JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 24 (2022) 172–175
Received: 21 April 2022
Accepted: 5 May 2022

Learning speed or prediction accuracy? Comparative analysis of programming frameworks for artificial intelligence

Szybkość uczenia czy dokładność predykcji? Analiza porównawcza szkieletów programistycznych do sztucznej inteligencji

Konrad Zdeb*, Piotr Żukiewicz, Edyta Łukasik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The purpose of the article is to analyze frameworks for artificial intelligence applications. In particular, the effectiveness, time-consumption and resources requirement. Linear regression, random forests and k nearest neighbors models were created for each framework. The learning data is a dataset containing informations about diamonds and their prices. Each model was designed to learn diamonds' prices and then make a prediction depending on its specific characteristics such as cut, color, and volume. The learning data was divided into sets of different sizes to show changes in a model depending on the amount of training data. Out of the three machine learning frameworks tested, TensorFlow proved to be the most accurate and SciKit-Learn the fastest.

Keywords: artificial intelligence; data prediction; framework

Streszczenie
Celem artykułu jest analiza szkieletów aplikacji do sztucznej inteligencji. Zbadane zostały: skuteczność, czasochłonność oraz ilość potrzebnych zasobów. Dla każdego frameworka stworzono modele regresji liniowej, lasów losowych i k najbliższych sąsiadów. Dane uczące to zbiory danych zawierające informację o diamencie oraz jego cenie. Każdy model miał za zadanie nauczyć się cen diamentów, a następnie dokonać predykcji w zależności od ich konkretnych cech tj. szlif, kolor, objętość. Dane uczące zostały podzielone na zbiory o różnej wielkości dzięki czemu można było zaobserwować zmianę w modelu w zależności od liczby danych treningowych. Z trzech przebadanych szkieletów programistycznych do uczenia maszynowego TensorFlow wykazał się największą skutecznością, a SciKit-Learn najkrótszym czasem dokonywania predykcji.

Słowa kluczowe: sztuczna inteligencja; predykcja danych; szkielety programistyczne

*Corresponding author
Email address: konrad.zdeb@pollub.edu.pl (K. Zdeb)

1. Wstęp

Uczenie maszynowe to jedna z odnóg sztucznej inteligencji, która cieszy się coraz większym zainteresowaniem [1]. Odpowiednio przygotowane modele potrafią przetworzyć dużą liczbę danych i „nauczyć się" przewidywać wyniki tych pobranych później. Algorytmów wykorzystywanych do uczenia maszynowego jest jednak wiele i nie zawsze wiadomo, który spełni oczekiwania użytkownika. Według autorów publikacji z tej dziedziny, którzy starają się nauczyć czytelnika wykorzystywania sztucznej inteligencji do swoich celów, różne szkielety odznaczają się cechami deklasującymi konkurencję [2] [3]. Trudno jest jednak jednoznacznie stwierdzić, który framework jest najlepszy, a bardzo prawdopodobne jest to, że nie istnieje taki, który jest bezsprzecznie najlepszy i deklasuje konkurencję.

Jednym z najważniejszych parametrów cechujących sztuczną inteligencję jest jej skuteczność. Określa ona, w jakim stopniu można być pewnym, że przewidziany przez algorytm wynik jest bliski prawdzie. W artykule zostały przedstawione trzy badane szkielety mające zastosowanie w sztucznej inteligencji. Są to: SciKit-Learn [4], PyTorch [5] i TensorFlow [6]. Każdy z badanych w artykule algorytmów posiada swój własny sposób liczenia jego trafności. Kolejnym czynnikiem, pod względem którego porównywane są klasyfikatory, jest czasochłonność. W pracy rozróżniono jej dwa rodzaje: czasochłonność uczenia i predykcji. Czasochłonność uczenia określa, w jakim czasie algorytm jest w stanie przetworzyć otrzymane dane oraz wyciągnąć z nich wnioski. Z kolei czasochłonność predykcji oznacza czas potrzebny algorytmowi na przetworzenie danej testującej i wskazanie wyniku na podstawie zebranych wcześniej danych. Podczas wykonywania skomplikowanych obliczeń na ogromnych zestawach danych bardzo ważna jest również kwestia wydajności sprzętowej. Przy porównywaniu algorytmów zwrócono uwagę na potrzebne do działania aplikacji zasoby procesora i pamięci systemowej.

