Laborversuch 1

Versuch	**Nearest-Neighbor-Methoden und Kernel-MLP für Klassifikation**
Fach	Intelligente Lernende Systeme
Semester	WS 2023/24
Fachsemester	TIN5/ITS5/WIN5
Labortermine	09.11.2023 16.11.2023
Abgabe bis spätestens	24.11.2023

Versuchsteilnehmer

Name: Vorname:

Semester: Matrikelnummer:

Bewertung des Versuches

Aufgabe:	1	2	3	4	5 (ZA)
Punkte maximal:	10	15	35	35	15 (ZP)
Punkte erreicht:					

Gesamtpunktezahl: /95 Note: Zeichen:

Anmerkungen:

Allgemeine Vorbemerkungen zu den Praktikumsaufgaben

- Sie sollten mit Python-Grundlagen vertraut sein (z.B. List-Comprehensions, Dicts, Numpy-Arrays, Definition von Funktionen, Definition von Klassen, Vererbung, etc.)
- Aus Zeitgründen wird bei den meisten Praktikums-Aufgaben schon ein Programmgerüst vorgegeben, das Sie nur noch vervollständigen müssen.
- Achten Sie hierbei auf Anweisungen in den Kommentarzeilen, z.B. `# !! INSERT !!` oder `# !! REPLACE !!` !
- Versuchen Sie trotzdem auch die anderen bereits vorgegebenen Progammteile zu verstehen !
- Sie finden die Programmgerüste und Datensammlungen im Vorlesungsverzeichnis: I-Platte, Unterverzeichnis Praktikum/PRAKTIKUMSVERZEICHNIS.
- Die Grobstruktur eines Python-Moduls `myModule.py` können Sie mit dem Shell-Kommando `pydoc myModule` ansehen. Hierbei werden Klassen, Funktionen, Variablen sowie die Docstrings des Moduls ausgegeben sowie deren Docstrings (=Kommentare unter Funktions/Klassennamen mit dreifachen Anführungszeichen ‘‘ ‘‘ ‘‘) angezeigt.
- Es wird (meist) dieselbe Notation wie in der Vorlesung verwendet (siehe Skript) !
- Z.B. bezeichnet `X` die Daten-Matrix `X`, deren Zeilen `X`[i] die Datenvektoren sind. Ähnlich enthalten die Zeilen der Design-Matrix PHI die Merkmalsvektoren PHI[i] und die Zeilen der Zielwerte-Matrix `T` die Zielvektoren bzw. Labels `T`[i].
- `N`, `M` und `D` sind üblicherweise Anzahl der Datenvektoren, Dimension der Merkmalsvektoren und Dimension der Datenvektoren.
- Details zu Python/Numpy-Funktionen finden Sie am einfachsten in den Online-Dokus. Um z.B. Informationen über die Funktion `numpy.argsort(.)` zu finden googeln Sie “numpy argsort”.
- Lesen Sie die Hinweise zu den Aufgaben !

Hinweise zur Abgabe der Ausarbeitungen:

- Erstellen Sie eine elektronische Ausarbeitung (als pdf) welche die Antworten zu allen Praktikumsaufgaben enthält.
- Die Ausarbeitung soll außerdem Snapshots der relevanten Programm-Teile bzw. Grafiken enthalten.
- Bitte laden Sie eine zip-Datei aller notwendigen Dateien auf Ilias hoch.
- Werfen Sie außerdem einen Ausdruck Ihrer Ausarbeitung in das Fach von Prof.Knoblauch.
- Abgabedatum steht jeweils auf dem Deckblatt des Aufgabenblatts.

