Paveikslėlis, kuriame yra juodas, tamsa

Automatiškai sugeneruotas aprašymas

**Kauno technologijos universitetas**

Informatikos fakultetas

**P176B101 Intelektikos pagrindai**

**Komandinis darbas**

|  |  |
| --- | --- |
|  |  |
| **Nedas Liaudanskis IFF-1/9**  **Ugnius Rinkevičius IFF-1/9**  **Martynas Kuliešius IFF-1/9**  Studentai | (parašas) (data) |
|  |  |
| **dėst. Nečiūnas Audrius**  **dėst. Nakrošis Arnas**  Dėstytojas | (parašas) (data) |
|  |  |

**KAUNAS, 2024**

Turinys

[1. Įvadas 3](#_Toc167056895)

[2. Duomenų rinkinys 3](#_Toc167056896)

[3. Algoritmai 5](#_Toc167056897)

[K-vidurkių algoritmas 5](#_Toc167056898)

[SOM (Self-Organizing Map) algoritmas 7](#_Toc167056899)

[Veikimo Principas 7](#_Toc167056900)

[Pradinė Inicijacija: 7](#_Toc167056901)

[Mokymo Procesas: 7](#_Toc167056902)

[1. Geriausiai Atitinkančio Vieneto (BMU) Paieška: 7](#_Toc167056903)

[2. Svorio Vektorių Atnaujinimas: 7](#_Toc167056904)

[a. Kaimynystės Funkcija: 7](#_Toc167056905)

[b. Mokymosi Greitis ir Kaimynystės Spindulys: 7](#_Toc167056906)

[3. Algoritmo Sustabdymo Kriterijai 8](#_Toc167056907)

[SOM Pritaikymai: 8](#_Toc167056908)

[4. Skaičiavimai 9](#_Toc167056909)

[K-vidurkių algoritmo skaičiavimai ir algoritmas 9](#_Toc167056910)

[Funkcijos: 10](#_Toc167056911)

[Duomenų skaitymas: 10](#_Toc167056912)

[Algoritmas: 11](#_Toc167056913)

[Testavimas: 12](#_Toc167056914)

[Nitrogen ir Potassium 12](#_Toc167056915)

[Nitroger ir Humidity 13](#_Toc167056916)

[Phosphorus ir potassium 15](#_Toc167056917)

[Potassium ir temperature 16](#_Toc167056918)

[Potassium ir Humidity 17](#_Toc167056919)

[SOM Algoritmo skaičiavimai ir algoritmas 18](#_Toc167056920)

[Funkcijos: 18](#_Toc167056921)

[Duomenų skaitymas: 19](#_Toc167056922)

[Algoritmas: 20](#_Toc167056923)

[Testavimas: 23](#_Toc167056924)

[Nitrogen ir Potassium 23](#_Toc167056925)

[Nitrogen ir Humidity 25](#_Toc167056926)

[Phosphorus ir Potassium 29](#_Toc167056927)

[Potassium ir Temperature 31](#_Toc167056928)

[Potassium ir Humidity 34](#_Toc167056929)

[DBSCAN Algoritmo skaičiavimai ir algoritmas 36](#_Toc167056930)

[Nitrogen ir Potassium 40](#_Toc167056931)

[Nitroger ir Humidity 40](#_Toc167056932)

[Phosphorus ir potassium 42](#_Toc167056933)

[Potassium ir temperature 43](#_Toc167056934)

[Potassium ir Humidity 44](#_Toc167056935)

[5. Algoritmų Palyginimas 45](#_Toc167056936)

[6. Išvados 46](#_Toc167056937)

1. Įvadas

Laboratorinio darbo tikslas yra sukurti ir panaudoti 3 mašininio mokymo algoritmų realizacijas pasirinktam duomenų rinkiniui. Duomenų rinkinį reikią išsamiai aprašyti, pavaizduoti. Algoritmus reikia trumpai aprašyti pavaizduojant jų veikimo principą.

**Darbo eiga:**

1. Pasirinkti duomenų rinkinį, kuris turi klasterių tarp tolydinio tipo duomenų.
2. Duomenų rinkinį sutvarkyti ir paruošti algoritmams(jeigu rinkinyje yra trūkstamų elementų ir netinkamų reikšmių).
3. Kiekvienam komandos nariui išsirinkti vieną mašininį algoritmą, jį realizuoti duomenų rinkiniui ir paaiškinti jo veikimo principą.
4. Gautus rezultatus tarp algoritmų palyginti.
5. Pateikti išvadas.
6. Duomenų rinkinys

Duomenų rinkinį šiam komandiniam darbui, pasirinkome tokį, kuris turi akivaizdžiai matomų klasteriu, kuriuos būtu lengva matyti ir atvaizduoti grafike. Duomenų rinkinys yra sudarytas iš tolydinio tipo duomenų ir vieno kategorinio tipo atributo. Jame netinkamų ar trūkstamų reikšmių nėra, todėl duomenų rinkinio tvarkyti nereikės.

Mūsų pasirinktas duomenų rinkinys pateikia informaciją apie tam tikrą kategorinį atributą „Crop“(Kruopa, lietuviškai), naudojant likusius tolydinius atributus. Jis pasako kokių sąlygų reikia norint užauginti tą „Crop“ atributą. Todėl šis duomenų rinkinys puikiai tinka klasterizavimui.

**Duomenų rinkinys:** <https://www.kaggle.com/datasets/varshitanalluri/crop-recommendation-dataset?select=Crop_Recommendation.csv>

**Duomenų rinkinio atributai:**

* Nitrogen – azoto kiekis dirvožemyje.
* Phosphorus – fosforo kiekis dirvožemyje.
* Potassium – kalio kiekis dirvožemyje.
* Temperature – temperatūra Celsijais.
* Humidity – drėgmė, %.
* pH\_Value – dirvožemio pH vertė.
* Rainfall – lietaus kiekis, mm.
* Crop – kruopos pavadinimas(vienintelis kategorinio tipo atributas).

1. Algoritmai

Šiam komandiniam darbui naudoti pasirinkome šiuos algoritmus:

* K-vidurkių algoritmas(Nedas Liaudanskis).
* SOM (Self-Organizing Map) algoritmas (Martynas Kuliešius)
* DBSCAN (Density-based spatial clustering of applications with noise) algoritmas (Ugnius Rinkevičius)

K-vidurkių algoritmas

K-vidurkio (angl. K-means) algoritmas yra populiarus klasterizacijos metodas, naudojamas duomenų taškų grupavimui į k klasterius, kurie parodo duomenų panašias savybes. Šis algoritmas siekia minimizuoti atstumus tarp duomenų taškų ir klasterių centrų (centroidų), taip užtikrindamas, kad duomenų taškai būtų kuo artimesni jų atitinkamiems centrams. Žemiau pateikiamas išsamus K-vidurkio algoritmo veikimo aprašymas ir paveikslėlis(pav. 1 K-vidurkių algoritmo atvaizdavimas), parodantis algoritmo veikimą.

**Veikimo Principas**

K-vidurkio algoritmas vykdomas keliais etapais:

**Pradiniai Centroidai:**

Pasirenkami k pradinių centroidų (klasterių centrų) taškai. Šie taškai gali būti pasirinkti atsitiktinai iš duomenų rinkinio arba naudojant kitas strategijas, pvz., K-means++ metodą, kuris užtikrina geresnį pradinį centroidų išdėstymą.

**Klasterizacija:**

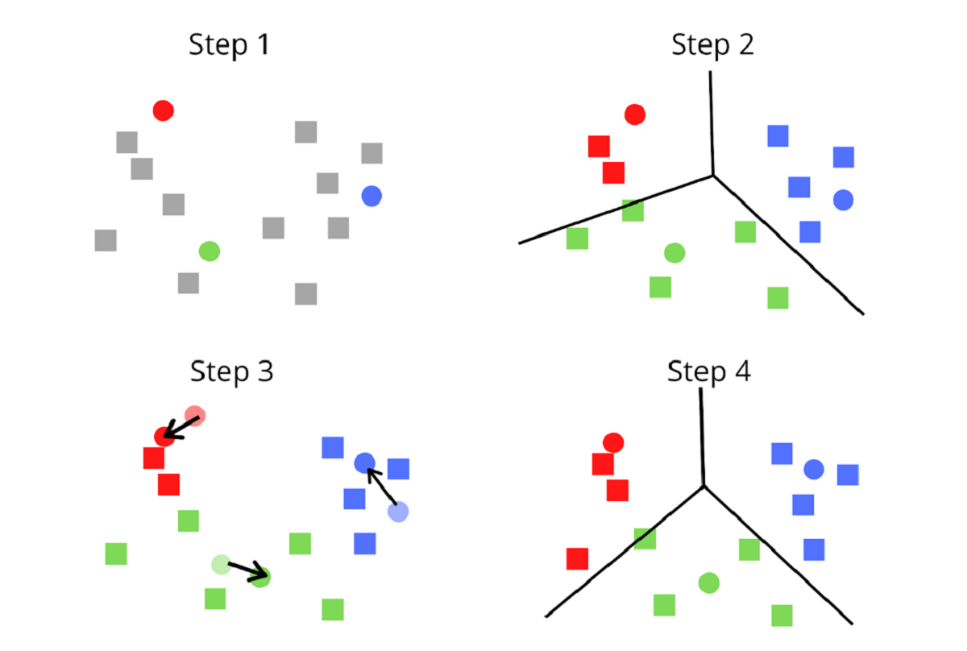
Kiekvienas duomenų taškas priskiriamas artimiausiam centroidui, taip sudarant k klasterį. Artumas paprastai matuojamas naudojant Euklido atstumą, bet gali būti naudojami ir kiti atstumo matavimo metodai.

**Centroidų Atnaujinimas:**

Kiekvieno klasterio centroidas perskaičiuojamas kaip visų to klasterio duomenų taškų vidurkis (tai yra, centroidas tampa naujuoju masyvo centru).

**Kriterijų Patikrinimas:**

Atliekami 2 ir 3 žingsniai tol, kol centroidai nebesikeičia arba kol nėra viršytas maksimalus leistinų iteracijų skaičius. Algoritmas konverguoja, kai duomenų taškų priskyrimai klasteriams stabilizuojasi ir centroidų pozicijos nebekinta.



pav. 1 K-vidurkių algoritmo atvaizdavimas

SOM (Self-Organizing Map) algoritmas

Savaime organizuojančio žemėlapio (SOM, angl. Self-Organizing Map) algoritmas yra dirbtinio neuroninio tinklo tipas, naudojamas neprižiūrimam mokymuisi, labiausiai skirtas duomenų matmenų mažinimui ir vizualizavimui, bet naudojamas ir klasterizacijai. Teuvo Kohonen sukurtas algoritmas siekia organizuoti duomenų taškus į žemėlapį, kuris išlaiko pradiinius duomenų ryšius. Tai leidžia suprasti ir vizualizuoti duomenų struktūrą.

Veikimo Principas

SOM algoritmas vykdomas keliais etapais:

Pradinė Inicijacija:

Sukuriamas tinklelis (dažniausiai 2D) su neuronais, kurių kiekvienas turi svorio vektorių, atitinkantį įvesties duomenų dimensiją. Šie svorio vektoriai dažniausiai inicijuojami atsitiktinai.

Mokymo Procesas:

Mokymas atliekamas iteratyviai koreguojant svorio vektorius pagal įvesties duomenis. Kiekviena iteracija apima šiuos žingsnius:

1. Geriausiai Atitinkančio Vieneto (BMU) Paieška:

Kiekvienam įvesties vektoriui nustatomas neuronas, kurio svorio vektorius yra artimiausias įvesties vektoriui pagal Euklido atstumą. Šis neuronas vadinamas geriausiai atitinkančiu vienetu (BMU).

1. Svorio Vektorių Atnaujinimas:

BMU ir jo kaimyninių neuronų svorio vektoriai koreguojami, kad jie būtų artimesni įvesties vektoriui. Koregavimo dydį kontroliuoja mokymosi greitis ir kaimynystės funkcija, kuri laikui bėgant mažėja.

* 1. Kaimynystės Funkcija:

Kaimynystės funkcija nustato, kiek turi būti koreguojami BMU supantys neuronai. Ji dažniausiai turi Gauss'o formą, kur BMU kaimynai atnaujinami mažiau, kuo jų atstumas nuo BMU didesnis.

