Projektbericht – Stroke

Inhaltsverzeichnis

Business Understanding

Ein Schlaganfall ist eine ernsthafte medizinische Notlage, die das Leben eines Menschen abrupt verändern kann. Die Symptome eines Schlaganfalls können plötzlich auftreten und sind vielfältig. Nach Angaben der Weltgesundheitsorganisation (WHO) ist der Schlaganfall die zweithäufigste Todesursache und für etwa 11% der Gesamttodesfälle. Es ist deswegen wichtig, frühzeitig Maßnahmen zu ergreifen, um das Risiko eines Schlaganfalls zu minimieren.

Das Ziel dieses Projektes ist es ein Frühwarnsystem zu erstellen, dass anhand von klinischen Merkmalen das Risiko eines Schlaganfalls prognostizieren kann. Dieses System soll medizinisches Personal bei der Vorsorge unterstützen.

Hierbei wird untersucht, ob ein Frühwarnsystem in Verbindung mit Methoden des maschinellen Lernens effektiv und präzise Vorhersagen treffen kann. Hierzu benötigen wir eine Kennziffer um unser System zu bewerten. In unserem Szenario betrachten wir nur die zwei Fälle Schlaganfallrisiko (positiver Fall) und kein Schlaganfall (negativer Fall). Es ist für uns am wichtigsten den positiven Fall (Schlaganfall) korrekt vorherzusagen. Deswegen haben wir uns als Performance-Metrik für den sogenannten Recall-Score entschieden. Der Recall-Score, auch bekannt als Sensitivität, ist eine Performance-Metrik, die in der Klassifikation verwendet wird, um die Fähigkeit eines Modells zu messen, positive Fälle korrekt zu identifizieren. Genauer gesagt gibt der Recall Score das Verhältnis der korrekt als positiv erkannten Fälle zur Gesamtzahl der tatsächlich positiven Fälle wieder. Als Ziel wollen wir einen RecallScore von **95%** erreichen.

Beschreibung der Daten

Die Datenbeschreibung ist separat im Jupyter Notebook „Stroke.ipynb“ enthalten

Beschreibung der Datenvorbereitung

Strukturelle Anpassungen

‚Stroke.py‘, function ‚init_data()‘

- Die Spalte ‚id‘ liefert keine wertvolle Information und wird daher entfernt.
- Wir haben uns vorerst gegen eine Hauptkomponentenanalyse (PCA) entschieden, da die Spalte ‚avg_gluceso_level‘ vermutlich wichtig für die Zielspalte ist, aber nicht normalverteilt zu sein sein (hoher Unterschied bei Median und Mean).
- Der Datensatz enthält nur eine einzige Zeile mit gender=‘Other’. Es wird entweder im Trainingsset oder im Testset ein Datensatz mit Gender='Other' fehlen. Daher wird dieser eine Eintrag gelöscht.
- Die Spalte ‚ever_married‘ enthält die Strings ‚Yes‘ und ‚No‘. Diese Werte werden in die numerischen Werte 1 und 0 transformiert.
- Die Werte in der Spalte ‚smoking_status‘ werden in die Ordnung: (‚never smoked‘, ‚Unknown‘, ‚formerly smoked‘, ‚smokes‘) überführt und in die numerischen Werte (0, 1, 2,

3) transformiert. Der Wert ‚Unknown' ist nicht zuzuordnen und wird daher neutral in der Mitte platziert.

- Die Spalten ‚gender', ‚work_type', ‚Residence_type' enthalten kategorische Daten. Da Algorithmen für maschinelles Lernen numerische Daten erfordern, werden diese ‚One Hot' encodiert.
- Nach den strukturellen Anpassungen sind die Datentypen aller Spalten auf ‚float64' konvertiert.
- Der Datensatz enthält nach den strukturellen Anpassungen 17 Spalten.

Inhaltliche Anpassungen

siehe ‚Stroke.py', function ‚prepare_data()'

- Die Spalten mit kontinuierlichen Daten werden standardskaliert. Eine Minmax Skalierung ist in der Datenvorbereitung parametrisiert und wurde während der folgenden Algorithmenevaluierung zum Vergleich genutzt.
- In der Spalte ‚bmi' fehlen Werte. Diese werden mit ‚KNNImputer' aufgeüllt. Es werden jeweils 5 Nachbarn als Basis für die aufzufüllenden Werte bestimmt.

Daten splitten

siehe ‚Stroke.py', function ‚split_dataset()'

- Die Daten werden in Trainings- und Testdaten im Verhältnis 80:20 aufgesplittet. Dabei wird nach der Zielspalte ‚stroke' stratifiziert.
- Der Datentyp der Zielspalte ‚stroke' wird in ‚int' konvertiert.

Anwendung der Algorithmen

Für das weitere Vorgehen im Projekt haben wir bekannte Klassifikationsalgorithmen angeschaut. Die Ergebnisse für diese Algorithmen finden sich im jeweiligen Abschnitt.

Allgemeines Vorgehen

Ein großes Problem in unserem Datensatz ist die Unbalanciertheit. Um dieses Problem zu bekämpfen, betrachten wir zwei Ansätze, die wir hier kurz erläutern. Im Allgemeinen gibt es keine universelle Lösung für dieses Problem.

Gewichtung von Klassen

Die Gewichtung von Klassen beinhaltet die Anwendung eines Gewichtungsfaktors auf die Klassen während des Trainings des Modells. Der Gewichtungsfaktor wird verwendet, um die Kosten für falsch klassifizierte Beispiele in den unterrepräsentierten Klassen zu erhöhen und somit die Bedeutung dieser Klassen für das Modell zu betonen. In scikit-learn besitzen viele der Algorithmen das Attribut „class_weights", wo man eine Gewichtung angegeben kann.

