Machine Learning

Klassifikation von numerischen Werten mit Java

Max Wenk
03. Juli 2023

THG-Schopfheim
Projektarbeit Informatik
Fachlehrer: T. Roths

Machine Learning

Klassifikation von numerischen Werten mit Java

Kurzfassung

In diesem Text wird eine Problemstellung vorgestellt und beschrieben, bei dem eine Blume (Lilie) verschiedenen Arten zugeordnet werden soll. Dabei gibt es drei verschiedene Arten, zwischen denen unterschieden werden soll, und es stehen nur quantitative Daten von bereits bestimmten Blumen aus dem “Iris”-Datensatz von R.A. Fisher zur Verfügung. Es wird ein Lösungsansatz vorgestellt, bei dem eine durchschnittliche Blume berechnet wird, die dann mit anderen Blumen zur Klassifikation überprüft werden kann. Dieser Ansatz wird auch als Machine Learning, insbesondere als Supervised Learning bezeichnet. Der Text behandelt die Vor- und Nachteile dieses Ansatzes und die konkrete Implementierung eines solchen Algorithmus in Java.

Machine Learning

Klassifikation von numerischen Werten mit Java

Inhaltsverzeichnis

Machine Learning

Klassifikation von numerischen Werten mit Java

Problemstellung

Es wird angenommen, wir haben eine Blume der Gattung Lilie (*Iris*) gefunden, und diese Blume soll einer speziellen Art an Lilie zugewiesen werden. Dabei soll zwischen drei unterschiedlichen Arten unterschieden werden.

Die drei Arten zwischen denen unterschieden werden soll sind: die Borsten-Schwertlilie (*Iris Setosa*), die Verschiedenfarbige Schwertlilie (*Iris Versicolor*) und Virginische Schwertlilie (*Iris Virginica*).

Die gefundene Blume soll nun klassifiziert werden. Es wird angenommen, dass nur einzelne quantitative Daten zu anderen Blumen vorhanden sind, die exakt zugeordnet werden können. Diese Daten entsprechen in diesem Fall je der Länge und Breite des Blüten- und Kelchblattes (in cm). Diese Daten wurden bereits in einem "*Iris*" Datensatz von R.A. Fisher aus dem Jahr 1936 festgehalten. Dieser Datensatz enthält jeweils die Länge und Breite einer Blume und ordnet ihnen eine der drei Lilienarten zu (Daten wurden empirisch erhoben). Insgesamt werden in dem Datensatz 150 Blumen klassifiziert.

Als nächstes kann nun versucht werden, die gefundene Blume, an Hand der Länge und Breite des Blüten- bzw. Kelchblattes, einer bereits klassifizierten Blume aus dem Datensatz zuzuordnen. Dieses Verfahren ist jedoch zum einen umständlich, da für jede neue gefundene Blume dieses Verfahren wiederholt werden muss, und zum anderen ist dieses Verfahren sehr ungenau, da es sich bei der zugeordneten Blume um einen Extremwert handeln kann, oder die Blume kann keiner konkreten Art zugewiesen werden, da die Blume beispielsweise bei den Maßen des Blütenblattes mit einer Art übereinstimmt, die Maße des Kelchblattes jedoch mit einer anderen Art übereinstimmen.

Lösungsansatz (Design Architekturmodell)

Um dem Problem der Ungenauigkeit entgegenzuwirken, könnte eine ideale Blume für jede Art bzw. Klasse berechnet werden. Diese idealen Blumen würden aus dem vorhandenen Datensatz bestimmt werden. Die gefundene Blume würde dann mit den verschiedenen idealen Blumen verglichen werden. Damit könnte die gefundene Blume quasi gleichzeitig mit allen Blumen aus dem Datensatz verglichen werden. Ein weiterer Vorteil dieses Ansatzes ist, dass die idealen Blumen nur einmal bestimmt werden müssen, und dadurch jede weitere gefundene Blume nur mit den bereits bestimmten idealen Blumen verglichen werden muss.

Ein weiterer Vorteil, der daraus folgt, ist, dass größere Datensätze verwendet werden können, ohne dass die Laufzeit überproportional, im Verhältnis zur Größe des Datensatzes, steigt.

Dieser Ansatz kann als *Machine Learning* (eine Form von schwacher künstlicher Intelligenz) angesehen werden, da der Algorithmus, der für diesen Ansatz benötigt wird, lernt Eigenschaften (*Features*) zu differenzieren, und verschiedenen Arten/Klassen zuzuweisen. Der Algorithmus kann aus Daten, die bereits erhoben wurden, auf neue Ergebnisse schließen, ohne dafür explizit programmiert zu sein. In diesem Fall wird dieser *Machine Learning* Ansatz auch *Supervised Learning* ("Überwachtes Lernen") genannt. Dabei werden aus bekannten Daten Muster und Zusammenhänge extrahiert. Dabei werden die Zusammenhänge zwischen den Eingabedaten (*Predictors*) und den Ausgabedaten (*Label*/"*Zielvariable*") aus dem Datensatz betrachtet, wodurch ein Muster gefunden werden soll. Das Ziel ist, dass der Algorithmus eine richtige *Zielvariable* nur an Hand von *Predictors* vorhersagen kann. Die Funktionsweise zwischen dem Trainieren

eines Modells und der Erweiterung auf neue, unbekannte Daten wird in *Abb. 2: Funktionsweise eines Machine Learning Programms* veranschaulicht.

Umsetzung (Design)

Datenvorverarbeitung

Zuerst muss der (bereits vorhandene) Datensatz verarbeitet werden. Dazu wird dieser eingelesen und "bereinigt" bzw. formatiert. Bei diesem Schritt werden beispielsweise ein Index oder ein Header herausgefiltert. Diese Information muss angegeben werden, da es das Programm sonst vor Probleme mit der Datenverarbeitung stellt (ein Header könnte noch gefiltert werden, bei einem Index würde dies schwerer werden, da das Programm nicht erkennen kann, ob der Index unter Umständen ein *Predictor* ist).

Des Weiteren muss der Datensatz auch in einer speziellen Form sein. Die Anzahl der *Predictors* kann beliebig sein und muss nicht angegeben werden. Jedoch muss der letzte Wert des Datensatzes, pro Zeile, eine *Zielvariable* sein (da es sich um einen *Supervised Learning* Algorithmus handelt). Die *Predictors* müssen eine numerische Größe ohne Einheit haben, und die *Zielvariable* kann beliebigen Datentyps sein. Jedoch muss, wie bereits oben genannt, Information zum Vorhandensein eines Headers bzw. Index gegeben werden.

