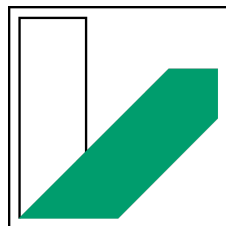




Lehrstuhl Angewandte Informatik IV
Datenbanken und Informationssysteme
Prof. Dr.-Ing. Stefan Jablonski

Institut für Angewandte Informatik
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth



**UNIVERSITÄT
BAYREUTH**

Entwicklung eines webbasierten Rahmenwerkes zur Evaluation von Klassifikatoren

Christian Gebhardt

September 24, 2021
Version: Draft / Final

Universität Bayreuth

Fakultät Mathematik, Physik, Informatik

Institut für Informatik

Lehrstuhl für Angewandte Informatik IV

Bachelorarbeit

Entwicklung eines webbasierten Rahmenwerkes zur Evaluation von Klassifikatoren

Christian Gebhardt

- | | |
|--------------------|-----------------------------------------------------------------------------------------------------------|
| <i>1. Reviewer</i> | Prof. Dr.-Ing. Stefan Jablonski
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth |
| <i>2. Reviewer</i> | Dr. Lars Ackermann
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth |
| <i>Supervisor</i> | Msc. Martin Käppel |

September 24, 2021

Christian Gebhardt

Entwicklung eines webbasierten Rahmenwerkes zur Evaluation von Klassifikatoren

Bachelorarbeit, September 24, 2021

Reviewers: Prof. Dr.-Ing. Stefan Jablonski and Dr. Lars Ackermann

Supervisor: Msc. Martin Käppel

Universität Bayreuth

Lehrstuhl für Angewandte Informatik IV

Institut für Informatik

Fakultät Mathematik, Physik, Informatik

Universitätsstrasse 30

95447 Bayreuth

Germany

Zusammenfassung

Die folgende Bachelorarbeit beschäftigt sich mit der Entwicklung eines webbasierten Rahmenwerkes zur Evaluation von Klassifikatoren. Klassifikationsalgorithmen sind ein Teilbereich des aufstrebenden Forschungsbereich des Machine Learnings, welcher in den letzten Jahren für einige Innovationen gesorgt hat. Einige bekannte Anwendungsbereiche sind unter anderem Spracherkennungssysteme, autonomes Fahren oder auch medizinische Diagnosen. Um Klassifikatoren zu vergleichen oder zu entscheiden, ob ein Klassifikator gewissen Anforderungen genügt, müssen diese anhand geeigneter Metriken evaluiert werden. Im Rahmen dieser Arbeit wird dem Leser ein Einstieg in den Bereich Machine Learning, insbesondere Klassifikation gegeben und es wird sich mit den theoretischen Grundlagen für eine spätere Evaluation dieser Klassifikatoren auseinandergesetzt. Schließlich wird eine praktische Lösung als Webanwendung implementiert.

Inhaltsverzeichnis

1	Einführung	1
2	Grundlagen des maschinellen Lernens	3
2.1	Definition und Begriffe	3
2.1.1	Vorteile von Machine Learning	4
2.1.2	Verschiedene Arten des Machine Learnings	5
2.2	Klassifikation	5
2.2.1	Binäre Klassifikation	6
2.2.2	Multinomiale Klassifikation	6
2.3	Die Machine Learning Pipeline	8
2.4	Datenaufbereitung	9
2.4.1	Fehlertypen	9
2.4.2	Strategien der Datenaufbereitung	10
2.4.3	Kodierung von kategorischen Werten	11
2.4.4	Aufteilung in Trainings- und Testdaten	12
3	Weiterführende Arbeiten	15
4	Eine Auswahl von Klassifikatoren	17
4.1	Naive Bayes	17
4.1.1	Vorgehensweise	18
4.1.2	Erweiterung auf kontinuierliche Attribute	20
4.2	Decision Tree	22
4.2.1	Aufbau	22
4.2.2	Pruning	26
4.2.3	Random Forest	27
4.3	Logistische Regression	28
4.3.1	Hypothese	29
4.3.2	Gradient Descent Verfahren	30
4.3.3	Neuronales Netz	34
5	Evaluation von Klassifikatoren	37

5.1	Metriken für Klassifikationsalgorithmen	37
5.1.1	Konfusionsmatrix	37
5.1.2	Definition der Metriken	38
5.1.3	Receiver Operating Characteristics (ROC)	42
5.2	Einfluss von Eigenschaften der Datensätze	43
5.3	Auswahl geeigneter Metriken	45
6	Implementierung des Evaluationsrahmenwerkes	47
6.1	Anforderungen	47
6.2	Architektur und Kommunikation der Komponenten	48
6.3	Funktionsweise	49
6.3.1	User Interface	49
6.3.2	Evaluation	53
6.4	Hilfsskripte und Anwendungsbeispiele	55
6.4.1	Evaluation eines eigenen Klassifikators	55
6.4.2	Datensatz Format	57
6.4.3	Generierung von Trainings- und Testindizes	58
7	Fazit	61
	Literaturverzeichnis	63

Einführung

Der Forschungsbereich Machine Learning hat in den vergangenen Jahren in verschiedensten Bereichen für Innovation gesorgt und wird auch in Zukunft noch mehr an Bedeutung gewinnen. Allein für den weltweiten Markt für künstliche Intelligenz, wozu der Bereich Machine Learning zählt, wird bis 2025 ein Umsatz von 126 Milliarden US-Dollar vorhergesagt.¹

Einige Beispiele für Machine Learning sind Spracherkennungssysteme, autonomes Fahren oder auch medizinische Diagnosen. In vielen dieser Bereiche finden sich Klassifikationsprobleme, welche in dieser Bachelorarbeit genauer betrachtet werden sollen und die Grundlage für das entwickelte Evaluationsrahmenwerk für Klassifikatoren bilden. Für medizinische Diagnosen könnte das Problem vereinfacht wie folgt beschrieben werden: Es soll anhand von verschiedenen Attributen entschieden werden, ob eine Person an einer bestimmten Krankheit leidet oder nicht. Um dieses Problem zu lösen, soll ein Klassifikator von bereits vorhanden Daten und Diagnosen lernen, um zukünftig selbst Diagnosen stellen zu können. Diese Klasse von Problemen beschränkt sich nicht nur auf medizinische Diagnosen, sondern kann in einem großen Spektrum von Bereichen gefunden werden, in denen anhand von verfügbarem Wissen ein Objekt einer bestimmten Klasse zugeordnet werden soll. So bieten sich allein für Klassifikationsprobleme eine Vielzahl an Anwendungsmöglichkeiten.

Um zu erkennen, ob ein Klassifikator eine ausreichende Leistung für eine dieser Anwendungsmöglichkeiten erfüllt und somit praktisch einsetzbar ist, gilt es seine Leistung zu messen und zu evaluieren. In dieser Bachelorarbeit soll dem Leser zunächst ein Einstieg in den Bereich Machine Learning, insbesondere Klassifikation gegeben werden, bevor theoretisch erklärt wird, wie die Leistung eines Klassifikators evaluiert werden kann. Schließlich wird eine Implementation eines Webframeworkes zur Evaluation von Klassifikatoren geliefert, welche die beschriebenen theoretischen Konzepte in einer praktischen Anwendung umsetzt.

¹<https://www.statista.com/statistics/607716/worldwide-artificial-intelligence-market-revenues>

Grundlagen des maschinellen Lernens

2.1 Definition und Begriffe

Der Bereich Machine Learning ist ein stark wachsender Bereich der Informatik, welcher versucht komplexe Probleme durch automatisiertes Lernen mit einem Algorithmus zu lösen. Der Algorithmus lernt aus verschiedenen Datenpunkten und generiert daraus Wissen und Erkenntnisse, was möglicherweise nicht durch fest vorgegebene Regeln eines Menschen möglich wäre. Eine präzise formale Definition wäre die folgende ([Gé17], S.4):

“Man sagt, dass ein Computerprogramm dann aus Erfahrungen E im Bezug auf eine Aufgabe T und ein Maß für die Leistung P lernt, wenn seine durch P gemessene Leistung bei T mit der Erfahrung E anwächst.”

Tom Mitchell 1997

Versuchen wir nun diese Definition anhand eines Beispiels zu verdeutlichen: Angenommen wir wollen anhand verschiedener gesundheitlicher Daten prognostizieren, ob eine Person an Diabetes erkrankt ist oder nicht. Der entsprechende Ausschnitt der Tabelle zu dem Beispiel ist in Abbildung 2.1 zu sehen.

In diesem Fall wäre die *Erfahrung* E eine Tabelle mit Datenpunkten, auch *Instanzen* genannt, die eben diese gesundheitlichen Parameter wie Blutzuckerspiegel, BMI, Insulinwerte, Alter, etc. enthält¹. Diese Parameter werden auch *Attribute* oder *Features* genannt (vgl. [Bra07], S.4). Die Attribute können als numerische oder kategorische Werte auftreten. Numerische Werte sind entweder kontinuierliche Werte, wie reelle Zahlen oder diskrete Werte, z.B. ganze Zahlen. Die kategorischen Werte hingegen können entweder in eine natürliche Reihenfolge gebracht werden, wie z.B. eine Notenskala von „Sehr Gut“ bis „Ungenügend“, dann werden diese ordinal genannt, andernfalls nominal. Beispiele für nominale Werte sind z.B. Namen oder Farben.

Ebenfalls enthält die Tabelle auch die Ergebnisse für jede Instanz. Diese werden *Zielattribute* oder auch *Labels* genannt. In unserem Beispiel enthalten diese die

¹Die Tabelle wurde um wenige unbrauchbare Datenpunkte bereinigt.

Information, ob die Person tatsächlich an Diabetes erkrankt ist oder nicht. Mit Hilfe dieser Daten wird der Algorithmus später lernen Vorhersagen auf neuen unbekannten Daten zu treffen. Die Aufgabe T wäre nun ein möglichst gutes Modell mit einer Hypothese $f(x_1, x_2, \dots, x_n) = y$ zu finden, welche die Werte der Attribute x_1, x_2, \dots, x_n einem $y \in \{0, 1\}$ zuordnet, wobei der Wert 0 ausdrückt, dass die Person nicht an Diabetes erkrankt ist, und analog der Wert 1 ausdrückt, dass sie an Diabetes erkrankt ist. Die Bedeutungen der einzelnen Attribute sowie deren Zusammenhänge mit dem Ausgang der späteren Diagnose sollen in dieser Funktion berücksichtigt werden.

Schließlich soll die Leistung P anhand verschiedener Metriken evaluiert werden, z.B. in Form der Genauigkeit der Vorhersage, also wie viele der Vorhersagen des Klassifikators korrekt waren.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Age	Outcome
3	1	89	66	23	94	28.1	21	0
4	0	137	40	35	168	43.1	33	1
6	3	78	50	32	88	31.0	26	1
8	2	197	70	45	543	30.5	53	1
13	1	189	60	23	846	30.1	59	1
...
753	0	181	88	44	510	43.3	26	1
755	1	128	88	39	110	36.5	37	1
760	2	88	58	26	16	28.4	22	0
763	10	101	76	48	180	32.9	63	0
765	5	121	72	23	112	26.2	30	0

Abbildung 2.1: Tabelle zur Analyse einer Diabetes-Erkrankung. Quelle: <https://github.com/plotly/datasets/blob/master/diabetes.csv>, zuletzt aufgerufen am 19.09.2021

2.1.1 Vorteile von Machine Learning

Eine naheliegende und einfache Methode wäre, Datenpunkte von Personen, die an Diabetes erkrankt sind, zu durchsuchen und Ähnlichkeiten zu finden. In den obigen Datenpunkten lassen sich einige auffällige Korrelationen erkennen. Zum Beispiel, dass neben einem hohen Blutzuckerspiegel erhöhter Blutdruck gepaart mit einem hohen BMI häufig ein starker Indikator für Diabetes ist. Mittels statistischer Modelle

lassen sich so Richtwerte für die einzelnen Parameter herleiten, mit denen konkrete Entscheidungsregeln aufgestellt werden können. Unter Verwendung dieser kann wiederum mit einer gewissen Wahrscheinlichkeit bestimmt werden, ob ein Mensch an Diabetes erkrankt ist oder nicht.

Diese Vorgehensweise wäre allerdings äußerst mühsam, weil sie kaum automatisiert durchführbar ist. Sie würde ein sehr genaues Verständnis der Datensätze voraussetzen und zudem besteht die Gefahr eines sogenannten *Bias*, was bedeutet, dass eine Person durch eine voreingenommene Meinung im Hinblick auf den Datensatz manche, eventuell weniger auffällige, Korrelation nicht findet und nur bekannte, bzw. offensichtliche Korrelationen betrachtet (vgl. [RRC19], S.2). Viele Machine Learning Algorithmen besitzen diesen Bias nicht, da sie ohne Vorwissen bzw. mit wenig Vorwissen arbeiten. Somit haben diese häufig, im Gegensatz zu einem Menschen, die Möglichkeit, neue Erkenntnisse aus den Daten zu gewinnen.

2.1.2 Verschiedene Arten des Machine Learnings

Im Folgenden wird eine kurze Unterscheidung zwischen den verschiedenen Arten des Machine Learnings getroffen. Wenn wir dem Algorithmus wie im oben genannten Beispiel Daten mit bekannten Ergebnissen liefern, sogenannte *Labeled Datasets*, sprechen wir bei dieser Art des maschinellen Lernens vom *Supervised Learning*, zu Deutsch: überwachtes Lernen.

Dem entgegen steht das *Unsupervised Learning* (unüberwachtes Lernen), bei welchem die Labels nicht vorhanden sind, bei den sogenannten *Unlabelled Datasets* (vgl. [Bra07], S.4-5). Der Algorithmus versucht hier lediglich so viel Informationsgehalt wie möglich aus den Daten zu gewinnen und mögliche Muster zu erkennen.

Innerhalb des Supervised Learnings muss wiederum zwischen der Problemstellung unterschieden werden. So gibt es Probleme, bei denen eine kontinuierliche numerische Größe die Ausgabe des Algorithmus sein soll. Ein Beispiel wäre hier die Vorhersage der Preise von Häusern, bestimmt aus möglichen Attributen wie Lage, Anzahl der Zimmer, Zustand, etc. Dies wird allgemein als *Regression* bezeichnet. Das oben genannte Beispiel, die Prognose von Diabetes, hingegen *klassifiziert* die Ergebnisse innerhalb einer endlichen Anzahl von Kategorien („Diabetiker“ oder „Kein Diabetiker“), hier sprechen wir folglich von *Klassifikation*.

2.2 Klassifikation

Der eigentliche Fokus dieser Arbeit soll auf der Evaluation von Klassifikatoren liegen. Deshalb ergibt es Sinn, diesen Teilbereich des maschinellen Lernens noch etwas

genauer zu betrachten, um dem Leser zunächst ein grundlegendes theoretisches Verständnis von Klassifikation und deren möglichen Problemklassen zu vermitteln.

Ein Klassifikator ist wie folgt definiert: Ein Klassifikator ist ein Algorithmus, der vorher nicht betrachtete Instanzen von Objekten klassifiziert (vgl. [Bra07], S.79). Er wendet also seine Erfahrung der Trainingsdaten auf neue, bisher ungesehene Instanzen an. Wir betrachten zwei Arten von Klassifikation: die *binäre Klassifikation* und die *multinomiale Klassifikation*. Ferner sind noch die hierarchische und die multi-labeled Klassifikation zu nennen, welche in dieser Arbeit allerdings nicht behandelt werden.

2.2.1 Binäre Klassifikation

Im vorigen Abschnitt wurde bereits das Problem der Prognose von Diabeteserkrankungen beschrieben. Hierbei handelt es sich um eine sogenannte *binäre Klassifikation*, da die Hypothese

$$f(x_1, x_2, \dots, x_n) = y \quad (2.1)$$

mit n Attributen x_1, x_2, \dots, x_n als Parameter lediglich auf einen Wert im Bereich $y \in \{0, 1\}$ abbildet. Es gibt also nur *zwei Kategorien*. Zudem sollte auch beachtet werden, dass jeder Datenpunkt bzw. jedes Objekt nur *genau einer* Kategorie zugeordnet wird, niemals aber kann ein Objekt mehr als einer Kategorie zugeordnet werden.

Diese Art von Klassifikation ist weit verbreitet und findet häufig bei reinen „Ja/Nein-Fragen“ ihre Anwendung. Ein weiteres Beispiel wäre die Einordnung von Spam E-Mails, genau wie zuvor gibt es auch hier nur zwei Kategorien: Spam oder kein Spam. Dabei muss beachtet werden, dass durch die Hypothese lediglich abstrakt das Klassifikationsproblem beschrieben wird. Es handelt sich hier im Allgemeinen um keine mathematische Funktion, sondern einen angelernten Algorithmus, der für gegebene Attribute eine Vorhersage zurückgibt.

2.2.2 Multinomiale Klassifikation

Eine allgemeinere Form der Klassifikation ist die *multinomiale Klassifikation* oder auch Mehrklassen-Klassifikation. Im Gegensatz zur binären Klassifikation, ordnet die Hypothese

$$f(x_1, x_2, \dots, x_n) = y \quad (2.2)$$

die n Attribute x_1, x_2, \dots, x_n genau einem Wert im Bereich $y \in \{0, 1, \dots, k-1\}$, $k > 2$ zu, wobei k der Anzahl der Klassen entspricht. Das Objekt kann also *genau einer* von k *Kategorien* zugeordnet werden, wobei *mindestens drei oder mehr* Kategorien

existieren müssen.² Jede Kategorie wird genau wie zuvor auf eine Zahl abgebildet, jetzt allerdings von 0 bis $k - 1$.

Betrachten wir nun als Beispiel für eine multinomiale Klassifikation die Tabelle in Abbildung 2.2. Diese enthält Datenpunkte mit Eigenschaften von verschiedenen Spezies der Schwertlilien. Gezeigt werden die ersten drei Reihen für jede der drei Spezies „Iris-Setosa“, „Iris-Versicolor“ und „Iris-Virginica“. Jeder Datenpunkt besitzt die Spalten Sepallänge, Sepalbreite, Petallänge, Petalbreite und abschließend den Namen der Spezies.³ Um die Labels, also die Namen der Spezies, nun auf einen numerischen Wert abzubilden, könnte jede der drei Spezies eine Zahl von 0 bis 2 zugewiesen bekommen. Die Unterscheidung zwischen binärer und multinomialer Klassifikation ist in den späteren Kapiteln für das Verständnis verschiedener Klassifikationsalgorithmen und insbesondere auch für deren Evaluation von Bedeutung.

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
50	7.0	3.2	4.7	1.4	Iris-versicolor
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
100	6.3	3.3	6.0	2.5	Iris-virginica
101	5.8	2.7	5.1	1.9	Iris-virginica
102	7.1	3.0	5.9	2.1	Iris-virginica

Abbildung 2.2: Tabelle zur Klassifikation von Schwertlilien. Quelle: <https://github.com/plotly/datasets/blob/master/iris.csv>, in Anlehnung an <http://archive.ics.uci.edu/ml/datasets/Iris>, zuletzt aufgerufen am 19.09.2021

²Beachte: Hier wird das sogenannte *Zero-based numbering* verwendet

³Die *Sepalen* sind die Kelchblätter der Blume, die *Petalen* die Kronblätter, siehe auch <https://de.wikipedia.org/wiki/Kelchblatt> und <https://de.wikipedia.org/wiki/Kronblatt>

2.3 Die Machine Learning Pipeline

Bevor wir uns tiefer mit der Thematik Klassifikatoren beschäftigen, soll zunächst gezeigt werden, wie der typische Workflow innerhalb einer Machine Learning Pipeline aussieht. Diese Vorgehensweise wird auch als Cross Industry Standard Process for Data Mining, kurz *CRISP-DM* bezeichnet, an welchem sich folgendes Kapitel orientiert (vgl. [OD08], S.9-18).

Es wird im folgenden nicht weiter auf die Schritte *Business Understanding* (im allgemeinen Problemverständnis) und *Data Understanding* eingegangen, welche vor dem Start der Pipeline geschehen müssen. Es ist natürlich unerlässlich, zuvor eine genaue Aufgabenstellung herauszuarbeiten und zu definieren. Genauso wichtig ist es zu verstehen, welche Daten für die Lösung dieses Problems relevant sein können und in welcher Form diese vorliegen müssen. Jeder Schritt innerhalb dieser Pipeline ist also essentiell, um am Ende ein möglichst gutes Modell im Hinblick auf die geforderte Performance des Klassifikators zu erhalten.

