Zawartość

REGERESJA LINIOWA

Wprowadzenie

Uczenie maszynowe to poddziedzina sztucznej inteligencji, która koncentruje się na opracowywaniu algorytmów i modeli, które umożliwiają komputerom uczenie się i dokonywanie przewidywań lub podejmowanie decyzji na podstawie danych.

Jednym z podstawowych kroków w uczeniu maszynowym jest wstępne przetwarzanie danych, które obejmuje czyszczenie, transformację i normalizację danych, aby nadawały się do analizy. Może to obejmować obsługę brakujących wartości, kodowanie zmiennych kategorycznych i skalowanie cech numerycznych.

Istnieją różne typy algorytmów uczenia maszynowego, w tym uczenie nadzorowane, uczenie nienadzorowane i uczenie przez wzmacnianie. Algorytmy uczenia nadzorowanego uczą się na podstawie oznaczonych przykładów, aby tworzyć przewidywania lub klasyfikować nowe wystąpienia. Z drugiej strony algorytmy uczenia nienadzorowanego mają na celu odkrywanie wzorców lub struktur w nieoznaczonych danych. Algorytmy uczenia przez wzmacnianie uczą się na podstawie interakcji ze środowiskiem, aby zmaksymalizować sygnał nagrody.

Po wybraniu algorytmu dane muszą zostać podzielone na zestawy treningowe i testowe. Zestaw treningowy służy do trenowania modelu, natomiast zestaw testowy służy do oceny jego wydajności na niewidzianych danych.

Wprowadzenie do regresji

Regresja liniowa to prosta, ale skuteczna technika stosowana do przewidywania ciągłej zmiennej zależnej na podstawie jednej lub większej liczby zmiennych niezależnych. Zakłada się, że związek między zmienną zależną a zmienną niezależną (zmiennymi niezależnymi) jest liniowy, co oznacza, że zmianę zmiennej zależnej można wyjaśnić za pomocą liniowej kombinacji zmiennych niezależnych. Celem regresji liniowej jest znalezienie najlepiej dopasowanej linii, która minimalizuje różnicę między wartościami obserwowanymi a wartościami przewidywanymi.

Po dopasowaniu modelu możemy go użyć do tworzenia prognoz na podstawie danych testowych.
Wydajność modelu można ocenić przy użyciu różnych metryk, takich jak średni błąd kwadratowy (MSE) lub R-kwadrat. MSE mierzy średnią kwadratową różnicę między przewidywanymi i rzeczywistymi wartościami, podczas gdy R-kwadrat wskazuje proporcję wariancji zmiennej zależnej, którą można wyjaśnić za pomocą zmiennych niezależnych.

Oprócz regresji liniowej istnieją inne rodzaje technik regresji, takie jak regresja wielomianowa, regresja grzbietowa i regresja lasso. Na przykład regresja wielomianowa może uchwycić relacje nieliniowe poprzez wprowadzenie członów wielomianowych do równania liniowego. Z drugiej strony regresja grzbietowa i regresja lasso to techniki regularyzacji, które pomagają zapobiegać nadmiernemu dopasowaniu poprzez dodanie członu kary do funkcji straty.

Warto zauważyć, że w regresji nie bierzemy pod uwagę relacji między różnymi cechami. Jednak w innych algorytmach uczenia maszynowego, takich jak głębokie uczenie, można badać relacje między atrybutami. W przypadku regresji kluczowa jest prostota, a naszym celem jest wykorzystanie znaczących cech, które mają bezpośredni wpływ na dane. Upraszczając nasze dane i wybierając odpowiednie cechy, możemy poprawić dokładność i wydajność naszego modelu regresji.

Cechy i etykiety regresji

W uczeniu maszynowym regresja jest techniką uczenia nadzorowanego, której celem jest przewidywanie ciągłych wartości liczbowych na podstawie cech wejściowych. Jest szeroko stosowana w różnych dziedzinach, w tym w finansach, opiece zdrowotnej i inżynierii. Modele regresji ustalają związek między zmiennymi niezależnymi (cechami) i zmiennymi zależnymi (etykietami), aby tworzyć prognozy.
Cechy to zmienne wejściowe, które służą do tworzenia prognoz w modelach regresji. Mogą to być wartości liczbowe lub kategoryczne. Cechy liczbowe reprezentują dane ciągłe, takie jak wiek, temperatura lub dochód. Cechy

kategoryczne z kolei reprezentują dane dyskretne i mogą przyjmować ograniczoną liczbę wartości, takich jak płeć lub zawód.

Etykiety, znane również jako zmienne docelowe, to wartości, które chcemy przewidzieć za pomocą modelu regresji. W kontekście regresji etykiety są ciągłymi wartościami liczbowymi. Na przykład w modelu przewidywania cen nieruchomości etykieta może być ceną domu na podstawie jego cech, takich jak rozmiar, liczba sypialni i lokalizacja.

Aby utworzyć model regresji, potrzebujemy zestawu danych, który zawiera zarówno cechy, jak i etykiety. Zestaw danych jest podzielony na dwa podzbiory: zestaw treningowy i zestaw testowy.

Zestaw treningowy służy do trenowania modelu regresji, podczas gdy zestaw testowy służy do oceny jego wydajności. Regresja liniowa jest jednym z najprostszych i najczęściej używanych algorytmów regresji. Zakłada ona liniowy związek między cechami i etykietami. Celem regresji liniowej jest znalezienie najlepiej dopasowanej linii, która minimalizuje różnicę między przewidywanymi wartościami a rzeczywistymi etykietami. Regresja wielomianowa jest rozszerzeniem regresji liniowej, które pozwala na nieliniowe relacje między cechami i etykietami. Dopasowuje funkcję wielomianową do danych, co pozwala uchwycić bardziej złożone wzorce.

Regresja wektorów nośnych (SVR) to algorytm regresji, który wykorzystuje maszyny wektorów nośnych (SVM) do znalezienia linii najlepiej dopasowanej.

Regresja drzewa decyzyjnego buduje model w formie struktury drzewa, gdzie każdy wewnętrzny węzeł reprezentuje cechę, każda gałąź reprezentuje regułę decyzyjną, a każdy węzeł liścia reprezentuje przewidywaną wartość. Drzewa decyzyjne są wszechstronne i mogą obsługiwać zarówno cechy liczbowe, jak i kategoryczne.

Wartości brakujące można imputować za pomocą technik, takich jak imputacja średniej lub imputacja regresji. Skalowanie cech zapewnia, że wszystkie cechy wnoszą równy wkład do modelu regresji, sprowadzając je do podobnej skali. Zmienne kategoryczne są zazwyczaj kodowane za pomocą technik, takich jak kodowanie one-hot lub kodowanie etykiet.

Szkolenie i testowanie regresji

Sztuczna inteligencja - uczenie maszynowe z Pythonem - regresja - szkolenie i testowanie regresji Regresja to podstawowa koncepcja w uczeniu maszynowym, która obejmuje przewidywanie wartości ciągłych na podstawie danych wejściowych. W tym materiale dydaktycznym zbadamy proces szkolenia i testowania regresji przy użyciu Pythona, Omówimy kroki związane ze szkoleniem modelu regresji, oceną jego wydajności i dokonywaniem prognoz na podstawie nowych danych.

1. Przygotowanie danych: Przed trenowaniem modelu regresji ważne jest odpowiednie przygotowanie danych. Obejmuje to czyszczenie danych, obsługę brakujących wartości i transformację zmiennych kategorycznych, jeśli to konieczne. Ponadto dane powinny zostać podzielone na zestawy treningowe i testowe, aby dokładnie ocenić wydajność modelu.

2. Wybór algorytmu regresji: Python oferuje kilka algorytmów regresji, każdy z własnymi mocnymi i słabymi stronami. Wybór algorytmu zależy od natury problemu i charakterystyki danych. Niektóre powszechnie stosowane algorytmy regresji w Pythonie obejmują regresję liniową, regresję wielomianową, regresję wektorów nośnych i regresję drzewa decyzyjnego.

3. Trening modelu regresji: Aby trenować model regresji w Pythonie, musimy dopasować algorytm do danych treningowych. Wiąże się to ze znalezieniem optymalnych wartości parametrów modelu na podstawie cech wejściowych i zmiennej docelowej. Proces treningu ma na celu zminimalizowanie różnicy między wartościami przewidywanymi a rzeczywistymi wartościami w zestawie treningowym.

4. Ocena wydajności modelu: Po wytrenowaniu modelu regresji ważne jest, aby ocenić jego wydajność, aby określić jego dokładność i niezawodność. W tym celu można użyć różnych metryk, takich jak średni błąd kwadratowy (MSE), pierwiastek średniego błędu kwadratowego (RMSE), średni błąd bezwzględny (MAE) i R-kwadrat. Te metryki dostarczają informacji o tym, jak dobrze model pasuje do danych treningowych i jak dobrze generalizuje na nowe dane.

5. Testowanie modelu regresji: Po wytrenowaniu i ocenie modelu można go przetestować na zestawie testowym, aby ocenić jego wydajność na niewidzianych danych. Proces testowania obejmuje wprowadzanie funkcji testowych do wytrenowanego modelu i porównywanie przewidywanych wartości z rzeczywistymi wartościami. Pozwala nam to zmierzyć dokładność predykcyjną modelu i określić, czy jest on nadmiernie dopasowany lub niedopasowany do danych.

6. Tworzenie prognoz: Po wytrenowaniu i przetestowaniu modelu regresji można go użyć do tworzenia prognoz na podstawie nowych, niewidzianych danych. Wprowadzając odpowiednie cechy do modelu, wygeneruje on przewidywane wartości na podstawie wzorców wyuczonych z danych treningowych. Te prognozy można wykorzystać w różnych zastosowaniach, takich jak prognozowanie przyszłych trendów, szacowanie wartości lub podejmowanie świadomych decyzji.

Na początek musimy zdefiniować nasze cechy i etykiety. Cechy, oznaczone jako x, reprezentują zmienne wejściowe, podczas gdy etykiety, oznaczone jako y, reprezentują zmienną wyjściową, którą chcemy przewidzieć. Użyjemy tablicy numpy do przechowywania cech, uzyskanych przez usunięcie kolumny etykiety z naszego zestawu danych. Podobnie, będziemy przechowywać etykiety w tablicy numpy.

Następnie skalujemy nasze funkcje za pomocą funkcji preprocessing.scale. Skalowanie danych jest zazwyczaj wykonywane w celu zapewnienia, że funkcje mieszczą się w określonym zakresie, często między -1 a 1. Może to poprawić dokładność i szybkość przetwarzania. Ważne jest, aby pamiętać, że jeśli planujemy używać klasyfikatora w czasie rzeczywistym na nowych danych, musimy skalować nowe wartości wraz z danymi treningowymi.

W procesie regresyjnego trenowania i testowania w uczeniu maszynowym z Pythonem ważne jest tworzenie zestawów treningowych i testowych. Aby to zrobić, możemy użyć funkcji train test split z biblioteki scikit-learn. Funkcja ta przyjmuje cechy (x) i etykiety (y), a także pożądany rozmiar testu (w tym przypadku 20% lub 0,2). Funkcja tasuje dane, zachowując połączenie cech i etykiet, i wyprowadza dane treningowe i testowe dla x iy.

Po dopasowaniu klasyfikatora ważne jest przetestowanie jego wydajności. Robi się to za pomocą funkcji score(), która jest synonimem testowania. Przekazujemy funkcje testowe (x test) i etykiety (y test), aby uzyskać wynik dokładności. Warto zauważyć, że trenowanie i testowanie na oddzielnych danych jest ważne, aby uniknąć nadmiernego dopasowania, w którym klasyfikator po prostu zapamiętuje dane treningowe i słabo radzi sobie z nowymi danymi.

Można również używać innych algorytmów oprócz regresji liniowej. Na przykład można użyć regresji wektorów nośnych (SVR). Przejście na SVR jest tak proste, jak zmiana klasyfikatora na SVR w kodzie. Należy jednak pamiętać, że wydajność różnych algorytmów może się znacznie różnić. W tym przykładzie SVR działa znacznie gorzej niż regresja liniowa, z wynikiem dokładności wynoszącym zaledwie 0,51. Warto poeksperymentować z różnymi algorytmami i ich parametrami, takimi jak różne jądra, aby poprawić wydajność.

Prognozowanie regresji i przewidywanie

Oprócz regresji liniowej i wielomianowej istnieją inne techniki regresji, które można wykorzystać do prognozowania i przewidywania zadań. Należą do nich między innymi regresja drzewa decyzyjnego, regresja lasu losowego, regresja wektorów nośnych i regresja sieci neuronowych. Każda technika ma swoje zalety i wady, a wybór, której z nich użyć, zależy od konkretnego problemu. Podczas oceny wydajności modelu regresji ważne jest użycie odpowiednich metryk. Powszechnie używane metryki regresji obejmują średni błąd kwadratowy (MSE), pierwiastek średniego błędu kwadratowego (RMSE), średni błąd bezwzględny (MAE) i R-kwadrat. Te metryki dostarczają informacji o tym, jak dobrze model jest w stanie przewidzieć zmienną zależną.

Mamy już pewne nieznane dane, ponieważ prognozujemy przyszłość, konkretnie okres 30 dni. Aby pracować z tymi nieznanymi danymi, musimy zmodyfikować nasze X. Zdefiniujemy nową zmienną o nazwie x lately, która będzie zawierać wartości X dla nieznanego okresu.

Aby to zrobić, ustawimy x lately równe x[-forecast out:], gdzie forecast_out to liczba dni, na które

chcemy prognozować. To da nam wartości X dla nieznanego okresu. Następnie musimy ustalić wartości Mi B w

równaniu **y = Mx + B**. Wykonaliśmy już regresję liniową, aby znaleźć te wartości dla znanych danych. Teraz użyjemy tych wartości, aby przewidzieć wartości dla nieznanych danych.

Aby dokonać przewidywań, musimy utworzyć zestaw prognoz. Użyjemy metody predict klasyfikatora, aby dokonać przewidywań na podstawie danych x lately. Możemy przekazać pojedynczą wartość lub tablicę wartości, aby przewidzieć wiele wartości na raz. W tym przypadku przekażemy tablicę x lately, aby przewidzieć wartości dla całego nieznanego okresu.

Piklowanie i skalowanie

Po wytrenowaniu modelu regresji ważne jest, aby go zapisać do wykorzystania w przyszłości. Pickling to proces w Pythonie, który umożliwia serializację obiektów i zapisanie ich na dysku. Pickling modelu regresji pozwala nam przechowywać go jako plik i ładować później bez konieczności ponownego trenowania. Ta możliwość jest szczególnie przydatna podczas pracy z dużymi zestawami danych lub czasochłonnymi procesami szkoleniowymi.

Aby zamarynować model regresji w Pythonie, najpierw importujemy niezbędne biblioteki, w tym bibliotekę scikit-learn do regresji i moduł pickle do serializacji. Następnie trenujemy model regresji przy użyciu odpowiednich danych i zapisujemy go za pomocą funkcji pickle.dump(). Ta funkcja przyjmuje dwa argumenty: wytrenowany obiekt modelu i obiekt pliku, do którego model zostanie zapisany. Zgodnie z konwencją, rozszerzenie pliku.pkl" jest powszechnie używane w przypadku plików zamarynowanych.

Skalowanie jest kolejnym ważnym krokiem w analizie regresji. Polega na przekształceniu cech wejściowych do standardowej skali, zapewniając, że wszystkie zmienne wnoszą równy wkład do modelu. Skalowanie jest szczególnie ważne w przypadku cech, które mają różne jednostki lub skale. Biblioteka scikit-learn języka Python udostępnia różne techniki skalowania, takie jak StandardScaler i MinMaxScaler, w celu normalizacji wartości cech.

Aby skalować funkcje wejściowe w Pythonje, najpierw importujemy niezbędne biblioteki, w tym bibliotekę scikit-learn do skalowania. Następnie tworzymy obiekt skalowania, dopasowujemy go do danych treningowych i transformujemy zarówno dane treningowe, jak i testowe za pomocą funkcji fit transform(). Zapewnia to, że parametry skalowania nauczone z danych treningowych są stosowane spójnie do danych testowych.

Skalowanie cech wejściowych poprawia wydajność modeli regresji, zapobiegając dominacji zmiennych o większych skalach w procesie uczenia się modelu. Pomaga również w interpretacji współczynników modelu regresji, ponieważ są one teraz w tej samej skali.

Pickling i skalowanie to podstawowe techniki w analizie regresji przy użyciu Pythona. Pickling pozwala nam zapisać wytrenowane modele regresji do wykorzystania w przyszłości, eliminując potrzebę ponownego trenowania. Skalowanie zapewnia, że wszystkie cechy wejściowe są w tej samej skali, co poprawia wydajność i interpretowalność modelu. Wykorzystując bibliotekę scikit-learn Pythona, programiści mogą łatwo wdrożyć te techniki i budować solidne modele regresji.

Zrozumienie regresji

Regresja to fundamentalna koncepcja w uczeniu maszynowym, która obejmuje przewidywanie ciąglej zmiennej wyjściowej na podstawie cech wejściowych. Jest to potężna technika używana do modelowania relacji między zmiennymi i dokonywania przewidywań. W tym materiale dydaktycznym rozważymy zawiłości regresji, skupiając się na jej zrozumieniu i implementacji przy użyciu Pythona.

Analiza regresji jest stosowana, gdy zmienna docelowa, którą chcemy przewidzieć, jest ciągła.

Pomaga nam zrozumieć, w jaki sposób zmienne wejściowe wpływają na zmienną wyjściową, poprzez oszacowanie relacji między nimi. Celem jest zbudowanie modelu, który może dokładnie przewidzieć wartość zmiennej wyjściowej dla nowych danych wejściowych.

Python udostępnia różne biblioteki i narzędzia do analizy regresji, a scikit-learn jest jednym z najpopularniejszych. Scikit-learn oferuje kompleksowy zestaw funkcji i klas do zadań regresji, ułatwiając implementację algorytmów

regresji w Pythonie.

Prosta regresja liniowa jest jedną z najbardziej podstawowych form regresji, w której naszym celem jest modelowanie relacji między pojedynczą zmienną wejściową (znaną również jako zmienna niezależna) a zmienną wyjściową (znaną również jako zmienna zależna). Zakłada się, że relacja jest liniowa, a celem jest znalezienie najlepiej dopasowanej linii, która minimalizuje różnicę między wartościami przewidywanymi a rzeczywistymi.

Multiple Linear Regression rozszerza koncepcję prostej regresji liniowej, biorąc pod uwagę wiele zmiennych wejściowych. Pozwala nam to modelować związek między wieloma zmiennymi niezależnymi a zmienną wyjściową. Cel pozostaje ten sam, tj. znalezienie linii najlepiej dopasowanej, która minimalizuje różnicę między wartościami przewidywanymi a rzeczywistymi.

Regresja wielomianowa to forma analizy regresji, w której związek między zmiennymi wejściowymi a zmienną wyjściową jest modelowany jako wielomian n-tego stopnia. Zapewnia bardziej elastyczne podejście do uchwycenia nieliniowych związków między zmiennymi. Wprowadzając człony wielomianowe, możemy dopasować krzywą do danych zamiast linii prostej.

Techniki regularizacji, takie jak Ridge Regression i Lasso Regression, są używane w celu zapobiegania nadmiernemu dopasowaniu w modelach regresji. Nadmierne dopasowanie występuje, gdy model jest zbyt złożony i przechwytuje szum w danych treningowych, co prowadzi do słabej generalizacji niewidzianych danych. Regularyzacja pomaga kontrolować złożoność modelu, dodając człon kary do funkcji straty, co zachęca do prostszych modeli.

Oprócz wyżej wymienionych technik regresji, dostępne są inne warianty i zaawansowane algorytmy, takie jak Support Vector Regression (SVR), Decision Tree Regression i Random Forest Regression. Każdy algorytm ma swoje mocne i słabe strony, a wybór algorytmu zależy od konkretnego problemu zestawu danych.

Aby zaimplementować regresję w Pythonie, musimy zainstalować niezbędne biblioteki. Najczęściej używanymi bibliotekami do zadań regresji są NumPy, Pandas i scikit-learn. NumPy zapewnia wydajne operacje numeryczne, Pandas oferuje możliwości manipulacji danymi, a scikit-learn zapewnia szeroki zakres algorytmów uczenia maszynowego, w tym regresję.

```
import requests
import pandas as pd
import numpy as np
from sklearn import preprocessing, svm
from sklearn.model_selection import cross_validate,train_test_split
from sklearn.linear_model import LinearRegression
import math,datetime
#import pickle

import matplotlib.pyplot as plt
from matplotlib import style
from matplotlib.pyplot import figure

style.use('ggplot')

#open file to read it content, features, parameters
FilePath = r"C:\Users\Agnieszka\Desktop\EITCA\Files\Animals.csv"
GetFile = open(FilePath)
DataFile = GetFile.read()
#print(DataFile)
GetFile.close()

import chardet

with open(FilePath, 'rb') as fp:
    result = chardet.detect(fp.read())
    #print(result)  # Wyświetli wykryte kodowanie
```

```
df = df.fillna('0.00')
#print(df)

#creating sum colun from 3 others
df = df[['Moose', 'Boars', 'Muflon','Deer','Dama','Fawn']]
df['DeerFamily'] = df['Deer']+df['Fawn']+df['Dama']
df = df[['Boars','DeerFamily']]
#print(df)

forecast_col = 'DeerFamily'
forecast_out = int(math.ceil(0.1*len(df)))
print(forecast_out)

df['label'] = (df[forecast_col].shift(-forecast_out))
df = df.fillna(0)
print(df[:4])
print(df.dtypes)
```

```
    #regression

    X = np.array(df.drop(['label'], axis=1))
    y = np.array(df['label'])
    X = preprocessing.scale(X)
    print(X[:2])
    # #y = np.array(df('label'))

    X_train, X_test, y_train, y_test = (
        train_test_split(X,y,test_size=0.2))

    clf = LinearRegression()
    clf.fit(X_train, y_train)
    #
    # with open('linearregrresion.pickle','wb') as f:
    #     pickle.dump(clf,f)
    #
    # pickle_in = open('linearregrresion.pickle','rb')
    # clf = pickle.load(pickle_in)

    accuracy = clf.score(X_test, y_test)
    print(f"accuracy: {accuracy}")

    X = X[:-forecast_out]
    X_lately = X[-forecast_out:]

    forecast_set = clf.predict(X_lately)

    print(f"forecast_out: {forecast_out}")
    print(f"accuracy: {accuracy}")
    print(f"forecast_set: {forecast_set}")
```

```
    last_value = df.iloc[-1].name
    one_time = 4
    next_value = last_value + one_time

    for i in forecast_set:
        next_value += one_time
        df.loc[next_value] = [np.nan for _ in range(len(df.columns)-1)] + [i]

    print(df)
    df['DeerFamily'].plot()
    df['Forecast'].plot()
    plt.legend(loc=4)
    plt.xlabel('DeerFamily')
    plt.ylabel('Boars')
    plt.show()
```

K-NEAREST (NACHYLENIE NAJLEPSZEGO DOPASOWANIA)

Wstęp

Uczenie maszynowe to gałąź AI, która koncentruje się na rozwijaniu algorytmów zdolnych do uczenia się i podejmowania przewidywań lub decyzji na podstawie da

W kontekście uczenia maszynowego, nachylenie najlepszego dopasowania odnosi się do znalezienia optymalnej linii, która najlepiej pasuje do danego zestawu punktów danych. Linia ta jest określana przez minimalizowanie różnicy między przewidywanymi wartościami a rzeczywistymi wartościami danych. Nachylenie najlepszego dopasowania jest powszechnie stosowane w regresji liniowej, podstawowym algorytmie uczenia maszynowego.