Im nächsten Schritt werden noch die Daten aus dem Datensatz in Trainings- und Testdaten unterteilt, um im Folgenden den Algorithmus trainieren zu können (beispielsweise 70% Trainingsdaten und 30% Testdaten).

Feature Engineering (Festlegen von Features) Metrik

Damit Muster in dem Datensatz erkannt werden können, müssen die Eigenschaften (*Predictors*) jeder Klasse aus dem Datensatz beachtet werden. In diesem Fall entspricht ein *Predictor* einem *Feature* (mit jeweils der gleichen Gewichtung). Die Gewichtung der *Features* könnte angepasst werden, wenn es sich bei den *Predictors* um nicht-lineare Zusammenhänge handelt (siehe *Bohnen Erweiterung*), was bei dem oben genannten Beispiel jedoch nicht der Fall ist. Mit Gewichtung ist der Faktor gemeint, mit dem ein *Predictor* verwendet wird.

Lernphase (Modellieren)

Während der Lernphase wird das eigentliche Modell des Algorithmus trainiert. Dazu werden Muster in den Trainingsdaten gesucht, die dann einer Klasse zugewiesen werden. In diesem Fall (Beispiel aus der *Problemstellung*), wird eine "ideale" bzw. durchschnittliche Blume berechnet. Dazu werden die gleichen *Predictors* bzw. *Features* einer Klassen miteinander addiert und dann durch die Anzahl der Einträge pro Klassen geteilt (es wird der Durchschnitt berechnet). Damit kann für jede Klasse (Blumenart) ein Durchschnitt für jedes *Feature* bestimmt werden.

Testphase

Während der Testphase wird der Algorithmus auf den Testdatensatz angewandt. Dadurch kann bestimmt werden, wie "gut" der Algorithmus trainiert wurde. Wurden beispielsweise alle *Predictors* des Testdatensatzes richtigen *Zielvariablen* zugeordnet, handelt es sich um ein gutes Modell. Werden beispielsweise nur 20% der *Predictors* richtig zugeordnet, sollten unter Umständen die *Features* anders gewählt werden (bzw. die Gewichtung kann

angepasst werden), oder es kommt zu einem Problem mit dem Datensatz (siehe *Probleme bei der Klassifikation* oder *Bohnen Erweiterung*).

In diesem Fall werden die *Zielvariablen* durch die “Entfernung” der *Predictors* zur durchschnittlichen Blume bestimmt (Berechnung der Diagonale in n-Dimensionen), wobei die kleinste Entfernung dem wahrscheinlichsten Ergebnis entspricht (siehe *Abb. 4.1: Darstellung einer Klassifikation*).

Interpretation der Ergebnisse (Evaluation des Modells)

Die Evaluation des trainierten Modells folgt direkt auf die Testphase. Dabei wird festgestellt, wie “gut” das Modell ist. Dieser Schritt muss manuell vorgenommen werden. Um diesen Schritt zu vereinfachen, können verschiedene *Confusion Matrices* ausgegeben werden. Je nachdem, wie viele Einträge (Zeile in dem Datensatz bzw. Daten pro Blume) richtigen Klassen zugeordnet wurden, ist der Algorithmus entsprechend gut bzw. schlecht. Um dies richtig evaluieren zu können, wurden drei verschiedene Arten von *Confusion Matrices* implementiert:

- Confusion Matrix: Eine “normale” *Confusion Matrix* wird ausgegeben (mit der Größe: Anzahl der Klassen x Anzahl der Klassen), in der jeweils alle klassifizierten Ergebnisse in einer “Tabelle” ausgegeben werden.
- Simplified Confusion Matrix: Eine angepasste *Confusion Matrix* wird ausgegeben (mit der Größe: 2 x Anzahl der Klassen), mit der angegeben werden kann, wie viele Datenpunkte pro Klasse richtig klassifiziert worden sind (es gibt pro Klasse nur *True True* und *False True*).
- Normalized Confusion Matrix: Eine *Confusion Matrix*, wie bereits oben beschrieben, wird ausgegeben, mit dem Unterschied, dass die Wahrscheinlichkeit (numerisch von 0-1) für jede Vorhersage angegeben wird.

Nutzung der Interpretation

Nachdem die Ergebnisse aus der Testphase evaluiert wurden, muss daraus ein Nutzen gezogen werden, um den Algorithmus bzw. das Modell verbessern zu können. Dies kann hauptsächlich durch eine andere Gewichtung der *Features* vorgenommen werden. Jedoch ist dafür manuelle Arbeit des Anwenders gefordert. Natürlich müssen auch Grenzen und Limitierungen des Algorithmus bzw. Modells beachtet werden (diese werden in *Probleme bei der Klassifikation* und *Bohnen Erweiterung* beschrieben).

Implementierung

Alle genannten Dateien und Datensätze wurden für die Implementierung des Algorithmus verwendet und sind somit auch mit jeweiligen Namen im Programm vorhanden. Das Programm ist in Java geschrieben. Zur Veranschaulichung der Beziehungen zwischen den einzelnen Dateien in dem Programm, kann *Abb. 1: Dateistruktur zur Verwendung des Algorithmus* betrachtet werden. Die Dateistruktur ist so gewählt, dass ein potentieller Nutzer den Algorithmus nicht beliebig verändern kann, sondern nur mit Hilfe von öffentlichen Methoden auf Teile des Programms Einfluss nehmen kann.

Datensatz bearbeiten

Betrachtet man den "*Iris*"-Datensatz ("Iris.csv"), so lässt sich erkennen, dass die jeweiligen Einträge nach Klassen bzw. Blumenart sortiert sind. Eine derartige Sortierung könnte das Programm vor Probleme stellen, da der Datensatz, wie bereits zuvor beschrieben, in Trainings- und Testdaten unterteilt wird. Daraus folgt, dass Klassen, die beispielsweise am Anfang gelistet werden, nur im Trainingsdatensatz vorhanden sind und somit nicht getestet werden können, oder dass Klassen nur am Ende gelistet sind, und somit nicht in dem Modell trainiert werden können. Dadurch wird das Modell nicht über die Existenz von später genannten Klassen informiert, was zu einem Fehler des Programms führt.

Um dieses Problem zu umgehen, wird der Datensatz manuell zufällig sortiert ("Iris_unordered.csv"). Dieser neue Datensatz wird dann für alle folgenden Berechnungen verwendet. Das genannte Problem wird nochmals in *Schlechte Unterteilung zwischen Trainings- und Testdaten* aufgegriffen.

