Kauno technologijos universitetas

Informatikos fakultetas

Laboratorinis darbas Nr. 1. Neraiškioji logika

Laboratorinio darbo ataskaita

Greta Intaitė

Studentas / Studentė

Kaunas, 2024

Turinys

Įvadas

Modulio „Mašininio mokymo algoritmai 2“ (P176B106) pirmo laboratorinio darbo metu naudojant neraiškiąją logiką sudaryta ekspertinė sistema, kurios tikslas įvertinti kandidato tinkamumą pasirinktai rolei. Sistema realizuojama nuo nulio: taisykles ir neraiškiųjų aibių ribas surenkant pokalbio metu ir realizuojant sistemą nenaudojant specializuotų bibliotekų. Toliau ataskaitoje papasakojama apie sprendžiamą problemą, duomenų surinkimą, suformuotas aibes ir taisykles, realizuotus algoritmus ir pateikiami testinių scenarijų rezultatai.

1. Problema. Ekspertų apklausa

Laboratorinio darbo metu pasirinkta kurti ekspertinę sistemą, kuri pateiktų rekomendacijas sprendimui apie kandidato tinkamumą hipotetinei rolei. Tam, kad būtų galima sudaryti tokią sistemą, reikalingos ekspertų žinios:

- Kriterijai, pagal kuriuos vertinami kandidatai
- Kriterijų skaitinės ribos (metai, procentai ir t.t.)
- Taisyklės, kuriomis remdamiesi ekspertai įvertina kandidato tinkamumą

Kriterijai, naudojami kandidatų vertinimui parenkami pagal internete rastą informaciją apie tai, kuo vadovaujasi personalo atrankos specialistai. Likę surenkami apklausos ir pokalbių su ekspertais metu.

1.1. Kriterijų neraiškiųjų aibių ribos

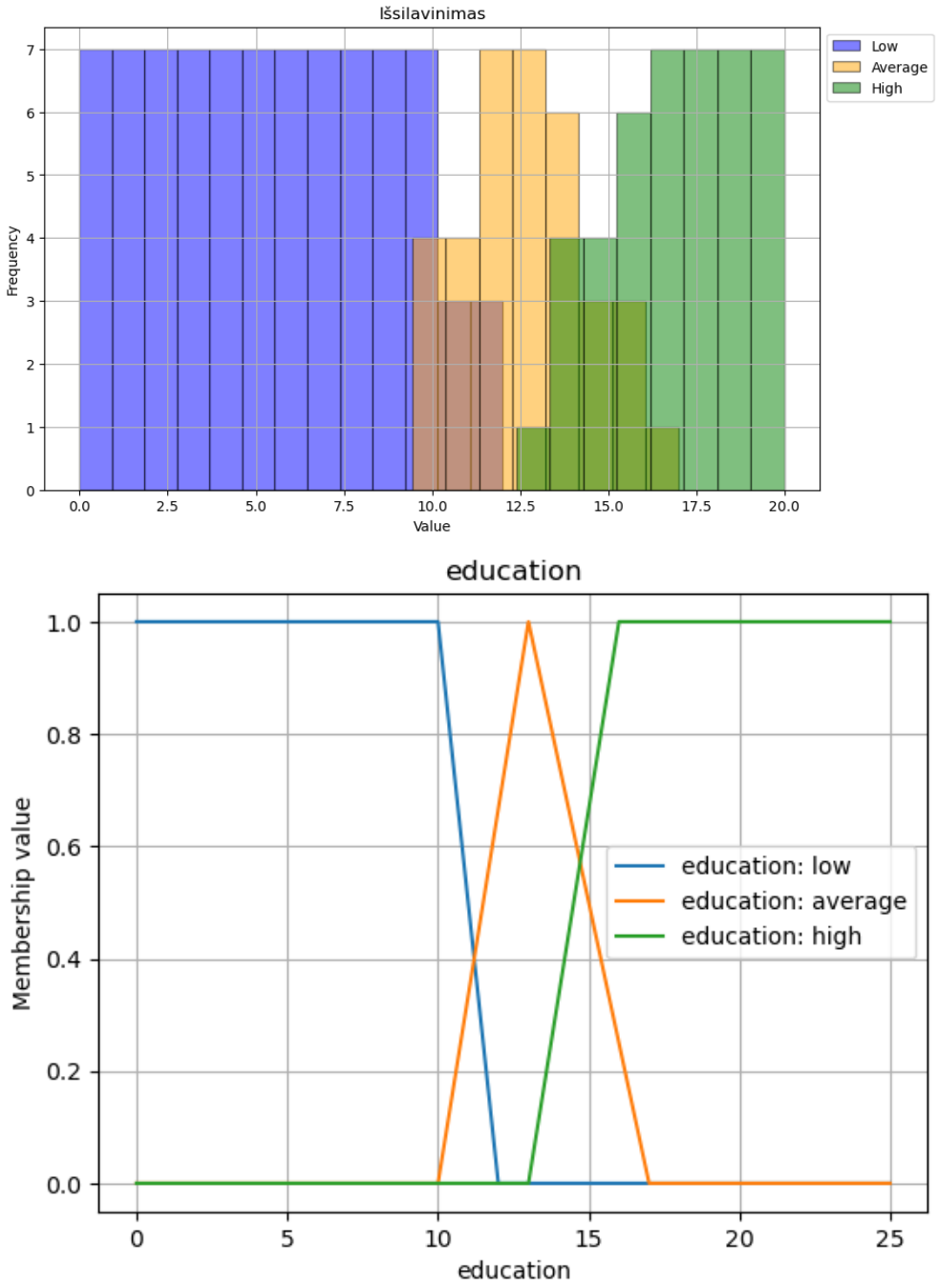
7 ekspertams pateikta Google Forms apklausa, kurią sudarė 16 klausimų, kuriais respondentai turėjo įvertinti kriterijus pagal 1 lentelėje pateiktą informaciją.

1 lentelė. Kriterijai. Tinkamumo lygis – išvestis, kiti – įvestys.

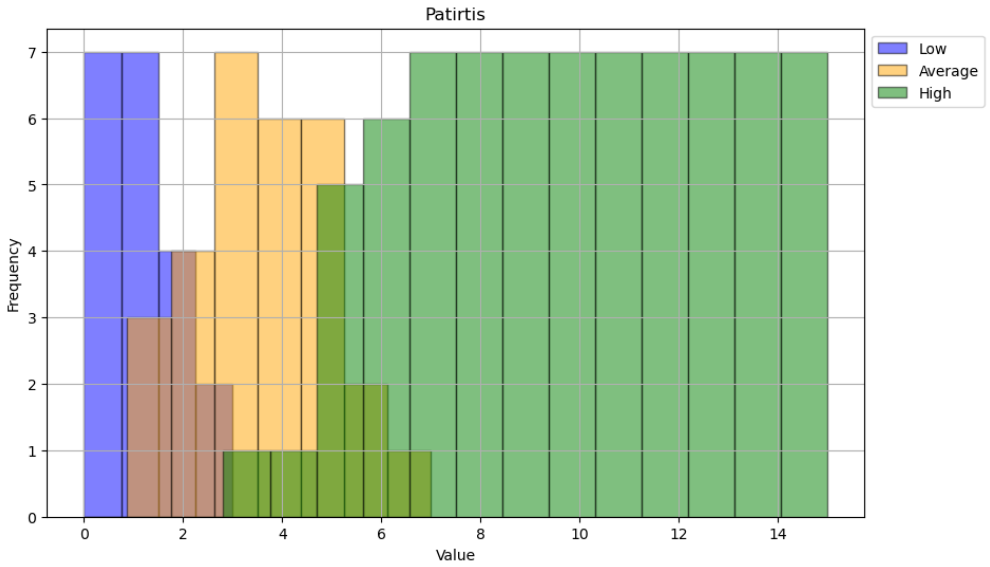
Kriterijus **Liet. / angl.**	**Low**	**Average**	**Decent**	**High/Good**
Išsilavinimas / education (metais)	Žemas išsilavinimo lygis (metais)	Vidutinis išsilavinimo lygis (metais)	-	Aukštas išsilavinimo lygis (metais)
Patirtis / experience (metais)	Nedidelė (metais)	Vidutinė (metais)	-	Didelė (metais)
Minkštųjų įgūdžių testo ir pokalbio rezultatas / soft skills (%)	Prastas rezultatas (%)	Vidutinis rezultatas (%)	Tinkamas, tačiau ne puikus (%)	Labai geras (%)
Savanorystės trukmė / volunteering (mėnesiais)	Trumpa	Vidutinė	-	Ilga
Tinkamumo lygis / suitability (%)	Žemas	Vidutinis	-	Geras

