Übersicht Data Analysis and Augmentation Toolkit (DAAT) *

Florian Bayeff-Filloff, Dominik Stecher und Kai Höfig

24. Juni 2022

*Supported by Interreg Österreich-Bayern 2014-2020 as part of the project KI-Net

Inhaltsverzeichnis

1 Vorraussetzungen

Das DAAT setzt die folgenden Python Pakete voraus:
NUMPY, PANDAS, SEABORN, MATPLOTLIB, SKLEARN, SCIPY, KERAS, TENSORFLOW und TQDM.

2 Klassen- und Funktionsübersicht

Nachfolgend sind alle im DAAT enthaltenen Klassen zur Analyse, Datengenerierung und Verifikation, jeweils mit ihren enthaltenen Funktionen, aufgeführt. Hilfsfunktionen und spezielle Testfunktionen sind nicht einzeln aufgeführt.

2.1 Analyser Klasse

Die ANALYSER Klasse umfasst Funktionen zur erleichterten visuellen Datensatzanalyse, sowie zur einfachen Bearbeitung.

Klassen Definition

```
class daat.Analyser(dataset, target_label)
```

Erstellt ein neues Analyser Objekt, initialisiert mit dem Datensatz `dataset` und dem optionalen Ziellabel `target_label`.

Attribut:

- dataset : pandas Dataframe
  Datensatz innerhalb des Analyser, fungiert als Getter

Parameter:

- dataset : pandas Dataframe
- target_label : str, optional [Default : None]

Methoden Definitionen

```
def set_target(target_label)
```

Setter für `target_label`.

Informationsanzeige

```
def show_dataset_info()
```

Zeigt allgemeine Datensatzinformationen in Tabellenform an.

```
def show_feature_info(feature, violin)
```

Zeigt allgemeine Featureinformationen als Tabelle und Werteverteilung als Histogramm (numerisch) oder Countplot (kategorisch), sowie Boxplot an.

Parameter:

- feature : str
- violin : bool, optional [Default: False]
  Ersetzt den Boxplot durch Violinplot

Visualisierung

```
def plot_features(x_feature, y_feature, typ, inline, color)
```

Erstellt eine grafische Übersicht für alle numerischen Features. Jedes Feature wird als eigener Plot angezeigt.

Parameter:

- x_feature : str (array), optional [Default: None]
  Feature(s) auf X-Achse; falls nicht gesetzt: alle
- y_feature : str, optional [Default: None]
  Feature auf Y-Achse.
- typ : str, optional [Default: None]
  Ausgabe Plottyp; verfügbar: hist, scatter, box, line, strip
  Falls kein Feature oder nur x_feature angegeben: Histogramm
  Falls y_feature angegeben: Scatterplot.
- inline : int, optional [Default: 3]
  Anzahl Plots pro Zeile.
- color : bool, optional [Default: False]
  Färbt Plot target_label entsprechend ein; maximal 10 Klassen.

```
def plot_pair()
```

Zeigt Pairplot aller numerischen Features an.

```
def plot_correlation(annot)
```

Zeigt Korrelationsmatrix aller numerischen Features an.

Bearbeitung

```
def transform_feature(feature)
```

Transformiert kategorisches `feature` in numerisches und gibt LabelEncoder Objekt zurück.

```
def show_outlier(feature, n_percent)
```

Markiert Datensatzeinträge nach `feature` filtert über die Winserize Methode mit `n_percent` als Ausreißer und zeigt sie farblich markiert in allen Features an.

```
def remove_outlier()
```

Entfernt alle mit `show_outlier()` markierten Datensatzeinträge.

```
def rename_feature(feature_name, new_name)
```

Benennt Feature um

```
def drop_feature(feature)
def drop_line(line)
def drop_nan(feature)
def fill_nan(feature, value)
```

Funktionen zum Entfernen Datensatzeinträgen nach Spalte (feature) oder Reihe (line)

2.2 Generator Klasse

Die GENERATOR Klasse stellt Funktionen zur schrittweisen Erzeugung virtueller Samples basierend auf den Originaldaten und Expertenwissen zur Verfügung. Dazu ist der Generator wie eine Pipeline aufgebaut, in die sequentiell Instruktionen für jedes Feature hinzugefügt werden.

Klassen Definition

```
class daat_lib.Generator(dataset, target)
```

Erstellt ein neues Generator Objekt, initialisiert mit dem Datensatz `dataset` und dem Ziellabel `target`.