Aby zaprogramować nachylenie najlepszego dopasowania, potrzebujemy zestawu danych, który zawiera cechy wejściowe i odpowiadające im wartości docelowe. Cechy wejściowe reprezentują zmienne niezależne, podczas gdy wartości docelowe reprezentują zmienną zależną, którą chcemy przewidzieć.

Trenowanie modelu
Po załadowaniu zestawu danych możemy przystąpić do jego podziału na zestawy treningowe i testowe. Zestaw treningowy służy do trenowania modelu uczenia maszynowego, natomiast zestaw testowy służy do oceny jego wydajności. Biblioteka scikit-learn udostępnia wygodną funkcję o nazwie „train_test_split" w tym celu. Oto przykładowy fragment kodu:

```
from sklearn.model_selection import train_test_split

# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Następnie możemy utworzyć instancję modelu regresji liniowej i dopasować ją do danych treningowych. Dostęp do modelu regresji liniowej w scikit-learn można uzyskać za pośrednictwem klasy „LinearRegression". Oto przykładowy fragment kodu:

```
from sklearn.linear_model import LinearRegression

# Create an instance of the linear regression model

model = LinearRegression()

# Fit the model to the training data

model.fit(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
```

Po dopasowaniu modelu możemy użyć go do tworzenia prognoz na danych testowych. W tym celu można użyć metody „predict" modelu. Oto przykładowy fragment kodu:

```
# Make predictions on the testing data

y_pred = model.predict(X_test.reshape(-1, 1))
```

Wydajność modelu
Aby ocenić wydajność naszego modelu, możemy obliczyć różne metryki, takie jak średni błąd kwadratowy (MSE) lub współczynnik determinacji (R-kwadrat). Te metryki dostarczają informacji o tym, jak dobrze działa nasz model. Oto przykładowy fragment kodu do obliczenia średniego błędu kwadratowego:

```
from sklearn.metrics import mean_squared_error

# Calculate the mean squared error

mse = mean_squared_error(y_test, y_pred)
```

Na koniec możemy zwizualizować nachylenie najlepszego dopasowania, kreśląc rzeczywiste punkty danych i przewidywaną linię. Biblioteka „matplotlib" zapewnia szeroki zakres funkcji do tworzenia wizualizacji w Pythonie. Oto przykładowy fragment kodu do tworzenia wykresu punktowego z linią najlepszego dopasowania:

```
import matplotlib.pyplot as plt

# Plot the actual data points

plt.scatter(X_test, y_test, color='blue', label='Actual')
```

Na początek przypomnijmy sobie równanie dla linii: y = MX + B. W tym równaniu X oznacza wartości osi x, podczas gdy M i B oznaczają odpowiednio nachylenie i odcinek od osi y. Naszym pierwszym krokiem jest obliczenie nachylenia, M.

Wzór na obliczenie nachylenia jest następujący:

M = ((średnia wartości X * średnia wartości Y) – (średnia wartości X * wartości Y)) / ((średnia wartości X)^2 – średnia z (wartości X^2))

Aby przetłumaczyć tę formułę na kod Pythona, musimy zaimportować funkcję mean z modułu statystycznego, a także moduł numpy jako NP. Użyjemy funkcji mean do obliczenia średnich wartości X i Y.

Następnie definiujemy kilka prostych wartości dla zmiennych X i Y. Na przykład X = [1, 2, 3, 4, 5, 6] i Y = [5, 4, 6, 5, 6, 7]. Możemy wizualizować te dane za pomocą modułu matplotlib, który importujemy jako PLT.

Po wizualizacji danych konwertujemy zmienne X i Y na tablice numpy za pomocą funkcji numpy.array(). Określamy również typ danych jako float64 za pomocą NP.float64.

Teraz zdefiniujmy funkcję, aby obliczyć najlepsze dopasowanie nachylenia. Przekażemy zmienne X i Y jako argumenty i zwrócimy nachylenie, M.

Pierwszym krokiem w obliczaniu nachylenia jest znalezienie średniej wartości X pomnożonej przez średnią wartości Y. Wartość tę przechowujemy w zmiennej M.

Następnie odejmujemy średnią wartości X pomnożoną przez wartości Y od M.

Po uzyskaniu pożądanych wartości możemy przystąpić do obliczenia najlepszego dopasowania nachylenia, oznaczonego jako M. Możemy odjąć średnią kwadratów X od średniej kwadratów X, a następnie podzielić wynik przez średnią kwadratów X pomniejszoną o kwadraty X. Ważne jest, aby pamiętać, że kolejność działań, znana jako PEMDAS (nawiasy, wykładniki, mnożenie, dzielenie, dodawanie, odejmowanie), musi być przestrzegana, aby uzyskać poprawny wynik.

Na koniec musimy obliczyć przecięcie oznaczone jako B. Będzie to tematem naszego następnego samouczka, w którym omówimy regresję liniową. Bądźcie czujni na następny film! Jeśli macie jakieś pytania lub komentarze, śmiało zostawcie je poniżej. Dziękujemy za oglądanie i za Wasze ciągłe wsparcie.

Podsumowując, najlepiej dopasowana linia jest obliczana przy użyciu nachylenia (M) i odcinka y (B). Równanie odcinka y najlepiej dopasowanej linii to B = średnia(Y) – M * średnia(X).

Gdy znamy nachylenie, obliczenie odcinka y jest proste. W naszej funkcji zmodyfikujemy ją tak, aby zawierała zarówno najlepiej dopasowane nachylenie, jak i odcinek. Zdefiniujemy M jak poprzednio i zwrócimy również B. Aby obliczyć B, po prostu ustawimy B równe średniej wartości Y pomniejszonej o M razy średnia wartości X.

Na koniec zwrócimy zarówno M, jak i B. Aby utworzyć linię pasującą do danych, możemy użyć równania y = MX + B. Możemy wygenerować listę wartości Y, używając pętli for z jednym wierszem. Dla każdej wartości X w naszym zestawie danych obliczamy MX + B. Tworzy to linię regresji, która reprezentuje najlepsze dopasowanie do naszych danych.

Teraz omówmy znaczenie linii regresji. Tworząc model dla danych przy użyciu równania y = MX + B, możemy dokonać przewidywań na podstawie modelu. Na przykład, jeśli chcemy przewidzieć wartość Y, gdy X równa się 8, możemy po prostu obliczyć MY + B. Pozwala nam to na dokonanie przewidywań na podstawie naszego modelu.

Możemy nawet zwizualizować przewidywanie, rysując je na wykresie punktowym z kolorem zielonym. W tym momencie udało nam się stworzyć model dla naszych danych i dokonać prognoz na podstawie tego modelu. Ważne jest jednak, aby ocenić dokładność naszej linii najlepszego dopasowania. Chcemy się upewnić, że jest to nie tylko linia najlepszego dopasowania, ale także linia dobrego dopasowania do naszych danych. Aby ocenić dokładność, możemy obliczyć, jak dobrze linia najlepszego dopasowania pasuje do danych. Da nam to wskazówkę co do dokładności linii.

Dokładność i pewność są ściśle powiązane w regresji liniowej, zwłaszcza gdy mamy do czynienia z małą liczbą wymiarów. Dokładność linii najlepiej dopasowanej odnosi się do tego, jak dobrze przewiduje ona zmienną zależną na

podstawie zmiennej niezależnej (zmiennych niezależnych). Mierzy ona bliskość przewidywanych wartości do wartości rzeczywistych. Z drugiej strony pewność w tym kontekście odnosi się do pewności lub niezawodności przewidywań dokonanych przez linię najlepiej dopasowaną. Wskazuje, jak bardzo możemy zaufać przewidywaniom.

Jednak w miarę wzrostu liczby wymiarów, jak w przypadku K najbliższych sąsiadów, różnica między dokładnością a pewnością staje się bardziej wyraźna. W przypadku K najbliższych sąsiadów dokładność odnosi się do tego, jak dobrze algorytm klasyfikuje nowe punkty danych na podstawie ich bliskości do danych treningowych. Pewność z drugiej strony odzwierciedla pewność algorytmu w jego przewidywaniach.

Po wybraniu algorytmu trenujemy model przy użyciu zestawu treningowego. Podczas procesu treningu model uczy się na podstawie danych, dostosowując swoje parametry wewnętrzne w celu zminimalizowania błędu lub zmaksymalizowania dokładności. Proces treningu obejmuje algorytm optymalizacji, który iteracyjnie aktualizuje parametry modelu na podstawie dostarczonych danych.
Po wytrenowaniu modelu oceniamy jego wydajność za pomocą zestawu testowego. Można użyć różnych metryk oceny, w zależności od charakteru problemu. Jedną z powszechnie używanych metryk jest R kwadrat, znany również jako współczynnik determinacji.
R kwadratowe mierzy część wariancji zmiennej zależnej, którą można wyjaśnić za pomocą zmiennych niezależnych. Waha się od 0 do 1, gdzie 0 oznacza, że model nie wyjaśnia żadnej wariancji, a 1 oznacza, że model doskonale wyjaśnia wariancję.

R kwadrat
Matematycznie, R kwadrat jest obliczany jako stosunek sumy kwadratów regresji (SSR) do całkowitej sumy kwadratów (SST). SSR reprezentuje sumę kwadratów różnic między przewidywanymi wartościami a średnią zmiennej zależnej. SST reprezentuje sumę kwadratów różnic między rzeczywistymi wartościami a średnią zmiennej zależnej.
R kwadrat można interpretować jako procent wariancji zmiennej zależnej, który jest uwzględniany przez zmienne niezależne. Wyższa wartość R kwadrat wskazuje na lepsze dopasowanie modelu do danych. Jednak przy ocenie wydajności modelu istotne jest uwzględnienie innych metryk oceny i czynników specyficznych dla danej domeny.
W Pythonie możemy obliczyć R kwadratowe za pomocą biblioteki scikit-learn.

$$r^2 = 1 - \frac{SE\hat{y}}{SE_{\bar{y}}}$$

Jeśli chodzi o programowanie uczenia maszynowego w Pythonie, jedną z podstawowych koncepcji, którą należy zrozumieć, jest metryka R kwadrat (R^2). R^2 to miara statystyczna, która reprezentuje proporcję wariancji zmiennej zależnej, którą można wyjaśnić za pomocą zmiennych niezależnych w modelu regresji. Zakres waha się od 0 do 1, gdzie 1 oznacza idealne dopasowanie, a 0 oznacza brak związku między zmiennymi.

Aby obliczyć R^2 w Pythonie, możemy wykorzystać bibliotekę scikit-learn. Najpierw musimy zaimportować niezbędne moduły:

```
python from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import r2_score
```

Następnie możemy utworzyć obiekt regresji liniowej i dopasować go do naszych danych: python

```
regression_model = LinearRegression()
```

```
regression_model.fit(X, y)
```

Funkcja `r2_score` przyjmuje prawdziwe wartości (`y`) i przewidywane wartości (`y_pred`) jako dane wejściowe i zwraca wynik R^2. Wyższy wynik R^2 wskazuje na lepsze dopasowanie modelu do danych.
Przechodząc do programowania R kwadratowego w R, możemy użyć wbudowanych funkcji i pakietów dostępnych w R. Aby obliczyć R^2, możemy użyć funkcji `lm()` w celu utworzenia modelu regresji liniowej i funkcji `summary()` w celu uzyskania wyniku R^2:
R
model <- lm(y ~ X)
summary(model)$r.squared

Tutaj `y` reprezentuje zmienną zależną, a `X` reprezentuje zmienne niezależne. Funkcja `lm()` dopasowuje model regresji liniowej, a funkcja `summary()` zapewnia podsumowanie modelu, w tym wynik R^2.

Zrozumienie i zaprogramowanie R^2 jest ważne w ocenie wydajności modeli regresji. Pozwala nam ocenić, jak dobrze model pasuje do danych i dostarcza wglądu w moc predykcyjną zmiennych niezależnych.

```
from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Linear Regression model

model = LinearRegression()

# Fit the model on the training data

model.fit(X_train, y_train)

# Make predictions on the testing data

y_pred = model.predict(X_test)

# Calculate R²

R² r_squared = model.score(X_test, y_test)
```

Kroki w uczeniu maszynowym

1. **Pierwszym krokiem** w programowaniu modelu uczenia maszynowego jest zdefiniowanie problemu i zebranie niezbędnych danych. Obejmuje to zrozumienie domeny problemu, zidentyfikowanie interesujących zmiennych i zebranie lub wygenerowanie odpowiedniego zestawu danych. Ważne jest, aby upewnić się, że zestaw danych jest reprezentatywny dla danego problemu i zawiera wystarczającą liczbę próbek, aby skutecznie trenować i oceniać model.

2. **Po przygotowaniu zestawu danych następnym krokiem** jest wstępne przetworzenie danych. Zazwyczaj obejmuje to zadania takie jak obsługa brakujących wartości, skalowanie cech numerycznych, kodowanie zmiennych kategorycznych i dzielenie zestawu danych na zestawy treningowe i testowe. Wstępne przetwarzanie jest niezbędne, aby upewnić się, że dane mają odpowiedni format do trenowania modelu uczenia maszynowego.

3. **Po wstępnym przetworzeniu danych następnym krokiem jest wybór odpowiedniego algorytmu uczenia maszynowego dla danego zadania**. Wybór ten zależy od charakteru problemu, dostępnych danych i pożądanego wyniku. Niektóre typowe algorytmy uczenia maszynowego obejmują regresję liniową, regresję logistyczną, drzewa decyzyjne, maszyny wektorów nośnych i sieci neuronowe. Ważne jest zrozumienie mocnych i słabych stron każdego algorytmu, aby podjąć świadomą decyzję.

4. **Po wybraniu algorytmu następnym krokiem jest trenowanie modelu przy użyciu zestawu danych treningowych.** Obejmuje to dopasowanie modelu do danych i dostosowanie jego parametrów w celu zminimalizowania błędu lub zmaksymalizowania metryki wydajności. Proces trenowania ma na celu uchwycenie

podstawowych wzorców i relacji w danych, umożliwiając modelowi dokonywanie dokładnych przewidywań lub decyzji.

5. **Po wytrenowaniu modelu, istotne jest, aby ocenić jego wydajność przy użyciu zestawu danych testowych**. Ten krok pomaga ocenić, jak dobrze model generalizuje do niewidzianych danych i dostarcza wglądu w jego możliwości predykcyjne. Różne metryki oceny mogą być używane w zależności od typu problemu, takie jak dokładność, precyzja, odwołanie, wynik F1 lub średni błąd kwadratowy. Ważne jest, aby interpretować te metryki w kontekście problemu i porównywać wydajność modelu z podejściami bazowymi lub najnowocześniejszymi.

Testowanie założeń
Testowanie założeń jest krytycznym aspektem programowania modeli uczenia maszynowego. Założenia są przyjmowane podczas procesu modelowania w celu uproszczenia problemu lub uczynienia pewnych przewidywań wykonalnymi.

Oprócz testowania założeń, istotne jest również sprawdzenie wydajności modelu na różnych zestawach danych lub za pomocą technik walidacji krzyżowej. Pomaga to ocenić solidność modelu i możliwości generalizacji. Walidacja krzyżowa obejmuje podział danych na wiele podzbiorów, szkolenie i ocenę modelu na różnych kombinacjach tych podzbiorów oraz agregację wyników. Ta technika zapewnia bardziej wiarygodną ocenę wydajności modelu w porównaniu z pojedynczym podziałem treningu i testu.

Testowanie założeń jest ważnym aspektem tego procesu, zapewniając, że założenia modelu są zgodne z domeną problemu. Ważne jest również sprawdzenie wydajności modelu przy użyciu odpowiednich technik

Klasyfikacja jest podstawowym zadaniem w uczeniu maszynowym, które obejmuje kategoryzację danych do wstępnie zdefiniowanych klas lub kategorii na podstawie ich cech. Jest szeroko stosowana w różnych domenach, takich jak rozpoznawanie obrazów, filtrowanie spamu, analiza sentymentów i diagnostyka medyczna.

Aby zrozumieć KNN, rozważmy prosty przykład klasyfikacji owoców na podstawie ich koloru i rozmiaru. Załóżmy, że mamy zbiór danych różnych owoców, z których każdy jest reprezentowany przez kolor (czerwony, żółty lub zielony) i rozmiar (mały, średni lub duży). Naszym celem jest sklasyfikowanie nowego owocu na podstawie jego koloru i rozmiaru. KNN działa poprzez obliczenie odległości między nowym wystąpieniem a wszystkimi wystąpieniami treningowymi. Używana metryka odległości może się różnić, ale powszechnie używane metryki obejmują odległość euklidesową i odległość Manhattanu. Następnie K najbliższych sąsiadów jest określanych przez wybranie K wystąpień o najmniejszych odległościach do nowego wystąpienia.

Wartość K jest ważnym parametrem w KNN i powinna być ostrożnie wybierana. Mniejsza wartość K może skutkować bardziej elastyczną granicą decyzyjną, ale jest bardziej podatna na szum i wartości odstające. Z drugiej strony, większa wartość K może prowadzić do płynniejszej granicy decyzyjnej, ale może potencjalnie błędnie klasyfikować instancje z różnych klas, które są blisko siebie.
Aby zaimplementować KNN w Pythonie, możemy wykorzystać bibliotekę scikit-learn. Następnie musimy wstępnie przetworzyć dane, dzieląc je na zestawy treningowe i testowe. Zestaw treningowy jest używany do zbudowania modelu KNN, podczas gdy zestaw testowy jest używany do oceny jego wydajności.

Po wstępnym przetworzeniu danych możemy utworzyć instancję klasy KNeighborsClassifier z biblioteki scikit-learn. Następnie możemy dopasować model do danych treningowych za pomocą metody fit(). Po wytrenowaniu modelu możemy dokonać przewidywań na podstawie danych testowych za pomocą metody predict(). Dokładność modelu można ocenić, porównując przewidywane etykiety z prawdziwymi etykietami z danych testowych.

K najbliższych sąsiadów (K-Nearest-Neighbors) to prosty i intuicyjny algorytm używany do zadań klasyfikacyjnych w uczeniu maszynowym. Określa klasę nowej instancji poprzez porównanie jej cech z K najbliższych sąsiadów w danych treningowych. Python, dzięki swoim rozbudowanym bibliotekom i narzędziom, zapewnia wygodną platformę do implementacji KNN i innych algorytmów uczenia maszynowego. Zrozumienie koncepcji i technik stojących za KNN jest ważne dla budowania bardziej złożonych i dokładnych modeli klasyfikacji.

Algorytm KNN
Algorytm KNN jest nieparametryczną metodą stosowaną do zadań klasyfikacji i regresji. Jest to prosty, ale skuteczny algorytm, który opiera się na zasadzie podobieństwa. Biorąc pod uwagę nowy punkt danych, KNN znajduje K

najbliższych sąsiadów w zestawie danych treningowych i przypisuje etykietę lub wartość na podstawie większości głosów lub średniej etykiet lub wartości sąsiadów.

1. Zaimportuj niezbędne biblioteki:

from sklearn.neighbors import KNeighborsClassifier

2. Przygotuj dane: – Załaduj zbiór danych do tablicy Pandas DataFrame lub NumPy. – Oddziel cechy (zmienne wejściowe) i zmienną docelową (zmienną wyjściową). – Podziel zbiór danych na zbiór treningowy i testowy.

3. Utwórz instancję klasyfikatora KNN:
python
knn = KNeighborsClassifier(n_neighbors=K)

4. Przeszkol klasyfikator, korzystając z danych treningowych:
knn.fit(X_train, y_train)

5. Przewiduj etykiety lub wartości dla danych testowych:
y_pred = knn.predict(X_test)

6. Oceń wydajność klasyfikatora:
Oblicz dokładność, precyzję, odwołanie lub inne istotne wskaźniki. – Porównaj przewidywane etykiety lub wartości z prawdziwymi etykietami lub wartościami.

7. Dopracuj algorytm:

Eksperymentuj z różnymi wartościami K, aby znaleźć optymalną liczbę sąsiadów. – Eksperymentuj z innymi hiperparametrami lub technikami, aby poprawić wydajność modelu.

Warto zauważyć, że KNN jest wszechstronnym algorytmem, który można wykorzystać zarówno do problemów klasyfikacji, jak i regresji. Może się jednak nie nadawać do dużych zbiorów danych ze względu na wysoką złożoność obliczeniową.

Po wstępnym przetworzeniu danych musimy zdefiniować nasze cechy (X) i etykiety (Y). Cechy obejmują wszystkie kolumny z wyjątkiem kolumny klasy, podczas gdy etykiety są tylko kolumną klasy. Przechowujemy je w tablicach numpy za pomocą funkcji `NP.array`.

Aby ocenić wydajność naszego modelu, przeprowadzamy walidację krzyżową. Dzielimy dane na zestawy treningowe i testowe, używając funkcji `cross_validation.train_test_split` z scikit-learn. Określając rozmiar testu na 0,2 (20%), 80% danych jest używanych do treningu, a 20% do testowania.
To kończy wstępne kroki programowania uczenia maszynowego w Pythonie, w szczególności aplikacji K najbliższych sąsiadów. Zajęliśmy się brakującymi danymi, usunęliśmy bezużyteczne dane, zdefiniowaliśmy nasze funkcje i etykiety oraz przeprowadziliśmy walidację krzyżową, aby przygotować się do trenowania i testowania naszego modelu.

KNN to algorytm klasyfikacji, który przewiduje klasę punktu danych na podstawie klas jego najbliższych sąsiadów. Na początek musimy zdefiniować klasyfikator. W tym przypadku użyjemy klasy KNeighborsClassifier z biblioteki scikit-learn. Możemy utworzyć instancję klasyfikatora, przypisując ją do zmiennej, tak jak poniżej:
classifier = KNeighborsClassifier()

Następnie musimy dopasować klasyfikator do naszych danych treningowych. Można to zrobić za pomocą metody `fit`, która przyjmuje cechy (X_train) i odpowiadające im etykiety (y_train). Kod dopasowujący klasyfikator wyglądałby następująco:

classifier.fit(X_train, y_train)
Po dopasowaniu klasyfikatora możemy ocenić jego wydajność, testując go na danych testowych. Można to zrobić za pomocą metody `score`, która oblicza dokładność klasyfikatora na danych testowych. Kod do obliczania dokładności wyglądałby następująco:

```
accuracy = classifier.score(X_test, y_test)
```

Dokładność jest miarą tego, jak dobrze klasyfikator działa, przy czym wyższe wartości oznaczają lepszą wydajność. W tym przypadku używamy terminu „dokładność" zamiast „pewność", aby zmierzyć wydajność klasyfikatora. Ważne jest, aby zauważyć, że KNN ma również koncepcję pewności, która jest związana z mechanizmem głosowania używanym w algorytmie.

Na koniec, gdy klasyfikator jest już wytrenowany, możemy go użyć do tworzenia prognoz na nowych, niewidzianych punktach danych. Aby to zrobić, możemy użyć metody `predict`, która przyjmuje cechy nowego punktu danych i zwraca przewidywaną klasę. Kod do tworzenia prognozy wyglądałby następująco:

```
prediction = classifier.predict(example_measures)
```

Warto wspomnieć, że w niektórych przypadkach powszechne jest zapisywanie wytrenowanego klasyfikatora do pliku za pomocą modułu pickle. Pozwala to na ponowne użycie klasyfikatora bez konieczności ponownego trenowania.

Teraz zajmijmy się sytuacją, w której mamy nieznaną liczbę przewidywań. Na przykład w jednym tygodniu możemy mieć 15 pacjentów, a w następnym 45. Aby poradzić sobie z tą zmiennością, musimy dynamicznie przekształcać dane bez zakodowania kształtu na stałe za każdym razem.
Aby to osiągnąć, możemy przekonwertować listę list na tablicę numpy. Gdy już mamy tablicę numpy, możemy użyć funkcji reshape. Zamiast określać stałą liczbę, możemy zastąpić ją długością przykładowych miar. Pozwala nam to automatycznie przekształcić dane tak, aby pasowały do pożądanego kształtu, którego oczekuje scikit-learn.

Odległość euklidesowa to miara odległości w linii prostej między dwoma punktami w przestrzeni wielowymiarowej. Jest powszechnie stosowana w uczeniu maszynowym do obliczania podobieństwa lub odmienności między punktami danych. Odległość euklidesowa między dwoma punktami, A i B, w przestrzeni dwuwymiarowej (x, y) może być obliczona przy użyciu następującego wzoru:
d(A, B) = sqrt((x2 – x1)^2 + (y2 – y1)^2)
Tutaj (x1, y1) i (x2, y2) oznaczają odpowiednio współrzędne punktów A i B. Pierwiastek kwadratowy sumy kwadratów różnic między współrzędnymi daje nam odległość euklidesową między dwoma punktami.

W Pythonie możemy obliczyć odległość euklidesową między dwoma punktami za pomocą modułu math. Możemy zdefiniować funkcję, która przyjmuje współrzędne dwóch punktów jako dane wejściowe i zwraca odległość euklidesową. Oto przykład:

```
import math

def euclidean_distance(point1, point2):
        x1, y1 = point1
        x2, y2 = point2
        distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
        return distance

point_A = (2, 3)
point_B = (5, 7)
distance = euclidean_distance(point_A, point_B)

print("Euclidean distance between point A and point B:", distance)
```

W tym przykładzie definiujemy funkcję `euclidean_distance`, która przyjmuje dwa punkty jako dane wejściowe i oblicza odległość euklidesową między nimi. Następnie tworzymy dwa punkty, `point_A` i `point_B`, i obliczamy odległość między nimi za pomocą funkcji `euclidean_distance`. Na koniec drukujemy wynik.

Odległość euklidesowa nie ogranicza się do przestrzeni dwuwymiarowych. Można ją rozszerzyć na dowolną liczbę wymiarów. W uczeniu maszynowym odległość euklidesowa jest często używana jako miara podobieństwa do porównywania punktów danych w przestrzeniach wielowymiarowych. Jest ona szczególnie przydatna w algorytmach

klastrowania, takich jak k-means, gdzie odległość między punktami danych jest używana do przypisywania ich do klastrów.

Oprócz zastosowania w klastrowaniu, odległość euklidesowa jest również używana w innych algorytmach uczenia maszynowego, takich jak k-najbliższych sąsiadów (k-NN). W k-NN odległość euklidesowa między nowym punktem danych a istniejącymi oznaczonymi punktami danych jest obliczana w celu określenia k najbliższych sąsiadów. Ci sąsiedzi są następnie wykorzystywani do dokonywania przewidywań lub klasyfikacji.

Algorytm KNN można podsumować w następujących krokach:
1. Krok 1: Załaduj zbiór danych treningowych – Algorytm wymaga oznaczonego zbioru danych, aby trenować model. Zbiór danych powinien zawierać zarówno cechy wejściowe, jak i odpowiadające im etykiety lub wartości wyjściowe.
2. Krok 2: Wybierz wartość K – Wartość K określa liczbę sąsiadów, którą należy wziąć pod uwagę podczas dokonywania przewidywań. Jest to ważny parametr, który należy starannie wybrać na podstawie problemu i zestawu danych. Mała wartość K może prowadzić do nadmiernego dopasowania, podczas gdy duża wartość K może skutkować niedopasowaniem.
3. Krok 3: Oblicz odległość – Dla każdego nowego punktu danych oblicz odległość do wszystkich istniejących oznaczonych punktów danych w zestawie danych treningowych, używając wybranej metryki odległości. Ten krok obejmuje obliczenie odległości między cechami wejściowymi nowego punktu danych a odpowiadającymi im cechami każdego oznaczonego punktu danych.
4. Krok 4: Wybierz K najbliższych sąsiadów – Zidentyfikuj K punktów danych o najmniejszych odległościach do nowego punktu danych. Te punkty danych są uważane za najbliższych sąsiadów.
5. Krok 5: Klasyfikuj lub przewiduj – W przypadku zadań klasyfikacji przypisz etykietę klasy, która jest najczęstsza wśród K najbliższych sąsiadów do nowego punktu danych. W przypadku zadań regresji oblicz średnią lub średnią ważoną wartości wyjściowych K najbliższych sąsiadów i przypisz ją jako wartość przewidywaną dla nowego punktu danych.
6. Oceń model: Zmierz wydajność algorytmu KNN, korzystając z odpowiednich wskaźników oceny, takich jak dokładność, precyzja, odwołanie lub średni błąd kwadratowy.

Algorytm KNN jest stosunkowo prosty do wdrożenia i może być łatwo zrozumiany przez początkujących w uczeniu maszynowym. Ma jednak pewne ograniczenia. Jednym z nich jest to, że może być kosztowny obliczeniowo, zwłaszcza w przypadku dużych zestawów danych. Ponadto wydajność algorytmu może być wrażliwa na wybór wartości K i metryki odległości.
W Pythonie istnieje kilka bibliotek i frameworków, które zapewniają implementacje algorytmu KNN, takie jak scikit-learn, TensorFlow i PyTorch. Biblioteki te oferują wygodne funkcje i klasy do trenowania i używania modeli KNN, a także narzędzia do oceny ich wydajności.

W przypadku zadań klasyfikacyjnych przypisujemy etykietę klasy, która pojawia się najczęściej wśród K najbliższych sąsiadów do nowej instancji.
W przypadku remisu możemy użyć mechanizmu głosowania lub rozważyć inne czynniki, aby rozstrzygnąć remis. W przypadku zadań regresji obliczamy średnią wartość zmiennej docelowej wśród K najbliższych sąsiadów i przypisujemy ją do nowej instancji.
Aby ocenić wydajność naszego algorytmu KNN, możemy podzielić nasz zbiór danych na zbiór treningowy i zbiór testowy. Używamy zbioru treningowego do trenowania algorytmu, a zbioru testowego do oceny jego dokładności predykcyjnej. Następnie możemy porównać przewidywane etykiety lub wartości z rzeczywistymi etykietami lub wartościami w zbiorze testowym, używając odpowiednich metryk oceny, takich jak dokładność, precyzja, odwołanie lub średni błąd kwadratowy.

Programowanie naszego własnego algorytmu K najbliższych sąsiadów w Pythonie pozwala nam lepiej zrozumieć, jak działa ten popularny algorytm uczenia maszynowego. Implementując algorytm od podstaw, możemy dostosować jego zachowanie i zbadać różne warianty i ulepszenia. Bogaty ekosystem bibliotek i frameworków Pythona zapewnia solidną podstawę do opracowywania i wdrażania modeli uczenia maszynowego w różnych domenach.

```python
import pandas as pd
import numpy as np
import random
from math import sqrt
from collections import Counter

def k_nearest_neighbors (data, predict, k=3):
    if len(data) >= k:
        warnings.warn('K is set to a value less than total voting groups!')
    distances = []

    for group in data:
        for features in data [group]:
            euclidean_distance = np.linalg.norm(np.array (features)-np.array (predict))
            distances.append([euclidean_distance, group])

    votes = [i[1] for i in sorted (distances) [:k]]
    print (Counter (votes).most_common(1))
    vote_result = Counter (votes).most_common(1)[0][0]

    return vote_result

df = pd.read_csv('breast-cancer-wisconsin.data')
df.replace('?', -99999, inplace=True)
df.drop( 'Sample_code_number', axis=1,inplace=True)

full_data = df.astype(float).values.tolist()
#print(full_data)

random.shuffle(full_data)

test_size = 0.2
train_set = {2: [], 4:[]}
test_set = {2: [], 4:[]}

train_data = full_data[:-int(test_size*len (full_data))]
test_data = full_data[-int(test_size*len (full_data)):]

for i in train_data:
    train_set[i[-1]].append(i[:-1])
for i in test_data:
    test_set[i[-1]].append(i[:-1])

correct = 0
total = 0

for group in test_set:
    for data in test_set[group]:
        vote = k_nearest_neighbors(train_set, data, k=5)
        if group == vote:
            correct +=1
        total +=1

print('Accuracy:', correct/total)
```

Jedno pytanie, które się pojawia, to **czy zwiększenie wartości K koniecznie prowadziłoby do zwiększenia dokładności.** Aby to zbadać, rozważmy przykład, w którym K jest ustawione na 25. Uruchamiając algorytm wielokrotnie z tą wartością K, obserwujemy, że dokładność waha się między 88% a 98%. Oznacza to, że zwiększenie K nie zawsze gwarantuje wyższą dokładność. Podobnie, gdy ustawimy K na 75, dokładność pozostaje stale wysoka na poziomie około 95-97%.
Warto zauważyć, że zbiór danych użyty w tym przykładzie ma około 600 punktów danych, z których tylko 30% jest złośliwych, a pozostałe 70% łagodnych. Ten skośny rozkład może mieć wpływ na dokładność algorytmu. Zwiększenie K do 200, co obejmuje więcej punktów danych, w rzeczywistości skutkuje niższą dokładnością. Dzieje się tak, ponieważ algorytm jest pod wpływem klasy większościowej (łagodnej) z powodu niezrównoważonego rozkładu. Dlatego ważne jest, aby ostrożnie wybrać wartość K na podstawie zbioru danych i problemu, o który chodzi. Podczas gdy 5 wydaje się być dobrym przypuszczeniem w tym przypadku, zaleca się eksperymentowanie z różnymi wartościami K, aby określić optymalny wybór dla danego zbioru danych.

Aby obliczyć pewność, dzielimy liczbę głosów na klasę przez wartość K. Na przykład, jeśli K jest ustawione na 5, mamy nadzieję uzyskać wynik pewności wynoszący 5. Drukując wyniki pewności wraz z wynikami klasyfikacji, możemy uzyskać wgląd w poziom pewności przewidywań.

W podanym kodzie wyniki ufności są wyświetlane dla każdego wyniku klasyfikacji. Większość prawidłowych przewidywań ma wynik ufności 1,0, co wskazuje na wysoką pewność. Istnieją jednak pewne nieprawidłowe przewidywania z niższymi wynikami ufności, takimi jak 0,8 i 0,6.
Poprzez dostosowanie rozmiaru testu możemy zaobserwować zmiany w wynikach ufności. Zmniejszenie rozmiaru testu poświęca więcej danych, co potencjalnie prowadzi do zmniejszenia dokładności. Jednak zmniejsza również liczbę wystąpień z 100% ufnością, podkreślając znaczenie uwzględnienia poziomów ufności w podejmowaniu decyzji

Na początek przejrzyjmy wyniki uzyskane z uruchomienia algorytmu. Wykonaliśmy 10 testów, przy czym początkowo uruchomiono 5 testów. Średnia dokładność uzyskana z tych testów wyniosła 96,2%. Ważne jest, aby zauważyć, że wartość k użyta w tych testach wynosiła 5.
Następnie dokonaliśmy pewnych modyfikacji kodu, aby zapewnić dokładne wyniki. Usunęliśmy zbędne polecenia print i upewniliśmy się, że wartość k pozostała spójna. Po uruchomieniu zmodyfikowanego kodu 25 razy uzyskaliśmy średnią dokładność 96,4%.

Idąc dalej, zbadaliśmy inną wersję algorytmu K najbliższych sąsiadów. Zastosowaliśmy te same modyfikacje do tej wersji i uruchomiliśmy ją 25 razy. Średnia dokładność osiągnięta w tym przypadku wyniosła 96,8%.

Teraz omówmy różnice między tymi dwiema wersjami. Po pierwsze, algorytm K najbliższych sąsiadów ma domyślny parametr o nazwie „n_jobs", który określa liczbę równoległych zadań do uruchomienia. Domyślnie ta wartość jest ustawiona na 1. Jednak można ją ustawić na -1, aby wykorzystać jak najwięcej równoległych zadań, co poprawi wydajność.
Dodatkowo algorytm K nearest neighbors ma również parametr o nazwie „radius", który umożliwia wykluczenie punktów spoza określonego promienia. Może to być przydatne w niektórych scenariuszach.

Warto wspomnieć, że dokładność osiągnięta przez naszą implementację była porównywalna z dokładnością innych bibliotek, takich jak scikit-learn.
Ponadto algorytm K najbliższych sąsiadów jest wszechstronny i może być stosowany zarówno do danych liniowych, jak i nieliniowych. Ta elastyczność sprawia, że jest to cenne narzędzie do zadań klasyfikacyjnych.

Algorytm K najbliższych sąsiadów jest skutecznym algorytmem uczenia maszynowego do zadań klasyfikacyjnych. Można go zaimplementować za pomocą Pythona i zapewnia on dokładne wyniki. Poprzez dostosowanie parametrów, takich jak „n_jobs" i „radius", wydajność algorytmu można dodatkowo zoptymalizować. Ponadto ważne jest, aby wziąć pod uwagę charakter analizowanych danych, ponieważ algorytm może obsługiwać zarówno dane liniowe, jak i nieliniowe.

Algorytm K najbliższych sąsiadów to algorytm klasyfikacji, który można również wykorzystać do regresji.
Działa on poprzez znalezienie K najbliższych oznaczonych punktów danych w zestawie treningowym do nowego wejścia, a następnie klasyfikowanie lub przewidywanie wartości tego wejścia na podstawie większości głosów lub średniej K sąsiadów.
W przypadku klasyfikacji, jeśli mamy nieznany punkt danych, możemy zmierzyć błąd kwadratowy między liniami regresji różnych klas. Klasa z mniejszym błędem kwadratowym byłaby przypisana do nowego punktu danych. Tę metodę można stosować w przypadku danych liniowych, umożliwiając klasyfikację nawet wtedy, gdy prognozowanie nie jest wymagane.

Jednak w przypadku zestawów danych z danymi nieliniowymi użycie linii regresji do klasyfikacji skutkowałoby słabą dokładnością. W takich przypadkach można zastosować algorytm K najbliższych sąsiadów. Polega on na zmierzeniu odległości między nowym punktem danych a jego K najbliższymi sąsiadami. Poprzez oddanie głosu większościowego wśród tych sąsiadów algorytm może sklasyfikować nowy punkt danych.

Na przykład rozważmy zbiór danych z pomarańczowymi i niebieskimi kropkami. Nawet jeśli istnieje linia najlepszego dopasowania dla obu klas, współczynnik determinacji będzie niski, co wskazuje na słabą pewność klasyfikacji algorytmu. Jednak używając K najbliższych sąsiadów, możemy sklasyfikować nowy punkt danych, mierząc odległość między nim a jego najbliższymi sąsiadami. Jeśli K jest równe trzem, rozważylibyśmy trzy najbliższe punkty i przeprowadzili głosowanie. W tym przypadku, jeśli dwa z trzech najbliższych punktów są pomarańczowe, sklasyfikowalibyśmy nowy punkt danych jako pomarańczowy.

Algorytm K najbliższych sąsiadów ma tę zaletę, że potrafi obsługiwać dane nieliniowe, co czyni go odpowiednim do zadań klasyfikacyjnych w takich przypadkach. Jest to wszechstronny i szeroko stosowany algorytm w uczeniu maszynowym.

```
def k_nearest_neighbors (data, predict, k=3):
    if len(data) >= k:
        warnings.warn('K is set to a value less than total voting groups!')
    distances = []

    for group in data:
        for features in data [group]:
            euclidean_distance = np.linalg.norm(np.array (features)-np.array (predict))
            distances.append([euclidean_distance, group])

    votes = [i[1] for i in sorted (distances) [:k]]
    #print (Counter (votes).most_common(1))
    vote_result = Counter (votes).most_common(1)[0][0]
    confidence = Counter (votes).most_common(1)[0][1] / k

    #print(vote_result, confidence)
    return vote_result, confidence
```

```
accuracies = []

for i in range(25):
    df = pd.read_csv('breast-cancer-wisconsin.data')
    df.replace('?', -99999, inplace=True)
    df.drop( 'Sample_code_number', axis=1,inplace=True)

    full_data = df.astype(float).values.tolist()
    #print(full_data)

    random.shuffle(full_data)

    #test_size = 0.2
    test_size = 0.4
    train_set = {2: [], 4:[]}
    test_set = {2: [], 4:[]}

    train_data = full_data[:-int(test_size*len (full_data))]
    test_data = full_data[-int(test_size*len (full_data)):]

    for i in train_data:
        train_set[i[-1]].append(i[:-1])
    for i in test_data:
        test_set[i[-1]].append(i[:-1])

    correct = 0
    total = 0

    for group in test_set:
        for data in test_set[group]:
            vote, confidence = k_nearest_neighbors(train_set, data, k=5)
            if group == vote:
                correct +=1
            else:
                print(confidence)
            total +=1

    print('Accuracy:', correct/total)
    accuracies.append(correct/total)

print(sum(accuracies)/len(accuracies))
```

```
accuracies = []

for i in range(25):

    df = pd.read_csv('breast-cancer-wisconsin.data')
    df.replace('?', -99999, inplace=True)
    df.drop( 'Sample_code_number', axis=1,inplace=True)

    X = np.array(df.drop('Class',axis=1))
    y = np.array(df['Class'])