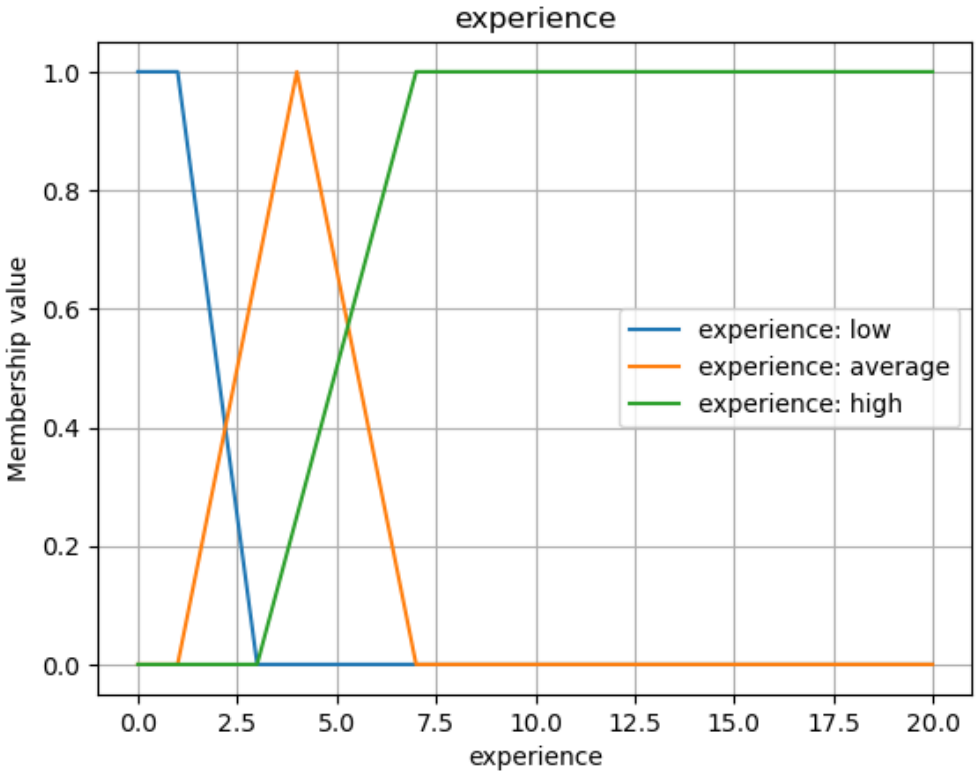
Atlikus apklausą ir gavus ekspertų įverčius jų reikšmių dažnių pasiskirstymas ir pagal tai sudarytų priklausomumo funkcijų grafikai pateikiami poromis toliau šiame poskyryje. Priklausomumo funkcijos parenkamos pagal tikslumo poreikį problemos sprendimui ir pasiskirstymo dažnių sudarytas formas.

Kadangi kandidatų įvertinimo problema nereikalauja didelio tikslumo pasirinka naudoti trapecijos ir trikampio priklausomybės funkcijas jas paskirstant „iš akies“ įvertinus neraiškiųjų aibių elementų dažnius ekspertų atsakymuose iliustruojančią diagramą. Trapecija naudojama, kai didelė dalis ekspertų į savo kriterijų aibių intervalus įtraukia daugiau nei vieną ar dvi reikšmės ir dažnių diagrama neturi vieno konkretaus maksimumo taško. Esant konkrečiam dažnių maksimumo taškui ir panašiems “šlaitams” parenkama trikampė priklausomumo funkcija.