* 1. Mokymosi Greitis ir Kaimynystės Spindulys:

Tiek mokymosi greitis, tiek kaimynystės spindulys mažėja laikui bėgant. Pradžioje jie būna dideli, kad leistų reikšmingus koregavimus ir tyrinėjimus, o vėliau sumažėja, kad leistų smulkesnius koregavimus ir konvergenciją.

1. Algoritmo Sustabdymo Kriterijai

SOM mokymosi procesas tęsiasi tol, kol įvykdomi sustabdymo kriterijai, tokie kaip maksimalus iteracijų skaičius, minimali svorio vektorių pokyčių suma arba inercijos pokytis minimalus.

SOM Pritaikymai:

* **Duomenų Vizualizavimas**: Kelių dimensijų duomenų sumažinimas iki 2D ar 3D tinklų, kad būtų lengviau juos tyrinėti.
* **Grupavimas**: Grupavimų nustatymas ir vizualizavimas sudėtinguose duomenų rinkiniuose.
* **Modelių Atpažinimas**: Modelių ir struktūrų duomenyse atpažinimas be išankstinių žinių apie etiketes.
* **Funkcijų Žemėlapiai**: Kelių dimensijų funkcijų susiejimas su mažesnio matmens reprezentacija.

**DBSCAN algoritmas**

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algoritmas yra populiarus klasterizavimo metodas, naudojamas duomenų taškams grupuoti pagal duomenų taškų tankį požymių erdvėje. Skirtingai nuo K-vidurkių, DBSCAN nereikalauja iš anksto nurodyti klasterių skaičiaus ir gali nustatyti bet kokios formos klasterius. Toliau pateikiamas išsamus DBSCAN algoritmo veikimo aprašymas ir paveikslas (1 pav. DBSCAN algoritmo pavaizdavimas), kuriame parodyta, kaip algoritmas veikia.

Veikimo principas

DBSCAN algoritmas įgyvendinamas keliais etapais:

**Pagrindiniai taškai ir apylinkės:**

* Pagrindiniai taškai: Duomenų taškai priskiriami pagrindiniams taškams, jei jie turi bent tam tikrą skaičių kaimyninių taškų (min\_samples), esančių tam tikru atstumu (eps).
* Kaimynystės: Taško kaimynystė apibrėžiama kaip visi taškai, esantys per atstumą eps nuo taško.

**Klasterizavimas:**

* Pagrindiniai taškai ir klasteriai: Jei randamas pagrindinis taškas, sudaromas naujas klasteris. Į klasterį įtraukiami visi jo kaimynystėje esantys taškai. Jei kaimyninis taškas taip pat yra pagrindinis taškas, jo kaimynai į klasterį įtraukiami rekursiškai.
* Pasienio taškai: Taškai, kurie yra pagrindinio taško kaimynystėje, bet neturi pakankamai kaimynų, kad patys būtų pagrindiniai taškai, klasifikuojami kaip pasienio taškai ir įtraukiami į pagrindinio taško klasterį.

**Triukšmo taškai:**

* Triukšmas: Taškai, kurie nėra nei pagrindiniai, nei pasienio taškai, klasifikuojami kaip triukšmas ir nepriskiriami jokiam klasteriui.

**Kriterijų tikrinimas:**

* Klasterio išplėtimas: Algoritmas iteratyviai plečia klasterius tikrindamas kiekvieno pagrindinio taško kaimynystę.
* Baigiama: Algoritmas baigiamas, kai visi taškai yra apdoroti ir negali būti priskirti jokiems kitiems klasteriams.

Atlikdamas šiuos veiksmus, DBSCAN gali nustatyti įvairios formos ir dydžio klasterius ir veiksmingai tvarkyti duomenų triukšmą.

**DBSCAN žingsnių santrauka:**

* Pagrindinių taškų nustatymas: Raskite visus taškus, kurių kaimynai yra ne mažesniu kaip eps atstumu.
* Formuoti klasterius: Iš kiekvieno pagrindinio taško išplėskite klasterius, pridėdami visus pasiekiamus taškus, esančius eps atstumu.
* Klasifikuoti triukšmą: Taškus, kurie nepasiekiami iš jokio pagrindinio taško, pažymėkite kaip triukšmą.

**Pavyzdžio parametrai:**

* eps: Didžiausias atstumas tarp dviejų pavyzdžių, kad jie būtų laikomi esančiais toje pačioje kaimynystėje.
* min\_samples: Pavyzdžių skaičius (arba bendras svoris) kaimynystėje, kad taškas būtų laikomas pagrindiniu tašku.

1. Skaičiavimai

K-vidurkių algoritmo skaičiavimai ir algoritmas

Šiam algoritmui realizuoti buvo naudota „Python“ programavimo aplinka. Kodas naudotas skaičiavimuose remiasi gerai žinoma biblioteka skirta skaičiuoti ir sudaryti klasterius naudojant k-vidurkių algoritmą. Šia algoritmas naudoja **Euklido** atstumo metriką. Apačioje pateiktas kodas aprašo funkcijas, kurias naudosime skaičiavimuose. Šis kodas suranda atstumą tarp taško ir centroidų ir naudojant tą atstumą, priskiria tašką vienam iš klasterių. Po priskyrimų, controidų taškai yra atnaujinami, kad labiau atitiktu duomenų rinkinį.

Funkcijos:

|  |
| --- |
| import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  def calculate\_distance(point1, point2):  return np.sqrt(np.sum((point1 - point2) \*\* 2))  def assign\_clusters(X, centroids):  clusters = np.zeros(len(X))  for i in range(len(X)):  distances = [calculate\_distance(X[i], centroid) for centroid in centroids]  cluster = np.argmin(distances)  clusters[i] = cluster  return clusters  def update\_centroids(X, clusters, n\_clusters):  centroids = np.zeros((n\_clusters, X.shape[1]))  for i in range(n\_clusters):  cluster\_points = X[clusters == i]  if len(cluster\_points) > 0:  centroids[i] = np.mean(cluster\_points, axis=0)  return centroids  def kmeans(X, n\_clusters, max\_iterations=100):  # Initialize centroids randomly  centroids = X[np.random.choice(X.shape[0], n\_clusters, replace=False)]  for \_ in range(max\_iterations):  # Assign clusters  clusters = assign\_clusters(X, centroids)    # Update centroids  new\_centroids = update\_centroids(X, clusters, n\_clusters)  # Check convergence  if np.all(centroids == new\_centroids):  break  centroids = new\_centroids  return clusters, centroids |

Skaičiavimams atlikti naudosime pirmus 1000 duomenų iš duomenų rinkinio, kad sumažintume skaičiavimo laiką.

Duomenų skaitymas:

|  |
| --- |
| data = pd.read\_csv("Crop\_Recommendation.csv")  data = data.head(1000)  attributes = ['Nitrogen','Phosphorus','Potassium','Temperature','Humidity','pH\_Value','Rainfall'] |

Algoritmas prieš atliekant skaičiavimus suranda geriausią klasterių skaičių naudojant „elbow“ metodą. Naudojant šį metodą yra ieškomas toks klasteriu skaičius, kuriame pasikeitimas tarp inercijos(reikšmė nusakanti vidinė klasterio dispersiją), nesikeičia dramatiškai. Mano aprašytame algoritme inercijos pokytis turi būti mažesnis nei **5% arba 10%** pokyčio, priklausomai nuo darytų testų. Algoritmas taip pat apskaičiuoja ir tikslumą, sudaryto grafiko.

Algoritmas:

|  |
| --- |
| for i in range(len(attributes)):  for j in range(i+1, len(attributes)):  X = data[[attributes[i], attributes[j]]].values  # Elbow method to find the optimal number of clusters  inertia = []  accuracies = []  for n\_clusters in range(1, 11):  clusters, centroids = kmeans(X, n\_clusters)  inertia.append(np.sum((X - centroids[clusters.astype(int)]) \*\* 2))  # Calculate accuracy as the percentage of data points correctly assigned to clusters  correct\_assignments = sum(clusters == clusters[:, np.newaxis]) # Counting correctly assigned points  accuracy = correct\_assignments / len(X) \* 100 # Calculate accuracy  accuracies.append(accuracy)  for n\_clusters in range(1, 6):  print(f"Inertia for {n\_clusters} clusters: {inertia[n\_clusters - 1]}")    # Calculate average accuracy  avg\_accuracy = np.mean(accuracies)  print(f"Average accuracy for K-means clustering: {avg\_accuracy:.2f}%")  # Calculate rate of change of inertia  max\_inertia = max(inertia)  inertia\_change = [(inertia[i] - inertia[i+1]) / max\_inertia for i in range(len(inertia)-1)]  print(f"Inertia change: {inertia\_change}")  # Find the number of clusters where the inertia change is less than 20%  num\_clusters = np.argmax(np.array(inertia\_change) < 0.05) + 1  # Perform K-means clustering with the selected number of clusters  clusters, centroids = kmeans(X, num\_clusters)  # Plot the elbow curve  plt.figure(figsize=(10, 6))  plt.plot(range(1, 11), inertia, marker='o', linestyle='--')  plt.xlabel('Number of clusters')  plt.ylabel('Inertia')  plt.title('Elbow Curve')  plt.grid(True)  plt.show()  # Plot the clusters  plt.figure(figsize=(10, 6))  plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis', s=50, alpha=0.5)  plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', c='red', s=200, label='Centroids')  plt.title('K-means Clustering: {} vs {}'.format(attributes[i], attributes[j]))  plt.xlabel(attributes[i])  plt.ylabel(attributes[j])  plt.legend()  plt.grid(True)  plt.show() |

Testavimas:

Algoritmas atlieka k-vidurkių klasterizavimą, tarp kiekvienos tolydinio tipo atributo poros. Prie kiekvienos poros atvaizduojama inercijos pakytis priklausantis nuo klasterių skaičiaus ir inercijos pokytis procentais, naudojamas parikti klasterių skaičių. K-vidurkio algoritmo suklasterizuoto grafiko vaizdas ir tikslumas. Šis skaičiavimas buvo atliktas naudojant du skirtingus, minimalius pokyčio reikalavimus, norint surasti tinkamiausius rezultatus. Apačioje yra pateikiami 5 geriausiai susidarę klasteriai tarp atributų porų, kai inercijos siekia 10% ir 5%.

Nitrogen ir Potassium

Šiame teste aiškiai matosi jog naudojant inerciją, kuri yra mažesnė už 10% duoda mums tikslesnį atsakymą. Tai matoma ir akivaizdžiai atsižvelgus į klasterių kiekius.

|  |  |
| --- | --- |
| **Inercija yra mažesnė nei 10%** | **Inercija yra mažesnė nei 5%** |
| Inertia for 1 clusters: 1011103.695  Inertia for 2 clusters: 498230.0002344708  Inertia for 3 clusters: 225452.28970163123  Inertia for 4 clusters: 148534.77307238575  Inertia for 5 clusters: 137396.16044612316  Average accuracy for K-means clustering: 34.34%  Inertia change: [0.5072414405186494, 0.26978213202241297, 0.0760728271586877, 0.011016291089968365, 0.04180428566655008, 0.00990366272343232, 0.0047607518291937325, 0.02056489599811957, 0.004484582505805133]  Paveikslėlis, kuriame yra tekstas, linija, Grafikas, diagrama  Automatiškai sugeneruotas aprašymas  Paveikslėlis, kuriame yra tekstas, ekrano kopija, Grafikas, linija  Automatiškai sugeneruotas aprašymas | Inertia for 1 clusters: 1011103.695  Inertia for 2 clusters: 498230.0002344708  Inertia for 3 clusters: 225452.28970163123  Inertia for 4 clusters: 148534.77307238575  Inertia for 5 clusters: 124112.16425667828  Average accuracy for K-means clustering: 34.07%  Inertia change: [0.5072414405186494, 0.26978213202241297, 0.0760728271586877, 0.02415440566232672, 0.028666171094191728, 0.010692384202261518, 0.01456667819983923, 0.00990448204421669, 0.004690601664588705] |