    X_train, X_test, y_train, y_test = (
        train_test_split(X,y,test_size=0.2))

    clf = neighbors.KNeighborsClassifier(n_jobs=1)
    clf.fit(X_train, y_train)

    accuracy = clf.score(X_test, y_test)
    #print(accuracy)

    accuracies.append(accuracy)

print(sum(accuracies)/len(accuracies))
```

SUPPORT VECTOR MACHINE (SVM)

Wstęp

Support Vector Machine (SVM) to potężny algorytm uczenia maszynowego, który jest szeroko stosowany w zadaniach klasyfikacji i regresji. Opiera się na koncepcji znalezienia optymalnej hiperpłaszczyzny, która oddziela różne klasy lub przewiduje wartości ciągłe. W tym materiale dydaktycznym przedstawimy kompleksowe wprowadzenie do Support Vector Machines i zbadamy ich zastosowania przy użyciu Pythona.

1. Wprowadzenie do maszyn wektorów nośnych:

Maszyny wektorów nośnych należą do rodziny algorytmów uczenia nadzorowanego. Są szczególnie przydatne w przypadku złożonych zestawów danych, które mają nieliniowe granice decyzyjne. SVM mają na celu znalezienie najlepszej możliwej hiperpłaszczyzny, która maksymalnie rozdziela punkty danych różnych klas. Hiperpłaszczyzna jest określana przez podzbiór punktów danych zwanych wektorami nośnymi, które leżą najbliżej granicy decyzyjnej.

2. Zasada działania maszyn wektorów nośnych:

Zasada działania maszyn wektorów nośnych polega na transformacji danych do przestrzeni o wyższym wymiarze, w której możliwe jest rozdzielenie liniowe. Osiąga się to za pomocą funkcji jądra, które mapują dane z oryginalnej przestrzeni cech do przestrzeni cech o wyższym wymiarze. Następnie SVM znajdują hiperpłaszczyznę, która maksymalizuje margines między wektorami nośnymi różnych klas.

3. Klasyfikacja za pomocą maszyn wektorów nośnych:

W przypadku klasyfikacji maszyny wektorów nośnych mają na celu znalezienie hiperpłaszczyzny, która oddziela punkty danych różnych klas z największym marginesem. Marginesem jest odległość między hiperpłaszczyzną a wektorami nośnymi. SVM mogą obsługiwać zarówno problemy klasyfikacji binarnej, jak i wieloklasowej. W przypadku klasyfikacji binarnej SVM używają funkcji decyzyjnej, aby przypisać nowe punkty danych do jednej z dwóch klas na podstawie ich położenia względem hiperpłaszczyzny.

4. Regresja z maszynami wektorów nośnych:

Maszyny wektorów nośnych mogą być również używane do zadań regresji. W regresji celem jest przewidywanie wartości ciągłych zamiast klas dyskretnych. Regresja SVM ma na celu znalezienie hiperpłaszczyzny, która najlepiej pasuje do punktów danych, jednocześnie minimalizując błąd. Hiperpłaszczyzna jest określana przez wektory nośne, a przewidywana wartość dla nowego punktu danych jest uzyskiwana przez ocenę położenia punktu danych względem hiperpłaszczyzny.

5. Implementacja maszyny wektorów nośnych w Pythonie:

Python udostępnia kilka bibliotek, które ułatwiają implementację maszyn wektorów nośnych. Jedną z najpopularniejszych bibliotek jest scikit-learn, która oferuje kompleksowy zestaw narzędzi do uczenia maszynowego. Scikit-learn udostępnia dedykowany moduł dla SVM, umożliwiając użytkownikom łatwe trenowanie i ocenianie modeli SVM. Ponadto bogaty ekosystem Pythona obejmuje biblioteki do wstępnego przetwarzania danych, wizualizacji i oceny modelu, co czyni go potężną platformą do implementacji SVM.

6. Zastosowanie maszyn wektorów nośnych:

Maszyny wektorów nośnych mają szeroki zakres zastosowań w różnych dziedzinach. Niektóre typowe zastosowania obejmują klasyfikację tekstu, rozpoznawanie obrazów, bioinformatykę i finanse. SVM są szczególnie dobrze przystosowane do problemów z danymi o dużej liczbie wymiarów i złożonymi granicami decyzyjnymi. Ich zdolność do obsługi dużych przestrzeni cech i nieliniowych relacji sprawia, że są popularnym wyborem w wielu scenariuszach z życia wziętych.

7. Wnioski:

Maszyny wektorów nośnych to wszechstronny i potężny algorytm uczenia maszynowego, który można wykorzystać zarówno do zadań klasyfikacji, jak i regresji. Są one szczególnie przydatne w przypadku złożonych zestawów danych, które mają nieliniowe granice decyzyjne. Dzięki Pythonowi i bibliotekom takim jak scikit-learn implementacja i stosowanie maszyn wektorów nośnych staje się proste. Wykorzystując możliwości maszyn SVM, badacze i praktycy mogą rozwiązywać szeroki zakres rzeczywistych problemów.

W matematyce wektor jest wielkością, która ma zarówno wielkość, jak i kierunek. Często jest reprezentowany jako tablica liczb lub współrzędnych w przestrzeni wielowymiarowej. W kontekście SVM wektory odgrywają ważną rolę w reprezentowaniu punktów danych i granic decyzyjnych.

Rozważmy problem klasyfikacji binarnej, w którym mamy dwie klasy, oznaczone jako dodatnie i ujemne. Każdy punkt danych w naszym zestawie danych można przedstawić jako wektor w przestrzeni cech. Przestrzeń cech jest matematyczną reprezentacją zmiennych wejściowych lub cech, których używamy do tworzenia przewidywań. Na przykład, jeśli klasyfikujemy obrazy kotów i psów, cechami mogą być wartości pikseli obrazów.

W SVM celem jest znalezienie hiperpłaszczyzny, która rozdziela klasy dodatnie i ujemne z maksymalnym marginesem. Hiperpłaszczyznę można postrzegać jako granicę decyzyjną, która dzieli przestrzeń cech na dwa regiony. Wektory w tym przypadku reprezentują punkty danych w przestrzeni cech, a ich pozycje względem hiperpłaszczyzny określają ich etykiety klas.

Koncepcja SVM
Aby zrozumieć koncepcję wektorów w SVM, rozważmy prosty przykład. Załóżmy, że mamy dwie klasy, A i B, i chcemy sklasyfikować nowy punkt danych na podstawie jego cech. Cechy punktu danych możemy przedstawić jako wektor w przestrzeni cech. Jeśli wektor leży po jednej stronie hiperpłaszczyzny, należy do klasy A, a jeśli leży po drugiej stronie, należy do klasy B.

Pozycja wektora względem hiperpłaszczyzny jest określana przez jego iloczyn skalarny z wektorem normalnym hiperpłaszczyzny. Iloczyn skalarny mierzy podobieństwo między dwoma wektorami i daje nam wartość skalarną. Jeśli iloczyn skalarny jest dodatni, wektor leży po jednej stronie hiperpłaszczyzny, a jeśli jest ujemny, wektor leży po drugiej stronie.

W SVM wektory nośne to punkty danych leżące najbliżej hiperpłaszczyzny. Wektory te odgrywają ważną rolę w definiowaniu granicy decyzyjnej i maksymalizowaniu marginesu. Poprzez maksymalizację marginesu dążymy do osiągnięcia lepszej generalizacji i poprawy zdolności modelu do dokładnego klasyfikowania niewidzianych danych.

Wektory nośne to wektory, które mają współczynniki niezerowe w modelu SVM. Współczynniki te określają znaczenie każdego wektora nośnego w definiowaniu granicy decyzyjnej. Poprzez dostosowanie współczynników możemy kontrolować położenie i orientację hiperpłaszczyzny, wpływając w ten sposób na wynik klasyfikacji.
Wektory są podstawowymi jednostkami w SVM, które reprezentują punkty danych w przestrzeni cech. Ich pozycje względem hiperpłaszczyzny określają etykiety klas, a wektory nośne odgrywają kluczową rolę w definiowaniu granicy decyzyjnej i maksymalizacji marginesu.

Aby zilustrować koncepcję SVM, rozważmy prosty problem klasyfikacji binarnej. Załóżmy, że mamy zbiór danych z dwiema klasami, oznaczonymi jako dodatnie i ujemne. Każdy punkt danych w zbiorze danych jest reprezentowany przez zestaw cech. Celem SVM jest znalezienie hiperpłaszczyzny, która rozdziela klasy dodatnie i ujemne z maksymalnym marginesem.
W SVM granica decyzyjna jest definiowana przez zestaw parametrów, w tym wagi przypisane do każdej cechy i termin odchylenia. Problem optymalizacji w SVM obejmuje znalezienie optymalnych wartości dla tych parametrów, co można osiągnąć za pomocą różnych technik matematycznych, takich jak programowanie kwadratowe.
Jedną z kluczowych cech SVM jest jego zdolność do obsługi nieliniowych granic decyzyjnych. Osiąga się to za pomocą techniki zwanej sztuczką jądra. Sztuczka jądra pozwala SVM na niejawne mapowanie danych wejściowych do przestrzeni cech o wyższym wymiarze, w której można znaleźć liniową granicę decyzyjną. Powszechnie używane funkcje jądra obejmują liniową, wielomianową, radialną funkcję bazową (RBF) i sigmoidalną.

Asercja wektora
Teraz omówmy asercję wektora nośnego w SVM. Asercja wektora nośnego odnosi się do faktu, że granica decyzyjna w SVM jest określana tylko przez podzbiór punktów danych treningowych, mianowicie wektory nośne. Te wektory nośne to punkty danych, które leżą na marginesie lub są błędnie sklasyfikowane. Pozostałe punkty danych, które są poprawnie sklasyfikowane i leżą z dala od marginesu, nie wpływają na granicę decyzyjną.

Asercja wektora nośnego ma kilka zalet. Po pierwsze, sprawia, że SVM jest wydajny obliczeniowo, ponieważ tylko podzbiór danych treningowych jest używany do określenia granicy decyzyjnej. Po drugie, sprawia, że SVM jest odporny na wartości odstające, ponieważ granica decyzyjna nie jest pod wpływem pojedynczych punktów danych, które są daleko od marginesu. Po trzecie, asercja wektora nośnego pozwala SVM na dobre uogólnianie do niewidzianych danych, ponieważ koncentruje się na najbardziej informacyjnych punktach danych.
Aby zaimplementować SVM z asercją wektorów nośnych w Pythonie, możemy użyć biblioteki scikit-learn, która zapewnia kompleksowy zestaw narzędzi do uczenia maszynowego. Scikit-learn zapewnia klasę o nazwie SVC (Support Vector Classifier), która umożliwia nam trenowanie modelu SVM z różnymi funkcjami jądra.
Poniżej znajduje się przykładowy fragment kodu, który pokazuje, jak zaimplementować SVM z asercją wektorów nośnych przy użyciu języka Python i pakietu scikit-learn:

```
from sklearn import svm

# Create a SVM classifier with a linear kernel

clf = svm.SVC(kernel='linear')

# Train the classifier on the training data

clf.fit(X_train, y_train)

# Make predictions on the test data

y_pred = clf.predict(X_test)

