Тема 1

Въведение в проблематиката на изкуствения интелект.

Principal Component Analysis (PCA)

(Анализ чрез разлагане на главни компоненти)

1. Разграничение на изкуствен интелект, алгоритмично учене от данни и дълбоко обучение

1.1. Изкуствен интелект (Artificial Intelligence)

Според (Chollet, 2017) основите на изкуствения интелект са поставени през 50-те години на миналия век, когато учени в областта на компютърните науки си задали въпроса, дали е възможно компютрите да мислят. Бихме могли да опишем това направление като опит за автоматизиране на когнитивни задачи, които обикновено се изпълняват от хора. Това е общо направление, включващо в себе си алгоритмичното учене от данни и дълбокото обучение. Поради тази причина в основата си е набор от инструменти и подходи, на базата на които се създават алгоритми, изпълняващи/решаващи определени задачи. Макар и да са изминали над 70 години, това направление е все още младо и алгоритмите са все още проста репрезентация на когнитивните задачи, изпълнявани от човека. В книгата си „Науките на изкуствения“[1] Хърбърт Саймън казва следното:

> *„Човешките същества, разглеждани като поведенчески системи, са съвсем прости. Привидната сложност на нашето поведение във времето до голяма степен е отражение на сложността на средата, в която се намираме.“*

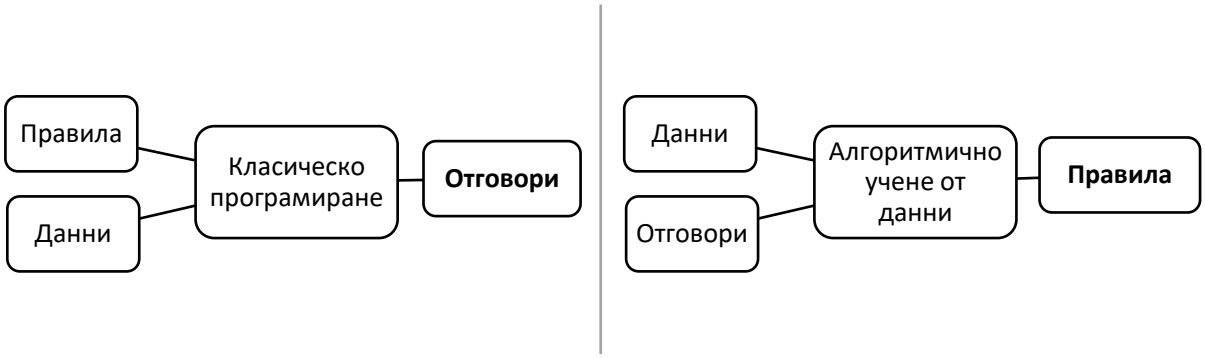
Именно поради тази причина алгоритмите в някои области не предоставят висока точност или точна репрезентация на когнитивната задача. Въпреки това, в области като класификацията на изображения, разпознаване на глас, превод на текст, точността е изключително висока.

[1] Simon, Herbert A. 1996. The Sciences of the Artificial (3rd ed.). Cambridge, MA: MIT Press. ISBN 9780262193740.

Фигура 1. Изкуствен интелект, алгоритмично учене от данни и дълбоко обучение.

1.2. Алгоритмично учене от данни (Machine Learning)

Алгоритмичното учене от данни възниква от въпроси, дали е възможно компютърът да надхвърли това, което ние знаем и можем ли да го накараме да продължи сам да се развива? А също, може ли вместо програмисти да създават правила за обработка на данни, компютърът да научи автоматично тези правила чрез разглеждане на данните? В класическото програмиране, разработчиците създават правилата ръчно, въвеждат данните, които се обработват, съобразно тези правила и в резултат се извежда отговор. При алгоритмичното учене от данни, обаче, се въвеждат както данните, така и отговорите, които се очаква да произлязат от тях и в резултат се получават правилата. След това тези правила могат да бъдат прилагани върху други данни с цел да се получи реален отговор на разглеждан въпрос. Схемите по-долу представят класическата и новата парадигми в програмирането според Шоле (Фигура 2).



Фигура 2. Основни парадигми в програмирането според (Chollet, 2017).

В новата парадигма процесът на получаване на правила се нарича трениране на модела, а не програмиране. Достъпни са множество от примери, свързани с дадена задача. Намира се някаква статистическа структура в тези примери, която впоследствие позволява на модела да дефинира правила за автоматизиране на задачата.

1.3. Дълбоко обучение (Deep Learning)

Алгоритмичното учене от данни е направление в изкуствения интелект. Дълбокото обучение, от своя страна, е специфично направление в алгоритмичното учене от данни, което поставя акцент върху обучението на последователни слоеве. „Дълбокото“ в името на това направление произлиза именно от идеята за изграждане на последователни слоеве на смислени репрезентации на данните. Броя слоеве, използвани за разработване на модела, определят „дълбочината“ му. Всеки слой преобразува данните по определен начин, като с всеки следващ слой, трансформирането става все по-смислено.

Съвременното дълбоко обучение включва десетки, а понякога и стотици последователно свързани слоеве и всички те се учат автоматично с тренировъчното подмножество. През годините дълбокото обучение постигна големи пробиви в класифицирането на изображения, четенето на текст, автономното шофиране, рекламното таргетиране.

