



Технически университет - София
филиал Пловдив

Курсов проект по дисциплина
Дискретни структури

**Тема: Клъстерен анализ. GMM алгоритъм за
клъстеризация.**

Изготвил: Тодор Пейчинов

Фак. №: 378950

Група: 41А

I. Клъстерен анализ

Клъстерен анализ или клъстеризация (*clustering*) е понятие от информатиката и математическото моделиране, с което се означава групиране на множество разнообразни обекти по такъв начин, че обектите от една и съща група (наречена *клъстер*) са най-сходни (по даден признак) помежду си в сравнение с обектите, отнесени към останалите клъстери. Клъстерният анализ е основна задача в анализа на данни и обичайно се използва в много области като разпознаване на образи, анализ на образи, извличане на информация, извличане на знания от данни, компресиране на данни, машинно обучение, биоинформатика.

Клъстерният анализ не обозначава един конкретен алгоритъм за класификация, а цялостната класификационна задача, която трябва да се реши. Решението може да се получи чрез множество алгоритми, които чувствително се различават по това как и колко ефективно се дефинират клъстерите. Обичайно ползваните определения за клъстери включват: групи от обекти с малки разстояния между тях, плътни области от пространството на данните, интервали, или конкретни статистически разпределения. Следователно, клъстеризацията може да се дефинира като многокритериална оптимизационна задача. Подходящият алгоритъм за клъстеризация и настройките на параметрите му (като функция за разстояние между обектите, праг на плътност, или брой очаквани клъстери в резултат от клъстеризацията) зависят от конкретния набор от данни и планираната употреба на резултатите от анализа.

В този смисъл клъстерният анализ е итеративен процес по извличане на знания от данни, който включва проби и грешки. Често е необходимо да се модифицират параметрите на модела и данните да се подлагат на предварителна обработка дотогава докато полученият резултат от анализа не постигне желаните качества. Съществуват над 100 публикувани алгоритъма за клъстеризация. Не може да се посочи един обективно „правилен“ алгоритъм, тъй като качеството на резултата от клъстеризацията зависи в голяма степен от преценката на изследователя и конкретните му нужди и цели – особеност, афористично изразена като „клъстеризацията е в очите на гледащия“ („clustering is in the eye of the beholder“).

Сред областите, в които клъстерният анализ намира приложение, са: медицина (медицинска образна диагностика), финанси, маркетинговите проучвания (пазарна сегментация, продуктово позициониране, разработка на нови продукти, определяне на тестови пазари), компютърна лингвистика (групиране на резултати при търсене по ключови думи), анализ на социални мрежи, криминалистика, математическа химия (*in silico* методи във фармацевтиката), климатология, и много други.

Процесът на разбиване на зададеното множество от физически или абстрактни обекти по групи от *сходни* обекти се нарича *клъстеризация*. Един клъстер представлява колекцията от обекти, които *приличат* един на друг *вътре в клъстера* и се *различават* от обекти от *други клъстери*. Клъстерният анализ е много важна част от човешката дейност. Дори още в ранното детство човекът се учи, например, как да различи котки от кучета или животни от растения чрез постоянно усъвършенстване на свои подсъзнателни схеми за клъстеризация. В търговията клъстерният анализ може да бъде използван за създаване на индивидуализирани маркетингови стратегии, базиращи се на откриване на различни групи потребители в съществуващите бази данни от клиенти. В

биологията той може да се използва за извеждането на таксономии на растения или животни, за категоризация на гени със сходна функционалност и т.н. Клъстеризацията може да подпомогне при идентификация на области от сходни по използване земи от базата данни с аеро или космически фотоснимки, за идентификация на групи сгради в един град съгласно техния тип, стойност и географско разположение. Тя също така може да подпомогне класификация на документи във Web с цел намиране на търсената информация. Клъстерният анализ може да бъде използван като автономното средство за анализ на разпределение на данни, за визуализация на характеристики на всеки клъстер и за фокусиране върху някой отделен клъстер с цел по-детайлния му анализ. От друга страна, той може да бъде използван като една предварителна стъпка в процеса за подготовката на данни за работа на други алгоритми, например класификация или описание на понятия.

Като клон на *математическата статистика*, изследванията върху клъстерния анализ от много години се фокусират основно върху *базиран на разстояние (distance-based)* подход към тази задача.. В *машинното самообучение* клъстеризацията е примерът на неуправлявано (без учител – unsupervised) обучение. В отличие от класификацията, клъстеризацията и самообучение без учител не използват предварително определени класове и вече класифицирани обучаващи примери. По тази причина клъстеризацията може да се разглежда по- скоро като една форма на *самообучение от наблюдения*, от колкото самообучение от примери. При *концептуалната клъстеризация* една група от обекти формира клъстер само, ако той може да бъде описан като понятие. Това се отличава от традиционната клъстеризация, която измерва сходството на база на геометричното разстояние. Концептуалната клъстеризация се състои от две компоненти: 1) откриване на подходящи клъстери и 2) формиране на описания за всеки клъстер, както при класификацията. И в този случай се прилагат критерии за силно вътре- клъстерното и слабо между-клъстерното сходство.

В литература е описано огромно количество различни алгоритми за клъстеризация. Изборът на конкретен алгоритъм зависи както от типа на налични данни, така и от конкретната цел на анализа. Ако клъстерният анализ се използва като инструмент за описание и изследване на данни, то е възможно да се опита няколко различни алгоритъма върху едни и същи данни, за да се види, какви зависимости могат да бъдат намерени.

II. Изисквания за методите на клъстерен анализ

Клъстеризацията е предмет на интензивни научни изследвания, където нейните потенциални приложения налагат свои специфични изисквания. Методите за клъстеризация, които се разработват и се прилагат, трябва да удовлетворяват на следните изисквания:

- **Разширяемост.** Съществуват много алгоритми за клъстеризация, които работят много добре върху малки множества от данни, съдържащи не повече от 200 обекта; обаче, една голяма база данни може да съдържа милиони обекти. Клъстеризацията, извършена върху *някоя извадка* от тези данни, може да доведе до изкривени резултати. Необходими са алгоритми за клъстеризация, които могат да работят върху много големи множества от данни.

- **Възможността за работа с различни типове атрибути.** Съществуват множество алгоритми, създадени за клъстеризация на непрекъснати (числови) данни. Обаче, някои приложения могат да изискват клъстеризация на данни от други типове, например двоични, номинални или от всички тези типове заедно.
- **Откриване на клъстери с различна форма.** Повечето от клъстерните алгоритми се базират върху Евклидово или абсолютното разстояние. Такива алгоритми имат тенденция да намират сферичните по форма клъстери с близък размер и плътност (гъстота). Обаче, един клъстер може да има произволна форма, поради това важно е да бъдат разработени алгоритми, способни да намират клъстери с произволна форма.
- **Минимизиране на изисквания към знанията за проблемната област за определяне на входни параметри.** Много от клъстерните алгоритми изискват от потребителя да въвежда определени параметри за желания клъстерен анализ (например броя на клъстери). Резултатите от клъстеризацията могат в значителна степен да зависят от тях. Определянето на подобни параметри често е доста трудна задача, особено за множества от данни, съдържащи обекти с голяма размерност. Това не само затруднява потребители, но и води до проблеми с контрола върху качеството на клъстеризацията.
- **Възможността за работа със зашумени данни.** Повечето от реалните бази данни съдържат екстремни или липсващи стойности, както и сгрешени данни. Някои от клъстерните алгоритми са чувствителни към подобни данни, което води до намаляване на качеството на клъстеризацията.
- **Нечувствителността към реда на постъпване на входни данни.** Някои клъстерни алгоритми са чувствителни към подредба на входни данни; например, едно и също множество от данни, когато е представено на такъв алгоритъм, при различна подредба на данни може да бъде разбито на абсолютно различни клъстери. По тази причина важно е да бъдат разработвани алгоритми нечувствителни към подредбата на данни.
- **Голяма размерност.** Една база данни може да съдържа голямо количество атрибути. Много от клъстерни алгоритми са добри при работата с данни с малка размерност, например с двумерни или тримерни данни. Човешките очи са много добър инструмент за определяне на качеството на клъстеризацията в пространството с до 3 размерности. Предизвикателството е да се клъстеризират обекти в многомерното пространство, особено отчитайки, че подобни данни могат да бъдат силно разпръснати и да имат много асиметрично разпределение в пространството.

- **Клъстеризация базирана на ограничения.** Някои реални приложения могат да изискват извършване на клъстеризацията при наличието на различни видове ограничения. Да предположим, че задачата е да бъдат избрани в града места за разполагане на зададения брой банкомати. За решаването ѝ вие можете да клъстеризирате жилищата с отчитане на такива ограничения като наличие на градски реки, пътища и т.н. Задачата за намиране на групи в данни с добро клъстеризационно поведение, които удовлетворяват зададени ограничения, е една амбициозна изследователска задача.
- **Разбираемостта и използваемостта.** Потребителите очакват резултатите от клъстеризацията да се поддават на интерпретация, да бъдат разбираеми и използваеми. С други думи, често клъстеризацията трябва да бъде свързана със семантичната интерпретация и приложения. Важно е да бъде разбрано, как една приложна цел може да повлияе на избора на конкретния метод за клъстеризация.

III. Типове данни, използвани при клъстерен анализ

Алгоритмите за клъстеризация, базирани на използване на основната памет, обикновено използват следните две структури данни:

Матрица на данните (или структура *обект-по-атрибут*). Тази структура има форма на релационна таблица или $n \times p$ матрица.

Матрица на различия (или структура *обект-по-обект*). Тя съхранява колекция от стойностите за близостта между всички двойки обекти. Често тя се представя чрез $n \times n$ матрица.

Как се изчислява различието между обекти, описани с атрибути от различни типове данни? Освен непрекъснати и номинални типове данни, съществуват и такива типове като двоични, атрибути с наредба (ordinal) и пропорционално-скалирани атрибути.

Непрекъснатите или **интервално-скалирани атрибути** са непрекъснати числови стойности на измервания, направени съгласно една приблизително линейна скала. Типичните примери са тегло, височина, температура и т.н. Трябва да се знае, че използваните мерни единици могат да окажат съществено влияние на резултатите от клъстерния анализ. Например, промяната от метри в инчи при измерване на височина може доведе до получаване на съвсем различни клъстерни схеми. В общия случай, представянето на един атрибут в по-малки мерни единици води до по-голям диапазон на стойностите на този атрибут и, следователно, до по-голям ефект върху структурата на клъстери. За да избегнем влияние от избора на мерните единици, данните трябва да

бъдат *стандартизирани* (макар, че в някои приложения на някои атрибути могат да бъдат целенасочено присвоени по-големи тегла).

За да бъдат стандартизирани непрекъснати атрибути, оригиналните измервания трябва да бъдат превърнати в *безразмерни единици*. В клъстерния анализ често се прилага *z-нормализация*.

Независимо от това, дали стойностите на непрекъснати атрибути са стандартизирани или не, различието (или сходство) между два обекта, описвани чрез такива атрибути, се измерва с помощта на някоя *мярка за разстояние*. Един общ клас от такива мерки се задава от функцията за *разстояние на Минковски*. Най-често използваните разстояния в клъстерния анализ са *Евклидово* (при $L=2$) и *абсолютното* (при $L=1$).

Един двоичен атрибут приема само два състояния: 0 или 1, където 0 означава, че атрибутът не присъства, а 1 – че присъства. Например, 1 за атрибут *Пушич* означава, че човекът пуши, а 0 - че не е.

При изчисляване на различието между двоични атрибути се използват два подхода. Първият от тях се базира на изчисляване на така наречена *таблица на съответствия* (*contingency table*). Ако всички двоични атрибути се третираат като имащи *еднакво тегло*, то за два обекта – x и y , описвани с p двоични атрибута, може да се построи следната 2×2 таблица на съответствията.

Един двоичен атрибут е *симетричен*, ако и двете му стойности имат еднакво тегло, т.е. няма никакво значение, кое от двата значения на такъв атрибут да бъде кодирано с 1, а кое с нула. Например, атрибутът *Пол*, приемащ състояния *мъж* или *жена*, е симетричен. Сходството, оценявано на базата на симетрични атрибути, се нарича *инвариантното сходство*, тъй като резултатът не се променя, при промяна на кодиране на някой от атрибутите. Оценката на сходство между два обекта, описвани със симетрични атрибути, най-често се прави чрез *прост коефициент на съвпадение* (*simple matching coefficient – SMC*).

Друга, често използвана мярка за различието на симетрични двоични атрибути е *разстоянието по Хеминг* (Hamming distance).

Един двоичен атрибут се нарича *асиметричен*, ако неговите състояния имат различна важност (тегло). Например, *положителен* и *отрицателен* резултат на атрибута *Тест за СПИН* имат съвсем различно тегло! По съгласение, за асиметричните атрибути чрез 1 се кодира състояние, което се среща по-рядко (положителен – за *Тест за СПИН*). Следователно, за асиметрични атрибути положителното съвпадение (съвпадение на 1-ци) е по-важно от отрицателното. По тази причина, асиметричните атрибути се разглеждат не като двоични, а като “единични”, т.е. имащи само едно състояние. Сходството, базирано върху асиметричните атрибути, се нарича *неинвариантното сходство*. Най-известна мярка за изчисляване на сходството между асиметрични атрибути е *Жакардов коефициент*.

Скаларното произведение е един вариант на разстоянието по Хеминг, което често се прилага като мярка за различието в случая на несиметрични атрибути.

Когато атрибутните стойности се кодират само с 0 и 1, само присъстващите признаци (т.е. 1-ци) допринасят за оценяването на сходството между обекти. Обаче, когато стойностите се кодират чрез -1 и $+1$ (както в невронни мрежи) – несъвпадащите стойности носят “наказателни” точки за тази мярка.

Текстовите документи често се представят като вектори, в които всеки атрибут представлява честотата на срещане на конкретния терм (дума) в документа. На практика, представянето е по-сложно, тъй като определени общи думи се игнорират, а прилагането на различни техники позволява отчитането на различни форми на една и съща дума, както и на различни дължини на документи и т.н.

Дори документите да съдържат хиляди или десетки хиляди атрибути (терми), всеки документ остава много разпръснат, тъй като има относително малък брой ненулеви атрибути. По този начин сходството между документи не трябва да зависи от броя на общи нулеви стойности, тъй като всяка двойка документи има голяма вероятност „да не съдържа” много едни и същи думи. С други думи, ако броим съвпадения от типа на $0 - 0$, то ще получим, че всеки документ е много сходен с повечето от други документи. По тази причина една мярка за определяне на сходство между документи трябва да игнорира съвпадения от типа на $0 - 0$, както това прави, например, Жакардов коефициент, но да е в състояние да работи с не-двоични вектори. Един от възможни варианти за такава мярка е разширения Жакардов коефициент, известен още като коефициент на Танимото (Tanimoto). По-често използвана мярка за оценяване на сходство между документи е мярката на косинус. На практика, тази мярка измерва косинус на ъгъла между вектори \mathbf{x} и \mathbf{y} , така че тя е 1, когато ъгълът между тези вектори е 0° , т.е. когато \mathbf{x} и \mathbf{y} са едни и същи, без да отчитаме тяхната дължина. Ако стойността на мярката е 0, то ъгълът между \mathbf{x} и \mathbf{y} е 90° и съответните документи нямат нито една обща дума.

Корелацията между два обекта, описани с двоични или непрекъснати атрибути е мярка за наличието на линейната зависимост между техните атрибути. Стойността на коефициента на корелация е между -1 и 1 , като стойностите 1 (-1) означават, че между \mathbf{x} и \mathbf{y} съществува перфектна линейна зависимост, т.е. $x_i = ay_i + b$, където a и b са константи.

Нелинейна зависимост. Стойността на коефициента на корелация равна на 0 означава липсва на линейната зависимост между атрибутите на сравняваните обекти. Обаче, в този случай може да съществува някаква нелинейна зависимост между тях.

Визуализация на корелация. Наличието на корелацията между двойка обекти \mathbf{x} и \mathbf{y} може лесно да се установи визуално чрез графично изобразяване на съответните двойки на атрибутните стойности. Стойностите на тези атрибути са генерирани по случаен начин (с нормално разпределение) така, че корелацията между \mathbf{x} и \mathbf{y} да варира от -1 до 1 . Всяко кръгче от един такъв график представлява един от 30 атрибута; неговата x координата отговаря на стойността на съответния атрибут в обекта \mathbf{x} , а неговата y координата – на стойността на същия атрибут в обекта \mathbf{y} .

Повечето от досега определени метрики за изчисляване на разстояние (например Евклидова) отчитат разликата в диапазона на стойностите на атрибутите, но не отчитат възможност за наличието на корелацията между атрибутите. Едно обобщение на Евклидовото разстояние – т.н. *разстояние по Махаланобис* (Mahalanobis distance) е

удобно да се ползва в случаите, когато атрибутите са корелират, имат различни диапазони на стойностите (т.е. различна вариабелност) и разпределението на данни е приблизително нормално (Гаусово). Практическото прилагане на разстоянието по Махаланобис е много скъпо от изчислителната гледна точка, но има смисъл когато атрибутите корелират. Ако атрибутите корелират слабо, но имат различни диапазони на стойностите, е достатъчно да се използват процедури по предварителната нормализация на атрибутите.

Номиналният атрибут е едно обобщение на двоичния, при което атрибутът може да приема повече от едно състояние. Типичният пример на номиналният атрибут е *Цвят*, който приема, например, състояния *червен*, *жълт*, *зелен*, *розов* и *син*. Често за удобство M състояния на един номинален атрибут се кодират чрез цели числа – от 1 до M , като не се предполага никаква наредба между тези числови стойности.

Номиналните атрибути могат да бъдат кодирани като *асиметрични* двоични атрибути чрез прилагане на *процедурата за бинаризация*. При тази процедура всеки номинален атрибут, приемащ M състояния, се заменя с M асиметрични двоични атрибути, приемащи стойност 1 за съответното състояние. Например, значение на номиналният атрибут *Цвят* = *червен*, се кодира със следните пет двоични атрибута: *Цвят_червен* = 1; *Цвят_жълт* = 0; *Цвят_зелен* = 0; *Цвят_розов* = 0; *Цвят_син* = 0. При използване на подобна бинаризация, различието между обекти, описвани с номинални атрибути, може да бъде оценено чрез вече ни познатия Жакардов коефициент .