# Evaluate the performance of the classifier

accuracy = clf.score(X_test, y_test)
```

W swojej istocie Support Vector Machine jest klasyfikatorem binarnym, którego celem jest znalezienie optymalnej hiperpłaszczyzny w wielowymiarowej przestrzeni cech. Hiperpłaszczyzna jest wybierana w taki sposób, aby maksymalnie oddzielała punkty danych różnych klas. Punkty danych znajdujące się najbliżej hiperpłaszczyzny, znane jako wektory nośne, odgrywają ważną rolę w definiowaniu granicy decyzyjnej.

Aby zrozumieć koncepcję SVM, rozważmy prosty przykład dwuwymiarowego zbioru danych z dwiema klasami. Celem jest znalezienie linii, która oddziela punkty danych klasy A od klasy B z maksymalnym marginesem. Margines jest zdefiniowany jako odległość między hiperpłaszczyzną a najbliższymi punktami danych każdej klasy. SVM ma na celu znalezienie hiperpłaszczyzny, która maksymalizuje ten margines.

Aby to osiągnąć, SVM wykorzystuje koncepcję jąder. Jądra przekształcają dane wejściowe w przestrzeń cech o wyższym wymiarze, w której łatwiej jest znaleźć separację liniową. Powszechnie stosowane jądra obejmują liniowe, wielomianowe, radialne funkcje bazowe (RBF) i sigmoidalne. Wybór jądra zależy od charakteru danych i rozpatrywanego problemu.

W SVM problem optymalizacji można sformułować jako problem programowania kwadratowego. Celem jest minimalizacja błędu klasyfikacji przy maksymalizacji marginesu. Ten problem optymalizacji można rozwiązać za pomocą różnych algorytmów, takich jak algorytm Sequential Minimal Optimization (SMO) lub szeroko stosowana biblioteka LibSVM.

Po wytrenowaniu modelu SVM można go używać do klasyfikowania nowych, niewidzianych punktów danych. Model przypisuje etykietę klasy do danych testowych na podstawie strony granicy decyzyjnej, po której się znajdują. SVM zapewnia również możliwość oszacowania prawdopodobieństwa przynależności punktu danych do określonej klasy, co może być przydatne w niektórych zastosowaniach.

Wektory nośne składnikami SVM
Wektory nośne są kluczowymi składnikami SVM. Są to punkty danych, które leżą najbliżej granicy decyzyjnej, znanej również jako hiperpłaszczyzna, i odgrywają ważną rolę w określaniu optymalnej hiperpłaszczyzny do klasyfikacji.

Celem SVM jest znalezienie hiperpłaszczyzny, która maksymalizuje margines, czyli odległość między wektorami nośnymi po obu stronach granicy decyzyjnej.

Aby obliczyć szerokość marginesu, używamy równania: szerokość = (x+ – x-) • w / ||w||, gdzie x+ i x- to wektory nośne, w to wektor wagi, a ||w|| to wielkość w. Celem jest maksymalizacja tej szerokości.
Aby uprościć równanie, szerokość możemy wyrazić jako: width = 2 / ||w||. Oznacza to, że aby zmaksymalizować szerokość, musimy zminimalizować wielkość wektora wagi w.

Aby jeszcze bardziej uprościć problem, możemy zminimalizować połowę wielkości wektora w do kwadratu, co jest równoważne zminimalizowaniu wielkości wektora w. Ta matematyczna wygoda pozwala nam podstawić równanie do Lagrange'a.

Lagrange'a to funkcja, która uwzględnia ograniczenia problemu. W przypadku SVM ograniczenie jest zdefiniowane jako: y • (x • w + b) – 1 = 0, gdzie y jest etykietą klasy, x jest wektorem cech, w jest wektorem wag, a b jest terminem odchylenia.

Wprowadzając mnożniki Lagrange'a, możemy rozwiązać problem optymalizacji i znaleźć optymalne wartości w i b, które minimalizują moduł wektora w, spełniając jednocześnie równanie ograniczenia.

Algorytm maszyny wektorów nośnych ma na celu znalezienie optymalnej hiperpłaszczyzny, która maksymalizuje margines między wektorami nośnymi. Osiąga się to poprzez minimalizację wielkości wektora wagowego w, z zastrzeżeniem równania ograniczenia obejmującego mnożniki Lagrange'a.
Rozumiejąc podstawy maszyn wektorów nośnych, możemy zastosować ten potężny algorytm do różnych problemów klasyfikacji i regresji.

Głównym celem SVM jest znalezienie najlepszej hiperpłaszczyzny, która rozdziela punkty danych różnych klas z maksymalnym marginesem. Aby to osiągnąć, SVM używa formuły matematycznej zwanej Lagrange'em, która obejmuje mnożniki Lagrange'a. Lagrange'a składa się z dwóch części: pierwsza część to wielkość wektora wagowego, oznaczona jako [w], a druga część to suma po mnożnikach Lagrange'a, oznaczona jako αi.

Ograniczenie w SVM jest definiowane przez wektory nośne, które są punktami danych najbliższymi granicy decyzyjnej. Równanie dla hiperpłaszczyzny w SVM jest podane przez [w]·[x] + b = 0, gdzie [w] jest wektorem wagowym, [x] jest wektorem wejściowym, a b jest terminem odchylenia. Modyfikując termin odchylenia, możemy przesunąć hiperpłaszczyznę w górę lub w dół.

Aby zoptymalizować SVM, musimy różniczkować Lagrange'a względem wektora wagowego [w] i członu odchylenia b. Różniczkowanie Lagrange'a względem [w] daje nam równanie [w] = Σ(αi * yi * xi), gdzie αi jest mnożnikiem Lagrange'a powiązanym z i-tym punktem danych, yi jest odpowiadającą etykietą klasy, a xi jest wektorem wejściowym.

Różniczkując Lagrange'a względem b otrzymujemy równanie Σ(αi * yi) = 0. Równania te stanowią podstawę rozwiązania problemu optymalizacji SVM.

Problem optymalizacji w SVM jest problemem programowania kwadratowego, który obejmuje maksymalizację Lagrange'a względem mnożników Lagrange'a αi. Mnożniki Lagrange'a odgrywają ważną rolę w określaniu wektorów nośnych i granicy decyzyjnej.

Jedną z wad SVM jest jego złożoność, zarówno pod względem matematyki, jak i samego problemu optymalizacji. Inną wadą jest to, że wszystkie zestawy cech muszą znajdować się w pamięci w celu optymalizacji, co może być niewykonalne w przypadku dużych zestawów danych. Istnieją jednak metody, takie jak sekwencyjna minimalna optymalizacja (SMO), których można używać do pracy z mniejszymi lub większymi zestawami danych.

Aby zrozumieć optymalizację SVM, rozważ problem klasyfikacji binarnej. Biorąc pod uwagę zbiór przykładów treningowych, z których każdy jest oznaczony jako należący do jednej z dwóch kategorii, algorytm treningowy SVM buduje model, który przypisuje nowe przykłady do jednej lub drugiej kategorii. Model ten przedstawia przykłady jako punkty w przestrzeni, odwzorowane tak, aby przykłady oddzielnych kategorii były podzielone wyraźną luką, która jest tak szeroka, jak to możliwe. Matematycznie problem optymalizacji można sformułować następująco:

Matematycznie problem optymalizacji można sformułować następująco:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

$$\text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

Tutaj $\mathbf{w}$jest wektorem normalnym do hiperpłaszczyzny, $b$jest członem odchylenia, $\mathbf{x}_i$reprezentuje wektory cech, a y_isą etykietami klas, które wynoszą +1 lub -1.

Celem jest minimalizacja $\frac{1}{2}\|\mathbf{w}\|^2$, co jest równoważne z maksymalizacją marginesu między klasami. Ograniczenia zapewniają, że każdy punkt danych jest poprawnie klasyfikowany z marginesem co najmniej 1.

W przypadkach, gdy dane nie są liniowo rozdzielne, SVM wprowadzają zmienne luzu, $\xi_i$aby umożliwić pewną błędną klasyfikację. Problem optymalizacji staje się wówczas następujący:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i$$

$$\text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

Oto Cparametr regularyzacji, który kontroluje kompromis między maksymalizacją marginesu i minimalizacją błędu klasyfikacji.

Aby rozwiązać problem optymalizacji, zazwyczaj używamy technik takich jak programowanie kwadratowe (QP). Jednak w przypadku dużych zbiorów danych QP może być kosztowne obliczeniowo. Zamiast tego możemy użyć algorytmu Sequential Minimal Optimization (SMO), który rozbija problem na mniejsze podproblemy, które są łatwiejsze do rozwiązania.

W Pythonie możemy zaimplementować SVM za pomocą bibliotek takich jak scikit-learn. Poniżej znajduje się przykład użycia scikit-learn do trenowania klasyfikatora SVM:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Only consider the first two classes for binary classification
X = X[y != 2]
y = y[y != 2]

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create an SVM classifier with a linear kernel
svm = SVC(kernel='linear', C=1.0)

# Train the classifier
svm.fit(X_train, y_train)

# Predict the labels of the test set
```

```
y_pred = svm.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

W tym przykładzie ładujemy zbiór danych Iris, wybieramy tylko dwie klasy do klasyfikacji binarnej i dzielimy dane na zbiory treningowe i testowe. Następnie tworzymy klasyfikator SVM z jądrem liniowym i trenujemy go na zbiorze treningowym. Na koniec przewidujemy etykiety zbioru testowego i oceniamy dokładność klasyfikatora.

Maszyny wektorów nośnych mogą również obsługiwać zadania klasyfikacji nieliniowej, używając funkcji jądra. Jądra przekształcają przestrzeń wejściową w przestrzeń o wyższym wymiarze, w której możliwa jest separacja liniowa. Do powszechnie używanych jąder należą jądro wielomianowe, jądro funkcji bazowej radialnej (RBF) i jądro sigmoidalne.

Na przykład jądro RBF jest zdefiniowane jako:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

Tutaj γjest parametr, który definiuje wpływ pojedynczego przykładu treningowego. Mały γoznacza szerszy wpływ, podczas gdy duży γoznacza węższy wpływ.

Maszyny wektorów nośnych to potężne narzędzia do zadań klasyfikacji i regresji liniowej i nieliniowej. Ich optymalizacja polega na znalezieniu hiperpłaszczyzny maksymalizującej margines między klasami, co można skutecznie rozwiązać za pomocą algorytmów takich jak SMO. Biblioteka scikit-learn języka Python zapewnia solidną implementację SVM, dzięki czemu jest dostępna dla praktycznych zastosowań.

Szczegółowy materiał dydaktyczny

Support Vector Machine (SVM) to potężna technika klasyfikacji w uczeniu maszynowym. Aby w pełni zrozumieć jej implementację i optymalizację, konieczne jest zrozumienie podstawowych pojęć i formuł matematycznych, które ją wspierają.

Równanie hiperpłaszczyzny, która jest podstawą SVM, jest podane wzorem:

$$\mathbf{x} \cdot \mathbf{w} + b = 0$$

Tutaj $\mathbf{x}$oznacza wektor cech, $\mathbf{w}$oznacza wektor wagowy, a $b$oznacza termin odchylenia.

W przypadku dodatniego wektora nośnego klasy równanie jest następujące:

$$\mathbf{x}_i \cdot \mathbf{w} + b = 1$$

W przypadku wektora wsparcia klasy negatywnej jest to:

$$\mathbf{x}_i \cdot \mathbf{w} + b = -1$$

Te równania są specyficzne dla wektorów nośnych, które są punktami danych leżącymi najbliżej granicy decyzyjnej. Sama granica decyzyjna jest zdefiniowana przez:

$$\mathbf{x} \cdot \mathbf{w} + b = 0$$

Rozważając punkt $\mathbf{x}_i$taki, że:

$$\mathbf{x}_i \cdot \mathbf{w} + b = 0.98$$

Ten punkt leży między hiperpłaszczyzną wektora nośnego a granicą decyzyjną. Jeśli wartość jest dodatnia, ale mniejsza niż 1, oznacza to, że punkt znajduje się w marginesie, ale nadal jest klasyfikowany jako część klasy dodatniej.

Klasyfikacja zestawu cech po wytrenowaniu klasyfikatora SVM jest określana przez znak funkcji decyzyjnej:

$$\text{sign}(\mathbf{x}_i \cdot \mathbf{w} + b)$$

Jeśli wynik jest dodatni, punkt należy do klasy dodatniej; jeśli ujemny, należy do klasy ujemnej. Jeśli wynik jest zerowy, punkt leży na granicy decyzyjnej.

Jeśli wynik jest dodatni, punkt należy do klasy dodatniej; jeśli ujemny, należy do klasy ujemnej. Jeśli wynik jest zerowy, punkt leży na granicy decyzyjnej.

Optymalizacja SVM obejmuje znalezienie optymalnego $\mathbf{w}$i b. Celem jest zminimalizowanie wielkości wektora wagowego $\mathbf{w}$, co można wyrazić jako:

$$\|\mathbf{w}\| = \sqrt{\sum_j w_j^2}$$

Jest to analogiczne do normy euklidesowej lub długości wektora w przestrzeni wielowymiarowej.

Optymalizacja podlega ograniczeniom:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) \geq 1$$

dla wszystkich próbek szkoleniowych i, gdzie $y_i$jest etykietą klasy (+1 lub -1).

Podsumowując, problem optymalizacji SVM ma na celu znalezienie wektora wag $\mathbf{w}$i odchylenia b, które minimalizują normę, $\mathbf{w}$spełniając jednocześnie ograniczenia klasyfikacji. Można to sformułować jako problem optymalizacji z ograniczeniami:

Zminimalizować:

$$\frac{1}{2}\|\mathbf{w}\|^2$$

Z zastrzeżeniem:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) \geq 1$$

Ten problem można rozwiązać, stosując różne techniki optymalizacji, takie jak programowanie kwadratowe. Po ustaleniu optymalnych $\mathbf{w}$i $b$klasyfikacja nowych punktów danych staje się prosta, wykorzystując funkcję decyzyjną.

W praktyce, przy użyciu języka Python, biblioteki takie jak scikit-learn udostępniają wbudowane funkcje do przeprowadzania klasyfikacji SVM, ułatwiając stosowanie SVM do rzeczywistych zestawów danych.

W kontekście maszyn wektorów nośnych (SVM) problem klasyfikacji można wyrazić matematycznie. Zmienna klasy, oznaczona jako y, przyjmuje wartości albo $+1$lub -1. Celem jest znalezienie hiperpłaszczyzny, która najlepiej rozdziela punkty danych różnych klas. Można to sformułować następująco:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

gdzie $\mathbf{w}$jest wektorem wagowym, $\mathbf{x}_i$reprezentuje wektor cech $i$próbki -tej, a $b$jest terminem odchylenia. Celem jest znalezienie wektora $\mathbf{w}$o najmniejszej wielkości, co oznacza minimalizację $||\mathbf{w}||$, przy jednoczesnym zapewnieniu, że ograniczenie $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$jest spełnione dla wszystkich próbek treningowych.

Aby rozwiązać ten problem optymalizacji, należy zminimalizować wielkość $\mathbf{w}$, lub bardziej formalnie, zminimalizować $\frac{1}{2}||\mathbf{w}||^2$, z zastrzeżeniem ograniczeń. Jest to problem programowania kwadratowego, charakteryzujący się wypukłym krajobrazem optymalizacji. Problemy wypukłe są korzystne, ponieważ mają pojedyncze globalne minimum, co czyni je łatwiejszymi do rozwiązania w porównaniu z problemami niewypukłymi.

Wizualizacja wypukłego krajobrazu optymalizacji, można go sobie wyobrazić jako krzywą w kształcie misy. Jeśli piłka zostanie umieszczona na powierzchni tej misy, stoczy się do najniższego punktu, reprezentującego minimalną wartość funkcji celu. W terminologii matematycznej oznacza to, że algorytm optymalizacji zbiegnie się do globalnego minimum $||\mathbf{w}||$.

Mając zbiór znanych cech $\mathbf{x}_i$, zadaniem jest znalezienie optymalnej $\mathbf{w}$i b. Rozważmy przykład, w którym wektory cech są dwuwymiarowe, tj. $\mathbf{w}$jest macierzą 1×2. Przykładem $\mathbf{w}$może być $[5, 3]$. Norma (lub wielkość) $\mathbf{w}$jest obliczana jako:

$$||\mathbf{w}|| = \sqrt{5^2 + 3^2} = \sqrt{34}$$

Jeśli $\mathbf{w}$były $[-5, 3]$, norma pozostaje $\sqrt{34}$. Jednak znak składników $\mathbf{w}$wpływa na iloczyn skalarny $\mathbf{x}_i \cdot \mathbf{w}$. Na przykład, jeśli $\mathbf{x}_i = [1, 2]$i $\mathbf{w} = [5, 3]$, iloczyn skalarny będzie wynosił:

$$\mathbf{x}_i \cdot \mathbf{w} = 1 \cdot 5 + 2 \cdot 3 = 11$$

Jeżeli $\mathbf{w} = [-5, 3]$, iloczyn skalarny zmienia się na:

$$\mathbf{x}_i \cdot \mathbf{w} = 1 \cdot (-5) + 2 \cdot 3 = 1$$

Pokazuje to, że znak składowych $\mathbf{w}$ma istotny wpływ na wynik iloczynu skalarnego, a tym samym na wynik klasyfikacji.

Proces optymalizacji w SVM obejmuje znalezienie wektora wag $\mathbf{w}$o najmniejszej wielkości, który spełnia ograniczenia klasyfikacji. Osiąga się to poprzez rozwiązanie problemu programowania kwadratowego wypukłego, zapewniając unikalne i optymalne rozwiązanie.

Optymalizacja maszyny wektorów nośnych (SVM) polega na znalezieniu optymalnej hiperpłaszczyzny, która oddziela różne klasy w zestawie danych. Tradycyjnie mieliśmy do czynienia z problemami, w których cechy umożliwiają proste

rozdzielenie za pomocą hiperpłaszczyzny o określonym nachyleniu. Jednak gdy zestaw cech ulega zmianie, kierunek i orientacja hiperpłaszczyzny również muszą się dostosować.

Na przykład rozważmy scenariusz, w którym początkowy wektor $\mathbf{w}$jest ustawiony na $[5, 5]$. Celem jest znalezienie odchylenia $b$takiego, że nierówność $y_i(\mathbf{x}_i \cdot \mathbf{w} + b) \geq 1$jest prawdziwa dla wszystkich próbek treningowych $(\mathbf{x}_i, y_i)$. Jeśli takie a $b$istnieje, zapisujemy wielkość i kierunek $\mathbf{w}$. Następnie dekrementujemy $\mathbf{w}$do $[4, 4]$i powtarzamy proces. To iteracyjne zmniejszanie $\mathbf{w}$jest kontynuowane, dopóki nie jest możliwe dalsze zmniejszanie bez naruszenia nierówności.

Podczas każdej iteracji istotne jest przetestowanie nie tylko bieżącego wektora $\mathbf{w}$, ale także jego wektorów ujemnych i permutacji, takich jak $[-5, 5]$, $[5, -5]$i $[-5, -5]$. To kompleksowe testowanie zapewnia, że żaden potencjalnie optymalny wektor nie zostanie pominięty.

Iloczyn skalarny odgrywa ważną rolę w tych obliczeniach. Na przykład, jeśli $\mathbf{w} = [5, 5]$, obliczamy iloczyn skalarny za pomocą wektorów takich jak $[-1, 1]$, $[1, -1]$i $[-1, -1]$. Każdy krok obejmuje ocenę czterech różnych konfiguracji, co sprawia, że proces jest intensywny obliczeniowo.

W kontekście optymalizacji maszyny wektorów nośnych (SVM) celem jest znalezienie optymalnej hiperpłaszczyzny rozdzielającej, która maksymalizuje margines między różnymi klasami w przestrzeni cech. Proces optymalizacji obejmuje znalezienie globalnego minimum problemu wypukłego, zapewniając, że rozwiązanie nie jest uwięzione w lokalnych minimach.

Na początek rozważ zainicjowanie wektora wagowego $\mathbf{w}$przy wartości początkowej, powiedzmy $\mathbf{w} = (10, 10)$. Ten wektor będzie iteracyjnie dostosowywany, aby spełnić ograniczenia SVM. Podstawowe ograniczenie dla SVM jest podane przez:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) \geq 1$$

gdzie $y_i$jest etykietą klasy, $\mathbf{x}_i$jest wektorem cech, $\mathbf{w}$jest wektorem wagowym, a $b$jest terminem odchylenia. Celem jest znalezienie $\mathbf{w}$i $b$takiego, aby ta nierówność była prawdziwa dla wszystkich próbek treningowych.

Proces optymalizacji obejmuje iteracyjne dostosowywanie $\mathbf{w}$i b. Początkowo, $\mathbf{w}$jest wstawiane do równania i b jest stopniowo dostosowywane, aby znaleźć wartość spełniającą ograniczenie. Każda kombinacja $\mathbf{w}$i b spełniająca warunek jest przechowywana w słowniku, gdzie kluczem jest wielkość $\mathbf{w}$, a wartością jest krotka $(\mathbf{w}, b)$.

Wielkość $\mathbf{w}$jest podana wzorem:

$$\text{magnitude}(\mathbf{w}) = \|\mathbf{w}\| = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$$

Słownik służy do śledzenia wszystkich prawidłowych par $\mathbf{w}$i b, a para o najmniejszej wielkości jest wybierana jako optymalne rozwiązanie. Ten proces zapewnia, że rozwiązanie odpowiada hiperpłaszczyznie o maksymalnym marginesie.

Iteracyjny proces optymalizacji można zwizualizować jako podejmowanie kroków w kierunku punktu minimalnego funkcji wypukłej. Jeśli kroki są zbyt duże, rozwiązanie może przekroczyć minimum, co wymaga mniejszych kroków w celu dostrojenia rozwiązania. Proces ten trwa, aż do osiągnięcia lub przybliżenia globalnego minimum.

W SVM problem jest wypukły, co oznacza, że nie ma ryzyka uwięzienia w minimach lokalnych. Zapewnia to, że znalezione minimum globalne jest rzeczywiście rozwiązaniem optymalnym.

Aby zweryfikować poprawność rozwiązania, można wykreślić granicę decyzyjną, wektory nośne i marginesy. Wektory nośne to punkty danych, które leżą najbliżej granicy decyzyjnej i spełniają równanie:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) = 1$$

Podczas optymalizacji, jeśli wartości wektorów nośnych są bardzo bliskie 1 (np. 1,001), oznacza to, że rozwiązanie jest bliskie optymalnemu. Obecność co najmniej jednego punktu danych w każdej klasie o wartości bliskiej 1 potwierdza identyfikację wektorów nośnych i poprawność optymalizacji.

Proces optymalizacji SVM obejmuje inicjalizację w, iteracyjne dostosowywanie w i b spełnianie ograniczeń SVM, przechowywanie prawidłowych rozwiązań i wybieranie rozwiązania o najmniejszej wielkości. Wypukła natura problemu zapewnia, że znalezione minimum globalne jest rozwiązaniem optymalnym, co można zweryfikować przez bliskość wartości wektorów nośnych do 1.

Aby skutecznie ustalić, czy w optymalizacji Support Vector Machine (SVM) dokonano ścisłego przybliżenia optymalnego rozwiązania, ważne jest obserwowanie wyników algorytmu. Gdy wyniki zarówno dla dodatnich, jak i ujemnych zestawów cech są bardzo zbliżone do jednego, wskazuje to na bliskość optymalnego rozwiązania. Jest to istotny znacznik w iteracyjnym procesie optymalizacji, wskazujący, czy kontynuować podejmowanie mniejszych kroków, czy zaakceptować znalezione minimum.

Optymalizacja to rozległa dziedzina, obejmująca znacznie więcej niż uczenie maszynowe. Konkretnie, optymalizacja SVM jest podzbiorem problemów optymalizacji wypukłej. Optymalizacja wypukła polega na znalezieniu minimum funkcji wypukłej w zbiorze wypukłym, zapewniając, że każde minimum lokalne jest również minimum globalnym.

Python udostępnia kilka modułów wspomagających optymalizację wypukłą. Jedną z godnych uwagi bibliotek jest CVXOpt, która została zaprojektowana do rozwiązywania problemów programowania kwadratowego (QP) i zawiera określone algorytmy dla SVM. Innym przydatnym modułem jest LibSVM, który jest również wykorzystywany w bibliotece Scikit-learn do zadań optymalizacji SVM. Jednak w celach edukacyjnych korzystne jest ręczne wdrożenie procesu optymalizacji, aby uzyskać głębsze zrozumienie podstawowych mechanizmów.

Aby to zobrazować, rozważmy następujący uproszczony przykład optymalizacji SVM:

1. **Funkcja obiektywna** : Celem jest zminimalizowanie funkcji:

$$\frac{1}{2}\|\mathbf{w}\|^2$$

z zastrzeżeniem ograniczeń:

$$y_i(\mathbf{w}\cdot\mathbf{x}_i+b)\geq 1 \quad \text{for all} \quad i.$$

2. **Mnożniki Lagrange'a** : Wprowadź mnożniki Lagrange'a $\alpha_i$dla każdego ograniczenia:

$$L(\mathbf{w},b,\alpha)=\frac{1}{2}\|\mathbf{w}\|^2-\sum_i \alpha_i\left[y_i(\mathbf{w}\cdot\mathbf{x}_i+b)-1\right].$$

3. **Optymalizacja** : Rozwiąż podwójny problem:

$$\max_\alpha \sum_i \alpha_i-\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i\cdot\mathbf{x}_j)$$

z zastrzeżeniem:

$$\sum_i \alpha_i y_i=0 \quad \text{and} \quad \alpha_i\geq 0.$$

4. **Rozwiązanie** : Optymalne wagi $\mathbf{w}$można wyprowadzić z:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i.$$

5. **Granica decyzyjna** : Granica decyzyjna jest dana przez:

$$\mathbf{w} \cdot \mathbf{x} + b = 0.$$

Dzięki ręcznemu kodowaniu tych kroków uzyskuje się intuicyjne zrozumienie procesu optymalizacji SVM. To praktyczne podejście demistyfikuje złożone formuły matematyczne i zwiększa zrozumienie poprzez praktyczną implementację.

Kroki SVM
Aby utworzyć SVM od podstaw, musimy zrozumieć podstawowe zasady i kroki. Przejdźmy przez proces:

1. Wstępne przetwarzanie danych:
Przed utworzeniem SVM konieczne jest wstępne przetworzenie danych. Obejmuje to obsługę brakujących wartości, skalowanie funkcji i kodowanie zmiennych kategorycznych. Wstępne przetwarzanie danych zapewnia optymalne działanie SVM poprzez usuwanie wszelkich odchyleń lub niespójności w danych.

2. Zdefiniuj klasę SVM:
W Pythonie możemy zdefiniować klasę SVM, która hermetyzuje niezbędne funkcje i atrybuty do trenowania i przewidywania. Klasa powinna zawierać metody inicjowania SVM, dopasowywania modelu do danych i dokonywania przewidywań.

3. Funkcje jądra:
SVM używają funkcji jądra do transformacji danych wejściowych do przestrzeni o wyższym wymiarze, gdzie łatwiej jest znaleźć oddzielającą hiperpłaszczyznę. Typowe funkcje jądra obejmują liniową, wielomianową i radialną funkcję bazową (RBF). Funkcje te wprowadzają nieliniowość do obsługi złożonych problemów klasyfikacji.

4. Trening SVM:
Proces treningu obejmuje znalezienie optymalnej hiperpłaszczyzny, która maksymalizuje margines, minimalizując jednocześnie błąd klasyfikacji. Osiąga się to poprzez rozwiązanie problemu optymalizacji wypukłej. Najczęstszym podejściem jest użycie algorytmu Sequential Minimal Optimization (SMO), który iteracyjnie aktualizuje parametry modelu aż do zbieżności.

5. Tworzenie prognoz:
Po przeszkoleniu SVM możemy użyć go do tworzenia prognoz na nowych, niewidzianych punktach danych. Klasa SVM powinna zawierać metodę przewidywania etykiet klas na podstawie nauczonych parametrów modelu.

Implementacja SVM od podstaw może być złożonym zadaniem, ale zapewnia głębsze zrozumienie podstawowych zasad. Warto jednak zauważyć, że biblioteki Pythona, takie jak scikit-learn, oferują wydajne i zoptymalizowane implementacje SVM, które są zalecane do praktycznych zastosowań.

Tworząc SVM od podstaw, zyskujemy wgląd w wewnętrzne działanie algorytmu i mamy elastyczność, aby dostosować go do konkretnych przypadków użycia. Jednak w większości scenariuszy wykorzystanie istniejących bibliotek jest bardziej wydajne i wygodne.

Szkolenie SVM
1. Wprowadzenie do Support Vector Machine (SVM):
Support Vector Machine to algorytm uczenia nadzorowanego, który analizuje dane i buduje granicę decyzyjną w celu klasyfikowania nowych instancji. Działa poprzez znalezienie optymalnej hiperpłaszczyzny, która rozdziela punkty danych na różne klasy, maksymalizując margines między klasami. SVM może obsługiwać zarówno dane liniowo rozdzielne, jak i nieliniowo rozdzielne, używając różnych funkcji jądra.

2. Proces szkolenia SVM:
Proces szkolenia SVM obejmuje znalezienie optymalnej hiperpłaszczyzny, która najlepiej oddziela punkty danych. Osiąga się to poprzez rozwiązanie problemu optymalizacji, którego celem jest zminimalizowanie błędu klasyfikacji i zmaksymalizowanie marginesu. Margines jest zdefiniowany jako odległość między hiperpłaszczyzną a najbliższymi punktami danych z każdej klasy.

3. Wstępne przetwarzanie danych:
Przed trenowaniem modelu SVM konieczne jest wstępne przetworzenie danych. Obejmuje to obsługę brakujących wartości, skalowanie cech i kodowanie zmiennych kategorycznych, jeśli jest to konieczne. Można zastosować techniki standaryzacji lub normalizacji, aby zapewnić, że wszystkie cechy mają takie samo znaczenie podczas trenowania.

4. Wybór funkcji jądra:
SVM używa funkcji jądra do transformacji danych wejściowych do przestrzeni o wyższym wymiarze, gdzie łatwiej jest znaleźć hiperpłaszczyznę oddzielającą klasy. Powszechnie używane funkcje jądra obejmują liniową, wielomianową, radialną funkcję bazową (RBF) i sigmoidalną. Wybór funkcji jądra zależy od charakteru danych i rozpatrywanego problemu.

5. Trening modelu SVM:
Aby trenować model SVM, potrzebujemy oznaczonego zestawu danych z cechami wejściowymi i odpowiadającymi im etykietami klas. Algorytm SVM uczy się optymalnej hiperpłaszczyzny, rozwiązując problem optymalizacji. Proces trenowania obejmuje znalezienie wektorów nośnych, które są punktami danych znajdującymi się najbliżej granicy decyzyjnej. Te wektory nośne odgrywają ważną rolę w definiowaniu granicy decyzyjnej.

6. Strojenie hiperparametrów:
SVM ma kilka hiperparametrów, które można dostroić, aby poprawić wydajność modelu. Niektóre z kluczowych hiperparametrów obejmują wybór funkcji jądra, parametr regularizacji (C) i parametr gamma (dla jądra RBF). Techniki walidacji krzyżowej można wykorzystać do znalezienia optymalnych wartości dla tych hiperparametrów.

7. Ocena modelu:
Po wytrenowaniu modelu SVM, istotne jest, aby ocenić jego wydajność. Typowe metryki oceny dla zadań klasyfikacyjnych obejmują dokładność, precyzję, odwołanie, wynik F1 i obszar pod krzywą ROC. W przypadku zadań regresji można użyć metryk, takich jak średni błąd kwadratowy (MSE) i R-kwadrat.

8. Zastosowania SVM:
SVM ma szeroki zakres zastosowań w różnych dziedzinach, w tym klasyfikacja obrazów, kategoryzacja tekstów, bioinformatyka, finanse i wiele innych. Jego zdolność do obsługi danych wielowymiarowych i nieliniowych relacji sprawia, że jest popularnym wyborem w przypadku wielu rzeczywistych problemów.

Support Vector Machine (SVM) to potężny algorytm uczenia maszynowego używany do zadań klasyfikacji i regresji. Znajdując optymalną hiperpłaszczyznę, która maksymalizuje margines między klasami, SVM może skutecznie obsługiwać złożone dane. Dzięki Pythonowi możemy łatwo implementować i trenować modele SVM, co pozwala nam rozwiązywać szeroki zakres problemów w różnych domenach.

Oto przykładowy fragment kodu, który pokazuje, jak używać SVM do klasyfikacji w Pythonie:

```
from sklearn import svm

# Create an SVM classifier with a linear kernel

clf = svm.SVC(kernel='linear')

# Fit the classifier to the training data

clf.fit(X_train, y_train)

# Predict the class labels for the test data

y_pred = clf.predict(X_test)

