**Philosophische Fakultät III**

**Sprach-, Literatur- und Kulturwissenschaften**

**Institut für Information und Medien, Sprache und Kultur (I:IMSK)  
Lehrstuhl für Informationswissenschaft**

Informationssysteme

INF-M 06.02 (B.A.)

SS 2017

Leitung:  Prof. Dr. Bernd Ludwig

Thema 7 – Lernen eines Naive-Bayes-Klassifikators für Restaurant-Bewertung in Tripadvisor

Philipp Hartl

1804734

Medieninformatik/Informationswissenschaft

[Philipp1.Hartl@stud.uni-regensburg.de](mailto:Philipp1.Hartl@stud.uni-regensburg.de)

Grundidee, Datenerhebung, ARFF Dateierstellung, Inputverarbeitung, Dokumentation & Interpretation

Andreas Hilzenthaler

1747245

Medieninformatik/Informationswissenschaft

[Andreas.Hilzenthaler@stud.uni-regensburg.de](mailto:Andreas.Hilzenthaler@stud.uni-regensburg.de)

Datenerhebung, WEKA-Implementierung & UI

Florian Kaindl

1806636

Medieninformatik/Informationswissenschaft

florian.kaindl.@stud.uni-regensburg.de

Custom-Classifier-Implementierung & mathematische Hintergründe (Dokumentation)

**Aufbau & Grundgedanke**

Wir haben uns für eine Implementierung des Naive-Bayes-Klassifikators in Java entschieden. Dies liegt einerseits an der Verfügbarkeit der WEKA Bibliothek in Java (oder Python), anderseits an der relativ einfachen Gestaltung einer UI in Java für den Nutzer, außerdem wurde im Laufe des Studiums bereits öfters mit Java programmiert. Grundsätzlich wollten wir unseren eigenen Classifier schreiben, allerdings wird es schwierig diesen auf Richtigkeit zu überprüfen, deshalb haben wir hier den Weg gewählt sowohl einen eigenen Klassifikator als auch den bereits durch WEKA bekannten, in Java portieren, Klassifikator einzubauen. Dies ermöglicht uns einen Vergleich beider Klassifikatoren sowie eine Interpretation der Unterschiede, sowie der „richtigeren“ Werte von WEKA anhand unseres Datensatzes.

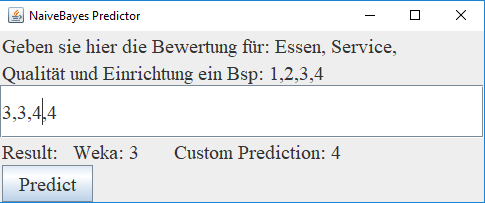
**Datenerhebung:**

Leider stellt TripAdvisor keinerlei öffentlich zugängliche API-Schnittstelle zur Verfügung und bietet nur Businesskunden Zugang zu ihren Daten, akademische Auswertungen sind explizit nicht erwünscht (<https://developer-tripadvisor.com/content-api/FAQ/>). Deshalb war es notwendig alle notwendigen Daten, per Hand, den jeweiligen Restaurant-Bewertungsseiten zu entnehmen. Die Informationen werden im ARFF-Format der „University of Waikato“ gespeichert, da dies für die wekabasierten Java Bibliotheken einfacher zu parsen ist. Am Anfang der Datei werden die Attribute initialisiert, diese sind basieren auf der Aufgabenstellung:

* Essen (bzw. Küche)
* Service
* Qualität
* Einrichtung
* Gesamtbewertung

Letztere wird von WEKA automatisch als Classifier interpretiert. Insgesamt stehen 100 Datensätze aus ganz Europa zur Verfügung, wobei die komplette Bewertungsspanne in einer Likert-Skala repräsentiert wird. Falls bei einigen Bewertungen nicht ganzzahlig waren, wurde mathematisch korrekt in eine Likert-Repräsentation umgewandelt. Es wurde ebenfalls darauf achtgegeben, dass sowohl genügend niedrige sowie hohe Bewertungen vorhanden sind.