Един **дискретен атрибут с наредба** (*ordinal*) е такъв номинален атрибут, в който неговите M състояния са подредени в определена, имаща смисъл наредба. Атрибутите с наредба се използват за описание на субективната оценка на такива величини, които не могат да бъдат измерени обективно. Например, професионалните звания често се представят като една наредба, от типа на *асистент*, *главен асистент*, *доцент*, *професор*. **Непрекъснатият атрибут с наредба** представлява един непрекъснат атрибут, измерен по неизвестна скала, т.е. при този атрибут важна не конкретната му стойност, а *относителния порядък* на неговите стойности. Например, относителното ранжиране в спорта (т.е. златен, сребърен, бронзов медал) често по-важно от конкретния резултат. Атрибутите с наредба могат да бъдат получени чрез дискретизацията на непрекъснатия атрибут. Значенията на атрибута с наредба могат да бъдат превърнати в *рангове*. Например, ако един атрибут с наредба f има M_f наредени състояния, то те могат да бъдат превърнати в рангове 1, ..., M_f .

При изчисляването на различието между обекти, третирането на атрибутите с наредба е почти същото, както и при непрекъснати атрибути. Ако f е атрибут с наредба от M_f наредени състояния, то различието между два обекта по отношение на този атрибут се изчислява следните 3 стъпки:

1. За всеки обект i стойността x_{if} на атрибута с наредба f се заменя със съответния му ранг: $r_{if} \in \{1, \dots, M_f\}$.
2. Тъй като различни атрибути с наредба могат да имат различен брой състояния, за да имат всички те едно и също тегло трябва техните стойности да бъдат

стандартизирани, например в интервал $[0.0, 1.0]$. Това може да бъде постигнато чрез замяна на рангове със техни нормализирани стойности.

3. Различието между обекти се изчислява чрез използване на някоя от мерките за разстояние, като атрибут f вече се третира като *непрекъснат* със стойностите.

Пропорционално-скалираният атрибут представлява някакво положително измерване, направено по една нелинейна скала. Различието между обекти, описвани чрез пропорционално-скалираните атрибути, може да бъде изчислено по следните три метода:

1. Всички пропорционално-скалирани атрибути да се третират като *обикновени непрекъснати* атрибути. Обаче, това не е добър подход при силно нелинейни скали.
2. Прилагане на *логаритмичната трансформация*: $y_{if} = \log(x_{if})$ и третирането трансформираният атрибут като непрекъснат със стойностите y_{if} . Могат да бъдат приложени и други трансформации в зависимост от известна дефиниция на измерваната стойност и конкретното приложение.
3. Третиране на x_{if} като един *непрекъснат атрибут с наредба*, ранжиране на съответните стойности и последващото им третиране като стойностите на непрекъснатия атрибут.

Последните два метода се смятат за най-ефективни, макар че изборът на конкретния метод може да зависи от конкретното приложение.

В общия случай базата данни може да съдържа p различни атрибута от всички описани по-горе типове. В този случай за получаване на добри резултати от клъстерния анализ е необходимо всички те да бъдат комбинирани в една обща матрица на различия, като всички атрибути да бъдат приведени към една обща скала с диапазон $[0.0, 1.0]$.

IV. Категории методи за клъстеризация

Съществуващите методи за клъстерния анализ могат да бъдат групирани по следните категории:

Методи за разделяне (*partitioning*). При зададена база от данни, съдържаща n обекта, един метод за разделяне разбива данни на k групи – клъстери, като $k \leq n$. Всички групи заедно трябва да удовлетворяват на следните изисквания: 1) всяка група трябва да съдържа най-малко един обект, и 2) всеки обект трябва да принадлежи точно към една група. (Това изискване може да бъде ослабено в някои методи за размито разделяне) При зададения брой на групи k , подлежащи на конструиране, методът за разделяне създава някое първоначално разбиване. След това той прилага една избрана *итеративна техника за преместване* на обекти, опитвайки се да подобри качеството на разбиване чрез преместване на обекти от една група в друга. Общият критерий за качеството на разделянето е “близостта” на обекти, намиращи се в един и същ клъстер, и “отдалечеността” на обекти, намиращи се в различни клъстери. Съществуват различни видове други критерии за оценка на качеството на разделянето.

За да бъде получено едно *глобално оптимално* решение при клъстеризацията чрез разделяне, е необходимо изчерпващото претърсване на всички възможни разбивания. Вместо това в повечето случаи се използват един от следните два популярни *евристични метода*: 1) алгоритъм *k-means* (т.е. *k-средни*), в който всеки клъстер се представя чрез средното на всички обекти от клъстера, и 2) алгоритъм *k-medoids*, в който клъстерът се представя чрез един от обекти, разположен близо до центъра на клъстера. Тези евристични методи за клъстеризация дават много добри резултати при намиране на сферични по форма клъстери в бази от данни с малки или средни размери. За намирането на клъстери със сложна форма или за клъстеризация на много големи бази данни, методите, базирани на разделяне, се нуждаят от определено разширение.

Йерархични методи. Един йерархичен метод създава йерархичната декомпозиция на зададеното множество от обекти (данни). Йерархичните методи за клъстеризация могат да бъдат разбити на *агломеративни* (*обединяващи*) и *разделящи* (*devisive*) в зависимост от начина за формиране на йерархичната декомпозиция. *Агломеративният подход* (който често се нарича подходът *отдолу - нагоре*) започва работата си третирайки всеки обект като отделна група – клъстер. След това той последователно слива групите, които са “близки” помежду си, докато или всички групи не се слейт в една (най-горното ниво в йерархията), или докато не бъде удовлетворен зададения критерий за прекратяване на сливането. Разделящият подход (който още се нарича подходът *отгоре - надолу*) започва работата си с поместване на всички обекти в един клъстер. След това той итеративно разбива всеки клъстер на по-малки клъстери, докато или всеки обект не бъде поместен в отделен клъстер, или докато не бъде удовлетворен зададения критерий за прекратяване на разделянето.

Основният недостатък на йерархичните методи се състои в това, че след извършването на някоя стъпка (сливане или разделяне), тя *никога не може да бъде преразгледана*, т.е. грешните решения никога не могат да бъдат поправени. Това е следствието от локалния, евристичен метод за търсене, използван, за да бъде избегната комбинаторната експлозия при пълното претърсване на всички възможни избори за намиране на глобалното оптимално решение. Съществуват два подхода за подобряване на качеството на йерархичната клъстеризация: 1) при всяко йерархично разделяне да бъде направен щателен анализ на всички “свързвания” на обекти, както това се прави в такива алгоритми като CURE и Chameleon, или 2) да се интегрира йерархичната

агломерация с итеративно преместване на обекти чрез първоначално използване на определен алгоритъм за йерархична агломеративна клъстеризация, а след това – уточняване на резултата чрез прилагане на метода за итеративното преместване, както това е направено в алгоритъма BIRCH.

Методите, базирани на плътността. Повечето от клъстерните методи за разбиване се базират върху изчисляване на разстояние между обектите. Такива методи могат да намират само сферични по форма клъстери и срещат големи затруднения при намирането на клъстери с произволна форма. Съществуват други клъстерни методи, базирани се на понятие *плътност*. Основната идея е да се продължава разрастване на дадения клъстер до тогава, докато неговата плътност (т.е. брой обекти или точки от данни в него) остава над определен праг. С други думи, за всеки обект, намиращ се в дадения клъстер, неговата околност със зададения радиус трябва да съдържа най-малко определен минимум от обекти. Подобният метод може да се използва за филтриране на шума (крайности), както и за откриване на клъстери с произволна форма.

Един типичен метод, създаващ клъстери в съответствие с прага на плътността, е алгоритъм DBSCAN. Алгоритъм OPTICS е един базиран на плътността метод, който изчислява определена клъстерна наредба с цел провеждане на автоматичен и интерактивен клъстерен анализ.

Решетъчни методи. Решетъчните методи “дискретизират” цялото пространство от обекти на едно крайно множество от клетки, които формират решетъчната структура. Всички операции по клъстеризация се изпълняват върху тази структура (т.е. върху дискретизираното пространство). Основното предимство на този подход е бързото време за обработка, което обикновено не зависи от броя на обекти, а само от броя на клетки във всяка от размерности на това дискретизирано пространство.

Типичният пример на такива алгоритми е STING. Алгоритмите CLIQUE и Wave-Cluster обединяват решетъчния подход с използването на плътността.

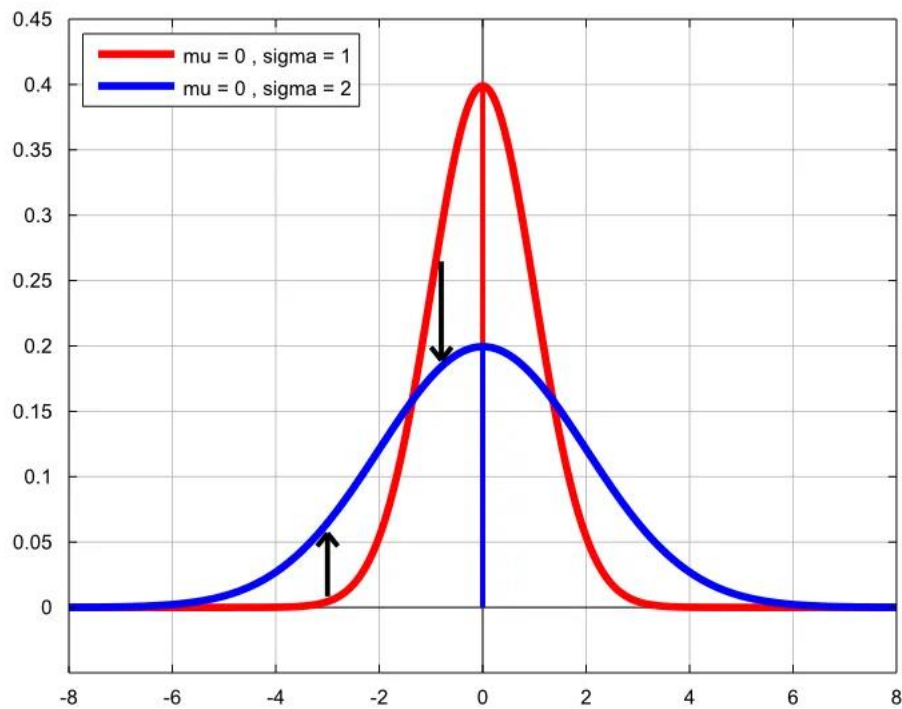
Методи, базирани на модели. Методите, базирани на модели, предполагат, че всеки клъстер може да бъде описан с помощта на определен модел, и опитват да “нагласят” по най-добрия начин наличните данни към тези модели. Един алгоритъм, базиран на модели, може да локализира клъстерните центрове чрез конструиране на определена функция за вероятностната плътност, описваща пространственото разпределение на обекти. Този подход позволява автоматично да се намира броят на клъстери, базирайки се на стандартните статистики, както и да се определят шум и екстремните обекти.

Някои от клъстерните алгоритми интегрират идеи от различни методи за клъстеризация, което води до определени затруднения с тяхното класифициране по категориите. Освен това, определени приложения могат да имат свои специфични критерии за клъстеризация, което изисква интегриране на няколко клъстеризационни методи.

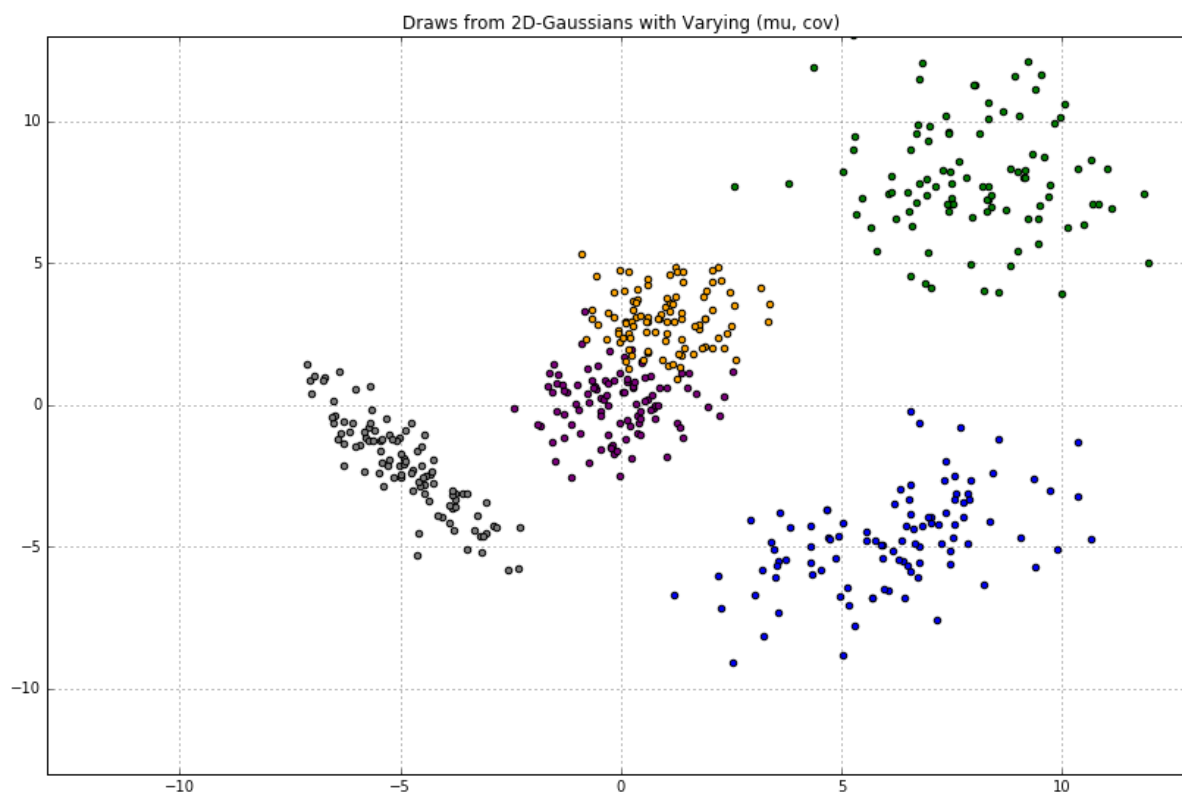
V. Gaussian Mixture Models (GMMs)

Gaussian mixture models могат да бъдат използвани за клъстериране на всякакъв вид данни по подобен начин на K-means алгоритмите. Въпреки това има редица преимущества на GMM, пред K-means алгоритмите.

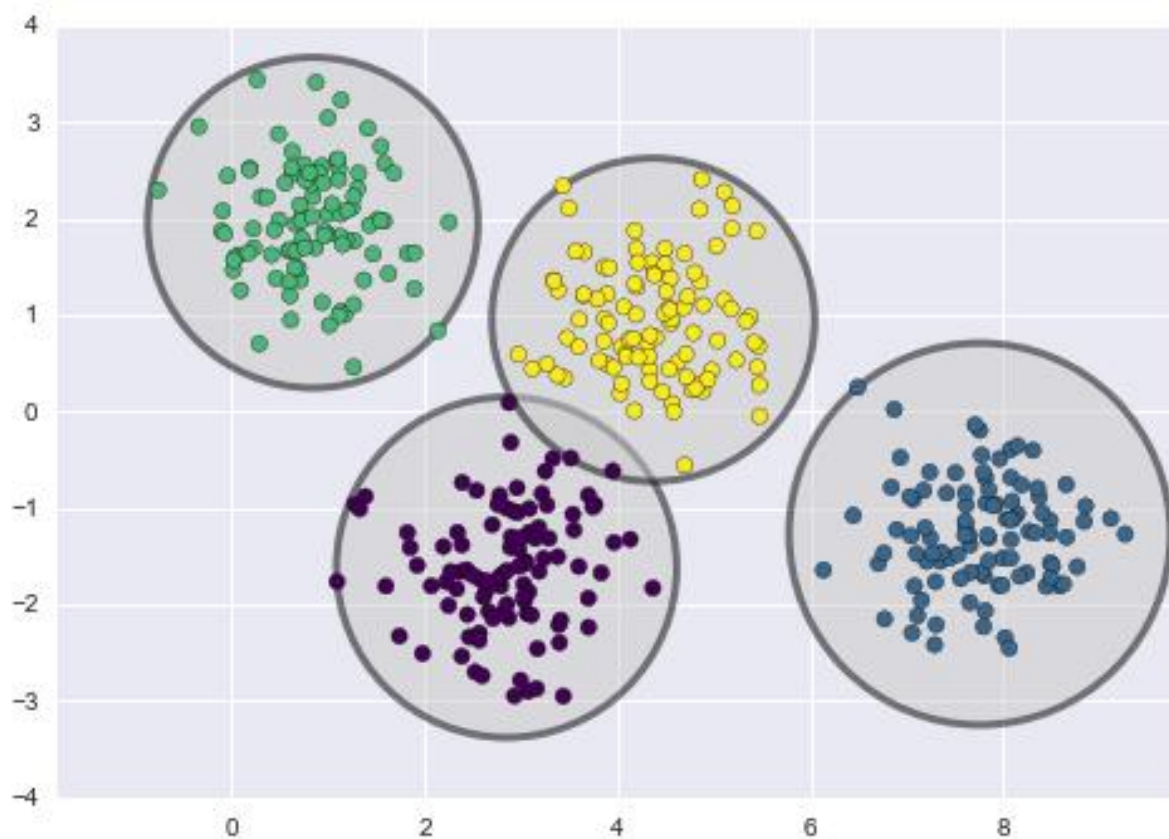
Първото и най-важно преимущество на GMM е, че K-means не отчита дисперсията, като под дисперсия се има предвид ширината на камбановидната крива.



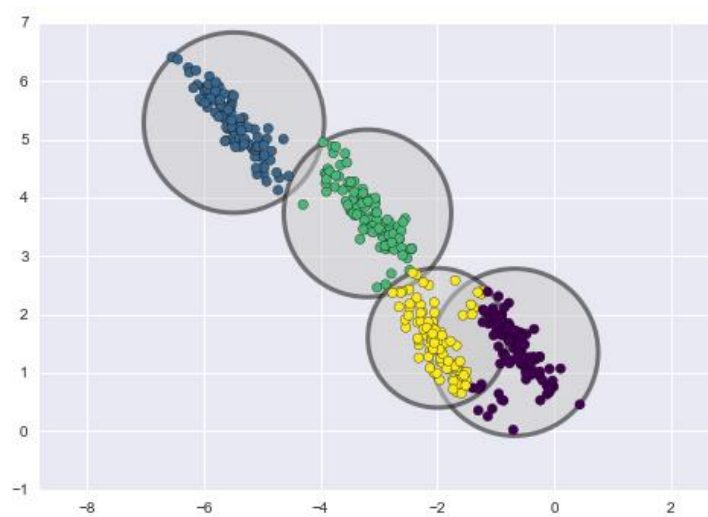
В две измерения дисперсията (по-точно ковариацията) определя формата на разпределението.