SMOTE (Synthetic Minority Over-sampling Technique)

Das Hauptziel von SMOTE besteht darin, die Anzahl der Minderheitsfälle zu erhöhen, indem synthetische Beispiele generiert werden, um das Ungleichgewicht auszugleichen. Dies geschieht durch Erzeugung neuer Beispiele durch Interpolation zwischen den vorhandenen Minderheitsfällen. Dieses Prozess wird nur auf die Trainingsdaten angwendet und passiert vor dem Trainieren des Algorithmus. Wir nutzen in unseren Projekt die Implementierung in der Python-Bibliothek „imblearn“.

Algorithmus 1: kNN

Prinzip:
Ähnlichkeiten zwischen Objekten ordnen diese in Nachbarschaften ein.
Voraussage richtet sich nach den Elementen der Nachbarschaft

Gründe für die Wahl des Algorithmus
Vorteile:

- Einfacher Algorithmus (wenig Paramter)
- Kleiner Datensatz, wenig Dimensionen

Skalierung ist für den Algorithmus wichtig und wurde mit StandardScaler bzw. MinMax Scaler durchgeführt.

Anwendung auf Trainingsdaten (auf gut Glück)

Zu Beginn betrachten wir den KNN Algorithmus naiv mit Standard Parameter ohne jegliche Optimierung um einen Schlaganfall vorherzusagen auf den Trainingsdaten.
Da die EDA gezeigt hat, dass der Datensatz sehr unbalanciert ist versprechen wir uns nicht allzu viel davon.

Naiv Ansatz fit und predict	Ergebnis
Parameter	knn = KNeighborsClassifier(n_neighbors=5,p=2, metric='minkowski')
Durchschnittlicher Recall-Score	0.08
Durchschnittlicher Precision-Score	0.79
Durchschnittler ROC-AUC Score	0.537

Der Precision Score von 0.79% zur Vorhersage eines Schlaganfalls ist für einen naiv Ansatz nicht mal so schlecht. Allerdings liegt der Recall bei 0.08 und ist somit unbrauchbar.
Der Datensatz ist sehr unbalanciert. Weniger als 5% des gesamten Datensatzes hat einen Stroke. Trainingsdaten und Testdaten wurden im Verhältnis 80:20 gesplittet. Dem Algorithmus fehlen vermutlich Daten mit Stroke.

Anwendung von Cross_Val_Score mit Variationen von k und p

In diesem Versuch wurden über 2 for Schleifen versucht mit Variation der Parameter k (1,2,5,10,50,100,150) und p (1,2,3,4,50,100) mit fit und Berechnung des cros_val_score ein guten Recall Wert zu bekommen.

Cross_Val_Score mit Variationen von k und p Scoring: recall	Ergebnis

Parameter	`for k in (1, 2, 5, 10, 50, 100, 150):`
	`# Nachbarn`
	`  for potenz in (1, 2, 3, 4, 50, 100)`
	`Rest: default`
	`scoring: recall`

Cross Val Score Mean: 0.08555706316900347	0.1007

Dieser Versuch wurde relativ schnell abgebrochen, da die Werte von recall zu schlecht waren. Vermutlich führt die weitere Aufteilung der Daten im Cross Val Score in Trainings- und Validierungsdaten zur weiteren Reduktion von Stroke Datensätzen. Dies könnte noch genauer untersucht werden.

Anwendung Oversampling SMOTE() fit und predict ohne Cross_Val_Score

In diesem Versuch wurde Oversampling mit SMOTE() **auf den gesamten Trainingsdatensatz** angewendet. (X_train UND y_train)

Oversampling (SMOTE) gesamter Trainingsdatensat- fit, predict ohne Cross_Val_Score	Ergebnis
Parameter	knn = KNeighborsClassifier(n_neighbors=5,p=2, metric='minkowski')
Durchschnittlicher Recall-Score	0.99
Durchschnittlicher Precision-Score	0.90
Durchschnittler ROC-AUC Score	0.937

Diese Werte sehen sehr gut aus. Allerdings könnte dies ein massives **Overfitting** bedeuten. Verwendung von Oversampling auf den gesamten Trainigsdatensatz (X_train und y_train) mit SMOTE() führten zuerst zu Overfitting auf den Trainingsdaten. Sehr gute Ergebnisse waren trügerisch.

Test auf Testdaten zur Prüfung des guten Ergebnisses

Predict auf den Testdaten	Ergebnis
Parameter	knn = KNeighborsClassifier(n_neighbors=5,p=2, metric='minkowski')
Durchschnittlicher Recall-Score	0.42
Durchschnittlicher Precision-Score	0.10
Durchschnittler ROC-AUC Score	0.617

Das Ergebnis auf den Testdaten mit einem Recall von 0.42 und Precision von 0.1 ist sehr schlecht. Wie vermutet führt die Anwendung von Oversampling auf den gesamten Datensatz zu massivem Overfitting auf die Trainingsdaten.

Anwendung von Oversampling – SMOTE() Pipeline und Cross_Val_Score

Die Verwendung von Oversampling auf den gesamten Datensatz lieferte trügersiche falsche Ergbnisse.

Abhilfe
Oversampling nicht auf den kompletten Trainingsdatensatz anwenden, sondern durch Verwendung von **imlearn pipeline und SMOTE()**. Hier wird nur für die fit Methode oversampled (Trainingsdaten), nicht jedoch für predict (Validierungsdaten)

```
knn_pipe = imb_pipe([('oversample', SMOTE()),('knn', KNeighborsClassifier(n_neighbors=23, p=2, metric='minkowski'))])
knn_results_cross_val_score = cross_val_score(knn_pipe, X_train, y_train, cv=5, scoring='recall')
```

Oversampling mit Pipe, Cross_Val_Score	Ergebnis
Parameter	knn = KNeighborsClassifier(n_neighbors=23,p=2, metric='minkowski') `scoring: recall`
Cross Val Score Mean: 0.08555706316900347	0.628

Wie erwartet zeigt sich eine deutliche Verbesserung des Cross Val Score Mean mit Oversampling 0.628. Weiter soll es mit der Optimierung der Parameter gehen.