Celem artykułu jest przetestowanie modeli utworzonych na podstawie szkieletów programistycznych do uczenia maszynowego, porównanie i wytypowanie najlepszych pod względem skuteczności, czasochłonności i potrzebnych zasobów. Ma to pomóc w wyborze frameworka najbardziej odpowiadającego potrzebom programisty i postawione-go przed nim zadania.

Przez autorów badań postawione zostały dwie hipotezy badawcze następującej treści:

H1: Model charakteryzujący się największą skutecznością pobiera jednocześnie najwięcej zasobów procesora ze wszystkich porównywanych sztucznych inteligencji.

H2: Żaden z modeli nie przoduje nad innym w każdej z kategorii jednocześnie.

Po przeprowadzeniu badań i na podstawie analizy otrzymanych wyników autorzy pracy wyciągną wnioski o ich prawdziwości lub nie.

2. Metodyka badań

2.1. Stanowisko badawcze

W celu uzyskania jak najbardziej wiarygodnych wyników pomiarowych badanie zostało przeprowadzone w odizolowanym środowisku maszyny wirtualnej. Zostało ono skonfigurowane w następujący sposób:

- na komputerze osobistym zainstalowano oprogramowanie VirtualBox w wersji 6.1.28;
- na maszynie wirtualnej zainstalowano system Windows 11 PRO w wersji 64-bit;
- na wirtualnym systemie zainstalowano Pythona w wersji 3.10.0.

2.2. Przygotowanie danych

Dla każdego szkieletu aplikacji dane były przygotowywane w identyczny sposób. Kod przygotowujący dane został przedstawiony na Listingu 1.

Pierwsze dwie linie oznaczają pobranie zbioru danych z pliku “diamonds_ready.csv”. Następnie usuwana jest kolumna “Unnamed: 0”, gdyż tworzona jest wraz ze zbiorem, a zbiór permutowany jest w celu zmienienia kolejności danych za pomocą metody *shuffle*. Python zapewnia bardzo szybkie i efektywne działanie na zbiorach. Wywołanie metody *iloc* dla zbioru pozwala na elastyczne wybieranie danych. W tym konkretnym przypadku pobierane są rekordy od indeksu 0 do indeksu *records_count*, który przyjmuje wartości 10000, 30000 albo 50000, w zależności od wielkości testowanej próbki. Dzięki wcześniejszemu przestawieniu dane różnią się od tych przy wcześniejszej próbie. Tak przygotowany zbiór należy rozdzielić na dane szukane „y”, w tym przypadku jest to kolumna „price” zawierająca ceny diamentów oraz dane opisujące tę cenę „X” tj. liczba karatów, szlif, kolor itd.

Proces uczenia polega na wprowadzaniu do algorytmu danych „X”. Algorytm następnie zwraca przewidywaną cenę, która porównywana jest z prawidłową wartością zapisaną w zbiorze „y”. Po procesie uczenia należy sprawdzić skuteczność danego modelu na danych, których nie miał okazji przetworzyć czyli na zbiorze testowym. W przypadku opisywanych badań dane zostały rozdzielone w proporcji 80% danych treningowych oraz 20% danych testowych. W ten sposób, dzięki metodzie train_test_split, w prosty sposób można przygotować dane. Przyjmuje ona jako parametr zbiory „X” i „y”, wielkość zbioru sprawdzającego oraz losową wartość całkowitą, która jest wykorzystana przy permutacji elementów.

Każda kolumna w zbiorze opisuje pojedynczą cechę danego diamentu co skutkuje tym, że ich wartości posiadają różne rzędy wielkości. Aby zniwelować błąd, który może sugerować modelowi, że większe wartości są bardziej znaczące niż mniejsze, wykonywany jest proces normalizacji. Klasa, która została wykorzystana w tym kodzie to *StandardScaler*. Najpierw za pomocą metody *fit_transform* obiekt analizuje dane, na których będzie operował, a następnie przeprowadza proces normalizacji na podstawie średniej i odchylenia standardowego. Tak przetworzone dane są gotowe do wykorzystania przez modele.