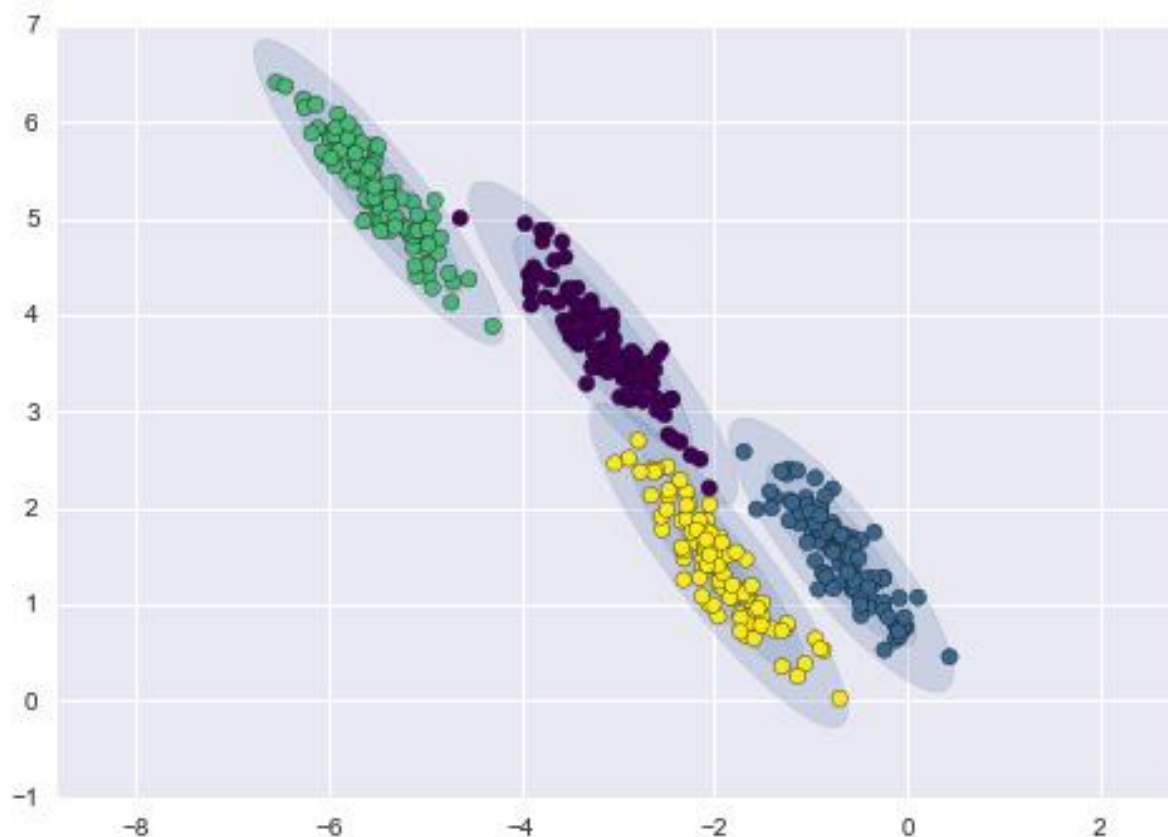
Един от начините да гледаме на K-means модела е, че при него се поставя окръжност (или при по-високите измерения хиперсфера) в центъра на всеки клъстер, с радиус дефиниран в най-отдалечената точка на клъстера.



Това работи добре, когато става въпрос за данни, които са кръгови. Когато данните, които използваме обаче приемат различни форми, резултата, който получаваме е нещо такова:



За разлика от тях Гасусовите модели могат да се справят с много продълговати клъстери:



Друга съществена разлика между K-means и GMMs е тази, че K-means извършва твърда класификация, докато GMM извършва мека класификация. Или казано с други думи, K-means ни показва коя точка от данни, на кой клъстер принадлежи, но не ни предоставя вероятностите дадена точка от данни да принадлежи към всеки от възможните клъстери.

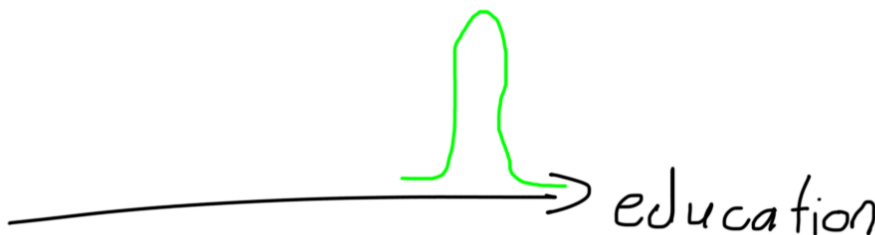
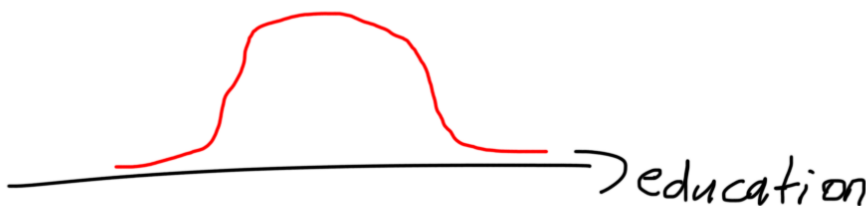
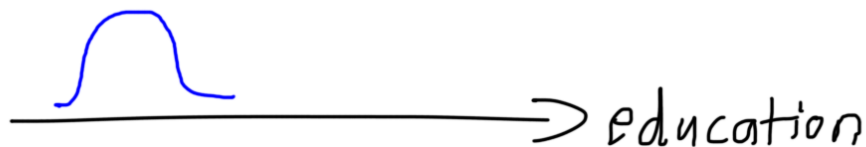
При извикването на функцията за прогнозиране (`gmm.predict(X)`) моделът ще присвои всяка точка от данни на един от клъстерите.

```
array([2, 1, 0, 1, 2, 2, 3, 0, 1, 1, 3, 1, 0, 1, 2, 0, 0, 2, 3, 3, 2, 2,
       0, 3, 3, 0, 2, 0, 3, 0, 1, 1, 0, 1, 1, 1, 1, 1, 3, 2, 0, 3, 0, 0,
       3, 3, 1, 3, 1, 2, 3, 2, 1, 2, 2, 3, 1, 3, 1, 2, 1, 0, 1, 3, 3, 3,
       1, 2, 1, 3, 0, 3, 1, 3, 3, 1, 3, 0, 2, 1, 2, 0, 2, 2, 1, 0, 2, 0,
       1, 1, 0, 2, 1, 3, 3, 0, 2, 2, 0, 3, 1, 2, 1, 2, 0, 2, 2, 0, 1, 0,
       3, 3, 2, 1, 2, 0, 1, 2, 2, 0, 3, 2, 3, 2, 2, 2, 2, 3, 2, 3, 1, 3,
       3, 2, 1, 3, 3, 1, 0, 1, 1, 3, 0, 3, 0, 3, 1, 0, 1, 1, 1, 0, 1, 0,
       2, 3, 1, 3, 2, 0, 1, 0, 0, 2, 0, 3, 3, 0, 2, 0, 0, 1, 2, 0, 3, 1,
       2, 2, 0, 3, 2, 0, 3, 3, 0, 0, 0, 0, 2, 1, 0, 3, 0, 0, 3, 3, 3, 0,
       3, 1, 0, 3, 2, 3, 0, 1, 3, 1, 0, 1, 0, 3, 0, 0, 1, 3, 3, 2, 2, 0,
       1, 2, 2, 3, 2, 3, 0, 1, 1, 0, 0, 1, 0, 2, 3, 0, 2, 3, 1, 3, 2, 0,
       2, 1, 1, 1, 1, 3, 3, 1, 0, 3, 2, 0, 3, 3, 3, 2, 2, 1, 0, 0, 3, 2,
       1, 3, 0, 1, 0, 2, 2, 3, 3, 0, 2, 2, 2, 0, 1, 1, 2, 2, 0, 2, 2, 2,
       1, 3, 1, 0, 2, 2, 1, 1, 1, 2, 2, 0, 1, 3])
```

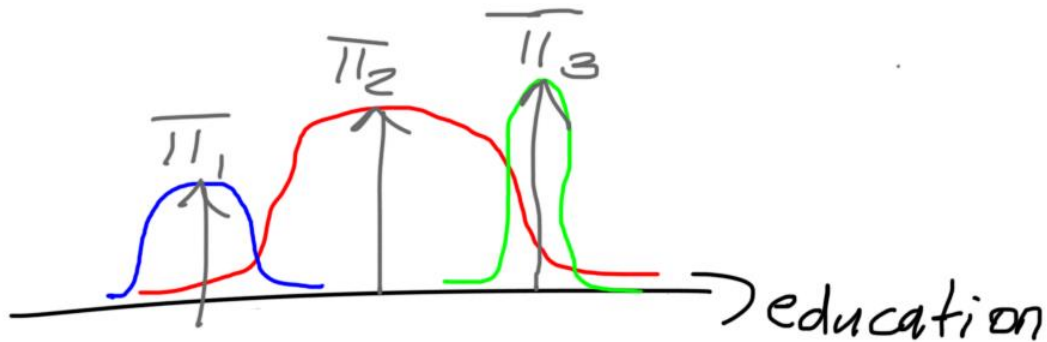
От друга страна обаче, при извикването на функцията `predict_proba` (`gmm.predict_proba(X)`), тя ще ни върне вероятностите точка от данни да принадлежи на всеки един от клъстерите.

```
array([[0.026, 0.000, 0.972, 0.002],  
       [0.000, 1.000, 0.000, 0.000],  
       [1.000, 0.000, 0.000, 0.000],  
       ...,  
       [1.000, 0.000, 0.000, 0.000],  
       [0.000, 1.000, 0.000, 0.000],  
       [0.000, 0.000, 0.000, 1.000]])
```

Както подсказва името, Гаусовия модел включва смесването (т.е. суперпозицията) на множество гаусови разпределения. За целите на обяснението, да предположим, че имаме три дистрибуции, съставени от проби от три различни класа. Синият Гаус представлява нивото на образование на хората, които съставляват по-ниската класа. Червеният Гаус представлява нивото на образование на хората, които съставляват средната класа, а зеленият Гаус представлява нивото на образование на хората, които съставляват висшата класа.



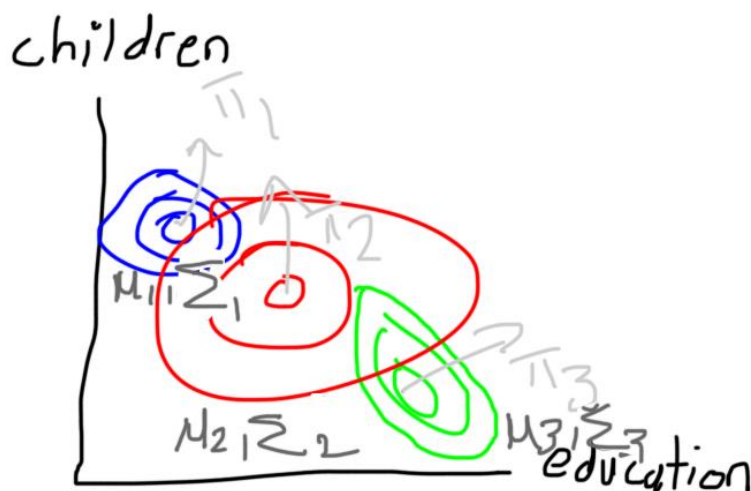
Без да знаем кои проби от кой клас идват, нашата цел е да използваме GMMs, за да присвоим точките от данни към подходящия клъстер. След като обучим модела, в идеалния случай бихме завършили с три разпределения на една и съща ос. След това, в зависимост от нивото на образование на дадена извадка (къде се намира на оста), ще я поставим в една от трите категории.



Всяко разпределение се умножава по тегло π , за да се отчете фактът, че нямаме равен брой проби от всяка категория. С други думи, може да сме включили само 1000 души от висшата класа и 100 000 души от средната класа.

Тъй като имаме работа с вероятности, теглата трябва да се добавят към 1, когато се сумират.

Ако решим да добавим друго измерение като броя на децата, тогава може да изглежда по следния начин:



Да предположим, че искаме да знаем каква е вероятността i -тата проба да идва от Gaussian k . Можем да изразим това като $p(z_i = k | \theta)$, където θ тук представлява средната стойност, ковариацията и теглото за всеки Гаус.

$$\theta = \{M1, M2, M3, \Sigma 1, \Sigma 2, \Sigma 3, \pi1, \pi2, \pi3\}$$

Може също да се срещне уравнението, записано като π . Това не трябва да се бърка с теглото, свързано с всеки Гаус. $p(z_i = k | \theta) = \pi_k$

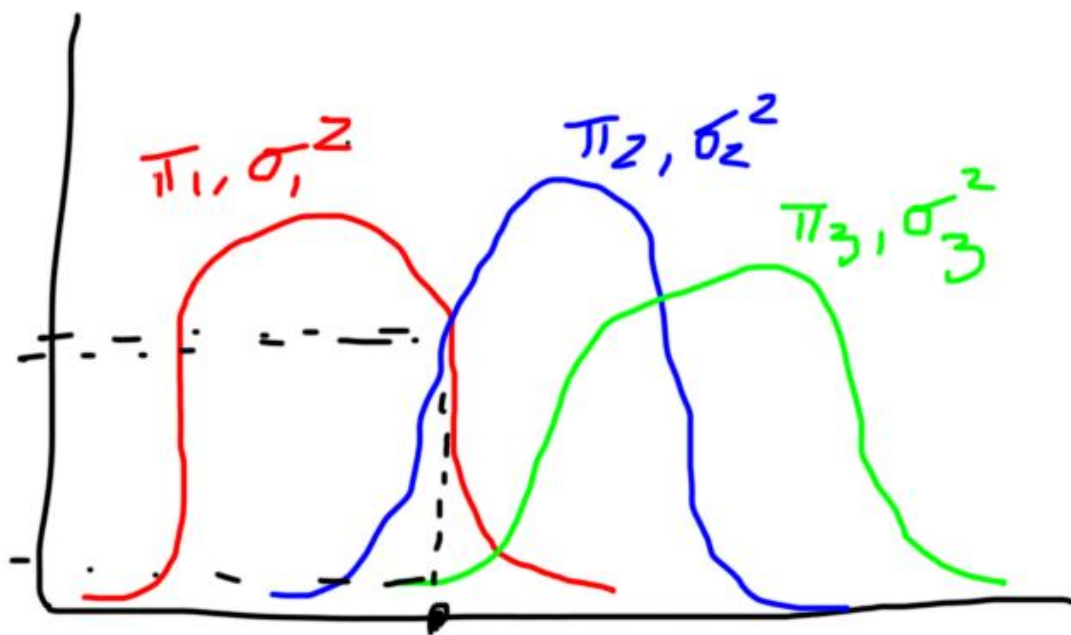
След това изразяваме вероятността да наблюдаваме точка от данни, като се има предвид, че идва от Гаус K като $p(x_i | z_i = k, M_k, \Sigma_k)$.

Да предположим, че имаме гаусово разпределение, където хоризонталната ос е различните IQ резултати, които индивидът би могъл да получи, от най-ниския до най-високия. Можем да разберем колко вероятно е даден индивид да има коефициент на интелигентност 120, като начертаем вертикална линия от позицията по оста x до кривата и след това погледнем съответната стойност на оста y . Стойността на y във всяка точка е равна на уравнението по-горе.

Ако искаме да знаем вероятността да наблюдаваме извадката i , като вземаме предвид всички различни разпределения, ние просто сумираме вероятностите да наблюдаваме извадката, като се има предвид, че тя идва от всяко от възможните гаусови разпределения.

$$p(x_i | \theta) = \sum_{k=1}^3 p(x_i | z_i = k, \theta) p(z_i = k | \theta)$$

Казано по друг начин, ние вземаме една извадка (ред) от нашия набор от данни, разглеждаме една характеристика (т.е. ниво на образование), начертаваме нейната позиция по оста x и сумираме съответните стойности на y (вероятност) за всяко разпределение.



За да разширим това до всички проби в нашия набор от данни. Предполагаме, че вероятността за наблюдение на една проба е независима от всички останали и тогава можем просто да ги умножим.

$$L(\theta) = p(x | \theta) = \prod_{i=1}^N \sum_{k=1}^3 p(x_i | z_i = k, \theta) p(z_i = k | \theta)$$

По-често вземаме логаритъм на вероятността, защото умножението на две числа вътре в логаритъм е равно на сумата от логаритмите на неговите съставни части и е по-лесно да добавяме числа, отколкото да ги умножаваме.

$$\log_b(M \cdot N) = \log_b^M + \log_b^N$$

$$\log(p(x | \theta)) = \sum_{i=1}^N \log \left(\sum_{k=1}^3 N(x_i | \mu_k, \sigma_k^2) \pi_k \right)$$

Тепърва трябва да обърнем внимание на факта, че имаме нужда от параметрите за всеки гаус (т.е. дисперсия, средна стойност и тегло) за да клъстерираме нашите данни, но трябва да знаем коя извадка към кой гаус принадлежи, за да оценим същите тези параметри.

Тук се намесва максимизирането на очакванията. На високо ниво алгоритъмът за максимизиране на очакванията може да бъде описан, както следва:

1. Започване със случайни Гаусови параметри (θ)

2. Повтаряне , докато не получим сближаване:

а) Стъпка на очакване: Изчисляване на $p(z_i = k | x_i, \theta)$. С други думи, проба i изглежда ли сякаш идва от клъстер k ?

б) Стъпка на максимизиране: Актуализиране на гаусовите параметри (θ), за да отговарят на присвоените им точки.

В стъпката на максимизиране искаме да увеличим максимално вероятността всяка проба да идва от разпределението. Спомнете си, че вероятността е височината на кривата в точка по оста x . Следователно искаме да модифицираме дисперсията и средната стойност на разпределението, така че височината на диаграмата във всяка точка от данни да е максимална.

Това повдига въпроса „Как трябва да подходим към избора на оптималните стойности за дисперсията и средната стойност.“ Използвайки общата форма на максимизиране на очакванията, можем да изведем набор от уравнения за средната стойност, дисперсията и теглото.