# Evaluate the classifier performance

accuracy = clf.score(X_test, y_test)
```

W tym przykładzie tworzymy klasyfikator SVM z liniowym jądrem, używając klasy SVC z modułu svm. Następnie dopasowujemy klasyfikator do danych treningowych, używając metody fit. Później możemy użyć metody predict, aby uzyskać przewidywane etykiety klas dla danych testowych. Na koniec oceniamy wydajność klasyfikatora, używając metody score, która zwraca dokładność przewidywań.

Optymalizacja SVM

Optymalizacja SVM może być dalej udoskonalona przez użycie sztuczki jądra i miękkiego marginesu. Sztuczka jądra pozwala SVM na wydajne radzenie sobie z nieliniowymi zadaniami klasyfikacji poprzez niejawne mapowanie danych wejściowych do przestrzeni cech o wyższym wymiarze. Miękki margines pozwala na pewne błędy błędnej klasyfikacji poprzez wprowadzenie zmiennej luzu, która rozluźnia ścisłe ograniczenie separacji. Te techniki umożliwiają SVM radzenie sobie z bardziej złożonymi i nakładającymi się dystrybucjami danych.

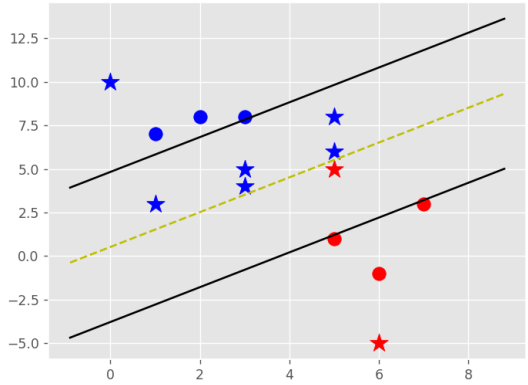
Optymalizacja SVM jest ważnym krokiem w trenowaniu maszyny wektorów nośnych do zadań klasyfikacyjnych. Polega na rozwiązaniu problemu optymalizacji wypukłej w celu znalezienia optymalnej hiperpłaszczyzny, która maksymalizuje margines między różnymi klasami. Python udostępnia potężne biblioteki, takie jak scikit-learn, do łatwej implementacji SVM. Wykorzystując techniki optymalizacji SVM, możemy uzyskać dokładne i solidne modele klasyfikacji.

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np

style.use('ggplot')


class Support_Vector_Machine:
    def __init__(self, visualization=True):
        self.visualization = visualization
        self.colors = {1: 'r', -1: 'b'}
        if self.visualization:
            self.fig = plt.figure()
            self.ax = self.fig.add_subplot(1, 1, 1)

    # train
    def fit(self, data):
        self.data = data
        # { ||w||: [w,b] }
        opt_dict = {}

        transforms = [[1, 1],
                      [-1, 1],
                      [-1, -1],
                      [1, -1]]

        all_data = []
        for yi in self.data:
            for featureset in self.data[yi]:
                for feature in featureset:
                    all_data.append(feature)

        self.max_feature_value = max(all_data)
        self.min_feature_value = min(all_data)
        all_data = None
```

```
# support vectors yi(xi.w+b) = 1

step_sizes = [self.max_feature_value * 0.1,
              self.max_feature_value * 0.01,
              # point of expense:
              self.max_feature_value * 0.001,
              ]

# extremely expensive
b_range_multiple = 2
# we dont need to take as small of steps
# with b as we do w
b_multiple = 5
latest_optimum = self.max_feature_value * 10

for step in step_sizes:
    w = np.array([latest_optimum, latest_optimum])
    # we can do this because convex
    optimized = False
    while not optimized:
        for b in np.arange(-1 * (self.max_feature_value * b_range_multiple),
                           self.max_feature_value * b_range_multiple,
                           step * b_multiple):
            for transformation in transforms:
                w_t = w * transformation
                found_option = True
                # weakest link in the SVM fundamentally
                # SMO attempts to fix this a bit
                # yi(xi.w+b) >= 1
                #
                # #### add a break here later..
                for i in self.data:
                    for xi in self.data[i]:
                        yi = i
                        if not yi * (np.dot(w_t, xi) + b) >= 1:
                            found_option = False
                            # print(xi,':',yi*(np.dot(w_t,xi)+b))
```

```
def predict(self, features):
    # sign( x.w+b )
    classification = np.sign(np.dot(np.array(features), self.w) + self.b)
    if classification != 0 and self.visualization:
        self.ax.scatter(features[0], features[1], s=200, marker='*', c=self.colors[classification])
    return classification

def visualize(self):
    [[self.ax.scatter(x[0], x[1], s=100, color=self.colors[i]) for x in data_dict[i]] for i in data_dict]

    # hyperplane = x.w+b
    # v = x.w+b
    # psv = 1
    # nsv = -1
    # dec = 0
    def hyperplane(x, w, b, v):
        return (-w[0] * x - b + v) / w[1]

    datarange = (self.min_feature_value * 0.9, self.max_feature_value * 1.1)
    hyp_x_min = datarange[0]
    hyp_x_max = datarange[1]


# (w.x+b) = 1
# positive support vector hyperplane
psv1 = hyperplane(hyp_x_min, self.w, self.b, 1)
psv2 = hyperplane(hyp_x_max, self.w, self.b, 1)
self.ax.plot([hyp_x_min, hyp_x_max], [psv1, psv2], 'k')

# (w.x+b) = -1
# negative support vector hyperplane
nsv1 = hyperplane(hyp_x_min, self.w, self.b, -1)
nsv2 = hyperplane(hyp_x_max, self.w, self.b, -1)
self.ax.plot([hyp_x_min, hyp_x_max], [nsv1, nsv2], 'k')

# (w.x+b) = 0
# positive support vector hyperplane
db1 = hyperplane(hyp_x_min, self.w, self.b, 0)
db2 = hyperplane(hyp_x_max, self.w, self.b, 0)
self.ax.plot([hyp_x_min, hyp_x_max], [db1, db2], 'y--')

plt.show()
```

```
data_dict = {-1: np.array([[1, 7],
                           [2, 8],
                           [3, 8], ]),

             1: np.array([[5, 1],
                          [6, -1],
                          [7, 3], ])}

svm = Support_Vector_Machine()
svm.fit(data=data_dict)

predict_us = [[0, 10],
              [1, 3],
              [3, 4],
              [3, 5],
              [5, 5],
              [5, 6],
              [6, -5],
              [5, 8]]

for p in predict_us:
    svm.predict(p)

svm.visualize()
```



Aby wprowadzić SVM, ważne jest zrozumienie koncepcji jąder. Jądra to funkcje matematyczne, które przekształcają dane wejściowe w przestrzeń o wyższym wymiarze, w której łatwiej jest oddzielić klasy. Innymi słowy, jądra pozwalają SVM uchwycić złożone wzorce w danych poprzez mapowanie ich na przestrzeń cech o wyższym wymiarze. Można używać różnych typów jąder, takich jak liniowe, wielomianowe, radialne funkcje bazowe (RBF) i sigmoidalne.

Jądro liniowe jest najprostszym typem jądra i nadaje się do danych liniowo rozdzielnych. Oblicza iloczyn skalarny między wektorami wejściowymi, skutecznie mierząc podobieństwo między nimi. Z drugiej strony jądro wielomianowe używa funkcji wielomianowej do mapowania danych w przestrzeni o wyższym wymiarze. To jądro jest przydatne, gdy dane są rozdzielalne przez zakrzywioną granicę. Jądro RBF jest popularnym wyborem w przypadku danych nieliniowo rozdzielnych. Używa funkcji Gaussa do pomiaru podobieństwa między wektorami wejściowymi. Na koniec jądro sigmoidalne jest często używane w zadaniach klasyfikacji binarnej. Mapuje dane w przestrzeni o wyższym wymiarze za pomocą funkcji sigmoidalnej.

W Pythonie biblioteka scikit-learn zapewnia kompleksowy zestaw narzędzi do implementacji SVM z różnymi jądrami. Biblioteka oferuje proste i intuicyjne API do trenowania i oceniania modeli SVM. Aby użyć SVM z scikit-learn, pierwszym krokiem jest zaimportowanie niezbędnych modułów.

Następnie musimy załadować zbiór danych i podzielić go na zbiory treningowe i testowe. Zbiór treningowy służy do trenowania modelu SVM, natomiast zbiór testowy służy do oceny jego wydajności. Po przygotowaniu danych możemy utworzyć obiekt klasyfikatora SVM i określić żądane jądro:

```
# Create an SVM classifier with a linear kernel

clf = svm.SVC(kernel='linear')

# Create an SVM classifier with an RBF kernel

clf = svm.SVC(kernel='rbf')
```

```
# Create an SVM classifier with a polynomial kernel

clf = svm.SVC(kernel='poly')

# Create an SVM classifier with a sigmoid kernel

clf = svm.SVC(kernel='sigmoid')
```

Po utworzeniu klasyfikatora możemy go wytrenować, korzystając z danych treningowych:

```
clf.fit(X_train, y_train)
```

Po wytrenowaniu modelu możemy go użyć do formułowania prognoz na podstawie nowych, niewidzianych dotąd danych:

```
y_pred = clf.predict(X_test)
```

Na koniec możemy ocenić wydajność modelu SVM, porównując przewidywane etykiety z prawdziwymi etykietami z zestawu testowego. Typowe metryki oceny obejmują dokładność, precyzję, odwołanie i wynik F1.

Maszyny wektorów nośnych (SVM) to potężne algorytmy uczenia maszynowego, których można używać do zadań klasyfikacji i regresji. Dzięki użyciu różnych typów jąder SVM mogą skutecznie obsługiwać zarówno dane liniowo rozdzielne, jak i nieliniowo rozdzielne. Python, z bogatym ekosystemem bibliotek, zapewnia wygodną platformę do implementacji SVM i eksperymentowania z różnymi jądrami. Zrozumienie SVM i jąder jest niezbędne dla każdego aspirującego praktyka AI i ML.

Powody istnienia jąder

Kernele w SVM służą dwóm głównym celom: umożliwiają algorytmowi obsługę nieliniowo rozdzielnych danych i zapewniają bardziej wydajne podejście obliczeniowe. W wielu scenariuszach z życia wziętych dane nie są liniowo rozdzielne, co oznacza, że linia prosta lub hiperpłaszczyzna nie mogą skutecznie oddzielić różnych klas. Kernele rozwiązują ten problem, mapując dane wejściowe na przestrzeń o wyższym wymiarze, gdzie możliwe staje się znalezienie liniowej granicy decyzyjnej, która oddziela klasy. Dzięki użyciu kerneli SVM może skutecznie obsługiwać dane nieliniowe i osiągać lepszą dokładność klasyfikacji.

Jedną z kluczowych zalet stosowania jąder w SVM jest to, że pozwalają one na użycie liniowego klasyfikatora w wielowymiarowej przestrzeni cech bez jawnego obliczania transformacji. Jest to znane jako sztuczka jądra. Sztuczka jądra unika konieczności jawnego obliczania przekształconej przestrzeni cech, co może być kosztowne obliczeniowo, szczególnie w przypadku dużych zestawów danych. Zamiast tego funkcja jądra niejawnie oblicza iloczyn skalarny między przekształconymi wektorami cech, co sprawia, że SVM jest wydajny obliczeniowo.

Istnieją różne typy jąder, które można stosować w SVM, każde z własnymi cechami i przydatnością do różnych typów danych. Niektóre z powszechnie używanych jąder obejmują jądro liniowe, jądro wielomianowe, jądro Gaussa (RBF) i jądro sigmoidalne. Jądro liniowe jest najprostszą formą jądra i nadaje się do danych liniowo rozdzielnych. Oblicza iloczyn skalarny między wektorami wejściowymi, zachowując oryginalną przestrzeń cech.

Jądro Gaussa (RBF) jest popularnym wyborem do obsługi danych nieliniowych. Przekształca dane w nieskończenie wymiarową przestrzeń cech, w której podobieństwo między dwoma punktami danych jest mierzone ich odległością w przestrzeni cech. Jądro Gaussa przypisuje wyższe wagi pobliskim punktom, przechwytując lokalną strukturę danych i umożliwiając SVM efektywne rozdzielanie nieliniowo rozdzielalnych klas.

Jądro sigmoidalne to inny typ jądra używanego w SVM. Jest ono szczególnie przydatne w przypadku zadań klasyfikacji binarnej i opiera się na funkcji sigmoidalnej. Jądro sigmoidalne mapuje dane do przestrzeni cech, w której podobieństwo między dwoma punktami jest określane przez funkcję sigmoidalną. To jądro jest często używane w aplikacjach, w których granica decyzyjna musi uchwycić złożone nieliniowe relacje.

Kernele odgrywają ważną rolę w SVM, umożliwiając algorytmowi obsługę nieliniowo rozdzielnych danych i zapewniając bardziej wydajne podejście obliczeniowe. Umożliwiają SVM transformację danych wejściowych do

przestrzeni cech o wyższym wymiarze, w której łatwiej jest znaleźć hiperpłaszczyznę oddzielającą klasy. Dzięki użyciu różnych typów kerneli SVM może skutecznie obsługiwać różne typy danych i osiągać lepszą dokładność klasyfikacji.

SVM z miękkim marginesem

Aby zrozumieć koncepcję SVM z miękkim marginesem, ważne jest, aby najpierw zrozumieć pojęcie SVM z twardym marginesem. W SVM z twardym marginesem algorytm zakłada, że dane są liniowo separowalne, co oznacza, że hiperpłaszczyzna może idealnie oddzielić klasy bez żadnej błędnej klasyfikacji. Jednak w scenariuszach z życia wziętych dane są często zaszumione lub zawierają nakładające się obszary, co utrudnia znalezienie idealnego liniowego oddzielenia.

SVM z miękkim marginesem rozwiązują to ograniczenie, dopuszczając pewien stopień błędnej klasyfikacji. Osiąga się to poprzez wprowadzenie zmiennej luzu dla każdego przykładu szkoleniowego, która mierzy stopień, w jakim punkt danych narusza margines lub kończy po niewłaściwej stronie granicy decyzyjnej. Celem SVM z miękkim marginesem jest zminimalizowanie sumy zmiennych luzu przy jednoczesnym maksymalizowaniu marginesu.

Równowaga między maksymalizacją marginesu a minimalizacją błędnej klasyfikacji jest kontrolowana przez parametr regularyzacji, często oznaczany jako C. Mniejsza wartość C pozwala na szerszy margines, ale może tolerować większą błędną klasyfikację, podczas gdy większa wartość C prowadzi do węższego marginesu, ale wymusza bardziej rygorystyczne zasady klasyfikacji. Wybór C zależy od konkretnego zestawu danych i pożądanego kompromisu między rozmiarem marginesu a błędną klasyfikacją.

Jednym z powszechnych podejść do rozwiązywania SVM z miękkim marginesem jest programowanie kwadratowe, w którym celem jest minimalizacja kwadratowej funkcji celu podlegającej ograniczeniom liniowym. Ten problem optymalizacji można skutecznie rozwiązać za pomocą różnych algorytmów, takich jak algorytm Sequential Minimal Optimization (SMO) lub metody punktów wewnętrznych.

Aby zaimplementować SVM z miękkim marginesem, możemy użyć biblioteki CVXOPT w Pythonie. CVXOPT to biblioteka optymalizacji wypukłej, która zapewnia przyjazny dla użytkownika interfejs do rozwiązywania problemów optymalizacyjnych. Oferuje ona szereg rozwiązań, w tym rozwiązania do programowania kwadratowego, co jest niezbędne dla SVM.

Po wstępnym przetworzeniu danych możemy zdefiniować problem SVM przy użyciu biblioteki CVXOPT. Celem SVM jest zminimalizowanie następującego problemu optymalizacji kwadratowej:

```
minimize (1/2) * x.T * P * x + q.T * x
subject to G * x <= h
           A * x = b
```

Tutaj `x` jest wektorem współczynników, `P` jest macierzą współczynników kwadratowych, `q` jest wektorem współczynników liniowych, `G` jest macierzą współczynników ograniczeń nierówności, `h` jest wektorem wartości ograniczeń nierówności, `A` jest macierzą współczynników ograniczeń równości, a `b` jest wektorem wartości ograniczeń równości.

Aby rozwiązać ten problem optymalizacji kwadratowej, możemy użyć funkcji `cvxopt.solvers.qp()` dostarczonej przez **CVXOPT**. Funkcja ta przyjmuje macierze i wektory zdefiniowane powyżej i zwraca optymalne rozwiązanie dla `x`.

Po uzyskaniu optymalnego rozwiązania możemy wyodrębnić wektory nośne z rozwiązania i użyć ich do tworzenia prognoz na nowych punktach danych. Wektory nośne to punkty danych, które leżą na marginesie lub po złej stronie hiperpłaszczyzny. Odgrywają one ważną rolę w definiowaniu granicy decyzyjnej SVM.

Oprócz SVM z miękkim marginesem możemy również zwiększyć wydajność SVM, używając jąder. Jądra pozwalają nam przekształcić dane wejściowe do przestrzeni o wyższym wymiarze, gdzie mogą stać się liniowo separowalne. Powszechnie używane jądra obejmują jądro liniowe, jądro wielomianowe i jądro funkcji bazowej radialnej (RBF).

CVXOPT udostępnia funkcje do definiowania różnych typów jąder, takich jak funkcja `cvxopt.solvers.kernel()`. Poprzez włączenie jądra do modelu SVM możemy obsługiwać złożone zestawy danych, które nie są liniowo rozdzielne w oryginalnej przestrzeni cech.

Podsumowując, w tym materiale dydaktycznym zbadaliśmy koncepcję SVM z miękkim marginesem i jądra, korzystając z biblioteki CVXOPT w Pythonie. Nauczyliśmy się, jak zaimplementować model SVM z miękkim marginesem, wstępnie przetworzyć dane, zdefiniować problem SVM, rozwiązać problem optymalizacji, wyodrębnić wektory nośne i wykorzystać jądra w celu zwiększenia wydajności. SVM to potężne narzędzia w uczeniu maszynowym, które można stosować w szerokim zakresie zadań klasyfikacji i regresji.

```
import numpy as np
from numpy import linalg
import cvxopt
import cvxopt.solvers


def linear_kernel(x1, x2):
    return np.dot(x1, x2)

def polynomial_kernel(x, y, p=3):
    return (1 + np.dot(x, y)) ** p

def gaussian_kernel(x, y, sigma=5.0):
    return np.exp(-linalg.norm(x - y) ** 2 / (2 * (sigma ** 2)))
```

```
class SVM(object):

    def __init__(self, kernel=linear_kernel, C=None):
        self.kernel = kernel
        self.C = C
        if self.C is not None: self.C = float(self.C)

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Gram matrix
        K = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = self.kernel(X[i], X[j])

        P = cvxopt.matrix(np.outer(y, y) * K)
        q = cvxopt.matrix(np.ones(n_samples) * -1)
        A = cvxopt.matrix( *args: y, (1, n_samples))
        b = cvxopt.matrix(0.0)
```

```
        if self.C is None:
            G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
            h = cvxopt.matrix(np.zeros(n_samples))
        else:
            tmp1 = np.diag(np.ones(n_samples) * -1)
            tmp2 = np.identity(n_samples)
            G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
            tmp1 = np.zeros(n_samples)
            tmp2 = np.ones(n_samples) * self.C
            h = cvxopt.matrix(np.hstack((tmp1, tmp2)))

        # solve QP problem
        solution = cvxopt.solvers.qp(P, q, G, h, A, b)

        # Lagrange multipliers
        a = np.ravel(solution['x'])
```

```
        # Support vectors have non zero lagrange multipliers
        sv = a > 1e-5
        ind = np.arange(len(a))[sv]
        self.a = a[sv]
        self.sv = X[sv]
        self.sv_y = y[sv]
        print("%d support vectors out of %d points" % (len(self.a), n_samples))

        # Intercept
        self.b = 0
        for n in range(len(self.a)):
            self.b += self.sv_y[n]
            self.b -= np.sum(self.a * self.sv_y * K[ind[n], sv])
        self.b /= len(self.a)
```

```
        # Weight vector
        if self.kernel == linear_kernel:
            self.w = np.zeros(n_features)
            for n in range(len(self.a)):
                self.w += self.a[n] * self.sv_y[n] * self.sv[n]
        else:
            self.w = None
```

```
    def project(self, X):
        if self.w is not None:
            return np.dot(X, self.w) + self.b
        else:
            y_predict = np.zeros(len(X))
            for i in range(len(X)):
                s = 0
                for a, sv_y, sv in zip(self.a, self.sv_y, self.sv):
                    s += a * sv_y * self.kernel(X[i], sv)
                y_predict[i] = s
            return y_predict + self.b

    def predict(self, X):
        return np.sign(self.project(X))
```

```
if __name__ == "__main__":
    import pylab as pl


    def gen_lin_separable_data():
        # generate training data in the 2-d case
        mean1 = np.array([0, 2])
        mean2 = np.array([2, 0])
        cov = np.array([[0.8, 0.6], [0.6, 0.8]])
        X1 = np.random.multivariate_normal(mean1, cov, size: 100)
        y1 = np.ones(len(X1))
        X2 = np.random.multivariate_normal(mean2, cov, size: 100)
        y2 = np.ones(len(X2)) * -1
        return X1, y1, X2, y2
```

```python
def gen_non_lin_separable_data():
    mean1 = [-1, 2]
    mean2 = [1, -1]
    mean3 = [4, -4]
    mean4 = [-4, 4]
    cov = [[1.0, 0.8], [0.8, 1.0]]
    X1 = np.random.multivariate_normal(mean1, cov, size: 50)
    X1 = np.vstack((X1, np.random.multivariate_normal(mean3, cov, size: 50)))
    y1 = np.ones(len(X1))
    X2 = np.random.multivariate_normal(mean2, cov, size: 50)
    X2 = np.vstack((X2, np.random.multivariate_normal(mean4, cov, size: 50)))
    y2 = np.ones(len(X2)) * -1
    return X1, y1, X2, y2
```

```python
def gen_lin_separable_overlap_data():
    # generate training data in the 2-d case
    mean1 = np.array([0, 2])
    mean2 = np.array([2, 0])
    cov = np.array([[1.5, 1.0], [1.0, 1.5]])
    X1 = np.random.multivariate_normal(mean1, cov, size: 100)
    y1 = np.ones(len(X1))
    X2 = np.random.multivariate_normal(mean2, cov, size: 100)
    y2 = np.ones(len(X2)) * -1
    return X1, y1, X2, y2
```