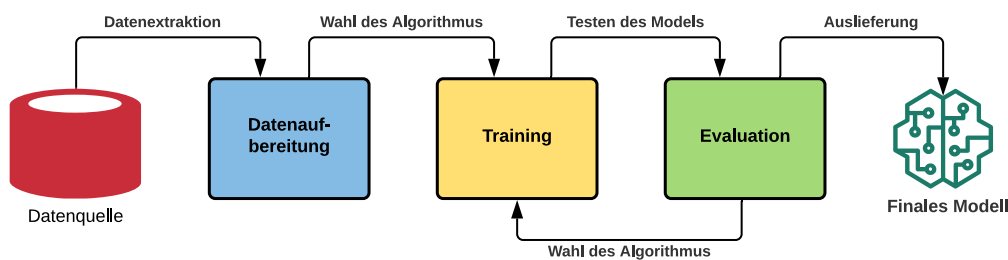


Abbildung 2.3: Der Ablauf der Machine Learning Pipeline

Betrachten wir nun den Ablauf der Machine Learning Pipeline, grafisch dargestellt in Abbildung 2.3. Bevor die eigentliche Pipeline beginnen kann, sollten möglichst viele, hochwertige Trainingsdaten gesammelt werden. Wie in 2.1 beschrieben, bilden diese die Erfahrung des Algorithmus, deshalb sind deren Umfang und Qualität äußerst wichtig für die spätere Performance des trainierten Modells.

Im Idealfall sind die benötigten Daten bereits in einer oder mehreren Datenquellen in Form von einer bekannten Datenbank, einem Online Repository oder Ähnlichem vorhanden. Diese Arbeit beschränkt sich auf *strukturierte Daten*, also Daten, die bereits in einem spezifizierten Format vorliegen, im Allgemeinen in Tabellenform. Dem entgegen stehen *unstrukturierte Daten*, welche keine vorher spezifizierte Form haben und erst in diese gebracht werden müssen (z.B. Textpassagen innerhalb eines Dokumentes oder Nutzungsdaten einer Website). Dieser Prozess ist deutlich aufwändiger und benötigt zusätzliche Methoden der Datenaufbereitung.

Sobald diese extrahiert und in das geeignete Format gebracht wurden, starten die drei Hauptaufgaben der Pipeline: die *Datenaufbereitung*, das *Training* des Modells und die *Evaluation* des Modells. Es ist auch anzumerken, dass die Pipeline stetige Feedback-Zyklen zwischen den Stufen enthält und so ständig basierend auf Ergebnissen der späteren Stufen Anpassungen innerhalb früher Stufen vorgenommen werden. Dies beschränkt sich nicht nur auf die Wahl des Algorithmus, sondern viel mehr auch darauf, wie Daten aufbereitet werden oder das Training des Modells verändert wird.

In Kapitel 4 wird eine Auswahl an verschiedenen Klassifikationsalgorithmen gegeben, welche auch in der späteren Implementierung genutzt worden sind und in Kapitel 5 werden geeignete Metriken und Methoden zur Evaluation von Klassifikatoren beschrieben. Im nächsten Abschnitt soll zuvor der erste Schritt, die Datenaufbereitung, genauer erläutert werden.

2.4 Datenaufbereitung

Die Datenaufbereitung ist der erste Schritt der Pipeline und zugleich auch ein äußerst wichtiger, denn er setzt die Grundlage für den späteren Erfolg des Modells. Das Training auf korrekten, vollständigen Daten ermöglicht es dem Modell, optimal mit Hilfe der Daten zu lernen und somit eine möglichst aussagekräftige Vorhersage zu treffen.

Wird der Algorithmus hingegen auf fehlerhaften Daten trainiert, kann dies aus diversen Gründen maßgeblich die Performance des Modells limitieren. Eine erste Intuition ist, dass ein Algorithmus, der auf realitätsfernen, inkorrekten Daten lernt, nicht in der Lage sein kann, das richtige Ergebnis vorherzusagen, weil er nie gelernt hat, wie dieses in Zusammenhang mit korrekten Attributen steht. Das Ziel sollte es also sein, die Anzahl fehlerhafter Daten zu minimieren.

Ferner ist der Nutzer der später implementierten Anwendung selbst für die Qualität und das Format der verwendeten Daten verantwortlich. Deshalb ist es oft sinnvoll, die verwendeten Datensätze vor der Nutzung mit den hier beschriebenen Methoden aufzubereiten.

2.4.1 Fehlertypen

Fehler können in zwei verschiedenen Formen auftreten: *Noisy Values*, frei übersetzt unbrauchbare Werte, und *Invalid Values*, also ungültige Werte (vgl. [Bra07], S.13). Unbrauchbare Werte entsprechen zwar den vorher definierten Datentypen und Wertebereichen der Spalten, wurden aber durch einen Eingabefehler, bzw. einen

Messfehler verfälscht. Diese treten häufig durch einen Tippfehler bei der Eingabe durch eine Person oder durch inakkurate Messgeräte auf. Diese Werte sind allerdings nicht zu verwechseln mit sogenannten *Outliers* (vgl. [Bra07], S.14).

Die *Outliers* oder Ausreißer, sind durchaus korrekte Werte, wirken allerdings durch ihre enorme Abweichung von den Standardwerten so, als wären sie Fehler. Hier ist es besonders wichtig, sich zunächst ein Bild über die Daten zu beschaffen und nach sorgfältiger Analyse nach eigenem Ermessen zu entscheiden, ob diese verwendet werden sollen oder nicht. Diese Werte können eventuell noch äußerst wichtig für den Lernprozess sein, weil sie seltene, aber dennoch reale Daten abbilden können. Für die Analyse der Daten ist es häufig hilfreich einige statistische Methoden anzuwenden (wie Vergleichen von Minima und Maxima, Bilden von Mittelwerten, etc.) und Streudiagramme oder Box-Plots zu erstellen, um so ein Gefühl für die Verteilung und die Wertebereiche der Daten zu bekommen (vgl. [OD08], S.14).

Ungültige Werte wiederum entsprechen nicht dem vordefinierten Datentyp oder liegen nicht im definierten Wertebereich, dies kann wie zuvor durch einen Eingabefehler passieren oder durch einen Fehler bei der Abspeicherung der Daten, zum Beispiel das Speichern von Zahlen als Strings anstatt als Integer, Floats oder Ähnlichem. Zuletzt kann es auch *fehlende Werte* geben, wenn wie zuvor Fehler unterlaufen sind und der Wert nicht abgespeichert worden ist oder wenn die benötigten Informationen nicht vorhanden gewesen sind.

2.4.2 Strategien der Datenaufbereitung

Eine erste Strategie wäre es nun Datenpunkte, in denen diese beschriebenen Fehler auftreten, zu entfernen. Dieses Vorgehen würde definitiv Fehler minimieren und eignet sich besonders bei einer geringen Anzahl fehlerhafter Datenpunkte. Oft gibt es allerdings einen hohen Prozentsatz fehlerhafte Datenpunkte, bei welchen die sinkende Performance des Modells, hervorgerufen durch eine zu kleine Menge von Trainingsdaten nach der Entfernung der Datensätze, nicht mehr vertretbar ist (vgl. [Kub15], S.203). In diesen Fällen ist diese Vorgehensweise nicht praktikabel.

Um trotzdem viele Datensätze als Trainingsdaten nutzen zu können, bietet es sich an fehlende oder offensichtlich falsche Werte zu *schätzen* und diese durch den Schätzwert zu ersetzen. Eine simple Methode wäre, für kategoriale Typen den am häufigsten vorkommenden Wert zu verwenden, was sich allerdings nur anbietet, wenn dieser Wert ohnehin stark dominant ist und auch tatsächlich davon ausgegangen werden kann, dass es im entsprechenden Kontext sinnvoll ist, diesen zu verwenden. Für numerische, kontinuierliche Typen kann diese Methode selbstverständlich nicht angewendet werden, hier sollte der Mittelwert oder Median, je nach Verteilung, vorgezogen werden.

Bei diesen beiden Vorgehen muss allerdings beachtet werden, dass in vielen Fällen die Werte abhängig von anderen Attributen des jeweiligen Datenpunktes sind und so ein standardisiertes Ersetzen die Qualität verschlechtern würde (vgl. [Kub15], S.203). Erneut ist es hier von besonderer Wichtigkeit, ein Verständnis für die vorliegenden Daten zu haben. Zum Beispiel kann in einer Tabelle mit körperlichen Messwerten von Frauen und Männern nicht die Größe des Mannes durch den gesamten Mittelwert geschätzt werden. Denn so würde er wohl generell zu klein geschätzt werden, weil die Größe der Frauen den Mittelwert senken würden (wenn davon ausgegangen wird, dass durchschnittlich große Frauen und Männer betrachtet werden). Sinnvoller wäre es, nur den Mittelwert der Männer zu verwenden.

Mittlerweile gibt es hierfür einige vordefinierte Methoden, die eine gute Schätzung für fehlerhafte oder fehlende Werte in Abhängigkeit anderer Attribute des Datensatzes mit Hilfe einiger statistischer Verfahren automatisieren können. Ein Beispiel wären hierfür die sogenannten *Imputer* der Python Machine Learning Bibliothek *scikit-learn*⁴.

Da in der heutigen Zeit Speicher meist keinen limitierenden Faktor mehr darstellt, gibt es die Tendenz, mehr Information zu speichern. Tatsächlich haben diese zusätzlichen Attribute oft aber keinen wirklichen Einfluss auf das Ergebnis der Klassifikation. Hier sollte, wann immer möglich, eine sogenannte *Dimensionsreduktion* vorgenommen werden. Dabei sollten möglichst viele irrelevante Attribute entfernt werden, um die Anzahl der Attribute x_1, x_2, \dots, x_n zu verringern. Dadurch wird die Dauer des Trainings verkürzt und unter Umständen auch die Qualität der Voraussage erhöht (vgl. [Gé17], S.205). Zur Identifizierung der irrelevanten Attribute bietet [Bra07] Kapitel 10.7 einen interessanten Ansatz über die Berechnung des *Informationsgewinns* je Attribut, welcher auch für das Kapitel 4.2 bei dem Aufbau von Decision Trees relevant sein wird.

2.4.3 Kodierung von kategorischen Werten

Zuletzt sollten im Falle der Klassifikation die Labels (Ergebniswerte) und auch andere kategorische Werte in eine geeignete numerische Form für den Machine Learning Algorithmus gebracht werden. Eine Lösung bietet die sogenannte *One-Hot-Kodierung* (vgl. [Gé17], S.63-64). Hierbei wird jedes Attribut bzw. Label als ein Vektor der Länge n dargestellt, wobei n die Anzahl der Kategorien ist. Anschließend wird jeder Kategorie ein fester Index $0, 1, \dots, n - 1$ im Vektor zugeordnet. Sollte nun eine kategorische Variable von einer bestimmten Kategorie $i, 0 \leq i < n$ sein, so ist nur

⁴siehe [PVG⁺11] und <https://scikit-learn.org/stable/modules/impute.html>

der Wert des Vektors an der Stelle i gleich 1. Alle anderen Indices erhalten den Wert 0. So würde ein Vektor eines Attributes x , das Kategorie 1 von insgesamt vier Kategorien (0, 1, 2 und 3) zugeordnet werden kann, wie folgt aussehen:

$$\vec{x} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad (2.3)$$

Ein Vorteil der One-Hot-Kodierung ist, dass der Klassifikator keine Relationen zwischen den Attributwerten finden kann, da es sich um linear unabhängige Einheitsvektoren handelt. So würde ein Klassifikator bei einer Kodierung mit ganzen Zahlen, wobei jedem Attributwert eine ganze Zahl zugeordnet wird, davon ausgehen, dass benachbarte Werte z.B. 1 und 2 sich ähnlicher sind als 1 und 5.

2.4.4 Aufteilung in Trainings- und Testdaten

Wenn wir uns an die Definition von Klassifikatoren aus Abschnitt 2.2 erinnern, bemerken wir, dass es nötig ist, den Klassifikator an vorher ungesehenen Instanzen zu bemessen, um ein realistisches Bild seiner Performance zu bekommen. Dies ist deshalb so wichtig, weil es grundsätzlich ein Leichtes wäre, den Klassifikator komplett auf die Trainingsdaten abzustimmen. Er könnte diese durch einfache Methoden auswendig lernen, ohne die tatsächlichen Beziehungen der Attribute zu den Labels zu verstehen.

Wenn wir die Trainingsdaten auch als Testdaten verwenden, würde die Performance z.B. gemessen an der Genauigkeit der Vorhersage (wie viele Instanzen der Testdaten sind richtig klassifiziert) extrem hoch sein. Dies ist wenig aussagekräftig, denn der Klassifikator soll, wie bereits gesagt, auch auf vorher nicht betrachteten Instanzen eine gute Genauigkeit erzielen. Genau deshalb sollten die Daten vorher in *Trainingsdaten* und *Testdaten* aufgeteilt werden. Die Trainingsdaten sind diejenigen Instanzen, von denen der Klassifikator die Ergebnisse kennt und mithilfe welcher er lernt. Die Testdaten wiederum sind diejenigen Instanzen, die er niemals während des Lernprozesses sieht und von denen er die Ergebnisse nicht kennt. Mit diesen Testdaten wird später seine Performance evaluiert.

Eine einfache Vorgehensweise wäre es, die Daten nur in ein Trainingsset und ein Testset zu teilen. Oft wird hier eine Aufteilung in 70% Trainingsdaten und 30% Testdaten vorgenommen. Ein großer Nachteil davon ist, dass die Ergebnisse stark

variieren können, abhängig davon, welche Instanzen für das Training bzw. für das Testen genutzt worden sind (vgl. [JWHT14], S.178). Dieses Problem kann gelöst werden, indem eine sogenannte *k-fache Kreuzvalidierung* (k-Fold Cross Validation) durchgeführt wird.

Bei der Kreuzvalidierung werden die Daten zufällig in k Gruppen aufgeteilt, wobei immer $k - 1$ Gruppen als Trainingsdaten genutzt werden und die nicht verwendete Gruppe die Testdaten enthält. Dieses Verfahren wird dann k mal wiederholt, bis die Datenpunkte jeder Gruppe einmal als Testdaten verwendet wurden. Somit werden alle Daten sowohl für das Training, als auch für das Testen genutzt. Abschließend kann über die k Testergebnisse das arithmetische Mittel genommen werden. Das Ergebnis hat im Allgemeinen eine deutlich niedrigere Varianz gegenüber der einfachen Aufteilung (vgl. [JWHT14], S.182).

Das in dieser Arbeit implementierte Evaluationsrahmenwerk nutzt standardmäßig eine 5x2 stratifizierte Kreuzvalidierung.⁵ Hier werden die Daten zufällig jeweils zu 50% in Trainings- und Testdaten aufgeteilt, danach werden Trainings- und Testdaten getauscht. Die Stratifizierung stellt dabei sicher, dass die Klassen gemäß ihrer Verteilung im Datensatz auf die beiden Gruppen verteilt werden, sodass keine Klasse unterrepräsentiert ist. Dies wird 5 Mal wiederholt. Es entstehen insgesamt also 10 Ergebnisse auf verschiedenen Testsets. Dieses Verfahren wurde in *Approximate statistical tests for comparing supervised classification learning algorithms* ([Die98]) als bewährte Methode für den Vergleich von Klassifikatoren vorgeschlagen und ermöglicht es dem Nutzer, die Ergebnisse für spätere statistische Tests (z.B. t-Test) zu verwenden.

⁵https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

Weiterführende Arbeiten

Die Auswahl geeigneter Metriken zur Evaluation von Klassifikatoren und mögliche Einflüsse auf diese bilden den theoretischen Kern des implementierten Evaluationsrahmenwerkes. Einige Aspekte der Evaluation wurden bereits in bestehenden wissenschaftlichen Arbeiten genauer untersucht. Im Folgenden werden drei wichtige Arbeiten genannt und kurz zusammengefasst, deren Erkenntnisse einen maßgeblichen Einfluss auf die gewählten Methoden der Evaluation gehabt haben. Diese theoretischen Erkenntnisse werden in Kapitel 5 angewandt und diskutiert.

Die erste Arbeit *A systematic analysis of performance measures for classification tasks* ([SL09]) stellt verschiedene qualitative Metriken zur Evaluation vor und vergleicht, inwiefern sich diese für verschiedene Szenarien eignen. Hierfür werden verschiedene Werte innerhalb einer Konfusionsmatrix geändert, um zu betrachten, wie sich die Metriken für die jeweiligen Anpassungen der Konfusionsmatrix verändern. Anschließend werden die Ergebnisse analysiert und Schlüsse gezogen, für welche Anwendungsfälle die verschiedenen Metriken geeignet sind.

Die zweite Arbeit *An Experimental Comparison of Performance Measures for Classification* ([FHOM09]) verwendet hierfür einen praktischen Ansatz innerhalb eines Experimentes. In dieser Arbeit werden einige Metriken für Datensätze verschiedener Größe und Klassenverteilungen für eine Auswahl an Klassifikatoren evaluiert. Es wird auch beobachtet, inwiefern die Metriken von unterschiedlichen Änderungen des Datensatzes beeinflusst werden. Anschließend wird versucht Korrelationen, Beziehungen, aber auch Unterschiede zwischen den Metriken für die angewandten Szenarien zu erkennen.

Die dritte Arbeit *An introduction to ROC analysis* ([Faw06]) erläutert die Anwendung der ROC Analyse (Receiver Operating Characteristic), um die probabilistischen Ergebnisse eines Klassifikators zu analysieren und somit optimale Schwellwerte für die Klassifikationsentscheidung hinsichtlich der benötigten Anforderungen zu finden. Außerdem wird beschrieben, wie die ROC Analyse auf den Mehrklassenfall erweitert werden kann.

Eine Auswahl von Klassifikatoren

Im folgenden Kapitel werden einige Klassifikationsalgorithmen betrachtet, von welchen auch Varianten in der späteren Implementierung als Vergleichsklassifikatoren verwendet werden. Ein Verständnis dieser Klassifikatoren soll es dem Leser erlauben, deren mögliche Eingabeparameter in der Anwendung anzupassen, sowie diese auch für eigene Klassifikationsprobleme einzusetzen und zu optimieren. Es werden im Folgenden alle in diesem Kapitel enthaltenen Klassifikatoren sowie ihre verwendeten Implementierungen in den Fußnoten gelistet: *Naive Bayes*¹, *Decision Tree*², *Random Forest*³, *Gradient Descent*⁴ und *neuronales Netz*⁵.

4.1 Naive Bayes

Der folgende Abschnitt behandelt einen Klassifikator, der auf einem zentralen Satz der Wahrscheinlichkeitstheorie basiert, dem *Satz von Bayes*, und orientiert sich dabei an den Werken [Kub15] Kapitel 2 und [Bra07] Abschnitt 3.2. Der Satz von Bayes, benannt nach dem englischen Mathematiker *Thomas Bayes*, ermöglicht es durch Umstellen der Formel der *bedingten Wahrscheinlichkeit*,

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad (4.1)$$

die *umgekehrte bedingte Wahrscheinlichkeit* eines Ereignisses A unter der Bedingung, dass ein zweites Ereignis B bereits eingetreten ist, zu berechnen:

¹https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB

²<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier>

³<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier>

⁴https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier

⁵https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (4.2)$$

4.1.1 Vorgehensweise

Um zu sehen, wie der Satz von Bayes eingesetzt werden kann, um ein Klassifikationsproblem zu lösen, betrachten wir nun die angepasste Tabelle zur Einordnung von Diabeteserkrankungen in Abbildung 4.1 als Beispiel. Die Tabelle wurde selbst erstellt und dient lediglich zur Veranschaulichung der Funktionsweise eines Naive Bayes Klassifikators.

	Geschlecht	Übergewicht	Hoher Blutzuckerspiegel	Diabetes
0	Männlich	Ja	Ja	Ja
1	Männlich	Nein	Nein	Nein
2	Männlich	Nein	Nein	Nein
3	Männlich	Ja	Ja	Ja
4	Männlich	Nein	Ja	Ja
5	Weiblich	Nein	Ja	Nein
6	Weiblich	Nein	Nein	Ja
7	Weiblich	Ja	Nein	Nein
8	Weiblich	Ja	Ja	Ja
9	Weiblich	Ja	Nein	Nein

Abbildung 4.1: Angepasste Diabetes-Tabelle mit kategorischen Attributen

Zur Vereinfachung wird hier eine binäre Klassifikation mit binären Attributen betrachtet. Das Verfahren ist allerdings genauso für multinomiale Klassifikationen mit Attributen, die im Allgemeinen mehr als zwei Werte annehmen, anwendbar.