Aufgabe 1: (4+2+4 = 10 Punkte)

Thema: Erzeugung Gauß-verteilter synthetischer gelabelter Daten für Klassifikation und Darstellung mit Matplotlib und IVISIT

a) Schreiben Sie eine Python-Funktion
`X,T=getGaussData2D(N,mu1,mu2,Sig11,Sig22,Sig12,t=0,C=2,flagOneHot=0)`
welche zweidimensionale Gauß-verteilte gelabelte Daten erzeugt (siehe Skript, Anhang B.8, Gleichung (B.67) und Abb. B.2). Die Funktion soll Datenmatrix $\mathbf{X}$ und Zielwertematrix (oder -vektor) $\mathbf{T}$ erzeugen.

Hinweise:

- Sie können dazu das Programmgerüst `GaussDataGeneration.py` aus dem Vorlesungsverzeichnis vervollständigen. Dort werden auch die Parameter genauer beschrieben.
- Sie können die Numpy-Funktion `np.random.multivariate_normal` zur Daten-Erzeugung verwenden.

b) Testen Sie Ihre Implementierung indem Sie Daten für zwei Klassen mit Parametern

$$\text{Klasse 1:} \quad N = 5, \quad \mu := \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad \mathbf{\Sigma} := \begin{pmatrix} 1 & 0.1 \\ 0.1 & 2 \end{pmatrix}$$

$$\text{Klasse 2:} \quad N = 8, \quad \mu := \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad \mathbf{\Sigma} := \begin{pmatrix} 2 & 0.2 \\ 0.2 & 1 \end{pmatrix}$$

generieren und die Matrizen $\mathbf{X}$ und $\mathbf{T}$ auf dem Bildschirm ausgeben. Verwenden Sie den Seed $s := 13$ für den Zufallszahlengenerator. Schätzen Sie außerdem Mittelwerts-Vektor $\boldsymbol{\mu}$ und Kovarianzmatrix $\mathbf{\Sigma}$ der Gesamtdaten mit (B.66) von Anh. B.8 im Skript.

Hinweise: Zum Setzen des Seeds verwenden Sie `np.random.seed`. Für die Summanden im Kovarianzen-Schätzer von (B.66) können Sie z.B. `np.outer` verwenden.

c) Integrieren Sie Ihre Implementierung in eine IVISIT-Simulation. Spielen Sie etwas herum um ein Verständnis von Mehrdimensionalen Gauss-Verteilungen und den Parametern μ und $\mathbf{\Sigma}$ zu gewinnen. Wählen Sie am Ende Seed $s = 14$ und (Bildschirmfoto machen)

$$\text{Klasse 1:} \quad N = 75, \quad \mu := \begin{pmatrix} -1.0 \\ 2.0 \end{pmatrix}, \quad \mathbf{\Sigma} := \begin{pmatrix} 2 & 0.5 \\ 0.5 & 3 \end{pmatrix}$$

$$\text{Klasse 2:} \quad N = 100, \quad \mu := \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \mathbf{\Sigma} := \begin{pmatrix} 2 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

Hinweise: Für IVISIT siehe Einführung zum Praktikum und Dokumentation auf ILIAS (Tutorials). Sie können das Programmgerüst `ivisit_GaussDataGeneration.py` verwenden. Starten Sie z.B. aus der Kommandozeile mit
`python ivisit_GaussDataGeneration.py ivisit_GaussDataGeneration.db`

Aufgabe 2: (8+4+3 = 15 Punkte)

Thema: Naive Implementierung K-Nearest-Neighbors-Suche für Klassifikation

a) Schreiben Sie eine Python-Funktion `idxKNN = getKNearestNeighbor(X, x, K=1)` welche zum Input-Vektor `x` eine Index-Liste der K Nearest-Neighbors in der Datenmatrix `X` zurückgibt (siehe Kap. 3.3.1 im Skript). Als Abstandsmaß soll die Euklidische Distanz nach (A.14) im Skript verwendet werden.