Parameter:

- dataset : pandas DataFrame
  Grunddatensatz zur Generierung
- target : string
  Ziellabel im Datensatz

Methoden Definitionen
Generator Pipeline Definition

```
def add_instruction(instruction)
```

Fügt `instruction` zur Erzeugungspipeline hinzu.

```
def remove_instruction(feature)
```

Entfernt Instruktion für `feature` aus Erzeugungspipeline.

```
def status()
```

Zeigt aktuell definierte Generator-Pipeline an.

Virtuelle Daten Generierung

```
def verify_setup(classification, balance)
```

Erzeugt virtuellen Datensatz gemäß definierter Pipeline. Zeigt Vergleichsübersicht aus zwei SVMs trainiert auf original und synthetischen Datensatz an. Parameter:

- classification : bool, optional [Default: True]
  Setzt Generatorart: True: Klassifikation; False: Regression
- balance : bool, optional [Default: False]
  Generator erzeugt bei Klassifikation Samples, um die Klassen auszugleichen.

```
def generate_syn_data(n_samples, classification, balance, equal)
```

Erzeugt synthetischen Datensatz der Länge `n_samples`.
Parameter:

- n_samples : int
  zu erzeugende Sampleanzahl.
- classification : bool, optional [Default: True]
  Setzt Generatorart: True: Klassifikation; False: Regression

- balance : bool, optional [Default: False]
  Generator erzeugt bei Klassifikation Samples, um die Klassen auszugleichen.

- equal: bool, optional [Default: False]
  Generator erzeugt je Klasse gleich viele Samples.

```
def get_syn_data(combine)
```

Gibt virtuelle erzeugten Datensatz zurück. Parameter:

- combine : bool, optional [Default: False]
  kombiniert synthetische mit original Daten für Rückgabe

2.3 Sample_Generator Klasse

Jede im DAAT verwendete Generatormethode erweitert die `Sample_Generator` Klasse. Eigene Methoden werden durch ihre Erweiterung hinzugefügt, mehr dazu in Abschnitt 4.

Klassen Definition

```
class Sample_Generator(rng_min = float('-inf'), rng_max = float('inf'))
```

Erstellt ein neues Sample_Generator Objekt, optional initialisiert mit der Untergrenze `rng_min` und Obergrenze `rng_max` für die zu erzeugenden virtuellen Samples.

Parameter:

- rng_min : float, optional [Default: -inf]

- rng_max : float, optional [Default: inf]

Methoden Definitionen

```
def run(org_data, syn_data, n_samples, f_id, f_dep_id)
```

Ausführende Funktion, implementiert Generatormethode und wird vom daat Generator aufgerufen.
Parameter:

- org_data : pandas Dataframe
  Original Grunddatensatz

- syn_data : pandas Dataframe
  Synthetischer Datensatz

- n_samples : int
  Anzahl zu generiernder Samples

- f_id : int
  Index des zu generierenden Features

- f_dep_id : int (array)
  Indexes der zu beachtenden Features / Abhängigkeiten

```
def get_val_rng()
```

Getter für rng_min und rng_max. Rückgabewert als Tupel.

```
def check_val_vs_rng(values)
```

prüft erzeugte Featurewerte auf angegebenen Wertebereich und passt sie entsprechend an.
Abgeleitete Generator Klassen

- *Gen_Distribution* - Erzeugt Sample über Distribution
- *Gen_Spline* - Erzeugt Sample über Spline
- *Gen_Recombine* - Erzeugt Sample über zufälligen kNN Wert
- *Gen_Cluster* - Erzeugt Sample über Distribution in Cluster
- *Gen_Next_Mean* - Erzeugt Sample über Durchschnitt der kNN Wert
- *Gen_None* - Platzhalter für keine Erzeugung

2.4 Instruction Klasse

Die INSTRUCTION Klasse kapselt die Instruktions Definitionen in ein Objekt.

Klassen Definition

```
class Instruction(feature, generator, feature_dep)
```

Erstellt ein neues Instruction Objekt, initialisiert mit `feature`, `generator` und optional `feature_dep`.