Listing 1: Przygotowanie danych do nauki

```
diamonds = pd.read_csv(r"C:\Users\piotr\OneDrive\Dokumenty\Studia\Magister
diamonds.drop(['Unnamed: 0'], axis=1, inplace=True)

diamonds = shuffle(diamonds)

diamonds = diamonds.iloc[:records_count, :]

X = diamonds.drop(['price'], axis=1)
y = diamonds['price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=óó)
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test  = sc.transform(X_test)
```

2.3. Proces uczenia modelu

Algorytm przetwarza dane w postaci kroków bądź epok [7]. Każdy krok polega na ciągłym uczeniu i sprawdzaniu czy dana, którą przewidział algorytm jest tą, której szukano. Jeżeli wykryty zostanie błąd model nanosi na siebie poprawki. Po wprowadzonych zmianach program sprawdza czy dla kolejnej danej przewidziany wynik jest poprawny. Proces ten powtarza się ustawioną przez programistę liczbę razy. Im dłużej model się uczy tym lepszą skuteczność powinien osiągnąć.

Na Rysunku 1 przedstawiony został wyświetlany przez Tensorflow proces uczenia.

```
Epoch 1/20
1067/1067 [==============================] - 3s 2ms/step - loss: 1382.2325 - mae: 1382.2325 - val_loss: 517.8335 - val_mae: 517.8335
Epoch 2/20
1067/1067 [==============================] - 2s 2ms/step - loss: 484.5997 - mae: 484.5997 - val_loss: 459.0359 - val_mae: 459.0359
Epoch 3/20
1067/1067 [==============================] - 2s 2ms/step - loss: 438.0830 - mae: 438.0830 - val_loss: 430.9727 - val_mae: 430.9727
Epoch 4/20
1067/1067 [==============================] - 2s 2ms/step - loss: 414.2224 - mae: 414.2224 - val_loss: 418.9761 - val_mae: 418.9761
Epoch 5/20
1067/1067 [==============================] - 2s 2ms/step - loss: 401.5052 - mae: 401.5052 - val_loss: 402.7807 - val_mae: 402.7807
Epoch 6/20
1067/1067 [==============================] - 2s 2ms/step - loss: 391.2015 - mae: 391.2015 - val_loss: 393.2258 - val_mae: 393.2258
Epoch 7/20
1067/1067 [==============================] - 2s 2ms/step - loss: 383.5373 - mae: 383.5373 - val_loss: 400.3570 - val_mae: 400.3570
Epoch 8/20
1067/1067 [==============================] - 2s 2ms/step - loss: 377.5865 - mae: 377.5865 - val_loss: 380.2634 - val_mae: 380.2634
Epoch 9/20
1067/1067 [==============================] - 2s 2ms/step - loss: 370.3160 - mae: 370.3160 - val_loss: 381.7056 - val_mae: 381.7056
Epoch 10/20
1067/1067 [==============================] - 2s 2ms/step - loss: 366.0886 - mae: 366.0886 - val_loss: 376.7699 - val_mae: 376.7699
Epoch 11/20
1067/1067 [==============================] - 2s 2ms/step - loss: 364.3582 - mae: 364.3582 - val_loss: 385.8737 - val_mae: 385.8737
Epoch 12/20
1067/1067 [==============================] - 2s 2ms/step - loss: 360.8728 - mae: 360.8728 - val_loss: 379.8013 - val_mae: 379.8013
Epoch 13/20
1067/1067 [==============================] - 2s 2ms/step - loss: 358.3898 - mae: 358.3898 - val_loss: 363.5269 - val_mae: 363.5269
Epoch 14/20
1067/1067 [==============================] - 2s 2ms/step - loss: 357.0932 - mae: 357.0932 - val_loss: 366.7127 - val_mae: 366.7127
Epoch 15/20
1067/1067 [==============================] - 2s 2ms/step - loss: 352.9424 - mae: 352.9424 - val_loss: 370.2906 - val_mae: 370.2906
Epoch 16/20
1067/1067 [==============================] - 2s 2ms/step - loss: 351.8788 - mae: 351.8788 - val_loss: 364.5327 - val_mae: 364.5327
Epoch 17/20
1067/1067 [==============================] - 2s 2ms/step - loss: 352.8774 - mae: 352.8774 - val_loss: 366.8039 - val_mae: 366.8038
Epoch 18/20
1067/1067 [==============================] - 2s 2ms/step - loss: 349.5326 - mae: 349.5326 - val_loss: 358.6613 - val_mae: 358.6613
Epoch 19/20
1067/1067 [==============================] - 2s 2ms/step - loss: 346.8861 - mae: 346.8861 - val_loss: 367.3414 - val_mae: 367.3414
Epoch 20/20
1067/1067 [==============================] - 2s 2ms/step - loss: 346.2907 - mae: 346.2907 - val_loss: 364.5911 - val_mae: 364.5911
```