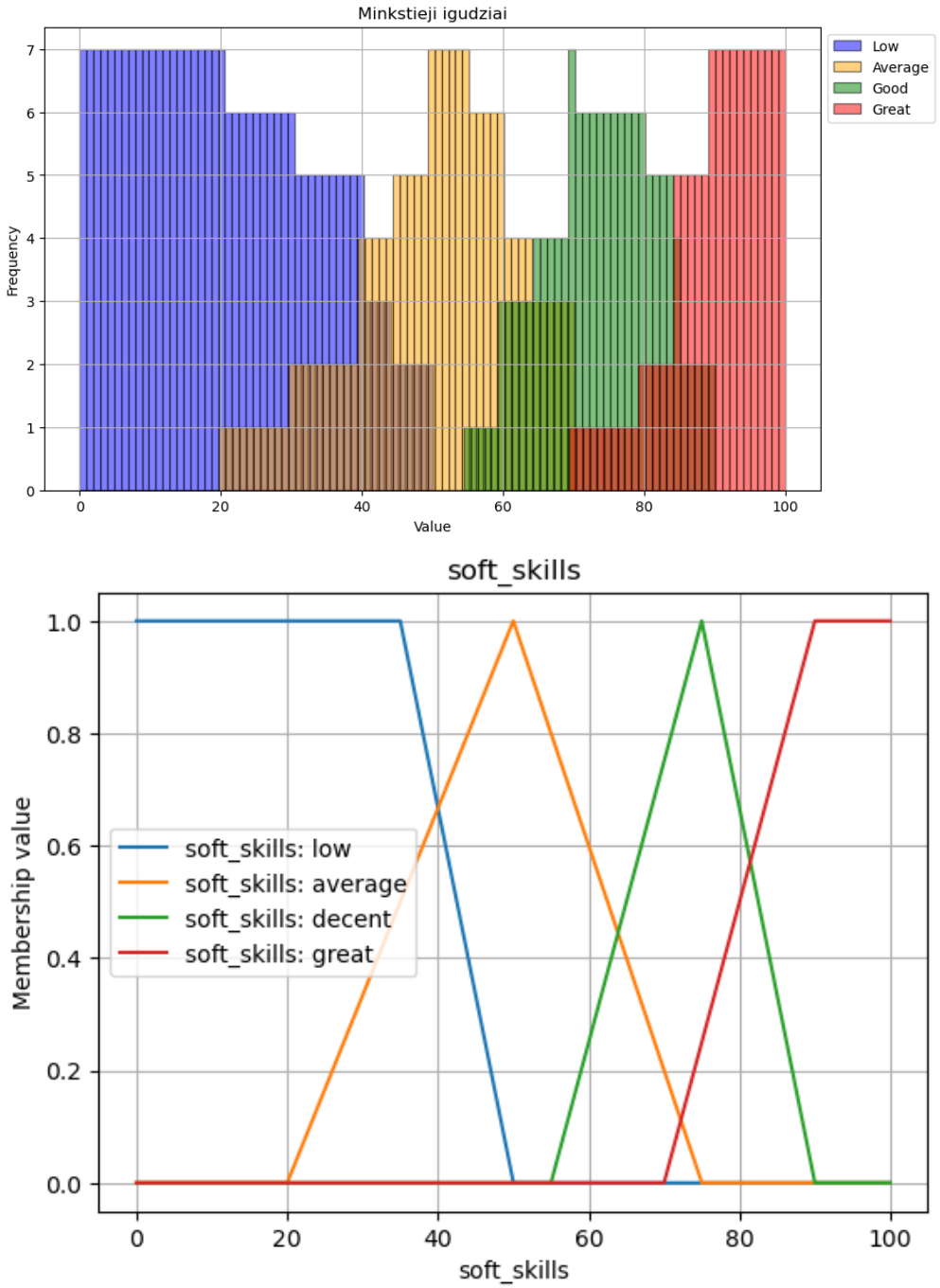


1 pav. Ekspertų apklausos rezultatai apie išsilavinimo lygio neraiškiųjų aibių ribas ir neraiškiosios aibės

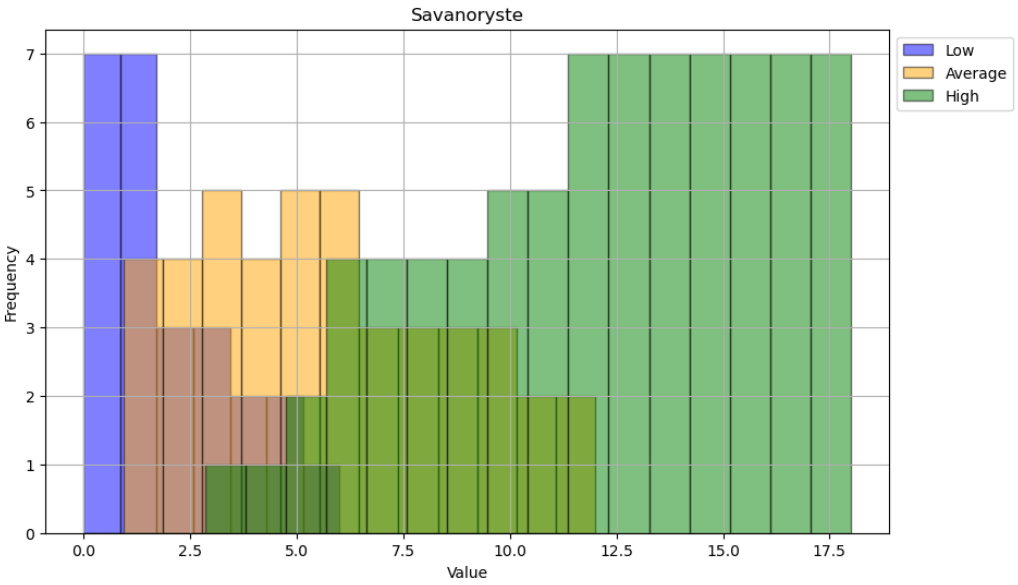


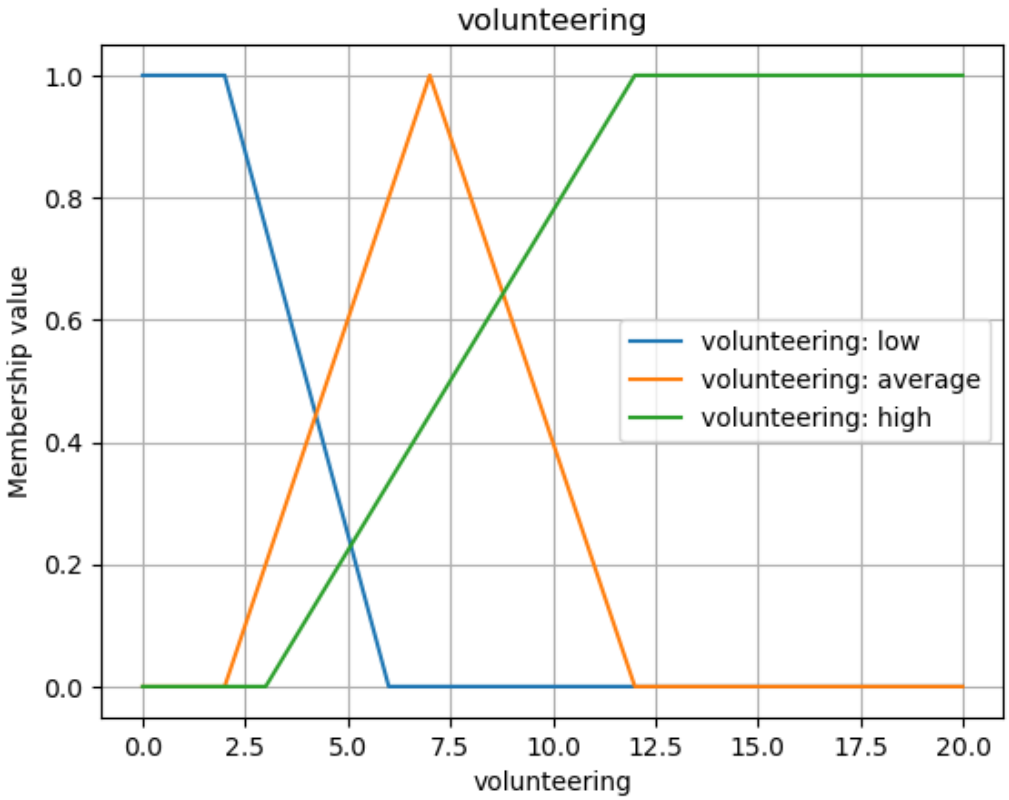


2 pav. Ekspertų apklausos rezultatai apie patirties lygio neraiškiųjų aibių ribas ir neraiškiosios aibės

