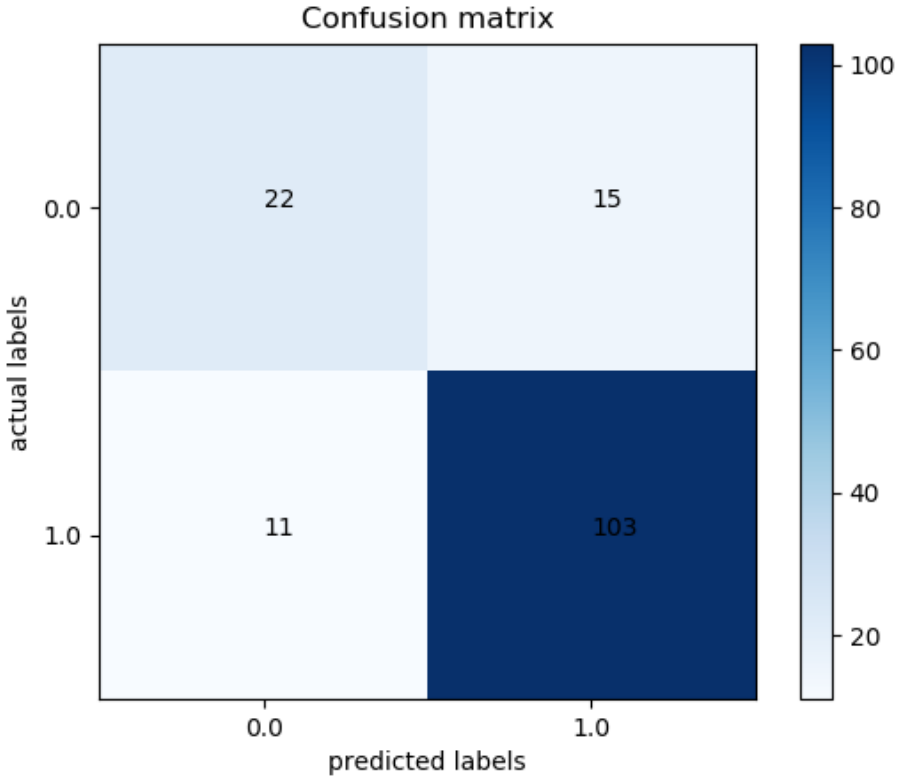
UNIVERSITÄT ZU LÜBECK - INSTITUT FÜR MEDIZINISCHE INFORMATIK

Report
Voice-based Early Diagnosis of Parkinson

Jesse Kruse - 675710
Konrad von Kügelgen - 676609
Falco Lentzsch - 685454

January 30, 2020



GENERELLE IDEE DES SZENARIO 2

Parkinson ist eine Erkrankung des zentralen Nervensystems. Dabei wird die Substanzia Nigra im Gehirn fortschreitend zerstört, was die Dopaminproduktion hemmt. Dieses ist unter anderem für die Funktionalität des Bewegungsapparates nötig. Dadurch wird die Motorik der Patienten mit Fortschreiten der Krankheit zunehmend schlechter. Betroffen sind davon auch die Muskulaturen, die für das Sprechen verantwortlich sind. Daher ist die Idee, Muster in den Aufnahmen zu erkennen, die typischerweise bei Erkrankten auftreten, um in der Folge möglichst früh und einfach Hinweise auf das Vorliegen der Erkrankung feststellen zu können. Dazu sollen verschiedenen maschinelle Lernalgorithmen ausprobiert und ihre Performance verglichen werden.