Rysunek 1: Przykładowy proces uczenia.

Ten model trenowany był przez 20 epok, a każda z nich zajęła mu od 2 do 3 sekund. Błąd oznaczany jest

przez „mae” (ang. mean absolute error), czyli średni błąd bezwzględny mówiący o tym, jakiego odchylenia od poszukiwanej wartości można się spodziewać. Im mniejszy jest dany błąd tym bardziej dokładny jest model. Przedstawiona na rysunku 1 sztuczna inteligencja zaczęła z bardzo wysokim błędem rzędu 1300, jednak kilka epok później zmniejszyła go do około 300. Zmieniając liczbę epok oraz liczbę rekordów w każdej epoce można usprawniać ten wynik, jednak w tym badaniu użyto najbardziej powszechnych metod i wartości.

2.4. Wyliczanie skuteczności

Skuteczność liczona jest za pomocą udostępnianych przez szkielety aplikacji metod, które przyjmowały jako dane wejściowe zbiór cen, które przewidział model oraz zbiór cen, które powinien przewidzieć (rzeczywistych) [8]. Dla SciKit-Learn i PyTorch wykorzystana została funkcja, która zwraca współczynnik determinacji R2, jako liczbę z zakresu od zera do jeden. Im bliżej jedynki jest dana skuteczność, tym dokładniejszy jest model. Dla TensorFlow wykorzystano funkcję „mean absolute percentage error”, która jest miarą dokładności przewidywania metody prognozowania w statystyce. Ten wariant metody zwraca wartość procentową.

2.5. Wyliczanie zużycia procesora i pamięci

Aby sprawdzić zużycie procesora i pamięci, przed i po procesie uczenia, pobierana jest informacja o aktualnym wykorzystaniu obu tych zasobów. Z uwagi na to, że w tle nie działa żaden inny mocno obciążający system proces, można przyjąć, że każde większe różnice spowodowane są przez badany model. Następnie obie te wartości są porównywane i ustala się stopień zużycia procesora i pamięci komputera.

3. Przebieg badania

Pierwszym krokiem w przeprowadzeniu badania jest trening aplikacji przygotowanymi wcześniej danymi. W tym celu każdy z klasyfikatorów poddano trzem sesjom treningowym; zawierającym 10000, 30000 oraz 50000 danych wejściowych. Algorytmem, który wykorzystano podczas uczenia, był algorytm regresji liniowej. Następnie klasyfikatory poddano próbie, której celem było ustalenie ich skuteczności. Każda aplikacja wykorzystywała własny sposób liczenia skuteczności, charakterystyczny dla użytego szkieletu programistycznego. Dodatkowo, zarówno podczas uczenia, jak i testowania, aplikacje zwracały czas potrzebny im do wykonania danych akcji oraz zużyte zasoby systemowe w postaci pamięci i mocy obliczeniowej procesora.

Model regresji liniowej zakłada, że dla zbioru danych zaobserwowanych $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^{n}$ istnieje liniowa relacja między zmienną zależną y_i a wektorem $p \times 1$ regresorów x_i. Przez uwzględnienie składnika losowego ε_i można zamodelować zależność postaci:

$$y_i = \beta_0 1 + \beta_1 \beta x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = x_i^T \beta + \varepsilon_i, \quad i = 1, \dots, n, \qquad (1)$$