Nitroger ir Humidity

Šiame teste aiškiai matosi jog naudojant inerciją, kuri yra mažesnė už 5% duoda mums tikslesnį atsakymą. Tai matoma ir akivaizdžiai atsižvelgus į klasterių kiekius.

|  |  |
| --- | --- |
| **Inercija yra mažesnė nei 10%** | **Inercija yra mažesnė nei 5%** |
| Inertia for 1 clusters: 1248869.8844682504  Inertia for 2 clusters: 722874.4240810182  Inertia for 3 clusters: 323807.1079909527  Inertia for 4 clusters: 231122.22664425176  Inertia for 5 clusters: 187120.31571031638  Average accuracy for K-means clustering: 31.60%  Inertia change: [0.42117715138210177, 0.31954274905106084, 0.07421500229879009, 0.03523338298182337, 0.026313004355472017, 0.0236462741416529, 0.009563028042287789, 0.018031445373680642, 0.008924581409059008] | Inertia for 1 clusters: 1248869.8844682504  Inertia for 2 clusters: 722872.142875453  Inertia for 3 clusters: 323807.1079909527  Inertia for 4 clusters: 231122.22664425176  Inertia for 5 clusters: 195035.3513642353  Average accuracy for K-means clustering: 32.27%  Inertia change: [0.4211789779979835, 0.3195409224351791, 0.07421500229879009, 0.028895624539286322, 0.030293678006713697, 0.014880148262482667, 0.008020269718029887, 0.021024703784398614, 0.004477127225997165] |

Phosphorus ir potassium

Atsižvelgus į tikslumus galima teigti, jog geriau pavaizduoja klasterius kai inerciją yra mažesnė nei 5%. Tačiau akivaizdžių pakyčių neįmanoma pamatyti.

|  |  |
| --- | --- |
| **Inercija yra mažesnė nei 10%** | **Inercija yra mažesnė nei 5%** |
| Inertia for 1 clusters: 641253.431  Inertia for 2 clusters: 445640.08940371615  Inertia for 3 clusters: 148961.3936480164  Inertia for 4 clusters: 88432.91258144104  Inertia for 5 clusters: 65148.374131025615  Average accuracy for K-means clustering: 33.27%  Inertia change: [0.305048413216652, 0.46265436006017996, 0.09439088843888833, 0.03631097679135135, 0.04117058806660414, 0.0060045178795558665, 0.007819174788830015, 0.00026655229416236724, 0.0012974219063550253] | Inertia for 1 clusters: 641253.431  Inertia for 2 clusters: 358680.88078584603  Inertia for 3 clusters: 148961.3936480164  Inertia for 4 clusters: 144792.26018755877  Inertia for 5 clusters: 64869.45037304801  Average accuracy for K-means clustering: 35.39%  Inertia change: [0.44065658997488305, 0.3270461833019489, 0.006501537861491209, 0.12463529386482264, 0.04096798354265731, 0.0070798132496741085, 0.006608080934856282, 0.006466457366674727, 0.0054106836590447245] |

Potassium ir temperature

Atsižvelgus į tikslumus galima teigti, jog geriau pavaizduoja klasterius kai inerciją yra mažesnė nei 10%. Tačiau akivaizdžių pakyčių neįmanoma pamatyti.

|  |  |
| --- | --- |
| **Inercija yra mažesnė nei 10%** | **Inercija yra mažesnė nei 5%** |
| Inertia for 1 clusters: 372237.69550421403  Inertia for 2 clusters: 90478.94464513256  Inertia for 3 clusters: 25454.98237884933  Inertia for 4 clusters: 14971.446941013042  Inertia for 5 clusters: 11689.795311602073  Average accuracy for K-means clustering: 37.04%  Inertia change: [0.7569323425920783, 0.17468398029437915, 0.02816355131265207, 0.008816011030172034, -0.003941733586964075, 0.011619347235613713, 0.0022770187924985395, 0.0020293098956035544, 0.0028753471228553733] | Inertia for 1 clusters: 372237.69550421403  Inertia for 2 clusters: 90478.94464513256  Inertia for 3 clusters: 25454.98237884933  Inertia for 4 clusters: 14971.446941013042  Inertia for 5 clusters: 11689.795311602073  Average accuracy for K-means clustering: 36.25%  Inertia change: [0.7569323425920783, 0.17468398029437915, 0.02816355131265207, 0.008816011030172034, 0.0022520428074028626, 0.0052997537577791545, 0.002396347909320216, 0.0012986948518428575, 0.002052310760988753] |

Potassium ir Humidity

Atsižvelgus į tikslumus galima teigti, jog geriau pavaizduoja klasterius kai inerciją yra mažesnė nei 5%. Tačiau akivaizdžių pakyčių neįmanoma pamatyti.

|  |  |
| --- | --- |
| **Inercija yra mažesnė nei 10%** | **Inercija yra mažesnė nei 5%** |
| Inertia for 1 clusters: 939573.1014682504  Inertia for 2 clusters: 431828.60554489493  Inertia for 3 clusters: 205928.90932637223  Inertia for 4 clusters: 73259.28835262856  Inertia for 5 clusters: 72422.92093789185  Average accuracy for K-means clustering: 35.24%  Inertia change: [0.540399139917814, 0.24042801551631712, 0.1412020211800697, 0.0008901568312563862, 0.04925305105596377, 0.005586937071156585, 0.0034995361765711394, -0.001834789021975696, 0.004058598291537209] | Inertia for 1 clusters: 939573.1014682504  Inertia for 2 clusters: 461691.1695743323  Inertia for 3 clusters: 269786.32260599115  Inertia for 4 clusters: 73259.28835262856  Inertia for 5 clusters: 45902.72450249071  Average accuracy for K-means clustering: 36.31%  Inertia change: [0.5086160205599143, 0.2042468506904419, 0.2091663053638446, 0.029115950432582992, 0.021027257454637162, 0.0034995361765711394, 0.004885484471521103, 0.0014758609851561893, 0.0014494008840408058] |

SOM Algoritmo skaičiavimai ir algoritmas

Šiam algoritmui realizuoti buvo naudota “Python” programavimo kalba. Kodas naudotas aprašyti SOM algoritmą yra skirtas suklasterizuoti ir vizualizuoti duomenis. Algoritmas inicijuoja neuronų tinklelį, kurio kiekvienam neuronui yra atsitiktinai priskiriamas svoris. Mokymo proceso metu, kiekvienam duomenų taškui algoritmas nustato geriausiai atitinkantį vienetą (BMU) ir koreguoja BMU bei jo kaimynų svorius, kad jie būtų artimesni įvesties vektoriui. Tiek BMU radimas, tiek Kaimynų svorių apskaičiavimui algoritmas naudoja Euklido atstumą. Mokymosi greitis ir kaimynystės spindulys laikui bėgant mažėja, kad žemėlapis būtų tiksliau sureguliuotas. Algoritmas vertina klasterizacijos efektyvumą naudodamas silueto koeficientą ir vizualizuoja klasterius bei centroidus, kad suteiktų įžvalgų apie duomenų struktūrą.

Funkcijos:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.metrics import silhouette\_score

# Self-Organizing Map (SOM) class

class SOM:

def \_\_init\_\_(self, m, n, dim, num\_iterations=100, alpha=0.5, sigma=1.0):

self.m = m # Grid height

self.n = n # Grid width

self.dim = dim # Dimensionality of input data

self.num\_iterations = num\_iterations

self.alpha = alpha # Initial learning rate

self.sigma = sigma # Initial neighborhood radius

self.weights = np.random.rand(m, n, dim) # Initialize weights

def \_neighborhood\_function(self, distance, radius):

return np.exp(-distance\*\*2 / (2 \* radius\*\*2))

def \_update\_weights(self, x, bmu\_idx, iteration):

learning\_rate = self.alpha \* np.exp(-iteration / self.num\_iterations)

radius = self.sigma \* np.exp(-iteration / self.num\_iterations)

for i in range(self.m):

for j in range(self.n):

neuron = np.array([i, j])

bmu = np.array(bmu\_idx)

dist = np.linalg.norm(neuron - bmu)

if dist <= radius:

influence = self.\_neighborhood\_function(dist, radius)

self.weights[i, j, :] += influence \* learning\_rate \* (x - self.weights[i, j, :])

def train(self, data):

for iteration in range(self.num\_iterations):

for x in data:

distances = np.linalg.norm(self.weights - x, axis=-1)

bmu\_idx = np.unravel\_index(np.argmin(distances), (self.m, self.n))

self.\_update\_weights(x, bmu\_idx, iteration)

def map\_data(self, data):

mapped\_data = []

for x in data:

distances = np.linalg.norm(self.weights - x, axis=-1)

bmu\_idx = np.unravel\_index(np.argmin(distances), (self.m, self.n))

mapped\_data.append(bmu\_idx)

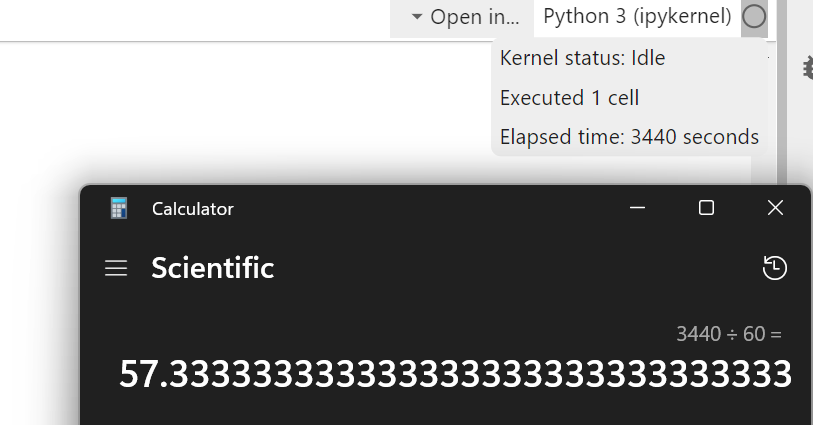
return mapped\_data

def get\_centroids(self):

centroids = self.weights.reshape(-1, self.weights.shape[-1])

return centroids

Skaičiavimams atlikti naudoju pirmus 1000 duomenų, kad rezultatai būtų kuo panašesni į K-vidurkių algoritmo rezultatus, tačiau mano aprašytas algoritmas nėra itin optimalus arba daro papildomų, neitin reikalingų skaičiavimų, dėl kurių jo darbo laikas yra gerokai didesnis negu K-vidurkių algoritmo. Žemiau esančiame paveiksliuke pavaizduotas šio algoritmo darbo laikas, kol perėjo ir suklasterizavo visus pasirinktus duomenis ir atvaizdavo rezultatus.