2. Кратка история на алгоритмичното учене от данни

През последните години дълбокото обучение достигна до погледа на обикновения човек, а бизнес инвестициите в изкуствения интелект никога не са били по-високи и все пак това не е първата успешна форма на алгоритмичното учене от данни. Може да се каже обаче, че повечето алгоритми за алгоритмично учене от данни, които се прилагат в бизнеса днес, не принадлежат на групата на дълбокото обучение. То не винаги е точният инструмент за работа – понякога няма достатъчно данни, за да може да се приложи даден алгоритъм от тази група, а понякога проблемът може да бъде решен като се приложи различен метод.

В този сегмент ще разгледаме накратко класическите подходи за алгоритмично учене от данни и ще опишем историческия контекст, в който са били развити според Франсоа Шоле (Chollet, 2017). Това ще ни позволи да поставим дълбокото обучение в по-широк контекст на алгоритмичното учене от данни и така ще разберем от къде идва то и защо е от значение за нас.

2.1. Вероятностно моделиране

Вероятностното моделиране е прилагането на принципите на статистиката към анализа на данни. Това е една от най-ранните форми на алгоритмично учене от данни и все още е широко приложима (Miller, 1993). Един от най-известните алгоритми в тази категория е алгоритъмът на Наивния Бейс. Това е вид, базиран на прилагането на теоремата на Бейс, като се приема, че променливите (входните данни) са независими (именно от това предположение идва името на алгоритъма). Тази форма на анализ на данни предхожда компютрите и е била прилагана на ръка десетилетия преди появата на първия компютър (Ji & Yu, 2011).

Тясно свързан модел е логистичната регресия, който се счита за основополагащ в съвременното алгоритмично учене от данни. Това е друг метод за класификация, подобно на описания по-горе, представен отдавна от Д. Р. Кокс, но благодарение на своята многостранна и проста природа, се използва и до днес. Дава интерпретация на относителната важност на променливите за модела. Това, за което логистичната регресия може да ни послужи е да дефинираме т. нар. сигмоиден неврон, тъй като, по същество, този метод представлява едноневронна мрежа (Ezukwoke & Samaneh, 2019).

2.2. Ранни невронни мрежи

Ранните невронни мрежи са напълно изместени от съвременните. Основните идеи се зараждат през 50-те години на миналия век, но минават десетилетия, докато дълбокото обучение се утвърди като важен подход. Били са необходими доста години, докато се открие ефективен начин за трениране на големи и сложни невронни мрежи. Така през средата на 80-те множество учени, без да имат връзка помежду си, преоткриват алгоритъма за обратно разпространение на грешката (Goodfellow & Courville, 2016). Този инструмент служи за трениране на големи вериги от параметрични операции, използващи оптимизационни методи (Nowozin & Wright, 2012).

Първото успешно приложение на невронни мрежи се осъществява през 1989 г. в Лаборатории „Бел“, когато Ян Лекюн комбинира по-ранните идеи за конволюционните невронни мрежи и алгоритъма за обратното разпространение на грешката, за да реши добре познатата днес задача за класифициране на ръчно изписани цифри. Впоследствие получената LeNet мрежа се е използва от Пощенската служба на САЩ през 90-те, с цел автоматизиране на разчитането на пощенските кодове от пликовете (LeCun et al., 1989).

2.3. Методи на ядрото

С времето невронните мрежи започнали да придобиват уважението на изследователите, докато не се появил друг подход на машинното обучение, който ги засенчил – методите на ядрото. Това са група класификационни алгоритми, като най-известния сред тях е Метода на поддържащите вектори[2]. Този метод е разработен от Владимир Вапник и Корина Кортес в началото на 90-те в Лаборатории „Бел“, но е важно да отбележим, че неговото по-старо линейно формулиране е представено още през 1963 г. от Вапник и Алексей Червоненкис. Идеята на Поддържащите вектори е да решават класификационни задачи като намират границата между две групи от точки, разделяйки ги в две категории. На тази граница може да бъде гледано като на линия или повърхност, която разделя данните ни в две пространства, всяко от тях съответстващо на дадена категория. Всичко, от което се нуждаем, за да класифицираме нови точки (новопостъпили данни) е да проверим от коя страна на границата попадат (Иванов & Танов, 2020).

2.4. Дърво на решенията и случайна гора

Дървото на решения е структура, подобна на блок-схема, която позволява класифициране на данни. Предимството му е, че резултатите лесно могат да бъдат интерпретирани и визуализирани (Kamiński et al., 2018). Методът набира популярност още през началото на този век, като след 2010 г. започва да бъде силно предпочитан пред методите на ядрото.

Методът на случайната гора (Breiman, 2001) е стабилен и практичен алгоритъм, който се базира на изграждането на множество дървета, като след това резултатите им се комбинират. Случайните гори са широко разпространени и служат за решаването на голям набор от задачи (Ho, 1995).

2.5. Невронни мрежи

Терминът невронна мрежа е всъщност препратка към невробиологията и някои от основните разработени концепции черпят вдъхновение от разбирането ни за мозъка. Въпреки това, моделите на дълбокото обучение не са модели на мозъка. Все още не са открити доказателства, че мозъкът прилага подобни механизми, както моделите. Това означава, че за разработването на алгоритми не са необходими знания по невробиология.

[2] От английски език: Support Vector Machines (SVM)

Но от друга страна е необходимо да сме наясно с някои математически концепции като тензори и градиенти.