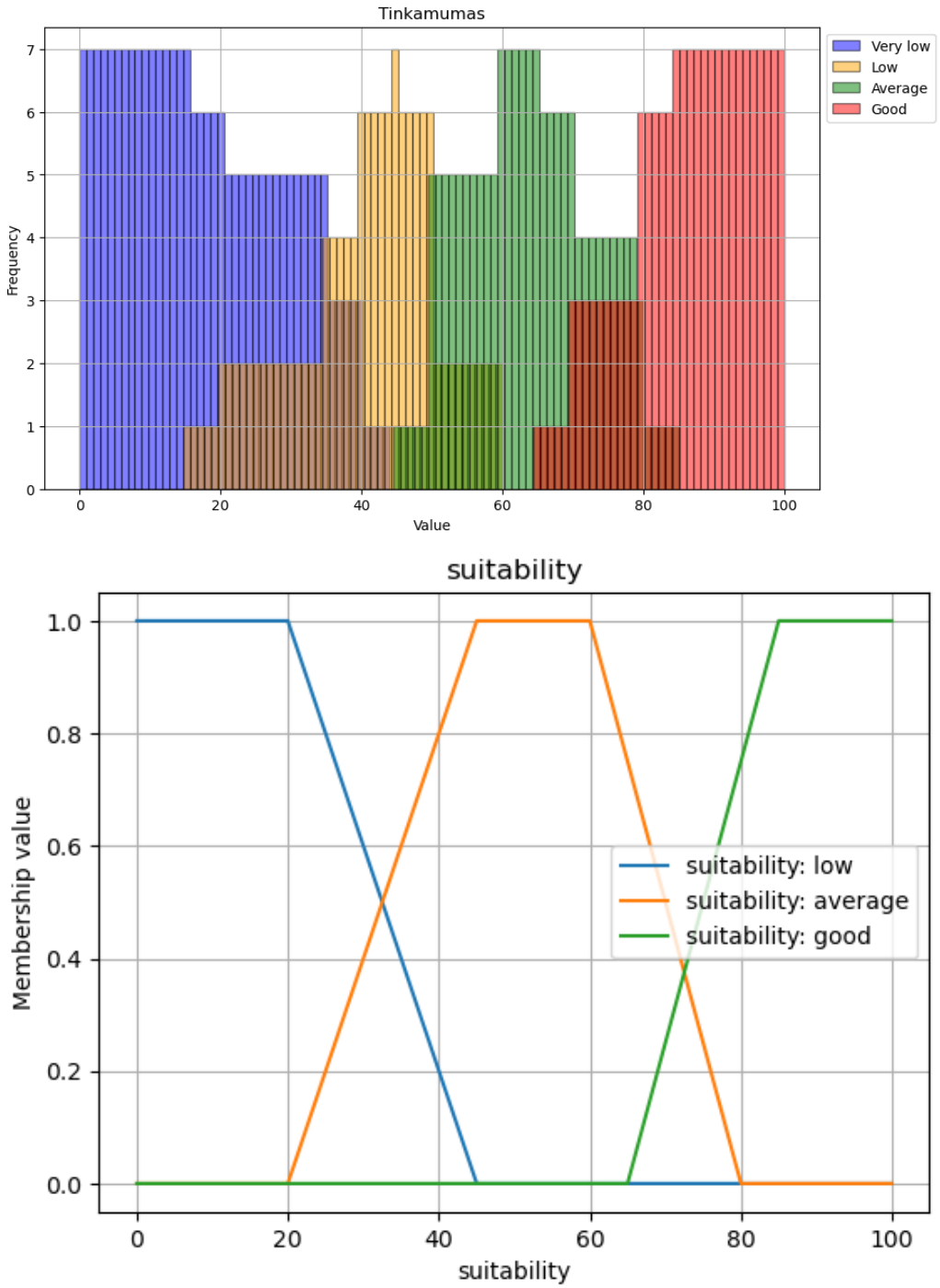


3 pav. Ekspertų apklausos rezultatai apie minkštųjų įgūdžių įvertinimo neraiškiųjų aibių ribas ir sudarytos neraiškiosios aibės





4 pav. Ekspertų apklausos rezultatai apie savanorystės trukmės neraiškiųjų aibių ribas ir neraiškiosios aibės



5 pav. Ekspertų apklausos rezultatai apie tinkamumo lygio neraiškiųjų aibių ribas ir neraiškiosios aibės

Užduoties aprašyme nurodyta, kad išvesties reikšmė turi turėti 3 aibes, tad grafike pateikiamos aibės yra modifikuojamos Low ir Average apjungiamos į vieną Average aibę.

1.2. Taisyklių rinkinys

Apibendrintai ekspertų pasiūlytą ir kuriant papildytą taisyklių rinkinį sudaro 8 taisyklės, suformuluotos naudojant Mamdani algoritmą:

IF education high AND experience high AND soft_skills NOT low THEN suitability is **good**.
IF education high AND experience average AND soft_skills great THEN suitability is **good**.

IF education average AND (experience high OR experience average) AND soft_skills great THEN suitability is **good**.

IF education high AND experience average AND soft_skills decent THEN suitability is **average**.
IF (education high OR education average) AND experience low AND soft_skills great THEN suitability is **average**.
IF education low AND (experience high OR experience average) AND soft_skills great THEN suitability is **average**.
IF education low AND experience low THEN suitability **low**.
IF (education high OR education average) AND (experience high OR experience average) AND soft_skills low THEN suitability **low**.

Remiantis aprašytu taisyklių rinkiniu galima spręsti, kad didelę įtaką kandidatų tinkamumui daro jų minkštieji įgūdžiai. O štai pasirinktos savanorystės trukmės ekspertai, kalbėdami apie taisykles, neįvardino. Buvo paminėta, kad savanorystė yra privalumas ir siejama su minkštaisiais įgūdžiais, tačiau neturi didelės įtakos renkantis tinkamiausią kandidatą.

2. Sistemos realizacija

Naudojant pirmame skyriuje aprašytas neraiškiąsias aibes ir sudarytą taisyklių rinkinį, kuriama ekspertinė sistema, paremta neraiškiąja logika.

Kad būtų paprasčiau paaiškinti sistemos veikimą, analizuojamas pavyzdys, kai naudotojas įveda reikšmes:

2 lentelė. Pavyzdžio kriterijų reikšmės

Kriterijus	**Reikšmė**
Išsilavinimas	16
Patirtis	5
Minkštųjų įgūdžių testo rezultatas	70
Savanorystės trukmė	1

2.1. Fuzifikacija. Skaitinės reikšmės į neraiškiąsias aibes

Naudotojui įvedus išsilavinimo lygį, patirties lygį, minkštųjų įgūdžių testo rezultatus ir savanorystės trukmės reikšmes skaičiais priklausomai nuo aibės formos naudojant jos priklausomumo funkciją yra apskaičiuojamas kiekvienos įvestos reikšmės priklausymo lygis kiekvienai tos kategorijos neraiškiajai aibei naudojant žemiau pateiktas funkcijas.

```
def trapezoid(x, a, m1, m2, b):
    if x >= m1 and x <= m2:
        return 1
    elif x > a and x < m1:
        return ((x - a)/(m1 - a))
    elif x < b and x > m2:
        return ((b - x)/(b - m2))
    else:
        return 0

def triangular(x, a, m, b):
    if x == m:
        return 1
    elif x > a and x < m:
        return (x - a) / (m - a)
    elif m < x and x < b:
        return (b - x) / (b - m)
    else:
        return 0
```

Po šio žingsnio gaunamas dictionary (liet. žodyno) tipo objektas, kuriame saugoma kiekvieno kriterijaus reikšmės priklausymo lygis to kriterijaus aibėms. Žinant, kokioms neraiškiosioms aibėms priklauso kiekviena reikšmė galima pereiti prie taisyklių implikacijos.

Nagrinėjamo pavyzdžio atveju gauta:

```
{'education': {'low': 0, 'average': 0.25, 'high': 1},
 'experience': {'low': 0, 'average': 0.6666666666666666, 'high': 0.5},
 'soft_skills': {'low': 0, 'average': 0.2, 'decent': 0.75, 'great': 0},
 'volunteering': {'low': 1, 'average': 0, 'high': 0}}
```

2.2. Implikacija. Priklausymas išeities aibėms

Implikacijos etapo metu naudojant taisyklių rinkinį (žr. 1.2 skyrių) apskaičiuojamas tinkamumo lygis – priklausymo kiekvienai išeities reikšmės – tinkamumo aibei lygis. Naudojant Mamdani algoritmą remiamasi tokiomis taisyklėmis:

- AND jungtukas keičiamas į min funkciją
- OR jungtukas keičiamas į max funkciją
- NOT jungtukas keičiamas į 1 – priklausymo kategorijai / neraiškiajai aibei reikšmė
- Taisyklės, kurių rezultatas yra tokia pati aibė, sujungiamos OR jungtukais.