Datenvorverarbeitung

Wie bereits beschrieben, werden in diesem Prozess, wenn vorhanden, ein Header bzw. Index aus dem Datensatz entfernt, um damit arbeiten zu können, und die Daten werden in Trainings- und Testdaten unterteilt.

Dazu werden zuerst die Anzahl der Reihen und Zeilen aus dem Datensatz extrahiert. Dies kann durch eine einfache Iteration, des Scanner Moduls, durch den Datensatz geschehen. Da es sich bei den *Predictors* um Fließkommazahlen (Float) handelt, und bei den *Zielvariablen* um Wörter (Char Array → String), müssen in Java zwei unterschiedliche Daten Arrays initialisiert werden, da in Java nur Daten gleichen Typs in einem Array gespeichert werden können.

Bei dem Daten Array für die *Predictors* wird ein zweidimensionaler Array des Float-Datentyps initialisiert. Im nächsten Schritt wird dann der Datensatz Zeile für Zeile gelesen, und die jeweiligen Zeilen werden in ihre Spalten aufgeteilt und dann in den Array geschrieben, wobei die letzte Spalte nicht übertragen wird, da an dieser Stelle die *Zielvariable* steht. Ist ein Header vorhanden, wird die erste Zeile übersprungen, und die berechnete Gesamtlänge des Arrays wird verkürzt. Diese Extraktion kann mit den Funktionen des Scanner Moduls vorgenommen werden. Ist ein Index vorhanden, so wird die erste Spalte jeder Zeile übersprungen, und die Gesamtbreite des Arrays wird verkürzt.

Danach werden die *Zielvariablen* in einen Array übertragen. Dabei wird nur ein eindimensionaler Array des String-Datentyps benötigt. Dazu wird jeweils die letzte Spalte in jeder Zeile gelesen und übertragen. Dieser Prozess kann wieder mit dem Scanner Modul umgesetzt werden. Bei der Extraktion der *Zielvariablen* muss nur das

Vorhandensein eines Headers beachtet werden. Die Funktionen für diese Prozesse sind in “CSVread.java” programmiert.

Das Unterteilen des Datensatzes in Trainings- und Testdaten wird, in der Hauptklasse, in der “dataSubdivision” Funktion zusammengefasst. Diese Funktion greift hauptsächlich auf die “ProcessData.java” Datei zu. Dazu wird der Prozentsatz an Trainingsdaten (durch den User vorgegeben) in dezimal mit der Anzahl an Zeilen aus dem Datensatz multipliziert und dann abgerundet, da es bei einer Multiplikation auch zu Nicht-Natürlichen-Zahlen kommen kann, diese aber nicht als Index für den Datensatz (Array) verwendet werden können. Danach wird ein Trainingsdatensatz bis zu dieser Grenze aus dem Datensatz extrahiert, und ein Testdatensatz ab dieser Grenze. Die Grenze liegt bei dem Programm standardmäßig bei 70%, kann jedoch vom User verändert werden. 70% wurden gewählt, da es sich um ein “gutes” Verhältnis aus Trainings- und Testdaten handelt, sodass es weder zu wenig Datenpunkte zum Trainieren gibt, als dass es zu wenige Datenpunkte zum Testen gibt. Dieser Prozess wird jeweils für die *Predictors* und die *Zielvariablen* durchgeführt.

Feature Engineering

Die *Features* wurden bereits im Voraus festgelegt, da die Zusammenhänge zwischen den einzelnen *Predictors* linear und alle Angaben in der gleichen Einheit sind.

In dieser Phase wird auch der Algorithmus zur Klassifikation festgelegt:
Der hier verwendete Algorithmus zur Klassifikation bestimmt eine ideale Blume, indem ein Durchschnittswert aus allen Blumen einer Klasse gebildet wird. Soll dann eine andere Blume klassifiziert werden, wird der euklidische Abstand der aktuellen Blume zu allen Durchschnittswerten der jeweiligen Klassen berechnet. Der geringste Abstand entspricht dann dem wahrscheinlichsten Ergebnis. Jeder *Predictor* entspricht dann einer Dimension bei der räumlichen Berechnung des Abstandes. In *Abb. 4.1 und Abb 4.2: Darstellung einer Distanz Klassifikation* wird eine zweidimensionale Klassifikation dargestellt.

Lernphase

In dieser Phase wird durch den Trainingsdatensatz iteriert, und die jeweiligen *Features* pro Dateneintrag werden addiert. Sobald alle Einträge aus dem Datensatz beachtet wurden, wird der Durchschnitt berechnet, indem der addierte Wert je *Feature* durch die Anzahl an Einträgen geteilt wird. Die dabei berechneten Werte entsprechen den *Features* der idealen bzw. durchschnittlichen Blume. Dabei wird unterschieden, zu welcher Klasse ein Dateneintrag gehört (die Durchschnitte für die jeweiligen Klassen werden separat berechnet).

Dieser Prozess wird in “ClassificationOfFloatValues.java” mit der Funktion “distanceClassification” realisiert. Dabei wird ein neues Objekt generiert, in dem alle temporären Variablen zur Berechnung des Durchschnitts gespeichert werden (die Funktionen befinden sich in “DistanceClassification.java”).

Testphase

Diese Phase ist die Grundlage für die Evaluation des Modells. Dafür wird durch den Testdatensatz iteriert, und es wird für jeden Dateneintrag der euklidische Abstand zu den durchschnittlichen Blumen je Klasse berechnet (siehe *Abb. 4.1 und Abb 4.2: Darstellung einer Distanz Klassifikation*) und in einem Array gespeichert. Danach wird bestimmt,

welcher Abstand der geringste ist, da dies dem wahrscheinlichsten Ergebnis entsprechen würde. Diese Information wird in einem weiteren Array gespeichert, der die Information zur Wahrscheinlichkeit der Zuordnung aller Klassen pro Dateneintrag enthält. Aus diesen Daten kann dann bestimmt werden, wie gut der Algorithmus ist bzw. wie oft er richtig gelegen ist, was die Grundlage für die nächste Phase darstellt.

Dieser Schritt wurde mit der Funktion “DistanceClassification.java/ testClassificationModel” implementiert.