За разлика от силният интерес, изразен към случайните гори и описаните по-горе методи, невронните мрежи били избягвани от учените. Въпреки това, малцина учени, като Йошуа Бенджио, Ян Лекюн и Джефри Хинтън работили върху изграждането на мрежи (LeCun, Bengio & Hinton, 2015).

През 2012 г. групата на Хинтън участва в годишното състезание на Кагъл за класифициране на изображения, наречено ImageNet. По това време класификационната задача била изключително трудна, тъй като множеството от данни се състояло от над милион цветни изображения с високо качество, които могат да бъдат класифицирани в хиляда категории (Deng et al., 2009). Първоначално се използвали класически подходи от компютърното зрение, а моделите били с точност под 75%. Тогава Алекс Крижевски, съветван от Хинтън, успява да постигне точност от 84%. Прилагането на конволюционни невронни мрежи върху множеството ImageNet с всяка година се увеличавало и в резултат класификационната точност нараснала до 96% и поради тази причина задачата се счита за решена (Krizhevsky, Sutskever & Hinton, 2015).

Повече за т. 1 и т. 2 можете да прочете в учебника Deep Learning with Python на Chollet, F. на стр. 3-8 и стр. 14-19.

3. Видове алгоритмично учене от данни

3.1. Контролирано обучение (Supervised Learning)

При контролираното обучение машината се учи от тренировъчни данни. Трениране модела, използвайки тренировъчни данни по такъв начин, че моделът може да генерализира своето учене за непознати данни. Важно е да се отбележи, че данните са обозначени (labeled), т.е. показват на модела точно какво учи. Направлението се нарича „контролирано“, тъй като на тренировъчните данни може да се гледа като на контрольор, който направлява модела при изучаването на дадена задача.

Пример: Нека съпоставим машина с куче, което е на място с множество кошници с различни плодове, а сред тях има и кошница с пържоли. Кошниците представляват данните и техните обозначения. Даваме на кучето да подуши една пържола. В ML това се равнява да подадем на машината данни, обозначени като пържола. Кучето ще започне да търси сред кошниците с плодове именно месото.

3.2. Неконтролирано обучение (Unsupervised Learning)

Характерното за това направление е, че трениране модел чрез тренировъчни данни, които не са обозначени. Целта на такова обучение е да разкрие скрити особености в данните.

Пример: Представете си, че при търсенето на пържолите, кучето е бутнало всички кошници и плодовете са се смесили, т.е. данните са загубили своите обозначения. За да успее да разграничи пържолите от плодовете, кучето трябва да премине през всички обекти (данни) и да пробере само пържолите. Това означава, че то ще трябва да сортира обектите в групи според миризмата им. В ML този процес се нарича клъстеризация (clustering).

3.3. Обучение с утвърждение (Reinforcement Learning)

Обучението с утвърждение е подход в ML, при който интелигентни програми, наричани агенти, съществуват и извършват действия в (не)позната среда, като се адаптират и учат, получавайки точки (награди). Обратната връзка, т.е. точките, може да бъде както положителна (нарича са награда - reward), така и отрицателна (нарича се наказание - punishment). По този начин агентът разбира кои действия, които извършва в средата се приемат като благоприятни и кои като неблагоприятни.

За разлика от ML, в RL:

- не се разглеждат множества от данни
- взаимодействията се извършват със среди, а не с данни и по този начин могат да бъдат описвани сценарии от реалния свят
- използването на среди, вместо на множества от данни предполага настройването на голям набор от параметри
- Средите могат да бъдат 2D или 3D симулации
- Целта на RL е да се постигне конкретна цел
- Наградите се получават от средата

Пример: При обучението с утвърждение, агентът непрекъснато получава обратна връзка от потребителя. Тази обратна връзка е под формата на награди (напр. оценка (положителна или отрицателна), която потребител дава на сериал в Нетфликс). На база тези награди, агентът разбира предпочитанията на потребителя и му предлага подобни сериали (при положителна оценка).

4. Кратко описание на логистичната регресия

Логистичната регресия е метод от групата на контролираното обучение, т.е. ще разполагаме с целевите стойности за трениране на модела под формата на вектори, които са част от тренировъчното подмножество. Нека приемем, че разполагаме с трите вектора X_A = (0.2, 0.5, 0.9, 1), X_B = (0.4, 0.01, 0.5, 0), X_C = (0.3, 1.1, 0.8, 0). Регресията има толкова входове, колкото са и векторите. Фигура 3 показва схематично невронна мрежа, състояща се от един единствен неврон, или с други думи – логистична регресия, като я описваме математически със следните изрази:

$$z = b + w_1x_1 + w_2x_2 + w_3x_3, \quad (1)$$

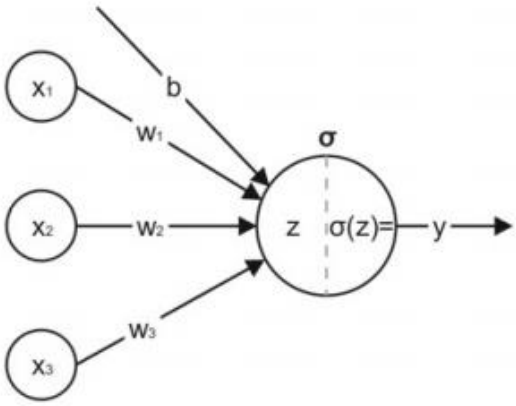
чрез който се изчислява логистичната (сигмоидната) функция:

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (2)$$

След като обединим двата израза получаваме:

$$y = \sigma(b + w_1x_1 + w_2x_2 + w_3x_3) \quad (3)$$

За да пресметнем логистичната функция се нуждаем от грешката b и теглата w_j. В последното уравнение наблюдаваме и параметъра e. Теглата и грешката също са параметри, тъй като техните стойности са неизвестни за нас. Целта на логистичната регресия е да намери или научи такъв вектор от тегла и такава грешка, които заедно да водят до постигането на класификация с висока производителност. Или с други думи обучението на един такъв модел по същество се свързва с търсенето на подходящи стойности за параметрите, които вече дефинирахме като тегла, заемащи всякакви стойности.