Parameter:

- feature : str
- generator : Sample_Generator
  Sample Generator Methode
- feature_dep : str (array)
  Liste der zu beachtenden Abhängigkeiten

Methoden Definitionen

Getter Methoden für Parameter

```
def get_feature()
def get_generator()
def get_feature_dep()
```

2.5 Verification Klasse

Klassen Definition

```
class Verification(org_data, syn_data)
```

Attribute:

- cache : int [Default: 200]
  Cache Größe für verwendete SVMs in Mb
- iterations : int [Default: 10000]
  Anzahl der maximalen Lern-Iterationen der SVMs

Parameter:

- org_data : pandas Dataframe

- syn_data : pandas Dataframe

Methoden Definitionen

```
def eval_class_data_set(target_label, train_ratio)
```

Trainiert vier SVMs, zwei je auf train_ratio% und 100% der org_data und syn_data Datensatz. Zeigt Ergebnisse als als Tabellenvergleich an.
Parameter:

- target_label : str
- train_ratio : float

3 Verwendung

Zur Demonstration des DAAT Prototypen beinhaltet das daat Paket das Jupyter Notebook `DAAT_Demostrator.ipynb`. Es zeigt die Verwendung des DAAT exemplarisch anhand des Iris Datensatzes. Zusätzlich ist in diesem Kapitel die allgemeine Verwendung des DAAT unterteilt nach den Bereichen *Analyse*, *Generierung* und *Verification* erklärt.

3.1 Analyse

Für die Datensatzanalyse stellt das DAAT die Analyser Klasse zur Verfügung. Die Datensatzanalyse läuft damit in zwei Schritten ab.

Schritt 1: Initialisierung
Die Analyser Klasse wird zurest mit dem Datensatz `dataset` als pandas Dataframe und optional dem Ziellabel `target` als String initialisiert. Alternativ kann das Ziellabel auch nachträglich über die Funktion `set_target()` neu gesetzt werden.

```
>>> import daat
>>> analyser = daat.Analyser(dataset, 'target')

# optional Ziellabel neu setzen
>>> analyser.set_target('target')
```

Schritt 2: Analysefunktionen aufrufen
Nachdem der Analyser initialisiert ist, können auf ihn alle in 2.1 aufgeführten Funktionen aufgerufen werden, z.B. `plot_feature_info()` (s. Abbildung 1)

```
>>> analyser.plot_feature_info('feature')
```

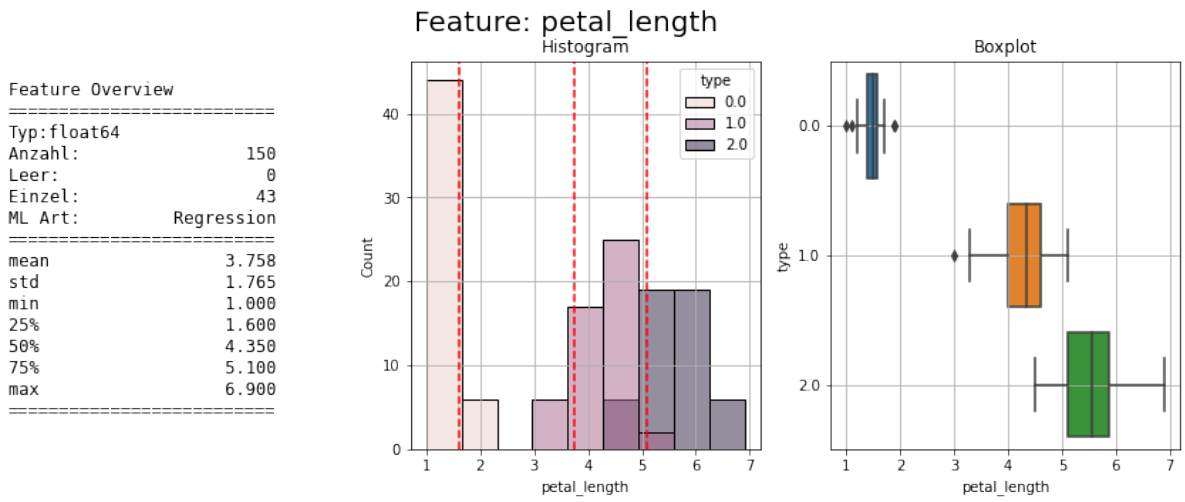

Abbildung 1: Iris show_feature_info() Beispiel

3.2 Generator

Die Verwendung der DAAT Generator läuft in drei Schritten ab.