Pagal pirmą taisyklę ieškomas minimumas tarp reikšmių, nurodančių išsilavinimo reikšmės priklausymą „high“ neraiškiajai aibei, patirties reikšmės priklausymą patirties kriterijaus „high“ aibei ir minkštųjų įgūdžių reikšmės priklausymą aibei „low“ atimtai iš vieneto. Šios reikšmės apskaičiuojamos ankstesniame žingsnyje (žr. 2.1). Gauta minimali reikšmė yra kandidato priklausymo lygis/laipsnis tinkamumo kriterijaus aibei „good“. Po šio žingsnio gaunamas žodynas, kuriame saugoma kandidato priklausomumo reikšmė kriterijaus „tinkamumas“ aibėms: „low“, „average“, „good“.

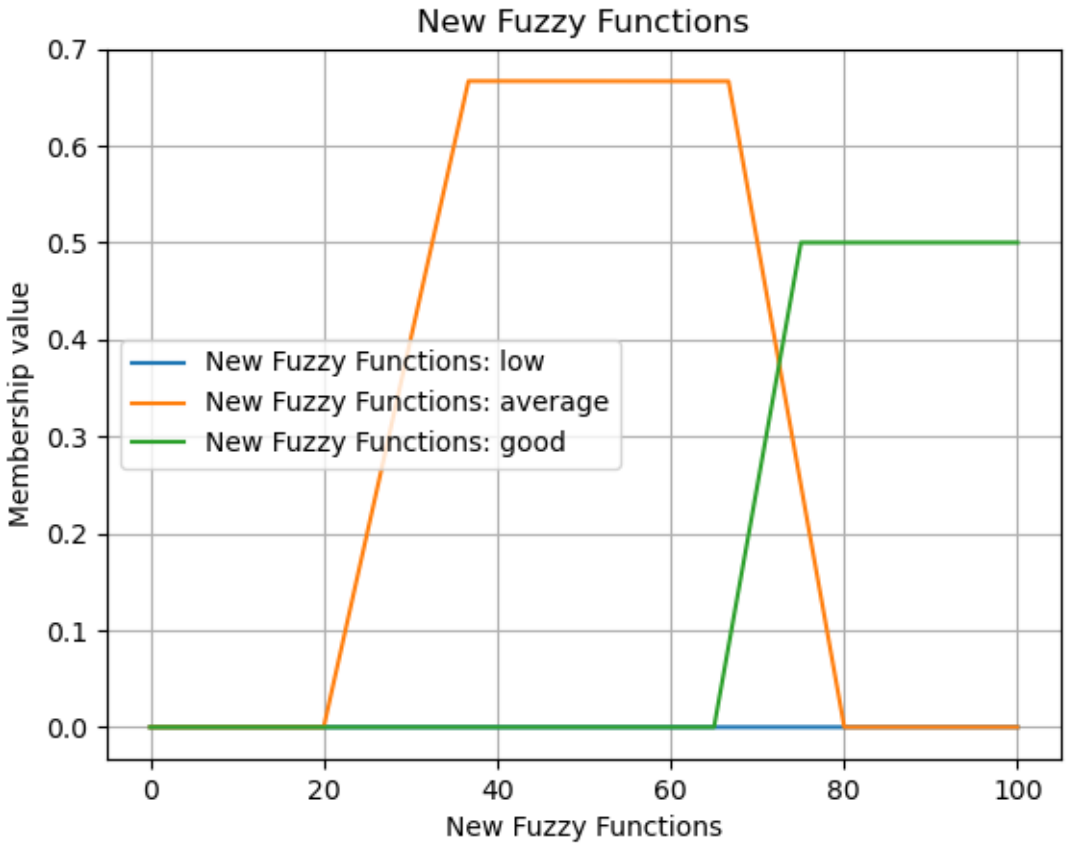
Nagrinėjamo pavyzdžio atveju šios reikšmės yra:

```
{'low': 0, 'average': 0.6666666666666666, 'good': 0.5}
```

2.3. Agregacija

Žinant, koks yra priklausomumo lygis kiekvienai išeities neraiškiajai aibei gaunamos naujos figūros, kurių maksimali y reikšmė ir yra priklausomumo tai aibei lygis, laipsnis.

Šiuo atveju gaunamos figūros atrodo taip:



6 pav. Agreguota figūra

Šias figūras sujungiame ir žvelgiame kaip į vieną figūrą. Tolimesni žingsniai skiriasi priklausomai nuo pasirinkto defuzifikacijos metodo.

2.4. Defuzifikacija

Defuzifikacijos metu naudojant neraiškiąsias aibes apskaičiuojama konkreti skaitinė reikšmė.

```
def defuzzification(self, values):
        rules = Rule()

        # 1. Getting degrees of membership for each category
        values = rules.calculate(values)

        # 2. Making new functions that define shapes of each category
        #    membership degree being the max value of y
        print(values)
        new_funcs = self.new_values(self.output_variable, values)
        self.som_mom(self.output_variable, values)
        new_funcs.modify_categories_md(values, self.output_variable)
        new_funcs.plot_categories(save=True)
        # 3. Calculating overlapping points
        _, intersect_points = self.find_intersections(new_funcs)

        # 4. Joining all xs and calculating max y values at each point
        points     =     self.calculate_joint_points(self.get_x_points(new_funcs,
intersect_points), new_funcs)
        #new_funcs.plot_categories(save=True, points=points)

        # 5. Filtering all points to keep the
```

```
        filtered_points = self.filter_points(new_funcs, points, values)
        print(f'Points after filtering: {filtered_points}')

        plots,     centroids     =     self.calculate_plots_centroids(new_funcs,
filtered_points)
        print(plots)
        print(centroids)

        return self.cog(plots, centroids)
```

2.4.1. Gravitacijos centro metodas (angl. center of gravity, COG)

Skaičiuojant COG metodu agregacijos metu gauta bendra figūra padalinama į mažesnes standartines: trikampius, stačiakampius ir trapecijas. Figūra atskiriama ties kampais, ir susikirtimo taškais. Pavyzdžio atveju gaunamos figūros: trikampis, stačiakampis, dvi trapecijos ir stačiakampis. Kokia figūra susidaro nustatoma palyginant figūros kampus žyminčius taškus tarpusavyje.

```
def calculate_plots_centroids(self, fvar, points):
        categories = fvar.get_categories()
        names_cat = list(categories.keys())
        plots = []
        centroids = []

        for i in range(len(points)-1):

            p1 = points[i] # current point
            p2 = points[i+1] # next point

            if p1[1] == 0 and p2[1] == 0:
                continue

            elif p1[1] == 0 or p2[1] == 0:
                plot, centroid = utils.triangle_plot_centre(p1, p2)
                print(f'Trikampis  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            elif p1[1] == p2[1]:
                plot, centroid = utils.rect_plot_centre(p1, p2)
                print(f'Stačiakampis  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            else:
                plot, centroid = utils.trap_plot_centre(p1, p2)
                print(f'Trapecija  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            if plot > 0:
                plots.append(plot)
                centroids.append(centroid)

        return plots, centroids
```

Naudojant šioms figūroms taikomas formules apskaičiuojamas kiekvienos iš jų plotas ir centroidas. Bendras visos figūros centroidas apskaičiuojamas pagal formulę:

$$centroidas = \frac{\sum_{i=1}^{n} x_i S_i}{\sum_{i=1}^{n} S_i}$$

S_i – figūros plotas, x_i – figūros centroidas.

```
    def cog(self, plots, centroids):
```

```
        product = 0
        sum = 0
        for i in range(len(plots)):
            sum += plots[i]
            product += (plots[i] * centroids[i])
        print(sum)
        print(product)
        return product/sum
```