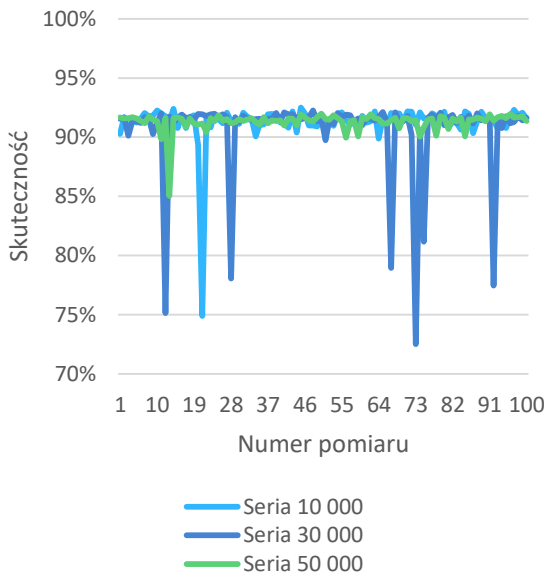
gdzie ^{T} oznacza transpozycję; $x_i^T \beta$, to iloczyn skalarny x_i i β.

Ponieważ algorytm regresji liniowej zawiera w sobie czynnik pseudolosowy, to każdorazowe pobieranie do badania losowej próbki z całości nie ma wpływu na wynik.

4. Wyniki

Podczas analizy statystycznej na danych wyjściowych dokonano działań mających na celu umożliwienie przedstawienia wyników w czytelny i zrozumiały dla czytelnika sposób oraz wyciągnięcie wniosków pozwalających na potwierdzenie bądź obalenie hipotez badawczych.

Badania wykazały, że skuteczność algorytmu SciKit-Learn wynosi średnio 91,09%. Średnia skuteczność algorytmu podczas badania klasyfikatorów wytrenowanych dla 10 000 danych wejściowych wynosi 91,30%, dla 30 000 – 90,67%, a dla 50 000 – 91,31% (Rysunek 2).

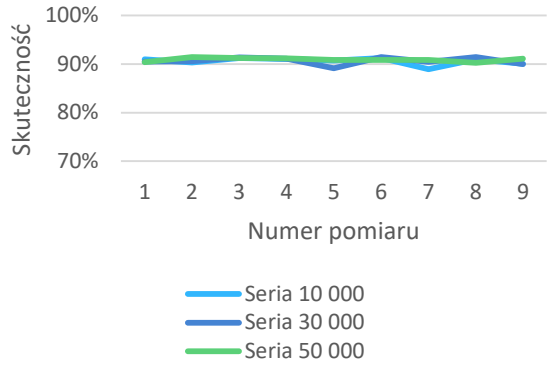


Rysunek 2: Skuteczność SciKit-Learn

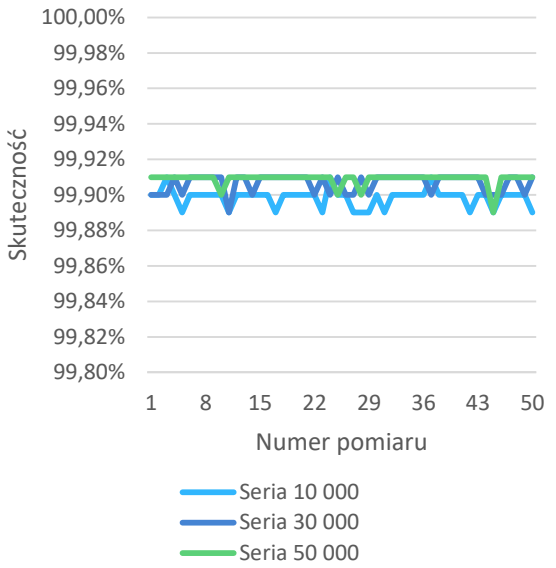
Badania wykazały, że skuteczność algorytmu PyTorch wynosi średnio 90,71%. Średnia skuteczność algorytmu podczas badania klasyfikatorów wytrenowanych dla 10 000 danych wejściowych wynosi 90,60%, dla 30 000 – 90,66%, a dla 50 000 – 90,88% (Rysunek 3).

Badania wykazały, że skuteczność algorytmu TensorFlow wynosi średnio 99,90%. Średnia skuteczność algorytmu podczas badania klasyfikatorów wytrenowanych dla 10 000 danych wejściowych wynosi 90,90% oraz 90,91% dla klasyfikatorów, których liczba danych wejściowych wynosiła 30 000 oraz 50 000 (Rysunek 4).