Zunächst müssen wir die *A-Priori Wahrscheinlichkeit* der Klassen des Zielwertes „Diabetes“ bestimmen. Diese entspricht der Wahrscheinlichkeit, dass eine zufällig betrachtete Person an Diabetes erkrankt ist. Da wir diese Wahrscheinlichkeit nicht wissen und auch keine weiteren empirischen Daten darüber verfügen, wie häufig eine zufällig gewählte Person im Einsatzgebiet des Klassifikators an Diabetes erkrankt

ist, können wir diese nur durch die Häufigkeit des Auftretens innerhalb der Tabelle schätzen:

- $P(\text{Diabetes} = \text{Ja}) = \frac{5}{10} = \frac{1}{2}$ und analog
- $P(\text{Diabetes} = \text{Nein}) = 1 - \frac{5}{10} = \frac{1}{2}$

Danach werden für jede Klasse die bedingten Wahrscheinlichkeiten der Attribute berechnet, also die Wahrscheinlichkeit, dass eine Person einen bestimmten Attributwert besitzt unter der Bedingung, dass die Person dieser Klasse angehört:

- $P(\text{Geschlecht} = \text{M} \mid \text{Diabetes} = \text{Ja}) = \frac{3}{5}$
- $P(\text{Geschlecht} = \text{W} \mid \text{Diabetes} = \text{Ja}) = \frac{2}{5}$
- $P(\text{Übergewicht} = \text{Ja} \mid \text{Diabetes} = \text{Ja}) = \frac{3}{5}$
- $P(\text{Übergewicht} = \text{Nein} \mid \text{Diabetes} = \text{Ja}) = \frac{2}{5}$
- $P(\text{Hoher Blutzuckerspiegel} = \text{Ja} \mid \text{Diabetes} = \text{Ja}) = \frac{4}{5}$
- $P(\text{Hoher Blutzuckerspiegel} = \text{Nein} \mid \text{Diabetes} = \text{Ja}) = \frac{1}{5}$

Analog können die Wahrscheinlichkeiten für die Klasse „Diabetes = Nein“ berechnet werden. Um jetzt einen neuen Datensatz mit unbekanntem Ergebniswert zu klassifizieren, wird mit Hilfe des Satzes von Bayes die *A-posteriori* Wahrscheinlichkeit der Klasse A unter den Bedingungen X_1, X_2, \dots, X_n , wie folgt umgeformt:

$$P(A \mid X_1 \cap X_2 \cap \dots \cap X_n) = P(A) \cdot P(X_1 \cap X_2 \cap \dots \cap X_n \mid A) \quad (4.3)$$

Unter der Annahme, dass die Attribute X_1, X_2, \dots, X_n paarweise *stochastisch unabhängig* sind, kann die Wahrscheinlichkeit auch ausgedrückt werden als:

$$P(A \mid X_1 \cap X_2 \cap \dots \cap X_n) = P(A) \cdot P(X_1 \mid A) \cdot P(X_2 \mid A) \cdot \dots \cdot P(X_n \mid A) \quad (4.4)$$

Wir könnten nun also in unserem Beispiel die Wahrscheinlichkeit $P(\text{Diabetes} = \text{Ja} \mid \text{Geschlecht} = \text{M} \cap \text{Übergewicht} = \text{Ja} \cap \text{Hoher Blutzuckerspiegel} = \text{Ja})$ durch Einsetzen der oben berechneten Werte ausrechnen:

$$\begin{aligned} P(\text{Diabetes} = \text{Ja} \mid \text{Geschlecht} = \text{M} \cap \text{Übergewicht} = \text{Ja} \cap \text{Hoher Blutzuckerspiegel} = \text{Ja}) &= \\ &= P(D = \text{Ja}) \cdot P(G = \text{M} \mid D = \text{Ja}) \cdot P(\text{ÜG} = \text{Ja} \mid D = \text{Ja}) \cdot P(\text{HB} = \text{Ja} \mid D = \text{Ja}) = \\ &= \frac{1}{2} \cdot \frac{3}{5} \cdot \frac{3}{5} \cdot \frac{4}{5} = 0,144 \end{aligned}$$

Danach sind noch die Wahrscheinlichkeiten der anderen Klassen für die gegebenen Attributwerte zu berechnen. In unserem Fall gibt es nur eine weitere Klasse:

$$\begin{aligned} P(\text{Diabetes} = \text{Nein} \mid \text{Geschlecht} = \text{M} \cap \text{ÜG} = \text{Ja} \cap \text{HB} = \text{Ja}) &= \\ &= \frac{1}{2} \cdot \frac{2}{5} \cdot \frac{2}{5} \cdot \frac{1}{5} = 0,016 \end{aligned}$$

Schließlich wählt der Klassifikator die höchste von allen berechneten A-Posteriori Wahrscheinlichkeiten⁶. In diesem Fall ist das die Klasse Diabetes = Ja, denn $0,144 > 0,016$.

4.1.2 Erweiterung auf kontinuierliche Attribute

Das vorher gewählte Beispiel beschränkt sich bisher ausschließlich auf kategoriale Attribute. Um die bedingten Wahrscheinlichkeiten für diese zu bestimmen, ist es lediglich erforderlich, die Häufigkeit dieser innerhalb jeder Klasse zu zählen. Sollten die Instanzen kontinuierliche, numerische Attribute aufweisen, bedarf es hingegen einer anderen Methode, die Wahrscheinlichkeiten zu bestimmen, denn hier ist einfaches Zählen nicht mehr möglich. Im Folgenden werden hierfür zwei Strategien aus [Kub15], S.30-32) beschrieben.

Die erste Möglichkeit wäre, die Attribute zu *diskretisieren*. Hier werden für jedes kontinuierliche Attribut Intervalle bestimmt, und jedem Intervall eine ganze Zahl oder ein String zugeordnet. Liegt der Wert innerhalb dieses Intervalls, so nimmt das Attribut den jeweiligen Wert der ganzen Zahl oder des Strings an. Zum Beispiel könnte ein Attribut Körpergröße in die Strings „klein“, „normal“ und „groß“ eingeteilt werden. Die Intervalle könnten wie folgt definiert sein:

- $x > 170\text{cm}$: klein
- $170\text{cm} \geq x \geq 185\text{cm}$: mittel
- $185\text{cm} < x$: groß

Eine 175 cm große Person würde also als „mittel“ eingestuft werden. Mit diesem Verfahren werden kontinuierliche Attribute zu kategorischen Attributen umgewandelt und die in Abschnitt 4.1.1 beschriebene Vorgehensweise die Häufigkeiten zu zählen kann wie zuvor angewandt werden.

⁶Beachte: Die Summe der A-Posteriori Wahrscheinlichkeiten muss nicht 1 ergeben

Der zweite Ansatz verwendet die *Gaußsche Normalverteilung*, um Wahrscheinlichkeiten von kontinuierlichen Attributen zu bestimmen und wird deshalb auch als *Gaussian Naive Bayes* bezeichnet. Dafür werden für alle Klassen die Normalverteilungen der Attribute bestimmt. Dies geschieht durch die Berechnung des Erwartungswertes μ (Arithmetisches Mittel des Attributs) und der Varianz σ . Diese werden dann in die Formel der gaußschen Normalverteilung eingesetzt:

$$L(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (4.5)$$

Mit dieser *Wahrscheinlichkeitsdichtefunktion* L kann die Wahrscheinlichkeitsdichte für einen konkreten Wert berechnet werden (das Ergebnis ist dann äquivalent zu der vorher bestimmten Wahrscheinlichkeit einer Kategorie). Nach den in Abbildung 4.2 dargestellten Normalverteilungen könnten beispielsweise Männer und Frauen anhand ihrer Körpergröße klassifiziert werden.

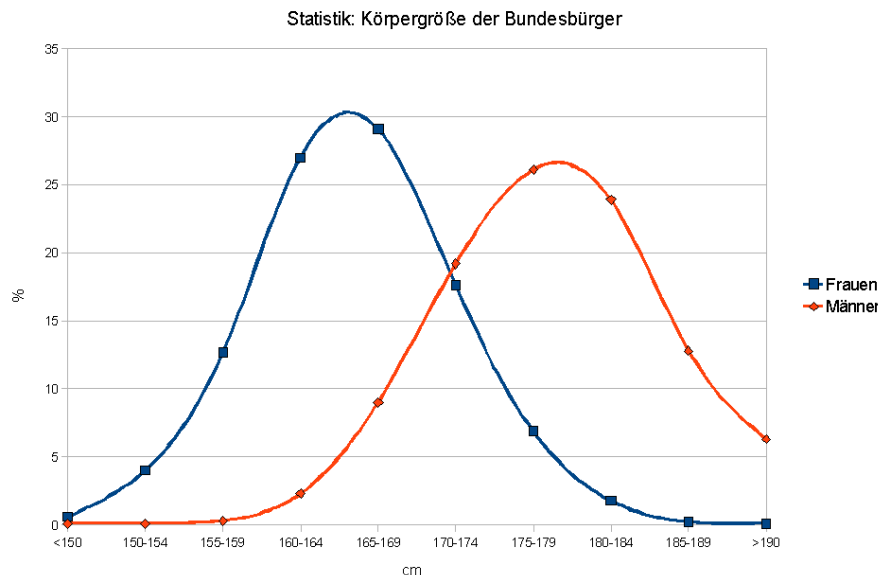


Abbildung 4.2: Normalverteilungen der Körpergröße von Frauen und Männern in Deutschland. Quelle: https://de.wikipedia.org/wiki/K%C3%B6rpergr%C3%B6%C3%9Fe_eines_Menschen, Stand: 06.05.2021

Wählen wir nun den Wert 160-164 cm, so entspräche der Wert der Wahrscheinlichkeitsdichtefunktion der Frauen an dieser Stelle ungefähr 0,27, für Männer ungefähr 0,025. Dieser Wert muss nun noch mit der A-Priori Wahrscheinlichkeit der Kategorie Frau/Mann multipliziert werden, um die A-Posteriori Wahrscheinlichkeit zu erhalten und anschließend eine Klassifikation nach höchster Wahrscheinlichkeit vornehmen zu können.

Häufig wird innerhalb der Berechnung der A-Posteriori Wahrscheinlichkeit der natürliche Logarithmus \ln für den finalen Wert verwendet, um einen Underflow für sehr kleine Werte zu verhindern. Durch Anwenden der Rechenregeln für Logarithmen kann so der *Score* der A-Posteriori Wahrscheinlichkeit $P(\text{weiblich} \mid 160\text{-}164)$ wie folgt berechnet werden (wir unterstellen an dieser Stelle, dass die A-Priori Wahrscheinlichkeit für Frauen und Männer = 0,5 ist):

$$\begin{aligned} \text{Score}(\text{weiblich} \mid 160\text{-}164) &= \ln(P(\text{weiblich}) \cdot L(160\text{-}164 \mid \text{Frau})) = \\ &= \ln(P(\text{weiblich})) + \ln(P(160\text{-}164 \mid \text{weiblich})) = \ln(0,5) + \ln(0,27) \approx \\ &\approx -0.69 + -1.31 \approx -2 \end{aligned}$$

Der Score für $P(\text{männlich} \mid 160\text{-}164)$ ist hingegen mit etwa -4,38 niedriger als der eben berechnete, so würde schließlich eine 160-164 große Person als weiblich klassifiziert werden. Natürlich sollten für eine optimale Klassifikation mehr Attribute als nur die Körpergröße verwendet werden, hier ging es lediglich um die Veranschaulichung der Berechnungen eines Gaussian Naive Bayes Klassifikators.

4.2 Decision Tree

Ein *Decision Tree* (oder auch Entscheidungsbaum) klassifiziert einen Datensatz nach einer Folge von Entscheidungsregeln, welche hierarchisch an den Zweigen innerhalb einer Baumstruktur getroffen werden. Es gibt eine Vielzahl von Algorithmen zur Erstellung eines solchen Baumes. Sehr prominente Vertreter sind unter anderem *ID3* und *C4.5*, deren Vorgehen vereinfacht in [CL16], S.96-108) erläutert werden und an denen sich der folgende Abschnitt orientiert. Zunächst werden die Grundlagen von Decision Trees an einem Beispiel erläutert und anschließend werden mögliche Optimierungen vorgestellt.

4.2.1 Aufbau

Betrachten wir ein einfaches Beispiel in Abbildung 4.3. Die Tabelle enthält Datenpunkte, deren Labels die Information enthalten, ob ein Verkehrsmittel auf einer bestimmten Strecke bei niedrigem, mittleren oder hohem Verkehrsaufkommen sowie in den Jahreszeiten Sommer oder Winter pünktlich war oder nicht. So könnte ein möglicher Entscheidungsbaum wie in Abbildung 4.4 dargestellt aussehen.

In dem dargestellten Baum bildet das Attribut „Jahreszeit“ den *Wurzelknoten*, das Attribut „Verkehrsaufkommen“ die nächsten *inneren Knoten* und die *Blätter* stellen die Anzahl der Instanzen des Pfades für das Label „Pünktlich = Ja“ oder „Pünktlich =

	Verkehrsaufkommen	Jahreszeit	Pünktlich
0	niedrig	Sommer	Ja
1	mittel	Winter	Nein
2	hoch	Winter	Nein
3	niedrig	Sommer	Ja
4	mittel	Winter	Ja
5	hoch	Sommer	Nein
6	mittel	Sommer	Ja
7	niedrig	Winter	Ja
8	mittel	Winter	Nein
9	hoch	Sommer	Ja
10	niedrig	Winter	Ja
11	mittel	Sommer	Ja

Abbildung 4.3: Tabelle zur Bestimmung der Pünktlichkeit von Verkehrsmitteln

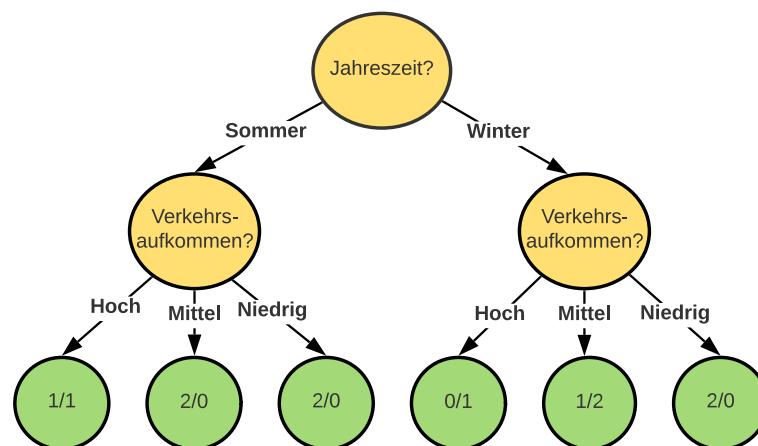


Abbildung 4.4: Beispiel eines Decision Trees für die Tabelle 4.3

Nein“ gegenüber. So gibt es beispielsweise für Verkehrsmittel mit den Attributwerten *Jahreszeit* = *Sommer* und *Verkehrsaufkommen* = *Mittel* zwei Verkehrsmittel, die pünktlich waren und keines, das nicht pünktlich war.

Im Allgemeinen wird so später die Entscheidung für das Label mit der höchsten Anzahl im Blatt Knoten getroffen. Der Pfad *Verkehrsaufkommen* = *Hoch* und *Jahreszeit* = *Sommer* ergäbe eine Pattsituation, da es jeweils eine Instanz gibt mit *Pünktlich* = *Ja* und eine Instanz mit *Pünktlich* = *Nein*, hier müsste die Entscheidung für eine der Kategorien zufällig getroffen werden, bzw. mehr Information gewonnen werden. Dies ist ein Beispiel eines Entscheidungsbaumes, es gibt allerdings eine Vielzahl an Möglichkeiten, den Baum aufzubauen. Die Problemstellung ist nun, eine optimale Reihenfolge der Verzweigung der Attribute zu finden, um eine möglichst genaue Klassifikation zu erzielen.

Eine Möglichkeit, das nächste Attribut zu wählen, wäre die Bestimmung des *Informationsgewinns* für alle möglichen Attribute und dann das Attribut mit dem höchsten Informationsgewinn zu wählen. Dieses Verfahren wird auch in dem von [Qui86] vorgestellten *ID3*-Algorithmus verwendet. Dafür müssen wir zunächst den Informationsgehalt I eines Attributes X berechnen, welcher nach dem US-amerikanischen Mathematiker *Claude Shannon* und Begründer der Informationstheorie wie folgt definiert ist:

$$I(X) = \sum_{i=1}^k -p(b_i) \cdot \log_2 p(b_i) \quad (4.6)$$

wobei k die möglichen Ausprägungen des Attributes, b_i die möglichen Werte des Attributes und $p(b_i)$ die relativen Häufigkeiten dieser Werte darstellt. Der Informationsgehalt eines Attributes X kann als Unordnung oder *Entropie* der Zielmenge verstanden werden. Sollte eine Entscheidung (Abzweigung) nach einem Attribut X eine *reine* Zielmenge bezüglich des Labels, in unserem Fall „Pünktlich Ja/Nein“ erzielen, so wären in den folgenden Kindern (in diesem Fall sind die Kinder *immer* Blätter) nur noch Instanzen eines Zielwertes (also Ja oder Nein). Die Entropie wäre also gleich 0 (beachte, dass hier direkt die Entropie 0 gesetzt werden muss, da der Logarithmus für den Wert 0 nicht definiert ist).

Die größte Entropie würde hingegen entstehen, wenn in den folgenden Kindern nach einer Abzweigung des Attributes X jeweils gleich viele Instanzen der beiden Labels stehen würden. Es gilt also diese Entropie zu minimieren, denn je niedriger die Entropie, desto besser trennt das Attribut X die Klassen nach der Verzweigung. Diese Erkenntnis zeigt sich auch in der Formel für die Berechnung des *Informationsgewinns*:

$$\text{gewinn}(X) = I(E) - G(X) \quad (4.7)$$

wobei $I(E)$ der Informationsgehalt vor der Verzweigung mit dem Attribut X ist und $G(X)$ der *Gain* von X nach der Verzweigung ist. Bestimmen wir als Beispiel die Wurzel, nach welcher gemäß dieser Berechnung innerhalb der Diabetes-Tabelle verzweigt werden sollte. Hierfür wird zunächst $I(E)$ berechnet, was in diesem Fall der Entropie der Klassen innerhalb der gesamten Tabelle entspricht, da zuvor noch keine Verzweigung stattgefunden hat:

$$I(\text{Tabelle}) = -\frac{8}{12} \cdot \log_2\left(\frac{8}{12}\right) - \frac{4}{12} \cdot \log_2\left(\frac{4}{12}\right) \approx 0,92$$

Der Gain $G(X)$ wird wie folgt für jedes Attribut berechnet:

$$G(X) = \sum_{j=1}^n \frac{|E_j|}{|E|} \cdot I(E_j) \quad (4.8)$$

wobei n der Anzahl der Ausprägungen des Attributes, $|E|$ der Anzahl der zuteilenden Instanzen, $|E_j|$ der Anzahl der Instanzen innerhalb des Teilbaumes j und $I(E_j)$ dem Informationsgehalt des Teilbaumes j entspricht. Der Gain $G(X)$ entspricht also dem gewichteten Informationsgehalt der Teilbäume nach Verzweigung nach Attribut X . Verzweigen wir beispielsweise nach dem Attribut „Jahreszeit“ (kurz JZ), so wäre $n = 2$, da das Attribut nur zwei mögliche Werte hat. Sei E_1 : JZ = S und E_2 : JZ = W, so wären die Informationsgehalte der beiden Ereignisse

- $I(E_1) = -\frac{5}{6} \cdot \log_2\left(\frac{5}{6}\right) - \frac{1}{6} \cdot \log_2\left(\frac{1}{6}\right) \approx 0,65$ und
- $I(E_2) = -\frac{3}{6} \cdot \log_2\left(\frac{3}{6}\right) - \frac{3}{6} \cdot \log_2\left(\frac{3}{6}\right) = 1$

Der Ast *Jahreszeit* = *Winter* hat eine gleiche Verteilung der beiden Klassen, er enthält also die maximale Entropie 1 (im zwei Klassen Fall). Berechnen wir nun den Gain des Attributes Jahreszeit:

$$G(JZ) = \frac{6}{12} \cdot 0,65 + \frac{6}{12} \cdot 1 = 0,825$$

Setzen wir dieses Ergebnis in die Formel des Informationsgewinns ein, so erhalten wir

$$\text{gewinn}(JZ) = 0,92 - 0,825 = 0,095$$

Berechnen wir analog den Gewinn des Attributes Verkehrsaufkommen, so erhalten wir ungefähr einen Wert von 0,63 für den Gain und einen Wert von 0,29 für den Gewinn. Die Verzweigung nach dem Attribut Verkehrsaufkommen in der Wurzel hat also einen höheren Informationsgewinn als unser vorheriger Baum aus Abbildung

4.4 mit dem Attribut Jahreszeit in der Wurzel. Er teilt die Klassen im Allgemeinen besser, denn wenn beispielsweise zuerst der Wert „niedrig“ für das Attribut Verkehrsaufkommen betrachtet wird, so haben 4 von 4 der Instanzen in der Tabelle den Wert „Pünktlich = Ja“.