Duomenų skaitymas:

# Load the dataset

data = pd.read\_csv("Crop\_Recommendation.csv")

data = data.head(1000)

# Select the attributes for clustering

attributes = ['Nitrogen', 'Phosphorus', 'Potassium', 'Temperature', 'Humidity', 'pH\_Value', 'Rainfall']

Algoritmas ciklu eidamas per duomenis skaičiuoja inerciją tarp klasterių skaičiaus, jeigu yra daugiau klasterių nei naudojama apskaičiuoja *silhouette score,*  kuris nustato kaip tiksliai duomenys susiklasterizavo ir šis įvertis yra ribose tarp –1 iki 1, -1 – kai duomenys nesusiklasterizuoja/ netinkamai susiklasterizuoja, 1 – kai duomenys idealiai susiklasterizuoja. Šį įvertį šiuo atveju galime naudoti kaip tikslumo nustatymą. Taip pat algoritmas kiekvienam klasterių kiekiui šį įvertį apskaičiuoja ir taip pat apskaičiuoja inerciją, kurią naudosiu algoritmo darbo nutraukimui. Taip pat algoritmas prieš išvesdamas rezultatus apskaičiuoja ir išgauna klasteriųi centroides, kurias taip pat atvaizduoja. Kai inercijos pokytis yra minimalus, algoritmas stabdo klasterizavimą ir nubraižo lenteles:

* **Inertia vs number of clusters**
* **Silhouette score vs numebr of clusters**
* **Inertia change vs number of clusters**
* **Geriausio klasterių kiekio plot graph.**

Algoritmas:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.metrics import silhouette\_score

# Self-Organizing Map (SOM) class

class SOM:

def \_\_init\_\_(self, m, n, dim, num\_iterations=100, alpha=0.5, sigma=1.0):

self.m = m # Grid height

self.n = n # Grid width

self.dim = dim # Dimensionality of input data

self.num\_iterations = num\_iterations

self.alpha = alpha # Initial learning rate

self.sigma = sigma # Initial neighborhood radius

self.weights = np.random.rand(m, n, dim) # Initialize weights

def \_neighborhood\_function(self, distance, radius):

return np.exp(-distance\*\*2 / (2 \* radius\*\*2))

def \_update\_weights(self, x, bmu\_idx, iteration):

learning\_rate = self.alpha \* np.exp(-iteration / self.num\_iterations)

radius = self.sigma \* np.exp(-iteration / self.num\_iterations)

for i in range(self.m):

for j in range(self.n):

neuron = np.array([i, j])

bmu = np.array(bmu\_idx)

dist = np.linalg.norm(neuron - bmu)

if dist <= radius:

influence = self.\_neighborhood\_function(dist, radius)

self.weights[i, j, :] += influence \* learning\_rate \* (x - self.weights[i, j, :])

def train(self, data):

for iteration in range(self.num\_iterations):

for x in data:

distances = np.linalg.norm(self.weights - x, axis=-1)

bmu\_idx = np.unravel\_index(np.argmin(distances), (self.m, self.n))

self.\_update\_weights(x, bmu\_idx, iteration)

def map\_data(self, data):

mapped\_data = []

for x in data:

distances = np.linalg.norm(self.weights - x, axis=-1)

bmu\_idx = np.unravel\_index(np.argmin(distances), (self.m, self.n))

mapped\_data.append(bmu\_idx)

return mapped\_data

def get\_centroids(self):

centroids = self.weights.reshape(-1, self.weights.shape[-1])

return centroids

# Load the dataset

data = pd.read\_csv("Crop\_Recommendation.csv")

data = data.head(1000)

# Select the attributes for clustering

attributes = ['Nitrogen', 'Phosphorus', 'Potassium', 'Temperature', 'Humidity', 'pH\_Value', 'Rainfall']

# Loop through each pair of attributes

for i in range(len(attributes)):

for j in range(i + 1, len(attributes)):

X = data[[attributes[i], attributes[j]]].values

# Store inertia and silhouette scores for different grid sizes

inertias = []

silhouette\_scores = []

inertia\_changes = []

for grid\_size in range(1, 11): # Increased range for better evaluation

som = SOM(m=grid\_size, n=grid\_size, dim=2, num\_iterations=100, alpha=0.5, sigma=1.0)

som.train(X)

mapped\_data = som.map\_data(X)

# Calculate inertia

inertia = sum(np.min(np.linalg.norm(som.weights - x, axis=-1))\*\*2 for x in X)

inertias.append(inertia)

# Calculate silhouette score if there are at least 2 clusters

if grid\_size > 1:

labels = [x[0] \* grid\_size + x[1] for x in mapped\_data]

silhouette = silhouette\_score(X, labels)

silhouette\_scores.append(silhouette)

print(f"Inertia for {grid\_size} clusters: {inertia}")

print(f"Silhouette Score for {grid\_size} clusters: {silhouette}")

else:

silhouette\_scores.append(-1) # Placeholder for 1 cluster case

# Calculate inertia change

if grid\_size > 1:

inertia\_change = (inertias[-2] - inertias[-1]) / inertias[-2]

inertia\_changes.append(inertia\_change)

if inertia\_change < 0.02: # Stopping criterion

print(f"Stopping at grid size {grid\_size} due to small inertia change: {inertia\_change:.4f}")

break

else:

inertia\_changes.append(None)

# Plot inertia vs number of clusters

plt.figure(figsize=(10, 6))

plt.plot(range(1, len(inertias) + 1), inertias, marker='o', linestyle='--')

plt.xlabel('Number of clusters (Grid size)')

plt.ylabel('Inertia')

plt.title('Inertia vs Number of Clusters')

plt.grid(True)

plt.show()

# Plot silhouette score vs number of clusters, skipping 1 cluster case

plt.figure(figsize=(10, 6))

plt.plot(range(2, len(silhouette\_scores) + 1), silhouette\_scores[1:], marker='o', linestyle='--')

plt.xlabel('Number of clusters (Grid size)')

plt.ylabel('Silhouette Score')

plt.title('Silhouette Score vs Number of Clusters')

plt.grid(True)

plt.show()

# Plot inertia change vs number of clusters

plt.figure(figsize=(10, 6))

plt.plot(range(2, len(inertia\_changes) + 1), inertia\_changes[1:], marker='o', linestyle='--')

plt.xlabel('Number of clusters (Grid size)')

plt.ylabel('Inertia Change')

plt.title('Inertia Change vs Number of Clusters')

plt.grid(True)

plt.show()

# Plot the SOM clusters for the best grid size (highest silhouette score)

best\_grid\_size = np.argmax(silhouette\_scores[1:]) + 2

som = SOM(m=best\_grid\_size, n=best\_grid\_size, dim=2, num\_iterations=100, alpha=0.5, sigma=1.0)

som.train(X)

mapped\_data = som.map\_data(X)

centroids = som.get\_centroids()

labels = [x[0] \* best\_grid\_size + x[1] for x in mapped\_data]

best\_silhouette = silhouette\_score(X, labels)

plt.figure(figsize=(10, 6))

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, alpha=0.5)

plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', c='red', s=200, label='Centroids')

plt.title(f'SOM Clustering: {attributes[i]} vs {attributes[j]} (Best grid size: {best\_grid\_size}x{best\_grid\_size})')

plt.xlabel(attributes[i])

plt.ylabel(attributes[j])

plt.legend()

plt.grid(True)

plt.text(0.05, 0.95, f'Silhouette Score: {best\_silhouette:.2f}', transform=plt.gca().transAxes, fontsize=12, verticalalignment='top')

plt.show()

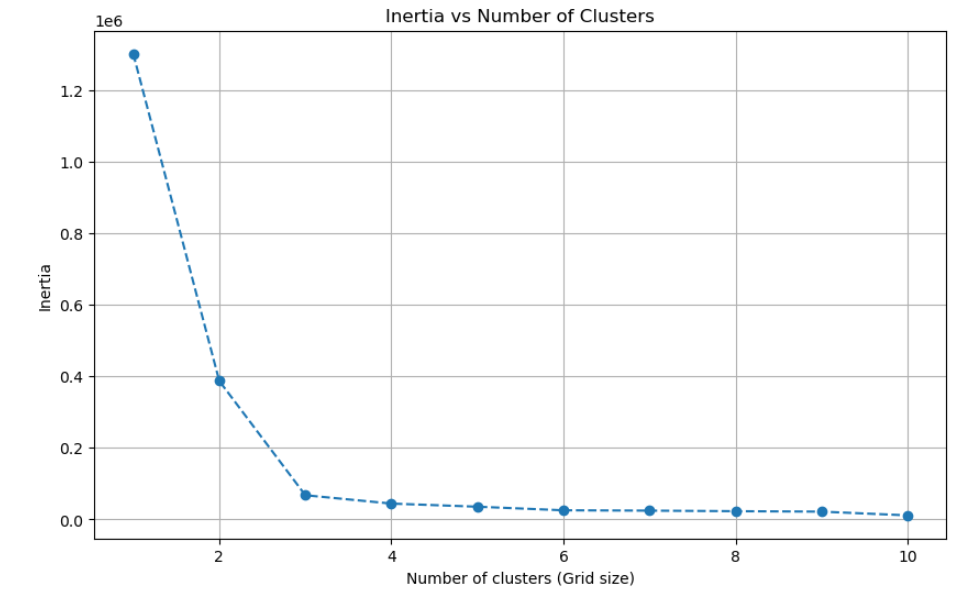
Testavimas:

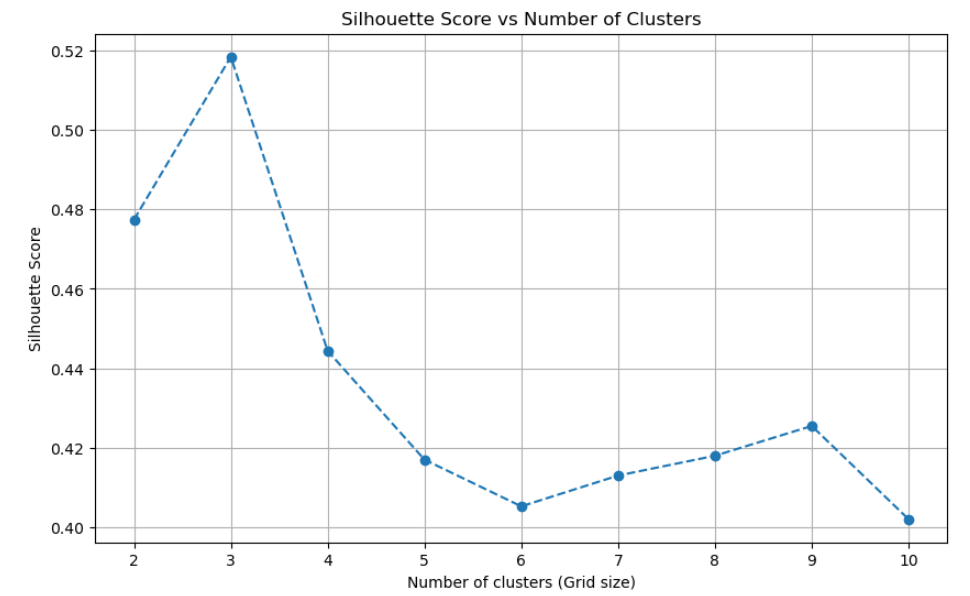
Algoritmu klasterizavimą tarp kiekvienos tolydinio tipo atributo poros, poto kiekvienam klasterių kiekiui apskaičiuoja inerciją ir silueto įvertį. Taip pat atvaizduoja inercijos ir silueto įverčio pokyčius taip klasterių. Testavimams naudosiu tų pačių atributų palyginimus kaip ir K-vidurkių algoritmo testavime:

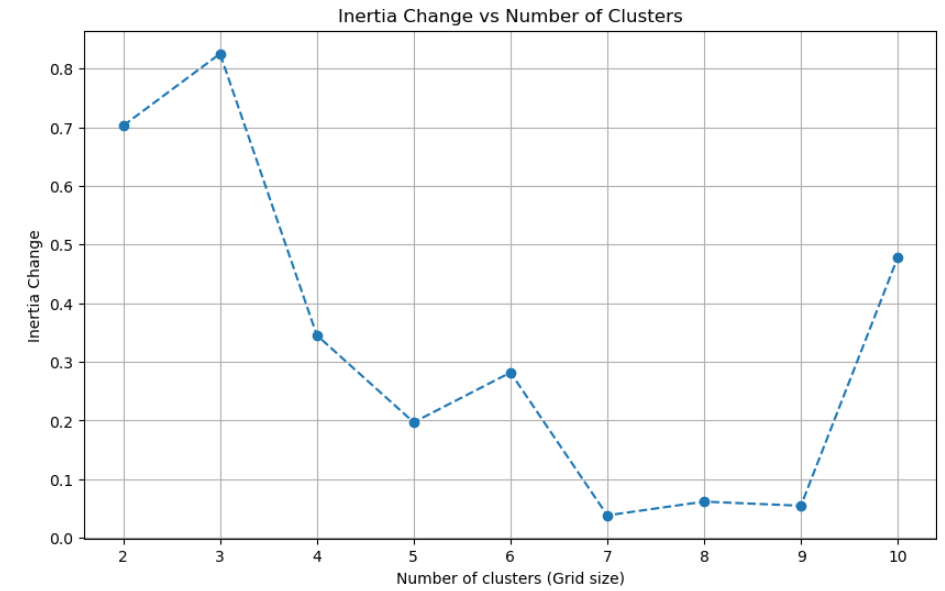
Nitrogen ir Potassium

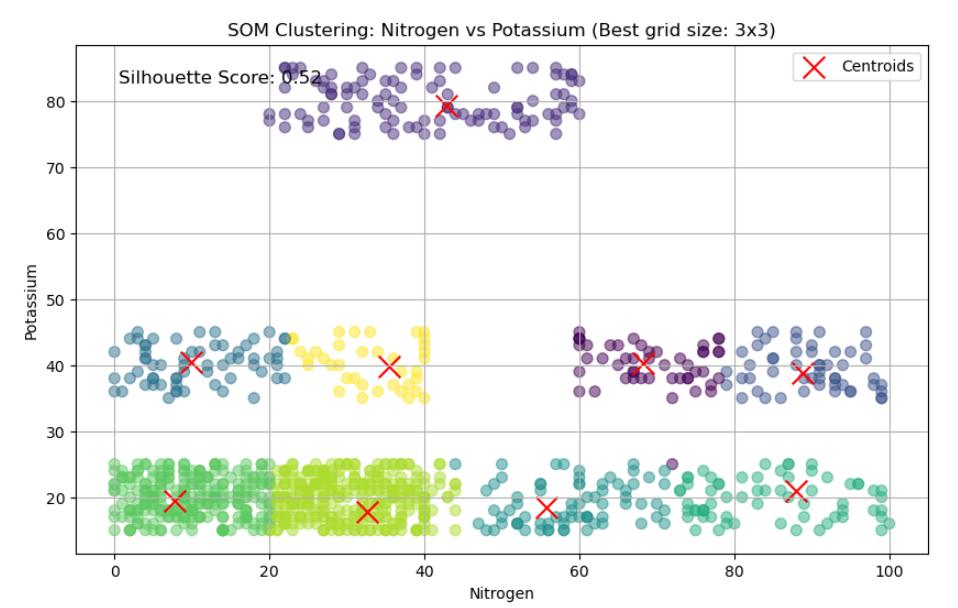
Šiam testavimui galime pažiūrėti kaip algoritmas apskaičiuoja inercijos, silueto įverčius, taip pat kaip apskaičiuoja centroides. Kaip matome, stabdymo funkcija nesustabdė skaičiavimų anksčiau, todėl skaičiavo inercijos ir silueto įverčius iki 10 klasterių. Šiem dviem atributam apskaičiuotas 0.52 silueto įvertis. Ir surado 9 centroides.

Inertia for 2 clusters: 387129.8195414671  
Silhouette Score for 2 clusters: 0.4773839838238481  
Inertia for 3 clusters: 67658.10312591238  
Silhouette Score for 3 clusters: 0.5182947527542455  
Inertia for 4 clusters: 44288.43676995161  
Silhouette Score for 4 clusters: 0.444424273082709  
Inertia for 5 clusters: 35559.379302970396  
Silhouette Score for 5 clusters: 0.41708187420070214  
Inertia for 6 clusters: 25535.60582501271  
Silhouette Score for 6 clusters: 0.4053504092878412  
Inertia for 7 clusters: 24553.408698658895  
Silhouette Score for 7 clusters: 0.41308679116455516  
Inertia for 8 clusters: 23033.989747370433  
Silhouette Score for 8 clusters: 0.41806156561761587  
Inertia for 9 clusters: 21770.450615944814  
Silhouette Score for 9 clusters: 0.4255475625096156  
Inertia for 10 clusters: 11362.003514673559  
Silhouette Score for 10 clusters: 0.4021133461197014





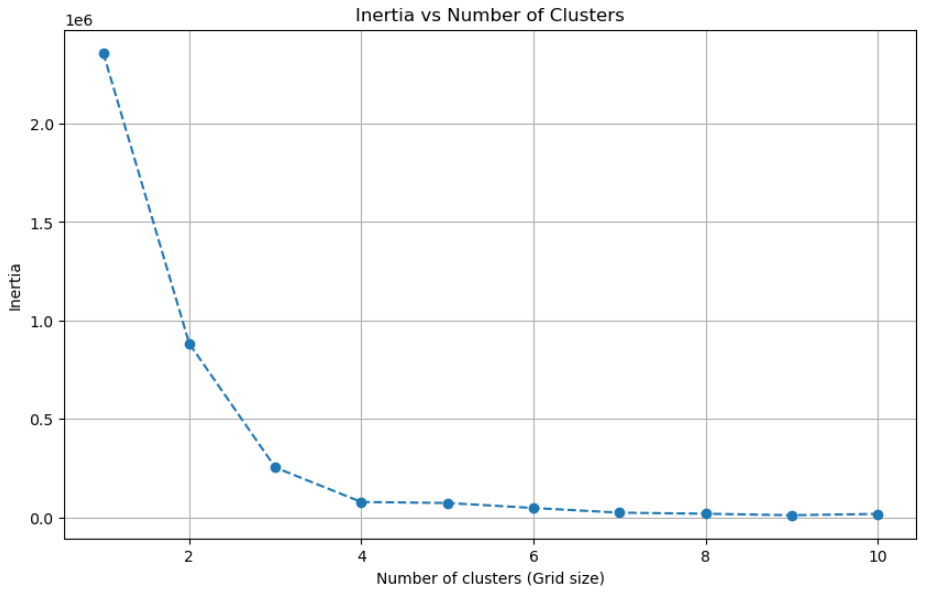


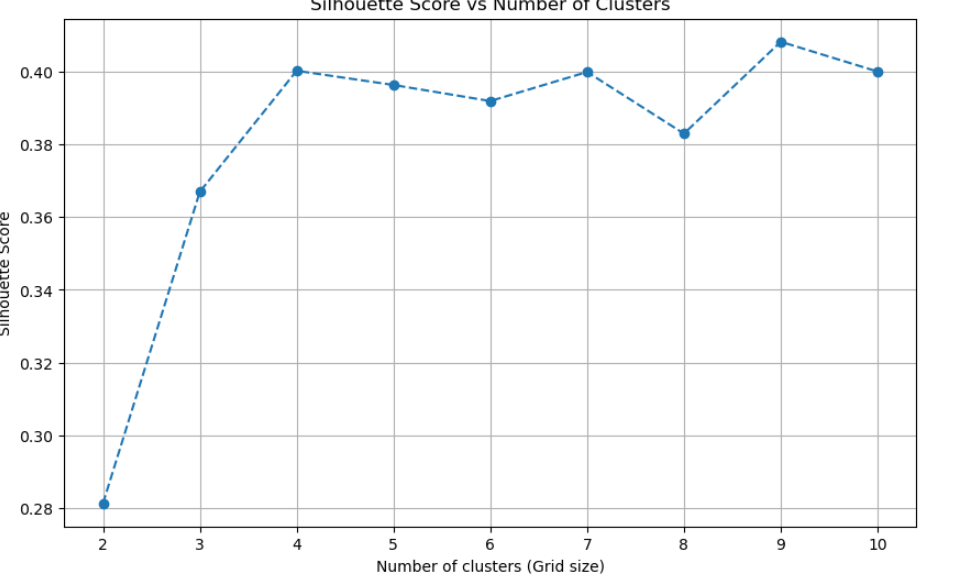


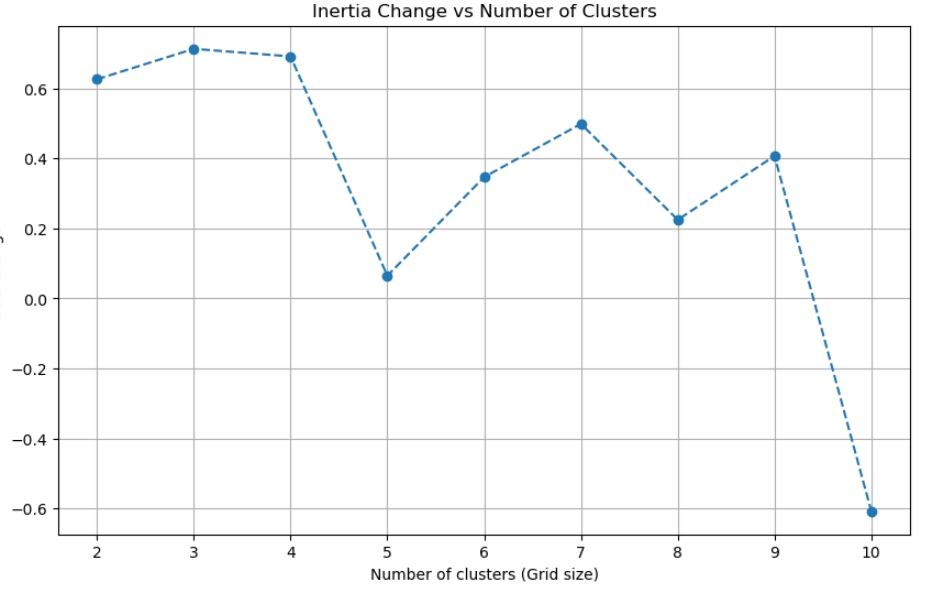
Nitrogen ir Humidity

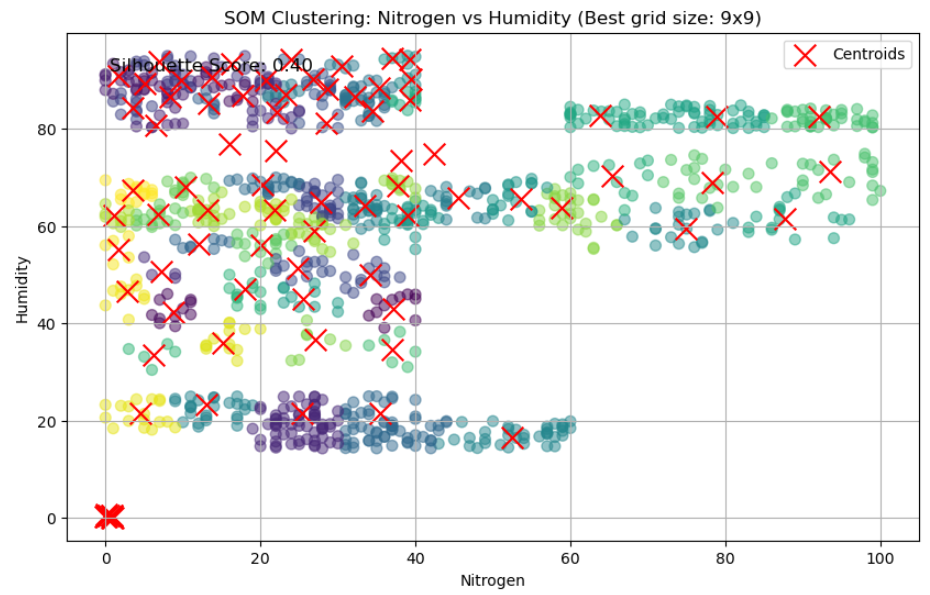
Šiam testavimui galime pažiūrėti kaip algoritmas apskaičiuoja inercijos, silueto įverčius, taip pat kaip apskaičiuoja centroides. Kaip matome, stabdymo funkcija stojo ties 10 klasterių. Šiem dviem atributam geriausias apskaičiuotas silueto įvertis - 0.40. Ir surado 81 centroidę. Pagal jų atvaizdavimą diagramoje galiu teigti, kad kažką neteisingo atliko programa.