Šio pavyzdžio atveju gaunamas centroidas ties $x = 61.78$. Vadinasi, kandidato tinkamumas yra 61.78%

2.4.2. Maksimumo vidurkis (MOM)

Naudojant MOM metodą nustatomos pirma ir paskutinė reikšmės, kurioje agreguotos figūros y reikšmė yra maksimali ir apskaičiuojamas šių reikšmių vidurkis.

```
def som_mom(self, fvar, values):
        max_key = max(values, key=lambda k: values[k])
        max_value = values[max_key]
        rez = self.find_max_values(fvar, values)
        print('SOM', min(rez[max_key]))
        print('MOM', np.mean(rez[max_key]))

def find_max_values(self, fvar, values, eps = 1e-2):
        fvar_cats = fvar.get_categories()
        keys = list(fvar_cats.keys())
        rez = dict()
        for key in keys:
            ind = [i for i in range(len(fvar_cats[key])) if abs(fvar_cats[key][i]-
values[key]) <= eps]
            xlist = np.array(fvar.x_bounds)
            xlist = xlist[ind]
            if np.abs(values[key] - 1) <= eps:
                rez[key] = list(xlist)
                pass
            else:
                b = np.min(xlist)
                c = np.max(xlist)
                rez[key] = [b,c]
        return rez
```

Pavyzdžio atveju, skaičiuojant MOM metodu gautas kandidato tinkamumas: 51.64%

3. Testavimas rezultatai

Naudojant realizuotą ekspertinę sistemą išbandomi 3 scenarijai siekiant ištestuoti sistemos veikimą.

3 lentelė. Testavimo duomenys ir gauti rezultatai

Išsilavinimo lygis (metai)	**Patirtis (metai)**	**Minkštųjų įgūdžių testo ir pokalbio rezultatas (%)**	**Savanorystės trukmė (mėnesiai)**	**Tinkamumas (%)**	
				COG	**MOM**
16	5	70	1	61.78	51.64
12	1	80	6	51.55	51.22
14	3	95	4	85.53	78.34

Testavimo rezultatai parodo, kad gravitacijos centro ir maksimumo vidurkių metodais gauti įverčiai gali skirti iki 10%. Taip pat pastebimas, kad didžiausią įtaką kandidato tinkamumui turi jo minkštųjų įgūdžių testo ir po pokalbio su personalo atrankos specialistu gauti rezultatai.

Išvados

1. Neraiškiosios aibės ir taisyklių rinkinys sudarytas apklausus dalykinės srities ekspertus ir įvertinus problemai reikalingą tikslumą.
2. Realizuoti ekspertinės sistemos fuzifikacijos, implikacijos, agregacijos ir defuzifikacijos procesai nenaudojant išorinių specializuotų bibliotekų.
3. Defuzifikacijos procesas atliekamas sėkmingai realizavus gravitacijos centro ir maksimumų vidurkio metodus.
4. Sistema ištestuota naudojant 3 skirtingus scenarijus, iliustruotas kriterijaus „minkštieji įgūdžiai“ reikšmingumas.
5. Nuspręsta nepanaudoti savanorystės trukmę atitinkančių neraiškiųjų aibių, tačiau užduoties sąlyga vis tiek įgyvendinta, kadangi be šio kriterijaus egzistuoja 3 įvestys su neraiškiosiomis aibėmis.
6. Neraiškiosiomis aibėmis paremta ekspertinė sistema leidžia imituoti žmogaus sprendimų priėmimo procesą.
7. Ekspertinės sistemos rekomendacijų tikslumas priklauso nuo sudaryto taisyklių rinkinio ir pasirinktų priklausomumo funkcijų.

