Experimentaufbau:

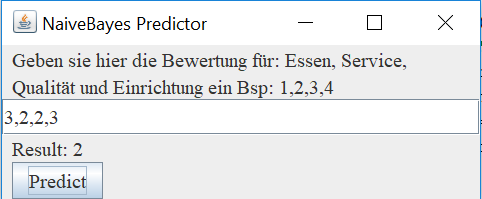
Wir haben uns für das Experiment dazu entschieden einen Naive Bayes Classifier, der die Gesamtbewertung vorhersagt, mit Hilfe von Java zu entwickeln. Dazu verwendeten wir die Weka- Libarary der University of Waikato.

Die Trainingsdaten für den Naive Bayes Classifier der Restaurantbewertungen wurden auf der Internetseite Tripadvisor gesammelt. Da die Tripadvisor API leider für Studenten nicht zugänglich war wurden die Trainingsdaten per Hand in das für Weka spezifische Arff-File geschrieben.

Das Arff- File ist folgendermaßen aufgebaut: Am Anfang der Datei werden die Attribute gesetzt, was in unserem Fall Essen, Service, Qualität, Einrichtung und Gesamtbewertung war. Das letzte Attribut wird im Arff-Format automatisch zum Classifier, was somit die Gesamtbewertung gewesen ist.

Die Trainingsdaten bestehen aus ca. 80 verschiedensten Restaurantbewertungen, die von 5-Sterne Bewertungen bis hin zu 1-Sterne Bewertungen reichen.

Um die Usability zu erhöhen wurde ein kleines User Interface integriert. Dieses wurde mit Java der Java Swing-Libarary integriert und besteht aus einem Beschreibungslabel, einem Texteingabefeld, einem Ergebnisausgabefeld und einem Button, um die Berechnung der Gesamtbewertung zu starten. In der UI-Klasse wir neben der Erstellung des UI auch die Eingabe des Inputfeldes verarbeitet. Die Eingabe des Inputfelds wird in ein Arff- File als Datensatz eingeschrieben und ein neues Testdaten Dokument entsteht. Zudem wird im Eingabefeld überprüft, ob die Eingabe valide ist.



Das User Interface der Anwendung

Für die Berechnung existiert die NaiveBayes-Klasse. In dieser werden zuerst zwei Arff- Dokumente, die aus Trainingsdaten und Testdaten bestehen, mit einem Loader eingelesen.

Danach wird der Classifier auf Gesamtbewertung gesetzt werden. Daraufhin wird ein Naive Bayes Classifier-Model mit Hilfe der Trainingsdaten gebildet, damit das Programm von selbst die Vorhersage der Gesamtbewertung lernen kann. Dieses liest die Trainingsdaten einzeln ein bis das Ende der Datei erreicht ist und gibt das fertige Model zurück.

Die Funktion „nb.classifyInstance(test.instance(0));“ berechnet durch das zuvor erstellte Modell die Gesamtbewertung der Testdaten aus. Danach wird noch der Classifier der TestDaten.arff auf den soeben berechneten Wert gesetzt. Da der Classifier den Index 4 im ersten Testdatensatz besitzt wir dieses als Ergebnis gespeichert und kann von der UI ausgegeben werden.

**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, zB. 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 zB. 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 in 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, ich nehme aber an, dass a) Weka nicht dieselbe (Gaußsche) Formel zur Berechnung von Einzelwahrscheinlichkeiten verwendet und b) meine Implementierung der Gauß—Funktion nicht ganz sauber ist.

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

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 Traingsdaten in randomisierte Trainings- und Testdaten zu Testzwecken dient. Tests mit diesen Funktionen zeigten, dass ein höherer Anteil von Trainigsdaten meist auch eine höhere Genauigkeit beim Bewerten der Testdaten einherbrachte, was auf eine grundlegende Richtigkeit der Implementierung hinweist.