Inertia for 2 clusters: 882491.5533005182  
Silhouette Score for 2 clusters: 0.2812531280183719  
Inertia for 3 clusters: 253333.73207076808  
Silhouette Score for 3 clusters: 0.3669720811490795  
Inertia for 4 clusters: 78213.41426179168  
Silhouette Score for 4 clusters: 0.4002792150216032  
Inertia for 5 clusters: 73142.29816629196  
Silhouette Score for 5 clusters: 0.39636703644665444  
Inertia for 6 clusters: 47792.14054812031  
Silhouette Score for 6 clusters: 0.39194745205000653  
Inertia for 7 clusters: 23967.377480912313  
Silhouette Score for 7 clusters: 0.39994143570053975  
Inertia for 8 clusters: 18582.886116514666  
Silhouette Score for 8 clusters: 0.3829809987655174  
Inertia for 9 clusters: 11020.898709168827  
Silhouette Score for 9 clusters: 0.4082610985145087  
Inertia for 10 clusters: 17720.256382654265  
Silhouette Score for 10 clusters: 0.4000650181741107  
Stopping at grid size 10 due to small inertia change: -0.6079





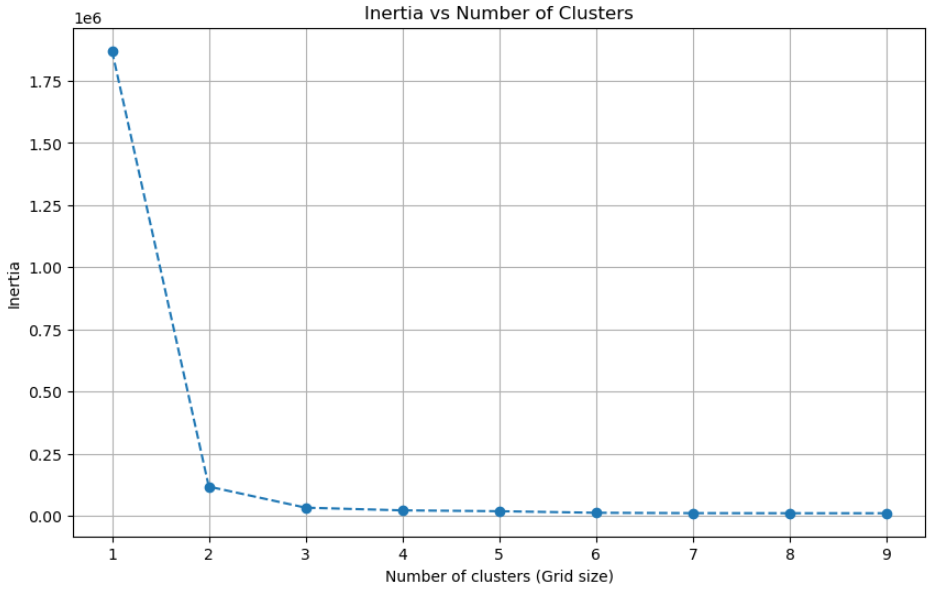


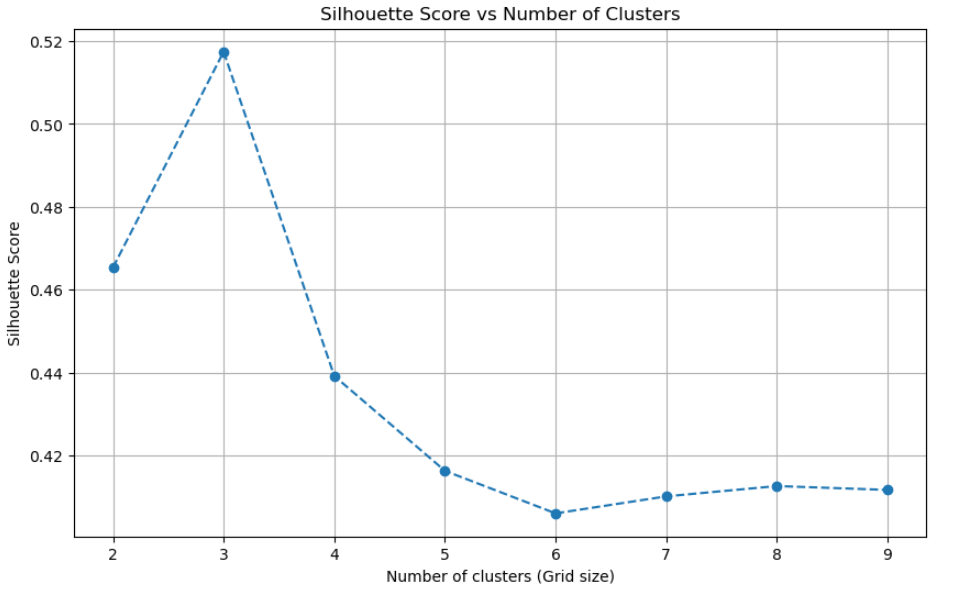


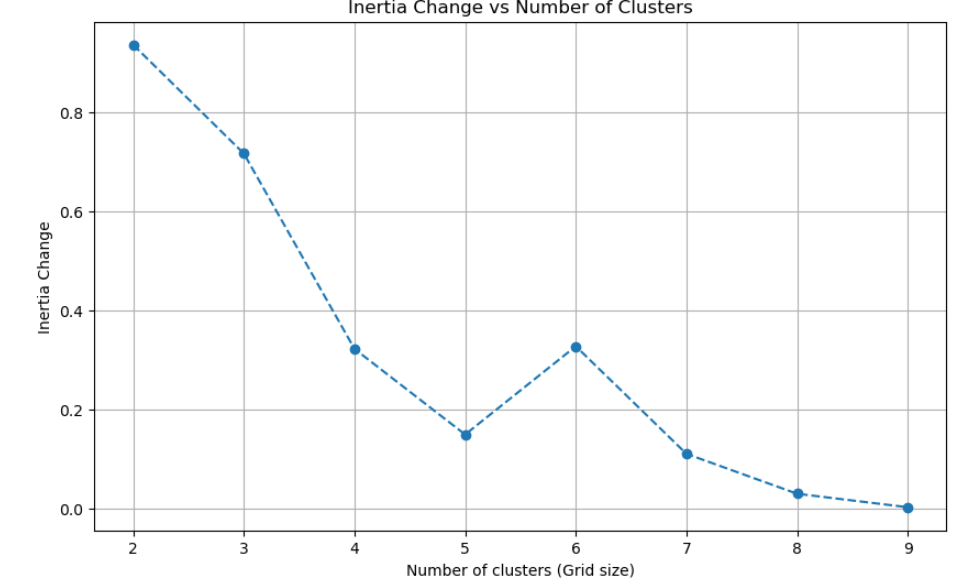
Phosphorus ir Potassium

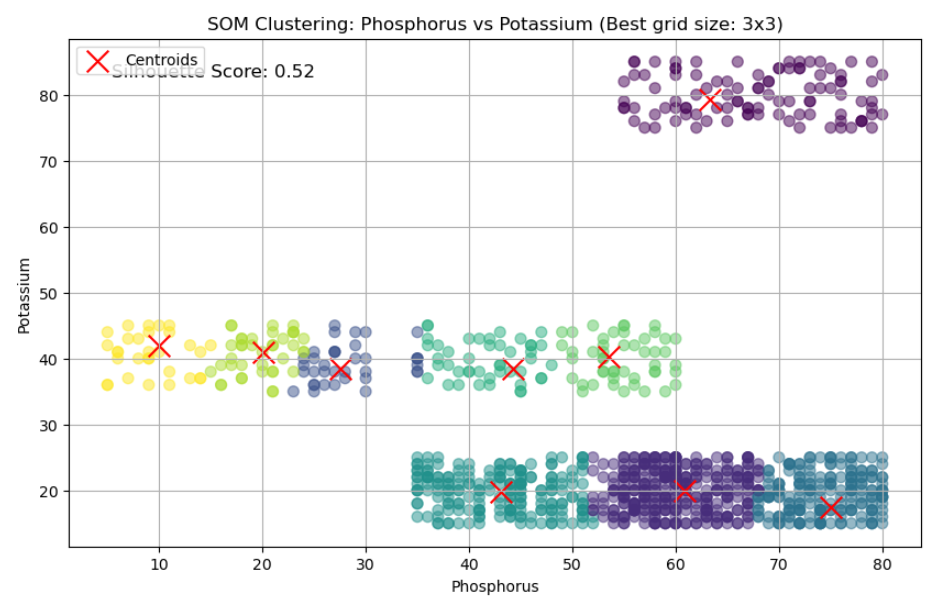
Šiam testavimui galime pažiūrėti kaip algoritmas apskaičiuoja inercijos, silueto įverčius, taip pat kaip apskaičiuoja centroides. Kaip matome, stabdymo funkcija stojo ties 9 klasteriais. Šiem dviem atributam geriausias apskaičiuotas silueto įvertis - 0.52. Ir surado 9 centroides.

Inertia for 2 clusters: 118128.7538256702  
Silhouette Score for 2 clusters: 0.46538147899244514  
Inertia for 3 clusters: 33390.310273336894  
Silhouette Score for 3 clusters: 0.5174161152537203  
Inertia for 4 clusters: 22603.052165750458  
Silhouette Score for 4 clusters: 0.43921888274289766  
Inertia for 5 clusters: 19207.778359486907  
Silhouette Score for 5 clusters: 0.4163556685731333  
Inertia for 6 clusters: 12900.233000120748  
Silhouette Score for 6 clusters: 0.40609586686060406  
Inertia for 7 clusters: 11464.475761103837  
Silhouette Score for 7 clusters: 0.4101963416792885  
Inertia for 8 clusters: 11112.572452612923  
Silhouette Score for 8 clusters: 0.4126841188516323  
Inertia for 9 clusters: 11073.732217848792  
Silhouette Score for 9 clusters: 0.411749714318795  
Stopping at grid size 9 due to small inertia change: 0.0035