Anwendung von Oversampling – SMOTE() Pipeline und Gridsearch

Gridsearch mit Oversampling smote() zur Suche der optimalen Parameter
Varianten 1-4 für param_grid zur Optimierung, je nach Rechnerleistung und Zeit
Scoring: recall

```
#Varianten von param_grid zur Optimierung je nach Rechenleistung
# Variante 1
param_grid = {"knn__n_neighbors": np.arange(1,200),"knn__p": np.arange(1,3)}

#Variante 2
#param_grid = {"knn__n_neighbors": np.arange(50,200),"knn__p": np.arange(1,3),"knn__weights": ['uniform', 'distance']}

#Variante 3
#param_grid = {"knn__n_neighbors": np.arange(70, 200), "knn__p": np.arange(1, 3),"knn__weights": ['uniform', 'distance'],"knn__metric":
['minkowski', 'euclidean', 'manhattan']}

#Variante 4
#param_grid = {"knn__n_neighbors": np.arange(50, 200), "knn__p": np.arange(1,3),
#           "knn__weights": ['uniform', 'distance'], "knn__algorithm": ['ball_tree', 'kd_tree', 'brute'],
#           "knn__metric": ['minkowski', 'euclidean', 'manhattan']}
```

Der Gridsearch lieferte die optimalen Parameter. Auffällig der sehr hohe Wert für die Anzahl der Neighbors mit 185. Evtl. liegt es daran, dass sehr wenige Datensätze mit Stroke vorhanden sind.

Ermittelte Optimale Parameter durch GridSearch
n_neighbors: 185
p=1
weights=uniform
metric= euclidean

Mit den gefundenen optimalen Daten wird nun ein Test auf den Testdaten durchgeführt

Ergebnisse der Vorhersage auf dem Testdatensatz

Die gewonnenen Parameter vom Gridsearch werden nun nochmals zum Trainining auf den Trainingsdaten angewendet (fit), um anschllließend ein predict auf den Testdaten durchzuführen.

Vorhersage eines Schlaganfalls (1)

Fit mit optimalen Werten und predict auf Testdaten	Ergebnis
Parameter	n_neighbors: 185 p=1 weights=uniform metric= euclidean
Durchschnittlicher Recall-Score	0.84
Durchschnittlicher Precision-Score	0.09
Durchschnittler ROC-AUC Score	0.698

Vorhersage dass kein Schlaganfall (0)

Fit mit optimalen Werten und predict auf Testdaten	Ergebnis
Parameter	n_neighbors: 185 p=1 weights=uniform metric= euclidean
Durchschnittlicher Recall-Score	0.56
Durchschnittlicher Precision-Score	0.99

Ergebnisbetrachtung und Ausblick
Vorhersage von Stroke (1)
Recall 84%
Precision: 0.09

Bei der Optimierung der Parameter wurde auf die Vorhersage eines Stroke optimiert mit möglichst hohem Recall, um möglichst alle Strokes vorhersagen zu können. Dies wurde mit 84% relativ gut erreicht. Allerdings mit einer sehr schlechten Precision von 0.09. Das heißt es werden sehr viele Schlaganfälle falsch positiv vorausgesagt werden.

Ideen und Ausblick
Es gibt Ideen für weitere Tests zur Optimierung mit Oversampling, Cros Validation und Gridsearch. Das Beste wäre mehr Daten von Patienten mit Schlaganfall zu erhalten. In wie weit weiteres Oversampling noch der Realität entspricht muss weiter geprüft werden.

Der Algorithmus eignet sich aktuell daher nicht für die treffsichere Aussage ob das Risiko eines Schlaganfalls vorliegt.

Dennoch könnte auch eine falsch positives Ergebnis zu einer Lebensänderung der betroffenen Patienten führen was dann letztendlich gesundheitlich förderlich ist. Die psychische Belastung bei einer falsch Vorhersage sollte dabei allerdings auch beachtet werden.

Algorithmus 2: Logistische Regression

Motivation für den Algorithmus:

- Ist effizient, braucht keine große Menge an Rechenressourcen
- Benötigt keine Skalierung
- Einfach zu regulieren

Ergebnisse (ohne Beachtung des imbalancierten Datensatz)

Als erstes wird die logistische Regression auf den nicht balancierten Datensatz angewendet ohne dies zu beachten. Dadurch soll untersucht werden, ob es wichtig ist dies beim weiteren Vorgehen zu berücksichtigen.

Für dies wird ein GridSearch über folgende Parameter durchgeführt:

- solver: lbfgs, newton-cg, newton-cholesky, liblinear
- C: 0.001, 0.01, 0.1, 1, 10
- penalty: l1, l2
- max_iter: 150, 200, 500

Mit dem GridSearch wurden folgende besten Parameter gefunden, mit einem Recall-score von 0.02 für die unterrepräsentierte Klasse 1 (auf Trainingsdaten):

- solver: lbfgs
- C: 1
- penalty: l2
- max_iter: 150

Diese besten Parameter erzielen folgende Score-Ergebnisse für Klasse 1 bei den Testdaten:

- Recall-Score: 0.0
- Precision-Score: 0.0
- ROC-AUC-Score: 0.5

Wir haben den Recall-Score für die Bewertung des Modells ausgesucht (Begründung: siehe oben) und an diesem muss nun das Modell bewertet werden. In diesem Fall ist ein Recall-Score von 0.0 erzielt worden. Das zeigt, wie groß der Einfluss der nicht balancierten Daten auf das Ergebnis ist und wie wichtig es ist diese im weiteren Prozess zu beachten.

Für die Korrektur der nicht balancierten Daten werden hier zwei Möglichkeiten gezeigt. Dies ist zum einen die Klassen zu gewichten. Dies kann bei einigen Klassifikatoren der Bibliothek ‚scikit-learn' als Parameter ‚class_weight' eingestellt werden. Die andere Möglichkeit ist, die

Klassenstärke in Balance zu bringen. Das kann durch ein Oversamling der unterrepräsentierten Klasse erfolgen. Diese Klasse wird dabei vergrößert indem neue Datensätze durch Interpolation von zufällig ausgewählten Datensätzen, bei denen die nächsten Nachbarn beachtet werden, künstlich erzeuget werden. Die andere Klasse bleibt unverändert.