Interpretation der Ergebnisse

Die Interpretation kann, wie üblich bei der Beurteilung von Klassifikationen, durch eine *Confusion Matrix* erfolgen. Bei einer *Confusion Matrix* können binäre Datenpunkte in einer Tabelle dargestellt werden. Um sinnvolle Vorhersagen treffen zu können, werden drei verschiedene Arten von Confusion *Matrices* im Programm implementiert (siehe *Abb. 3: Variationen von Confusion Matrices*). Um diese Matrizen berechnen bzw. darstellen zu können, werden aus den Daten aus dem Testdatensatz Zielvariablen vorhergesagt und mit den tatsächlichen Zielvariablen verglichen. Trifft die Vorhersage zu, so ist die Vorhersage korrekt, und wenn nicht, ist sie falsch. Die drei *Confusion Matrices* sind:

- Confusion Matrix: Eine normale *Confusion Matrix*, in der jede Vorhersage einer Klasse zugewiesen wird. Daraus folgt, dass die diagonal verlaufenden Werte (von oben links, nach unten rechts) die korrekte Klassifikation angeben.
- Simple Confusion Matrix: Eine vereinfachte *Confusion Matrix*, in der nur richtig und falsch klassifizierte Werte pro Klasse angegeben werden.
- Normalized Confusion Matrix: Die gleiche Matrix wie die erste, jedoch sind die Angaben nicht absolut, sondern werden als Wahrscheinlichkeit der Klassifikation (0-1) angegeben.

Dieser Teil des Programms wird in “DATA_evaluation.java” implementiert. In den jeweiligen Funktionen sind auch die jeweiligen Befehle zur Ausgabe der *Confusion Matrices*. Um die Effizienz des Programms zu erhöhen, wird zuerst die normale *Confusion Matrix* berechnet, und die anderen Matrizen werden dann aus dieser Matrix berechnet.

Nutzung der Interpretation

Bei der Anwendung des Algorithmus auf den Testdaten werden, wie bereits beschrieben, alle Elemente bzw. Werte für Blumen den richtigen Klassen/Arten zugewiesen. Werden die *Confusion Matrices* aus dieser Klassifikation betrachtet, so folgt daraus, dass der Algorithmus in den Testdaten fehlerfrei agiert, da alle Testwerte richtig zugewiesen werden. Jedoch könnte dies auch auf eine fehlerhafte Implementierung des Algorithmus zurückgeführt werden. Zur Überprüfung kann der Datensatz manipuliert werden, indem beispielsweise der Datensatz “Iris_unordered_2.csv” verwendet wird, bei dem eine *Zielvariable* verändert wurde.

Diese Veränderung wird korrekt als falsch in einer der Confusion Matrices erkannt. Daraus folgt, dass die Implementierung nicht fehlerbelastet ist.

Anmerkung zur Implementierung

Der Algorithmus zur Klassifikation und alle anderen benötigten Funktionen befinden sich in einem Package mit dem Namen “classification package”. Auf dieses Package soll von einer “main.java” Datei zugegriffen werden. Das Package enthält eine Datei mit dem Namen "Classification Of Float Values", in dem die User relevanten Funktionen stehen. Nur über die Funktionen in dieser Datei kann der User Einfluss auf den Algorithmus nehmen.

Die einzelnen Funktionen und Variablen, die vom User verändert werden können bzw. angegeben werden müssen, können im Beispielprogramm oder auf der GitHub Page zu diesem Projekt (siehe Quellen → Source Code) eingesehen werden.

Das Programm verfügt des Weiteren über eine Fehlererkennung. Das Programm kann erkennen, ob nötige Befehle, die von einem User eingegeben werden müssen, bereits eingegeben wurden. Wenn dies nicht der Fall ist, pausiert das Programm und bricht ab. Dadurch kann eine lange Fehlersuche vermieden werden, da die User-Errors eingegrenzt werden können. Da, wenn ein notwendiger Befehl fehlt, das Programm zu einem bestimmten Zeitpunkt die Berechnungen nicht korrekt ausführen kann, führt die Fehlererkennung dazu, dass keine “falschen” Fehler des Programms aufgeworfen werden.

Bohnen Erweiterung

In diesem Beispiel/Erweiterung der Anwendung des Programms, wird ein anderer Datensatz verwendet. Der in diesem Beispiel verwendete Datensatz (siehe *Bohnen-Datensatz*) enthält verschiedene Parameter zu verschiedenen Bohnenarten (Klassen). Der Datensatz hat 13000+ Dateneinträge, 7 verschiedene Klassen und 16 Parameter/*Features*:

1. Fläche der Bohne
2. Umfang der Bohne
3. längste Distanz von einem Bohnen Ende zum anderen
4. längste Distanz (senkrecht zur vorherigen Achse)
5. Verhältnis zwischen den beiden vorherigen Parametern
6. Exzentrizität der Bohne
7. Anzahl der Pixel im kleinsten konvexen Polygon, die die Fläche der Bohne beinhalten kann
8. Durchmesser eines Kreises mit der gleichen Fläche der Bohne
9. Verhältnis der Pixel des Hüllkörpers
10. Verhältnis der Pixel in der konvexen Form zu denen in der Bohne
11. Rundheit der Bohne
12. Kompaktheit der Bohne
13. “Form Faktor 1”
14. “Form Faktor 2”
15. “Form Faktor 3”
16. “Form Faktor 4”

Der Datensatz wurde mit Hilfe eines *Computer Vision System* erstellt, das mit Bildern von Bohnen versorgt wurde.

Anmerkung: Da der Datensatz wesentlich größer ist (mehr 90x größer), benötigt das Programm auch etwas länger. Des Weiteren wurde der Datensatz auch manipuliert, da im ursprünglichen Datensatz die Einträge nach Klasse sortiert wurden, und deshalb Probleme entstehen können (siehe *Schlechte Unterteilung zwischen Trainings- und Testdaten*). Deshalb wurde die Reihenfolge der Einträge zufällig verändert. Dies konnte durch eine Evaluation einer vorherigen Klassifikation festgestellt werden.

Feature Engineering

Die bereits genannten *Features* werden alle mit gleicher Gewichtung beachtet. Das Programm ist so geschrieben, dass die Anzahl an *Features* egal ist, bzw. dass es mindestens ein *Feature* geben muss.

Lern- und Testphase

Die Lern- bzw. Testphase verläuft exakt gleich, wie bei dem zuvor beschriebenen Beispiel mit dem kleineren Datensatz, nur mit mehr *Features* bzw. n = 16 in *Abb 4.1: Darstellung einer Distanz Klassifikation* bei sieben Klassen. Das Programm ist in einer Art und Weise programmiert, so dass es automatisch erkennen kann, wie viele *Features* ein Datensatz besitzt, und sich dementsprechend anpasst.

Interpretation der Ergebnisse

Werden dann die verschiedenen *Confusion Matrices* betrachtet (siehe *Abb. 5: Evaluation von Bohnen* bei *Normalized Confusion Matrix*) so lässt sich erkennen, dass die Testdaten bei manchen Klassen gut richtig zugewiesen, aber einige Klassen auch sehr schlecht erkannt werden. Im Durchschnitt werden ca. 55% der Daten aus dem Testdatensatz richtig erkannt, was ein relativ schlechtes Ergebnis ist, verglichen mit dem vorherigen Beispiel, bei dem alle Testdaten richtig klassifiziert wurden.