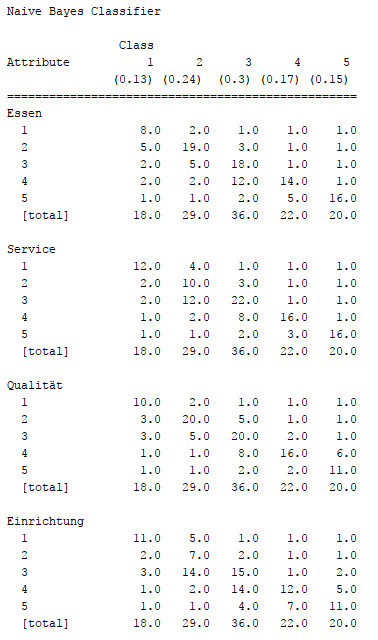
**Nutzerinterface**

Das UI der Anwendung basiert auf der Java-Swing Bibliothek. Eine kurze Beschreibung weißt den Nutzer auf die richtige Syntax der Eingabe hin, welche man in einem Eingabefeld tätigen kann. Um die Berechnung der Klassifikatoren zu starten muss der Nutzer den „Predict“-Button betätigen. Anschließend wird die Eingabe auf richtige Syntax überprüft, hierbei wird die Länge der Eingabe geprüft, sowie eine kurze Regex Abfrage um zu gewährleisten, dass nur Likert-Skala valide Zahlen eingegeben werden können. Anschließend wird eine neue ARFF-Datei erstellt bzw. überschrieben, wenn schon vorhanden, in die, die Benutzereingaben eingetragen werden. Diese wird dann von beiden Klassifikatoren weiterverarbeitet bzw. eingelesen.

Das User Interface der Anwendung (Ausgabe mit kurzem Datensatz)

**WEKA-basierter Klassifikator**

Über die öffentlich zugänglichen Bibliotheken für Java, werden der Trainingsdatensatz sowie der von der UI generiert Test- bzw. Vorhersagedatensatz über den bereitgestellten ARFF-Fileloader eingelesen. Aus diesem Trainingsset wird nun mit Hilfe von WEKA ein Naive-Bayes-Klassifikator gelernt, hierbei werden alle Zeilen einzeln eingelesen und der Naive-Bayes verbessert sich mit jeder neuen Information. Mit Hilfe des gelernten Modelles kann nun mit *nb.classifyInstance* die Vorhersage für das vom Nutzer eingegebene Restaurant berechnet werden. Der Ausgabewert des Klassifikators hat den Index vier, dieser wird ausgelesen und gespeichert. Das Ergebnis wird schlussendlich an die Benutzeroberfläche weitergegeben.

**Ergebnisse & Interpretation:**

Der abschließende Classifier (Konsolenausgabe des Klassifikators – print(nb)) zeigt, dass vor allem bei hohen Bewertungen immer Essen und Service eine Rolle spielen und diese Bewertungen weniger abhängig von der Qualität sowie der Einrichtung sind. Bei niedrigeren Bewertungen ist dies nicht mehr so eindeutig feststellbar. Grundsätzlich gibt es nur wenig bis keine Ausreißer (z.B. Vier Sterne Essensbewertung bei Gesamtbewertung Zwei), jedes Attribut ist in seiner jeweiligen Klasse am häufigsten vertreten, allerdings bei der mittleren (drei )Bewertung ist die Differenz nach oben (vier) am geringsten (z.B. Einrichtung). Dies lässt den Eindruck erwecken, dass die Unterscheidung in diesen Attributen für die Nutzer in diesem Bereich oft nicht eindeutig ist.