Nach dem Bestimmen der Wurzel kann dieses Verfahren rekursiv für jeden weiteren Knoten fortgeführt werden. Vorheriger Informationsgehalt ist nun für den aktuellen Knoten zu berechnen und der Gain entsprechend für alle noch nicht betrachteten Attribute. Dieses Vorgehen wird so lange fortgesetzt, bis entweder innerhalb eines Pfades alle Attribute betrachtet worden sind bzw. keinen Informationsgewinn mehr liefern oder ein Knoten nur Instanzen einer Klasse enthält. In letzterem Fall ist es nicht nötig, weitere Attribute zu betrachten, da die Instanzen bereits vollständig getrennt wurden und keine neuen Informationen mehr gewonnen werden können (da die Entropie innerhalb dieses Knotens 0 beträgt).

Bisher ist das Verfahren ausschließlich auf kategoriale Attribute angewandt worden. Wenn numerische Attribute verwendet werden sollen, ist es nötig diese Attribute zu diskretisieren und in kategoriale Attribute umzuwandeln. Das Verfahren der Diskretisierung von Attributen ist bereits in Abschnitt 4.1.2 erläutert worden. Um eine möglichst gute Trennung der Klassen zu erreichen, kann erneut der Informationsgehalt als Kriterium betrachtet werden. Hier werden die Intervalle so gewählt, dass ein maximaler Informationsgewinn entsteht. Das genaue Vorgehen stammt aus dem populären C4.5 Algorithmus aus [Qui93] und kann dort nachgeschlagen werden.

Das in diesem Abschnitt beschriebene Verfahren ist die Grundlage vieler Varianten von Decision Tree Algorithmen, wobei natürlich eine Reihe von individuellen Optimierungen vorgenommen werden kann. Eine dieser Möglichkeiten wird im nächsten Abschnitt erläutert.

4.2.2 Pruning

Decision Trees, welche nach dem in letzten Abschnitt präsentierten Algorithmus aufgebaut werden, haben einen Nachteil, weshalb es oft nötig ist diese zu optimieren, um eine befriedigende Performance zu erhalten.

Der Nachteil ist, dass sie zu *Overfitting* tendieren (vgl. [Kub15], S.188). Overfitting beschreibt das Phänomen, wenn ein Klassifikator sehr gute Ergebnisse bezogen auf die Genauigkeit innerhalb der Trainingsdaten erreicht, allerdings signifikant schlechter in der späteren Evaluation mit unbekannten Testdaten abschneidet. Dies liegt daran, dass durch diesen Algorithmus im Allgemeinen äußerst große Bäume erzeugt werden, da Attribute so lange verzweigt werden, bis die Klasse perfekt separiert oder alle Attribute betrachtet worden sind. Somit werden oft in den

tieferen Stufen Attribute betrachtet, die wenig Informationsgehalt bezogen auf das Label bieten, da in den oberen Ebenen die Attribute mit höherem Informationsgehalt bereits betrachtet worden sind.

Attribute in tieferen Ebenen haben also oft nur eine minimale, eventuell sogar zufällige Korrelation mit dem Label. Sie werden allerdings trotzdem betrachtet, weil sie einen Informationsgewinn liefern. Dies ist insofern problematisch, da somit irrelevante Attribute Auswirkungen auf den Klassifikationsprozess haben und später die Performance auf unbekannten Instanzen verschlechtern können.

Eine Lösung für dieses Problem bietet das sogenannte *Pruning*. Dabei wird der Entscheidungsbaum insofern modifiziert, dass ansonsten generierte Teilbäume abgeschnitten und innere Knoten frühzeitig zu Blättern umgewandelt werden. Damit wird zwar der Versuch der perfekten Klassifikation der Trainingsdaten aufgegeben, aber dafür wird die Performance auf späteren Testdaten verbessert. Es gibt mehrere Parameter, mit denen die Tiefe des Baumes eingeschränkt werden kann:

Eine direkte Methode ist, die *maximale Tiefe* vor Beginn des Aufbaus des Baumes festzulegen. Im Gegensatz dazu ist ein etwas dynamischerer Parameter die *minimale Anzahl der Instanzen*, die ein Knoten besitzen muss, um sich weiter verzweigen zu dürfen (vgl. [PVG⁺11], S.174). In beiden Fällen verkleinern die beiden Parameter den Baum bei gezielter Anpassung und können so dem Overfitting entgegenwirken.

4.2.3 Random Forest

Es ist auch möglich, anstatt eines einzelnen Entscheidungsbaumes mehrere Entscheidungsbäume zu verwenden. Hierdurch wird sich eine bessere Performance versprochen, da mehrere Modelle für die Klassifikation betrachtet werden. Diese Menge an Bäumen wird auch *Random Forest* genannt. Random Forests werden mithilfe der *Bootstrap Aggregation* (oder auch einfach *Bagging*) und dem sogenannten *Feature Bagging* aufgebaut.

Die Bootstrap Aggregation teilt im ersten Schritt die Daten, ähnlich wie in 2.4.4 besprochen, in Trainings und Testdaten (hier Validationsdaten innerhalb des Trainingsprozesses). Der Unterschied ist, dass die einzelnen Instanzen hierbei *mehrfach* in den n *Bags*, den späteren Trainingsdaten für die jeweiligen n Decision Trees, auftreten dürfen.

Jeder Decision Tree erhält also einen „Durchschnitt“ der Daten, auf denen er trainiert. Das Verbot von mehrfachem Auftreten der Daten in unterschiedlichen Bags würde die Menge der verwendeten Daten und deren Vielfalt einschränken. So wird auch eine Minimierung der Varianz der Trainingsdaten erzielt, was für den Trainingsprozess bedeutet, dass die Ergebnisse des Klassifikators weniger von den Trainingsdaten abhängig sind (vgl. [JWHT14], S.317).

Im zweiten Schritt werden nun für jeden dieser Bags eine echte Teilmenge der ursprünglichen Attribute bzw. Features *zufällig gewählt*, um somit einen Pruningeffekt wie oben beschrieben zu erreichen. Indem für jeden dieser Bags unterschiedliche Attribute gewählt werden, werden insgesamt trotzdem möglichst viele der ursprünglichen Attribute betrachtet. Für die Klassifikation ist eine häufig verwendete Anzahl beim Bagging \sqrt{n} Attribute (vgl. [RRC19], S.90). Danach werden die einzelnen Decision Trees mit den zufälligen Teilmengen der Attribute auf den jeweiligen Bag trainiert.

Schließlich wird die Performance der Decision Trees anhand der zugehörigen Validationsdaten gemessen und diejenige Teilmenge der Attribute mit der besten Performance verwendet. Hier werden sich erneut häufig die Attribute in höheren Ebenen befinden, die stärker mit dem Label korreliert sind. Allerdings haben alle Attribute zumindest eine Chance, auf die Entscheidung einzuwirken.

In der finalen Evaluation des kompletten Random Forests erhalten alle Decision Trees alle Instanzen der Testdaten und stimmen über ihre jeweiligen Klassen ab. Jede Instanz wird also derjenigen Klasse zugeordnet, die unter der kompletten Menge der Decision Trees am häufigsten gewählt worden ist.

Es ist erneut zu beachten, dass die späteren Testdaten nicht innerhalb des Trainingsprozesses verwendet werden dürfen. Hierfür werden bei der Klassifikation mit mehreren Modellen vorab Daten herausgenommen und als Testdaten gehalten, im Gegensatz zu den Validationsdaten, die während des Trainingsprozesses genutzt werden

4.3 Logistische Regression

Regressionsverfahren sind Methoden des statistischen Lernens, bei welchen mit Hilfe einer (nicht-)linearen Funktion in Abhängigkeit der Attribute möglichst genau die Ausgabe, in unserem Fall die Wahrscheinlichkeiten der Labels, approximiert werden soll. Diese bilden die Grundlage für die sogenannten *neuronalen Netze*, deren Aufbau später in Abschnitt 4.3.3 kurz beschrieben wird.

Die *logistische Regression* ist eine Variante der Regressionsverfahren, bei der die Ausgabe der Wahrscheinlichkeit der Zugehörigkeit zu einer von *zwei* Kategorien entspricht. Anhand dieser Wahrscheinlichkeiten kann im Kontext der Klassifikation ein Label gewählt werden. Bei dem Verfassen dieses Kapitels wurden die Werke [RRC19] Abschnitt 3.5, [JWHT14] Abschnitt 4.3 und [Nie18] als Hilfestellungen verwendet.

4.3.1 Hypothese

Betrachten wir zunächst eine einfache logistische Regression mit nur einem Attribut X und einem binären Label. Grundsätzlich können auch mehrere Parameter gewählt werden und, wie wir sehen werden, auch mehrklassige Labels. Wir beschränken uns zunächst allerdings auf diesen einfachen Fall.

Um die *Wahrscheinlichkeiten* der Klassen zu schätzen, müssen wir eine Funktion wählen, deren Ausgabe innerhalb eines Wertebereichs zwischen 0 und 1 liegt. Wie wir bereits aus Abschnitt 2.2 wissen, wird diese auch als *Hypothese* bezeichnet. Im Falle der logistischen Regression entspricht die Hypothese also einer konkreten mathematischen Funktion. Hierfür eignet sich eine *sigmoide Funktion* (eine S-förmige Funktion). Wir wählen die logistische Funktion:

$$P(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}} \quad (4.9)$$

In Abbildung 4.5 sehen wir eine solche sigmoide Funktion. Die roten Punkte stellen „Nein“-Instanzen dar, während die grünen Punkte „Ja“-Instanzen des Labels darstellen. Der Wert der sigmoiden Funktion bildet also die Wahrscheinlichkeit der Zugehörigkeit zu Klasse „Ja“ unter der Bedingung des Wertes von Attribut X , ab.

Wählen wir die *Schwelle* $y = 0,5$, so werden Instanzen mit einer Wahrscheinlichkeit größer 50% der Klasse „Ja“ zugeordnet. Nehmen wir an, die x -Achse wäre das Gewicht von Mäusen, mit einem mittleren Gewicht von knapp unter 20g. So könnten Mäuse mit einem Gewicht größer 20g als übergewichtig und Mäuse mit einem Gewicht unter 20g als nicht übergewichtig klassifiziert werden, ohne deren Größe zu betrachten.

Wie sich erkennen lässt, werden somit auch Instanzen falsch klassifiziert, z.B. die beiden roten Instanzen mit einem Wert knapp über 20, welches möglicherweise Mäuse sind, die in Relation zu ihrer Größe nicht übergewichtig sind (oder analog die grüne Instanz links der gestrichelten Linie, die fälschlicherweise nicht als übergewichtig klassifiziert wurde).

Um uns den Zusammenhang der linearen Funktion des Gewichts mit der projizierten Wahrscheinlichkeit innerhalb der Sigmoid Funktion zu verdeutlichen, betrachten wir die *Log Odds*. Die Log Odds setzen in unserem Beispiel die Wahrscheinlichkeit einer übergewichtigen Maus in Relation zu einer nicht übergewichtigen Maus im Hinblick auf ihr Gewicht X . Wollen wir die Log Odds des Labels schätzen, so kann dies über eine lineare Funktion, die wir durch Umformen von Gleichung 4.9 erhalten, geschehen (siehe auch [JWHT14], S.132):

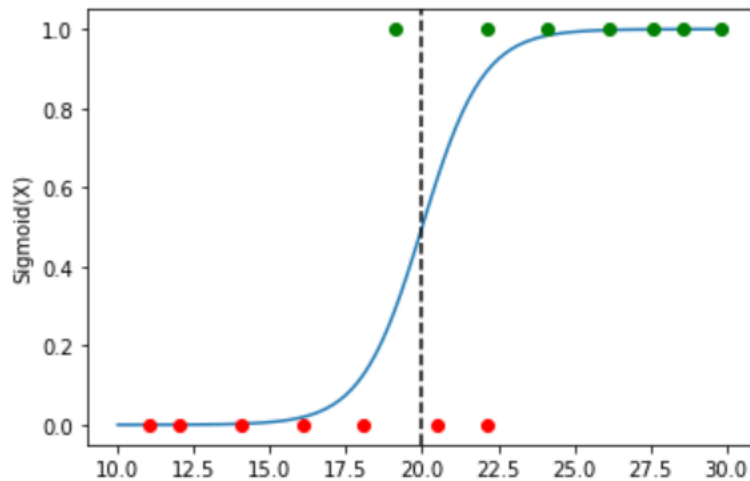


Abbildung 4.5: Eine sigmoide Funktion mit beispielhaften Instanzen und Schwelle = 0,5

$$\log\left(\frac{P(X)}{1 - P(X)}\right) = \beta_0 + \beta_1 X \quad (4.10)$$

wobei die Koeffizienten β_0 dem *Bias* (der Verschiebung) und β_1 dem *Weight* (der Gewichtung) des Attributes entsprechen (dies ist auch auf mehrere Parameter $\beta_0, \beta_1, \dots, \beta_n$ mit $\beta_0 + \beta_1 X_1, \dots, \beta_n X_n$ erweiterbar). Die Log Odds wägen die Wahrscheinlichkeiten der beiden Klassen gegeneinander ab ($p(x)$ für Klasse 1 und $1-p(x)$ für Klasse 0).

Ein höherer Wert entspricht folglich einer höheren Wahrscheinlichkeit der Klasse 1 und ein niedrigerer Wert entspricht analog einer niedrigeren Wahrscheinlichkeit der Klasse 1 (und einer höheren Wahrscheinlichkeit der Klasse 0). Mit der Veränderung des Gewichts um eine Einheit verändern sich also auch ihre Log Odds, wobei dies nicht in einem linearen Zusammenhang geschieht.

Betrachten wir erneut Abbildung 4.5. So lässt sich erkennen, dass sich z.B. die Wahrscheinlichkeit von Übergewicht im Bereich von 12,5g bis 15g nicht sonderlich verändert, aber im Bereich 17,5g bis 22,5g einen extremen Anstieg erfährt. Das liegt daran, dass der Bereich um 20g für besonders viele Mäuse in diesem Beispiel die Grenze zum Übergewicht zu sein scheint.

4.3.2 Gradient Descent Verfahren

Nachdem die Hypothese Funktion erläutert worden ist, soll unser Ziel jetzt sein, möglichst gute Parameter β_0 und β_1 für diese zu finden, um die Wahrscheinlichkeiten der Klassen in Abhängigkeit ihrer Attribute möglichst genau zu schätzen. Das *Gradient Descent Verfahren* bietet hierfür eine Möglichkeit.

Um dieses Verfahren anwenden zu können, müssen wir eine *Kostenfunktion* $J(\theta)$ definieren, mit der gemessen werden kann, wie genau die Trainingsdaten durch die Funktion klassifiziert werden. Wir verwenden die Kostenfunktion *Logistic Loss*:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i)) \quad (4.11)$$

wobei θ ein Vektor der abhängigen Parameter $\beta_0, \beta_1, \dots, \beta_n$ ist und $h(x)$ die Hypothese der Instanz x_i der Trainingsdaten. Die Hypothese $h(x)$ berechnet also die Wahrscheinlichkeit der Klasse „Ja“ für die Instanz x_i . Wir wählen als Hypothese $h(x)$ die logistische Funktion aus dem vorherigen Abschnitt mit den Parametern des Vektors θ . Die Variable y_i ist der tatsächliche Wert der Instanz x_i (0 oder 1).

Die Idee hinter dieser Funktion ist, dass für den tatsächlichen Wert $y=1$ der Term $(1 - y_i) \log(1 - h(x_i))$ entfällt. Dann sind die Kosten $y_i \log(h(x_i))$ umso kleiner, je höher die Wahrscheinlichkeit $h(x_i)$ für die Klasse 1 ist. Eine gute Vorhersage wird also weniger bestraft, denn für hohe Wahrscheinlichkeiten der Klasse 1 sind die Kosten fast 0. Für niedrigere Wahrscheinlichkeiten werden sie betragsmäßig größer (beachte $\log(x)$, $x < 1$ ist negativ, $\log(x)$, $x = 1$ ist 0).

Analog entfällt der Term $y_i \log(h(x_i))$ für $y=0$, also wird hier eine höhere Wahrscheinlichkeit für Klasse 1 im Term $(1 - y_i) \log(1 - h(x_i))$ bestraft. Diese Kosten werden für alle m Instanzen aufsummiert und schließlich mit $-\frac{1}{m}$ multipliziert, um ein positives arithmetisches Mittel zu erhalten.

Das Optimierungsproblem besteht darin, optimale Werte für die Parameter zu finden, die den mittleren Fehler minimieren. In Abbildung 4.6 wird dies zur besseren Anschauung für den 2-dimensionalen Fall für nur einen Parameter dargestellt. Für n Parameter hätte diese Funktion $n + 1$ Dimensionen.

Entlang der x-Achse steht der Parameter θ , entlang der y-Achse stehen die Kosten der Funktion in Abhängigkeit des Wertes von θ . Die gewählte Kostenfunktion hat eine günstige Eigenschaft: Sie ist konvex, also ist jedes ihrer lokalen Minima auch ein *globales Minimum*. Das Minimum der Funktion entspricht also immer den global niedrigsten Kosten, was die Anwendbarkeit des folgenden Verfahrens ermöglicht.

Anfangs können wir zufällige Werte für unsere Parameter aus θ wählen, häufig werden diese mit den Werten 0 oder 1 initialisiert. Um nun von den bekannten Kosten der eingezeichneten roten Punkte (welche das Ergebnis der Kostenfunktion für zufällig gewählte Parameter sein könnte) das Minimum zu finden, müssen die *ersten partiellen Ableitungen* für alle Parameter aus θ gebildet werden. Diese ergeben sich nach der Kostenfunktion in Gleichung 4.11 wie folgt:

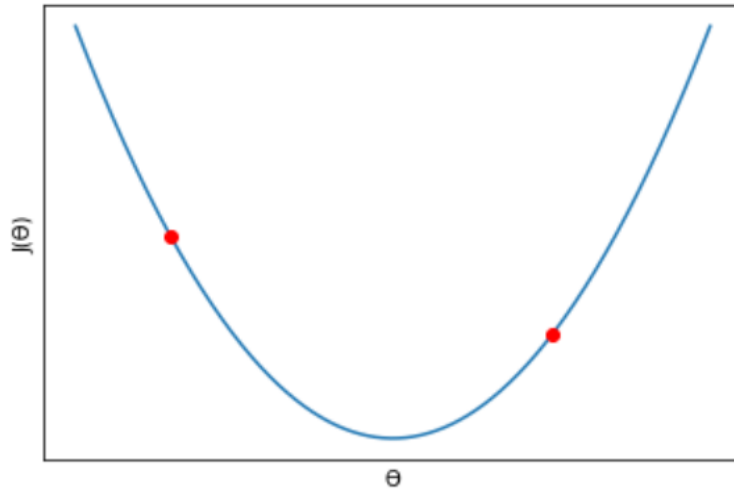


Abbildung 4.6: Veranschaulichung der 2-dimensionalen Kostenfunktion

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h(x^i) - y^i) \cdot x_j^i \quad (4.12)$$

wobei x^i Instanz i entspricht und x_j^i dem aktuellen Wert des Parameters j aus θ . Der Term $(h(x^i) - y^i)$ berechnet wie zuvor die Kosten. Ein negativer Wert der Ableitung (eine negative Steigung) zeigt uns also das, dass das Minimum weiter rechts liegen muss, wir also unseren Parameter θ_j erhöhen müssen. Analog bedeutet ein positiver Wert, dass wir den Parameter θ_j verringern müssen (beide Fälle sind angedeutet mit den roten Punkten in Abbildung 4.6. Dies geschieht mit folgender Korrektur:

$$\theta_j = \theta_j - \alpha \cdot \frac{\partial J}{\partial \theta_j} \quad (4.13)$$

Hier wird der Wert von θ_j aktualisiert, indem der alte Wert auf der rechten Seite um die Ableitung $\frac{\partial J}{\partial \theta_j}$ multipliziert mit der *Learning Rate* α erhöht bzw. verringert wird. Diese Korrektur ist umso größer, je weiter θ_j vom Minimum entfernt ist und analog umso kleiner, sobald er in der Nähe des Minimums liegt. Die Funktion konvergiert also gegen unseren angestrebten optimalen Wert von θ_j , wenn wir dieses Verfahren k -mal wiederholen, für eine ausreichend hohe Anzahl an k Schritten.