Nutzung der Interpretation

Das schlechte Ergebnis könnte sich natürlich auf eine *Schlechte Unterteilung zwischen Trainings- und Testdaten* zurückführen lassen, wofür auch die stark variierende Anzahl an Testdaten je Klasse ein Indiz ist. Vergleicht man jedoch den Bohnen-Datensatz mit dem Blumen-Datensatz, so kann man neben der Größe einen anderen signifikanten Unterschied erkennen. Bei dem Blumen-Datensatz sind alle *Features* in cm angegeben und haben somit einen linearen Zusammenhang zueinander. Bei dem Bohnen-Datensatz variieren die Einheiten oder Größen der *Features*. Es gibt Längen, Flächen, Verhältnisse und andere Parameter/Faktoren. Dabei sind die *Features* untereinander größtenteils nicht linear zueinander. Diese Problematik wird in *Lineare vs. Nicht-Lineare Datenpunkte* detaillierter ausgeführt. Um dieses Problem zu beheben, könnte man die Gewichtungen der einzelnen *Features* verändern, jedoch wäre dies schwierig bei Faktoren, die in keinerlei Zusammenhang mit beispielsweise einer Länge stehen.

Probleme bei der Klassifikation

Die hier genannten Probleme gehen nur auf Schwierigkeiten des beschriebenen Algorithmus ein. Allgemeine Schwierigkeiten von Machine Learning Modellen bzw. Algorithmen, die aus einem verhältnismäßig kleinen Datensatz eine allgemeine Bedingung für eine Vorhersage extrahieren, werden hier nicht genannt, sind jedoch auch in diesem Fall vorhanden und sollten deshalb auch bei der Evaluation bzw. Interpretation der Ergebnisse beachtet werden.

Solche allgemeinen Probleme sind beispielsweise: *Statistical Bias*; *Coverage Bias*; *Sampling Bias*; *Reporting Bias*; *Experimenter's Bias*; …

Schlechte Unterteilung zwischen Trainings- und Testdaten

Wie bereits beschrieben, wird beim Machine Learning ein Datensatz in einen Trainings- und einen Testdatensatz unterteilt. Der Trainingsdatensatz trainiert das Machine Learning Modell bzw. sucht Muster im Datensatz, während der Testdatensatz das trainierte Modell überprüft, um gegebenenfalls Veränderungen am Modell vorzunehmen. Dazu muss der ursprüngliche Datensatz an einer bestimmten Stelle unterteilt werden (im beschriebenen Programm liegt der Anteil der Trainingsdaten bei 70%). Dies kann ein Problem werden, wenn zum Beispiel die Daten für eine einzelne Klasse nur am Ende des Datensatzes aufgeführt werden, und somit nicht im Trainingsdatensatz liegen. Zum einen würde der Algorithmus diese Klasse für zukünftige Vorhersagen nicht kennen, da er ja nicht damit trainiert worden ist. Daraus folgt auch, dass diese Klasse in der Testphase einer anderen Klasse zugeordnet werden würde, was einem schlechten Ergebnis entsprechen würde.

Ein ähnliches Problem wäre, wenn beispielsweise eine Klasse aus den Trainingsdaten nicht im Testdatensatz vorhanden ist und somit nicht getestet werden würde.

Natürlich könnte man dieses Problem umgehen, indem man einfach den Testdatensatz verkleinert oder ganz weg lässt. Ein Nebeneffekt wäre, dass durch einen größeren Trainingsdatensatz die Klassifikation genauer wird, da es mehr Datenpunkte für den Algorithmus gibt. Jedoch wird es dadurch schwieriger, bis unmöglich das Modell zu evaluieren. Dadurch wird es auch schwieriger das Modell anzupassen, und somit auch zu verbessern, da man nicht weiß ob das Modell durch die Anpassung besser oder schlechter wird.

Dieses Problem wurde im Programm durch eine Manipulation des Datensatzes umgangen. Dazu wurden die Einträge im Datensatz zufällig sortiert, sodass die verschiedenen Klassen ausreichend für eine korrekte Klassifikation verteilt sind. Jedoch wird das Problem dadurch nicht behoben, sondern nimmt nur an Signifikanz ab, da die Wahrscheinlichkeit abnimmt, dass so ein Fall eintritt. Ein weiterer Nachteil ist, dass es einen weiteren manuellen Schritt durch den User voraussetzt.

Eine Möglichkeit, dieses Problem vollständig zu beheben, wäre, vor der Unterteilung des Datensatzes die Anzahl an unterschiedlichen Klassen zu zählen, und den Klassen gleichzeitig einen Standardwert zuzuweisen. Natürlich kann es sein, dass es sich bei dem Standardwert um einen Extremwert der Klasse handelt, und die daraus folgende Klassifikation somit schlechter wäre, jedoch wäre dies besser als der derzeitige Ansatz. Dieser Ansatz ist jedoch nicht implementiert und stellt daher einen Verbesserungsvorschlag dar.

Das Ziel des Algorithmus sollte es sein, Muster auf Grundlage des Datensatzes möglichst "gut" und automatisiert zu bestimmen.

Lineare vs. Nicht-Lineare Datenpunkte

Wie bereits beschrieben in *Nutzung der Interpretation* in *Bohnen Erweiterung*, muss bei der Klassifikation von numerischen Werten beachtet werden, ob es sich bei den einzelnen *Features* im Datensatz um lineare oder nicht-lineare Parameter handelt. Beispielsweise ist der Zusammenhang zwischen zwei Angaben in m (Meter) linear zueinander, da es sich um die gleiche Einheit und Skalierung handelt (vergleiche Blumen-Datensatz). Ist eine Längenangabe in cm angegeben, müsste diese mit dem Faktor 10^{-2} multipliziert werden, damit es sich um die gleiche Einheit handelt. Handelt es sich beispielsweise um eine Längenangabe in m und eine Fläche in m^2, so handelt es sich nicht mehr um einen linearen Zusammenhang, sondern um einen Quadratischen. Das gleiche gilt auch für Verhältnisse, Parameter und Faktoren, die in einer anderen Einheit stehen. Jedoch ist es in solchen Fällen schwerer, den konkreten Zusammenhang zu finden, wie zum Beispiel bei einer Länge und einer Fläche. Doch inwiefern ist das ein Problem für den Algorithmus?