Фигура 3. Сигмоиден неврон с три входа и три тегла. Пример на логистичната регресия.

5. Пример 1. Логистична регресия

В този пример ще изградим елементарен класификационен модел като приложим логистичната регресия. Множеството от данни се състои от 20 променливи, генерирани както те, така и стойностите им на случаен принцип.

Зареждаме необходимите библиотеки. Ще работим с numpy и scikit-learn.

```
from numpy import mean

from numpy import std

from sklearn.datasets import make_classification

from sklearn.model_selection import cross_val_score

from sklearn.model_selection import RepeatedStratifiedKFold

from sklearn.linear_model import LogisticRegression
```

Следващата стъпка е да дефинираме множеството от данни, върху което ще приложим логистичната регресия. В X ще се съдържат независимите променливи, а y ще бъде зависимата променлива. Множеството от данни ще се състои от 1000 наблюдения и 20 характеристики (променливи). n_informative представлява броя променливи, които ще послужат при извършването на класификацията. С n_redundant се дефинира броя ненужни променливи. Използваме random_state, за да фиксираме множеството от данни. Ако нямаме

зададена стойност, всеки пък, когато зареждаме множеството, характеристиките ще имат различни стойности.

X, y = make_classification(n_samples=1000, n_features=20, n_informative=10, n_redundant=10, random_state=7)

Дефинираме метода, по който ще извършим класификацията на данните, а именно чрез логистичната регресия.

model = LogisticRegression()

Следва да оценим модела. Това става като първо приложим крос валидация чрез KFold. Ще разгледаме този метод подробно в следващите примери. На този етап от значение е само точността на модела (accuracy). В n_scores ще се запишат всички изчислени скорове, като самото изчисление се осъществява с помощта на финкцията cross_val_scores.

cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)

В случая ще вземем предвид средното и стандартното отклонение от всички изчислени точности чрез функциите mean и std.

print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))

Резултатът, който се отпечатва в конзолата е: 0.824 (0.034).

6. Пример 2. Логистична регресия и PCA

В този пример в допълнение към логистичната регресия ще приложим и един от най-разпространените методи за намаляване на размерността на множествата от данни – Principal Component Analysis.

Зареждаме необходимите библиотеки – numpy и scikit-learn.

from numpy import mean

from numpy import std

from sklearn.datasets import make_classification

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
```

Дефинираме множеството от данни, както направихме в предходния пример.

```
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10, n_redundant=10, random_state=7)
```

Следващата стъпка е да дефинираме pipeline. Целта на pipeline е да събере няколко стъпки, които могат да бъдат валидирани заедно, като могат да им се задават различни параметри. Дефинирането на pipeline става чрез обекта steps, който съдържа в себе си два елемента (две стъпки) – PCA и логистичната регресия. С n_components избираме броя на компонентите на PCA или с други думи, до колко променливи ще бъде сведено множеството от данни.

```
steps = [('pca', PCA(n_components=10)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
```

Оценката на модела се извършва по същия начин, описан в предходния пример.

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

Извеждаме точността на модела.

```
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Резултатът, който се отпечатва в конзолата е: 0.824 (0.034).

7. Пример 3. Линейна регресия и PCA

В този пример ще разгледаме PCA в комбинация с линейна регресия. Множеството от данни съдържа информация за жилищата в Бостън. Наблюденията са 506 на брой, а характеристиките са 14. Последната колона „MEDV" съдържа число, означаващо средната стойност на обитаваните жилища в хиляди долари. Целта е да създадем модел, който успешно да предскаже цената на жилище на база на характеристиките му.

Първоначално са импортирани библиотеките и са заредени данните чрез load_boston функцията.

```
data = load_boston()

boston = pd.DataFrame(data.data, columns=data.feature_names)

boston['MEDV']=data.target
```

Преди да преминем към моделиране, проверяваме корелационната матрица на данните.

```
fig = plt.figure(figsize=(16,12))

ax = fig.add_subplot(111)

sns.heatmap(boston.corr(),annot=True)
```

Фигура 4. Корелационна матрица

Целевата характеристика е високо корелирана с LSTAT и RM, но освен това има и мултиколинеарност между някои от предсказващите характеристики. DIS е високо корелирана с INUDS, INOX и AGE. Това би повлияло върху стабилността на модела.

За да премахнем високите корелации между предсказващите характеристики ще приложим метода на главните компоненти (PCA).

Следва да дефинираме целевата променлива и предсказващите.

y = boston[‘MEDV’]

X = boston.iloc[:, :13]

Оригиналният набор от данни се състои от 12 характеристики. Ще намалим размерността, като приложим анализ на главните компоненти.