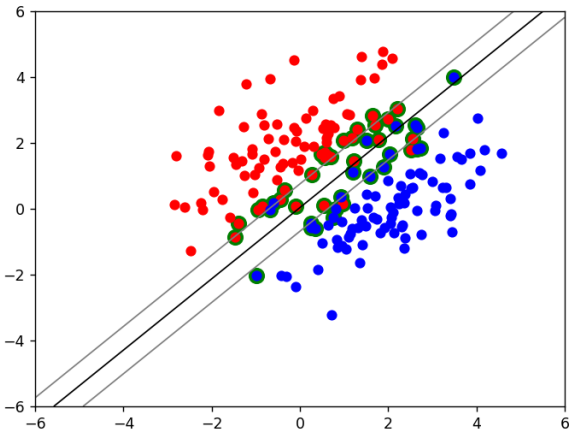
```python
def split_train(X1, y1, X2, y2):
    X1_train = X1[:90]
    y1_train = y1[:90]
    X2_train = X2[:90]
    y2_train = y2[:90]
    X_train = np.vstack((X1_train, X2_train))
    y_train = np.hstack((y1_train, y2_train))
    return X_train, y_train
```

```python
def split_test(X1, y1, X2, y2):
    X1_test = X1[90:]
    y1_test = y1[90:]
    X2_test = X2[90:]
    y2_test = y2[90:]
    X_test = np.vstack((X1_test, X2_test))
    y_test = np.hstack((y1_test, y2_test))
    return X_test, y_test
```

```python
def plot_margin(X1_train, X2_train, clf):
    def f(x, w, b, c=0):
        # given x, return y such that [x,y] in on the line
        # w.x + b = c
        return (-w[0] * x - b + c) / w[1]

    pl.plot( *args: X1_train[:, 0], X1_train[:, 1], "ro")
    pl.plot( *args: X2_train[:, 0], X2_train[:, 1], "bo")
    pl.scatter(clf.sv[:, 0], clf.sv[:, 1], s=100, c="g")

    # w.x + b = 0
    a0 = -4;
    a1 = f(a0, clf.w, clf.b)
    b0 = 4;
    b1 = f(b0, clf.w, clf.b)
    pl.plot( *args: [a0, b0], [a1, b1], "k")

    # w.x + b = 1
    a0 = -4;
    a1 = f(a0, clf.w, clf.b, c: 1)
    b0 = 4;
    b1 = f(b0, clf.w, clf.b, c: 1)
    pl.plot( *args: [a0, b0], [a1, b1], "k--")

    # w.x + b = -1
    a0 = -4;
    a1 = f(a0, clf.w, clf.b, -1)
    b0 = 4;
    b1 = f(b0, clf.w, clf.b, -1)
    pl.plot( *args: [a0, b0], [a1, b1], "k--")

    pl.axis("tight")
    pl.show()
```

```python
def plot_contour(X1_train, X2_train, clf):
    pl.plot( *args: X1_train[:, 0], X1_train[:, 1], "ro")
    pl.plot( *args: X2_train[:, 0], X2_train[:, 1], "bo")
    pl.scatter(clf.sv[:, 0], clf.sv[:, 1], s=100, c="g")

    X1, X2 = np.meshgrid( *xi: np.linspace(-6, stop: 6, num: 50), np.linspace(-6, stop: 6, num: 50))
    X = np.array([[x1, x2] for x1, x2 in zip(np.ravel(X1), np.ravel(X2))])
    Z = clf.project(X).reshape(X1.shape)
    pl.contour( *args: X1, X2, Z, [0.0], colors='k', linewidths=1, origin='lower')
    pl.contour( *args: X1, X2, Z + 1, [0.0], colors='grey', linewidths=1, origin='lower')
    pl.contour( *args: X1, X2, Z - 1, [0.0], colors='grey', linewidths=1, origin='lower')

    pl.axis("tight")
    pl.show()
```

```python
def test_linear():
    X1, y1, X2, y2 = gen_lin_separable_data()
    X_train, y_train = split_train(X1, y1, X2, y2)
    X_test, y_test = split_test(X1, y1, X2, y2)

    clf = SVM()
    clf.fit(X_train, y_train)

    y_predict = clf.predict(X_test)
    correct = np.sum(y_predict == y_test)
    print("%d out of %d predictions correct" % (correct, len(y_predict)))

    plot_margin(X_train[y_train == 1], X_train[y_train == -1], clf)


def test_non_linear():
    X1, y1, X2, y2 = gen_non_lin_separable_data()
    X_train, y_train = split_train(X1, y1, X2, y2)
    X_test, y_test = split_test(X1, y1, X2, y2)

    clf = SVM(polynomial_kernel)
    clf.fit(X_train, y_train)
    y_predict = clf.predict(X_test)
    correct = np.sum(y_predict == y_test)
    print("%d out of %d predictions correct" % (correct, len(y_predict)))

    plot_contour(X_train[y_train == 1], X_train[y_train == -1], clf)

def test_soft():
    X1, y1, X2, y2 = gen_lin_separable_overlap_data()
    X_train, y_train = split_train(X1, y1, X2, y2)
    X_test, y_test = split_test(X1, y1, X2, y2)

    clf = SVM(C=1000.1)
    clf.fit(X_train, y_train)

    y_predict = clf.predict(X_test)
    correct = np.sum(y_predict == y_test)
    print("%d out of %d predictions correct" % (correct, len(y_predict)))

    plot_contour(X_train[y_train == 1], X_train[y_train == -1], clf)

# test_linear()
# test_non_linear()
test_soft()
```



Parametry SVM

1. Funkcja jądra:
Funkcja jądra jest ważnym składnikiem SVM, ponieważ mapuje dane wejściowe do przestrzeni o wyższym wymiarze, gdzie stają się one liniowo rozdzielne. SVM używają różnych funkcji jądra, takich jak liniowa, wielomianowa, radialna funkcja bazowa (RBF) i sigmoidalna. Każde jądro ma własny zestaw parametrów, które można dostroić, aby zoptymalizować wydajność SVM.

2. Parametr regularizacji (C):
Parametr regularizacji (C) kontroluje kompromis między osiągnięciem niskiego błędu w danych treningowych a minimalizacją złożoności granicy decyzyjnej. Mniejsza wartość C prowadzi do szerszego marginesu i umożliwia więcej błędnych klasyfikacji, podczas gdy większa wartość C skutkuje węższym marginesem i mniejszą liczbą błędnych klasyfikacji. Ważne jest znalezienie optymalnej wartości C, aby zapobiec nadmiernemu lub niedostatecznemu dopasowaniu.

3. Parametr gamma:
Parametr gamma (γ) określa wpływ pojedynczego przykładu treningowego na granicę decyzyjną. Niska wartość gamma wskazuje na szerszy zakres wpływu, podczas gdy wysoka wartość gamma koncentruje się na pobliskich punktach. Wyższa wartość gamma może prowadzić do nadmiernego dopasowania, podczas gdy niższa wartość może skutkować niedopasowaniem. Istotne jest, aby wybrać odpowiednią wartość gamma, aby zrównoważyć złożoność modelu i zdolność do generalizacji.

4. Parametr stopnia:
Parametr stopnia (d) jest specyficzny dla jąder wielomianowych i kontroluje stopień funkcji wielomianowej używanej do klasyfikacji. Wyższe wartości stopnia mogą uchwycić bardziej złożone relacje w danych, ale mogą również zwiększyć ryzyko nadmiernego dopasowania. Wybór odpowiedniej wartości stopnia jest ważny, aby osiągnąć równowagę między złożonością modelu a uogólnieniem.

5. Parametr współczynnika:
Parametr współczynnika (coef0) jest kolejnym parametrem używanym w jądrach wielomianowych i sigmoidalnych. Kontroluje on wpływ wielomianów wysokiego stopnia na granicę decyzyjną. Wyższa wartość coef0 może prowadzić

do nadmiernego dopasowania, podczas gdy niższa wartość może skutkować niedopasowaniem. Ważne jest eksperymentowanie z różnymi wartościami coef0, aby znaleźć optymalną równowagę.

6. Wagi klas:
W SVM wagi klas mogą być przypisywane w celu adresowania niezrównoważonych zestawów danych, w których jedna klasa ma znacznie mniej próbek niż inne. Przypisując wyższe wagi klasie mniejszościowej, SVM mogą nadać jej większe znaczenie podczas procesu szkolenia. Pomaga to w osiągnięciu zrównoważonej granicy decyzyjnej i poprawia ogólną wydajność klasyfikatora.

7. Przeszukiwanie siatki i walidacja krzyżowa:
Aby określić optymalną kombinację parametrów SVM, powszechnie stosuje się techniki przeszukiwania siatki i walidacji krzyżowej. Przeszukiwanie siatki obejmuje systematyczne badanie różnych kombinacji parametrów i ocenę ich wydajności za pomocą walidacji krzyżowej. Wybierając kombinację parametrów, która daje najwyższą dokładność lub inną metrykę oceny, możemy dostroić SVM do konkretnego zadania.

KLASTROWANIE, K-MEANS, K-SHIFT

Klastrowanie to podstawowa technika w uczeniu maszynowym, która polega na grupowaniu podobnych punktów danych. Jest ona szczególnie przydatna, gdy chcemy odkryć wzorce lub struktury w zestawie danych bez wcześniejszej wiedzy lub oznaczonych przykładów. W tym materiale dydaktycznym przyjrzymy się dwóm popularnym algorytmom klastrowania: k-means i przesunięciu średniej, i nauczymy się, jak je wdrożyć za pomocą Pythona.

K-means, k-shift

K-means to prosty i szeroko stosowany algorytm klastrowania, którego celem jest partycjonowanie zbioru danych na k odrębnych klastrów. Algorytm rozpoczyna się od losowego zainicjowania k centroidów klastra i iteracyjnie przypisuje każdy punkt danych do najbliższego centroidu. Następnie przelicza centroidy na podstawie średniej punktów danych przypisanych do każdego klastra. Proces ten trwa do momentu konwergencji, w której centroidy nie zmieniają się już znacząco lub osiągnięto maksymalną liczbę iteracji.
Jedną z kluczowych zalet k-means jest jej wydajność, dzięki czemu nadaje się do dużych zestawów danych. Ma jednak pewne ograniczenia. K-means wymaga określenia liczby klastrów, k, z góry, co może być trudne, gdy optymalna liczba klastrów jest nieznana. Ponadto k-means jest wrażliwy na początkowe pozycje centroidów, co potencjalnie skutkuje różnymi przypisaniami klastrów dla różnych inicjalizacji.

Mean shift to kolejny algorytm klastrowania, który nie wymaga wcześniejszego określenia liczby klastrów. Zamiast tego iteracyjnie przesuwa każdy punkt danych w kierunku mody jego lokalnego sąsiedztwa aż do zbieżności. Moda reprezentuje najwyższą gęstość punktów danych w danym promieniu. Powstałe klastry są definiowane przez punkty zbieżności. W porównaniu do k-means, przesunięcie średniej jest bardziej elastyczne, ponieważ automatycznie określa liczbę klastrów na podstawie rozkładu danych. Może obsługiwać klastry o nieregularnych kształtach i jest mniej wrażliwe na początkowe pozycje punktów danych. Jednak przesunięcie średniej może być kosztowne obliczeniowo, szczególnie w przypadku dużych zestawów danych.

Aby użyć metody k-means, najpierw musimy zaimportować niezbędne moduły:

```
from sklearn.cluster import KMeans
```

Następnie tworzymy instancję klasy KMeans i określamy żądaną liczbę klastrów:

```
kmeans = KMeans(n_clusters=3)
```

Następnie możemy dopasować model do naszych danych i uzyskać przypisania klastrów:

```
kmeans.fit(data)
```

```
labels = kmeans.labels_
```

Podobnie, aby użyć przesunięcia średniej, importujemy wymagany moduł:

```
from sklearn.cluster import MeanShift
```

Tworzymy instancję klasy MeanShift i określamy przepustowość, która kontroluje rozmiar lokalnego sąsiedztwa:

```
meanshift = MeanShift(bandwidth=0.5)
```

Dopasowujemy model do naszych danych i uzyskujemy przypisania klastrów:

```
meanshift = MeanShift(bandwidth=0.5)
```

Gdy już mamy przydziały klastrów, możemy analizować i wizualizować wyniki. Na przykład możemy nanieść punkty danych na wykres w różnych kolorach, reprezentujących różne klastry. Możemy również obliczyć różne metryki, takie jak wynik sylwetki, aby ocenić jakość klastrowania.

Skalowanie
Skalowanie jest kolejnym czynnikiem branym pod uwagę przy stosowaniu algorytmów klastrowania. Każdy punkt danych musi zostać porównany ze wszystkimi innymi punktami, co może być kosztowne obliczeniowo. Jednak po wytrenowaniu algorytmu i ustaleniu centrów klastrów nowe punkty można klasyfikować na podstawie tych centroidów bez konieczności dalszego szkolenia.

W praktyce algorytmy klastrowania mogą być używane w półnadzorowanym uczeniu maszynowym. Po klastrowaniu grupy można przetłumaczyć na nadzorowane uczenie maszynowe, gdzie klastry służą jako etykiety dla zadań klasyfikacyjnych przy użyciu innych algorytmów, takich jak maszyny wektorów nośnych.

Aby zastosować algorytm k-means, możemy użyć scikit-learn, popularnej biblioteki uczenia maszynowego w Pythonie. Scikit-learn zapewnia prostą implementację algorytmu k-means, dzięki czemu można go łatwo zastosować w rzeczywistych przykładach.

Algorytm klastrowania k-means jest szeroko stosowaną techniką grupowania podobnych punktów danych. Iteracyjne przypisywanie punktów danych do klastrów i aktualizowanie centrów klastrów do momentu osiągnięcia zbieżności. Może jednak mieć trudności z klastrowaniem grup o różnej wielkości ze względu na przestrzeganie odległości euklidesowej. Skalowanie jest również brane pod uwagę podczas korzystania z algorytmów klastrowania, ale po przeszkoleniu nowe punkty można klasyfikować na podstawie ustalonych centroidów. Scikit-learn zapewnia łatwą w użyciu implementację algorytmu k-means do praktycznych zastosowań.

W tym materiale dydaktycznym wprowadzimy koncepcję klasteryzacji w kontekście sztucznej inteligencji i uczenia maszynowego przy użyciu Pythona. W szczególności omówimy algorytmy klasteryzacji k-means i przesunięcia średniej.

Na początek musimy zaimportować niezbędne biblioteki. Użyjemy modułu matplotlib.pyplot do wizualizacji danych i modułu sklearn.cluster do algorytmów klastrowania. Zaimportujemy również numpy jako np do operacji numerycznych. Kod do importowania tych bibliotek wygląda następująco:

```
import matplotlib.pyplot as plt

from matplotlib import style

style.use('ggplot')

import numpy as np

from sklearn.cluster import KMeans
```

Następnie zdefiniujemy zestaw wartości początkowych dla naszych danych. Wartości te zostaną zapisane w tablicy numpy. Na przykład możemy użyć następujących wartości: [1, 1,5, 1,85, 8,88, 1,06, 9,11]. Wartości te reprezentują cechy naszych punktów danych. Ważne jest, aby pamiętać, że liczba elementów w tej tablicy może się różnić w zależności od pożądanej liczby klastrów.

Po zdefiniowaniu danych możemy je zwizualizować za pomocą funkcji wykresu punktowego z matplotlib.pyplot. Narysujemy pierwszy i drugi element tablicy jako współrzędne x i y, odpowiednio. Ustawimy rozmiar znaczników na 150, a szerokość linii na 5. Kod do wizualizacji danych wygląda następująco:

```
plt.scatter(x[:,0], x[:,1], s=150, linewidths=5)

plt.show()
```

Po wizualizacji danych możemy przejść do procesu klastrowania. W tym celu wykorzystamy algorytm k-means. Aby utworzyć klasyfikator k-means, musimy zdefiniować liczbę klastrów, która w tym przypadku wynosi 2. Możemy to zrobić, inicjując wystąpienie klasy KMeans z parametrem n_clusters ustawionym na 2. Kod tworzenia klasyfikatora wygląda następująco:

```
clf = KMeans(n_clusters=2)
```

Po utworzeniu klasyfikatora możemy dopasować go do naszych danych, używając metody dopasowania. Przypisze ona każdy punkt danych do jednego z klastrów na podstawie ich podobieństwa. Możemy uzyskać dostęp do centrów klastrów i etykiet, używając odpowiednio atrybutów cluster_centers_ i labels_. Kod dopasowujący klasyfikator i uzyskujący dostęp do atrybutów wygląda następująco:

```
clf.fit(x)

centroids = clf.cluster_centers_

labels = clf.labels_
```

Aby zwizualizować klastry, możemy przypisać różne kolory do punktów danych na podstawie ich etykiet. Możemy utworzyć listę kolorów i przypisać kolor do każdej etykiety. Na przykład możemy użyć zielonego dla etykiety 0 i czerwonego dla etykiety 1. Następnie możemy nanieść każdy punkt danych na wykres z odpowiadającym mu kolorem. Kod do wizualizacji klastrów jest następujący:

```
colors = ["g.", "r."]

for i in range(len(x)):

    plt.plot(x[i][0], x[i][1], colors[labels[i]], markersize=10)

plt.scatter(centroids[:,0], centroids[:,1], marker="x", s=150, linewidths=5)

plt.show()
```

Algorytm k-means jest jednym z najprostszych i najszerzej stosowanych algorytmów klastrowania. Jego celem jest podzielenie danych na k odrębnych klastrów, gdzie każdy punkt danych należy do klastra o najbliższej średniej. Algorytm działa iteracyjnie, zaczynając od początkowego zestawu k centroidów i przypisując każdy punkt danych do najbliższego centroidu. Centroidy są następnie aktualizowane na podstawie średniej punktów danych w każdym klastrze. Ten proces jest powtarzany aż do osiągnięcia konwergencji, kiedy centroidy nie zmieniają się już znacząco.

Z drugiej strony algorytm przesunięcia średniej jest nieparametrycznym algorytmem klastrowania, który nie wymaga wcześniejszego określenia liczby klastrów. Zamiast tego iteracyjnie przesuwa centroidy w kierunku najgęstszych regionów danych. Algorytm zaczyna od początkowego zestawu centroidów i oblicza wektor przesunięcia średniej dla każdego punktu danych, który wskazuje kierunek w kierunku najgęstszego regionu. Centroidy są następnie aktualizowane przez przesuwanie ich w kierunku wektora przesunięcia średniej. Ten proces jest powtarzany aż do uzyskania zbieżności.

W przypadku danych nienumerycznych, takich jak zmienne kategoryczne, konieczne jest przekształcenie ich w reprezentację numeryczną, która może być używana przez algorytmy klastrowania. Jednym z powszechnych podejść jest użycie kodowania one-hot, w którym każda kategoria jest reprezentowana przez cechę binarną. Na przykład, jeśli mamy zmienną kategoryjną „color” z trzema kategoriami (red, green, blue), utworzylibyśmy trzy cechy binarne (color_red, color_green, color_blue) i przypisalibyśmy wartość 1 do odpowiadającej cechy dla każdego punktu danych.

Inną techniką obsługi danych nieliczbowych jest użycie miary podobieństwa, która może porównywać różnice między zmiennymi kategorialnymi. Jedną z takich miar jest współczynnik podobieństwa Jaccarda, który oblicza stosunek przecięcia do sumy dwóch zbiorów. Miary tej można użyć do obliczenia podobieństwa między punktami danych na podstawie ich atrybutów kategorialnych, umożliwiając algorytmom klastrowania pracę z danymi nieliczbowymi.

Aby zilustrować omówione powyżej koncepcje, rozważmy przykład, w którym mamy zbiór danych preferencji klientów dla różnych produktów. Zbiór danych obejmuje zmienne kategoryczne, takie jak kategoria produktu, marka i segment klientów. Możemy użyć algorytmów k-means lub klasteryzacji przesunięcia średniej, aby zidentyfikować grupy klientów o podobnych preferencjach, które następnie mogą być wykorzystane do ukierunkowanego marketingu lub spersonalizowanych rekomendacji.

```
df = pd.read_excel('titanic.xls')
df.drop(['body', 'name'], axis=1, inplace=True)
#przekonwertowanie  kolumnn w DataFrame na liczby:
df = df.apply(pd.to_numeric, errors='coerce')
df.fillna(0, inplace=True)

#funkcja konwertująca na na dane liczbowe
def handle_non_numerical_data(df):
    columns = df.columns.values
    for column in columns:
        text_digit_vals = {}
        def convert_to_int(val):
            return text_digit_vals[val]

        if df[column].dtype != np.int64 and df[column].dtype != np.float64:
            column_contents = df[column].values.tolist()
            unique_elements = set(column_contents)
            x = 0
            for unique in unique_elements:
                if unique not in text_digit_vals:
                    text_digit_vals[unique] = x
                    x+=1

            df[column] = list(map(convert_to_int, df[column]))

    return df
df = handle_non_numerical_data(df)
print(df.head())
```

Przed zastosowaniem algorytmu k-means konieczne jest wstępne przetworzenie danych. Ten krok obejmuje obsługę brakujących wartości, skalowanie cech numerycznych i kodowanie zmiennych kategorycznych. Dla uproszczenia załóżmy, że zbiór danych jest już wstępnie przetworzony.

Aby zastosować klasterowanie k-means, musimy wybrać liczbę klastrów, k. Ten wybór zależy od problemu i może wymagać pewnej wiedzy domenowej. Możemy użyć metody łokcia, aby określić optymalną wartość k. Metoda łokcia polega na wykreśleniu sumy kwadratów wewnątrz klastra (WCSS) względem liczby klastrów i wybraniu wartości k, przy której zmiana WCSS zaczyna się stabilizować.

```
wcss = []

for i in range(1, 11):

    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)

    kmeans.fit(data)

    wcss.append(kmeans.inertia_)
```

Po ustaleniu odpowiedniej wartości k możemy zastosować algorytm k-średnich do zbioru danych:

```
kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)

kmeans.fit(data)
```

Po dopasowaniu modelu k-means możemy uzyskać dostęp do etykiet klastra przypisanych do każdego punktu danych, używając atrybutu `labels_`. Możemy również uzyskać współrzędne centroidów klastra, używając atrybutu `cluster_centers_`.

Przesunięcie średniej
Przesunięcie średniej to kolejny algorytm klastrowania, który nie wymaga wcześniejszego określenia liczby klastrów. Zamiast tego automatycznie odkrywa liczbę klastrów i ich lokalizacje w danych. Algorytm przesunięcia średniej działa poprzez iteracyjne przesuwanie punktów danych w kierunku trybu oszacowania gęstości jądra. Proces ten trwa do momentu osiągnięcia konwergencji.

Aby zastosować klasterowanie przesunięcia średniej w Pythonie, możemy użyć klasy `MeanShift` z biblioteki sklearn:

from sklearn.cluster import MeanShift

ms = MeanShift()

ms.fit(data)

Podobnie jak w przypadku k-means, możemy uzyskać dostęp do etykiet klastra przypisanych do każdego punktu danych, używając atrybutu `labels_`. Dodatkowo, możemy uzyskać współrzędne centroidów klastra, używając atrybutu `cluster_centers_`.