Priedai

1 priedas. Programinis kodas

```
from fuzzy_variable import FuzzyVariable
from fuzzy_rules import Rule
import numpy as np
import utils

class FuzzyLogic:
    def __init__(self, variables, output):
        self.input_variables = variables
        self.output_variable = output

    def calculate_suitability(self, values):
        mds = self.fuzzification(values)
        return self.defuzzification(mds)

    def fuzzification(self, values):
        result = dict()
        for i in range(len(self.input_variables)):
            result[self.input_variables[i].var_name]                                    =
self.input_variables[i].get_membership_values(values[i])
        return result

    def defuzzification(self, values):
        rules = Rule()

        # 1. Getting degrees of membership for each category
        values = rules.calculate(values)

        # 2. Making new functions that define shapes of each category
        #    membership degree being the max value of y
        print(values)
        new_funcs = self.new_values(self.output_variable, values)
        self.som_mom(self.output_variable, values)
        new_funcs.modify_categories_md(values, self.output_variable)
        new_funcs.plot_categories(save=True)
        # 3. Calculating overlapping points
        _, intersect_points = self.find_intersections(new_funcs)

        # 4. Joining all xs and calculating max y values at each point
        points     =     self.calculate_joint_points(self.get_x_points(new_funcs,
intersect_points), new_funcs)
        #new_funcs.plot_categories(save=True, points=points)

        # 5. Filtering all points to keep the
        filtered_points = self.filter_points(new_funcs, points, values)
        print(f'Points after filtering: {filtered_points}')

        plots,     centroids     =     self.calculate_plots_centroids(new_funcs,
filtered_points)
        print(plots)
        print(centroids)

        return self.cog(plots, centroids)

    def find_max_values(self, fvar, values, eps = 1e-2):
        fvar_cats = fvar.get_categories()
        keys = list(fvar_cats.keys())
        rez = dict()
        for key in keys:
```

```
            ind = [i for i in range(len(fvar_cats[key])) if abs(fvar_cats[key][i]-
values[key]) <= eps]
            xlist = np.array(fvar.x_bounds)
            xlist = xlist[ind]
            if np.abs(values[key] - 1) <= eps:
                rez[key] = list(xlist)
                pass
            else:
                b = np.min(xlist)
                c = np.max(xlist)
                rez[key] = [b,c]
        return rez

    def new_values(self, fvar, values):
        x_md = self.find_max_values(fvar, values) # finds x points where
membership function reaches the degree of membership
        cbounds_org = fvar.cat_bounds
        cbounds_new = []
        for i in range(fvar.cat_num):
            bound = cbounds_org[i]
            x0 = bound[0]
            x1 = bound[len(bound)-1]
            new_points = x_md[fvar.cat_names[i]]
            if len(new_points) == 1:
                x_mid = x_md[fvar.cat_names[i]][0]
                cbounds_new.append([x0, x_mid, x1])
            else:
                cbounds_new.append([x0, new_points[0], new_points[1], x1])

        modified_suit = FuzzyVariable('New Fuzzy Functions', fvar.x_bounds,
fvar.cat_num, list(fvar.get_categories().keys()), cbounds_new)

        return modified_suit

    def find_intersections(self, fvar):

        intersection_points = dict()
        categories = fvar.get_categories()
        cnames = fvar.cat_names
        all_points = []

        for i in range(fvar.cat_num):
            temp = []
            for j in range(fvar.cat_num):
                if cnames[i] == cnames[j]:
                    continue
                calc = dict()
                match = self.find_matching_points(fvar.x_bounds,
fvar.categories[cnames[i]], fvar.x_bounds, fvar.categories[cnames[j]])
                calc['cat'] = cnames[j]
                #calc['y values'] = xy
                calc['(x y)'] = match
                temp.append(calc)
                if len(match) > 0:
                    all_points.append(match)
            intersection_points[cnames[i]] = temp
        #print('Matching points', intersection_points.values)
        return intersection_points, sum(all_points,[])

    def find_matching_points(self, x1_list, y1_list, x2_list, y2_list):
        matching_points = []

        for x1, y1, x2, y2 in zip(x1_list, y1_list, x2_list, y2_list):
            if abs(x1-x2) <= 0.01 and abs(y1-y2) <= 0.01 and y1 > 0 and y2 > 0:
```

```
                matching_points.append((x1, y1))
        #print(f'matching points: {matching_points}')
        return matching_points

    def calculate_plots_centroids(self, fvar, points):
        categories = fvar.get_categories()
        names_cat = list(categories.keys())
        plots = []
        centroids = []

        for i in range(len(points)-1):

            p1 = points[i] # current point
            p2 = points[i+1] # next point

            if p1[1] == 0 and p2[1] == 0:
                continue

            elif p1[1] == 0 or p2[1] == 0:
                plot, centroid = utils.triangle_plot_centre(p1, p2)
                print(f'Trikampis  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            elif p1[1] == p2[1]:
                plot, centroid = utils.rect_plot_centre(p1, p2)
                print(f'Stačiakampis  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            else:
                plot, centroid = utils.trap_plot_centre(p1, p2)
                print(f'Trapecija  = Taškai {p1}, {p2}')
                print(f'Plotas: {plot}, Centroidas: {centroid}')

            if plot > 0:
                plots.append(plot)
                centroids.append(centroid)

        return plots, centroids

    def cog(self, plots, centroids):
        product = 0
        sum = 0
        for i in range(len(plots)):
            sum += plots[i]
            product += (plots[i] * centroids[i])
        print(sum)
        print(product)
        return product/sum

    def som_mom(self, fvar, values):
        max_key = max(values, key=lambda k: values[k])
        max_value = values[max_key]
        rez = self.find_max_values(fvar, values)
        print('SOM', min(rez[max_key]))
        print('MOM', np.mean(rez[max_key]))



    '''
    Joins all x's where fuzzy variables start, intersect, reach their
max/membership degree.
        <param> fvar - FuzzyVariable object
        <param> matches - list of tuples holding intersection points
```

```
    Returns single list
    '''
    def get_x_points(self, fvar, matches):
        x_bounds = []
        #print(matches)

        for i in range(fvar.cat_num):
            #if isinstance(fvar.cat_bounds[i], str):
                #print(fvar.cat_bounds[i])
            x_bounds += fvar.cat_bounds[i]
            #print(f'step {i}: bounds: {x_bounds}')

        for j in matches:
            #if isinstance(j[0], str):
                #print(j[0])
            x_bounds += [j[0]]

        x_bounds = set(x_bounds)
        x_bounds = sorted(x_bounds)

        return x_bounds

    def calculate_joint_points(self, xs, fvar):
        '''
        Calculates max y values at each significant x

        Params:
        --------
        xs - list of significant x values
        fvar - FuzzyVariable object

        Returns:
        --------
        A list of points, that contain significant x's and max y values(every
function is compared the highest y is chosen)
       '''
        list_of_points = [] # holds (x,y) of significant points
        categories = fvar.get_categories()

        for x in xs:
            ys = []
            for cat in fvar.cat_names:
                if x != 0.0:
                    y = categories[cat][int(x/0.01)-1]
                else:
                    y = categories[cat][0]
                ys.append(y)
                #if y == 0.75:
                    #print(ys)
            pair = (x, np.max(ys))
            list_of_points.append(pair)
        return list_of_points

    def filter_points(self, fvar, points, values):
        #print(f'Points before filtering: {points}')
        bounds = fvar.cat_bounds
        mds = list(values.values())
        categories = list(fvar.get_categories().values())
        new_points = []

        current_sequence = [points[0]]

        for i in range(1, len(points)):
            x0, y0 = points[i]
```

```
            '''
            if self.is_valid == False:
                if len(current_sequence) > 1:
                    new_points.extend([current_sequence[0],   current_sequence[-
1]])
                current_sequence = []
                continue
            '''
            x1, y1 = points[i-1]

            if abs(round(y0, 2) - round(y1, 2)) <= 0.011 and abs(x0-x1) >= 0:
                current_sequence.append(points[i])
            else:
                if len(current_sequence) > 1:
                    new_points.extend([current_sequence[0],   current_sequence[-
1]])
                current_sequence = [points[i]]
            #print(f'Point: ({x0},{y0})     =   Sequence: {current_sequence}')
        if len(current_sequence) > 1:
            new_points.extend([current_sequence[0], current_sequence[-1]])
        #print('Filtered points in filtering process', new_points)
        new_points = self.remove_duplicates(new_points)
        return new_points

    def remove_duplicates(self, lst):
        new_lst = []

        for i in range(len(lst)):
            is_unique = True
            for j in range(i+1, len(lst)):
                if lst[i] == lst[j]:
                    #print(lst[j])
                    is_unique = False
            if is_unique:
                new_lst.append(lst[i])
        return new_lst
```

2 priedas. Taisyklių klasė

```
import re
import utils

class Rule:
    def __init__(self):
        pass


    @staticmethod
    def operator(val1, val2, op):
        match op:
            case 'and':
                return min(val1, val2)
            case 'or':
                return max(val1, val2)
            case 'not':
                if val1 is not None:
                    return 1 - val1
                elif val2 is not None:
                    return 1 - val2
                else:
                    return None
            case default:
                return None
```

```
    def calculate(self, dictionary):

        # AND -> min()
        # OR -> max()
        # NOT -> 1 - x
        education = dictionary['education']
        experience = dictionary['experience']
        soft_skills = dictionary['soft_skills']
        volunteering = dictionary['volunteering']
        suitability = dict()

        #suitability['very  low']  =  Rule.calc_very_low(education,  experience,
soft_skills, volunteering)
        suitability['low']  =  Rule.calc_low(education,  experience,  soft_skills,
volunteering)
        suitability['average']   =   Rule.calc_average(education,   experience,
soft_skills, volunteering)
        suitability['good'] = Rule.calc_good(education, experience, soft_skills,
volunteering)
        return suitability

    @staticmethod
    def calc_low(education, experience, soft_skills, volunteering):
        r1 = Rule.lowr1(education, experience, soft_skills, volunteering)
        r2 = Rule.lowr2(education, experience, soft_skills, volunteering)
        return max(r1, r2)

    @staticmethod
    def calc_average(education, experience, soft_skills, volunteering):
        r1 = Rule.avgr1(education, experience, soft_skills, volunteering)
        r2 = Rule.avgr2(education, experience, soft_skills, volunteering)
        r3 = Rule.avgr3(education, experience, soft_skills, volunteering)
        return max(r1, r2, r3)

    @staticmethod
    def calc_good(education, experience, soft_skills, volunteering):
        r1 = Rule.goodr1(education, experience, soft_skills, volunteering)
        r2 = Rule.goodr2(education, experience, soft_skills, volunteering)
        r3 = Rule.goodr3(education, experience, soft_skills, volunteering)
        return max(r1, r2, r3)

    #
    # ---------------------- GOOD -----------------------------
    #
    @staticmethod
    def goodr1(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(education['high'], experience['high'], 'and')
        p2 = Rule.operator(soft_skills['low'], None, 'not')
        p3 = Rule.operator(p1, p2, 'and')
        return p3

    @staticmethod
    def goodr2(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(education['high'], experience['average'], 'and')
        p2 = Rule.operator(p1, soft_skills['great'], 'and')
        return p2

    @staticmethod
    def goodr3(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(experience['high'], experience['average'], 'or')
        p2 = Rule.operator(education['average'], p1, 'and')
        p3 = Rule.operator(p2, soft_skills['great'], 'and')
        return p3
```

```
    #
    # ------------------------ AVERAGE ------------------------
    #
    @staticmethod
    def avgr1(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(education['high'], experience['average'], 'and')
        p2 = Rule.operator(p1, soft_skills['decent'], 'and')
        return p2

    @staticmethod
    def avgr2(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(education['high'], education['average'], 'or')
        p2 = Rule.operator(p1, experience['low'], 'and')
        p3 = Rule.operator(p2, soft_skills['great'], 'and')
        return p3

    @staticmethod
    def avgr3(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(experience['high'], experience['average'], 'or')
        p2 = Rule.operator(p1, education['low'], 'and')
        p3 = Rule.operator(p2, soft_skills['great'], 'and')
        return p3

    #
    # ------------------------- LOW -----------------------------
    #
    @staticmethod
    def lowr1(education, experience, soft_skills, volunteering):
        return Rule.operator(education['low'], experience['low'], 'and')

    @staticmethod
    def lowr2(education, experience, soft_skills, volunteering):
        p1 = Rule.operator(education['high'], education['average'], 'or')
        p2 = Rule.operator(experience['high'], experience['average'], 'or')
        p3 = Rule.operator(p1, p2, 'and')
        p4 = Rule.operator(p3, soft_skills['low'], 'and')
        return p4
```