1 INTRO

In der *intro.py*-Datei ging es zunächst darum, sich mit Funktionalitäten und gewissen Aufrufen vertraut zu machen. Obwohl es in diesem Szenario um eine stimmenbasierte Diagnose von Parkinson geht und wir so früher oder später mit Audiodateien hantieren müssen, tasteten wir uns zunächst mit einem gegebenen Datensatz an das Thema heran. Ein Weindatensatz stand uns zur Verfügung, mit welchem wir im *intro.py* Funktion und Visualisierung testeten.
Bevor wir uns dem Wein widmeten, starteten wir mit einer Visualisierung von zufälligen Punkten. Hier erhielten wir durch den Funktionsaufruf *make_blobs* die Matrix X und den Vektor y. Während X eine zweizeilige Matrix mit je 500 Einträgen war, standen in y an den korrespondierenden Stellen die sog. Labels um die Daten aus X zu klassifizieren. Die Zeilen in X stehen für die x- und y-Werte eines zweidimensoinalen Punktes. So erhalten wir also 500 verschiedene Punkte, welche je einer Klasse/einem Cluster (durch y) zugeordnet sind. Insgesamt gab es 4 Cluster. // Mithilfe von *matplotlib.scatterplot* konnten wir unsere Datenpunkte in den zugehörigen Clustern visualisieren. Das resultierende Streudiagramm zeigte also 4 Punktwolken unterschiedlicher Farben in einem kartesischen Koordinatensystem. Dazu nutzen wir im Aufruf den gegebenen Vektor mit Farbbezeichnungen und ließen die Funktion unsere Datenpunkte, je nach zugehörigem Cluster, mit einer Farbe mappen und stellten das Ergebnis des Plots schließlich visuell dar.

Nun widmeten wir uns dem Datensatz der Weine. Dazu erhielten wir eine .data Datei, die wir einlasen. Jede Zeile dieser Datei enthält 14 Werte. Der erste ist das zugehörige Cluster und darauf folgend, mit einem Komma getrennt, stehen die Werte zu bestimmten Eigenschaften eines Weines. Jede Zeile repräsentiert einen Wein. Wir lesen hier also die erste Spalte als Label-Vektor ein und die restlichen Daten als Matrix mit der *shape* $(178, 13)$. Die Matrix und der Vektor werden von der Funktion zurückgegeben. Anschließend nutzen wir verschiedene Funktionen um die Daten mithilfe verschiedener Funktionen aufzubereiten und dar zu stellen:

1.1 Vorverarbeitung

Zunächst mussten die Daten normiert werden. Wir suchen dazu für jede Klasse zunächst das Maximum und das Minimum und speichern diese in Arrays. Anschließend durchlaufen wir unsere Daten pro Klasse. Dabei ist das der skalierte Wert

$$x_{new} = (x - Min_{Klasse}) / Max_{Klasse} \tag{1.1}$$

Um Daten in Trainings und Test-Daten zu unterteilen, nutzen wir die Funktion *split*. Diese erhält eine Matrix mit Daten und einen Vektor mit den zugehörigen labels. Außerdem wird

eine Zahl zwischen 0 und 1 übergeben, welche aussagt, wie groß der Anteil der Trainingsdaten sein soll. 0,2 resultiert in 20% Trainsingsdaten und 80% Testdaten. Wir haben den Aufruf an eine existierende Funktion *train_test_split* weitergeleitet, welche uns eben genau 4 Elemente zurück liefert - X_train, X_test, y_train und y_test. Als random state übergeben wir nichts, was bedeutet, dass die Funktion den *np.random* state nimmt, um die Daten vorher zu mischen. Jeder Aufruf der Funktion liefert also unterschiedliche Aufteilungen der Daten zurück.

Nun wurden verschiendene Algorithmen zur visuellen Trennung der Daten verwendet und die Ergebnisse ähnlich wie vorher beschrieben dargestellt.