Um die Schrittgröße der Korrektur anzupassen und um sicherzustellen, dass das Optimum nicht durch zu große Schritte verpasst wird, sollte ein kleiner Wert für die Learning Rate α gewählt werden. Er darf allerdings auch nicht zu klein gewählt werden, da sonst deutlich mehr Schritte benötigt werden, um das Minimum zu

erreichen, und so die Laufzeit zunimmt (vgl. [RRC19], S.32-34). Das Verfahren wird noch einmal in folgendem Pseudocode zusammengefasst:

Algorithm 1 Logistic Gradient Descent

```
1: procedure OPTIMIZE PARAMS( $\theta$ , X, Y, k,  $\alpha$ )
2:    $\theta \leftarrow 0$ 
3:   for  $l \leftarrow 0$  to  $k - 1$  do
4:     for  $j \leftarrow 0$  to  $n - 1$  do
5:        $\frac{\partial J}{\partial \theta_j} \leftarrow \frac{1}{m} \sum_{i=1}^m (h(x^i) - y^i) \cdot x_j^i$ 
6:        $\theta_j \leftarrow \theta_j - \alpha \cdot \frac{\partial J}{\partial \theta_j}$ 
7:   return  $\theta$ 
```

Die $m \times n$ Matrix X entspricht dabei den m Trainingsinstanzen mit n Attributen, der Vektor Y den zugehörigen Labelwerten, der Vektor θ enthält wie zuvor beschrieben die $n + 1$ Parameter unserer Funktion. Das Argument k ist die Anzahl der Schritte und α ist unsere Learning Rate. Der Algorithmus terminiert also, nachdem er unsere Parameter aus θ , die hier zuvor mit 0 initialisiert wurden, k -mal korrigiert hat.

Um das in vorherigen Abschnitten angesprochene Problem des Overfittings zu minimieren, sollte zwar eine hohe Schrittzahl k gewählt werden, um ein ausreichend gutes Minimum zu finden, allerdings nicht zu hoch, da sonst die Gefahr besteht, dass der Algorithmus die Funktion zu stark an die Trainingsdaten angleicht. Dies ist alternativ auch zu verhindern, in dem eine Grenze für die Kosten gesetzt wird und der Algorithmus terminiert, sobald diese unterschritten wird.

Die Performance kann nun an den Testdaten gemessen werden. Hierfür werden die Testinstanzen in unsere neue Hypothese mit den optimierten Parametern aus der Prozedur gegeben und die Klasse, je nach der gewählten Schwelle, anhand der ausgegebenen Wahrscheinlichkeit bestimmt.

Das Verfahren kann wie bisher auch auf kategoriale Attribute angewandt werden, es ist nur nötig, diese in einem geeigneten Format zu kodieren. Dafür bietet sich beispielsweise das bereits in Abschnitt 2.4.3 besprochene One-Hot-Encoding an. Wir haben bisher nur den Fall für Labels mit zwei Klassen betrachtet, es gibt zwei Möglichkeiten, das Verfahren für den multinomialen Fall anzupassen.

Die erste Möglichkeit, wäre das Verfahren insofern zu erweitern, dass wir n Modelle für n Klassen trainieren. Jedes Modell beschränkt sich darauf, die Wahrscheinlichkeit einer Klasse i zu schätzen. Die Klasse i entspricht dabei dem Wert 1 und *alle anderen Klassen* dem Wert 0. Am Ende haben wir also ein Modell mit einer Hypothese für jede Klasse, wobei die Hypothese für jede Klasse unterschiedliche Parameter besitzt. Wenn wir nun Instanzen klassifizieren wollen, so geben wir diese in jede

Hypothese und wählen die Klasse mit der höchsten Wahrscheinlichkeit als Ausgabe ihrer Hypothese. Diese Variante wird als *One-vs-All Klassifikation* bezeichnet.

Die zweite Möglichkeit ist der *Softmax* Ansatz, der dies in einem Modell vereint. Die grundlegende Idee ist hier, die logistische Funktion durch eine Softmax-Funktion als Hypothese zu ersetzen, die es ermöglicht, die Wahrscheinlichkeiten für jede Klasse j der insgesamt C Klassen direkt zu schätzen:

$$P(X_j) = \frac{e^{z_j}}{\sum_{i=1}^C e^{z_i}}, \quad z_j = \sum_{i=0}^n \beta_i X_i \quad (4.14)$$

wobei z_j unserer linearen Funktion mit den Parametern für die Klasse j entspricht. Wir haben also *unterschiedliche* Parameter für jede Klasse, die in der Matrix θ mit $n + 1$ Reihen (wie bei dem Vektor im zwei Klassen Fall) und C Spalten enthalten sind. Entsprechend muss auch die Kostenfunktion angepasst werden:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^C -y_j \log(h(x_i)) \quad (4.15)$$

wobei $y_j = 1$ für die tatsächliche Klasse des Labels, für alle anderen 0. Es wird also versucht die Wahrscheinlichkeit des tatsächlichen Labels zu maximieren, indem nur die Kosten für das tatsächliche Label betrachtet werden. Das übrige Verfahren ist analog zu dem Gradient Descent Algorithmus der einfachen logistischen Regression, eine genauere Beschreibung findet sich unter <http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/>.

4.3.3 Neuronales Netz

Das logistische Modell, welches im letzten Abschnitt besprochen worden ist, bildet nun eine Einheit eines *neuronalen Netzes*, ein sogenanntes *Neuron*. Das neuronale Netz besteht aus mehreren Schichten, den *Layers*, welche wiederum aus einem oder mehreren Neuronen bestehen. In einem *Feedforward* neuronalem Netz ist jede vorhergehende Schicht mit der folgenden Schicht vernetzt, es gibt also keine Zyklen. Im Folgenden werden wir uns zur Einfachheit auf solche Feedforward Netze beschränken. Ein beispielhafter Aufbau der Struktur eines solchem neuronalen Netzes findet sich in Abbildung 4.7.

Der Trainingsprozess eines neuronalen Netztes lässt sich wie folgt vereinfacht darstellen: Zunächst erhält das *Input Layer* die Attribute der Daten, generell bietet es sich an, jedem Neuron des Input Layers genau ein Attribut zuzuweisen. Die Neuronen des Input Layers berechnen nun eine *Activation Function* σ , hier kann die Sigmoid Funktion aus dem letzten Abschnitt verwendet werden.

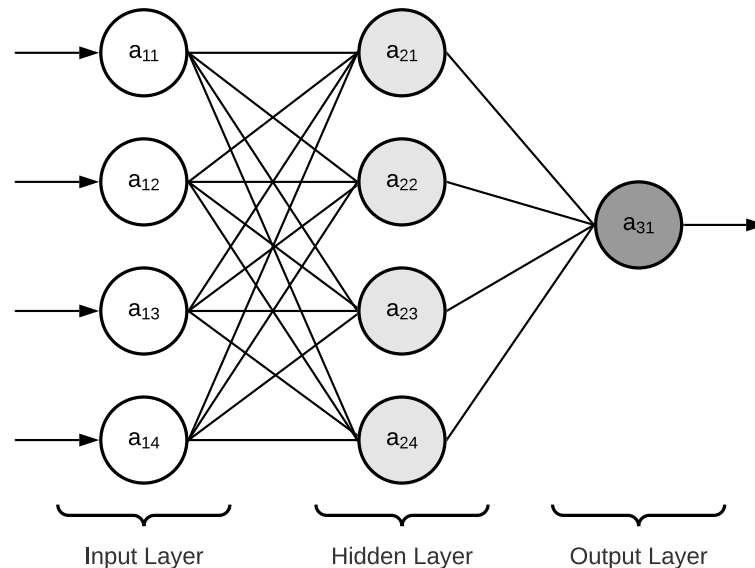


Abbildung 4.7: Aufbau eines neuronalen Netzes mit vier Neuronen im Input Layer, vier Neuronen im Hidden Layer und einem Neuron im Output Layer

Der berechnete Wert entspricht dann der Ausprägung des Attributes, abgebildet auf ein Intervall von 0 bis 1, wobei ihm durch das *Weight* des Attributes und dem *Bias* eine Relevanz für die spätere Klassifikation zugeordnet wird. Dieser Wert wiederum wird an jedes Neuron der nächsten Schicht der ersten *Hidden Layer* weitergereicht. Jedes Neuron des Hidden Layers errechnet dann die Sigmoid Funktion der Werte von den mit ihm vernetzten Neuronen aus dem vorhergehenden Layer, wobei diesen Werten erneut Gewichte und ein Bias zugeordnet werden.

In unserem Beispiel gibt es genau ein Hidden Layer, wobei es potentiell auch keines oder mehrere geben kann. Mit der Anzahl der Hidden Layers steigt auch die Komplexität des neuronalen Netzes. Diese zunehmende Komplexität ist auch der Vorteil gegenüber einem einzelnen logistischen Regressionsmodell.

Es ermöglicht dem neuronalen Netz im Allgemeinen komplexere Muster zu erkennen, denn wenn die Anzahl der Gewichte verglichen wird, fällt auf, dass in unserem Beispiel $4 \cdot 4 + 4 = 20$ verwendet werden. Im einfachen Fall der linearen Regression werden jedem Attribut nur ein Gewicht zugeordnet, also insgesamt 4 (sozusagen ein neuronales Netz ohne Hidden Layer). Diese zusätzlichen Hidden Layers ermöglichen es dem neuronalen Netz tiefere logische Verbindungen zu schließen und sich nicht auf eine lineare Trennung der Klassen zu beschränken (siehe auch [Nie18], Kapitel 4: *A visual proof that neural nets can compute any function*).

Schließlich werden im *Output Layer* die Wahrscheinlichkeiten der Zugehörigkeit zu den einzelnen Klassen berechnet. Im Zwei-Klassen-Fall genügt wie in Abbildung 4.7 ein einzelnes Sigmoid Neuron. Bei mehreren Klassen C bietet es sich an $|C|$

Neuronen mit einer Softmax Funktion, wie am Ende des vorherigen Abschnittes 4.3.2 beschrieben, zu verwenden. Die Wahl fällt wie zuvor auf die Klasse mit der höchsten Wahrscheinlichkeit.

Dieser Abschnitt umfasst lediglich die Grundlagen und den Aufbau neuronaler Netze. Es wird nur eine oberflächliche Beschreibung der komplexen Funktionsweise von neuronalen Netzen gegeben, welche über den Inhalt dieser Arbeit hinausgeht. Eine verständliche Erklärung einfacher neuronaler Netze findet sich in dem Buch [Nie18].

Evaluation von Klassifikatoren

In Kapitel 5 sollen einige Metriken und Methoden zur Evaluation von Klassifikatoren erläutert werden. Des Weiteren soll diskutiert werden, welchen Einfluss unterschiedliche Eigenschaften der Datensätze auf die spätere Performance und die Auswahl der Metriken haben können.

5.1 Metriken für Klassifikationsalgorithmen

5.1.1 Konfusionsmatrix

Bevor wir beginnen, verschiedene Metriken zu definieren, müssen wir zunächst betrachten, wie die Ergebnisse nach dem Test eines Klassifikators beschrieben werden können. Diese lassen sich für jeden Datenpunkt in eine der vier Kategorien einteilen: *richtig positiv*, *falsch positiv*, *falsch negativ* und *richtig negativ*. Dies lässt sich kompakt als 2×2 *Konfusionsmatrix* darstellen:

$$\begin{pmatrix} rp & fp \\ fn & rn \end{pmatrix} \quad (5.1)$$

wobei rp der Anzahl der richtig positiven Ergebnisse, fp der Anzahl der falsch positiven Ergebnisse, fn der Anzahl der falsch negativen Ergebnisse und rn der Anzahl der richtig negativen Ergebnisse entspricht. Für den Mehrklassenfall mit n Klassen, wobei $n > 2$ ist, erhält man eine $n \times n$ Konfusionsmatrix, die wie in Tabelle 5.1 dargestellt werden kann.

Der Wert einer Zelle Z_{ij} entspricht dabei der Anzahl der Instanzen der Klasse i , welche als Klasse j klassifiziert worden sind. Falls $i = j$, sind die Werte der Zellen genau die Anzahl der richtig klassifizierten Instanzen für die Klasse i , diese Zellen verlaufen entlang der Diagonale der Tabelle. Falls $i \neq j$, sind die Werte der Zellen genau die Anzahl von Instanzen, die fälschlicherweise als Klasse j klassifiziert wurden, obwohl sie tatsächlich Klasse i angehören.

		Gewählte Klasse				
		1	2	3	...	n
Tatsächliche Klasse	1	richtig	falsch	falsch	falsch	falsch
	2	falsch	richtig	falsch	falsch	falsch
	3	falsch	falsch	richtig	falsch	falsch
	...	falsch	falsch	falsch	richtig	falsch
	n	falsch	falsch	falsch	falsch	richtig

Tabelle 5.1: Konfusionsmatrix für mehrere Klassen

5.1.2 Definition der Metriken

Von einer gegebenen Konfusionsmatrix kann nun die Performance des Klassifikators für verschiedene Maßstäbe, die *Metriken*, abgeleitet werden. Die in diesem Kapitel verwendeten Metriken finden sich auch in [SL09], S.430. Im Folgenden werden zunächst Metriken definiert, die sich besonders für die binäre Klassifikation mit einer 2×2 Konfusionsmatrix eignen.

Die Konfusionsmatrix enthält die im vorigen Abschnitt beschriebenen Werte rp , fp , fn und rn , die den Ergebnissen aus den gesamten m Testinstanzen mit $m = rp + fp + fn + rn$ entsprechen. Zudem haben alle verwendeten Metriken einen Wertebereich von 0 bis 1, wobei 0 dem schlechtesten Wert und 1 dem bestmöglichen Wert entspricht.

- **Genauigkeit (Accuracy):** Die Genauigkeit ist die Anzahl der korrekt klassifizierten Instanzen geteilt durch die Anzahl aller Instanzen m und ist ein einfaches Maß, um die Effektivität eines Klassifikators zu evaluieren.

$$\text{Genauigkeit} = \frac{rp + rn}{m} \quad (5.2)$$

- **Relevanz (Precision):** Die Relevanz ist die Anzahl der richtig positiv klassifizierten Instanzen geteilt durch die Anzahl der richtig positiv klassifizierten Instanzen plus die Anzahl der falsch positiven Instanzen. Sie bestimmt die Übereinstimmung der Klassen des Labels mit den positiv klassifizierten Instanzen.

$$\text{Relevanz} = \frac{rp}{rp + fp} \quad (5.3)$$

- **Sensitivität (Recall):** Die Sensitivität ist die Anzahl der richtig positiv klassifizierten Instanzen geteilt durch die Anzahl der richtig positiv klassifizierten

Instanzen plus die Anzahl der falsch negativ klassifizierten Instanzen. Sie bestimmt die Effektivität des Klassifikators auf positiven Labels.

$$\text{Sensitivität} = \frac{rp}{rp + fn} \quad (5.4)$$

- **Spezifität (Specifity):** Die Spezifität ist die Anzahl der richtig negativ klassifizierten Instanzen geteilt durch die Anzahl der richtig negativ klassifizierten Instanzen plus die Anzahl der falsch positiven Instanzen. Sie bestimmt die Effektivität des Klassifikators auf negativen Labels.

$$\text{Spezifität} = \frac{rn}{rn + fp} \quad (5.5)$$

- **F-Score:** Der F-Score kombiniert die weiter oben beschriebenen Metriken Relevanz und Sensitivität, wobei durch den Parameter β der Sensitivität eine relative Gewichtung gegenüber der Relevanz gegeben werden kann.

$$\text{F-Score} = \frac{(1 + \beta^2)rp}{(1 + \beta^2)rp + \beta^2 fn + fp} \quad (5.6)$$

- **AUC:** Die AUC (Area Under Curve) kombiniert ebenso die Parameter Relevanz und Sensitivität, indem sie die Fläche unter der Kurve bemisst, wenn die Relevanz und die Sensitivität die Achsen für ein zweidimensionales Koordinatensystem bilden. Sie ermittelt die Fähigkeit des Klassifikators Fehler zu vermeiden.

$$\text{AUC} = \frac{1}{2} \left(\frac{rp}{rp + fn} + \frac{rn}{rn + fp} \right) \quad (5.7)$$

Betrachten wir jetzt Metriken, die sich für den Mehrklassenfall mit Klassen $C_i, 1 \leq i \leq l$ eignen. Erinnern wir uns an die in Tabelle 5.1 dargestellte Konfusionsmatrix, so kann für jede Klasse k auch eine 2×2 Konfusionsmatrix gebildet werden. Dies geschieht, indem der Eintrag $i = j = k$ den richtig positiven, der Rest der Zeile $k = i$, wobei $k \neq j$ summiert den falsch negativen, der Rest der Spalte $k = j$, wobei $k \neq i$ summiert den falsch positiven und alle weiteren übrigen Werte $k \neq i$ und $k \neq j$ summiert den richtig negativen zugeordnet werden.

Jetzt können für jede Klasse die zuvor beschriebenen Metriken berechnet werden. Um nun allerdings ein einziges Ergebnis über alle Klassen zu erhalten, werden Makro bzw. Mikro gemittelte Werte gebildet. Makro Metriken (gekennzeichnet mit Index M), berechnen die Metrik für jede individuelle Klasse unabhängig und bilden schließlich für diese das arithmetische Mittel, wobei dadurch jeder Klasse die gleiche Gewichtung zufällt, unabhängig ihrer Verteilung innerhalb des Datensatzes.