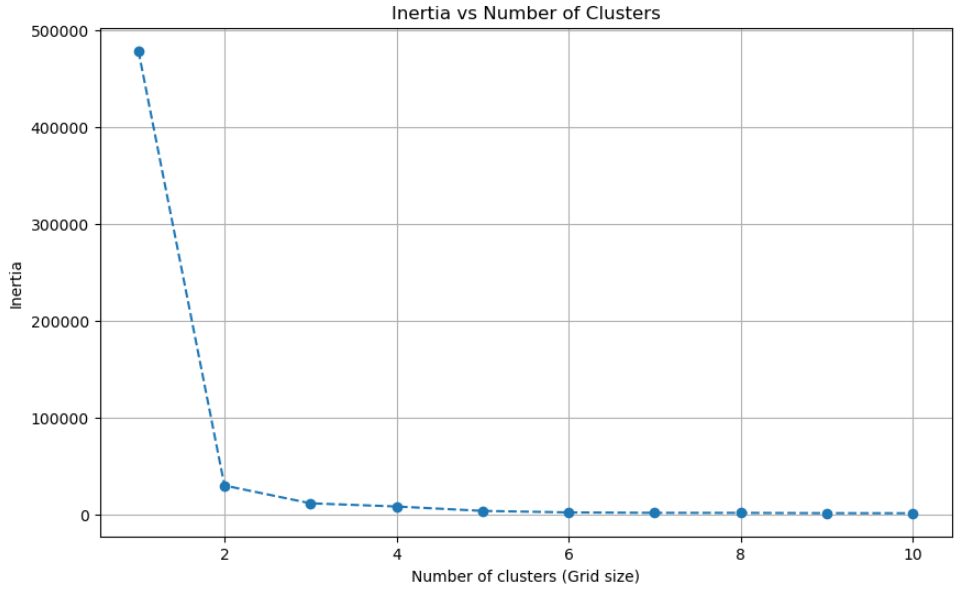


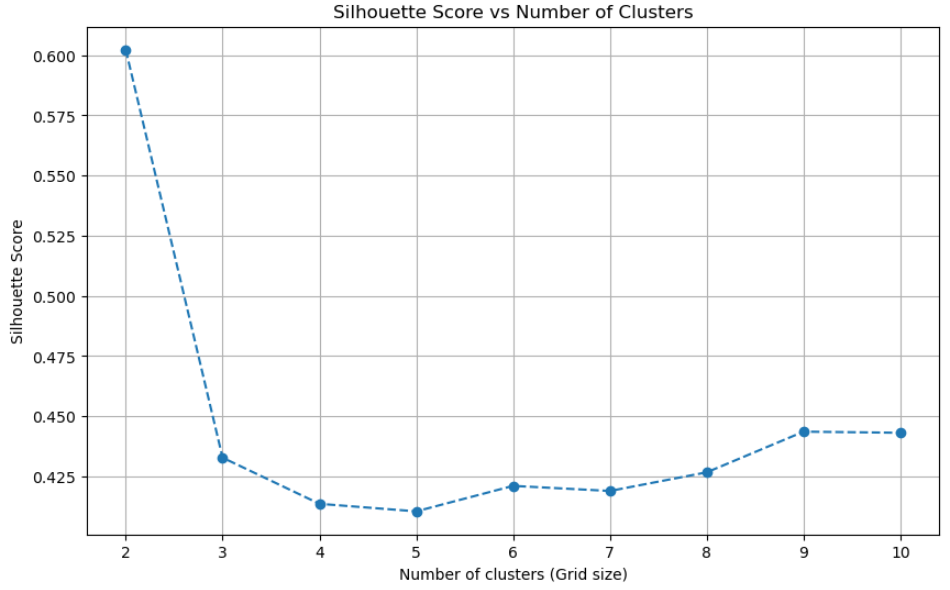


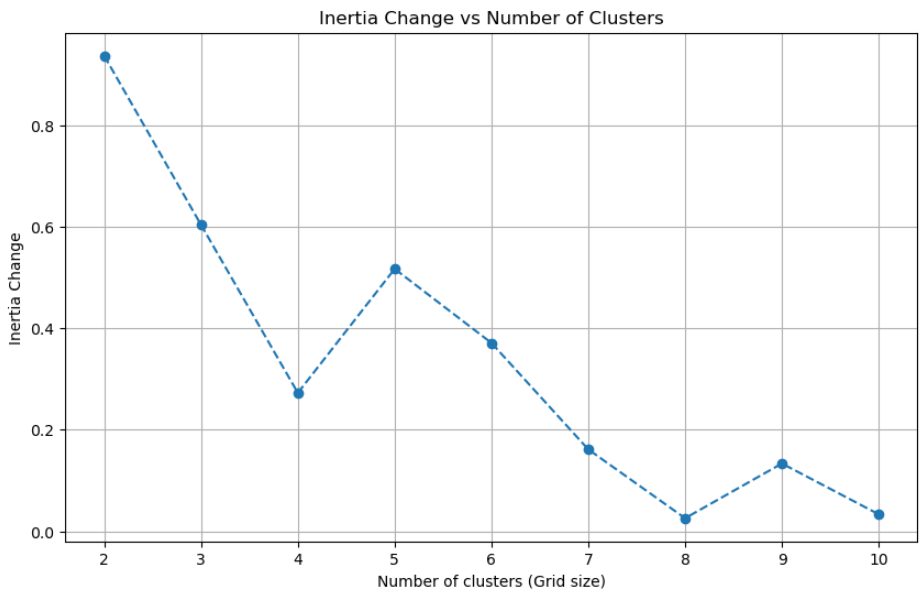
Potassium ir Temperature

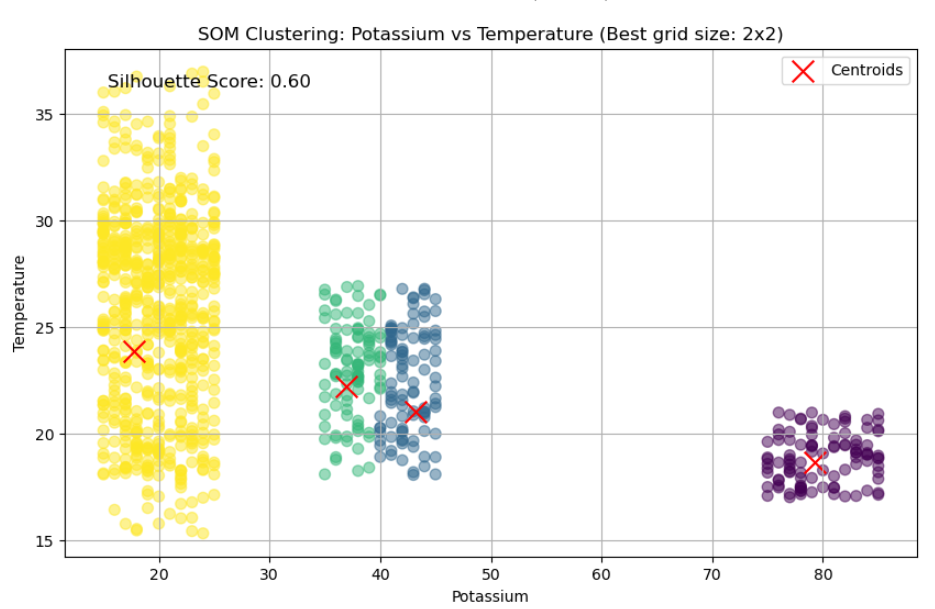
Šiam testavimui galime pažiūrėti kaip algoritmas apskaičiuoja inercijos, silueto įverčius, taip pat kaip apskaičiuoja centroides. Kaip matome, stabdymo funkcija stojo ties 2 klasteriais. Šiem dviem atributam geriausias apskaičiuotas silueto įvertis - 0.60. Ir surado 4 centroides.

Inertia for 2 clusters: 30389.758588538403  
Silhouette Score for 2 clusters: 0.6022765373742679  
Inertia for 3 clusters: 12045.395678653482  
Silhouette Score for 3 clusters: 0.4327912925200386  
Inertia for 4 clusters: 8760.883088849336  
Silhouette Score for 4 clusters: 0.4136234184561722  
Inertia for 5 clusters: 4232.569980837759  
Silhouette Score for 5 clusters: 0.41048384716578673  
Inertia for 6 clusters: 2662.265456851551  
Silhouette Score for 6 clusters: 0.4210763475069233  
Inertia for 7 clusters: 2232.6901189088258  
Silhouette Score for 7 clusters: 0.41896015951164767  
Inertia for 8 clusters: 2175.104079845669  
Silhouette Score for 8 clusters: 0.42670710134509726  
Inertia for 9 clusters: 1884.5750028592433  
Silhouette Score for 9 clusters: 0.44360607314347383  
Inertia for 10 clusters: 1821.406199024877  
Silhouette Score for 10 clusters: 0.4431309881853912





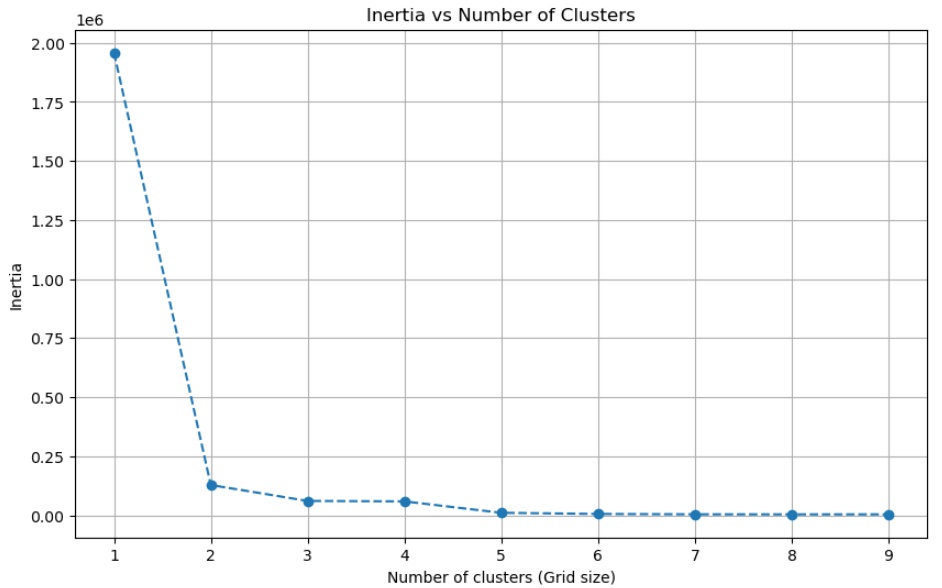


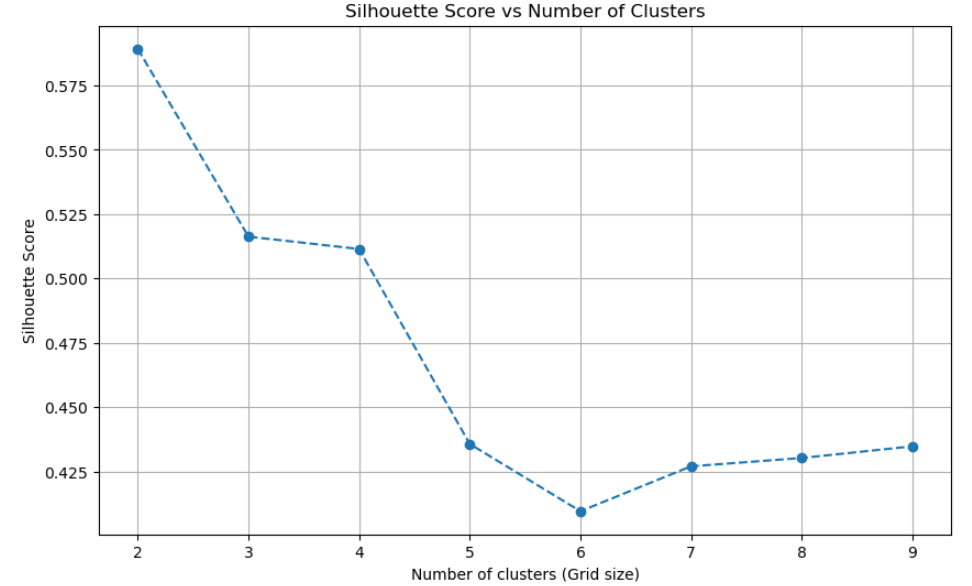


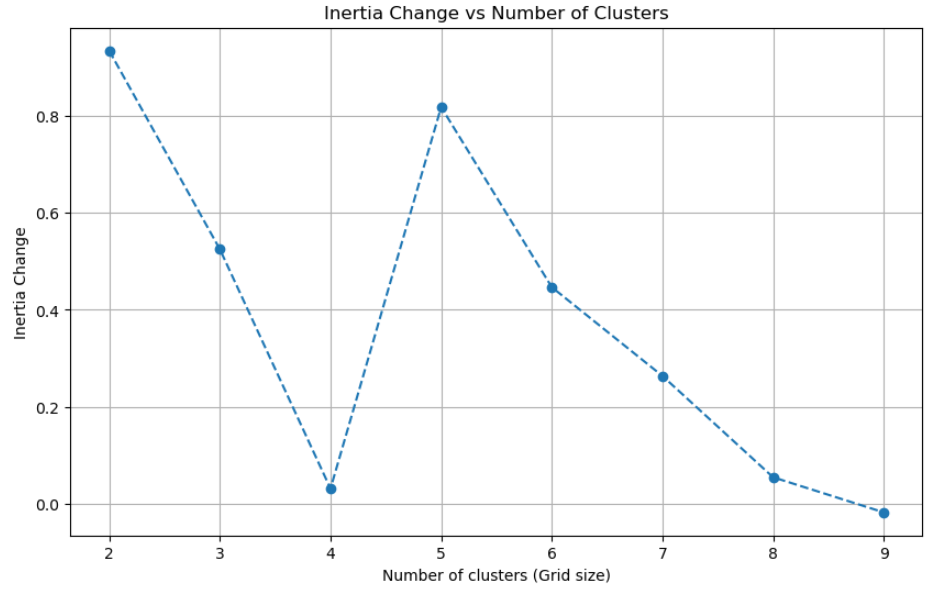
Potassium ir Humidity

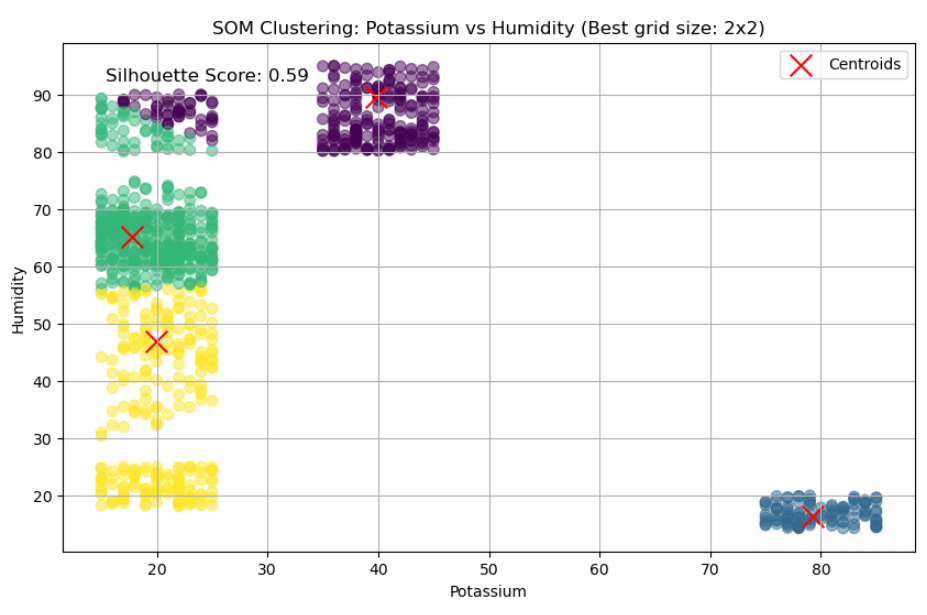
Šiam testavimui galime pažiūrėti kaip algoritmas apskaičiuoja inercijos, silueto įverčius, taip pat kaip apskaičiuoja centroides. Kaip matome, stabdymo funkcija stojo ties 9 klasteriais. Šiem dviem atributam geriausias apskaičiuotas silueto įvertis - 0.59. Ir surado 4 centroides.