1.2 Principal Component Analysis

Für die **Principal Component Analysis (PCA)** stellt *sklearn* eine fertige Funktion zur Verfügung. Die PCA dient dazu die Hochdimensionale Daten in einen geringer dimensionalen Raum abzubilden. Dabei werden linear abhängige Variablen zusammengefasst, sodass möglichst wenig Information verloren geht. Was unter der PCA "leidet" ist die Interpretierbarkeit. Hatten vorher konkrete beobachtete Variablen, werden werden diese nun teilweise zu neuen zusammengefasst, die unter Umständen nur noch schwer in unsere Lebenswelt übertragbar sind. Da es bei unserer Problemstellung aber ohnehin um das automatische lernen von Features geht können wir dies vernachlässigen. Durch die Verschiebung unserer Koordinatenachsen erhalten wir eine Trennung der entsprechenden Klassen in unserer Visualisierung.

1.3 Linear Discriminant Analysis

Die **Linear Discriminant Analysis (LDA)** ist ebenfalls ein Verfahren, das genutzt wird um die Dimensionalität der Daten zu verringern. Dabei wird eine linearer Diskriminator gesucht, also in 2D beispielsweise eine Gerade, die gegebene Datenpunkte möglichst gut von einander trennt. Auf diese könnten nun die Datenpunkte abgebildet werden, um einen möglichst geringen Datenverlust mit sich zu bringen.

1.4 Support Vector Machine

Außerdem kam eine **Support-Vektor-Machine(SVM)** zum Einsatz. Dabei handelt es sich um einen machinelearning Algorithmus, der iterativ einen optimalen Discriminator zwischen Klassen findet.Dabei wird optimiert er auf einen möglichst großen Abstand der Klassenpunkte zu dieser Trennebene. Der Unterschied zu den vorherigen Methoden ist, dass er theoretisch in der Lage ist nicht lineare Hyperebenen zwischen Klassen zu finden. Wie diese Ebene aussieht ist dabei abhängig von dem gewählten Kernel. In unserem Fall nutzten wir allerdings einen Linearen, sodass wir ebenfallse eine Gerade erhalten haben. Außerdem wurde eine Klasse entfernt, da die klassische SVM zwei Klassen trennen kann.

2 Parkinson Python Skript

Nachdem wir uns nun durch den Datensatz der Weine ein wenig an das Clustern und die Visualisierung von Daten herangetastet hatten, widmeten wir uns der medizinischen Fragestellung der Parkinson Sprachdiagnose.

Datensatz

Den Datensatz erhielten wir von der Internetseite des *UCI Machine Learning Repository*. Dieser hatte die *shape* (758, 755). Die ersten beiden Zeilen waren Beschreibungen der Features (Spalten), die wir für unsere numerischen Kalkulation nicht benötigten. Für jeden Patienten - hier waren es circa 250 - existieren in dem Datensatz drei Zeilen mit je 758 Eigenschaften. Wichtig für unsere Arbeit war die letzte Spalte. Diese enthält Information über die Ausprägung der Krankheit. War die Zahl 1, so ist der Patient mit Parkinson diagnostiziert, 0 glich Gesundheit. Die letzte Spalte stellte also unsere *Labels* für den weiteren Verlauf dar, der Rest als *Daten*.

2.1 Classifier und Performanz

Die Performanz eines *Classifiers* kann man durch verschiedene Methoden darstellen. In diesem Szenario betrachteten wir zwei dieser Möglichkeiten:

1. Confusion-Matrix
2. Statistische Werte

In Abschnitt 2.2 dieses Dokumentes ist die Verwendung des RandomForestClassifiers erklärt. *Fig. 2.1* zeigt uns das Ergebnis. Eine Confusion-Matrix zeigt ein Verhältnis von der Vorhersage zur Wirklichkeit. Man stellt hiermit dar, wie gut eine Vorhersage auf die Wirklichkeit passt - in diesem Beispiel also wie gut der Classfier eine Vorhersage treffen konnte. Man vergleicht dazu die eigentliche Wahrheit (*actual*) mit der Vorhersage (*prediction*). Wir sehen auf der x-Achse die vorhergesagten Labels und auf der y-Achse die Wahrheit. Auf der Diagonalen von links oben nach rechts unten sehen wir die perfekten Übereinstimmungen. Hier ist die Vorhersage korrekt und entspricht der Wahrheit. 22 "Zeilen" unseres Testdatensatzes wurden korrekt als nicht krank vorhergesagt und 103 korrekt als krank. 15 "Zeilen" des Testdatensatzes wurden zwar als krank vorhergesagt, jedoch sind diese in Wahrheit Gesund. (Hier wird von Zeilen gesprochen, da zu einem Patienten theoretisch 3 Zeilen mit Daten gehören.)