Ergebnisse (Methode der Klassengewichte)

Der Parameter ‚class_weight' kann bei den Klassifikatoren unter ‚scikit-learn' (bei denen er vorhanden ist) entweder mit ‚balanced' (automatische Anpassung durch die inverse Häufigkeit der jeweiligen Klasse) oder durch ein Übergabe eines Dictonary mit den gewollten Werten {0:1, 1:2} direkt eingestellt werden.

Für die optimale Parametersuche wurde das GridSearch mit folgenden Parameter ausgeführt:

- solcer: lbfgs, newton-cg, newton-cholesky, liblinear
- class_weight: {0:1, 1:10}, {0:1, 1:100}, {0:1, 1:1000}, {0:10, 1:1}, {0:100, 1:1}, balanced
- C: 0.001, 0.01, 0.1, 1, 10
- penalty: l1, l2
- max_iter: 150, 200, 500

Für diese Auswahl an Parameter ist der beste Parametersatz für die logistische Regression mit einem Recall- Score von 0.97 auf den Trainingssatz folgende:

- solver: lbfgs
- class_weight: {0:1, 1:100}
- C: 0.001
- penalty: l2
- max_iter: 150

Mit diesem optimierten Parametersatz werden folgende Scores für die logistische Regression und der Klasse 1 bei den Testdaten erreicht:

- Recall-Score: 0.96
- Precision-Score: 0.07
- ROC-AUC-Score: 0.64

Es wurden in allen drei Scores eine Verbesserung erzielt, aber das war auch nicht schwierig ;-). Dieses Ergebnis ist viel besser und entspricht auch dem vorgegebenen Ziel (Recall-Score > 0.95). Die beiden anderen Scores sind nicht gut. Der geringe Percision-Score sagt aus, dass die Qualität der Aussage nicht so gut ist. D.h. dass bei vielen falsch eine Diagnose gemacht wurde, obwohl sie gesund sind.

Ergebnisse (Methode des Oversamlings)

Nun wird die nächste Methode untersucht, um mit dem nicht balancierten Datensatz einen guten Score zu erreichen. Dies kann mit Hilfe der Bibliothek ‚imlearn' und der Funktion ‚smote' (für

oversampling) erzeugt werden.

Für einen bessere Vergleichbarkeit der beiden Möglichkeiten, die nicht balancierten Daten zu berücksichtigen, wird hier das GridSearch mit den gleichen Parametern wie bei der Möglichkeit mit den Klassengewichten (siehe oben) durchgeführt. Es fehlt nur der Parameter ‚class_weight'. Dieser wird hier nicht benötigt, weil die unterrepräsentierte Klasse künstlich vergrößert wurde.

Hierbei sind die folgenden Parameter die Besten für die logistische Regression mit dem oversampling-Ansatz für die Klasse 1. Mit diesen wird ein Recall-Score von 0.97 bei den Trainingsdaten erreicht:

- solver: liblinear
- C: 0.001
- penalty: l1
- max_iter: 100

Mit dieses Parametersatzes werden folgende Scores für die logistische Regression mit Oversampling-Methode für die Klasse 1 erreicht:

- Recall-Score: 0.90
- Precision-Score: 0.08
- ROC-AUC-Score: 0.69

Hier wurde ein schlechterer Recall-Score erzielt als bei dem Ansatz mit den Klassengewichten. Mit diesem wird das vorgegebene Ziel von einem Recall = 0.95 nicht erfüllen. Die beiden anderen Scores haben sich leicht verbessert. Sind aber immer noch nicht zufriedenstellend.

Zusammenfassung:
Für die Bewertung der Modelle und somit deren zugehörigen Parametern wurde der Recall-Score mit einem Wert von 0.95 ausgewählt, um möglichst alle Stroke-Fälle vorherzusagen. Hierbei wird in Kauf genommen, dass auch bei nicht-Stroke-Fällen diese als Stroke-Fälle klassifiziert werden. Da es um die Gesundheit und somit um schwerwiegende Folgen für die Menschen geht, ist es besser öfters bei einem gesunden Menschen einer ärztlichen Kontrolluntersuchung durchzuführen.

Das Ziel war es hier einen Recall-Score von mindestens 0.95 zu erreichen. Dies wurde hier mit der logistischen Regression erreicht!

Algorithmus 3: Support Vector Machine

Am Anfang betrachten wir eine naive Anwendung der Support Vector Machine ohne irgendwelche Methoden um die Mehrheit von Klasse 0 (Kein Schlaganfall) auszugleichen. Hierzu führen wir ein GridSearch über das folgende Parametergitter aus:

- 'kernel': ['linear', 'sigmoid', 'rbf'],
- 'C': [0.1, 0.2, 0.3, 1, 2, 3, 10, 20, 30]

Mit diesem Gitter erhalten wir die folgende Resultate:

Naiv	Ergebnis
Beste Parameter	{'C': 30, 'kernel': 'rbf'}
Durchschnittlicher Recall-Score	0.08
Durchschnittlicher Precision-Score	0.111
Durchschnittler ROC-AUC Score	0.629

Wir wollen den Recall-Score optimieren und in diesem Fall sehen wir, dass ein naiver Ansatz katastrophale Resultate liefiert. Deswegen ist es entscheidend, dass wir Maßnahmen für das Missverhältnis in den Klassen treffen müssen.