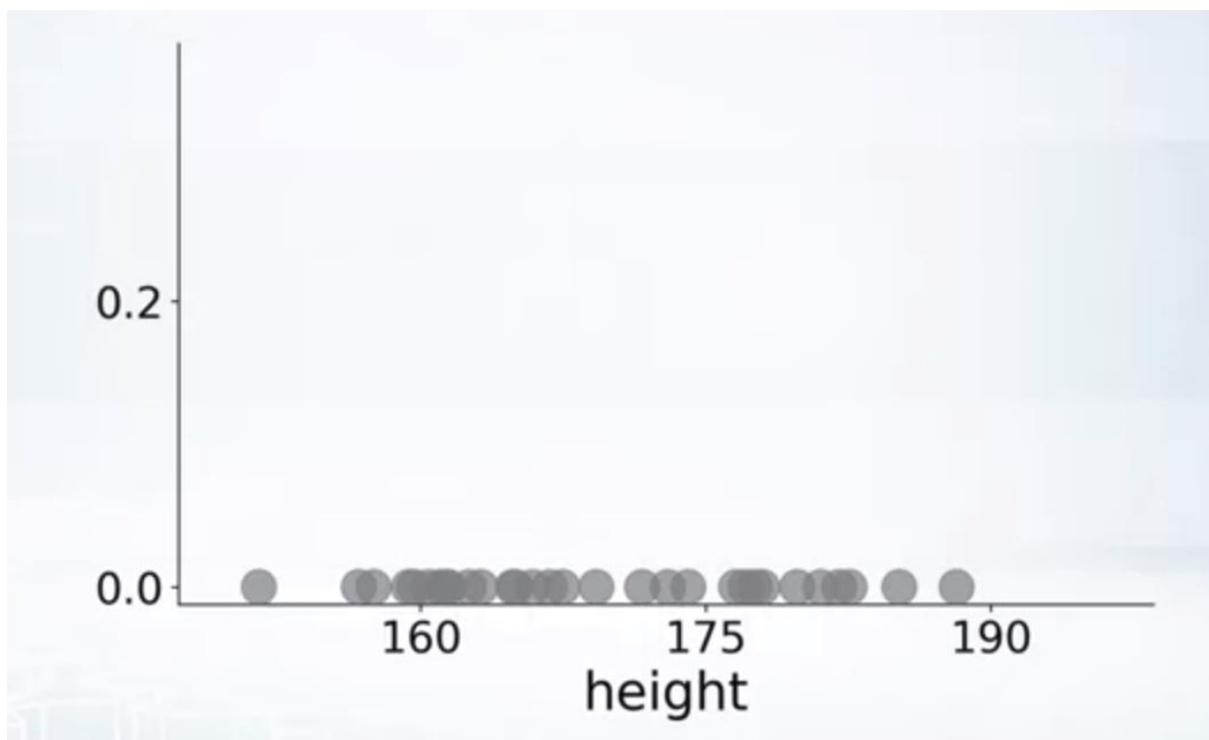
$$\begin{aligned} \max \sum_{i=1}^N E q(z) \log p(x_i, z_i | \theta) &= \sum_{i=1}^N \sum_{k=1}^3 q(z_i = k) \log \left(\frac{1}{N} \exp \left(-\frac{(x_i - Mk)^2}{2\sigma^2} \right) \pi_k \right) = \\ &= \sum_{i=1}^N \sum_{k=1}^3 q(z_i = k) \left(\log \frac{\pi_k}{N} - \frac{(x_i - Mk)^2}{2\sigma^2} \right) \\ \frac{\partial}{\partial Mk} &= \sum_{i=1}^N q(z_i = k) \left(0 - \frac{2(x_i - Mk)(-1)}{2\sigma^2} \right) \\ 0 &= \sum_{i=1}^N q(z_i = k)(x_i) - \sum_{i=1}^N q(z_i = k)Mk \\ Mk &= \frac{\sum_{i=1}^N q(z_i = k)x_i}{\sum_{i=1}^N q(z_i = k)} \end{aligned}$$

Можем да следваме същия процес, за да получим уравненията за ковариацията и теглото.

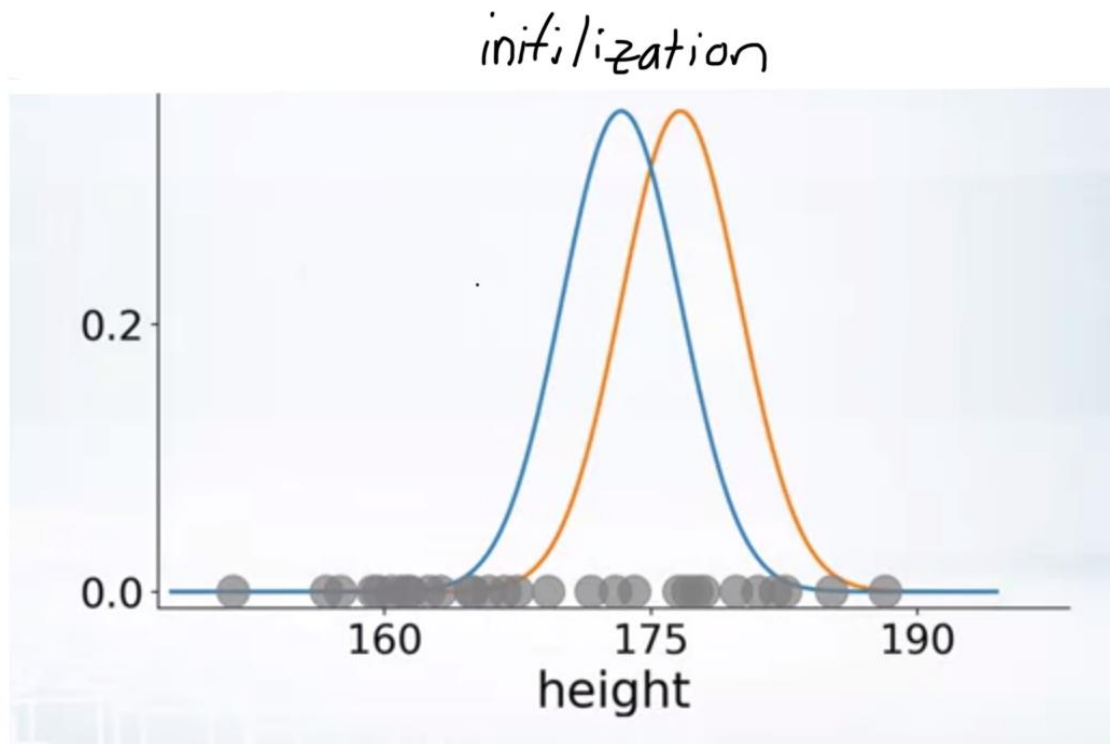
$$Mc = \frac{\sum_{i=1}^N q(zi = k)xi}{\sum_{i=1}^N q(zi = k)}$$
$$\sigma_c^2 = \frac{\sum_{i=1}^N (xi - Mc)^2 p(zi = k)}{\sum_{i=1}^N p(zi = k)}$$
$$\pi_c = \frac{\sum_{i=1}^N p(zi = k)}{N}$$

След като имаме уравненията, ние просто ги прилагаме по време на стъпката на максимизиране. Тоест, ние добавяме числата във всяко уравнение, за да определим най-добрата средна стойност, ковариация, тегло и след това задаваме гаусовите параметри съответно.

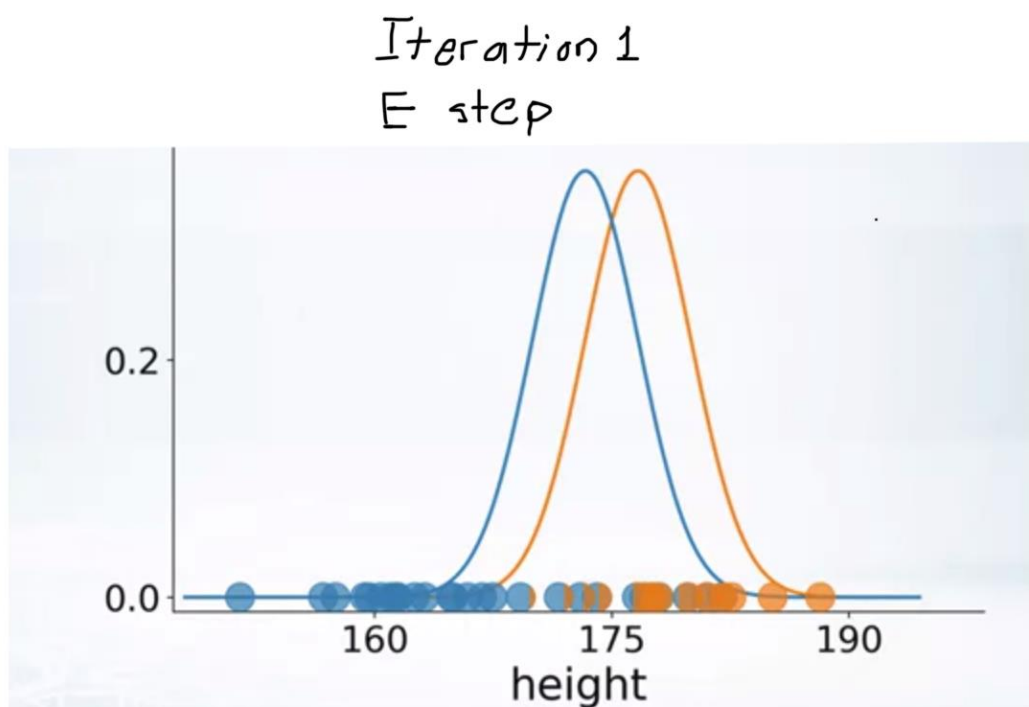
Нека да разгледаме математиката в действие. Първоначално не знаем кои точки са свързани с кое разпределение.



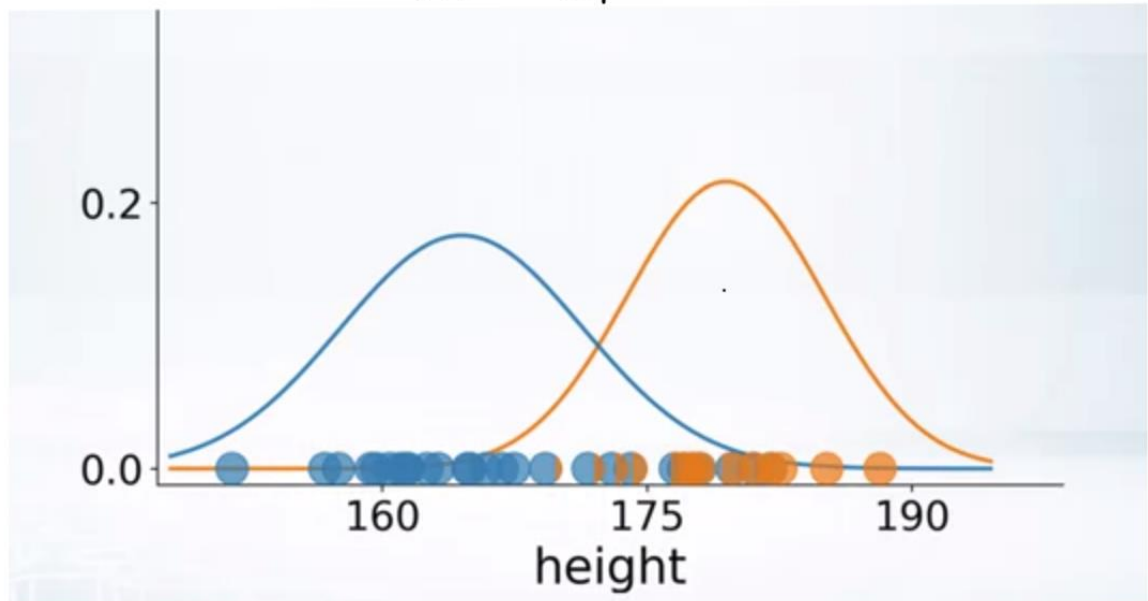
Започваме с К Гауси (в този случай $K=2$) с произволна средна стойност, дисперсия и тегло.



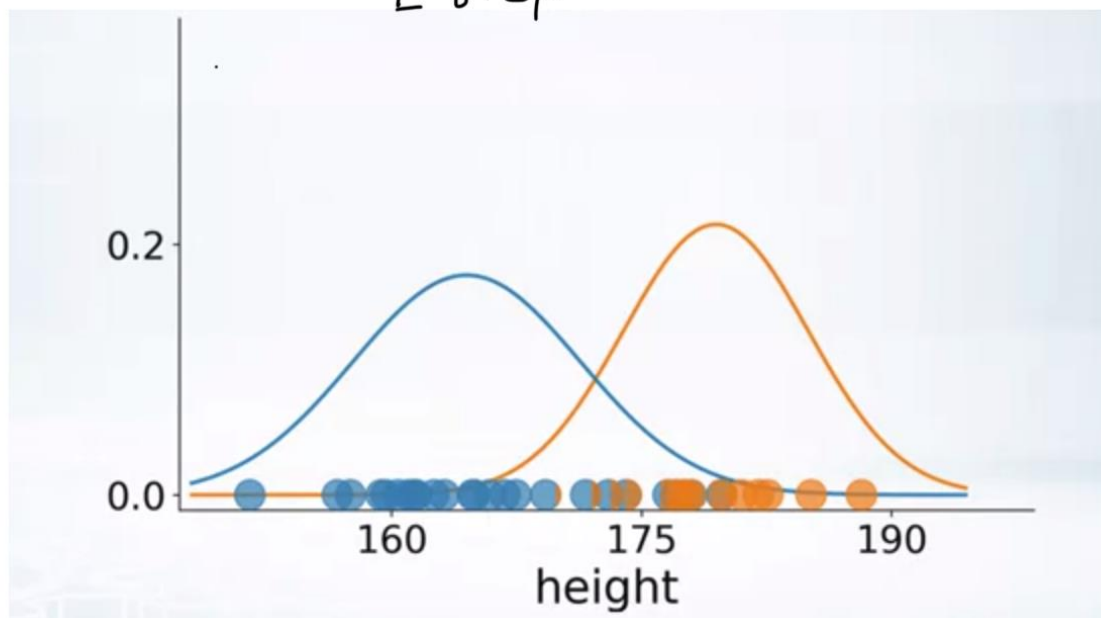
След това повтаряме стъпките за очакване и максимизиране, докато не стигнем до там да имаме много малка или никаква промяна тита (θ).



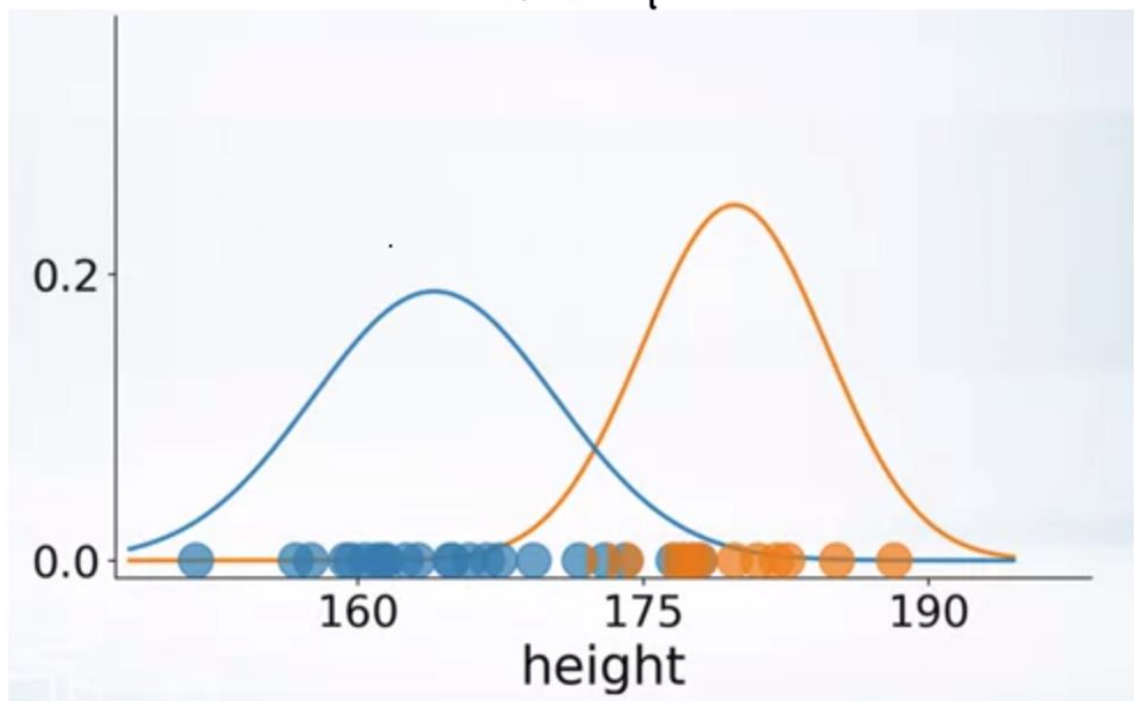
Iteration 1
M step



Iteration 2
E step



Iteration 2
M step



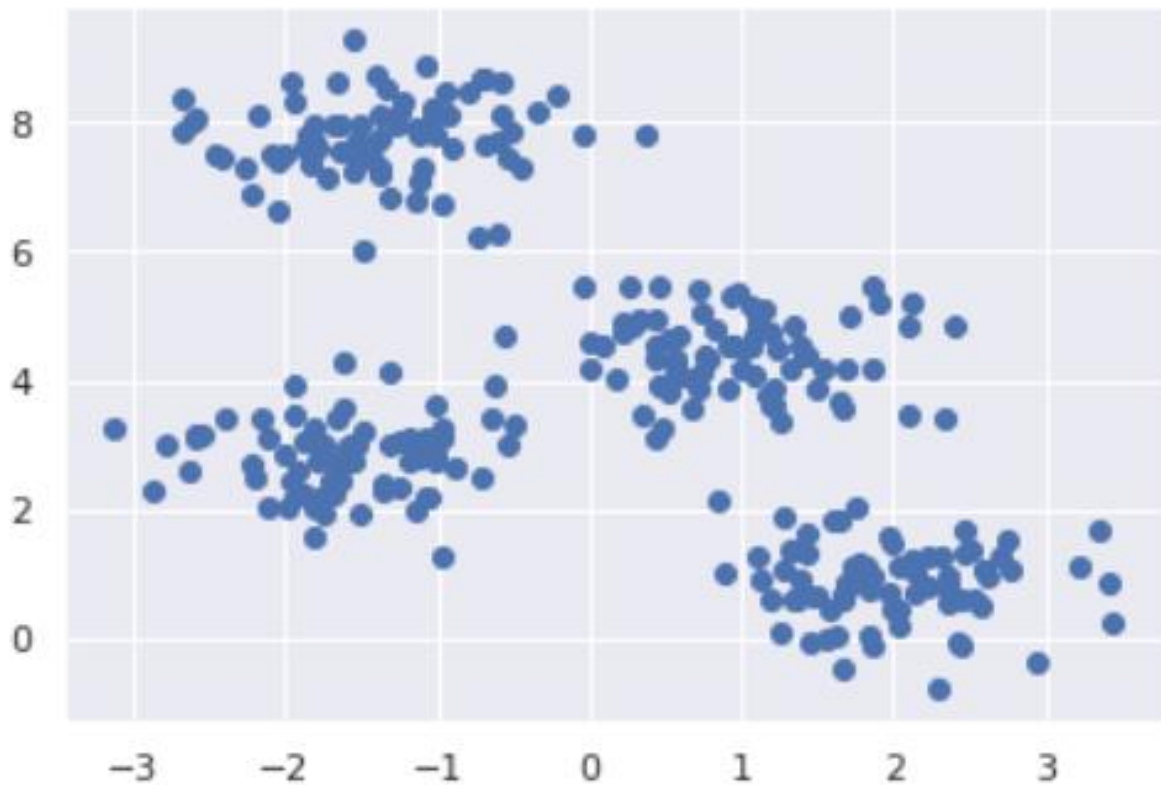
Струва си да се отбележи, че алгоритъмът е податлив на локални максимуми.