Schritt 1: Generator Initialiserung
Nach dem Import wird die `Generator` Klasse mit dem Datensatz `dataset` als pandas Dataframe und Ziellabel `target` als String initialisiert. Nachträglich kann das Ziellabel mit der Funktion `set_target()` neu gesetzt werden.

```
>>> import daat
>>> gen = daat.Generator(dataset, "target")
>>> gen.set_target("target")
```

Schritt 2.1: Instruktionen definieren
Nach der Initialisierung werden für jedes Feature Instruktionen angelegt. Diese definieren die zu verwendende Generatormethode und welche anderen Features davon beachtet werden sollen. Hier kann für jedes Feature nur eine Instruktion definiert werden.

```
>>> inst_1 = daat.Instruction('f_1', daat_lib.Gen_Distribution())
>>> inst_2 = daat.Instruction('f_2', daat_lib.Gen_Cluster(3), ['f_1'])
```

Schritt 2.2: Instruktionen hinzufügen
Nun werden die zuvor definierten Instruktionen dem Generator mit der Methode `add_instruction()` in der gewünschten Reihenfolge hinzugefügt. Mit dem Befehl `remove_instruction()` kann eine Instruktion entfernt werden.

```
>>> gen.add_instruction(inst_1)
>>> gen.add_instruction(inst_2)

>>> gen.remove_instruction('f_1')
```

Schritt 2.3: Erzeugungsdefinition anzeigen lassen
Zu jedem Zeitpunkt kann die aktuell definierte Erzeugungspipeline mit dem Befehl `status()` abgerufen und angezeigt werden.

Schritt 3: Datengenerierung und Abruf
Sind für alle Features Instruktionen definiert und zu Generator hinzugefügt worden, startet der Befehl `generate_syn_data()` den Datenerzeugungsprozess und generiert die als Parameter übergebene Anzahl synthetischer Samples. Mit dem Befehl `get_syn_data()` kann der erzeugte Datensatz anschließend abgerufen werden, ohne den Prozess neu starten zu müssen.

```
>>> gen.generate_syn_data(10000)
>>> ds_syn = gen.get_syn_data(combine=True)
```

3.3 Verification

Zur Verifikation des erzeugten Datensatzes behinhaltet das DAAT das Verifications Modul (s. Abschnitt 2.5). Die Verwendung des Verifications Moduls läuft in zwei Schritten ab:
Schritt 1: Initialisierung
Die Verification Klasse wird mit original und synthetischen Datensatz als pandas Dataframe initialisiert.

```
>>> import daat
>>> verify = daat.Verification(og_dataset, syn_dataset)
```

Schritt 2: Testfunktionen aufrufen
Nachdem der Initialisierung können die in Abschnitt 2.5 aufgeführten Funktionen aufgerufen werden.

```
>>> train_ratio = 0.7
>>> verify.eval_class_data_set('target_label, train_ratio)
```

4 Eigene Erweiterung

Durch den modularen Aufbau und Implementierung des DAAT Generators und der darin verwendeten Generatorobjekte, lässt sich dieser mit geringen Aufwand um neue Funktionalitäten und weitere Generatormethoden erweitern. Allgemein läuft die Erweiterung in zwei Schritten ab:

1. SAMPLE_GENERATOR Klasse mit eigener Python Klasse erweitern.
2. Eigene Klasse der Pipeline als Instruktion hinzufügen.

4.1 Neue Sample_Gen Unterklasse erstellen

Das Erstellen einer vom Generator erkannten Unterklasse läuft in den folgenden Schritten ab:

1. Eigene von `Sample_Gen` abgeleitet Klasse erstellen.
2. Den Basisklassen __init__() Konstruktor aufrufen.
3. `run()` Funktion der Basisklasse überladen.
4. Numpy-Array mit den erzeugen Featurewerten zurückgeben.

```
class Gen_X(Sample_Generator):
    def __init__(self, rng_min = None, rng_max = None, << eigene Parameter >>):
        super().__init__(rng_min, rng_max)
        << eigene Werte >>

    def run(self,data,syn_data,n_samples,f_id,f_dep_id):
        result = np.zeros((n_samples))
        f_val = org_data[:, f_id]
        f_min, f_max = self.get_val_rng()

        << eigener Algorithmus / neue Funktionalitaet >>

        result = self.check_vals_vs_rng(result, f_min, f_max)
        return result
```

4.2 Eigene Methode der Pipeline hinzufügen

Nachdem die eigene Methode definiert wurde, kann sie wie die vordefinierten Methoden der Pipeline hinzugefügt werden. Genaueres dazu in Kapitel 3.2 Schritt 2.2.