Преди това трябва да стандартизираме характеристиките чрез функцията StandardScaler. Така те биват преобразувани в стойност между -1 и 1. Защо?

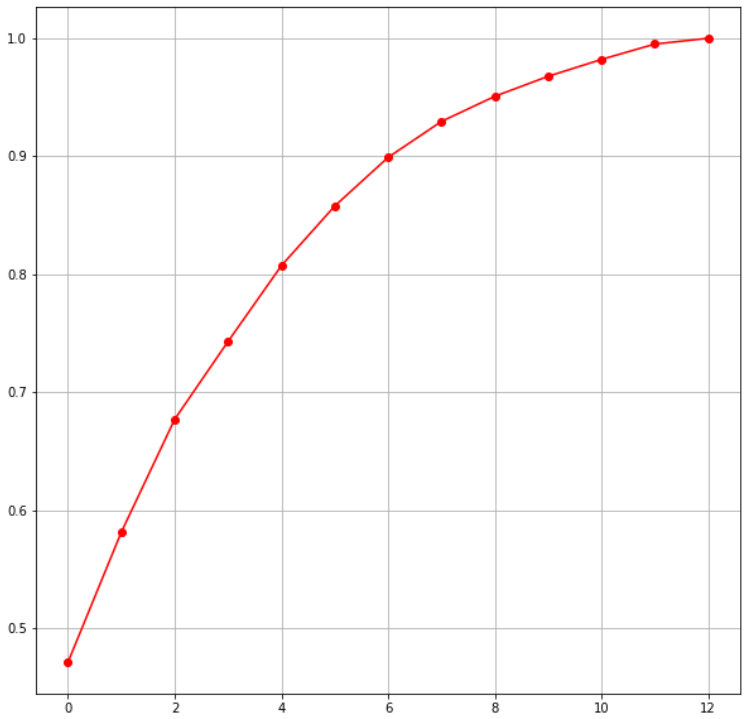
```
X = StandardScaler().fit_transform(X)
```

След това дефинираме обект, в който съхраняваме PCA функцията, след което я прилагаме върху обекта X, съдържащ предсказващите ни характеристики.

```
pca = PCA()

x_pca = pca.fit_transform(X)
```

За да определим оптималния брой компоненти, начертаваме графика, на която се вижда какъв процент от дисперсията се обяснява за съответния брой компоненти (Фиг. 5).

```
plt.figure(figsize=(10,10))

plt.plot(np.cumsum(pca.explained_variance_ratio_), 'ro-')

plt.grid()
```



Фигура 5.

От графиката може да се види, че 6 компонента обясняват близо 90% от дисперсията.

Следва да дефинираме PCA със шест на брой компонента и да го приложим върху предсказващите ни характеристики.

```
pca = PCA(n_components=6)

X = pca.fit_transform(X)
```

Можем да видим дисперсията, която се обяснява за всеки брой компоненти, като изпълним реда:

```
pca.explained_variance_ratio_
```

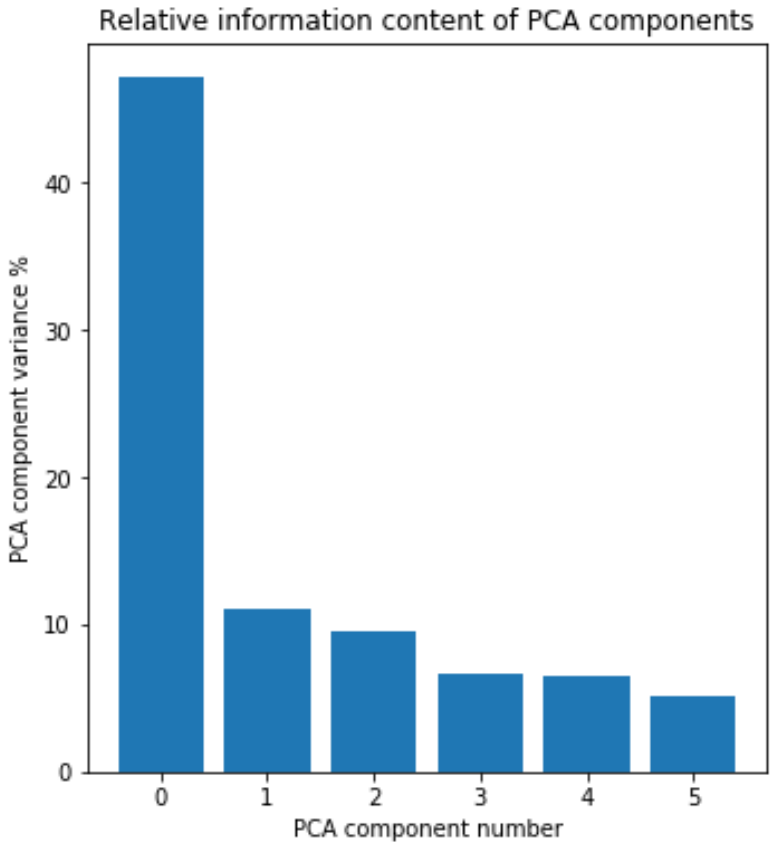
Полученият резултат е:

1	2	3	4	5	6
0.47129606	0.11025193	0.0955859	0.06596732	0.06421661	0.05056978

Сборът от всички е 0.86 или 86%.