Anwendung mit Klassengewichtung

Die erste Möglichkeit ist mit der Definition von Klassengewichten. Im unseren Fall existiert eine Implementierung in scikit-learn. Diese wird durch den Parameter ‚class_weight' angesteuert. Eine Voreinstellung kann mit dem Parameter ‚balanced' angesteuert werden. Dieser berechnet ein Klassengewicht proptional zu den Anzahl der Klassen. Die exakte Formel ist gegeben durch

$$\#\text{Testdaten}/(2*\#\text{Testdaten der Klasse } i).$$

Wir erstellen einen Parametergitter mit den folgenden Optionen:

- 'svc__kernel': ['linear', 'sigmoid', 'rbf'],
- 'svc__class_weight': [{0: 1, 1: 10}, {0: 1, 1: 100}, {0: 1, 1: 1000}, 'balanced']

Mit diesem Gitter erhalten wir folgendes Resultat:

Klassengewicht	Ergebnis
Beste Parameter	{'svc__class_weight': {0: 1, 1: 1000}, 'svc__kernel': 'linear'}
Durchschnittlicher Recall-Score	1.000
Durchschnittlicher Precision-Score	0.056
Durchschnittler ROC-AUC Score	0.701

Auf dem ersten Blick ist es ein hervorragendes Resultat, da wir ein durchschnittlichen Recall-Score von 1 erhalten haben. Es ist somit auch wichtig andere Performance-Metriken zu betrachten. Wir sehen insbesondere das der Precision-Score sehr niedrig ist. Hieraus können wir ableiten, dass wir sehr viele falsch positive Fälle mit diesem Klassifikator bestimmen. Es ist abzuklären, ob ein solches

Verhalten wünschenswert ist.

Anwendung mit Undersampling

Wir machen nun ein weiteren naiven Ansatz um das Ungleichgewicht zu bekämpfen. Wir nutzen die ‚under-samping' Strategie, d.h. wir entfernen Einträge aus der dominiernde Klasse (Kein Schlaganfall). Eine Implementierung ist in der Bibliothek ‚imblearn' mit der Klasse ‚RandomUnderSampler' gegeben, welche zufällig die Einträge auswählt. Im Parameter ‚strategy' können wir auswählen wie das Verhältnis der einzelnen Klasse nach dem Undersampling sein soll.

Wir erstellen hierfür das folgende Parametergitter:

- 'estimator__kernel': ['linear', 'sigmoid', 'rbf'],
- 'under__sampling_strategy': [0.4, 0.5, 0.6, 'auto']

Mit diesem Gitter erhalten wir folgendes Resultat:

Undersampling	Ergebnis
Beste Parameter	{'estimator__kernel': 'rbf', 'under__sampling_strategy': 'auto'}
Durchschnittlicher Recall-Score	0.788
Durchschnittlicher Precision-Score	0.117
Durchschnittler ROC-AUC Score	0.823

Anwendung mit SMOTE

Im nächsten Schritt verwenden wir den SMOTE Algorithmus für ein Oversampling in unseren Trainingsdaten. Wie nutzen als Parametergitter die folgende Optionen:

- 'estimator__kernel': ['linear', 'sigmoid', 'rbf']

Als Resultat des GridSearchCV erhalten wir die folgende Werte:

SMOTE	Ergebnis
Beste Parameter	{'estimator__kernel': 'linear'}
Durchschnittlicher Recall-Score	0.813
Durchschnittlicher Precision-Score	0.134
Durchschnittler ROC-AUC Score	0.847

In allen drei Metriken verbessern wir uns mit der Verwendung des SMOTE-Verfahrens.

Anwendung mit SMOTE und Undersampling

Im Originalarbeit zum SMOTE-Verfahren notieren die Autoren N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer (siehe https://arxiv.org/pdf/1106.1813.pdf, Seite 352 Abschnitt 7) eine Verbesserung in den meisten Fällen, wenn man eine Kombination von Undersampling und dem SMOTE-Verfahren durchführt. Wir probieren diese Kombination und testen Sie auf das folgende Parametergitter:

- 'estimator__kernel': ['linear', 'sigmoid', 'rbf'],
- 'under__sampling_strategy': [0.4, 0.5, 0.6, 'auto'].

Aus diesem GridSearch erhalten wir das folgende Resultat:

SMOTE+Undersampling	Ergebnis
Beste Parameter	{'estimator__kernel': 'linear'}
Durchschnittlicher Recall-Score	0.808
Durchschnittlicher Precision-Score	0.131
Durchschnittler ROC-AUC Score	0.846

Wir stellen fest, dass wir für unseren Datensatz keine Verbesserung erhalten.

Vergleich und Ausblick

Es ist klar, dass die Methode mit den modifizierten Klassengewichte den besten Wert für Recall-Score liefert. Auf den Trainingsdaten liefert es einen Score von 1. Dieser Wert sollte ein stützig machen und wir wollen jetzt diese Parameter auf den Testdaten anwenden. Die Testdaten hat eine Größe von 1022 Einträgen. Für den optimalen Algorithmus mit Klassengewicht erhalten wir die folgende Werte:

Klassengewicht (Testdaten)	Klasse Schlaganfall	Klasse Kein Schlaganfall
Recall-Score	0.96	0.14
Precision-Score	0.05	0.99

Wir haben für die Testdaten auch noch den Score für die Klasse „Kein Schlaganfall“ mitberechnet um ein besseren Vergleich zu ermöglichen. Der Recall-Score für die Klasse „Kein Schlaganfall“ liegt bei 0.14 und verdeutlicht, dass wir in sehr vielen Fällen einfach ein Schlaganfall hevorsagen, auch wenn es kein Grundlage dafür geben kann. Diese Vermutung bestätigt der Precision-Score. In absoluten Zahlen erhalten wir 836 falsche negative Vorhersagen. Hierbei muss mit einem Experten abgeklärt werden, ob dies wünschenswert ist. Da Schlaganfall plötzlich eintreten und das Leben sehr stark beinträchtigen (bis zum Tod) kann, finden wir die große Anzahl an falsche negativ Resultate (Schlaganfallrisiko, auch wenn kein Risiko vorliegt) nicht so schlimm. Der Algorithmus erreicht somit den gewünschten Wert.