Hinweise: Vervollständigen Sie das Programmgerüst `KNearestNeighborSearch.py` aus dem Praktikumsverzeichnis. Die Euklidische Distanz ||.|| können Sie entweder "von Hand" oder mit Hilfe der Numpy-Funktion `numpy.linalg.norm(.)` berechnen. Um die Indexe der K Nearest Neighbors zu bestimmen können Sie die Distanzen z.B. mit Hilfe von `numpy.argsort(.)` sortieren.

b) Schreiben Sie dann eine Funktion `P = getClassProbabilities(t,C)` im selben Python-Modul, welche zur Klassen-Label-Liste `t` und Klassenzahl `C` die Klassenverteilung `P` zurückgibt, d.h. `P[c]` soll jeweils die Wahrscheinlichkeit von Klasse $c \in \{0, 1, \ldots, C-1\}$ sein (z.B. $p(c)$ nach (3.7) oder $p(c|\mathbf{x})$ nach (3.8) im Skript).

Hinweise: Zählen Sie einfach für jede Klasse $c = 0, 1, \ldots, C-1$ wie oft sie in der Label-Liste `t` vorkommt, entweder über eine FOR-Schleife oder eleganter mit `np.unique` (mit `return_counts=True`), und dividieren sie jeweils durch die Gesamtzahl der Labels in `t`.

c) Schreiben Sie schließlich eine Funktion `c=classify(P)` welche aus der Klassenverteilung `P` die Klassenentscheidung c nach (3.10) im Skript bestimmt.

Hinweise: Sie den Index der Klasse c mit der maximalen Wahrscheinlichkeit $P(c)$ ermitteln, z.B. mit `np.argmax(P)` oder hier besser über Vergleichen mit `np.max(P)`. Denn falls mehrere Klassen c maximales $P(c)$ haben (`np.argmax(P)` bevorzugt dann immer die Klasse mit dem kleinsten Index), dann soll die Entscheidung besser zufällig unter diesen Klassen ausgewählt werden (mit `np.random.randint`).

d) Testen Sie Ihre Implememententierung mit einer Datenmatrix aus dreidimensionalen Datenvektoren $\mathbf{x}_1 = \begin{pmatrix}1 & 2 & 3\end{pmatrix}^{\mathrm{T}}$, $\mathbf{x}_2 = \begin{pmatrix}-2 & 3 & 4\end{pmatrix}^{\mathrm{T}}$, $\mathbf{x}_3 = \begin{pmatrix}3 & -4 & 5\end{pmatrix}^{\mathrm{T}}$, $\mathbf{x}_4 = \begin{pmatrix}4 & 5 & -6\end{pmatrix}^{\mathrm{T}}$, $\mathbf{x}_5 = \begin{pmatrix}-5 & 6 & 7\end{pmatrix}^{\mathrm{T}}$, $\mathbf{x}_6 = \begin{pmatrix}6 & -7 & 8\end{pmatrix}^{\mathrm{T}}$ und dem Input-Vektor $\mathbf{x} = \begin{pmatrix}3.5 & -4.4 & 5.3\end{pmatrix}^{\mathrm{T}}$ und $K = 3$. Geben Sie alle Euklidischen Abstände zu $\mathbf{x}$ an, die Liste der K Nearest Neighbors, die Klassenverteilung P und die Entscheidung c (Bildschirmfoto).

Hinweise: Siehe Hauptprogramm im Programmgerüst.

e) Die IVISIT-Simulation `ivisit_KNearestNeighborSearch.py` im Praktikumsverzeichnis ist eine Erweiterung von Aufg.1c, bei der ein Input $\mathbf{x}$ per Mausklick auf das Schaubild eingegeben und klassifiziert werden kann. Vervollständigen Sie wieder den Code und testen Sie damit Ihren KNN-Klassifikator:

- Betrachten Sie zunächst den Code: Welche Rollen haben die neuen Funktionen `bind(.)` und `onPressedB1_datvec(.)`?
- Spielen Sie mit der Simulation: Verändern Sie die Datenverteilung, Seeds, K und den Input $\mathbf{x}$. Für welche $\mathbf{x}$ wird die Region der KNN (=türkiser Kreis) sehr groß? Was passiert für sehr große $K \to N$? Was ist der Vorteil von ungeradem K?
- Geben Sie für die Datenverteilung von Aufg.1c für Seed $s = 22$ und $K = 11$ die Resultate (Schaubild/Klassenverteilung/Entscheidung) für $\mathbf{x} \approx (-5; -2)$ an (Bildschirmfoto).

