ und „content“ deutlich. Die auf TV bezogenen Vorhersagen zeigen auf, dass unsere Modelle nicht in der Lage sind, zwischen traditionellem Fernsehen und Online-Streaming zu unterscheiden. Da die Beschreibungsvorhersagen auf Basis eines sehr großen, zuvor gelernten Trainingskorpus generiert wurden, ist davon auszugehen, dass die

neuronalen Netze „verstanden“ haben, in welches Feld bzw. welchen Bereich von Apps die gegebene Vevo-App fällt und auf Video bezogene Anwendungen intern als solche in einer Art Cluster zusammengefasst wurden. Aus den in Tabelle 4 präsentierten Ergebnissen geht außerdem hervor, dass die Beschreibungstokens, die aus API-Calls abgeleitet werden, im Vergleich mit den anderen Netzen deutlich allgemeiner formuliert sind. Die Domäne zu der die App gehört, lässt sich zwar weiterhin erahnen an Wörtern wie „movies“, „live“ und „tv“, jedoch wurden keine n-grams wie „tv channels“ oder „tv shows“ gelernt.

Zusammenfassend kann festgehalten werden, dass alle drei Modelle intuitiv gut nachvollziehbare Vorhersagen für Wörter treffen, die die gegebene App charakterisieren. Insbesondere die Tatsache, dass es auch Überlappungen in den Vorhersagen gibt, zeigt, dass die für die Analyse gewählten Eigenschaften in Apps ein auf Korrelation basiertes Lernen ermöglichen und hinreichend individuell sind, um die Funktionalität von Apps zu beschreiben. Durch geeignete Wahl von Hyperparametern für die neuronalen Netze und „Early Stopping“ wurde davor bereits sichergestellt, dass sich die Modelle nicht zu sehr an die Trainingsdaten anpassen oder sie auswendig lernen. Indem die drei Modelle unabhängig voneinander teilweise die gleichen Vorhersagen treffen, lässt sich auch die qualitative Bestätigung ableiten, dass die Menge, Auswahl und Vorverarbeitung von Features aus den jeweiligen Apps mithilfe von TF-IDF nach analytisch nützlich Gesichtspunkten erfolgt ist.

5. Fazit

Im Zuge dieses Projekts wurde der Fokus auf die Korrelation semantischer Eigenschaften in Android-Anwendungen mithilfe von Machine Learning gelegt. Das Ziel bestand darin, ein System zu erarbeiten, um auf Basis einer gegebenen Android-App über deren Inhalt oder Zweck möglicherweise nichts bekannt ist, eine möglichst realitätsnahe Beschreibung ihrer Funktionalität herzuleiten. Diese Beschreibung sollte in natürlicher Sprache intuitiv erfassbar sein.

Nach anfänglicher Erörterung der Problemstellung und Zielsetzung wurde eine Aufstellung der Features vorgenommen, die in einer Android-App enthalten sind und in der Ausführung von Apps eine wesentliche Rolle spielen. Durch einen Fokus auf „Resource IDs“ wurde eine Eigenschaft identifiziert, die durch die vom Hersteller häufig angewandte „Obfuscation“ nicht umfasst wird. Die Einbeziehung von Konstanten, die BenutzerInnen gegenüber bei der Ausführung angezeigt werden, sowie die Extraktion aufgerufener API-Methoden zielen direkt darauf ab, die Funktionalität der jeweiligen App möglichst gut abzubilden. Der Aufruf einer API-Methode, um etwa die Kamera anzusteuern, lässt direkte Rückschlüsse auf das Verhalten von Apps zu.

Angesichts der Heterogenität der gewählten Features war es notwendig, ein Verfahren zu finden, das die Daten dennoch in Relation zueinander setzen und für die Weiterverarbeitung mit Machine Learning aufbereiten kann. Neben der individuellen Verarbeitung, etwa durch Bilden eines Call Graphs und der Extraktion relevanter Knoten bei API-Aufrufen, hat die Anwendung des TF-IDF-Verfahrens hat eine Transformation von Features zu Vektorrepräsentation ermöglicht. Anschließend wurde ein „dichtes“ neuronales Netzwerk entworfen und die verwendeten Hyperparameter an die vom Netzwerk abzubildenden Aufgaben angepasst.

Für die nachfolgende Evaluierung der Architektur wurden mehrere tausend Android-Apps mitsamt ihren Beschreibungen bezogen, aufbereitet und drei neuronale Netze trainiert. Um einen Einblick zu bekommen, warum gewisse Wörter vorhergesagt werden, wurde ein Algorithmus angewandt, der Entscheidungen von neuronalen Netzen nachvollziehbar macht. Die Fallstudie einer App für Videostreaming hat veranschaulicht, welche App-Beschreibungen sich in der Praxis aus den Archiven von Android-Apps ableiten lassen und inwieweit sie plausibel sind.

Die Erkenntnisse aus diesem Projekt zeigen einen innovativen Ansatz, um die Funktionalität von Apps ohne vorherige Kenntnis von Metadaten automatisiert herzuleiten. Das dabei verfolgte Konzept kann eingesetzt werden, um bestehende Beschreibungen von Apps zu verbessern, Diskrepanzen aufzudecken und Anwendungen kontextuelle einzuordnen. Dies wiederum unterstützt sicherheitsorientierte Analysen von Mobilanwendungen und trägt zu einem besseren Verständnis bei, welche Funktionalität in Apps tatsächlich enthalten ist.

Literaturverzeichnis

[1] T. Mikolov, I. Sutskever und K. Chen, „Distributed Representations of Words and Phrases and their Compositionality," *Neural Information Processing Systems NIPS,* 2013.

[2] S. Lundberg und S.-I. Lee, „A Unified Approach to Interpreting Model Predictions," *Advances in Neural Information Processing Systems 30,* p. 9, 2017.

[3] Y. Cao und Y. Fratantonio, „ EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," *Proceedings of the Network and Distributed System Security Symposium (NDSS),* 2015.

[4] H. Kaiming und X. Zhang, „Deep Residual Learning for Image Recognition," *CVPR,* 2016.

[5] S. Ruder und J. Bingel, „Sluice networks: Learning what to share between loosely related tasks," *CoRR abs/1705.08142,* 2017.

[6] N. Srivastava, G. Hinton und A. Krizhevsky, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research 15,* 2014.

[7] Y. Kim, „Convolutional Neural Networks for Sentence Classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP),* 2014.

[8] N. Viennot, E. Garcia und J. Nieh, „A Measurement Study of Google Play," *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems,* 2014.

[9] S. Lundberg und S.-I. Lee, „A Unified Approach to Interpreting Model Predictions," *Advances in Neural Information Processing Systems 30,* 2017.