Natürlich müssen wir uns noch Fragen, ob ein Overfitting vorliegt. Wir erhalten auf den Testdaten den folgenden Score:

Klassengewicht (Trainingsdaten)	Klasse Schlaganfall	Klasse Kein Schlaganfall
Recall-Score	1	0.14
Precision-Score	0.05	1

Formal liegt also ein Overfitting im Sinne von dem Recall-Score vor. Da aber das Ergebnis auf den Testdaten dennoch sehr hoch ist, finden wir den Algorithmus immer noch für einen sinnvollen Kandidaten.

Wir wollen im Vergleich noch den zweitbesten Algorithmus (SMOTE) auf den Testdaten auswerten.

Wir erhalten als Resultat:

Klassengewicht	Klasse Schlaganfall	Klasse Kein Schlaganfall
Recall-Score	0.80	0.71
Precision-Score	0.13	0.99

Der SMOTE-Algorithmus liefert ein ausgeglichernes Ergebnis in beiden Klassen im Vergleich zu dem Klassengewicht. Hierbei ist natürlich die Frage, ob dies lohnenswert ist, da wir deutlich mehr Fälle von Schlaganfällen mit diesem Algorithmus übersehen.

Das Potential mit der Support Vector Machine ist noch nicht ausgeschöpft. Der nächste Schritt wäre es noch eine Regularisierung hinzufügen und diese mit dem GridSearch zu optimieren. Aus zeitlichen Gründen haben wir es hier nicht gemacht.

Algorithmus 4: Random Forest

Motivation für den Einsatz von Random Forest

- kann gut mit unbalancierten Daten umgehen
- schnell trainierbar im Vergleich zu meinen ersten Versuchen mit MLPClassifier
- gut geeignet für hochdimensionale Daten
- Daten müssen nicht skaliert oder transformiert werden. Kann gut mit Ausreißern und fehlenden Daten umgehen. Dennoch arbeite ich mit dem preprozessierten Datensatz.
- Random Forest wird gerne in der Medizin eingesetzt um Gesundheitstrends und -risiken zu beurteilen

Wie Random Forest arbeitet

Random Forest Classifier ist einer der meist genutzten Algorithmen im Machine Learning.
Die Grundidee ist es, viele Decision Trees zu trainieren und über deren Vorhersagen für die finale Vorhersage abzustimmen.

Fitting

Jeder Decision Tree erhält für das Training aus den Testdaten zufällig Daten mit Zurücklegen.

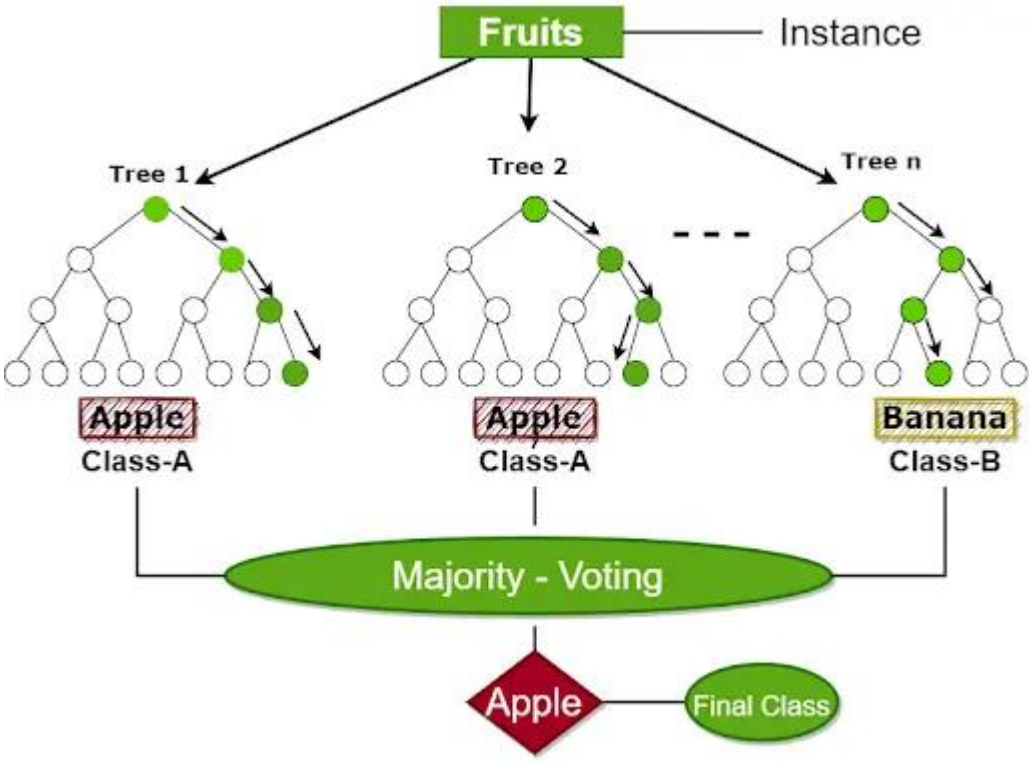
Beispiel
Trainingsdaten: [1, 2, 3, 4, 5]
Decision Tree 1: [1, 1, 3, 4, 4]
Decision Tree 2: [1, 1, 3, 5, 5]
Decision Tree 3: [2, 2, 2, 3, 4]

Jeder Decision Tree nutzt beim Training nur ein zufälliges Subset der Features.

Jeder Baum im Wald ist ein Unikat.

Vorhersage

Es wird ein Voting über alle Decision Trees durchgeführt.



Vor- und Nachteile im Vergleich zum Decision Tree

- keine Probleme mit Overfitting
- langsamer bei Training und Vorhersage
- besserer Umgang mit hochdimensionalen Daten

Evaluation 1 (auf gut Glück)

Es werden einige Parameter auf gut Glück variiert.