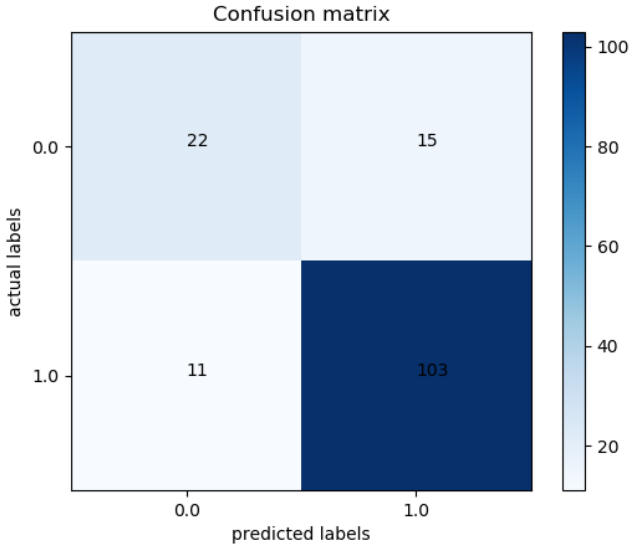


Figure 2.1: Ergebnis des RandomForestClassifiers

2.2 Prozentuale Spaltung, Evaluation: Confusion-Matrix

Bevor wir uns an die aufwendigere Verarbeitung und Klassifizierung des Datensatzes machten, testeten wir den *RandomForestClassifier*. Dazu teilten wir unseren Datensatz ohne die ersten beiden Spalten (diese repräsentieren ID und Geschlecht) in 80% Trainings- und 20% Testdaten durch eine gegebene Funktion *split_into_training_testing* auf. Den gewählten Classifier trainierten wir nun also auf den Teil des Datensatzes. Dieser erhielt eine Matrix mit Features und die dazu gehörigen Labels um die Krankheitsausprägung anzuzeigen. Der default der *n_estimators* ist 100 und entspricht der Anzahl an Binärbäumen des RandomForestClassfiers. Der Classifier erstellt also mehrere Binärbäume auf Grundlage von Features aus dem Trainingsset, die als Ergebnis 1 oder 0 (krank oder gesund) liefern. Idealerweise werden also die Features der Testdaten von den erstellten Binärbäumen des trainierten Classifiers analysiert und "ausgewertet". Die Klassifizierung der Daten geschieht dann als eine Art Mehrheitsvotum. Wenn mehr Binärbäume das Ergebnis 1 entscheiden, so wird das Label dieses Datensatzes (also dieser Zeile) als krank eingestuft. Durch das Trainieren des Classifiers, das sogenannte *fitting*, lässt sich auf Basis der Testdatensätze eine Vorhersage errechnen. Diese Vorhersage, erhalten wir als Liste von Nullen und Einsen und können diese mit der Wahrheit vergleichen, das Ergebnis als Confusion-Matrix plotten und so eine Einschätzung der Classifier-Performanz gewinnen.