Това може да се види и като начертаем графика:

```
fig = plt.figure(figsize=(12,6))

fig.add_subplot(1,2,1)

plt.bar(np.arange(pca.n_components_), 100 * pca.explained_variance_ratio_)

plt.title('Relative information content of PCA components')

plt.xlabel("PCA component number")

plt.ylabel("PCA component variance %")
```



Фигура 6.

Вече можем да преминем към моделиране на данните. Първо разделяме множествата X и y на две части – тренировъчно и тестово, с помощта на процедурата train_test_split от библиотеката sklearn.model_selection. Идеята е върху тренировъчното множество (train set) да се построи моделът, след което да се тества върху тестовото множество (test set). В случая големината на тестовото множество е 30% (test_size=0.3). Разделянето се извършва на случаен принцип.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.30, random_state = 1)
```

Следва създаване на linreg обект, в който се запазва LinearRegression процедурата.

```
linreg = LinearRegression()
```

Следва трениране на модела върху тренировъчното множество и пресмятане на оценка върху тренировъчното и тестовото множество, както и върху цялото множество, и принтираме резултатите.

```
linreg.fit(X_train, y_train)


# model evaluation for training set

linreg.fit(X_train, y_train)

y_train_predict = linreg.predict(X_train)

score = linreg.score(X_train,y_train)

mse = ((mean_squared_error(y_train, y_train_predict)))


print("The model performance for training set")

print("--------------------------------------")

print('Score is {}'.format(score))

print('MSE is {}'.format(mse))

print("\n")


# model evaluation for testing set

score = linreg.score(X_test,y_test)

y_test_predict = linreg.predict(X_test)

mse = ((mean_squared_error(y_test, y_test_predict)))


print("The model performance for testing set")

print("--------------------------------------")

print('Score is {}'.format(score))

print('MSE is {}'.format(mse))
```

```
# model evaluation for the full set

score = linreg.score(X,y)

y_full_predict = linreg.predict(X)

mse = ((mean_squared_error(y, y_full_predict)))


print("The model performance for the full set")

print("--------------------------------------")

print('Score is {}'.format(score))

print('MSE is {}'.format(mse))
```

8. Пример 4. Метод на поддържащите вектори (SVM) и PCA

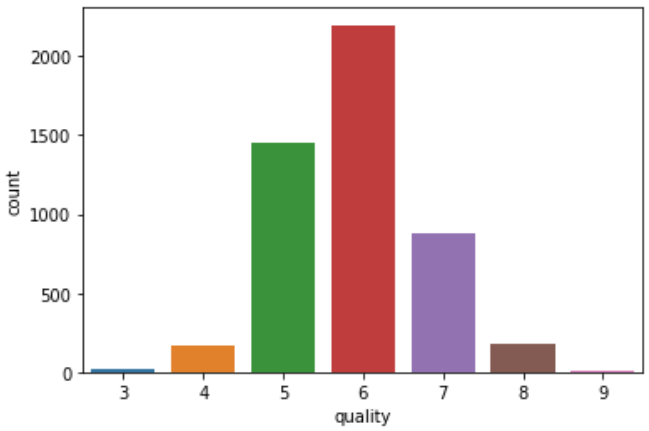
Основната цел на този пример е да се състави класификационен модел, чрез който да се определи принадлежността на всяко едно наблюдение и по-точно, на кой клас принадлежи. Използва се метода на поддържащите вектори в комбинация с ядро с радиална базисна функция (rbf kernel). Валидирането на модела се осъществява с KFold метода. Използва се метода на главните компоненти (Principal Component Analysis), за да се преобразува множеството от наблюдения. Концепцията може да се прилага както за метода на най-близките съседи, така и за този на поддържащите вектори. Крайната цел е да се определи дали моделът е точен и надежден.

Описание на множеството

Множеството, което ще използваме се казва „Wine Quality - White“[3]. Данните са свързани с белия вариант на португалското вино „Vinho Verde“, като са на разположение 4898 наблюдения (проби) и 12 характеристики, базирани на физикохимични тестове.

[3] На разположение на адрес: https://archive.ics.uci.edu/ml/datasets/wine+quality

- Fixed acidity (фиксирана киселинност) – комбинация от киселини, които не се изпаряват лесно или изобщо
- Volatile acidity (летлива киселинност) – количеството оцетна киселина във виното, при твърде високи нива може да доведе до неприятен вкус на оцет
- Citric acid (лимонена киселина) – в малки количества придава свежест и приятен аромат на виното
- Residual sugar (остатъчна захар) – количеството захар, останало след спиране на ферментацията; вина с повече от 45g/l се считат за сладки
- Chlorides (хлориди) – количеството сол във виното
- Free sulfur dioxide (свободен серен диоксид) – в комбинация с бисулфатен йон предотвратява микробния растеж и окисляването на виното
- Total sulfur dioxide (общ серен диоксид) – количеството свободни и свързани форми на SO_2; при високи нива се усеща неприятен мирис
- Density (плътност) – плътността на водата в зависимост от процентното съдържание на алкохол и захар
- pH – описва колко киселинно или основно е виното по скала от 0 (много кисело) до 14 (много основно); повечето вина са между 3-4 по скалата на Ph
- Sulphates (сулфати) – добавка за вино, която може да допринесе за нивата на серен диоксид, която действа като антимикробно средство и антиоксидант
- Alcohol (алкохол) – процентното съдържание на алкохол във виното
- Quality – качеството на виното (оценка от 0 до 10)



Фигура 7. Разпределение на пробите според тяхното качество

При този модел е приложен метода на главните компоненти, за да се редуцира размерността на множеството. С този метод могат да станат видни трендове, нетипични стойности и клъстери. Алгоритъмът започва с импортирането на необходимите библиотеки. Ще са нужни pandas, за да се зареди множеството от данни, а от sklearn ще се използва функцията SCV, тъй като ще се приложи метода на поддържащите вектори.

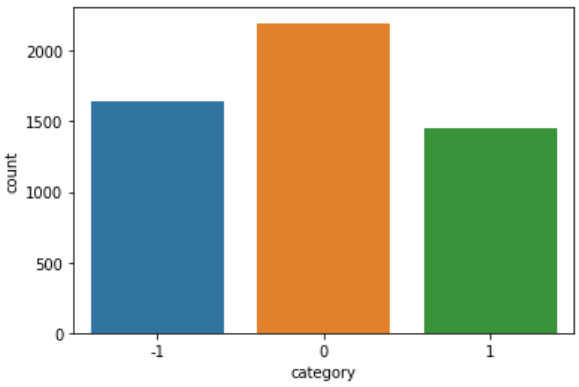
```
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.svm import SVC
```

Следва множеството winequality-white да бъде заредено. След това последната характеристика, quality, е нужно да бъде преобразувана. Това е качеството на виното, като то бива оценено със стойности от 0 до 10. В множеството най-ниската стойност е 3, а най-високата е 9. Преобразуването става като се направи копие на колоната quality и се отдели като самостоятелен вектор и бъде създаден list, наречен category. Тази нова характеристика ще съдържа стойности -1 (ниско качество – оценки до 5), 0 (средно качество – оценка 6) и 1 (високо качество – оценки от 7 до 10). Те бяха разпределени така, тъй като по-голямата част от пробите са с оценка 5 и 6, а е необходимо и към трите класа да принадлежат сходен брой проби. Оценките са разпределени в трите категории чрез функциите if и append.