Die Mikro Metriken (gekennzeichnet mit Index μ) berechnen den aggregierten Durchschnitt für alle Klassen. Hierbei wird zusätzlich die Klassenverteilung miteinbezogen, da innerhalb dieser Berechnung indirekt jeder Klasse ihr Gewicht entsprechend ihrer Verteilung zugewiesen wird. Die Variablen rp_i, fp_i, fn_i, rn_i entsprechen erneut den Werten der Konfusionsmatrix respektive der Klasse i und m_i ist die Summe dieser, so können folgende Metriken für den Mehrklassenfall berechnet werden:

- **Makro Genauigkeit (Macro Average Accuracy):** Die Makro Genauigkeit berechnet die durchschnittliche Effektivität des Klassifikators pro Klasse.

$$\text{Genauigkeit}_M = \frac{\sum_{i=1}^l \frac{rp_i + rn_i}{m_i}}{l} \quad (5.8)$$

- **Makro Fehlerrate (Macro Average Error Rate):** Die Makro Fehlerrate berechnet den durchschnittlichen Fehler pro Klasse.

$$\text{Fehlerrate}_M = \frac{\sum_{i=1}^l \frac{fp_i + fn_i}{m_i}}{l} \quad (5.9)$$

- **Makro Relevanz (Macro Average Precision):** Die Makro Relevanz berechnet die durchschnittliche Übereinstimmung pro Klasse der Labels mit den positiv klassifizierten Instanzen.

$$\text{Relevanz}_M = \frac{\sum_{i=1}^l \frac{rp_i}{rp_i + fp_i}}{l} \quad (5.10)$$

- **Makro Sensitivität (Macro Average Recall):** Die Makro Sensitivität berechnet die durchschnittliche Effektivität pro Klasse des Klassifikators auf positiven Labels.

$$\text{Sensitivität}_M = \frac{\sum_{i=1}^l \frac{rp_i}{rp_i + fn_i}}{l} \quad (5.11)$$

- **Makro F-Score (Macro Average F-Score):** Der Makro F-Score kombiniert wie analog für den binären Fall beschrieben die beiden Maße Makro Relevanz und Makro Sensitivität.

$$\text{F-Score}_M = \frac{(1 + \beta^2) \text{Relevanz}_M \text{Sensitivität}_M}{\beta^2 \text{Relevanz}_M + \text{Sensitivität}_M} \quad (5.12)$$

- **Mikro Relevanz (Micro Average Precision):** Die Mikro Relevanz berechnet die Übereinstimmung der Klassen des Labels mit den positiven Instanzen für alle Klassen zusammen (indirekte Gewichtung der Klassen).

$$\text{Relevanz}_\mu = \frac{\sum_{i=1}^l rp_i}{\sum_{i=1}^l (rp_i + fp_i)} \quad (5.13)$$

- **Mikro Sensitivität (Micro Average Recall):** Die Mikro Sensitivität berechnet die Effektivität des Klassifikators auf positiven Labels für alle Klassen zusammen (indirekte Gewichtung der Klassen).

$$\text{Sensitivität}_\mu = \frac{\sum_{i=1}^l rp_i}{\sum_{i=1}^l (rp_i + fn_i)} \quad (5.14)$$

- **Mikro F-Score (Micro Average F-Score):** Der Mikro F-Score kombiniert wie analog für den binären Fall beschrieben die beiden Maße Mikro Relevanz und Mikro Sensitivität (bezieht also ebenfalls indirekt die Gewichtung der Klassen mit ein).

$$\text{F-Score}_\mu = \frac{(1 + \beta^2) \text{Relevanz}_\mu \text{Sensitivität}_\mu}{\beta^2 \text{Relevanz}_\mu + \text{Sensitivität}_\mu} \quad (5.15)$$

Zuletzt betrachten wir noch eine *probabilistische* Metrik. Diese Metrik berechnet nicht die endgültigen Entscheidungen des Klassifikators, sondern die zugrunde liegenden Wahrscheinlichkeiten, die der Klassifikator jeder Klasse zuordnet.

Diese Wahrscheinlichkeiten, stehen für jede der insgesamt m Testinstanzen und für jede der l Klassen in einer $m \times l$ Matrix. In einer Zelle an der Stelle i, k steht also die Wahrscheinlichkeit $p_{i,k}$ die der Klassifikator der Testinstanz i für die Klasse k zuordnet. In einer separaten Matrix stehen die tatsächlichen Ergebnisse im *One-Hot-Encoding* Format, hier hat eine Zelle $y_{i,k}$ den Wert 0, falls es sich nicht um diese Klasse handelt und 1, falls es die korrekte Klasse ist.

- **Log Loss:** Der Log Loss (oder auch Logistic Loss bzw. Cross Entropy Loss) berechnet den durchschnittlichen logistischen Verlust bezogen auf die den Klassen zugeordneten Wahrscheinlichkeiten des Klassifikators, also inwiefern die Wahrscheinlichkeiten von 0 (perfekter Wert für alle Klassen die nicht

dem Label entsprechen) bzw. 1 (perfekter Wert für die Klasse, die dem Label entspricht) abweichen.

$$\text{Log Loss} = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^l y_{i,k} \log(p_{i,k}) \quad (5.16)$$

5.1.3 Receiver Operating Characteristics (ROC)

Eine Möglichkeit den Klassifikator für verschiedene Schwellen (Thresholds) zu evaluieren bieten die sogenannten *Receiver Operating Characteristics* (kurz ROC). Zur Erinnerung, die Schwelle einer Klasse bestimmt, ab welcher Wahrscheinlichkeit eine Instanz als diese klassifiziert wird. Hierfür werden erneut die Wahrscheinlichkeiten für die Zugehörigkeit der jeweiligen Klassen auf einem Testset berechnet, dann wird für die jeweiligen Schwellen der Wert der Relevanz mit dem der Sensitivität grafisch in einer *ROC Kurve* gegenübergestellt.

In Abbildung 5.1 findet sich eine exemplarische ROC Kurve für einen binären Klassifikator. Einen Punkt mit einer True Positive Rate (Relevanz) von 1 und einer False Positive Rate (Sensitivität) von 0 würde man als eine Schwelle mit perfektem Klassifikationsergebnis interpretieren.

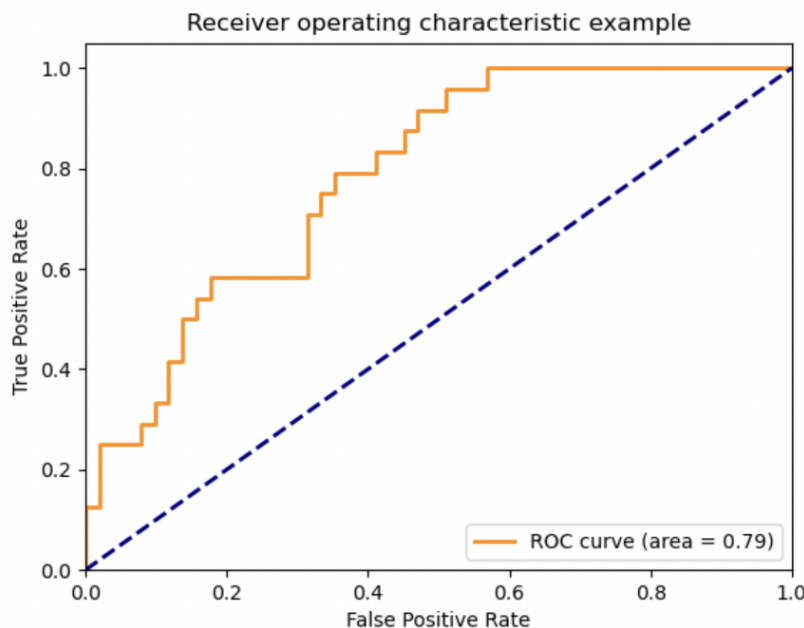


Abbildung 5.1: Beispiel einer ROC Kurve, Quelle: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html?highlight=roc%20curve, Stand: 19.09.2021

Eine höhere Schwelle geht generell mit einer niedrigeren FPR, allerdings auch mit einer niedrigeren TPR einher. Analog steigt die TPR, aber auch die FPR, wenn die Schwelle niedriger gewählt wird, wie im Graph zu sehen ist (hier werden keine konkreten Werte der Schwellen dargestellt, diese können aber bei Bedarf von der Scikit Learn Funktion `roc_curve`¹ zurückgegeben werden).

Schließlich kann für jeden Punkt bzw. direkt für die komplette Kurve die Area Under Curve berechnet werden (siehe voriger Abschnitt). Die gestrichelte Linie ist als Orientierung zusehen. Sollte der Klassifikator diese Linie unterschreiten, ist der Klassifikator nicht besser als eine zufällige Entscheidung bei der Klassenwahl (vgl. [Faw06], S.863).

Nach dem üblichen Verfahren sollte der Klassifikator sich für diejenige Klasse mit der höchsten Wahrscheinlichkeit entscheiden. Im binären Fall wäre das genau die Klasse mit einer Wahrscheinlichkeit größer 50%, die Schwelle liegt also bei 0.5. Dieses Vorgehen führt in den meisten Fällen zu den besten Ergebnissen, da es natürlich logisch ist die Klasse mit der höchsten Wahrscheinlichkeit zu wählen.

Manchmal lassen sich allerdings bessere Ergebnisse mit anderen Schwellen erzielen, besonders wenn gefordert wird, dass ein Klassifikator bestimmte Klassen besser erkennt, weil diese eine höhere Priorität haben. Hier wird auch von *ungleichen Fehlerkosten* (Unequal Classification Errors Cost) gesprochen (vgl. [Faw06], S.861). Nützlich ist diese Art der Analyse auch bei ungleich verteilten Klassen, da diese direkt im Graph repräsentiert werden. Mit der Betrachtung einer ROC Kurve können also die TPR/FPR Verschiebungen der Klassen analysiert und somit eine optimale Schwelle entsprechend der eigenen Anforderungen an den Klassifikator gewählt werden.

Dieses Modell ist auch auf den multinomialen Fall erweiterbar, indem für jede Klasse eine Kurve gezeichnet wird, wobei nach dem *One vs. Rest* Prinzip die Wahrscheinlichkeit der Klasse der Summe aller Wahrscheinlichkeiten der anderen Klassen gegenüber gestellt wird (eine Implementierung hierfür findet sich auch in Kapitel 6).

5.2 Einfluss von Eigenschaften der Datensätze

Dieser Abschnitt geht auf die verschiedenen Charakteristika von Datensätzen ein und welche Auswirkungen diese auf die Performance des Klassifikators und die Aussagekraft und Wahl der Metriken haben.

¹https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve

Die *Anzahl der Datenpunkte* des Datensatzes kann einen großen Einfluss auf die spätere Performance des Klassifikators haben. Bis zu einem gewissen Punkt steigt mit der Größe des Datensatzes auch die Performance des Klassifikators, weil dadurch häufig Overfitting reduziert werden kann. Eine höhere Anzahl an Datenpunkten liefert meist mehr Informationen über die realen Daten, womit Muster besser erkannt werden können (vgl. [AAAB⁺21] und [RJ⁺91]). Zudem steigt mit einer größeren Anzahl an Testinstanzen auch die Aussagekraft der Ergebnisse und die statistische Signifikanz.

Ein weiterer zu betrachtender Faktor ist die *Klassenverteilung* innerhalb des Datensatzes. Bei einer stark ungleichmäßigen Klassenverteilung kann die Genauigkeit des Klassifikators auf kleineren Klassen gegenüber den größeren Klassen sinken, weil, wie im vorigen Punkt beschrieben, die Performance oft bis zu einem gewissen Grad mit einer höheren Anzahl an Datenpunkten steigt. Ferner ist es hier sinnvoll, eine andere Metrik als die Genauigkeit zu verwenden, da bei dieser nicht die individuelle Klassenverteilung mit einbezogen wird und Klassen mit einer kleineren Anzahl an Datenpunkten kaum eine Gewichtung haben (vgl. [Mos13], S.30).

In [FHOM09], S.18-20 konnte gezeigt werden, dass unter anderem die Performance gemessen an den qualitativen Metriken Genauigkeit, Makro Genauigkeit und Makro F-Score der Klassifikatoren sehr ähnlich sind für Datensätze mit gleich verteilten Klassen, sich aber stark unterscheiden können für ungleich verteilte Klassen. Es bietet sich also an mehrere dieser Metriken zu betrachten. So könnte zum Beispiel ein Klassifikator für einen Datensatz mit 100 Instanzen, von denen 98 positiv und 2 negativ sind, die einfache Regel verwenden, jede Instanz als positiv zu klassifizieren und würde dabei, obwohl er alle negativen Instanzen falsch klassifiziert, eine Genauigkeit von 98% erhalten.

Die *Qualität* eines Datensatzes, auf welche schon in Abschnitt 2.4.1 eingegangen worden ist, kann sich auch die Performance des Klassifikators auf späteren realen Daten auswirken. Mögliche Fehlertypen verringern die nutzbaren Instanzen des Datensatzes oder im Falle einer Verwendung nach der Datenaufbereitung die Aussagekraft, da diese manipuliert worden sind und nicht aus der ursprünglichen Datenerhebung stammen. Die Qualität eines Datensatzes kann allerdings auch durch eine nicht repräsentative Auswahl der Instanzen gemindert werden (vgl. [Bra07], S.232). Sollten die gewählten Instanzen nicht ein geeignetes Abbild der später zu klassifizierenden Daten widerspiegeln, kann auch dadurch die spätere Performance verschlechtert werden.

Eine weitere Eigenschaft ist die *Trennschärfe* der Klassen. Sollten Klassen ähnliche Werte für ihre Attribute besitzen, so überschneiden sich diese. Durch zunehmende Überschneidungen der Attributwerte wird eine Unterscheidung der Klassen

erschwert (vgl. [FGG⁺18], S.262-263, [SMGC14]), da sehr ähnliche Instanzen mitunter verschiedenen Klassen angehören können. Der Klassifikator muss somit mehr Information gewinnen, um eine eindeutige Entscheidung zu treffen.

5.3 Auswahl geeigneter Metriken

Die Auswahl geeigneter Metriken soll dabei helfen, eine fundierte Entscheidung zu treffen, ob der Klassifikator den Anforderungen der späteren Anwendung genügt. Die gewählten Metriken sollen dem Nutzer einen Überblick über die Leistung des Klassifikators geben. In diesem Abschnitt werden noch einmal die Metriken aus Abschnitt 5.1.2 aufgegriffen und es wird erklärt, welche Metriken für welche Szenarien, auch bezogen auf die Eigenschaften des Datensatzes, verwendet werden sollten.

Allgemein sind die Anforderungen sehr anwendungsspezifisch und können für verschiedene Szenarien variieren. So könnte eine Anforderung an einen Klassifikator für den in Abschnitt 2.1 vorgestellten Diabetes Datensatz zum Beispiel sein, dass dieser wenige positive Instanzen fälschlicherweise als negativ klassifiziert, hier wäre also die Sensitivität eine äußerst relevante Metrik.

Zunächst sollte also spezifiziert werden, welche Wichtigkeit verschiedene Klassen haben und welche Arten von Fehlern relevant sind. Entsprechend kann dann die Performance auf den gewählten Klassen durch die Metriken Relevanz, Sensitivität und Spezifität genauer untersucht werden. Eine Kombination aus diesen bildet wie schon erwähnt der F-Score, diesem kann, falls gewünscht, durch den Parameter β der Relevanz bzw. Sensitivität eine besondere Gewichtung gegeben werden. Alternativ kann auch die AUC verwendet werden.

Im Mehrklassenfall muss außerdem zwischen den Varianten Makro und Mikro Averaging unterschieden werden. Makro Averaging sollte verwendet werden, wenn jeder Klasse die gleiche Priorität, unabhängig von Klassenverteilung gegeben wird. Mikro Averaging hingegen sollte verwendet werden, wenn die Klassenverteilung mit in die Metrik einbezogen werden soll.

Wie schon im vorigen Abschnitt beschrieben, sollte unbedingt im Fall der ungleichen Klassenverteilungen darauf geachtet werden, nicht die Genauigkeit als alleiniges Maß der Performance zu betrachten, sondern auch Metriken, welche ungleiche Klassenverteilungen in ihre Berechnung einbeziehen. Des Weiteren sollte eine k-fache Kreuzvalidierung durchgeführt werden, um die Varianz der Ergebnisse zu beobachten und zu analysieren, welche Aussagekraft die Metriken haben. Dies gilt insbesondere für Datensätzen mit wenigen Datenpunkten.

Implementierung des Evaluationsrahmenwerkes

In diesem Kapitel soll die konkrete Implementierung des webbasierten Evaluationsrahmenwerkes, welche auf den vorherigen theoretischen Kapiteln aufbaut, erläutert werden. Das komplette Projekt ist über Github¹ erhältlich und kann wie dort in den Readme Dateien beschrieben installiert und getestet werden.

6.1 Anforderungen

Bevor wir die Implementierung betrachten, sollen zunächst einige Anforderungen an das Evaluationsrahmenwerk spezifiziert werden. Diese umfassen unter anderem:

- **Auswertung von Metriken:** Die in Abschnitt 5.1.1 definierten Metriken sollen zur Evaluation von Ergebnissen eines Klassifikators zur Verfügung stehen.
- **ROC Analyse:** Die in Abschnitt 5.1.2 beschriebene ROC Analyse soll verwendet werden können, um die probabilistischen Ergebnisse eines Klassifikators für verschiedene Schwellen zu vergleichen und so eine optimale Schwelle zu finden.
- **K-fold Cross Validation:** Die in Abschnitt 2.4.4 beschriebene k-fache Kreuzvalidierung soll genutzt werden können, um den Klassifikators auf verschiedenen Trainings- und Testdatensätze zu evaluieren und somit die Varianz der Ergebnisse zu analysieren.
- **Vergleich mit anderen Klassifikatoren:** Alle Klassifikatoren, die in Kapitel 4 erläutert worden sind, sollen die gleichen Trainings- sowie Testdaten wie der zu evaluierende Klassifikator erhalten und deren Ergebnisse zum Vergleich nutzbar sein.

¹<https://github.com/Christian-Gebhardt/classifier-evaluation-framework-webapp> und <https://github.com/Christian-Gebhardt/classifier-evaluation-framework-API>

6.2 Architektur und Kommunikation der Komponenten

Die Architektur der Anwendung, dargestellt in Abbildung 5.1, kann in zwei Komponenten unterteilt werden. Diese sind einmal die *Webapplikation*² die mit der JavaScript Bibliothek *React*³ entwickelt worden ist und zum anderen das *Application Programming Interface*⁴ (kurz API), das mit dem Python Webframework *Flask*⁵ erstellt worden ist.

Im Folgenden sollen diese Komponenten kurz erläutert werden, um ein Verständnis der Funktionsweise der Anwendung zu erhalten, ohne dass dabei zu tiefe Kenntnisse über die verwendeten Technologien nötig wären.

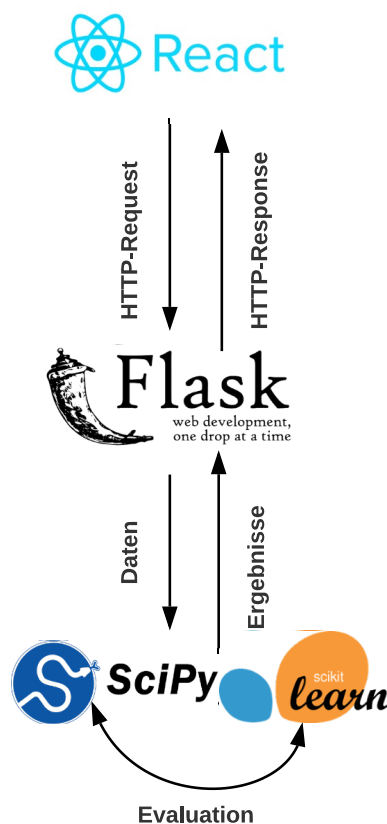


Abbildung 6.1: Architektur des Evaluationsrahmenwerkes

²<https://github.com/Christian-Gebhardt/classifier-evaluation-framework-webapp>

³<https://reactjs.org/>

⁴<https://github.com/Christian-Gebhardt/classifier-evaluation-framework-API>

⁵<https://flask.palletsprojects.com>

Die React-Webanwendung implementiert das User Interface und ist für die Eingabe der Daten und das Anzeigen der Evaluationsergebnisse zuständig. Diese kann in jedem aktuellen Internetbrowser⁶ angezeigt werden. Dafür sind mit Hilfe von React verschiedene HTML-Ansichten erzeugt worden, welche alle das Design Paket Material UI⁷ verwenden. Für Grafiken und Datendarstellungen ist zusätzlich das Paket Recharts⁸ verwendet worden. Die Ansichten sind über das React-Router Paket navigierbar.

Sobald der Nutzer alle nötigen Parameter zur Evaluation eingegeben hat und die Auswertung sehen möchte, wird mit der Java Script Bibliothek Axios⁹ eine entsprechende HTTP-Anfrage mit den eingegebenen Daten als JSON Request Body für die Auswertung an die Flask API gestellt.

Die Flask API wiederum verarbeitet die HTTP Anfrage, indem sie die gesendeten Eingabedaten zunächst empfängt und mithilfe der Python SciPy Bibliotheken NumPy¹⁰ und Pandas¹¹ in ein geeignetes Format für die Machine Learning Bibliothek Scikit Learn¹² bringt. Danach werden die benötigten Methoden von Scikit Learn aufgerufen, um alle nötigen Berechnungen für die Evaluation durchzuführen.

Die Ergebnisse wiederum werden in ein geeignetes JSON Format gebracht, welches mit der HTTP-Antwort an die Webapplikation zurückgesendet wird. Schließlich werden die empfangenen Resultate in der Auswertungsansicht der Webapplikation entsprechend grafisch dargestellt.

6.3 Funktionsweise

6.3.1 User Interface

Zunächst soll das User Interfaces beschrieben werden. Das User Interface besteht aus den verschiedenen Ansichten *Eingabe*, *Evaluation* und *Vergleich*. Hilfestellungen zur Nutzung finden sich in den Python Skripts, welche im Github Repository¹³ unter dem Ordner „scripts“ als py (Python) oder ipynb (Python Jupyter Notebook) verfügbar sind. Eine zusätzliche Erklärung der Skripte findet sich im nächsten Abschnitt 6.4.