Сега, след като вече разбираме как работят моделите на Гаус, нека да разгледаме как можем да ги приложим. Първото нещо, с което започваме е да импортираме необходимите библиотеки.

```
import numpy as np
from sklearn.datasets.samples_generator import make_blobs
from sklearn.mixture import GaussianMixture
from matplotlib import pyplot as plt
import seaborn as sns
sns.set()
```

Генерираме произволни 4-и клъстера.


```
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
plt.scatter(X[:,0], X[:,1])
```



Оптималният брой на кълстерите може да се намери чрез някой готови алгоритми като например метода на лактя.

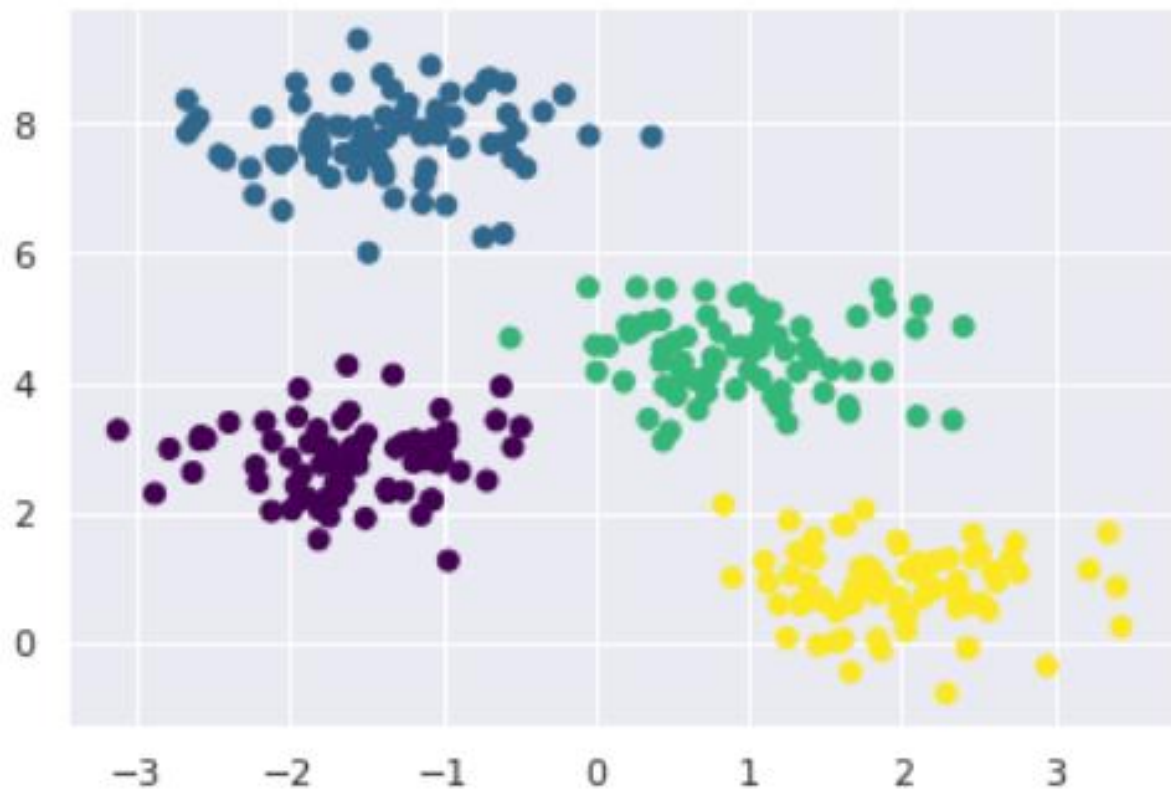
Следващата стъпка е да обучим нашия модел, използвайки оптимален брой кълстери (в случая 4).

```
gmm = GaussianMixture(n_components=4)
gmm.fit(X)
```

Използваме метода за прогнозиране, за да получим списък от точки и съответните им групи.

```
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis');
```

И резултата от клъстеризацията с GMMs метода е следният:



VI. Първи опит за клъстеризация

Библиотеки

Първото нещо, с което започваме е инсталирането и вмъкването на необходимите библиотеки, чийто библиотечни функции са необходими за програмната реализация на клъстериращия алгоритъм, който ще бъде използван за клъстерирането на данните, обект на нашето изследване.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture
import warnings
warnings.filterwarnings("ignore")
```

Първата импортирана библиотека е Pandas. Тя е мощна и широко използвана библиотека за обработка и анализ на данни в Python, която предоставя удобни структури и функции за работа с таблични данни, като например:

DataFrame: Основната структура на данни в Pandas, която представлява двуизмерна таблица с етикетирани редове и колони. DataFrame е много гъвкава и позволява лесно манипулиране, филтриране и обработка на данни.

Series: Едномерна структура от данни, която може да бъде разглеждана като колона в таблица. Series има индекс, който позволява лесно достъпване до стойности.

Основните функции на Pandas включват създаване на DataFrame от речник, четене на данни от csv файл, филтриране на данни, операции върху колони, групиране на данни и обработка на липсващи стойности.

Следващата използвана библиотека е numpy. Библиотеката NumPy (Numerical Python) е една от най-основните и широко използвани библиотеки в Python за научни изчисления. Тя предоставя мощни инструменти за работа с големи масиви от данни и множество математически функции за бързи операции върху тези масиви.

Основни характеристики на NumPy:

- **N-dimensional array object (ndarray):** Основната структура на данни в NumPy е n-мерният масив, който позволява лесно манипулиране на големи набори от данни. Масивите в NumPy могат да бъдат едномерни (вектори), двумерни (матрици) или с по-висока размерност.
- **Бързи операции върху масиви:** NumPy е оптимизиран за извършване на векторизирани операции върху масиви, което го прави много по-бърз от стандартните Python списъци при работа с големи данни.
- **Математически функции:** NumPy предоставя богата библиотека от математически функции за извършване на операции като линейна алгебра, статистика, тригонометрия и други.

- Интеграция с други библиотеки: NumPy е основата за много други научни и машинно обучителни библиотеки в Python, като SciPy, Pandas, Matplotlib и Scikit-learn.

Основно библиотеката NumPy се използва за създаване на масиви, основни операции върху масиви, линейна алгебра, индексване и слайсинг, както и работа с многомерни масиви.

NumPy е изключително важна библиотека за всеки, който работи с научни изчисления или анализ на данни в Python. Тя предоставя основата за много други библиотеки и инструменти, като същевременно предлага мощни и гъвкави възможности за работа с масиви и математически операции.

Друга библиотека, която ни е необходима е Matplotlib. Matplotlib е библиотека за създаване на статични, анимационни и интерактивни визуализации на данни в Python. Тя е една от най-широко използваните библиотеки за визуализация на данни в научните изчисления и анализа на данни. Matplotlib позволява лесно създаване на различни видове графики, включително линейни графики, хистограми, scatter графики, бар графики, пай графики и много други.

Основните характеристики на Matplotlib са:

- Гъвкавост: Matplotlib предлага голямо разнообразие от графични типове и стилове. Можете да създавате прости линейни графики, както и сложни 3D визуализации.
- Интерфейс подобен на MATLAB: Matplotlib е вдъхновен от MATLAB и предоставя сходен интерфейс, което улеснява преминаването от MATLAB към Python.
- Интерактивност: Визуализациите, създадени с Matplotlib, могат да бъдат интерактивни, което позволява на потребителите да увеличават, намаляват и манипулират графиките в реално време.
- Интеграция: Matplotlib работи добре с други научни и машинно обучителни библиотеки като NumPy, Pandas и SciPy.

Основни компоненти на Matplotlib:

- `pyplot`: Подмодул на Matplotlib, който предоставя функции за създаване и управление на графики. Той е предназначен да бъде лесен за използване и е вдъхновен от MATLAB.
- Фигури и оси: В Matplotlib графиките се създават във фигури (Figure), които съдържат една или повече оси (Axes). Фигурите са основните контейнери за всички визуални елементи на графиката.

Основни функции, които предоставя подмодула `pyplot` на Matplotlib:

- `plot()`: Създава линейна графика.
- `hist()`: Създава хистограма.
- `scatter()`: Създава scatter графика.
- `bar()`: Създава бар графика.
- `title()`: Добавя заглавие на графиката.
- `xlabel()` и `ylabel()`: Добавят етикети на осите.
- `legend()`: Добавя легенда към графиката.
- `show()`: Показва графиката на екрана.
- `subplots()`: Създава множество подграфики в една фигура.

Matplotlib и неговият подмодул `pyplot` са основни инструменти за визуализация на данни в Python, предлагайки мощни и гъвкави възможности за създаване на различни типове графики и визуализации.

Seaborn е библиотека за визуализация на данни в Python, която е изградена върху Matplotlib и е предназначена да улеснява създаването на привлекателни и информативни статистически графики. Seaborn предлага високо ниво на абстракция и предоставя множество стилове и цветови палитри, които правят визуализациите по-привлекателни и лесни за разбиране.

Основни характеристики на Seaborn:

- Интеграция с Pandas: Seaborn работи добре с Pandas DataFrame обекти, което улеснява визуализирането на таблични данни.

- Статистически графики: Seaborn предоставя функции за създаване на различни видове статистически графики, включително графики на разпределение, регресионни графики, графики на категорийни данни и много други.
- Темизирание и стилове: Seaborn предлага множество вградени стилове и цветови палитри, които помагат за създаването на привлекателни визуализации с минимални усилия.
- Простота: Seaborn улеснява създаването на сложни визуализации с малко количество код.

Основно seaborn се използва за създаване на графика на разпределение (distribution plot), създаване на scatter графика с линейна регресия, създаване на boxplot за категорийни данни и създаване на heatmap.

Основни функции в Seaborn:

- `sns.histplot()`: Създава хистограма с опция за добавяне на Kernel Density Estimate (KDE).
- `sns.kdeplot()`: Създава KDE графика.
- `sns.scatterplot()`: Създава scatter графика.
- `sns.lineplot()`: Създава линейна графика.
- `sns.lmplot()`: Създава scatter графика с линейна регресия.
- `sns.boxplot()`: Създава boxplot.
- `sns.violinplot()`: Създава violin графика.
- `sns.heatmap()`: Създава heatmap.
- `sns.pairplot()`: Създава диаграми за двойки променливи.

Seaborn предоставя мощни и гъвкави инструменти за визуализация на данни, които улесняват създаването на сложни и привлекателни графики. Тази библиотека е особено полезна при работа със статистически данни и е тясно интегрирана с Pandas, което я прави изключително удобна за анализ на данни.

Друго нещо, което е необходимо за конкретното изследване и импортване е класът `GaussianMixture` от подмодула `mixture` на библиотеката `scikit-learn`. `scikit-learn` е

популярна библиотека за машинно обучение, която предлага различни инструменти за моделиране, включително алгоритми за класификация, регресия, клъстеризация и др. Класът GaussianMixture се използва за създаване и обучение на модел на Гаусова смес (Gaussian Mixture Model, GMM). Гаусовата смес е вероятностен модел, който предполага, че данните са генерирани от смес от няколко гаусови разпределения с неизвестни параметри.

Основни параметри и методи на GaussianMixture:

Параметри:

- `n_components`: Броят на гаусовите компоненти в сместа.
- `covariance_type`: Типът на ковариационната матрица (например 'full', 'tied', 'diag', 'spherical').
- `max_iter`: Максималният брой итерации за алгоритъма за максимално правдоподобие.
- `tol`: Толеранс за критерия за сходимост.

Методи:

- `fit(X)`: Обучава модела на базата на данните X.
- `predict(X)`: Предсказва клъстера за всеки образец в X.
- `predict_proba(X)`: Оценява вероятностите за принадлежност на всеки образец в X към всеки клъстер.
- `score_samples(X)`: Оценява логаритмичното-правдоподобие на всеки образец в X.

С GaussianMixture в scikit-learn могат лесно да моделират сложни данни, които не следват стандартни разпределения, и да се извършва клъстеризация въз основа на вероятностни методи.

Командата `import warnings` импортира стандартния модул `warnings` в Python, който позволява работа с предупреждения (`warnings`). Предупрежденията са съобщения, които сигнализират за потенциални проблеми или неблагоприятни условия в кода, но които не са достатъчно сериозни, за да предизвикат грешка и спиране на програмата.

Командата `warnings.filterwarnings("ignore")` настройва модула `warnings` да игнорира всички предупреждения. Това означава, че ако в кода се появи предупреждение, то няма да бъде показано на потребителя.

Отваряне на файла, който ще клъстерираме

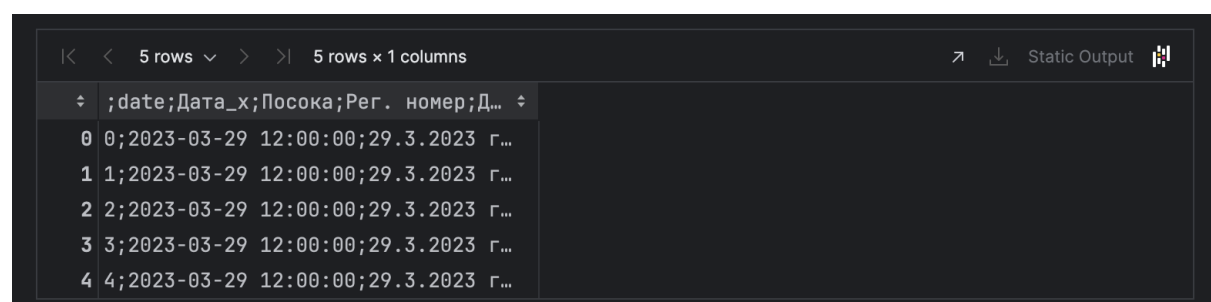
В следващата стъпка от реализацията на клъстеризацията на данните, обект на нашето изследване, отваряме файла с данни посредством Python, като това се случва по следния начин:

```
data_path = "result.csv"
df = pd.read_csv(data_path)
df.head()
```

Чрез променливата `data_path` запазваме пътя, където се намира файла, който искаме да отворим. В случая файла се намира в директорията, където е и проекта, за това не е необходимо да бъде написан пълния път, а това може да се случи директно чрез името на файла. На втория ред, това което правим е да създадем обект от тип `DataFrame`, където съхраняваме `DataFrame` – а, който функцията, четяща `csv` файлове от библиотеката `pandas` ни връща. `DataFrame` обекта от своя страна ни е необходим за по-нататъчната ни работа с файла, който желаем да клъстеризираме. Чрез `df.head()` визуализираме данните, които сме запазили в `DataFrame`-а, за да проверим дали данните, с които ще работим в последствие са такива, каквито очакваме. Метода `head()`, връща първите няколко даннови стойности, които са съхранени в обекта. Алтернатива на този ред е просто да напишем името на променливата за `DataFrame`, като това би довело до визуализация на всички данни, които се съхраняват в `DataFrame`-а.

Коригиране на файла и данновите стойности

След изпълнение на тези няколко реда, резултата който се визуализира е следният:



	;date;Дата_х;Посока;Рег. номер;Д...
0	0;2023-03-29 12:00:00;29.3.2023 Г...
1	1;2023-03-29 12:00:00;29.3.2023 Г...
2	2;2023-03-29 12:00:00;29.3.2023 Г...
3	3;2023-03-29 12:00:00;29.3.2023 Г...
4	4;2023-03-29 12:00:00;29.3.2023 Г...

Така стигаме до извода, че данните не са коректно съхранени във файла, с който разполагаме, така че да бъдат правилно запазени в DataFrame. След по-подробно проучване за изискванията относно четене на CSV файл, посредством библиотеката Pandas и последващото съхраняване в DataFrame, стигаме до извода, че проблема произлиза от факта, че разделянето на данните, във файла, с който разполагаме става посредством „;“, а правилният подход за съхранение на данни, в CSV файл, който в последствие, ще бъде използван, за да бъде четен посредством Python и неговите библиотеки, които са ни предоставени е чрез „;“. Затова следващото нещо, което правим е преработка на CSV файла, като всяка „;“, бива сменена с „;“. Това се случва по следния начин:

```
input_file = 'result.csv'
output_file = 'result_comma.csv'

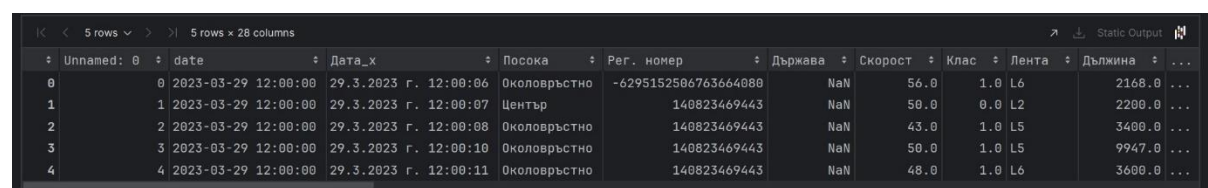
with open(input_file, 'r', encoding='utf-8') as f_in, open(output_file,
'w', encoding='utf-8') as f_out:
    for line in f_in:
        updated_line = line.replace(';', ',')
        f_out.write(updated_line)
```

Това, което се случва е запазване в две променливи на стринговете – име на входния файл (файла, който желаем да преработим) и име на изходния файл, а именно, файла, в който желаем да запазим актуализирания файл. Така се преминава през целия csv файл, всяка „;“ бива заменена с „;“ и новият актуализиран файл се съхранява, така че да разполагаме с него при бъдеща необходимост.

След като вече сме поправили файла, правим отново проверка, дали всичко е наред с него.