```
data = pd.read_csv("winequality-white.csv", delimiter = ';')
quality = data["quality"].values
category = []
for num in quality:
    if num<6:
        category.append(-1)
    elif num>6:
        category.append(1)
    else:
        category.append(0)
```

Създава се нов DataFrame с име df, в който е добавена category (целочислен тип) като колона в цялото множество. След това се дефинира променливата y, която съдържа само последната колона – category.

```
category = pd.DataFrame(data=category, columns=["category"]).astype(int)
df = pd.concat([data,category],axis=1)
y = df.iloc[:,12].values
```



Фигура 8. Разпределение на класовете

Дефинираме и Y, който съдържа всички колони без category. Чрез функцията StandardScaler мащабираме данните за всички наблюдения.

```
from sklearn.preprocessing import StandardScaler
Y = df.iloc[:,:12].values
Y = StandardScaler().fit_transform(Y)
```

Прилага се метода на главните компоненти, за да се преобразуват данните. В случая множеството е сведено до четири главни компонента, върху които ще бъде изпълнена предстоящата класификация. Тренира се pca върху цялото множество Y, преобразува се с командата transform и резултата от тези действия бива записан в X. Четирите главни компонента съдържат следните дисперсии:

Първа компонента	0.27888907
Втора компонента	0.13217419

Трета компонента	0.11426434
Четвърта компонента	0.09040513

Фигура 9. Дисперсиите на четирите главни компонента

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 4)
X = pca.fit(Y).transform(Y)
print('explained variance ratio (first four components): %s'
    % str(pca.explained_variance_ratio_))
```

Следва моделът да бъде fit-нат. В модела се използва rbf ядро. След няколко теста беше установено, че моделът има най-висок score коефициент при константа C = 1000.

```
model = SVC(kernel='rbf', C=1000, gamma='auto', decision_function_shape='ovr')
model.fit(X, y)
```

Множеството се дели на 15 равни части с командата Kfold. Целта на kf.split(X) е да раздели множеството X на 15 подмножества, наречени индекси, като 14/15 от тях се третират като индекси за тренировъчното множество, а 1/15 от индексите се запазват за тестовото множество. Разпределението на множеството на тренировъчно и тестово се извършва по такъв начин, че модела да постига максимална стойност на score.

```
from sklearn.model_selection import KFold
n_splits=15 #15
kf = KFold(n_splits=n_splits)
k=0
sm=0
```

Дефинира се цикъл на база променливите train_index и test_index, които взимат стойности от множеството. При всяко повторение на цикъла стойностите на индексите ще се променят последователно. Идеята на този цикъл е моделът да се тренира на множеството X[train_index], y[train_index], като използваме командата fit. После се пресмята score на модела за тестовото множество.

```
for train_index, test_index in kf.split(X):
   model.fit(X[train_index],y[train_index])
   sc_test  = model.score(X[test_index],y[test_index])
```

Използва се цикъл if, чиято цел е да намери онези индекси, чийто тестови score е най-висок. Използва се sm = 0 за сравнение. Индексите, за които е достигнат най-висок score се запазват в train_ maxindex и test_ maxindex.

```
 if sm < sc_test :
      sm=sc_test
      ksm=k
      train_maxindex = train_index
      test_maxindex =  test_index
```