Ergebnisse des Weka Naive Bayes-Classifier

**Custom Naive Bayes Klassifikator (CNB)**

In der Klasse „CustomNaiveBayes“ ist ein eigener Klassifikator implementiert, der dem UI Methoden bereitstellt, um auf derselben Datengrundlage wie der WEKA-Klassifikator eine Zuordnung des Testvektors zu ermitteln. Er bedient sich der Weka-Bibliotheken nur zur Auslesung der arff-Dateien. Im Aufbau ist er eng an eine Python-Implementation des Algorithmus zu finden unter „https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/“ gehalten, ist aber auf die Projektsprache Java und ihrer Datentypen sowie die Erfordernisse unseres Projekts angepasst. So kann die Entscheidung des CNB mit der der Weka-Implementierung direkt abgeglichen werden.

**CNB – Mathematische Erläuterung der Funktionen**

1. **Lernen der Trainingsdaten**

Wenn das Programm gestartet wird, liest der CNB die Datei „TrainingsDaten.arff“ aus, und speichert die Daten als eine Liste aus Integer-Arrays, wobei jedes Array einen Bewertungsvektor darstellt, dessen erste vier Werte Teilbewertungen sind, und der letzte Wert die Gesamtbewertung, also die Klasse des Vektors darstellt. Beispiel: [2,3,3,2,2] hat die Teilbewertungen 2,3,3,2 und die Gesamtbewertung 2.

Diese Liste wird als Parameter in die Methode *summarizeByClass* übergeben, die zunächst die Methode *separateByClass* aufruft, und ihr wiederum die Liste übergibt. Die teilt die enthaltenen Vektoren nach ihren letzten Werten und damit ihren Klassenzugehörigkeiten auf, und speichert sie in getrennten Listen in einer neuen Liste *separated*, deren Indizes als Klasse verstanden werden können, z.B. würde separated.get(0) alle Bewertungsvektoren, die zur Gesamtbewertung 1 gehören zurückliefern, separated.get(1) alle Vektoren der Kategorie 2 usw.

Zurück in *summarizeByClass* wird diese *seperated*-Liste auf die Mittelwerte und Standardabweichungen der Attribute jeder Klasse untersucht, indem die einzelnen Vektorenlisten nach und nach der *summarize*-Methode übergeben werden. So erhält diese also z.B. alle Bewertungsvektoren, die der Klasse „2“ zugeordnet sind, und teilt sie nach Attribut (Einzelbwertung) auf, sozusagen spaltenweise, um aus diesen Bewertungen je einen Mittelwert (Durchschnitt) und eine *Standardabweichung* davon zu berechnen, zu der je die Methoden *mean* und *stdev* herangezogen werden. Diese *summaries* werden in *summarizeByClass* in dann letztendlich in eine wieder nach Klasse aufgeteilte Liste gesteckt und zurückgegeben.

Nun liegt also eine Liste aus fünf Listen vor, eine für jede Klasse, die wiederum für jedes Attribut einen Mittelwert und eine Normalverteilung bereithalten. So hat der Algorithmus aus den Trainingsdaten „gelernt“, welche Werte typisch für jede Gruppe sind.

1. **Klassifizieren eines neuen Bewertungsvektors**

Wenn nun ein Vektor (Array aus vier Teilbewertungen) durch den User eingegeben wurde, kommt die *predict*-Methode zum Einsatz, die unter Heranziehen oben besprochener *summaries* und des Bewertungsvektors eine Klassifikation als Integer (Klasse 1-5, bzw. intern als Indizes 0-4) ausgibt. Dazu erstellt die Funktion eine HashMap, die für einen Key (die Klasse) eine Wahrscheinlichkeit der Zugehörigkeit des Bewertungsvektors speichert. Dazu ruft sie die Methode *calculateClassProbabilities* auf, die aus den *summaries* für jede Klasse nach und nach die Mittelwerte und Standardabweichungen eines Attributs ausliest, und diese mit dem entsprechenden Attribut des Input-Vektors in die *calculateProbability*-Methode speist. Sie berechnet also nun die Wahrscheinlichkeit dafür, dass das Input Attribut einen gegebenen Wert hat, wenn der Mittelwert und die Standardabweichung für diesen Wert in dieser Klasse vorliegen. Der dazu benutzte Algorithmus ist die Gauß-Funktion, wir gehen also zum Zweck der Klassifikation davon aus, dass die Werte normalverteilt sind.