⁶Möglicherweise unterstützen ältere Versionen des Internetexplorers nicht alle Funktionalitäten von React, siehe: <https://reactjs.org/docs/react-dom.html#browser-support>

⁷<https://material-ui.com/>

⁸<https://recharts.org/>

⁹<https://axios-http.com/>

¹⁰<https://numpy.org/>

¹¹<https://pandas.pydata.org/>

¹²<https://scikit-learn.org/>

¹³<https://github.com/Christian-Gebhardt/classifier-evaluation-framework-API>

Evaluation von Klassifikatoren

HomeProjekt

EINGABEEVALUATIONVERGLEICH

EIGENERVERGLEICH

Labelvektor (y_true) ⓘ

Bitte auswählen

Ergebnisvektor (y_pred) ⓘ

Bitte auswählen

Ausprägung des Zielattributes

☒ binär
 ☐ multinomial

Wird benötigt um geeignete Metriken zu wählen

Art der Metriken

☒ qualitativ
 ☐ probabilistisch

Wird benötigt um geeignete Metriken zu wählen

Abbildung 6.2: Ausschnitt aus dem User Interface der Webanwendung, Ansicht Eingabe/Eigener

Die Eingabe ermöglicht es dem Nutzer, die benötigten Parameter für die Evaluation des eigenen Klassifikators bzw. für den Vergleich mit vordefinierten Scikit Learn Klassifikatoren zu übergeben.

Mithilfe der Eingabemaske *Eigener* kann der Nutzer die Ergebnisse seines Klassifikators auf einem Testset evaluieren, ein Ausschnitt davon wird in Abbildung 6.2 gezeigt. Hier müssen der Labelvektor (y_{true}), also die wahren Werte der Labels, und der Ergebnisvektor (y_{pred}), die vorhergesagten Werte der Labels (bzw. die probabilistischen Werte als Matrix) als npy (Numpy Matrix Format) Dateien durch den Nutzer eingegeben werden (siehe auch die Beispiele im Python Skript *evaluation_own_classifier_format*).

Der Nutzer kann dann je nach Anzahl der Klassen die in 5.1.2 definierten binären bzw. multinomialen Metriken wählen. Ist die Anzahl der Klassen nicht bekannt, so kann immer die Checkbox „multinomial gewählt werden“, da diese auch im binären Fall berechnet werden können, während binäre Metriken im Mehrklassen Fall nicht berechnet werden können. Falls probabilistische Werte eingegeben werden, muss zusätzlich die Art der Metriken von „qualitativ“ auf „probabilistisch“ gesetzt werden. Dann ist es möglich, die probabilistische Metrik Log Loss zu berechnen, sowie einen ROC Kurvenplot erstellen zu lassen.

Sobald alle Eingaben erfolgt sind, kann mit dem „Senden“-Button eine HTTP Anfrage mit den entsprechenden Parametern an die Flask API gestellt werden. Die Antwort mit den Ergebnissen wird dann entsprechend unter dem Tab *Evaluation* dargestellt. Das komplette Eingabeverfahren wird noch einmal in dem Aktivitätsdiagramm in Abbildung 6.3 zusammengefasst.

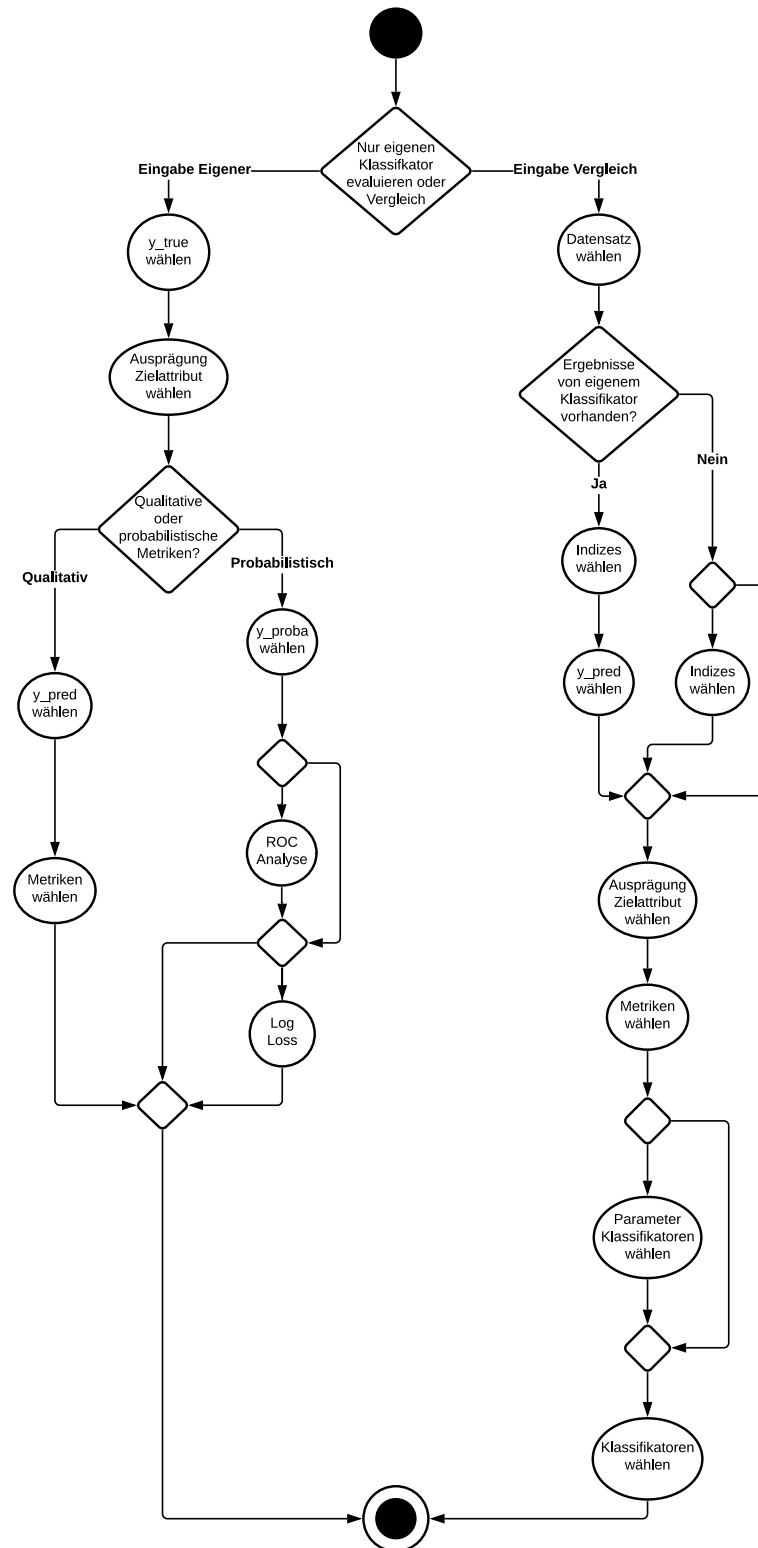


Abbildung 6.3: Aktivitätsdiagramm für die Eingabe innerhalb der Webanwendung

Die zweite Eingabemaske *Vergleich* ermöglicht es dem Nutzer, seinen eigenen Klassifikator auf mehreren Testsets mit den in Kapitel 4 beschriebenen Klassifikatoren, implementiert durch die Machine Learning Bibliothek *Scikit Learn*, zu vergleichen. Ein Ausschnitt der Eingabemaske findet sich in Abbildung 6.4. Alternativ können auch nur die gewählten Scikit Learn Klassifikatoren für verschiedene Parameter auf einem gegebenen Datensatz evaluiert werden, ohne dass vorher ein eigenes Modell erstellt, trainiert und getestet werden müsste.

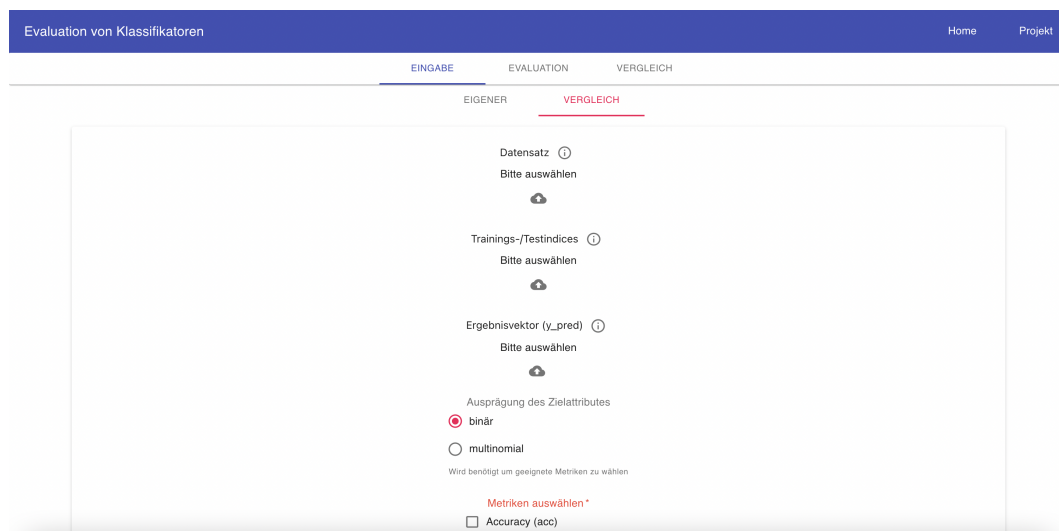


Abbildung 6.4: Ausschnitt aus dem User Interface der Webanwendung, Ansicht Eingabe/-Vergleich

Der Nutzer kann in der Maske einen gewählten Datensatz in npy oder csv Format (siehe Beispiele im Python Skript *dataset_format*) hochladen. Dieser Datensatz ist für beide Optionen obligatorisch. Will er einen eigenen Klassifikator evaluieren, so kann er die verwendeten Trainings- und Testindizes als npz (Numpy Zipped Format) und die vorhergesagten Werte des Klassifikators auf den Testsets als npy Datei hochladen (siehe Beispiele im Python Skript *generating_kxn_cross_validation*).

Die folgende Auswahl der Metriken erfolgt analog zur Ansicht der ersten Eingabemaske, während hier allerdings nur qualitative Metriken zur Auswahl stehen. Zuletzt können die gewünschten Vergleichsklassifikatoren gewählt werden, welchen individuelle Parameter über den Button mit dem Label „Parameter“ gegeben werden können. Einige dieser Parameter sind bereits für die jeweiligen Klassifikatoren in Kapitel 4 erläutert worden. Für mehr Informationen siehe auch die entsprechenden Links der Dokumentation der verwendeten Scikit Learn Klassifikatoren zu Beginn von Kapitel 4.

Soll kein eigener Klassifikator in den Vergleich mit einbezogen werden, so müssen keine Ergebnisse und Indices hochgeladen werden. Hier genügt der entsprechen-

de Datensatz, auf dem die Klassifikatoren getestet werden sollen. Sollten Indices angegeben werden, so verwenden die Klassifikatoren diese Trainings- und Testsets, andernfalls wird wie in 2.4.4 beschrieben eine 5x2-fache Kreuzvalidierung durchgeführt.

6.3.2 Evaluation

Nachdem der Nutzer die Eingabe mit geeigneten Daten ausgefüllt hat, wird eine HTTP Anfrage, welche die eingegebenen Daten enthält, an die Flask API gestellt. Die Verarbeitung der Anfrage findet in *app.py* statt, hier wird wiederum eine geeignete Methode *evaluate_input*, falls nur der eigene Klassifikator evaluiert werden soll, bzw. *compare_clfs*, falls der Klassifikator mit Scikit Learn Klassifikatoren verglichen werden soll, aufgerufen. Diese beiden Methoden finden sich in der Datei *evaluation_service.py*, welche von *app.py* importiert wird. Die Implementierung, sowie eine kurze Dokumentation findet sich (wie bereits erwähnt) unter <https://github.com/Christian-Gebhardt/classifier-evaluation-framework-API>.

Sollte die Methode *evaluate_input* aufgerufen werden, so werden für einen gegebenen Ergebnisvektor *y_pred*, sowie einen gegebenen Labelvektor *y_true* alle ausgewählten Metriken mit der Methode *evaluate_metric* berechnet. Außerdem wird eine Konfusionsmatrix berechnet, sowie ein Klassifikationsreport, welcher noch einmal alle wichtigen Informationen zusammenfasst. Falls ein probabilistischer Ergebnisvektor *y_proba* verwendet wird, kann optional ein ROC Graph mit Hilfe der Methode *plot_multiclass_roc* berechnet werden. Die Ergebnisse werden dann an die Webapplikation zurückgeliefert und dort entsprechend Abbildung 6.5 bzw. 6.6 im Tab *Evaluation* visualisiert.

Falls hingegen die Methode *compare_clfs* aufgerufen wird, so werden alle Metriken wie zuvor für den eigenen Klassifikator berechnet, falls Daten für diesen eingegeben wurden, sowie alle ausgewählten Vergleichsklassifikatoren mit der Methode *map_classifier* für gegebene Parametereinstellungen initialisiert und auf den eingegebenen Trainingsindizes trainiert. Sind diese hingegen nicht eingegeben worden, werden die Indizes standardmäßig mit der Methode *generate_5x2cv* für eine 5x2-fache Kreuzvalidierung generiert.

Anschließend werden die Klassifikatoren auf den entsprechenden Testsets evaluiert und gewählte Metriken werden ausgewertet. Genau wie zuvor werden nun die Ergebnisse an die Webapplikation zurückgesendet und im Tab *Vergleich* entsprechend Abbildung 6.7 dargestellt. Hier ist die originale Version des bekannten Datensatzes zur Klassifikation einer Diabeteserkrankung gewählt worden.

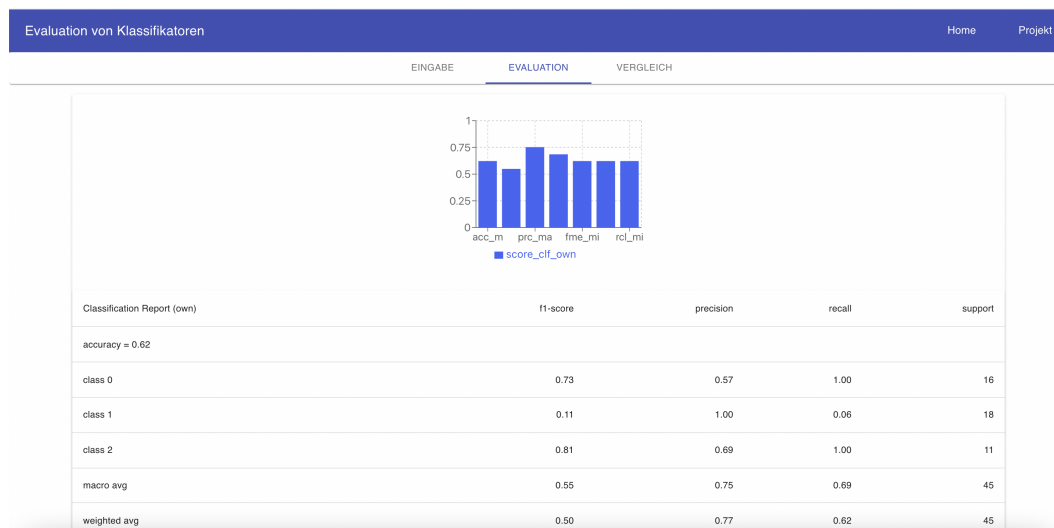


Abbildung 6.5: Ausschnitt aus dem User Interface der Webanwendung, Ansicht Evaluation (qualitativ)

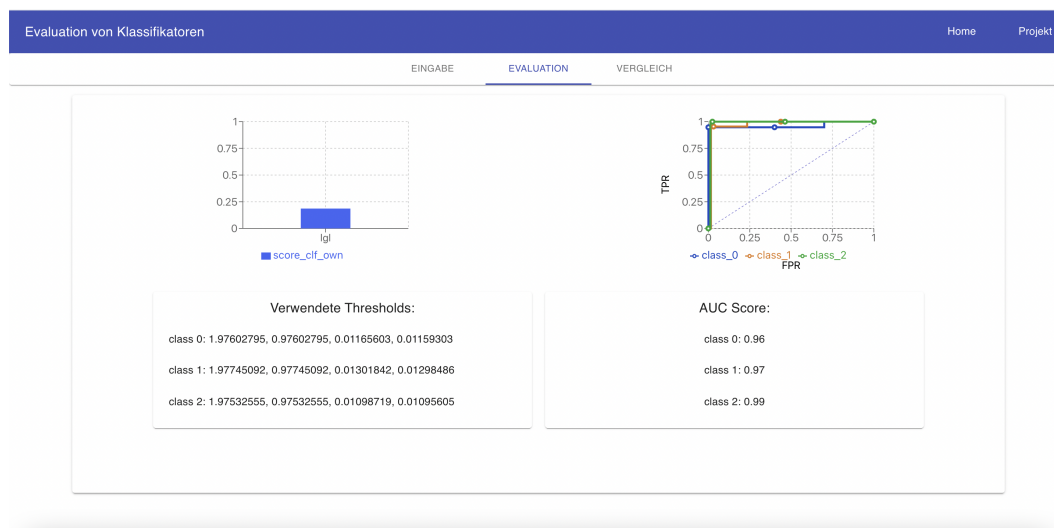


Abbildung 6.6: Ausschnitt aus dem User Interface der Webanwendung, Ansicht Evaluation (probabilistisch)



Abbildung 6.7: Ausschnitt aus dem User Interface der Webanwendung, Ansicht Vergleich, ausgewerteter Datensatz: <https://archive.ics.uci.edu/ml/datasets/diabetes>

6.4 Hilfsskripte und Anwendungsbeispiele

Die folgenden Python Skripte sollen anhand von Beispielen demonstrieren, wie die Anwendung genutzt werden kann. Hiermit soll dem Leser ein möglichst einfacher Einstieg in die Benutzung ermöglicht werden. Die Skripte finden sich im Github Repository unter dem Verzeichnis *Skripte*: (<https://github.com/Christian-Gebhardt/classifier-evaluation-framework-API/tree/main/scripts>). Mögliche Testdatensätze finden sich im *UCI Machine Learning Repository* (<https://archive.ics.uci.edu>), aus welchem ich selbst einige Datensätze zum Testen der Anwendung verwendet habe.

6.4.1 Evaluation eines eigenen Klassifikators

Das folgende Skript *evaluation_own_classifier*, dargestellt in Abbildung 6.8, zeigt, wie die Ergebnisse eines eigenen Klassifikators innerhalb der Anwendung verwendet werden können.

In Zelle [1] werden, wie auch bei den beiden anderen Skripten die benötigten Bibliotheken importiert. Zelle [2] verwendet direkt die Methode *load_iris*, welche den bereits bekannten Iris Datensatz als Beispiel zur Verfügung stellt (siehe https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html). Hier kann generell jeder Datensatz, der für eine in Kapitel 2 beschriebene Klassifikationsaufgabe geeignet ist und in einem für den Klassifikator geeignetem

```

[1]: from sklearn.datasets import load_iris
    from sklearn.linear_model import SGDClassifier
    from sklearn.calibration import CalibratedClassifierCV
    from sklearn.model_selection import train_test_split
    import numpy as np

[2]: """
    @author Christian Gebhardt
    @email christian.gebhardt@uni-bayreuth.de
    """
    # Script that shows how to get y_true (true label vector), y_pred (predicted label vector) or y_proba (probability matrix)
    # as input for evaluation framework

    # Example dataset, replace this with your feature data matrix X and target vector y.
    df = load_iris()
    X = df.data
    y = df.target

[3]: # Now split them into training and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True)

    # y_test is already the true label vector for later evaluation, so save y_test in .numpy format
    np.save("y_true_iris", y_test)

[4]: # Train your classifier with training data and predict test labels (this might be a different process for your own model/classifier
    # but in the end you should have a vector with predicted labels)

    clf = SGDClassifier()
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)

    # Save y_pred in .numpy format
    np.save("y_pred_iris", y_test)

[5]: # Finally if you want to analyze probabilistic values from the prediction of your classifier (or any sklearn classifier)
    # you need to have a nxc matrix, where n is the number of instances and c is the number of classes. So in every row there should
    # be all probabilities for each class. With most sklearn classifiers you can do the following:

    # This works for the SGD classifier we defined previously
    calibrator = CalibratedClassifierCV(clf, cv='prefit')
    model = calibrator.fit(X_train, y_train)
    y_proba = model.predict_proba(X_test)

    # print("Probabilistic matrix:\n{}".format(y_proba_iris)) # uncomment this line to check probabilistic matrix format

    # save y_proba in .numpy format
    np.save("y_proba_iris", y_proba)

```

Abbildung 6.8: Darstellung des Python Skriptes `evaluation_own_classifier` in Jupyter Lab

Format vorliegt, verwendet werden. Die Feature Matrix wird hier der Variable `X` und der Labelvektor der Variable `y` zugewiesen.