Algorytmy klastrowania, takie jak k-means i mean shift, są potężnymi narzędziami do grupowania podobnych punktów danych. W tym materiale dydaktycznym zbadaliśmy algorytmy k-means i mean shift przy użyciu Pythona i zastosowaliśmy je do zbioru danych Titanic. Dzięki zrozumieniu i zastosowaniu tych technik możemy uzyskać cenne spostrzeżenia i podejmować decyzje oparte na danych w różnych domenach.

```
#[...]
df = handle_non_numerical_data(df)
print(df.head())

df.drop(['sex','boat'], axis=1, inplace=True)
#clustering
X = np.array(df.drop(['survived'], axis=1).astype(float))
y = np.array(df['survived'])

clf = KMeans(n_clusters=2)
clf.fit(X)

#W przypadku algorytmu klastrowania maszyna znajdzie klastry,
# ale następnie przypisze im dowolne wartości, w kolejności,
# w jakiej je znajdzie. Tak więc grupa survivors może być 0 lub 1,
# w zależności od stopnia losowości
correct = 0
for i in range(len(X)):
    predict_me = np.array(X[i].astype(float))
    predict_me = predict_me.reshape(-1, len(predict_me))
    prediction = clf.predict(predict_me)
    if prediction[0] == y[i]:
        correct += 1

print(correct/len(X))
```

```
   pclass  survived  sex      age  ...  cabin  embarked  boat  home.dest
0       1         1  0.0  29.0000  ...    0.0       0.0   2.0        0.0
1       1         1  0.0   0.9167  ...    0.0       0.0  11.0        0.0
2       1         0  0.0   2.0000  ...    0.0       0.0   0.0        0.0
3       1         0  0.0  30.0000  ...    0.0       0.0   0.0        0.0
4       1         0  0.0  25.0000  ...    0.0       0.0   0.0        0.0

[5 rows x 12 columns]
0.6103896103896104
```

Chociaż k-means jest potężnym algorytmem, ma pewne ograniczenia. Zakłada, że klastry są sferyczne i mają równe wariancje. Ponadto wymaga wcześniejszego określenia liczby klastrów (k). Jednak w wielu scenariuszach z życia wziętych optymalna liczba klastrów jest nieznana. Aby rozwiązać te ograniczenia, opracowano alternatywne algorytmy klastrowania, takie jak przesunięcie średniej.

Mean shift to nieparametryczny algorytm klastrowania, który nie wymaga wcześniejszego określenia liczby klastrów. Zamiast tego iteracyjnie przesuwa centroidy w kierunku trybu rozkładu danych, w którym gęstość punktów danych jest najwyższa. Pozwala to na automatyczne określenie liczby klastrów przez mean shift na podstawie danych. Algorytm rozpoczyna się od początkowych centroidów i oblicza wektor przesunięcia średniej dla każdego centroidu. Centroidy są następnie aktualizowane przez przesuwanie ich w kierunku wektora przesunięcia średniej. Ten proces jest powtarzany aż do uzyskania zbieżności.

Oprócz k-means i przesunięcia średniej możliwe jest również tworzenie niestandardowych algorytmów klastrowania dostosowanych do konkretnych potrzeb. Jednym z przykładów jest niestandardowy algorytm k-means, który rozszerza podstawowy algorytm k-means poprzez włączenie dodatkowych ograniczeń lub rozważań. Ta personalizacja pozwala algorytmowi lepiej dopasować się do konkretnych wymagań danego problemu.

Niestandardowy algorytm k-means
Niestandardowy algorytm k-means można wdrożyć, modyfikując standardowy algorytm k-means, aby uwzględnić dodatkowe kroki lub ograniczenia. Na przykład można wprowadzić ograniczenie, które zapewnia, że każdy klaster ma minimalną liczbę punktów danych. Można to osiągnąć, dostosowując krok aktualizacji centroidu, aby zapobiec zbytniemu zmniejszaniu się klastrów. Inna personalizacja może obejmować włączenie wiedzy o domenie lub wcześniejszych informacji o danych do procesu klastrowania.

Aby zaimplementować niestandardowy algorytm k-means w Pythonie, można zacząć od podstawowego algorytmu k-means i dodać pożądane dostosowania. Może to obejmować modyfikację kroku aktualizacji centroidu, wprowadzenie dodatkowych ograniczeń lub włączenie dodatkowych kroków wstępnego przetwarzania danych. Poprzez dostosowanie algorytmu możliwe staje się zajęcie się konkretnymi wyzwaniami lub wymaganiami danego problemu.

Klastrowanie to potężna technika w uczeniu maszynowym służąca do grupowania podobnych punktów danych. Algorytm k-means jest szeroko stosowany do klasteryzacji, ale ma ograniczenia, takie jak zakładanie klastrów sferycznych i wymaganie, aby liczba klastrów była określona z góry. Przesunięcie średniej to alternatywny algorytm klasteryzacji, który nie wymaga określania liczby klastrów i może automatycznie ją określić na podstawie danych. Ponadto można opracować niestandardowe algorytmy k-means w celu spełnienia określonych wymagań lub ograniczeń. Poprzez dostosowywanie algorytmów klasteryzacji możliwe staje się dostosowanie ich do konkretnych potrzeb danego problemu.

```
import matplotlib.pyplot as plt
from matplotlib import style
style.use('ggplot')
import numpy as np

X = np.array([[1, 2],
              [1.5, 1.8],
              [5, 8 ],
              [8, 8],
              [1, 0.6],
              [9,11]])

plt.scatter(X[:,0], X[:,1], s=150)
plt.show()
colors = 10*["g","r","c","b","k"]

#będziemy wybierać K=2. Zaczniemy budować naszą klasę K Means:
class K_Means:
    def __init__(self, k=2, tol=0.001, max_iter=300):
        self.k = k
        self.tol = tol
        self.max_iter = max_iter

    def fit(self,data):

        self.centroids = {}
```

```
        for i in range(self.k):
            self.centroids[i] = data[i]

        for i in range(self.max_iter):
            self.classifications = {}

            for i in range(self.k):
                self.classifications[i] = []

            for featureset in data:
                distances = [np.linalg.norm(featureset-self.centroids[centroid])
                             for centroid in self.centroids]
                classification = distances.index(min(distances))
                self.classifications[classification].append(featureset)
```

```
#Następnie musimy utworzyć nowe centroidy, a także zmierzyć ruch centroidów.
#Jeśli ten ruch jest mniejszy niż nasza tolerancja ( self.tol), to wszystko gotowe.
            prev_centroids = dict(self.centroids)

            for classification in self.classifications:
                self.centroids[classification] = (
                    np.average(self.classifications[classification],
                               axis=0))
```

Poszczególne kroki algorytmu k-średnich:
1. Wybierz liczbę klastrów k, które chcesz utworzyć.

2. Zainicjuj k centroidów klastra losowo lub przy użyciu wstępnie zdefiniowanej metody.

3. Przypisz każdy punkt danych do najbliższego centroidu klastra na podstawie odległości euklidesowej.

4. Zaktualizuj centroidy klastra, obliczając średnią wszystkich punktów danych przypisanych do każdego klastra.

5. Powtarzaj kroki 3 i 4 do momentu uzyskania zbieżności, tj. gdy przypisania klastrów nie zmienią się już znacząco.

Teraz zanurkujmy w implementację klastrowania k-means od podstaw przy użyciu Pythona. Użyjemy biblioteki NumPy do wydajnych obliczeń numerycznych.

Najpierw musimy zaimportować niezbędne biblioteki:

```
import numpy as np

import matplotlib.pyplot as plt
```

Następnie definiujemy funkcję `k_means`, która przyjmuje jako dane wejściowe zbiór danych i liczbę klastrów:

```
def k_means(dataset, k):

    # Step 2: Initialize cluster centroids

    centroids = initialize_centroids(dataset, k)

    # Initialize an array to store the cluster assignments

    cluster_assignments = np.zeros(len(dataset))

    # Repeat until convergence

    while True:

        # Step 3: Assign data points to the nearest cluster centroid
```

```
        new_cluster_assignments = assign_to_nearest_centroid(dataset, centroids)
        # Step 4: Update cluster centroids
        new_centroids = update_centroids(dataset, new_cluster_assignments, k)
        # Check for convergence
        if np.array_equal(new_cluster_assignments, cluster_assignments):
            break
        # Update cluster assignments and centroids
        cluster_assignments = new_cluster_assignments
        centroids = new_centroids
    return cluster_assignments, centroids
```

Przyjrzyjmy się implementacji krok po kroku:

1. Definiujemy funkcję `initialize_centroids`, aby losowo zainicjować centroidy klastra.

2. Definiujemy funkcję `assign_to_nearest_centroid`, aby przypisać każdy punkt danych do najbliższego centroidu klastra na podstawie odległości euklidesowej.

3. Definiujemy funkcję `update_centroids`, aby aktualizować centroidy klastra, obliczając średnią wszystkich punktów danych przypisanych do każdego klastra.

4. Powtarzamy kroki 3 i 4 aż do uzyskania zbieżności, która jest określana przez sprawdzenie, czy przypisania klastrów pozostają niezmienione.

Gdy algorytm k-means zbiegnie się, otrzymujemy przypisania klastrów i ostateczne centroidy klastrów. Następnie możemy użyć tych wyników do analizy i wizualizacji wyników klastrowania.

Klastrowanie metodą k-means to potężna technika grupowania podobnych punktów danych. Implementując ją od podstaw przy użyciu Pythona, zyskujemy głębsze zrozumienie wewnętrznych mechanizmów algorytmu. Wiedzę tę można zastosować do różnych rzeczywistych problemów, takich jak segmentacja klientów, kompresja obrazu i wykrywanie anomalii.

Estymacja gęstości jądra
Aby zrozumieć klasterowanie przesunięć średnich, omówmy najpierw koncepcję estymacji gęstości jądra. Estymacja gęstości jądra to technika używana do szacowania podstawowej funkcji gęstości prawdopodobieństwa zestawu punktów danych. Polega ona na umieszczeniu funkcji jądra, zazwyczaj Gaussa, w każdym punkcie danych i zsumowaniu wkładów ze wszystkich jąder w celu uzyskania estymacji gęstości w dowolnym punkcie przestrzeni danych.

W klasteryzacji przesunięcia średniej estymacja gęstości jądra jest używana do obliczenia gradientu funkcji gęstości w każdym punkcie danych. Gradient wskazuje w kierunku najstromszego wznoszenia, co odpowiada modzie rozkładu gęstości. Wektor przesunięcia średniej jest następnie obliczany jako różnica między bieżącym punktem danych a szacowaną modą, a punkt danych jest przesuwany w kierunku wektora przesunięcia średniej.

Proces przesunięcia średniej jest powtarzany iteracyjnie dla wszystkich punktów danych, aż do osiągnięcia zbieżności. Zbieżność jest zwykle definiowana jako niewielkie przesunięcie punktów danych lub gdy wektor przesunięcia średniej staje się pomijalny. Pod koniec procesu punkty danych są przypisywane do trybu, do którego się zbiegają, co skutkuje klastrowaniem danych.

Jedną z zalet klasteryzacji przesunięcia średniej jest możliwość automatycznego określenia liczby klastrów w danych. W przeciwieństwie do algorytmu k-means, który wymaga wcześniejszego określenia liczby klastrów, klasteryzacja przesunięcia średniej może wykryć liczbę klastrów na podstawie samych danych. Dzięki temu jest ona szczególnie

użyteczna w scenariuszach, w których liczba klastrów jest nieznana lub gdy dane nie tworzą naturalnie dobrze rozdzielonych klastrów.

Aby zaimplementować klasterowanie przesunięcia średniej w Pythonie, możemy wykorzystać bibliotekę scikit-learn, która zapewnia wygodną implementację algorytmu. Implementacja scikit-learn pozwala nam określić różne parametry, takie jak szerokość pasma funkcji jądra, która kontroluje gładkość estymacji gęstości, oraz kryterium zatrzymania dla zbieżności.

W Pythonie możemy wykorzystać bibliotekę scikit-learn do wykonania klasteryzacji przesunięcia średniej. Poniższe kroki opisują proces:

1. Załaduj zbiór danych Titanic do Pandas DataFrame.
2. Wstępnie przetwórz zbiór danych, obsługując brakujące wartości i kodując zmienne kategoryczne.
3. Wybierz odpowiednie cechy do klastrowania, takie jak wiek, taryfa i płeć.
4. Skaluj wybrane cechy, używając odpowiedniej techniki skalowania, takiej jak standaryzacja lub normalizacja.
5. Zaimportuj klasę MeanShift z modułu sklearn.cluster.
6. Utwórz instancję klasy MeanShift z odpowiednimi parametrami, takimi jak przepustowość.
7. Dopasuj model przesunięcia średniej do skalowanych danych, używając metody dopasowania.
8. Uzyskaj etykiety klastrów dla każdego punktu danych, używając atrybutu labels_ dopasowanego modelu.
9. Przeanalizuj powstałe klastry i zinterpretuj ustalenia.

Badając klastry wygenerowane przez klasterowanie przesunięć średnich, możemy uzyskać wgląd w wzorce przeżycia pasażerów Titanica. Na przykład możemy odkryć, że pewne klastry mają wyższy odsetek ocalałych, co wskazuje, że pasażerowie o podobnych cechach mieli większe szanse na przeżycie.

Algorytmy klastrowania, takie jak k-means i mean shift, dostarczają cennych narzędzi do analizowania i rozumienia wzorców w zestawach danych. Stosując klasterowanie mean shift do zestawu danych Titanic, możemy odkryć ukryte struktury i uzyskać wgląd w wzorce przetrwania pasażerów. Python, z bogatym ekosystemem bibliotek, takich jak scikit-learn, umożliwia nam skuteczne wdrażanie i eksplorowanie tych algorytmów.

```python
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np
#from sklearn import preprocessing, cross_validation
import pandas as pd
from sklearn.cluster import KMeans, MeanShift
from sklearn import datasets, preprocessing, svm, neighbors
from sklearn.model_selection import (cross_validate,
                                     train_test_split)
from sklearn.linear_model import LinearRegression

#EXECUTE
df = pd.read_excel('titanic.xls')
original_df = pd.DataFrame.copy(df)
df.drop(['body', 'name'], axis=1, inplace=True)
# df.convert_objects(convert_numeric=True)
print(df.head())
df.fillna(0, inplace=True)
```

```python
def handle_non_numerical_data(df):
#[...]


df = handle_non_numerical_data(df)
df.drop(['ticket','home.dest'], axis=1, inplace=True)
print(df.head())
print(df.info())

X = np.array(df.drop(['survived'], axis=1).astype(float))
X = preprocessing.scale(X)
y = np.array(df['survived'])

clf = MeanShift()
clf.fit(X)
```

```
labels = clf.labels_
cluster_centers = clf.cluster_centers_

original_df['cluster_group']=np.nan
print(original_df.head())
print(original_df.info())

for i in range(len(X)):
    original_df.loc[i, 'cluster_group'] = labels[i]

n_clusters_ = len(np.unique(labels))
survival_rates = {}
for i in range(n_clusters_):
    temp_df = original_df[(original_df['cluster_group'] == float(i))]
    # print(temp_df.head())

    survival_cluster = temp_df[(temp_df['survived'] == 1)]

    survival_rate = len(survival_cluster) / len(temp_df)
    # print(i,survival_rate)
    survival_rates[i] = survival_rate

print(survival_rates)
```

```
Survival rates
{0: 0.372626582278481, 1: 1.0, 2: 0.1, 3: 0.7586206896551724}
```

```
print(original_df[ (original_df['cluster_group']==1) ])
```

```
      pclass  survived  ...                        home.dest cluster_group
0          1         1  ...                      St Louis, MO           1.0
1          1         1  ...     Montreal, PQ / Chesterville, ON           1.0
2          1         0  ...     Montreal, PQ / Chesterville, ON           1.0
4          1         0  ...     Montreal, PQ / Chesterville, ON           1.0
6          1         1  ...                        Hudson, NY           1.0
10         1         0  ...                      New York, NY           1.0
11         1         1  ...                      New York, NY           1.0
16         1         0  ...                      Montreal, PQ           1.0
17         1         1  ...                      Montreal, PQ           1.0
24         1         1  ...                               NaN           1.0
```

```
print(original_df[ (original_df['cluster_group']==0) ].describe())
print(original_df[ (original_df['cluster_group']==2) ].describe())
```

```
            pclass     survived  ...        body  cluster_group
count  1248.000000  1248.000000  ...  115.000000         1248.0
mean      2.342147     0.370994  ...  160.973913            0.0
std       0.813333     0.483264  ...   98.660277            0.0
min       1.000000     0.000000  ...    1.000000            0.0
25%       2.000000     0.000000  ...   69.500000            0.0
50%       3.000000     0.000000  ...  165.000000            0.0
75%       3.000000     1.000000  ...  257.000000            0.0
max       3.000000     1.000000  ...  328.000000            0.0

[8 rows x 8 columns]
```

```
       pclass   survived        age  ...        fare       body  cluster_group
count    47.0  47.000000  47.000000  ...   47.000000    4.00000           47.0
mean      1.0   0.680851  37.189717  ...  197.843438  119.25000            2.0
std       0.0   0.471186  17.661868  ...   59.051881   16.52019            0.0
min       1.0   0.000000   0.916700  ...   83.158300   96.00000            2.0
25%       1.0   0.000000  24.000000  ...  151.550000  115.50000            2.0
50%       1.0   1.000000  38.000000  ...  211.500000  123.00000            2.0
75%       1.0   1.000000  50.000000  ...  262.375000  126.75000            2.0
max       1.0   1.000000  67.000000  ...  263.000000  135.00000            2.0

[8 rows x 8 columns]
```

Podsumowanie klastrowania

Klastrowanie to technika uczenia się bez nadzoru, której celem jest identyfikacja inherentnych struktur lub wzorców w zestawie danych. Algorytm k-means to prosta, ale potężna metoda klastrowania, która partycjonuje dane na k

odrębnych klastrów. Działa poprzez iteracyjne przypisywanie każdego punktu danych do najbliższego centroidu klastra i aktualizowanie centroidów na podstawie średniej przypisanych punktów. Proces ten trwa do momentu konwergencji, w której przypisanie punktów do klastrów pozostaje niezmienione.

Przesunięcie średniej z dynamiczną szerokością pasma jest szczególnie przydatne w przypadku zbiorów danych, które wykazują zmienną gęstość lub nierównomiernie rozłożone klastry. Poprzez adaptacyjne dostosowywanie szerokości pasma algorytm może skutecznie uchwycić podstawową strukturę danych, co skutkuje dokładniejszymi wynikami klastrowania.

Algorytmy klastrowania, takie jak k-means i przesunięcie średniej, są potężnymi narzędziami w uczeniu maszynowym do identyfikowania wzorców w danych. Stosując te algorytmy w Pythonie przy użyciu bibliotek, takich jak scikit-learn, możesz łatwo wykonać klastrowanie na swoich zestawach danych. Ponadto przesunięcie średniej z dynamiczną przepustowością zapewnia elastyczne podejście do klastrowania danych o różnej gęstości. Zrozumienie i wykorzystanie tych algorytmów może znacznie zwiększyć Twoją zdolność do wydobywania cennych spostrzeżeń z danych.

Zmiany w kodzie:

```
class Mean_Shift:
    def __init__(self, radius=100, radius_norm_step = 100):
        self.radius = radius
        self.radius_norm_step = radius_norm_step

    def fit(self, data):
        if self.radius == None:
            all_data_centroid = np.average(data, axis=0)
            all_data_norm = np.linalg.norm(all_data_centroid)
            self.radius = all_data_norm / self.radius_norm_step

        centroids = {}

        for i in range(len(data)):
            centroids[i] = data[i]

    def fit(self, data):
        centroids = {}

        for i in range(len(data)):
            in_bandwidth = []
            centroids[i] = data[i]

        while True:
            new_centroids = []
            for i in centroids:
                in_bandwidth = []
                centroid = centroids[i]
                weights = [i for i in range(self.radius_norm_step)][::-1]


                for featureset in data:
                    # if np.linalg.norm(featureset - centroid) < self.radius:
                    #     in_bandwidth.append(featureset)
                    distance = np.linalg.norm(featureset-centroid)
                    if distance == 0:
                        distance = 0.00000000001
                    weight_index = int(distance/self.radius)
                    if weight_index > self.radius_norm_step-1:
                        weight_index = self.radius_norm_step-1
                    to_add = (weights[weight_index]**2)*[featureset]

                    in_bandwidth +=to_add
                new_centroid = np.average(in_bandwidth, axis=0)
                new_centroids.append(tuple(new_centroid))
```

```
        uniques = sorted(list(set(new_centroids)))

        to_pop = []

        for i in uniques:
            for ii in [i for i in uniques]:
                if i == ii:
                    pass
                elif np.linalg.norm(np.array(i)-np.array(ii)) <= self.radius:
                    #print(np.array(i), np.array(ii))
                    to_pop.append(ii)
                    break

        for i in to_pop:
            try:
                uniques.remove(i)
            except:
                pass
```

```
            for featureset in data:
                # compare distance to either centroid
                distances = [np.linalg.norm(featureset - self.centroids[centroid])
                for centroid in self.centroids]
                # print(distances)
                classification = (distances.index(min(distances)))

                # featureset that belongs to that cluster
                self.classifications[classification].append(featureset)

        def predict(self,data):
            #compare distance to either centroid
            distances = [np.linalg.norm(data-self.centroids[centroid])
            for centroid in self.centroids]
            classification = (distances.index(min(distances)))
            return classification

clf = Mean_Shift()
clf.fit(X)

centroids = clf.centroids
print(centroids)

colors = 10*['r','g','b','c','k','y']

for classification in clf.classifications:
    color = colors[classification]
    for featureset in clf.classifications[classification]:
        plt.scatter(featureset[0],featureset[1], marker = "x", color=color,
        s=200, linewidths = 5, zorder = 10)

for c in centroids:
    plt.scatter(centroids[c][0],centroids[c][1], color='g', marker = "*",
    s=200, linewidths = 5)

plt.show()
```