f) Laufzeit/Komplexitätsanalyse: Wie viele Rechenschritte in Abhängigkeit von N :=`len(X)`, D :=`len(x)` und K braucht Ihr Programm ungefähr, um eine Anfrage zu bearbeiten? (Angabe in O-Notation reicht, z.B. $O(N)$ oder $O(N^2)$ etc.). Wie könnte man das Verfahren beschleunigen?

Aufgabe 3: (6+15+5+9 = 35 Punkte)

Thema: Python-Modul für k-Nearest-Neighbor-Klassifikatoren

Da wir im folgenden verschiedene Klassifikations-Verfahren implementieren wollen lohnt es sich ein eigenes Python-Modul dafür zu erstellen (siehe Programmgerüst `Classifier.py`).

a) Versuchen Sie zunächst den Aufbau der Basis-Klasse `Classifier` zu verstehen:

- Betrachten Sie den Aufbau des Moduls durch Eingabe von `pydoc Classifier`. Welche Klassen gehören zu dem Modul und welchen Zweck haben sie jeweils?
- Betrachten Sie nun die Basis-Klasse `Classifier` im Quelltext: Wozu dienen jeweils die Methoden `__init__(self,C)`, `fit(self,X,T)`, `predict(self,x)` und `crossvalidate(self,S,X,T)` ?

b) Versuchen Sie den Code von `crossvalidate(self,S,X,T)` zu verstehen (siehe Skript, Kap. 2.6.4).

- Was enthält die Variable `idxS`? Geben Sie `idxS` z.B. für $N = 9$ und und $S = 3$ an!
- Was enthalten dann die Variablen `idxVal` und `idxTrain` im zweiten Durchlauf der FOR-Schleife?
- Warum werden Aufruf `self.fit(X[perm[idxTrain]],T[perm[idxTrain]])` die Daten- und Zielwertematrix doppelt (über `perm`) indiziert?
- Welche Trainingsdaten `X[n]` werden also in den S Durchläufen der FOR-Schleife jeweils an `self.fit(.)` übergeben falls die Permutation `perm=[3,1,0,8,2,4,7,5,6]` war? Schreiben Sie in Kurzform z.B. `1.Durchlauf: X[3],X[5],...`.
- Was wird als Ergebnis zurückgegeben? Welchen Evaluierungsmaßen im Skript entsprechen sie? Geben Sie die Gleichungsnummern aus dem Skript an (vgl. Kap. 2.6.2).

c) Die von Classifier abgeleitete Klasse KNNClassifier implementiert den **naiven KNN-Klassifikator** von Aufg. 2. Vervollständigen Sie die Funktion `predict(.)`

Hinweis: Importieren Sie `KNearestNeighborSearch.py` von Aufg. 2 und verwenden Sie die dort definierten Funktionen.

d) Die Klasse `FastKNNClassifier` implementiert ähnlich einen **KNN-Klassifikator mit KD-Tree** (siehe Kap. 3.3.2 im Skript). Vervollständigen Sie die Funktionen `fit(.)` und `predict(.)`.

Hinweis: Für den KD-Tree können Sie die SciPy-Klasse `scipy.spatial.KDTree` benutzen (siehe Online-Doku). Zum Bestimmen der Indexe der K Nearest Neighbors zu Input `x` mit dem KD-Tree können Sie dessen Methode `query(x,K)` benutzen, welche die k minimalen Distanzen und die zugehörigen Indexe liefert (siehe Online-Doku).

e) Die `Klasse KernelMLPClassifier` implementiert als **einfaches Neuronales Netz** ein dreischichtiges **Kernel-MLP** (siehe Skript, Kap. 1.3.4, Beispiel MLP3 von Seite 47 und Abb. 1.17C). Vervollständigen Sie die Funktionen `fit(.)` und `predict(.)`. Für die Zielwerte muss man hierbei One-Hot-Vektoren verwenden, und für die Klassentscheidungen wählt man den Index $k \in \{0, 1, \ldots, C-1\}$ des größten Outputs y_k.