3 priedas. Kriterijai ir neraiškiosios aibės

```
from utils import trapezoid_mf, trapezoid, triangular
import matplotlib.pyplot as plt
import numpy as np

# Defines a linguistic variable.
# Each object has:
# - range of x values
# - fuzzy values - categories,
# - number of said categories
# - bounds of every category - coefficients for a membership function
class FuzzyVariable:

    def __init__(self, name, x_bounds, nr_categories, cats, cat_bounds):
        self.var_name = name
        self.x_bounds = x_bounds
        self.cat_num = nr_categories
        self.cat_bounds = cat_bounds
        self.cat_names = cats
        self.categories = self.pair_cats_bounds(cats, cat_bounds)
```

```
    def pair_cats_bounds(self, categories, bounds):
        cat = dict()
        for i in range(self.cat_num):
            if len(bounds[i]) == 4 or len(bounds[i]) == 3:
                cat[categories[i]] = trapezoid_mf(self.x_bounds, bounds[i])
            else:
                raise NotImplementedError("More points are not supported.")
        return cat

    def get_categories(self):
        return self.categories

    def plot_categories(self, xlabel=None, title=None, save=False, y_vals=None,
points = None):
        if xlabel == None:
            xlabel=self.var_name

        if title == None:
            title = self.var_name

        keys = list(self.categories.keys())

        for i in range(self.cat_num):
            plt.plot(self.x_bounds,  self.categories[keys[i]],  label=f"{title}:
{keys[i]}")

        if points != None:
            x_values = [point[0] for point in points]
            y_values = [point[1] for point in points]

            # Plot the points
            plt.scatter(x_values, y_values)
            for i, point in enumerate(points):
                plt.text(point[0],       point[1],       f'({round(point[0],2)},
{round(point[1],2)})', fontsize=6, ha='right')

        plt.xlabel(xlabel)
        plt.ylabel('Membership value')
        plt.title(title)
        plt.legend()
        plt.grid(True)
        if save:
            plt.savefig(f'{title.replace(" ","")}.png')
        plt.show()


    def get_membership_values(self, x):
        if len(self.cat_bounds[0]) == 4:
            membership = dict()
            keys = list(self.categories.keys())

            for i in range(self.cat_num):
                membership[keys[i]] = trapezoid_mf(x, self.cat_bounds[i])

            return membership
```

```
        else:
            raise NotImplementedError("More points are not supported.")

            return None

    def modify_categories_md(self, values, fvar):
        for i in range(self.cat_num):
            val = values[self.cat_names[i]]
            lst = fvar.categories[self.cat_names[i]]
            self.categories[self.cat_names[i]] = [x if x <= val else val for x in
lst]
```

4 priedas. Papildomos funkcijos

```
import numpy as np

def trapezoid_mf(x_array, arg_list):
    type = len(arg_list)
    if len(arg_list) == 4:
        a, m1, m2, b = arg_list
    elif len(arg_list) == 3:
        a, m, b = arg_list

    if not isinstance(x_array, (int, float)):
        p = []
        for x in x_array:
            if type == 4:
                p.append(trapezoid(x, a, m1, m2, b))
            elif type == 3:
                p.append(triangular(x, a, m, b))
        return np.array(p)
    else:
        if type == 4:
            return trapezoid(x_array, a, m1, m2, b)
        elif type == 3:
            return triangular(x_array, a, m, b)

def trapezoid(x, a, m1, m2, b):
    if x >= m1 and x <= m2:
        return 1
    elif x > a and x < m1:
        return ((x - a)/(m1 - a))
    elif x < b and x > m2:
        return ((b - x)/(b - m2))
    else:
        return 0

def triangular(x, a, m, b):
    if x == m:
        return 1
    elif x > a and x < m:
        return (x - a) / (m - a)
    elif m < x and x < b:
        return (b - x) / (b - m)
    else:
```

```
        return 0

#           ------- TRIANGLE ----------
def triangle_plot_centre(p1, p2):
    hpoint = None
    lpoint = None

    if p1[1] == 0:
        hpoint = p2
        lpoint = p1
        centroid = tria_centr(hpoint, lpoint)
        if centroid < lpoint[0]:
            centroid += lpoint[0]
    else:
        hpoint = p1
        lpoint = p2
        centroid = tria_centr(hpoint, lpoint)
        if centroid < hpoint[0]:
            centroid += hpoint[0]

    plot = triangular_plot(hpoint, lpoint)

    return plot, centroid

def triangular_plot(hpoint, lpoint):

    b = hpoint[1]
    a = hpoint[0] - lpoint[0]

    return a*b/2

def tria_centr(hpoint, lpoint):
    return (hpoint[0] + hpoint[0] + lpoint[0])/3
#           ----------------------------

#          -------- RECTANGLE -----------
def rect_plot_centre(p1, p2):
    plot = rect_plot(p1, p2)
    centroid = rect_centr(p1, p2)

    return plot, centroid

def rect_plot(p1, p2):
    return abs(p1[0]-p2[0])*p1[1]

def rect_centr(p1, p2):
    return (p1[0] + p2[0])/2
#          --------------------------------

#          -------- TRAPEZOID ------------
def trap_plot_centre(p1, p2):
    h = abs(p1[0] - p2[0])
    plot = trap_plot(p1, p2, h)

    if p1[1] > p2[1]:
        a = p2[1]
```

```
            b = p1[1]
            centroid = trap_centr(a, b, h)
            if centroid < p2[0]:
                centroid += p2[0]
        else:
            a = p1[1]
            b = p2[1]
            centroid = trap_centr(a, b, h)
            if centroid < p1[0]:
                centroid += p1[0]

        return plot, centroid

def trap_plot(p1, p2, h):
    return ((p1[1] + p2[1])*h)/2

def trap_centr(a, b, h):
    top = h*(2*a+b)
    bottom = 3 * (a+b)
    return top/bottom
```