Parameter	nur Default Werte
Crossval recall score	[0. 0.025 0. 0. 0.], mean=0.005

Parameter	max_features=None, n_estimators=75, criterion='entropy',

Crossval recall score	[0. 0. 0. 0. 0.], mean=0.000

Parameter	max_depth=1, criterion='gini',
Crossval recall score	[0. 0. 0. 0. 0.], mean=0.000

Parameter	max_depth=5, max_feature=None, n_estimators=100, criterion='gini', min_samples_leaf=2
Crossval recall score	[0. 0. 0. 0. 0.], mean=0.000

Dieser Ansatz ist unbrauchbar.

Evaluation 2 (Klassengewichte)

Um die Imbalancierung des Datensatzes in der Zielspalte auszugleichen, wird der Parameter ‚class_weight' genutzt.

Parameter für alle Modelle	max_depth=5, max_feature=None, n_estimators=100, criterion='gini', min_samples_leaf=2
Parameter ‚class_weight' Scores	Modell 1: class_weight='balanced_subsample' CVS: [0.675 0.7 0.667 0.7 0.8], mean=0.708 Scores auf Testdaten: Klasse 1: Recall / AUC = **0.760 / 0.768** Klasse 0: Recall = 0.777
	Modell 2: class_weight='balanced' CVS: [0.65 0.7 0.667 0.7 0.8], mean=0.703 Scores auf Testdaten: Klasse 1: Recall / AUC = **0.760 / 0.768** Klasse 0: Recall = 0.776
	Modell 3: class_weight={0: 0.25, 1: 0.75} CVS: [0.025 0.025 0.051 0.05 0.025], mean=0.035 Scores auf Testdaten: Klasse 1: Recall / AUC = 0.040 / 0.518 Klasse 0: Recall = 0.953

	Modell 4: class_weight={0: 0.1, 1: 0.9} CVS: [0.4 0.45 0.359 0.525 0.5], mean=0.447 Scores auf Testdaten: Klasse 1: Recall / AUC = 0.400 / 0.649 Klasse 0: Recall = 0.899
	Modell 5: class_weight={0: 0.0487, 1: 0.9513} CVS: [0.65 0.7 0.667 0.7 0.8], mean=0.703 Scores auf Testdaten: Klasse 1: Recall / AUC = **0.760 / 0.767** Klasse 0: Recall = 0.775
	Modell 6: class_weight={0: 0.001, 1: 0.999} CVS: [0.95 0.825 0.923 0.825 0.925], mean=0.890 Scores auf Testdaten: Klasse 1: Recall / AUC = 0.860 / 0.723 Klasse 0: Recall = 0.586 Modell 7: class_weight={0: 0.0001, 1: 0.9999} CVS: [1. 0.925 0.949 0.825 0.975], mean=0.935 Scores auf Testdaten: Klasse 1: Recall / AUC = 0.880 / 0.702 Klasse 0: Recall = 0.524

Modelle 1,2,5 erzielen die besten Scores. Dies war zu erwarten, da die Gewichte zur vorliegenden Imbalance zwischen Klasse 1 und 2 passen. Dies zeigt sich im Vergleich der verschiedenen Scores.

Bei sehr hohen Gewichten für die Klasse „1“ ist der Recall Score höher. Dies geht jedoch zu Lasten des Recall Scores für Klasse „0“ und der AUC Score sinkt.

Im Folgenden wird auf den AUC Score optimiert, nicht auf den Recall Score.

Dieser Ansatz liefert gute Ergebnisse und wird weiter verfolgt.

Optimierung mit GridSearch

Für Modell 2 werden die Parameter mittels eines GridSearchCV optimiert (5 folds, Scoring='roc_auc'). Um den Rechenaufwand einzudämmen, erfolgt dies iterativ. In jeder Iteration werden einige Parameter getestet und in der nachfolgenden Iteration werden die besten Parameter übernommen.

Nach drei Iterationen ergibt sich die optimierte Parameterkombination:

- class_weight: 'balanced'
- max_features=None
- criterion: 'entropy'

- min_samples_leaf: 1
- max_depth: 3
- n_estimators': 1000

Mit diesen Parametern ergeben sich mit den Testdaten folgende Scores:

- Klasse 1: Recall / AUC / precision = 0.800 / **0.734** / 0.110
- Klasse 0: Recall / precision = 0.669 / 0.985

Bemerkenswerterweise ist nach der Optimierung der AUC Score auf den Testdaten gesunken.

Evaluation 3 (Oversampling)

Anstelle des Parameters ‚class_weight' wird nun Oversampling genutzt, um die Unbalanciertheit des Datensatzes auszugleichen. Dies hat sich bei den anderen in diesem Projekt eingesetzten Algorithmen bewährt.

Es wird die in Evaluation 2 optimierte Parameterkombination genutzt.

Es ergeben sich mit den Testdaten folgende Scores:

- Klasse 1: Recall / AUC / precision = 0.800 / **0.755** / 0.125
- Klasse 0: Recall / precision = 0.711 / 0.986

Ein erneuter Grid Search scheint sich nicht zu lohnen. Ein schneller Versuch liefert den gleichen AUC Score:

- Klasse 1: Recall / AUC / precision = 0.860 / **0.755** / 0.112
- Klasse 0: Recall / precision = 0.650 / 0.989

Evaluation 4 (Undersampling)

Anstelle von Oversampling kann auch Undersampling eingesetzt werden, um die Unbalanciertheit des Datensatzes auszugleichen.

Es wird die in Evaluation 2 optimierte Parameterkombination genutzt.