```
df = pd.read_csv(output_file)
df.head()
```

Резултатът е следният:



	Unnamed: 0	date	Дата_х	Посока	Рег. номер	Държава	Скорост	Клас	Лента	Дължина	...
0	0	2023-03-29 12:00:00	29.3.2023 г. 12:00:00	Околовръстно	-6295152506763664080	NaN	56.0	1.0	L6	2168.0	...
1	1	2023-03-29 12:00:00	29.3.2023 г. 12:00:07	Център	140823469443	NaN	50.0	0.0	L2	2200.0	...
2	2	2023-03-29 12:00:00	29.3.2023 г. 12:00:08	Околовръстно	140823469443	NaN	43.0	1.0	L5	3400.0	...
3	3	2023-03-29 12:00:00	29.3.2023 г. 12:00:10	Околовръстно	140823469443	NaN	50.0	1.0	L5	9947.0	...
4	4	2023-03-29 12:00:00	29.3.2023 г. 12:00:11	Околовръстно	140823469443	NaN	48.0	1.0	L6	3600.0	...

Резултатът вече напълно удовлетворява очакванията ни и можем да продължим напред с клъстеризацията.

Нещо, което се забелязва обаче е, че при направените измервания има доста невалидни стойности (NaN), които биха попречили в следствие за извършване на клъстеризацията. Затова преди да продължим нататък, това което трябва да се направи е тези стойности или да бъдат премахнати, или да бъдат заменени с валидни такива. Затова подхода, който използваме е да бъдат заменени всички NaN стойности с най-често срещаната стойност в съответната категория, като това обаче би могло да навреди по някакъв начин на изследваните данни, тъй като добавяйки най-често срещаната стойност в дадена категория на местата на невалидните стойности, ще създаде стойности, които ще са част от дадени клъстери, а това реално не би било така, ако тези стойности не бъдат насилствено сменяни по този начин, като зависи от конкретния анализ, който правим това би било или не проблем.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most_frequent')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
df_imputed.to_csv('imputed_file.csv', index=False)
imputed_df = pd.read_csv('imputed_file.csv')
imputed_df.head()
```

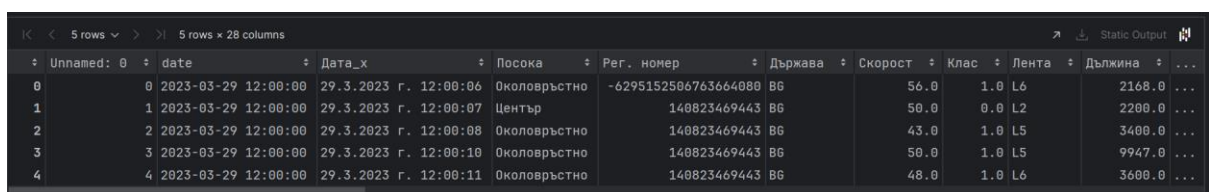
Тези редове код извършват обработка на данни чрез импутиране на липсващи стойности в един DataFrame. Ето какво прави всеки ред:

1. `'from sklearn.impute import SimpleImputer'`: Импортира класа `'SimpleImputer'` от модула `'sklearn.impute'`, който се използва за импутиране (попълване) на липсващи стойности в данните.
2. `'imputer = SimpleImputer(strategy='most_frequent')'`: Създава инстанция на `'SimpleImputer'`, като задава стратегия за импутиране на `"most_frequent"`, което означава, че липсващите стойности ще бъдат заменени с най-често срещаната стойност за съответния атрибут.
3. `'df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)'`: Прилага импутирането върху DataFrame `'df'`, като замества липсващите стойности с най-често срещаните стойности за всяка колона. Резултатът е нов DataFrame `'df_imputed'`, който съдържа импутираните данни.
4. `'df_imputed.to_csv('imputed_file.csv', index=False)'`: Запазва импутирания DataFrame `'df_imputed'` във файл с име `"imputed_file.csv"`, като не включва индекса в резултатния CSV файл.

5. ``imputed_df = pd.read_csv('imputed_file.csv')``: Зарежда данните от "imputed_file.csv" обратно в DataFrame с име ``imputed_df``.

6. ``imputed_df.head()``: Показва първите няколко реда от DataFrame ``imputed_df``, за да се потвърди, че данните са импутирани и са заредени коректно.

Резултата е следният:



	Unnamed: 0	date	Дата_х	Посока	Рег. номер	Държава	Скорост	Клас	Лента	Дължина	...
0	0	2023-03-29 12:00:00	29.3.2023 г. 12:00:06	Околовръстно	-6295152506763664080	B6	56.0	1.0	L6	2168.0	...
1	1	2023-03-29 12:00:00	29.3.2023 г. 12:00:07	Център	140823469443	B6	50.0	0.0	L2	2200.0	...
2	2	2023-03-29 12:00:00	29.3.2023 г. 12:00:08	Околовръстно	140823469443	B6	43.0	1.0	L5	3400.0	...
3	3	2023-03-29 12:00:00	29.3.2023 г. 12:00:10	Околовръстно	140823469443	B6	50.0	1.0	L5	9947.0	...
4	4	2023-03-29 12:00:00	29.3.2023 г. 12:00:11	Околовръстно	140823469443	B6	48.0	1.0	L6	3600.0	...

С просто око се вижда, че резултатите за държава, които до преди редакцията, бяха NaN, вече са заменени от най – често срещаната стойност за категорията – „BG“.

Избор на категории за клъстеризация

Следващата стъпка от първия ни опит за клъстеризация на данните е определяне на категориите по които ще създаваме бъдещите клъстери. Тъй като данните, с които работим са от един месец, а повечето измервания представляват метеорологичните условия, то очакванията са да няма някаква зависимост, която може да бъде разгледана, а напротив – повечето измервания да бъдат прекалено близки едно спрямо друго, а това би довело до клъстери, по които трудно би се направил някакъв анализ и съответно извод. Затова категориите, към които се насочваме са изцяло свързани с измерванията направени за превозните средства. При първия опит се спираме на колоните със „Скорост“ и „Дължина“, тъй като най-логично звучи при превозни средства с по-голяма дължина, скоростта на движение да бъде по-ниска. Очакванията са да намерим някаква подобна зависимост и да видим дали тя ще се потвърди.

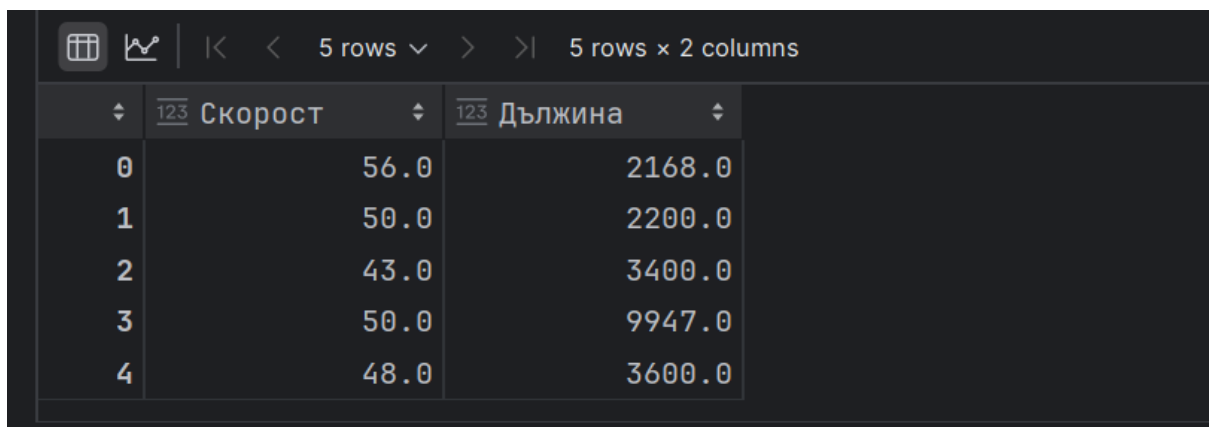
Начинът, по който оставяме само избраните колони е следният:

```
selected_columns = ['Скорост', 'Дължина']
data_selected = imputed_df[selected_columns]
data_selected.to_csv('selected_data.csv', index=False)
data_selected.head()
```

Тези редове код извличат само определени колони от DataFrame `imputed_df` и след това ги запазват в нов CSV файл. Ето какво правят ред по ред:

1. `selected_columns = ['Скорост', 'Дължина']`: Създава списък от имена на колони, които искаме да изберем от DataFrame `imputed_df`. В този случай се избират колоните 'Скорост' и 'Дължина'.
2. `data_selected = imputed_df[selected_columns]`: Избира само колоните, указани в списъка `selected_columns`, от DataFrame `imputed_df`. Резултатът е нов DataFrame `data_selected`, който съдържа само тези избрани колони.
3. `data_selected.to_csv('selected_data.csv', index=False)`: Запазва данните от DataFrame `data_selected` във файл с име "selected_data.csv", като не включва индекса в резултатния CSV файл.
4. `data_selected.head()`: Показва първите няколко реда от DataFrame `data_selected`, за да се потвърди, че са избраните колони и данните са запазени коректно.

Резултата е следният:



	Скорост	Дължина
0	56.0	2168.0
1	50.0	2200.0
2	43.0	3400.0
3	50.0	9947.0
4	48.0	3600.0

Стандартизиране на данните

След като сме оставили само категориите, по които сме решили да извършваме клъстеризацията, следващата стъпка е стандартизирането на данните. Като данните се стандартизират с цел подготовка за анализ или обучение на модели. Ето няколко причини за стандартизирането на данни:

1. Улеснява сравненията: Стандартизирането преобразува данните, така че те имат средно значение 0 и стандартно отклонение 1. Това улеснява сравнението между различните характеристики или между различни образци от данни.
2. Подобрява производителността на моделите: Много алгоритми за машинно самообучение, като например методите за градиентно спускане, работят по-ефективно, ако данните са стандартизирани. Това може да доведе до по-бърза сходимост и по-добри резултати на модела.
3. Предотвратява проблеми с мащабирането на характеристиките: Когато характеристиките имат различни мащаби (например една е в диапазона от 0 до 1, а друга е в диапазона от 1000 до 10000), това може да доведе до проблеми при обучението на модела. Стандартизирането премахва този проблем, като нормализира мащабите на характеристиките.
4. Подобрява интерпретацията: Когато данните са стандартизирани, коефициентите на модела са по-лесни за интерпретация. Те показват как се променя целевата променлива при промяна на едно стандартизирано отклонение в съответната характеристика.
5. Подобрява резултатите на някои алгоритми: Някои алгоритми, като например методите, базирани на разстояния (например k-means клъстеризация), могат да бъдат силно засегнати от разликите в мащаба на характеристиките. Стандартизирането може да помогне да се намали този проблем.

При първият опит за стандартизиране на данните, използваме следния метод:

```
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

numeric_features = data_selected.select_dtypes(include=['int'],
```

```
'float'])).columns
categorical_features =
data_selected.select_dtypes(include=['object']).columns
encoder = LabelEncoder()

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features), # Нормализиране на
        # числовите данни
        ('cat', encoder, categorical_features) # Кодиране на маркери за
        # категоричните данни
    ],
    remainder='passthrough' # Позволява останалите колони да останат без
    # промяна
)

X = data_selected.values
X_preprocessed = preprocessor.fit_transform(X)
```

Този метод обаче се оказва неподходящ поради факта, че използваме данни посредством DataFrame, което обаче се оказва неподходящо за съответния алгоритъм. Затова извършваме стандартизацията посредством Z - стандартизация:

```
means = data_selected.mean()
stds = data_selected.std()

# Z-стандартизация на данните
z_standardized_data = (data_selected - means) / stds
z_standardized_data
```

`means = data_selected.mean()`: Изчислява средните стойности за всяка колона от DataFrame `data_selected` и ги записва в променливата `means`.

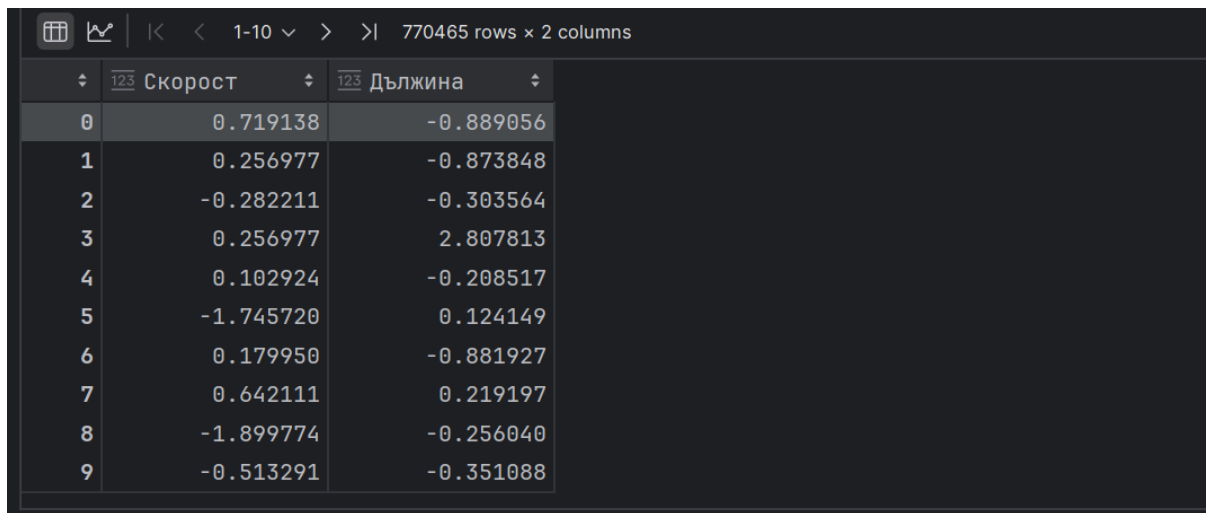
`stds = data_selected.std()`: Изчислява стандартното отклонение за всяка колона от DataFrame `data_selected` и ги записва в променливата `stds`.

`z_standardized_data = (data_selected - means) / stds`: Извършва Z-стандартизация на данните в DataFrame `data_selected`, като от всяка стойност се изважда средната стойност за тази колона и след това се разделя на стандартното отклонение за тази колона. Резултатът е нов DataFrame `z_standardized_data`, който съдържа Z-стандартизираните данни.

`z_standardized_data`: Показва DataFrame `z_standardized_data`, който съдържа Z-стандартизираните данни.

За целите на Z-стандартизацията, когато прилагаме $(data_selected - means) / stds$, всяка стойност в DataFrame `data_selected` се замества със стойност, която е средно отклонение далеч от средната стойност за тази колона. Това нормализира разпределението на данните около средната им стойност и ги прави подходящи за анализ или обучение на модели, които изискват стандартизирани данни.

След стандартизацията данните изглеждат по следния начин:



	123 Скорост	123 Дължина
0	0.719138	-0.889056
1	0.256977	-0.873848
2	-0.282211	-0.303564
3	0.256977	2.807813
4	0.102924	-0.208517
5	-1.745720	0.124149
6	0.179950	-0.881927
7	0.642111	0.219197
8	-1.899774	-0.256040
9	-0.513291	-0.351088

Клъстеризация посредством метода GMM (Gaussian Mixture Model)

След като данните, с които работим вече са стандартизирани можем да преминем към клъстеризацията.

```
n_clusters = 3
gmm = GaussianMixture(n_components=n_clusters)
gmm.fit(z_standardized_data)
labels = gmm.predict(z_standardized_data)
```

Тези редове код се отнасят към клъстеризацията на данните, използвайки модела на смесени гаусови модели (Gaussian Mixture Model - GMM). Ето какво правят тези редове:

1. `n_clusters = 3`: Задава броя на клъстерите, които се опитваме да идентифицираме в данните. В този случай се опитваме да идентифицираме 3 клъстера.

2. ``gmm = GaussianMixture(n_components=n_clusters)``: Създава инстанция на модела на смесени гаусови модели (GMM) със зададения брой компоненти, равен на броя на клъстерите.

3. ``gmm.fit(z_standardized_data)``: Обучава модела на GMM върху стандартизираните данни (``z_standardized_data``), като се опитва да моделира разпределението на данните като смеска от гаусови разпределения.

4. ``labels = gmm.predict(z_standardized_data)``: Прогнозира клъстера, към който всяка точка от данните принадлежи, като използва обученния модел на GMM върху стандартизираните данни. Резултатът е масив от маркировки (labels), като всяка маркировка указва клъстера, към който принадлежи съответната точка от данните.

Визуализация

След като клъстеризацията вече е факт, следващата стъпка към която можем да преминем преди да започнем да анализираме данните е визуализацията.

Визуализацията на клъстерите е важна по няколко основни причини:

1. Визуално разбиране на данните: Визуализацията предоставя интуитивно разбиране за структурата на данните и връзките между тях. Това е особено полезно при анализ на данни без надзор (unsupervised learning), където няма ясно дефинирани целеви променливи.

2. Проверка на резултатите от клъстеризацията: Визуализацията позволява на анализатора да провери резултатите от клъстеризацията и да оцени дали алгоритъмът за клъстеризация е успешен в идентифицирането на различните групи в данните.

3. Идентифициране на взаимосвързаността между данните: Често визуализацията може да помогне за идентифициране на взаимосвързаността или шаблоните между клъстерите. Това може да даде по-добро разбиране за природата на различните групи в данните.

4. Комуникация на резултатите: Визуализацията прави възможно по-лесното комуникиране на резултатите от анализа на данни както на технически, така и на

неспециалистични аудитории. Графичното представяне на резултатите прави тяхната интерпретация по-достъпна и разбираема.

5. Подкрепа за вземане на решения: Визуализацията може да подпомогне в процеса на вземане на решения, като предостави визуално представяне на данните и взаимоотношенията между тях, което може да бъде използвано за вземане на информирани решения.