Inertia for 2 clusters: 128349.2181804093  
Silhouette Score for 2 clusters: 0.5891948954095843  
Inertia for 3 clusters: 60971.4769425288  
Silhouette Score for 3 clusters: 0.5162634540554102  
Inertia for 4 clusters: 59067.514690616066  
Silhouette Score for 4 clusters: 0.5114593505836975  
Inertia for 5 clusters: 10734.319870270216  
Silhouette Score for 5 clusters: 0.43574122604142634  
Inertia for 6 clusters: 5939.05827433034  
Silhouette Score for 6 clusters: 0.409616107685907  
Inertia for 7 clusters: 4373.38911472809  
Silhouette Score for 7 clusters: 0.42706637019777366  
Inertia for 8 clusters: 4134.16430735656  
Silhouette Score for 8 clusters: 0.43029380515374654  
Inertia for 9 clusters: 4207.508436336911  
Silhouette Score for 9 clusters: 0.43483122944653  
Stopping at grid size 9 due to small inertia change: -0.0177









DBSCAN Algoritmo skaičiavimai ir algoritmas

Šiam algoritmui įgyvendinti buvo naudojama Python programavimo kalba. DBSCAN algoritmui aprašyti naudojamas kodas skirtas duomenims klasterizuoti ir vizualizuoti. Algoritmas veikia nustatant pagrindinius taškus, kurie turi tam tikrą skaičių kaimyninių taškų, esančių tam tikru atstumu. Vykstant klasterizavimo procesui, algoritmas nustato pagrindinius taškus ir išplečia klasterius, įtraukdamas visus taškus, esančius pagrindinių taškų kaimynystėje. Taškai, kurie nepasiekiami nė iš vieno branduolio taško, klasifikuojami kaip triukšmas. Ir nustatant pagrindinius taškus, ir plečiant klasterius algoritmas naudoja Euklido atstumą. Parametrai eps (didžiausias atstumas tarp dviejų taškų, kad jie būtų laikomi kaimynais) ir min\_samples (mažiausias taškų skaičius tankiam regionui sudaryti) yra labai svarbūs nustatant klasterius. Algoritmas įvertina klasterizavimo efektyvumą naudodamas silueto koeficientą ir vizualizuoja klasterius bei centroidus, kad būtų galima suprasti duomenų struktūrą.  
  
  
 **Funkcijos:**

|  |
| --- |
| import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  from sklearn.metrics import silhouette\_score    def calculate\_distance(point1, point2):  return np.sqrt(np.sum((point1 - point2) \*\* 2))    def region\_query(X, point\_idx, eps):  neighbors = []  for i in range(len(X)):  if calculate\_distance(X[point\_idx], X[i]) < eps:  neighbors.append(i)  return neighbors    def expand\_cluster(X, labels, point\_idx, cluster\_id, eps, min\_samples):  seeds = region\_query(X, point\_idx, eps)  if len(seeds) < min\_samples:  labels[point\_idx] = -1 # Mark as noise  return False  else:  labels[point\_idx] = cluster\_id  for seed\_idx in seeds:  labels[seed\_idx] = cluster\_id    while seeds:  current\_point = seeds[0]  results = region\_query(X, current\_point, eps)  if len(results) >= min\_samples:  for i in range(len(results)):  result\_point = results[i]  if labels[result\_point] == 0 or labels[result\_point] == -1:  if labels[result\_point] == 0:  seeds.append(result\_point)  labels[result\_point] = cluster\_id  seeds = seeds[1:]  return True    def dbscan(X, eps, min\_samples):  labels = np.zeros(len(X))  cluster\_id = 0  for point\_idx in range(len(X)):  if labels[point\_idx] == 0:  if expand\_cluster(X, labels, point\_idx, cluster\_id + 1, eps, min\_samples):  cluster\_id += 1  return labels |

**Duomenu nuskaitymas:**

|  |
| --- |
| data = pd.read\_csv("Crop\_Recommendation.csv")  data = data.head(1000)    attributes = ['Nitrogen', 'Phosphorus', 'Potassium', 'Temperature', 'Humidity', 'pH\_Value', 'Rainfall'] |

**Algoritmas:**

|  |
| --- |
| for i in range(len(attributes)):  for j in range(i + 1, len(attributes)):  X = data[[attributes[i], attributes[j]]].values    # Perform DBSCAN clustering for different eps values  min\_samples = 5 # The number of samples (or total weight) in a neighborhood for a point to be considered as a core point  eps\_values = np.linspace(1, 10, 10)  inertia\_values = []  num\_clusters = []    for eps in eps\_values:  labels = dbscan(X, eps, min\_samples)  unique\_labels = np.unique(labels)  num\_clusters.append(len(unique\_labels) - (1 if -1 in unique\_labels else 0)) # Exclude noise  if num\_clusters[-1] == 0:  inertia\_values.append(np.nan)  print(f"No clusters found for eps={eps}. Skipping inertia calculation.")  continue  inertia = np.sum([np.min([calculate\_distance(X[i], X[j]) for j in range(len(X)) if labels[j] == labels[i]]) for i in range(len(X)) if labels[i] != -1])  inertia\_values.append(inertia)  print(f"Inertia for eps={eps}: {inertia}")    # Remove NaN values from inertia\_values and corresponding eps\_values and num\_clusters  valid\_indices = ~np.isnan(inertia\_values)  eps\_values = np.array(eps\_values)[valid\_indices]  inertia\_values = np.array(inertia\_values)[valid\_indices]  num\_clusters = np.array(num\_clusters)[valid\_indices]    # Sort num\_clusters and corresponding inertia\_values  sorted\_indices = np.argsort(num\_clusters)  num\_clusters = num\_clusters[sorted\_indices]  inertia\_values = inertia\_values[sorted\_indices]    # Calculate rate of change of inertia  if len(inertia\_values) > 1:  max\_inertia = max(inertia\_values)  inertia\_change = [(inertia\_values[i] - inertia\_values[i + 1]) / max\_inertia for i in range(len(inertia\_values) - 1)]  print(f"Inertia change: {inertia\_change}")    # Find the optimal index where the inertia change is less than 5%  optimal\_index = np.argmax(np.array(inertia\_change) < 0.05) + 1  optimal\_eps = eps\_values[optimal\_index]  labels = dbscan(X, optimal\_eps, min\_samples)    # Calculate silhouette score  if len(np.unique(labels)) > 1:  silhouette\_avg = silhouette\_score(X, labels)  print(f'Silhouette Score for eps={optimal\_eps}: {silhouette\_avg}')    # Plot the elbow curve with the number of clusters  plt.figure(figsize=(10, 6))  plt.plot(num\_clusters, inertia\_values, marker='o', linestyle='--')  plt.xlabel('Number of Clusters')  plt.ylabel('Inertia')  plt.title('Elbow Curve')  plt.grid(True)  plt.show()    # Plot the clusters  plt.figure(figsize=(10, 6))  plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, alpha=0.5)  plt.title('DBSCAN Clustering: {} vs {}'.format(attributes[i], attributes[j]))  plt.xlabel(attributes[i])  plt.ylabel(attributes[j])  plt.grid(True)  plt.show()    # Calculate the number of correctly assigned points  unique\_labels = np.unique(labels)  correct\_assignments = 0  for label in unique\_labels:  points\_in\_cluster = X[labels == label]  if len(points\_in\_cluster) > 0:  distances\_within\_cluster = np.sum([calculate\_distance(point, np.mean(points\_in\_cluster, axis=0)) for point in points\_in\_cluster])  correct\_assignments += len(points\_in\_cluster) - distances\_within\_cluster / len(points\_in\_cluster)    accuracy = (correct\_assignments / len(X)) \* 100 # Calculate accuracy  print(f"Accuracy for DBSCAN Clustering: {accuracy:.2f}%")  else:  print("No valid eps values found for clustering. Skipping further calculations.") |

**Testavimas:**

Algoritmas atlieka k-vidurkių klasterizavimą, tarp kiekvienos tolydinio tipo atributo poros. Prie kiekvienos poros atvaizduojama inercijos pakytis priklausantis nuo klasterių skaičiaus ir inercijos pokytis procentais, naudojamas parikti klasterių skaičių. DBSCAN algoritmo suklasterizuoto grafiko vaizdas ir tikslumas. Šis skaičiavimas buvo atliktas naudojant du skirtingus, minimalius pokyčio reikalavimus, norint surasti tinkamiausius rezultatus. Apačioje yra pateikiami 5 geriausiai susidarę klasteriai tarp atributų porų.

Nitrogen ir Potassium

|  |
| --- |
| Average accuracy for DBSCAN clustering: 59.99%  Inertia change: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2497784728294078, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.22852262116274885, 0.0, 0.0, 0.0, 0.30257860453813135] |

Nitroger ir Humidity

|  |
| --- |
| Average accuracy for DBSCAN clustering: 46.34%  Inertia change: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.14651547266470846, 0.0, 0.32228310294834933, 0.30251686262908006] |

Phosphorus ir potassium

|  |
| --- |
| Average accuracy for DBSCAN clustering: 62.87% Inertia change: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2583040292124898, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.19357022220903172, 0.0, 0.0, 0.07264939070086696, 0.1016009454858472 |

Potassium ir temperature

|  |
| --- |
| Inertia change: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.016176257889666538, 0.18858966133039645, 0.720922395917342] Average accuracy for DBSCAN clustering: 51.89% |

Potassium ir Humidity

|  |
| --- |
| Inertia change: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3569124068413089, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2028604300378419, 0.0, 0.0, 0.20154148719484577, 0.020711613126760033] Average accuracy for DBSCAN clustering: 59.10% |

1. Algoritmų Palyginimas

Iš apačioje pateiktos lentelės galime matyti, jog K-vidurkių algoritmas yra greičiausias, visų porų skaičiavimus įvykdo per 6 min, palyginus su kitais algoritmais šis algoritmas yra greičiausias. Tačiau atsižvelgiant į tikslumą, geriausią tikslumą pasiekia SOM algoritmas. Šio algoritmo vidutinis tikslumas siekia 76%. Tinkamiausių atributų poros šiek tiek skiriasi, tačiau vizualiai šios poros yra labai panašios. Tai parodo, jog algoritmai geriausiai suklasterizuoja tada, kai akivaizdžiai galima matyti klasterius. Visi algoritmai yra tinkami, ir kiekvienas turi savų privalumų ir trūkumų.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **K-vidurkių** | **SOM** | **DBSCAN** |
| Vidutinis tikslumas | 33% | 76.5%(0.53) | 44.2% |
| Daugiausiai naudojamas klasterių skaičius | 3 | 2 | 2 |
| Tinkamiausia atributų pora | Potassium ir temperature | Potassium vs pH\_Value | Potassium vs Rainfall |
| Skaičiavimo greitis | 6 min. | 59 min. | ~8 min. |
| Ar tinka šis metodas? | Taip | Taip | Taip |
| Blogiausias rezultatas | Temperature ir pH\_Values | Phosphorus vs Temperature | Nitrogen vs Humidity |

1. Išvados

Norint suklasterizuoti duomenų rinkinį greičiausiai, bet kuriuo atveju reikia rinktis K-Vidurkių algoritmą dėka jo didelės spartos, tačiau, jeigu norima išgauti tikslesnį klasterizavimo rezultatą, reikia panaudoti SOM algoritmą. Kadangi tikslumo žymėjimas ne visiems algoritmams yra vienodas, todėl SOM algoritmo tikslumo įverčiui skaičiuojamas “*Silhouette Score”,* todėl šį reikia paversti į procentalią reikšmę tarp –1 ir 1.