Дефинират се X_train, y_train, X_test, y_test, като те присвояват индексите на данните от най-добре fit-натия модел. Пресмятаме скоровете за както за тренировъчното, така и тестовото множество.

```
X_train = X[train_maxindex]
y_train = y[train_maxindex]
X_test = X[test_maxindex]
y_test = y[test_maxindex]
model.fit(X_train,y_train)
train_score = model.score(X_train, y_train)
test_score  = model.score(X_test, y_test)
y_test_pred = model.predict(X_test)
```

Импортират се confusion matrix и classification report за тестовото и цялото множество.

```
print ('k=',ksm, ',  n_folds=',n_splits  )
```

```
from sklearn.metrics import classification_report, confusion_matrix

## test set score and reports
print ('\n svm.score for the test set :\n ', model.score(X[test_maxindex], y[test_maxindex]) )
print ('\n confussion matrix for the test set :\n',confusion_matrix(y_test, y_test_pred))
print ('\n classification report for the test set :\n',classification_report(y_test,y_test_pred))

## full set score and reports
y_p = model.predict(X)
print ('\n svm.score for the full set :\n ', model.score(X, y) )
print ('\n confussion matrix for the full set :\n',confusion_matrix(y, y_p))
print('\n classification report for the full set :\n',classification_report(y,y_p))
```

Налице са следните две таблици за тестовото множество:

svm.score for the test set: 0.8496932515337423

svm.score показва точността на модела. В този случай за тестовото множество моделът е с точност 84%.

confussion matrix for the test set:

101	11	1
9	147	23
0	5	29

Фигура 10. Confusion matrix на тестовото множество

Целта на тази матрица е да оцени производителността на класификационните модели, като този описан по-горе. Броят правилни и грешни прогнози са обобщени в матрица 3x3, като са разделени по класове. В конкретния случай, моделът е разпознал 326 наблюдения. Когато сумираме стойностите по диагонал, получаваме правилно прогнозираните наблюдения – 277, а грешно прогнозираните са 49. Например, в първи клас (-1 = вина с ниско качество) са разпознати 101 вина като такива с ниско качество, 11 са

разпознати като среднокачествени и 1 е разпознато като висококачествено. Във втори клас (0 = средно качество) 9 вина със средно качество са разпознати като такива с ниско качество, 147 са правилно разпознати и 23 среднокачествени са разпознати като висококачествени. В трети клас (1 = високо качество) 5 висококачествени вина са разпознати като такива със средно качество и 29 вина с високо качество са разпознати правилно.

Елементите на тази матрица дефинират показатели, чрез които можем да оценим ефективността на модела. Показателите са композирани в доклад, наречен classification report, описани по-долу.

classification report for the test set:

	Precision	Recall	F1-score	Support
-1	0.92	0.89	0.91	113
0	0.90	0.82	0.86	179
1	0.55	0.85	0.67	34
Accuracy			**0.85**	**326**
Macro avg.	0.79	0.86	0.81	326
Weighted avg.	0.87	0.85	0.86	326

Фигура 11. Classification report на тестовото множество

• Accuracy – представлява score коефициент при класификационните задачи (= 0.85)

• Precision – отговаря на въпроса каква част от положителните наблюдения са правилно разпознати

- Precision(-1) = 101 / (101 + 9 + 0) = 0.92
- Precision(0) = 147 / (11 + 147 + 5) = 0.90
- Precision(1) = 29 / (1 + 23 + 29) = 0.55

• Recall – неговата стойност показва до каква степен модела разпознава добре даден клас. В конкретния случай означава, че модела е допуснал грешки. Колкото по-близо е recall до 1, толкова по-малко наблюдения са класифицирани грешно.

- Recall(-1) = 101 / (101 + 11 + 1) = 0.89
- Recall(0) = 147 / (9 + 147 + 23) = 0.82

- Recall(1) = 29 / (0 + 5 + 29) = 0.85

• F1-score – това е показател за прецизността на модела, защото съчетава в себе си Precision и Recall показателите.

Резултатите за цялото множество са следните:

svm.score for the full set: 0.8789301755818701

Score показва, че моделът е с точност 88%.

confussion matrix for the full set:

1471	168	1
141	1916	141
3	139	918

Фигура 12. Confusion matrix на цялото множество

Моделът е разпознал 4898 наблюдения. Когато сумираме стойностите по диагонал, получаваме правилно прогнозираните наблюдения – 4305, а грешно прогнозираните са 593. В първи клас са разпознати 1471 нискокачествени вина правилно, 168 са разпознати като среднокачествени и 1 е разпознато като висококачествено. Във втори клас 141 вина със средно качество са разпознати като такива с ниско качество, 1916 са правилно разпознати и 141 среднокачествени са разпознати като висококачествени. В трети клас 139 висококачествени вина са разпознати като такива със средно качество и 918 вина с високо качество са разпознати правилно и 3 вина с високо качество са класифицирани като нискокачествени.

classification report for the full set:

	Precision	Recall	F1-score	Support
-1	0.91	0.90	0.90	1640
0	0.86	0.87	0.87	2198
1	0.87	0.87	0.87	1060

Accuracy			**0.88**	**4898**
Macro avg.	0.88	0.88	0.88	4898
Weighted avg.	0.88	0.88	0.88	4898

Фигура 13. Classification report на цялото множество

Препоръчителна литература:

1. Dimensionality Reduction in Data Science, Garzon, M., **Chapter 1 (p. 1 - 28), Chapter 3 (p. 67 - 77), Chapter 4 (p. 79 - 86)**
2. Deep Learning with Python, Chollet, F., **Chapter 1 (p. 4 - 19)**