Die *calculateProbability*-Methode wird in *calculateClassProbabilities* einmal für jedes Attribut im Input-Vektor aufgerufen, und die zurückgelieferten Wahrscheinlichkeiten zu einer Gesamtwahrscheinlichkeit für die aktuelle Klasse aufmultipliziert und in die Hashmap eingetragen. Diese Hashmap liegt dann der *predict*-Methode vor, die anschließend nur noch überprüft, für welchen Key (Klasse) die höchste Gesamtwahrscheinlichkeit in der Map vorliegt, und liefert den Key als Ergebnis, also also Klassifikation zurück.

1. **Auswertung**

Beim Testen des CNB stellen wir folgendes fest:

Die Klassifikationen sind tendenziell nachvollziehbar, es gibt aber Ausreißer (3,1,3,4 -> 5). Bessere Ergebnisse liefert die Weka-Implementierung. Wo die Fehler im CNB sind, bleibt noch abschließend zu klären, wir nehmen aber an, dass a) Weka nicht dieselbe (Gaußsche) Formel zur Berechnung von Einzelwahrscheinlichkeiten verwendet und b) unsere Implementierung der Gauß—Funktion nicht ganz sauber ist.

|  |  |  |
| --- | --- | --- |
| **Zufallsgenerierte Werte** | **Weka** | **CNB** |
| 4,1,2,3 | 2 | 1 |
| 4,1,3,5 | 3 | 5 |
| 1,1,3,3 | 1 | 1 |
| 1,2,3,3 | 2 | 2 |
| 2,1,2,2 | 2 | 1 |
| 3,2,2,2 | 2 | 2 |
| 4,2,2,2 | 2 | 2 |
| 3,1,2,2 | 2 | 1 |
| 1,3,2,1 | 2 | 2 |
| 3,1,3,4 | 3 | 5 |
| 3,4,5,3 | 3 | 5 |
| 3,2,2,4 | 3 | 2 |
| 3,3,3,4 | 3 | 3 |
| 3,4,5,2 | 3 | 3 |
| 3,3,4,4 | 3 | 5 |
| 3,1,3,4 | 3 | 5 |
| 4,4,4,3 | 4 | 4 |
| 1,2,4,4 | 4 | 5 |
| 5,1,2,4 | 5 | 3 |

Dies entspricht einer Übereinstimmung von 42%, was deutlich besser ist als pures Raten der Bewertungen (1 zu 5 -> 20%). Bei einer kürzeren Datei (50%) nimmt dies deutlich ab, hier ist nur noch 5% (1 aus 19) Übereinstimmung, wobei WEKA größtenteils gleichbleibt. Dieser massive Unterschied legt eine gute Skalierung des eigenen Klassifikators nahe. Bei größeren Datenmengen fallen bereits angesprochene Unterschiede wie die Implementierung der Gauß-Funktion nicht mehr so stark ins Gewicht.

1. **Sonstiges**

Weitere Methoden wie *getPredictions* und *getAccuracy* liegen vor, um die Verarbeitung vieler Inputvektoren zu ermöglichen, und die Genauigkeit der Klassifikation anhand von Testdaten zu berechnen. Sie kommen zum Stand der Abgabe nicht zum Einsatz. Ebenso wurde Code auskommentiert, der dem Aufteilen der Trainingsdaten in randomisierte Trainings- und Testdaten zu Testzwecken dient. Tests mit diesen Funktionen zeigten, dass ein höherer Anteil von Trainingsdaten meist auch eine höhere Genauigkeit beim Bewerten der Testdaten einherbrachte, was auf eine grundlegende Richtigkeit der Implementierung hinweist.