Hinweise: Berechnen Sie in `fit(.)` die synaptischen Gewichtsmatrizen $\mathbf{W^z}$ und $\mathbf{W^z}$ nach (1.28) im Skript. Ebenso können Sie in `predict(.)` die Feuerraten `z` und `y` der Hidden-Layer und Output-Layer nach (1.28) im Skript berechnen. Die dafür notwendigen Vektor-Operation in Numpy sind `np.dot` (Matrix-Multiplikation bzw. Matrix-Vektor-Multiplikation), `np.linalg.inv` (Matrix-Invertierung) und `.T` (Matrix/Vektor-Transponierung; z.B. ist `X.T` die Transponierte von `X`).

f) Testen Sie Ihre Implementierungen der drei Klassifikatoren mit den 3D-Datenvektoren von Aufg.2d. Initialisieren Sie den Zufallszahlengenerator mit Seed $s = 20$ und geben Sie jeweils Klassenentscheidung, und A-Posteriori-Klassenverteilung (bzw. beim Kernel die Outputs $\mathbf{y}$) aus. Führen Sie außerdem jeweils eine Kreuzvalidierung mit $S = 2$ durch. Machen Sie ein Bildschirmfoto von der Ausgabe.

Hinweise: Der Modultest ist bereits komplett implementiert. Schauen Sie sich den Code an und versuchen Sie zu verstehen wie die Klassifikatoren benutzt werden. Zur Überprüfung des Kernel-MLP: Für den angegebenen Input `x` sollte der Output `y= [ 3.57588429e-06 -4.88406250e-08 1.00029482e+00]` sein.

g) Die IVISIT-Simulation `ivisit_Classifier.py` im Praktikumsverzeichnis ist eine Erweiterung von Aufg.1 und Aufg.2 zum Testen Ihrer Klassifikatoren und bereits fertig implementiert. Machen Sie sich mit dem Code und den neuen Funktionen vertraut (Code anschauen, Herumspielen) und beantworten Sie die folgenden Fragen:

- Evaluieren Sie KNN und Kernel-MLP bezüglich Klassifikationsleistung (mit Fehlermaß `err` nach (2.102) im Skript für Kreuzvalidierung mit $S = 3$). Ausgangspunkt sind zunächst die Daten von Aufg. 1c. Den Hyperparameter K von KNN können Sie von Hand optimieren, sodass `err` möglichst klein wird.
- Was passiert im Fall vieler Daten (N_1, N_2 groß), falls die Daten gut getrennt sind gegenüber dem Fall wenn die Daten stark überlappen? Warum funktioniert KernelMLP im letzteren Fall nicht so gut?
- Vergleichen Sie für die Datenparameter von Aufg.1c die Laufzeiten der drei Verfahren für Kreuzvalidierung für Datenzahl $N = 10, 100, 1000$ (mit $N_1 = N_2 = N/2$). Tragen Sie die Werte jeweils in ein Schaubild ein.

Zusatzaufgabe 4 4+3+6+2 = 15 Punkte

Thema: k-NN-Klassifikation von Satellitenbilder Japanischer Wälder

Implementieren Sie in einem neuen Python-Skript einen Nearest-Neighbor-Klassifikator zur Klassifikation verschiedener (japanischer) Wald-Typen auf Satellitenbildern. Hierfür enthält die Datensammlung `ForestTypesData.csv` (siehe Praktikumsverzeichnis) $N = 524$ Datensätze, wobei jeder Datensatz aus $D = 27$ Bild-Merkmalen der Satellitenaufnahmen besteht (deren genaue Bedeutung uns hier nicht näher zu interessieren braucht). Zusätzlich enthält jeder Datensatz in der ersten Spalte ein Klassenlabel, welches den Datensatz einer von $C = 4$ Klassen zuordnet. Hierbei bedeuten: 's'=Sugi-Zypressenwald, 'h'=Hinoki-Zypressenwald, 'd'=Mischlaubwald, 'o'=unbewaldet.