Zelle [3] teilt den Datensatz in Trainings- und Testdaten mit Hilfe der Methode `train_test_split` (siehe https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html). Anschließend kann bereits die Variable `y_test` im `numpy` Format gespeichert werden, da diese die korrekten Labels der späteren Testdaten enthält.

Anschließend wird in Zelle [4] eine Scikit Learn Implementierung des Stochastic Gradient Descent Klassifikators mit `SGDClassifier` initialisiert und mit der Methode `fit` trainiert (siehe https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html). Mit der Methode `predict` werden dann die Labels der Testdaten von dem bereits trainierten Klassifikator vorhergesagt und im `numpy` Format gespeichert. Je nach Implementierung des eigenen Klassifikators bzw. einer anderen Machine Learning Bibliothek als Scikit Learn können sich das Training und das Testen eines Modells unterscheiden. Es ist nur wichtig, dass die vorhergesagten Werte später als diskrete numerische Werte für die jeweiligen Klassen in einem Vektor vorliegen.

Zelle [5] beschreibt das Vorgehen für vorhergesagte probabilistische Werte im Zielvektor. Hier stellt Scikit Learn die Klasse `CalibratedClassifierCV` zur Verfügung,

welche mit der Methode `predict_proba` probabilistische Werte für die Wahrscheinlichkeiten der Klassen zurückliefert (siehe <http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html>).

6.4.2 Datensatz Format

Das nächste Skript `dataset_format` in Abbildung 6.9 zeigt, wie eigene Datensätze oder auch Datensätze aus einer Webressource in ein geeignetes Format für die Anwendung gebracht werden können. Die verwendeten Numpy und Pandas Methoden sind in den jeweiligen Dokumentationen zu finden: <https://numpy.org/doc/stable/> und <https://pandas.pydata.org/docs/>.

```
[1]: from sklearn.datasets import load_iris
import numpy as np
import pandas as pd

[2]: """
@author Christian Gebhardt
@email christian.gebhardt@uni-bayreuth.de
"""
# Script that shows how to bring datasets in a supported format.
# If you have data (X, y) already just use .npy format
df = load_iris()
X = df.data
y = df.target
print("Shapes of X and y before reshaping: {}".format(X.shape, y.shape))
Shapes of X and y before reshaping: (150, 4), (150,)

[3]: # bring data in supported shapes and stack X (nxm) and y (nx1) together in one matrix (nxm+1), with last column (!) as target column
y = y.reshape((150, 1))
dataset = np.hstack((X, y))
# print("Final dataset:\n {}".format(dataset))
# saving dataset as .npy (replace filename as desired)
np.save("dataset_iris", dataset)

[4]: # If you have a csv just send it as csv (first row must be header, target variable in last column, separator must be ','), using a example
# from Github here you can pass any url with csv content here
dataset = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/diabetes.csv")
# dataset.head() # uncomment this line to see dataset
# saving file as csv (make sure to use .csv suffix)
dataset.to_csv("dataset_diabetes.csv", sep=',', encoding='utf-8')

[5]: # If you are using classification datasets from the machine learning data repository (or any other repository) at
# https://archive.ics.uci.edu/ml/index.php or in general .data files you can also fetch them directly with pandas
# (but you need to know the separator), often they don't have a header, so make sure to use header=None
dataset = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data", sep=',', header=None)
# dataset.head() # uncomment this line to see dataset
# converting csv file to numpy (since this dataset has no header line, alternatively you can add a fake header row)
dataset = dataset.to_numpy()
# saving dataset as .npy (target column is already last column, if not change format)
np.save("dataset_wine", dataset)
# Finally if the dataset contains categorical variables as strings or missing values, you need to convert them in a sklearn supported
# format see https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder,
# https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html
# and https://scikit-learn.org/stable/modules/impute.html for more information
```

Abbildung 6.9: Darstellung des Python Skriptes `dataset_format` in Jupyter Lab

Zu Beginn wird in Zelle [2] erneut der Iris Datensatz als Beispiel geladen und die Dimensionen der Matrix X und dem Vektor y angezeigt. Es fällt auf, dass der Vektor y keine zweite Dimension hat. Deshalb wird dieser in Zelle [3] mit der Numpy Methode `reshape` in eine geeignete Form gebracht. Anschließend werden X und y *horizontal* zu einer einzigen Matrix zusammengeführt und als npy gespeichert. Hier ist zu beachten, dass der Labelvektor y die letzte Spalte der Matrix einnehmen muss, um später von der API verwendet werden zu können.

Die nachfolgende Zelle [4] zeigt, wie csv Dateien über eine URL geladen werden und später verwendet werden können. Hier ist zu beachten, dass Dateien im CSV Format eine Kopfzeile enthalten müssen. In Zelle [5] wird eine Datei im *data* Format geladen, welche keine Kopfzeile enthält. In diesem Fall kann die CSV Datei über die Methode *to_numpy* in ein *numpy array* umgewandelt werden und als *npz* gespeichert werden.

In beiden Fällen muss allerdings erneut darauf geachtet werden, dass die Labels die letzte Spalte einnehmen. Andernfalls muss der Nutzer die Daten entsprechend umformen (z.B. indem Spalten mittels Array Slicing getauscht werden). Zudem sollten unvorbereitete Datensätze entsprechend der in Kapitel 2.4 besprochenen Methoden aufbereitet werden und kategoriale Werte, die als Zeichenketten vorliegen, durch eine numerische Kodierung wie z.B. die One-Hot-Kodierung ersetzt werden.

6.4.3 Generierung von Trainings- und Testindizes

Das in Abbildung 6.10 dargestellte Skript *generating_kxn_cross_validation*, soll abschließend zeigen, wie Trainings und Testindizes für eine Kreuzvalidierung generiert werden können, um diese sowohl für den eigenen Klassifikator als auch als Eingabe für Vergleichsklassifikatoren zu verwenden.

Zunächst wird in Zelle [2] eine eigene Methode *generate_kxn_cv* definiert, welche Trainings- und Testindizes gemäß einer $k \times n$ Kreuzvalidierung aufteilt und zurückgibt. Hierbei wird die Klasse *StratifiedKFold* verwendet, welche die Indizes gemäß der Klassenverteilung des Datensatzes bestimmt (siehe https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html). Sollen Indizes nur nach einer k -fachen Kreuzvalidierung bestimmt werden, so kann der Parameter n gleich 1 gewählt werden. In Zelle [3] wird die Methode mit Beispielparametern aufgerufen und das Ergebnis im *npz* Format gespeichert.

Schließlich wird in Zelle [4] gezeigt, wie ein Klassifikator auf diese Indizes trainiert und getestet werden könnte und wie die Ergebnisse gespeichert werden müssen. Hierfür wird über das $k*n$ lange Array der Indizes iteriert und dabei der Klassifikator mit den entsprechenden Indizes trainiert und getestet. Die Ergebnisse werden in einer neuen Liste *y_pred* gesammelt, anschließend in ein Numpy Array umgewandelt und als *npz* Datei gespeichert.

```
[1]: from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import SGDClassifier
import numpy as np

[2]: """
@author Christian Gebhardt
@email christian.gebhardt@uni-bayreuth.de
"""
# Script that shows how to generate cross validation indices for own classifiers and .npz file for comparison classifiers
# and train/test own classifier with them

# Method to generate k x n cross validation trainings- and test indices.
def generate_kxn_cv(X, y, k, n):
    train_indices = []
    test_indices = []

    assert n > 0, "n must be greater than 0"
    assert k > 0, "k must be greater than 0"

    for i in range(k):
        skf = StratifiedKFold(n_splits=n, shuffle=True)
        skf.get_n_splits(X, y)
        train_index, test_index = skf.split(X, y)
        for j in range(n):
            train_indices.append(train_index[j])
            test_indices.append(test_index[j])

    return (train_indices, test_indices)

[3]: # Example dataset, replace this with your feature data matrix X and target vector y.
df = load_iris()
X = df.data
y = df.target

# Example indices with k=5 and n=2, replace this with your desired k and n.
k = 5
n = 2
train_indices, test_indices = generate_kxn_cv(X, y, k, n)

# print("train_indices before saving:\n{}".format(train_indices)) # uncomment this line to check train indices
# print("test_indices before saving:\n{}".format(test_indices)) # uncomment this line to check test indices

# Saving train_indices and test_indices to a .npz file for input in evaluation framework. You can use them now to train
# and test your classifier, remember to use this file as later input for comparison classifiers. You can replace file with
# your desired filename (or output stream) but the keys 'train_indices' and 'test_indices' should stay the same.
np.savez(file="train_test_indices", train_indices=train_indices, test_indices=test_indices)

[4]: # Loading train_indices and test_indices for possible later use.
indices = np.load(file="train_test_indices.npz")

# Access with keys 'train_indices' and 'test_indices'.
train_indices = indices['train_indices']
test_indices = indices['test_indices']

# print("train_indices after loading:\n{}".format(train_indices)) # uncomment this line to check train indices
# print("test_indices after loading:\n{}".format(test_indices)) # uncomment this line to check test indices

# Use indices like this for training and testing with own classifier (example with sklearn classifier)
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=10000)

y_pred = []
for i in range(len(train_indices)):
    clf.fit(X[train_indices[i]], y[train_indices[i]])
    y_pred.append(clf.predict(X[test_indices[i]]))

y_pred = np.asarray(y_pred)

# print("Results CV before saving:\n{}".format(y_pred)) # uncomment this line to check results y_pred

# Save them as .npy file for later input in evaluation framework, replace with desired filename.
np.save("y_pred_own_cv", y_pred)

# Loading npy file for possible later use.
y_pred = np.load("y_pred_own_cv.npy")

# print("Results CV after loading:\n{}".format(y_pred)) # uncomment this line to check results y_pred
```

Abbildung 6.10: Darstellung des Python Skriptes `generating_kxn_cross_validation` in Jupyter Lab

Fazit

In der Arbeit ist der Leser zunächst an das Thema Machine Learning, insbesondere Klassifikation herangeführt worden. Anschließend sind einige Klassifikatoren erläutert worden, deren Implementierungen später als Vergleich bei der Evaluation des eigenen Klassifikators dienen sollen. Die in Kapitel 5 definierten Metriken und Analysemethoden haben dem Leser umfangreiche Werkzeuge geliefert, um möglichst viele relevante Informationen aus der Evaluation zu gewinnen. Es ist erläutert worden, inwiefern Eigenschaften von Datensätzen einen Einfluss auf das Evaluationsergebnis haben können und wann der Nutzer bestimmte Metriken und Methoden nutzen sollte.

Schließlich ist in Kapitel 6 eine konkrete Implementierung eines Webframeworkes zur Evaluation von Klassifikatoren aufbauend auf der vorherigen Theorie geliefert worden. Die Anwendung ermöglicht es, Ergebnisse eines eigenen Klassifikators auf verschiedenen qualitativen und einer probabilistischen Metrik zu evaluieren, sowie eine ROC Analyse durchzuführen, um das Verhalten für verschiedene Schwellen zu beobachten. Sie liefert dem Nutzer ein einfach bedienbares Interface und stellt die Ergebnisse grafisch dar. Des Weiteren kann der Nutzer eigene Klassifikatoren mit den in Kapitel 4 beschriebenen Klassifikatoren vergleichen. Auch ohne eigenen Klassifikator und Ergebnisse kann er die von Scikit Learn implementierten Klassifikatoren auf unterschiedlichen Datensätzen eines Klassifikationsproblems testen.

Insgesamt sind die in 6.1 spezifizierten Anforderungen erfüllt worden, allerdings könnte das User Interface ausführlicher ausgestaltet werden, indem unter anderem mehr Möglichkeiten der Datenvisualisierung gegeben oder auch mehr zusätzliche Funktionalitäten implementiert bzw. bestehende verbessert werden. Da z.B. bereits ein Kreuzvalidierungsverfahren verwendet wird, könnte der Nutzer weiterführend auch selbst statistische Tests auf den erhaltenen Ergebnissen durchführen um Varianz und Aussagekraft dieser zu bestimmen. Daraus könnte direkt abgeleitet werden, für welchen Datensatz, welcher Klassifikator am besten geeignet ist. Diese Funktionalität könnte zukünftig auch automatisiert und direkt durch die Anwendung ausgeführt werden. Des Weiteren könnte die Auswahl an Vergleichsklassifikatoren erhöht oder standardisierte Verfahren zur Datenaufbereitung hinzugefügt werden.

Literaturverzeichnis

- [AAAB⁺21] ALTHNIAN, Alhanoof ; ALSAEED, Duaa ; AL-BAITY, Heyam ; SAMHA, Amani ; DRIS, Alanoud B. ; ALZAKARI, Najla ; ABOU ELWAFa, Afnan ; KURDI, Heba: Impact of Dataset Size on Classification Performance: An Empirical Evaluation in the Medical Domain. In: *Applied Sciences* 11 (2021), Nr. 2, S. 796
- [Bra07] BRAMER, Max: *Principles of data mining*. Bd. 180. Springer, 2007
- [CL16] CLEVE, Jürgen ; LÄMMEL, Uwe: *Data Mining*. 2. Berlin and Boston : De Gruyter Oldenbourg, 2016 (De Gruyter Studium). <http://dx.doi.org/10.1515/9783110456776>. <http://dx.doi.org/10.1515/9783110456776>. – ISBN 978–3–11–045675–2
- [Die98] DIETTERICH, Thomas G.: Approximate statistical tests for comparing supervised classification learning algorithms. In: *Neural computation* 10 (1998), Nr. 7, S. 1895–1923
- [Faw06] FAWCETT, Tom: An introduction to ROC analysis. In: *Pattern recognition letters* 27 (2006), Nr. 8, S. 861–874
- [FGG⁺18] FERNÁNDEZ, Alberto ; GARCÍA, Salvador ; GALAR, Mikel ; PRATI, Ronaldo C. ; KRAWCZYK, Bartosz ; HERRERA, Francisco: *Learning from imbalanced data sets*. Bd. 10. Springer, 2018
- [FHOM09] FERRI, César ; HERNÁNDEZ-ORALLO, José ; MODROIU, R: An experimental comparison of performance measures for classification. In: *Pattern Recognition Letters* 30 (2009), Nr. 1, S. 27–38
- [Gé17] GÉRON, Aurélien: *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow*. O'Reilly Verlag, 2017 http://www.content-select.com/index.php?id=bib_view&ean=9783960101147. – ISBN 9783960101147
- [JWHT14] JAMES, Gareth ; WITTEN, Daniela ; HASTIE, Trevor ; TIBSHIRANI, Robert: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. – ISBN 1461471370

-
- [Kub15] KUBAT, Miroslav: *An Introduction to Machine Learning*. 1st. Springer Publishing Company, Incorporated, 2015. – ISBN 3319200097
- [LCMH19] LUQUE, Amalia ; CARRASCO, Alejandro ; MARTÍN, Alejandro ; HERAS, Ana de l.: The impact of class imbalance in classification performance metrics based on the binary confusion matrix. In: *Pattern Recognition* 91 (2019), S. 216–231
- [Liu21] LIU, Shanhong: *Artificial intelligence software market revenue worldwide 2018-2025*. <https://www.statista.com/statistics/607716/worldwide-artificial-intelligence-market-revenues>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [Mos13] MOSLEY, Lawrence: A balanced approach to the multi-class imbalance problem. (2013). <https://lib.dr.iastate.edu/etd/13537/>
- [Nie18] NIELSEN, Michael A.: *Neural Networks and Deep Learning*. Version: 2018. <http://neuralnetworksanddeeplearning.com/> Zuletzt aufgerufen am 19.09.2021
- [OD08] OLSON, D.L. ; DELEN, D.: *Advanced Data Mining Techniques*. Springer Berlin Heidelberg, 2008 <https://books.google.de/books?id=2vb-LZEn8uUC>. – ISBN 9783540769170
- [PVG⁺11] PEDREGOSA, F. ; VAROQUAUX, G. ; GRAMFORT, A. ; MICHEL, V. ; THIRION, B. ; GRISEL, O. ; BLONDEL, M. ; PRETTENHOFER, P. ; WEISS, R. ; DUBOURG, V. ; VANDERPLAS, J. ; PASSOS, A. ; COURNAPEAU, D. ; BRUCHER, M. ; PERROT, M. ; DUCHESNAY, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830
- [Qui86] QUINLAN, J. R.: Induction of Decision Trees. In: *Mach. Learn.* 1 (1986), März, Nr. 1, 81–106. <http://dx.doi.org/10.1023/A:1022643204877>. – DOI 10.1023/A:1022643204877. – ISSN 0885–6125
- [Qui93] QUINLAN, J. R.: *C4.5: Programs for Machine Learning*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993. – ISBN 1558602380
- [RJ⁺91] RAUDYS, Sarunas J. ; JAIN, Anil K. u. a.: Small sample size effects in statistical pattern recognition: Recommendations for practitioners. In: *IEEE Transactions on pattern analysis and machine intelligence* 13 (1991), Nr. 3, S. 252–264

-
- [RRC19] REBALA, Gopinath ; RAVI, Ajay ; CHURIWALA, Sanjay: *An Introduction to Machine Learning*. 2019. <http://dx.doi.org/10.1007/978-3-030-15729-6>. <http://dx.doi.org/10.1007/978-3-030-15729-6>. – ISBN 978-3-030-15728-9
- [SL09] SOKOLOVA, Marina ; LAPALME, Guy: A systematic analysis of performance measures for classification tasks. In: *Information processing & management* 45 (2009), Nr. 4, S. 427–437
- [SMGC14] SMITH, Michael R. ; MARTINEZ, Tony ; GIRAUD-CARRIER, Christophe: An instance level analysis of data complexity. In: *Machine learning* 95 (2014), Nr. 2, S. 225–256
- [ZZ121] *Scikit Learn*. <https://scikit-learn.org>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ221] *React*. <https://reactjs.org/>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ321] *Flask*. <https://flask.palletsprojects.com/>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ421] *Recharts*. <https://recharts.org/>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ521] *Material UI*. <https://material-ui.com/>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ621] *Scipy*. <https://www.scipy.org/>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ821] *Diabetes Datensatz*. <https://github.com/plotly/datasets/blob/master/diabetes.csv>, 2021. – Zuletzt aufgerufen am 19.09.2021
- [ZZ921] *Iris Datensatz*. <https://github.com/plotly/datasets/blob/master/iris.csv>, 2021. – Zuletzt aufgerufen am 19.09.2021

Abbildungsverzeichnis

2.1	Tabelle zur Analyse einer Diabetes-Erkrankung	4
2.2	Tabelle zur Klassifikation von Schwertlilien	7
2.3	Die Machine Learning Pipeline	8
4.1	Angepasste Diabetes-Tabelle	18
4.2	Normalverteilungen der Körpergröße von Frauen und Männern	21
4.3	Verkehrsmittel Pünktlichkeit Tabelle	23
4.4	Beispiel eines Decision Trees	23
4.5	Sigmoide Funktion	30
4.6	Kostenfunktion	32
4.7	Neuronales Netz	35
5.1	ROC Kurve	42
6.1	Architektur der Anwendung	48
6.2	User Interface Eingabe Eigener	50
6.3	Aktivitätsdiagramm Eingabe Webanwendung	51
6.4	User Interface Eingabe Eigener	52
6.5	User Interface Evaluation Qualitativ	54
6.6	User Interface Evaluation Probabilistisch	54
6.7	User Interface Vergleich	55
6.8	Evaluation eigener Klassifikator Skript	56
6.9	Datensatz Format Skript	57
6.10	Generierung von Indizes Skript	59

Tabellenverzeichnis

5.1	Konfusionsmatrix für mehrere Klassen	38
-----	------------------------------------------------	----

Name: Christian Gebhardt

Matrikelnr.: 1579200

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Bayreuth, 27. September. 2021

Christian Gebhardt