Der RandomForrest-Estimator liefert die Scores:

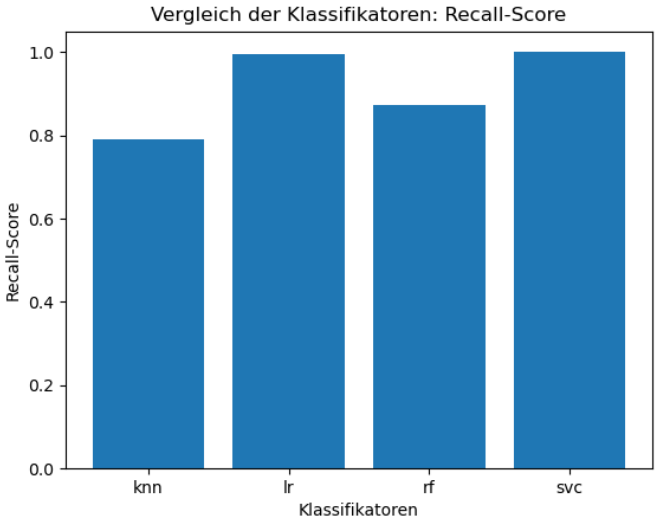
- Recall / AUC / precision Klasse 1 = 0.880 / **0.736** / 0.100
- Recall / precision Klasse 0 = 0.593 / 0.990

Fazit

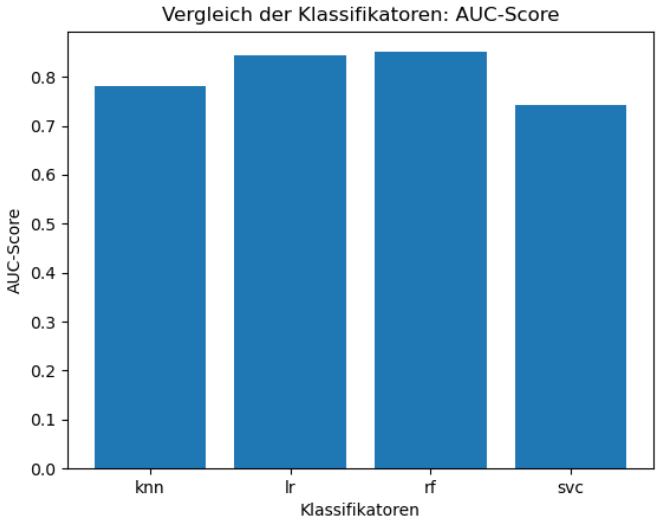
Der RandomForrest-Estimator eignet sich für Vorhersagen auf diesem Datensatz. Die besten Scores werden mit Oversampling der Daten erreicht. Ohne Oversampling mit Klassengewichten sind die Scores nur leicht schlechter.

Bewertung und Ausblick

Vergleich der 4 Algorithmen



Bei dem Recall-Score schneidet die Support Vector Machine mit einem Score von 1.0 am besten ab. Dicht gefolgt von der Logistischen Regression mit einem Score von 0.99. Mit einem etwas größeren Abstand folgen dann Random Forest (0.87) und KNN (0.79).



Bei dem AUC-Score sieht die Reihenfolge anders aus. Dort hat der Random Forest den besten Score mit 0.85. Hier ist wieder die logistische Regression mit einem Score von 0.84 auf den zweiten Platz. Dann folgen KNN mit einem Score von 0.78 und die Vector Machine mit einem Score von 0.74.

Zur besseren Übersicht:

	KNN	Log Regression	Random Forest	SVC

Recall-Score	0.79	0.99	0.87	**1.00**
AUC-Score	0.78	0.84	**0.85**	0.74

Voting - Ensemble Verfahren

Bei einem Voting-Ensemble geht es darum, eine Vorhersage zu treffen, die dem Durchschnitt mehrerer Klassifikatoren entspricht. Im Idealfall soll eine bessere Vorhersage als von jedem im Ensemble verwendeten Modell erzielt werden. Dabei kann nicht garantiert werden, dass das Voting-Ensemble eine bessere Leistung bietet als jedes einzelne im Ensemble verwendete Modell. Es gibt zwei Ansätze zur Vorhersage der Mehrheitsentscheidung für die Klassifizierung. Es gibt Hard-Voting und Soft-Voting. Bei der harten Abstimmung werden die Vorhersagen für jedes Klassenlabel summiert und das Klassenlabel mit den meisten Stimmen vorhergesagt. Der Hard-Voting-Klassifikator klassifiziert Daten basierend auf Klassenbezeichnungen und den jedem Klassifikator zugeordneten Gewichtungen. Der Soft-Voting-Klassifikator klassifiziert Daten basierend auf den Wahrscheinlichkeiten und Gewichtungen, die jedem Klassifikator zugeordnet sind

Hier wird das Hard-Voting umgesetzt, da wir anhand der Klassenbezeichnung klassifizieren möchten. In der folgende Tabelle werden die Scores des Vtoing-Klassifikators abgebildet.

	Recall-Score	Precision-Score	F1	AUC-Sore
Klasse 1	0.86	0.09	0.15	0.69
Klasse 0	0.53	0.99	0.69	-

Hier ist zu sehen, dass in diesem Fall der Voting-Klassifikator nicht besser abschneidet als einige einzelne Klassifikatoren. Daher ist es besser einen einzelnen Klassifikator zu benutzen.

Ausblick

Basierend auf diesem Projekt gibt es zahlreiche Möglichkeiten zur Verbesserung, die bisher noch nicht ausgeschöpft wurden. Ein wichtiger Aspekt ist die Anwendung von dimensionsreduzierenden Methoden wie der Hauptkomponentenanalyse. Dies könnte eine solide Grundlage für die Optimierung unserer Algorithmen bieten.

Aufgrund von Zeitbeschränkungen haben wir uns bei der Optimierung unserer Algorithmen auf den Recall-Score konzentriert. Bei einem Vergleich stellten wir jedoch fest, dass unsere Algorithmen eine hohe Fehlklassifikationsrate für die Klasse "nicht Schlaganfall" aufweisen. Dieses Problem könnte behoben werden, wenn wir uns auf die Optimierung anhand mehrerer Performance-Metriken konzentriert hätten.

Aus Zeitgründen haben wir Ensemble-Verfahren weitgehend vernachlässigt und uns lediglich auf das Voting-Verfahren konzentriert. Hier versteckt sich das größte Optimierungspotential, wenn wir Methoden wie Boosting oder Bagging berücksichtigen. Erste Ergebnisse in diese Richtung sind bereits in den Optimierungen der Random Forests erkennbar.