Versuchen Sie zunächst den Aufbau des Programmgerüsts `V1A4_ForestClassification.py` zu verstehen:

- Laden der Wald-Daten mit der Pandas-Funktion `read_csv` (oder `read_table`; siehe Pandas-Tutorial).
- Die eingelesenen Daten werden wie üblich in Daten-Matrix `X` und Labels `T` umgewandelt. Beachten Sie hierzu:
  - Nach dem Laden z.B. mit `forestdata = pd.read_csv(filename)` liefert `forestdata.values` ein Daten-Array welches die Daten der Tabelle `ForestTypesData.csv` enthält.
  - Die Klassenlabels stehen in der ersten Spalte von `ForestTypesData.csv`, die Bild-Merkmale in den restlichen 27 Spalten.
  - Sie können die Umwandlung der Text-Label 's','h','d','o' in numerische Klassenlabels 0,1,2,3 etwa mit Hilfe einer List-Comprehension erledigen.

Lösen/beantworten Sie nun die folgenden Fragen:

a) Erstellen Sie einen k-Nearest-Neighbor-Klassifikator für die Wald-Daten. Importieren Sie hierfür wieder das Modul `Classifier.py` aus Aufgabe 3 und verwenden Sie eine geeignete Klasse für den Klassifikator.

b) Testen Sie den Klassifikator für $k = 3$ durch Kreuzvalidierung mit $S = 5$. Geben Sie die Klassifikationsfehlerwahrscheinlichkeit und Verwechslungsmatrix an.

c) Versuchen Sie die Klassifikationsleistung zu optimieren: Führen Sie eine Grid-Search (siehe Skript, Kap. 2.7.1) für $S \in \{1, 2, 3, 5, 10\}$ und $K \in \{1, 3, 5, 7, 9, 11\}$? Was sind die optimalen Hyperparameter?

d) Wiederholen Sie das Experiment für das KernelMLP (statt KNN). Beim Verwenden von $h^z = \tanh$ geht etwas schief. Warum? Was kann man dagegen tun? Versuchen Sie als Alternative auch die folgenden Aktivierungsfunktionen:

$$h^z(a) = \mathrm{sgn}(a) \cdot \sqrt(|a|) h^z(a) = \mathrm{sgn}(a) \cdot \log(1 + |a|) h^z(a) = \mathrm{sgn}(a) \cdot a^2 h^z(a) = a^3$$

Führen Sie wieder eine Grid-Search für S und die genannten Aktivierungsfunktionen h_z durch und finden sie die optimalen Hyperparameter.

e) Theoretische Zusatzfrage: Bis auf k gelten k-NN-Verfahren als parameterfrei, d.h. unabhängig von irgendwelchen Annahmen über die Datenverteilung. Ist diese Annahme wirklich gerechtfertigt?

Hinweise:

- Zur Programmieraufgabe: Sie können das Programmgerüst `ForestClassification.py` aus dem Praktikumsverzeichnis verwenden.
- Zur Theorie-Frage: Was ist z.B. falls sich der Wertebereich verschiedener Merkmalsdimensionen um mehrere Größenordnungen unterscheiden? Wie könnte man hier abhelfen?
- Zu den Aktivierungsfunktionen $h^z(a)$: Verwenden Sie die Numpy-Funktionen `np.sign`, `np.sqrt`, `np.multiply` (Komponentenweise Multiplikation), `np.log`. Wenden Sie die Funktionen immer auf ganze Vektoren bzw. Matrizen an (hier die Gram-Matrix), damit Ihre Implementierung effizient bleibt!