W Tabeli 1 zestawiono wszystkie wyniki dla poszczególnych używanych w badaniach szkieletów programistycznych.



Rysunek 3: Skuteczność PyTorch.



Rysunek 4: Skuteczność TensorFlow.

Tabela 1: Zestawienie wyników wszystkich algorytmów

	Średnia			
Algorytm	Skuteczność [%]	Czasochłonność [s]	Zużycie procesora [%]	Zużycie pamięci [%]
SciKit-Learn	91,09	0,04	14,64	-0,004
PyTorch	90,71	474,91	-24,22	-0,04
TensorFlow	99,90	68,55	-5,22	0,26

5. Wnioski

Badania wykazały, że największą skutecznością w dokonywaniu predykcji charakteryzuje się szkielet programistyczny TensorFlow o średniej trafności na poziomie 99,90%. Jego średni sumaryczny czas trenowania i dokonywania predykcji wynosi 68,55s. Jest to czas znacznie większy od algorytmu SciKit-Learn, ale ponad 6 krotnie mniejszy od algorytmu PyTorch. Plasuje to algorytm TensorFlow na drugim miejscu pod względem czasochłonności, co dowodzi nieprawdziwości hipotezy badawczej H1: Model charakteryzujący się największą skutecznością potrzebuje jednocześnie najwięcej czasu na wykonanie operacji ze wszystkich porównywanych sztucznych inteligencji.

Badania wykazały, że największą oszczędnością czasu charakteryzuje się szkielet programistyczny SciKit-Learn o średniej sumie czasów trenowania i dokonywania predykcji na poziomie 0,04s (Tabela 1). Jednoczesna dominacja szkieletu TensorFlow w skuteczności dokonywania predykcji dowodzi słuszności hipotezy badawczej H2: Nie istnieje model, który jest najlepszy w każdej kategorii.

W toku prowadzonych badań wyciągnięto wnioski, że poziom zużycia zasobów procesora i pamięci systemowej przez algorytmy jest znikomy. Pomimo upewnienia się, że podczas badania nie pracowały żadne zbędne programy, niezależne od użytkownika procesy systemowe powodowały znaczne wypaczenie pomiarów. Ujemne pomiary zużycia procesora i pamięci systemowej były prawdopodobnie spowodowane zamknięciem działającego w tle procesu systemowego, które zostało odczytane przez algorytm jako zysk, czyli ujemne zużycie.

Podsumowując, z trzech przebadanych szkieletów programistycznych do uczenia maszynowego, pod uwagę warto brać tylko dwa. Algorytm PyTorch wykazał się jednocześnie najmniejszą skutecznością jak i największą czasochłonnością. Użytkownicy oczekujący otrzymania rezultatów natychmiast lub ci, którzy potrzebują przeprocesować ogromną liczbę danych w rozsądnym czasie, powinni wybrać algorytm SciKit-Learn. Z kolei programiści, którym przede wszystkim zależy na bezbłędności dokonywanych szacunków, powinni zdecydować się na algorytm TensorFlow.

Literatura

[1] X. D. Zhang, A Matrix Algebra Approach to Artificial Intelligence, Springer, 2020.

[2] J. Brownlee, Deep learning for computer vision: image classification, object detection, and face recognition in python, Machine Learning Mastery, 2019.

[3] R. Garreta, G. Moncecchi, Learning scikit-learn: machine learning in python, PACKT Publishing Ltd., 2013.

[4] G. Hackeling, Mastering Machine Learning with scikit-learn, PACKT Publishing Ltd., 2017.

[5] E. Stevens, L. Antiga, T. Viehmann, Deep learning with PyTorch, Manning Publications, 2020.

[6] N. McClure, TensorFlow machine learning cookbook, PACKT Publishing Ltd., 2017.

[7] D. Sarkar, R. Bali, T. Sharma, Practical machine learning with Python. A Problem-Solvers Guide To Building Real-World Intelligent Systems, Apress, Berkeley, CA, 2018.

[8] S. Raschka, V. Mirjalili, Python Machine Learning: Machine Learning and Deep Learning with Python. Scikit-Learn, and TensorFlow, Second edition, PACKT Publishing Ltd., 2017.