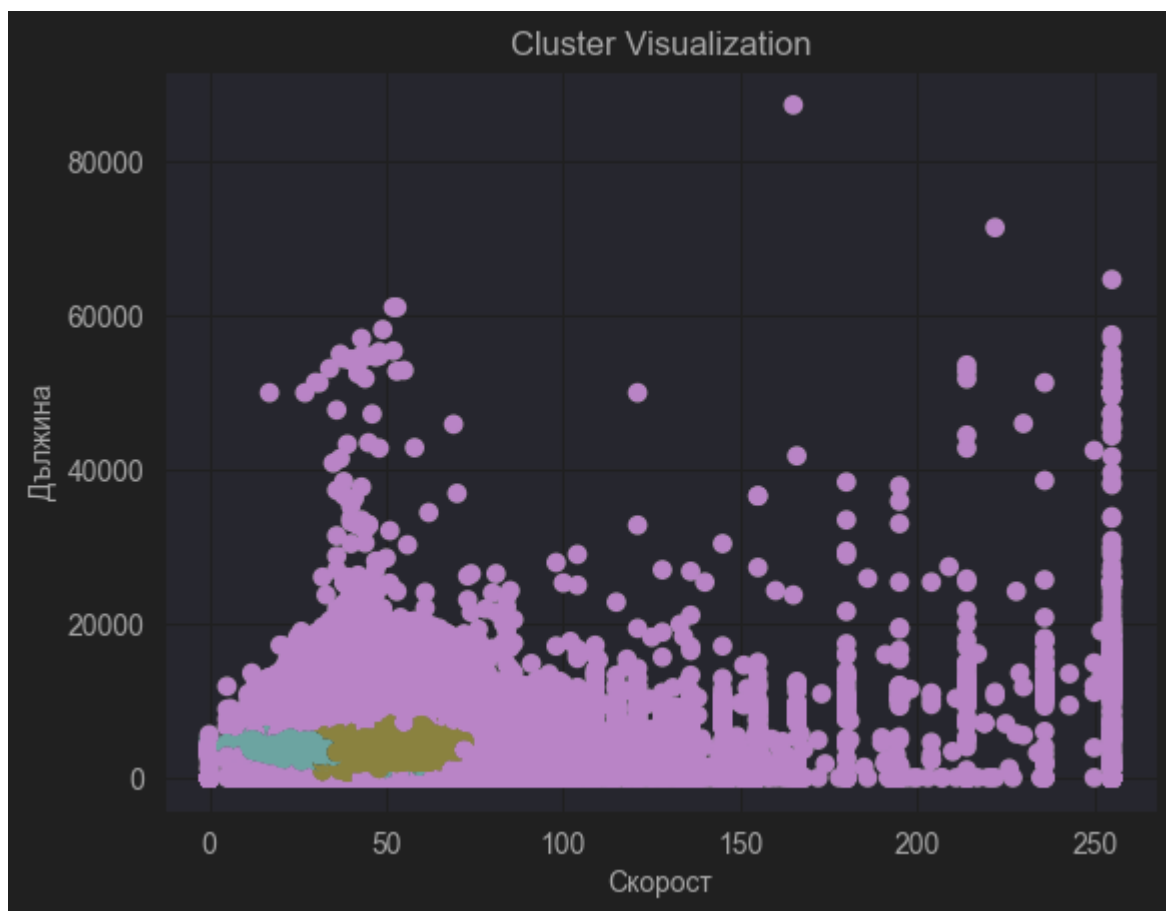
Програмната реализация на визуализацията е следната:

```
import matplotlib.pyplot as plt
plt.scatter(data_selected['Скорост'], data_selected['Дължина'], c=labels,
            cmap='viridis')
plt.xlabel('Скорост')
plt.ylabel('Дължина')
plt.title('Cluster Visualization')
plt.show()
```

Този код визуализира клъстерите в два измерения върху диаграма за разсейване (scatter plot). Ето какво прави всяка част от кода:

1. `import matplotlib.pyplot as plt`: Импортира модула `matplotlib.pyplot`, който се използва за визуализация на данни в Python.
2. `plt.scatter(data_selected['Скорост'], data_selected['Дължина'], c=labels, cmap='viridis')`: Създава диаграма за разсейване, където по оста x се показва стойността на колоната 'Скорост', по оста y се показва стойността на колоната 'Дължина', а цветът на всяка точка се определя от масива `labels`, който съдържа маркировките на клъстерите. Цветовата карта (colormap) е зададена да бъде 'viridis', която е една от вградените цветови карти в matplotlib.
3. `plt.xlabel('Скорост')`: Задава надпис на x-оста на диаграмата, който е 'Скорост'.
4. `plt.ylabel('Дължина')`: Задава надпис на y-оста на диаграмата, който е 'Дължина'.
5. `plt.title('Cluster Visualization')`: Задава заглавие на диаграмата, което е 'Cluster Visualization'.

6. `plt.show()`: Показва диаграмата за разсейване на екрана.



Изводите, които могат да се направят по конкретната диаграма са няколко. Първият е, че образуваните клъстери не са плътни и доста от точките, представляващи данни, са на много големи разстояния една от друга, което показва, че няма добре оформени клъстери. Това от своя страна не позволява да бъдат направени конкретни изводи за зависимости между избраните категории. Разглеждайки стойностите на осите може да се направи заключение, че голяма част от измерванията са некоректни (outliners) – до това заключение стигаме, тъй като в графиката виждаме превозни средства с дължина над 80000 метра, което физически е невъзможно да бъде забелязано, друго притеснително измерване, което се забелязва са превозните средства, които се движат със скорост от 250 км/ч и такива, които се движат със скорост под 0-та. При тези данни можем да говорим за грешки в измервателните уреди. Основното заключение от първия опит е, че данните съдържат голям брой некоректни даннови стойности, които трябва да бъдат пречистени, за да говорим за изводи свързани с някакви зависимости между избраните категории.

Програмна реализация

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture
import warnings
warnings.filterwarnings("ignore")

data_path = "result.csv"
df = pd.read_csv(data_path)
df.head()

input_file = 'result.csv'
output_file = 'result_comma.csv'

with open(input_file, 'r', encoding='utf-8') as f_in, open(output_file,
'w', encoding='utf-8') as f_out:
    for line in f_in:
        updated_line = line.replace('; ', ',')
        f_out.write(updated_line)

df2 = pd.read_csv(output_file)
df2.head()

from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most_frequent')
df_imputed = pd.DataFrame(imputer.fit_transform(df2), columns=df2.columns)
df_imputed.to_csv('imputed_file.csv', index=False)
imputed_df = pd.read_csv('imputed_file.csv')
imputed_df.head()

selected_columns = ['Скорост', 'Дължина']
data_selected = imputed_df[selected_columns]
data_selected.to_csv('selected_data.csv', index=False)
data_selected.head()

means = data_selected.mean()
stds = data_selected.std()

# Z-стандартизация на данните
z_standardized_data = (data_selected - means) / stds
z_standardized_data

n_clusters = 3
gmm = GaussianMixture(n_components=n_clusters)
gmm.fit(z_standardized_data)
labels = gmm.predict(z_standardized_data)

import matplotlib.pyplot as plt
plt.scatter(data_selected['Скорост'], data_selected['Дължина'], c=labels,
cmap='viridis')
plt.xlabel('Скорост')
plt.ylabel('Дължина')
plt.title('Cluster Visualization')
plt.show()
```

VII. Втори опит за клъстеризация

При вторият опит за клъстеризация фокусът вече е насочен към това, първо да бъдат прочистени невалидните данни .

Започваме отново с включване на познатите ни библиотеки.

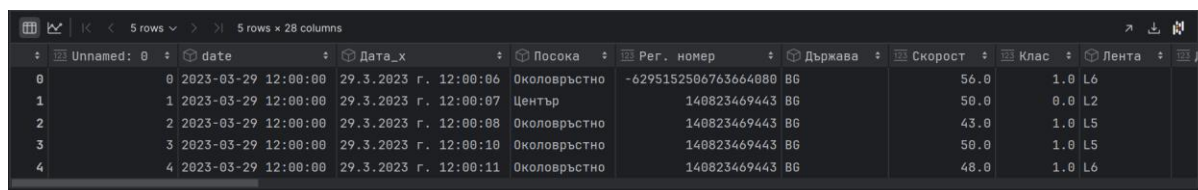
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture

import warnings
warnings.filterwarnings("ignore")
```

След това прочитаме файла, като вече използваме файла, в който „;“ са заменени с „,“, „“ и празните стойности са заменени с най-често срещаната стойност в дадена колона, като с този файл разполагаме от опит №1.

```
df = pd.read_csv('imputed_file.csv')
df.head()
```

При проверката на файла, който четем, получаваме следния резултат:

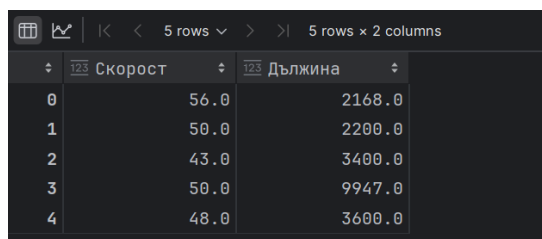


	Unnamed: 0	date	Дата_х	Посока	Рег. номер	Държава	Скорост	Клас	Лента
0	0	2023-03-29 12:00:00	29.3.2023 г. 12:00:06	Околовръстно	-6295152506763664080	B6	56.0	1.0	L6
1	1	2023-03-29 12:00:00	29.3.2023 г. 12:00:07	Център	140823469443	B6	50.0	0.0	L2
2	2	2023-03-29 12:00:00	29.3.2023 г. 12:00:08	Околовръстно	140823469443	B6	43.0	1.0	L5
3	3	2023-03-29 12:00:00	29.3.2023 г. 12:00:10	Околовръстно	140823469443	B6	50.0	1.0	L5
4	4	2023-03-29 12:00:00	29.3.2023 г. 12:00:11	Околовръстно	140823469443	B6	48.0	1.0	L6

Следващата стъпка е да селектираме категориите данни, по които ще извършваме клъстеризацията.

```
selected_columns = ['Скорост', 'Дължина']
data_file = df[selected_columns]
data_file.head()
```

Резултата от изпълнението на тези редове:



	Скорост	Дължина
0	56.0	2168.0
1	50.0	2200.0
2	43.0	3400.0
3	50.0	9947.0
4	48.0	3600.0

Стандартизиране данните като метода за стандартизация, който използваме тук е StandardScaler. `StandardScaler` е един от много трансформатори за скалиране, предоставени от библиотеката за машинно самообучение в Python, scikit-learn. Той се използва за стандартизиране на функциите (features) чрез премахване на средното и скалиране до единично стандартно отклонение.

Ето как работи StandardScaler:

1. Пресмята средната стойност и стандартното отклонение: При инициализирането, `StandardScaler` анализира всички стойности във всеки признак и пресмята тяхната средна стойност и стандартното отклонение.
2. Скалиране на признаците: След това, за всяка стойност във всяка функция, се изважда средната стойност и след това резултатът се разделя на стандартното отклонение. Това пренарежда данните около 0 и ги мащабира, така че стандартното отклонение е 1.
3. Прилагане на преобразуването: Когато единствената функция, `fit_transform`, се извика, `StandardScaler` изчислява средните стойности и стандартното отклонение на данните и след това ги прилага върху данните. Когато вече са били извикани `fit` методи, можете да използвате метода `transform`, за да преобразувате нови данни, използвайки средните стойности и стандартното отклонение, които са били научени по време на обучението.

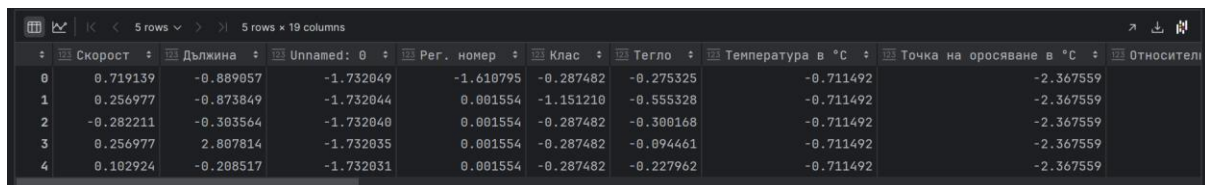
`StandardScaler` е особено полезен, когато работите с алгоритми, които изискват стандартизирани функции, като например методи, базирани на разстояния (например метода на k-средните или метода на опорните вектори), или регресионни методи, като линейната регресия или логистичната регресия.

```
from sklearn.preprocessing import StandardScaler
numeric_columns = df.select_dtypes(include=['number']).columns
scaler = StandardScaler()
data_file[numeric_columns] = scaler.fit_transform(df[numeric_columns])
data_file.head()
```

- from sklearn.preprocessing import StandardScaler: Импортира класа StandardScaler от библиотеката scikit-learn. StandardScaler се използва за

стандартизиране на данните чрез премахване на средното и скалиране до единично стандартно отклонение.

- `numeric_columns = df.select_dtypes(include=['number']).columns`: Избира имената на колоните, които съдържат числови данни. Това става чрез метода `select_dtypes`, който филтрира колоните спрямо техния тип данни и `include=['number']`, който ограничава избора само до числови типове данни.
- `scaler = StandardScaler()`: Създава инстанция на `StandardScaler`, която ще използваме за стандартизиране на данните.
- `data_file[numeric_columns] = scaler.fit_transform(df[numeric_columns])`: Стандартизира данните в числовите колони, зададени от `numeric_columns`, използвайки метода `fit_transform` на `StandardScaler`. Това извлича средните стойности и стандартните отклонения от данните и след това стандартизира данните. Стандартизираните данни се записват обратно в съответните колони на `DataFrame data_file`.
- `data_file.head()`: Показва първите няколко реда от `DataFrame data_file`, след като са били стандартизирани данните. Това ни позволява да видим как изглеждат данните след промяната.



	Скорост	Дължина	Unnamed: 0	Рег. номер	Клас	Тегло	Температура в °C	Точка на оросяване в °C	Относител
0	0.719139	-0.889057	-1.732049	-1.610795	-0.287482	-0.275325	-0.711492	-2.367559	
1	0.256977	-0.873849	-1.732044	0.001554	-1.151210	-0.555328	-0.711492	-2.367559	
2	-0.282211	-0.303564	-1.732040	0.001554	-0.287482	-0.300168	-0.711492	-2.367559	
3	0.256977	2.807814	-1.732035	0.001554	-0.287482	-0.094461	-0.711492	-2.367559	
4	0.102924	-0.208517	-1.732031	0.001554	-0.287482	-0.227962	-0.711492	-2.367559	

Проверка на липсващите стойности

```
missing_values = data_file.isnull().sum()  
print(missing_values)
```

Скорост	0
Дължина	0
Unnamed: 0	0
Рег. номер	0
Клас	0
Тегло	0
Температура в °C	0
Точка на оросяване в °C	0
Относителна влажност в %	0
Интензивност на валеж в mm/h	0
Видимост в m	0
Средна скорост на вятър в m/s	0
Средна посока на вятър в °	0
Темп. на повърхността на настилка в °C	0
Точка на замръзване в °C	0
Височина на водния стълб в µm	0
Темп. под повърхн. на дълб. 5 см в °C	0
Макс. скорост на вятър в m/s	0
Посока на вятъра при макс. скорост в °	0

dtype: int64

От тук на пръв поглед можем да отбележим, че вече във файла няма невалидни стойности. Но връщайки се на графиката от опит №1, виждаме, че има доста даннови единици, които се въртят около 0-та, което също можем да приемем, че е невалидна стойност, тъй като ако превозно средство се движи със скорост 0 км/ч или по-малко, то това няма как да бъде коректно измерване, както и няма как да съществува превозно средство с дължина 0м или такава, по – малко от 0м. Затова следващата стъпка към

която преминаваме е да заменим всички стойности „0“ с „NaN“, така че те също да бъдат разпознати от средата като невалидни стойности.

```
data_file = df.replace(0, np.nan)
```

След замяна на нулвеите стойности, отново правим проверка за броя на невалидните стойности.

```
missing_values = df3.isnull().sum()  
print(missing_values)
```

Unnamed: 0.1	0
Unnamed: 0	1
date	0
Дата_х	0
Посока	0
Рег. номер	0
Държава	0
Скорост	470
Клас	11641
Лента	0
Дължина	7621
Тегло	28540
Станция	0
Дата_у	0
Температура в °C	347
Точка на оросяване в °C	4655
Относителна влажност в %	0
Интензивност на валеж в mm/h	769645
Видимост в m	0
Средна скорост на вятър в m/s	12105
Средна посока на вятър в °	20009

Темп. на повърхността на настилка в °C	0
Точка на замръзване в °C	770465
Състояние на повърхността	0
Предупреждение за състояние на повърхността	0
Височина на водния стълб в µm	718642
Темп. под повърхн. на дълб. 5 см в °C	0
Макс. скорост на вятър в m/s	72
Посока на вятъра при макс. скорост в °	72

dtype: int64

Резултата, който получаваме е доста по-различен от първоначалния, което от своя страна доказва тезата ни, че е имало доста повече невалидни резултати, отколкото сме предполагали при първият ни опит за клъстеризация.

Отново правим проверка на промените:

```
df3.head()
```

	Unnamed: 0.1	Unnamed: 0	date	Дата_х	Посока	Рег. номер	Държава	Скорост	Клас
0	0	NaN	2023-03-29 12:00:00	29.3.2023 г. 12:00:06	Околовръстно	-6295152506763664080	BG	56.0	
1	1	1.0	2023-03-29 12:00:00	29.3.2023 г. 12:00:07	Център	140823469443	BG	50.0	
2	2	2.0	2023-03-29 12:00:00	29.3.2023 г. 12:00:08	Околовръстно	140823469443	BG	43.0	
3	3	3.0	2023-03-29 12:00:00	29.3.2023 г. 12:00:10	Околовръстно	140823469443	BG	50.0	
4	4	4.0	2023-03-29 12:00:00	29.3.2023 г. 12:00:11	Околовръстно	140823469443	BG	48.0	

От тук виждаме, че нулевите стойности успешно вече са заменени с „NaN“.