2.3 Cross-Subject Validation, Evaluation: Statistische Werte

Für die Cross-Subject Validation (Kreuzvalidierung) eignete sich in unserem Szenario die Verwendung von Statistiken als Einschätzung und Bewertung des verwendeten Classifiers. Grund dafür ist die zu Grunde liegende Methodik *Leave-One-Out (LOO)*. Hierbei wird der Datensatz in k Teile unterteilt. Hier waren es circa $k = 250$, da wir nach Patienten IDs unterteilten. Wir nahmen also $\frac{249}{250}$ als Trainingsdaten und $\frac{1}{250}$ als Testdaten. Wie auch zuvor gilt es, den gewählten Classifier zu trainieren um anschließend eine Vorhersage auf Testdaten bekommen zu können. Dies exerzierten wir für 250 Fälle durch. Nach jeder Vorhersage des Classifiers, speicherten wir uns die statistischen Größen Präzision, Sensitivität und Spezifität. Diese Werte wurden dann pro Durchlauf aufaddiert und am Ende durch die Anzahl der Durchläufe geteilt. So erhielten wir Durchschnittswerte der drei statistischen Größen, um die Perfomanz des Classifiers nach k Vorhersagen besser einschätzen zu können.

3 Ergebnisse

Wir probierten verschiedene Classifier in der Kreuzvalidierung aus: den *DecisionTree*, den in 2.2 beschriebenenen *RandomForest*, *K-Means*, *K-Nearest Neighbors* und *AdaBoost*, um ein paar zu nennen. Die besten Ergebnisse erzielten wir mit dem AdaBoostClassifier $[85\%, 71\%, 20\%] \hat{=} [Präz, Sens, Spez]$ und das zweit beste mit dem RandomForestClassifier. Dieser gab uns eine Präzision von 82%, eine Sensitivität von 72% und eine Spezifität von 18%. Diese Werte schwanken bei mehreren Tests um $\pm 5\%$.
Wie dem Betrachter auffällt, ist die Spezifität unterirdisch. *(Siehe Fig. 3.1)* Ein Wert unter 50% ist jedoch ungewöhnlich, denn bevor ein Classifier alle Datensätze als "krank" einstuft und damit eine gute Sensitivität erhalten würde, wäre es sinvoller eine 50 zu 50 Verteilung zu raten. Wir vermuten also, dass unsere Berechnungen der Spezifität fehlerhaft sind, konnten jedoch bisher nichts feststellen. Dies zeigt jedoch ein wunderbares Beispiel der statistischen Interpretierbarkeit auf. Wenn ein Classifier mit einer guten Sensitivität angepriesen wird, bezeugt dieser Wert nicht direkt die Güte dieser Klassifizierung. Die Wichtigkeit der Werte ist je nach Situation unterschiedlich. In unserem Szenario wollen wir idealerweise eine hohe Sensitivität und Spezifität haben. Patienten sollen, wenn sie als klassifiziert werden, auch bestenfalls dieser Klasse

entsprechen. Viele falsch als krank (niedrige Spezifität) oder viele falsch als gesund (niedrige Sensitivität) diagnostizierte Patienten möchte man in jedem Fall vermeiden.

```
============================
Classifier:  KNeighborsClassifier
****Results****

 Cross subject validation: [0.7658730158730165, 0.6553571428571425, 0.20198412698412704] [acc, sensivity, specifity]
============================
Classifier:  KMeans
****Results****

 Cross subject validation: [0.45899470899470907, 0.5484126984126982, 0.17182539682539685] [acc, sensivity, specifity]
============================
Classifier:  DecisionTreeClassifier
****Results****

 Cross subject validation: [0.7658730158730161, 0.6678571428571426, 0.18849206349206354] [acc, sensivity, specifity]
============================
Classifier:  RandomForestClassifier
****Results****

 Cross subject validation: [0.8161375661375659, 0.7176587301587302, 0.18115079365079367] [acc, sensivity, specifity]
============================
Classifier:  AdaBoostClassifier
****Results****

 Cross subject validation: [0.8492063492063491, 0.7117063492063491, 0.1950396825396826] [acc, sensivity, specifity]
============================
```

Figure 3.1: Ergebnisse der Classifier