Wie bereits beschrieben, handelt es sich bei dem Algorithmus um einen linearen Distanz-Klassifikationsalgorithmus. Das heißt, es wird von einem Datenpunkt der euklidische Abstand zu den jeweiligen Klassen berechnet. Dafür wird der eindimensionale Abstand der einzelnen *Features* zwischen dem Punkt und der Klasse betrachtet, und diese werden dann mit gleicher Gewichtung miteinander verrechnet (siehe *Abb. 4.1: Darstellung einer Distanz Klassifikation*). Jedoch kann das nur korrekt funktionieren, wenn es sich dabei um gleichwertige *Features* handelt bzw. der Zusammenhang zwischen den *Features* linear ist. Ist der Zusammenhang zwischen zwei *Features* nicht linear, so wird ein *Feature* übergewichtet, was zu einer Verzerrung der Ergebnisse führt.

Quellen

Source Code

- https://github.com/max-acc/java-float-classification

Dokumente und Bücher

- "Hands-On Machine Learning mit dem Raspberry Pi" von Stephan Laage-Witt
  Eine grobe Einführung in die Funktionsweise von Machine Learning Algorithmen, aus einem Kurs im Phaenovum Lörrach.
- "Introduction to Machine Learning" von Nils J. Nilsson
  Eine vertiefende Einführung in Machine Learning und die Funktionsweise. Das Dokument kann auf http://robotics.stanford.edu/~nilsson/mlbook.html eingesehen werden.

Anm.: Die Dokumente können auf Nachfrage eingesehen werden, unterliegen jedoch einem Urheberrecht und sind unter Umständen nicht öffentlich einsehbar.

Links

- https://www.ibm.com/topics/machine-learning
- https://www.oracle.com/de/artificial-intelligence/machine-learning/what-is-machine-learning/
- https://developers.google.com/machine-learning/glossary
- https://de.wikipedia.org/wiki/Beurteilung_eines_binären_Klassifikators
- https://seos-project.eu/classification/classification-c04-p01.de.html
- https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b0485fba23aabda526358f31cb5a382b66a08270
- https://www.datacamp.com/blog/classification-machine-learning
- https://de.wikipedia.org/wiki/Softwaredesign

Datensätze

- Iris-Datensatz
  Fisher,R. A.. (1988). Iris. UCI Machine Learning Repository.
  https://doi.org/10.24432/C56C76

- Bohnen-Datensatz
  Dry Bean Dataset. (2020). UCI Machine Learning Repository.
  https://doi.org/10.24432/C50S4B

Machine Learning

Klassifikation von numerischen Werten mit Java

Fachwortverzeichnis

Iris	Wissenschaftlicher Name für die Kategorie der Schwertlilien.
Machine Learning	Ein Programm, welches ein Modell mit einem Datensatz trainiert, um sinnvolle Vorhersagen für unbekannte/neue Daten treffen zu können.
Feature	Eine Eingangsvariable für ein Machine Learning Modell, zum Beispiel je die Blütenlänge oder /-weite.
Label/ Zielvariable	Eine Zielvariable, die im Datensatz den jeweiligen Features eine Klasse zuweist.
Supervised Learning	Trainieren eines Machine Learning Modells mit den Features und ihren zugehörigen Labels.
Predictor/ Prädiktor	Einzelne Eingangsvariablen pro Element in einem Datensatz. Die Prädiktoren für ein Element bestehen meist aus mehreren Features.
Computer Vision System	System zur Datenverarbeitung von visueller Information mit Hilfe von Machine Learning.

Abbildungen

Alle Abbildungen und Grafiken wurden zur grafischen Veranschaulichung und als visuelle Unterstützung selbst erstellt.

Abb. 1: Dateistruktur zur Verwendung des Algorithmus

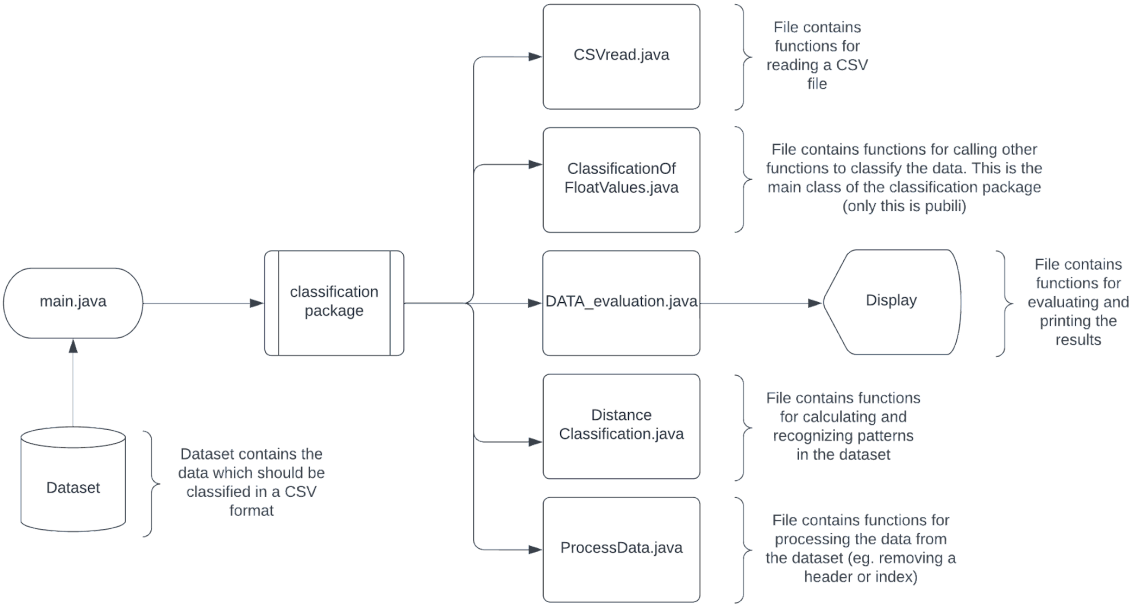


Abb. 2: Funktionsweise eines Machine Learning Programms

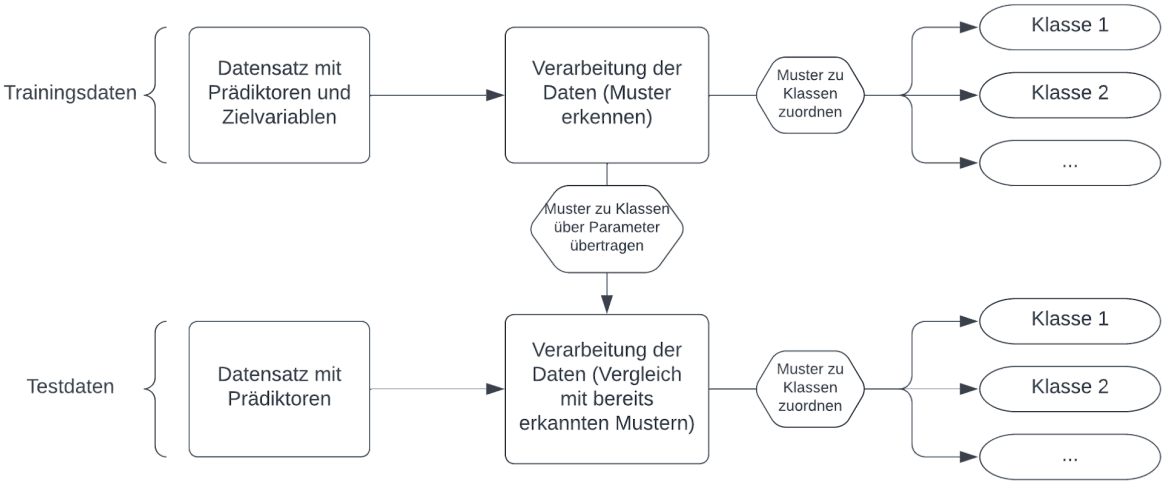


Abb. 3: Variationen von Confusion Matrices

Im Programm wird immer die erste Confusion Matrix berechnet, und die beiden anderen werden dann aus dieser hergeleitet.

W entspricht einer korrekten Klassifikation, während F für eine fehlerhafte Zuordnung steht.

Confusion Matrix:

		Tatsächliche Werte		
		K1	K2	K3
Vorhergesagte Werte	K1	W	F	F
	K2	F	W	F
	K3	F	F	W

Anm.: W bzw. F gibt den absoluten Wert der jeweiligen Ereignisse an.

Simple Confusion Matrix:

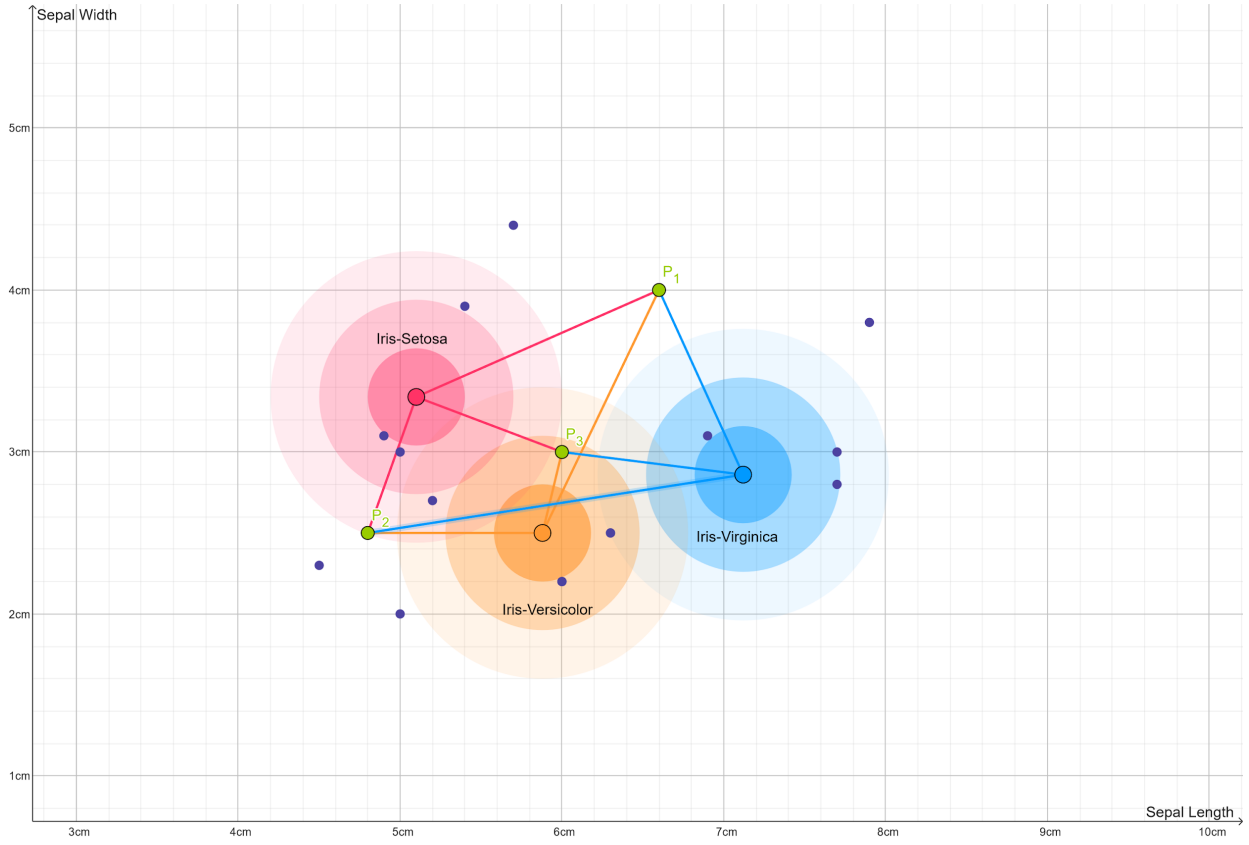
		Tatsächliche Werte	
		Wahr	Falsch
Vorhergesagte Werte	K1	W	F
	K2	W	F
	K3	W	F

Normalized Confusion Matrix:

		Tatsächliche Werte		
		K1	K2	K3
Vorhergesagte Werte	K1	WN	FN	FN
	K2	FN	WN	FN
	K3	FN	FN	WN

Anm.: WN bzw. FN gibt die Wahrscheinlichkeit der jeweiligen Ereignisse an (0-1).

Abb. 4.1: Darstellung einer Distanz Klassifikation



Erklärung

Die eingezeichneten dunkelblauen Punkte stellen zufällig ausgewählte Datenpunkte aus dem "*Iris*" Datensatz dar (jeweils fünf Stück pro Klasse). Die verwendeten *Predictors* sind, in diesem Beispiel, nur die Kelchblatt Länge und Weite. Der rote, orange und blaue Punkt stellt jeweils eine ideale bzw. durchschnittliche Blume dar. Dazu wurde jeweils der Durchschnitt der Länge und Weite für jede Klasse berechnet.

Die Punkte P_1, P_2 und P_3 stellen Testwerte dar, die klassifiziert werden sollen. Dazu wird jeweils der Abstand zwischen den Punkten und den Durchschnittswerten berechnet. Dies kann mit dem Satz des Pythagoras berechnet werden:

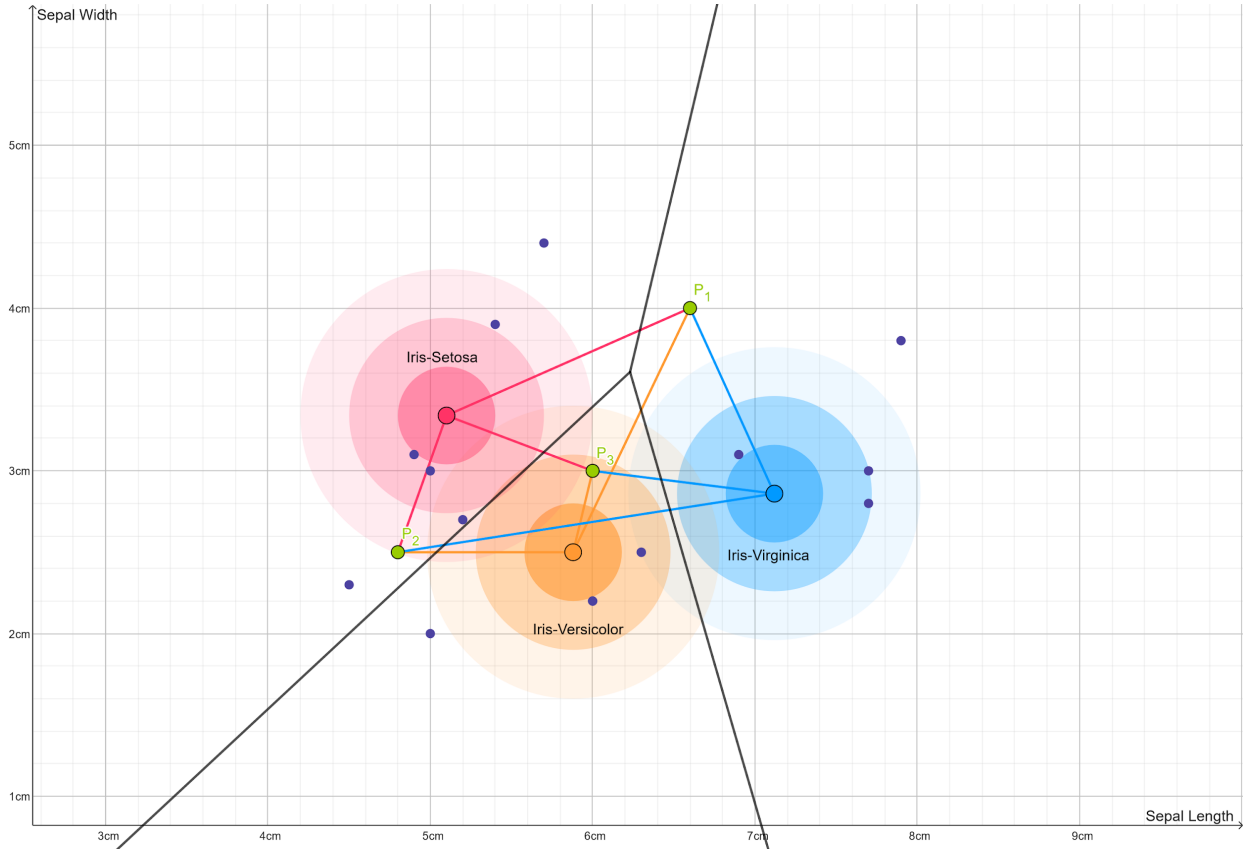
$$\text{Abstand} = \sqrt{\left(P_{1_{\text{Länge}}} - \text{Avg}_{1_{\text{Länge}}}\right)^2 + \left(P_{1_{\text{Weite}}} - \text{Avg}_{1_{\text{Weite}}}\right)^2}$$

Wobei jeweils der Index bei *Avg* für eine der idealen bzw. durchschnittlichen Blumen steht. Die beschriebene Gleichung berechnet den Abstand für zwei *Predictors*. Für eine beliebige Anzahl *n* an *Predictors* kann folgende Gleichung verwendet werden:

$$\text{Abstand} = \sqrt{\left(P_{1_1} - \text{Avg}_{1_2}\right)^2 + \left(P_{1_2} - \text{Avg}_{1_2}\right)^2 + \ldots + \left(P_{1_n} - \text{Avg}_{1_n}\right)^2}$$

Der zweite Index steht jeweils für den *Predictor*, der in der Rechnung verwendet wird.

Abb. 4.2: Darstellung einer Distanz Klassifikation



Erklärung

Die eingezeichneten Geraden stellen die Mittelsenkrechten zwischen den einzelnen Punkten der idealen bzw. durchschnittlichen Blumen dar. Je nachdem, in welchem Drittel sich ein Testpunkt befindet, kann er einer Klasse zugeordnet werden.

Befindet sich ein Testpunkt direkt auf einer der Geraden, kann der Punkt nicht eindeutig klassifiziert werden.

Abb. 5: Evaluation von Bohnen

Confusion Matrix:

		Tatsächliche Werte						
		K1	K2	K3	K4	K5	K6	K7
Vorhergesagte Werte	K1	824	2	0	0	189	287	1
	K2	68	53	6	0	39	0	78
	K3	0	0	6	0	0	0	85
	K4	0	0	0	21	0	0	0
	K5	302	15	3	0	175	0	11
	K6	515	2	0	0	18	1207	0
	K7	0	7	13	0	0	0	127

Simple Confusion Matrix:

		Tatsächliche Werte	
		Wahr	Falsch
Vorhergesagte Werte	K1	845	479
	K2	53	191
	K3	6	85
	K4	21	0
	K5	175	331
	K6	1207	535
	K7	127	20

Normalized Confusion Matrix (manuell gerundet):

		Tatsächliche Werte						
		K1	K2	K3	K4	K5	K6	K7
Vorhergesagte Werte	K1	0.64	0.0	0	0	0.14	0.22	0.0
	K2	0.28	0.22	0.02	0	0.16	0	0.32
	K3	0	0	0.07	0	0	0	0.93
	K4	0	0	0	1.0	0	0	0
	K5	0.60	0.03	0.0	0	0.35	0	0.02
	K6	0.30	0.0	0	0	0.1	0.69	0
	K7	0	0.05	0.09	0	0	0	0.86

Anm.: 0.0 entspricht einer Rundung einer Zahl kleiner als 10^{-2}.