Следващата стъпка е да изберем колоните, по които ще клъстеризираме:

```
selected_columns = ['Скорост', 'Дължина']
data_selected = df3[selected_columns]
data_selected.head()
```

	Скорост	Дължина
0	56.0	2168.0
1	50.0	2200.0
2	43.0	3400.0
3	50.0	9947.0
4	48.0	3600.0

След това правим една проверка за броя на невалидните стойности:

```
num = data_selected.isnull().sum()
print(num)
```

```
Скорост      470
Дължина      7621
dtype: int64
```

След което отново заменяме невалидните стойности с най-често срещаните за съответните колонии:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most_frequent')
df_imputed = pd.DataFrame(imputer.fit_transform(data_selected),
                           columns=data_selected.columns)
df_imputed.head()
```

	123 Скорост	123 Дължина
0	56.0	2168.0
1	50.0	2200.0
2	43.0	3400.0
3	50.0	9947.0
4	48.0	3600.0

Прави една проверка дали всички невалидни стойности са премахнати:

```
num = df_imputed.isnull().sum()
print(num)
```

```
Скорост      0
Дължина      0
dtype: int64
```

От резултата виждаме, че вече всички невалидни стойности са премахнати, така че можем да преминем към следващата стъпка, а именно изпълняване на метода на

лакътя. Тъй като алгоритъма, който използваме (GMM) изисква предварително задаване на броя на клъстерите, които ще създаде, то за да определим реално колко е оптималният им брой ние ще използваме Метода на лакътя. Методът на лакътя (Elbow Method) е техника за определяне на оптималния брой клъстери при клъстеризацията на данни. Ето как работи:

1. Избор на брой клъстери: Стартира се със зададен диапазон от възможни брой клъстери, който обикновено е в интервала от 1 до предварително зададен максимум.
2. Изчисляване на критерия за клъстеризация: За всеки брой клъстери се извършва клъстеризация на данните. След това се изчислява някакъв критерий, който оценява качеството на клъстеризацията. Този критерий може да бъде сумата на квадратните отклонения на точките от техните центроиди (SSE), коефициентът на Силует (Silhouette Coefficient), инерцията на клъстерите и др.
3. Визуализиране на резултатите: Резултатите от стъпка 2 се визуализират чрез графика, която показва стойностите на критерия за клъстеризация за всеки брой клъстери.
4. Избор на оптимален брой клъстери: Визуализираният график обикновено показва форма, приличаща на "лакът" (elbow). Точката, в която формата на графика се променя, се нарича "лакът". Това е областта, където нарастването на критерия за клъстеризация спира да бъде толкова значимо. Оптималният брой клъстери се избира обикновено в точката, където се намира "лакътят".
5. Прилагане на клъстеризация с избрания брой клъстери: След като е избран оптималният брой клъстери, се извършва клъстеризация на данните с този брой клъстери.

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Примерни данни
X = df_imputed

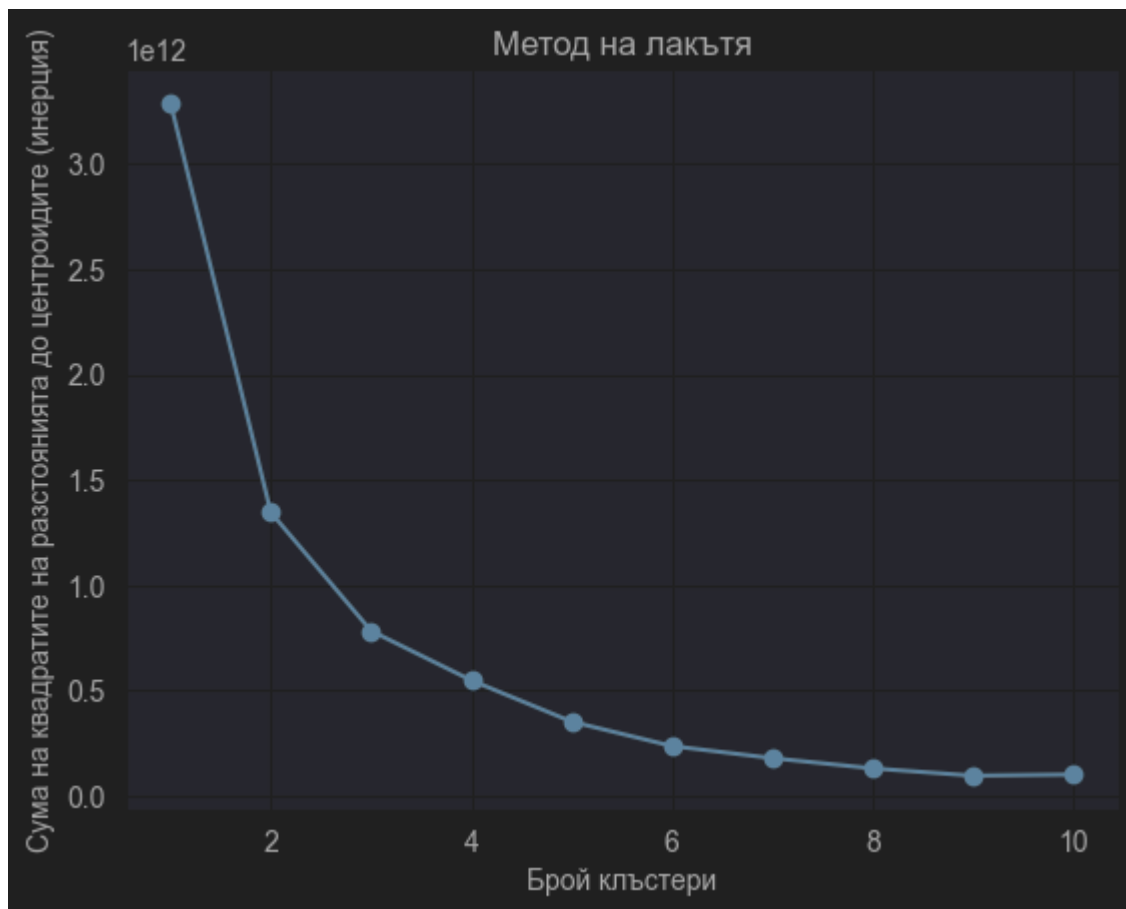
# Създаване на списък, в който ще съхраним стойности на вариацията за всяко k
variations = []

# Ползване k от 1 до 10
for k in range(1, 11):
```

```
# Създаване на k-means модел с k кластера
kmeans = KMeans(n_clusters=k)
kmeans.fit(X)

# Записване на вариацията за текущото k
variations.append(kmeans.inertia_)

# Визуализация на метода на лакътя
plt.plot(range(1, 11), variations, marker='o')
plt.xlabel('Брой клъстери')
plt.ylabel('Сума на квадратите на разстоянията до центроидите (инерция)')
plt.title('Метод на лакътя')
plt.show()
```



Тъй като от графиката се вижда, че разстоянията от 4 клъстера надолу, започват да стават почти еднакви, то приемаме, че оптималният брой на клъстерите е 4 броя.

Преди да преминем обаче към клъстеризацията е необходимо да премахнем данновите стойности, които няма как да бъдат реални – например прекалено големи, или прекалено малки стойности, които се различават от повечето даннови стойности с порядъци – такива стойности се приемат за аномалии и пречат на изводите, които искаме да направим относно някаква зависимост между данните. Метода, който използваме за премахване на аномалиите е Isolation Forest. Isolation Forest е алгоритъм

за откриване на аномалии, който работи на принципа, че аномалиите се различават от нормалните наблюдения по това, че са рядки в многомерни пространства. Този метод се основава на дървовидни структури за разделяне на данните.

Ето как работи Isolation Forest:

1. Избира на случаен признак и случаен праг: Всяко дърво в изолационния процес започва с избирането на случаен признак от данните и случаен праг върху този признак.
2. Разделяне на данните: Пространството се разделя от прага в избрания признак, като се разполагат точки по отношение на този праг.
3. Продължаване на разделянето: Процесът се повтаря, като се избират случайни признаци и прагови стойности, и се продължава с разделянето на пространството на по-малки области.
4. Построяване на дърво: Този процес продължава, докато всеки елемент в набора от данни не бъде разделен в изолирани области, като се създаде дърво с произволна дълбочина.
5. Оценка на аномалиите: Алгоритъмът оценява аномалиите като избира тези наблюдения, които могат да бъдат открити с по-малко разделящи стъпки. Това е основата на оценката за аномалии: нормалните точки ще бъдат по-лесно открити, докато аномалиите, които са по-редки, ще бъдат изолирани по-бързо. В резултат, аномалиите се смятат за тези точки, които изискват по-малко стъпки, за да бъдат отделени от другите точки в пространството.

Isolation Forest има няколко предимства, включително ефективността му за големи данни, устойчивостта си към преобразувания на данните и удобството при обучението му, което не изисква адаптивни параметри.

```
from sklearn.ensemble import IsolationForest

# Създаване на модел за Isolation Forest
model = IsolationForest(contamination=0.1) # contamination е долята на
аномалиите в данните
```

```
# Подготовка на данните X, които съдържат вашите данни
model.fit(X) # Трениране на модела
predictions = model.predict(X) # Предсказване на аномалиите

cleaned_df = df_imputed[predictions == 1] # Избиране на редовете, които не
са маркирани като аномалии

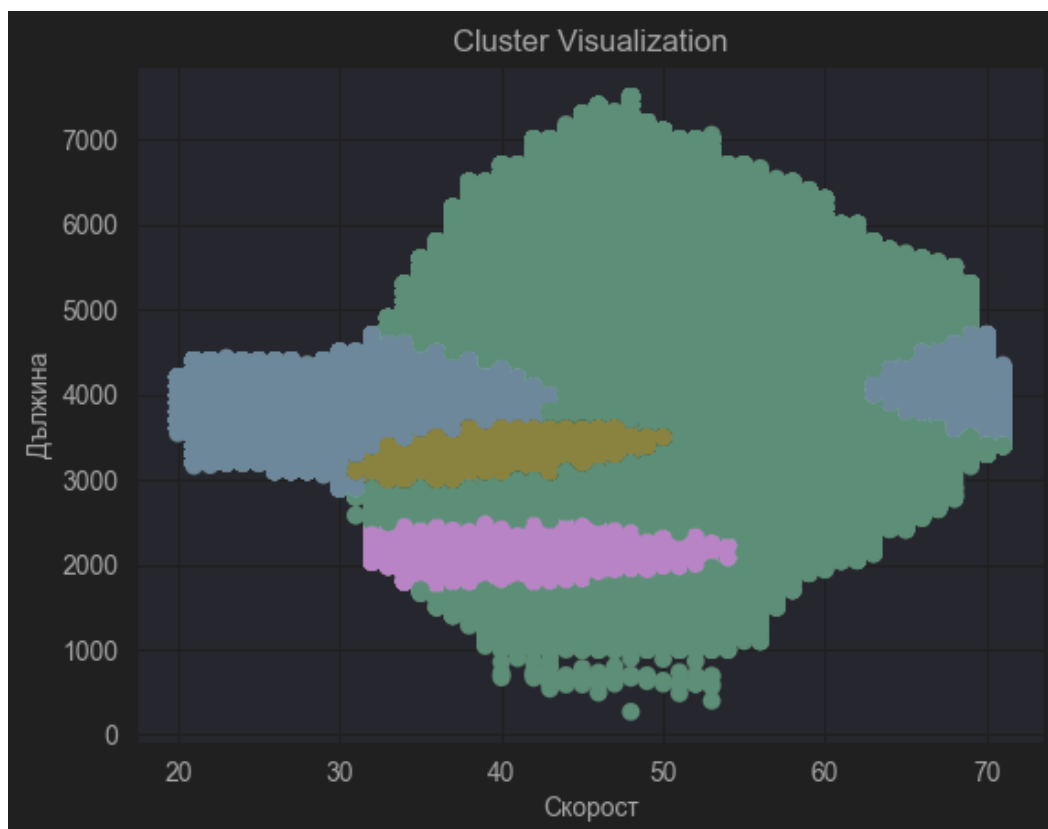
# Запазване на промените в нов CSV файл
cleaned_df.to_csv('cleaned.csv', index=False)
```

След като вече сме премахнали и аномалиите, можем да преминем към клъстеризацията.

```
n_clusters = 4
gmm = GaussianMixture(n_components=n_clusters)
gmm.fit(cleaned_df)
labels = gmm.predict(cleaned_df)
```

След извършване на клъстеризацията е време и за визуализацията:

```
import matplotlib.pyplot as plt
plt.scatter(cleaned_df['Скорост'], cleaned_df['Дължина'], c=labels,
            cmap='viridis')
plt.xlabel('Скорост')
plt.ylabel('Дължина')
plt.title('Cluster Visualization')
plt.show()
```



Изводът, който може да бъде направен след този опит е, че тук вече клъстерите са много по-плътни, така че вече може да се говори за зависимости между данните. Нещо, което не изглежда добре в тази визуализация е това, че единият клъстер се намира в два различни края на графиката, това не би трябвало да е коректна клъстеризация. Този проблем е възможно да произлиза от стандартизацията на данните, и това, че използваните данни за клъстеризация са стандартизирани, преди данните да бъдат разчистени. Друг проблем, който може би води до тази визуализация е, това че невалидните стойности не се премахват, а се заменят с най-често срещаната в колоната, което може би от своя страна също води до някакви аномалии.

VIII. Трети опит за клъстеризация

Подходът, който ще използваме при този опит е почти същия като при опит №2, но тук редът на действията е променен.

1) Добавяме библиотеките

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture

import warnings
warnings.filterwarnings("ignore")
```

2) Прочитаме файла

```
data = 'result_comma.csv'
df = pd.read_csv(data)
df.head()
```

3) Оставяме само колоните, които ще използваме за клъстеризация

```
selected_columns = ['Скорост', 'Дължина']
data_selected = df[selected_columns]
data_selected.head()
```

4) Проверяваме броя на липсващите стойности

```
missing_values = data_selected.isnull().sum()
print(missing_values)
```

```
Скорост    1427
Дължина    1427
dtype: int64
```

Нещо интересно, което се проявява тук е, че има някаква зависимост между невалидните стойности – когато на даден ред измерената стойност за скорост е невалидна, то и тази за дължина също е невалидна.

5) Заменяме нулевите стойности с „NaN“

```
data_selected = data_selected.replace(0, np.nan)  
data_selected.head()
```

6) Отново правим проверка за броя на невалидните стойности

```
missing_values = data_selected.isnull().sum()  
print(missing_values)
```

```
Скорост      1897  
Дължина      9048  
dtype: int64
```

Тук вече зависимостта между невалидните стойности е нарушена.

7) Премахваме всички невалидни стойности

```
data_selected = data_selected.dropna()  
data_selected.head()
```

8) Проверяваме дали успешно сме премахнали всички невалидни стойности

```
missing_values = data_selected.isnull().sum()  
print(missing_values)
```

```
Скорост      0  
Дължина      0  
dtype: int64
```


9) Прилагаме алгоритъма за премахване на аномалии IsolationForest

```
from sklearn.ensemble import IsolationForest

# Създаване на модел за Isolation Forest
model = IsolationForest(random_state= 10,contamination=0.1) #
contamination е долята на аномалиите в данните

# Подготовка на данните X, които съдържат вашите данни
model.fit(data_selected) # Трениране на модела
predictions = model.predict(data_selected) # Предсказване на аномалиите

cleaned_df = data_selected[predictions == 1] # Избиране на редовете, които
не са маркирани като аномалии

# Запазване на промените в нов CSV файл
cleaned_df.to_csv('correct_cleaned.csv', index=False)
```

10) Използваме метода на лакътя за определяне на оптималния брой на клъстерите за клъстеризация

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

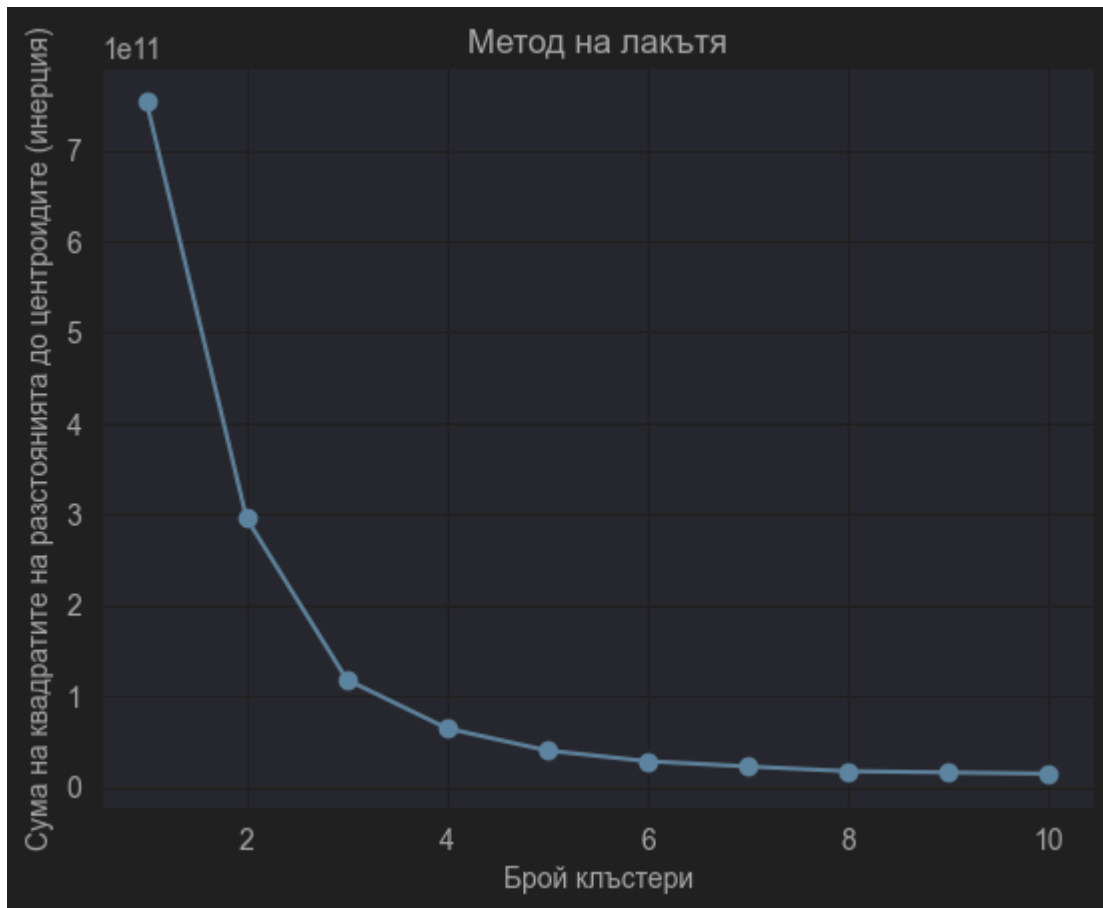
# Примерни данни
# предполагаме, че X е вашият набор от данни
X = cleaned_df

# Създаване на списък, в който ще съхраним стойности на вариацията за
всяко k
variations = []

# Ползваме k от 1 до 10 за пример
for k in range(1, 11):
    # Създаване на k-means модел с k кластера
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)

    # Записване на вариацията за текущото k
    variations.append(kmeans.inertia_)

# Визуализация на метода на лакътя
plt.plot(range(1, 11), variations, marker='o')
plt.xlabel('Брой клъстери')
plt.ylabel('Сума на квадратите на разстоянията до центроидите (инерция)')
plt.title('Метод на лакътя')
plt.show()
```



Прилагайки метода на лакътя, ясно можем да видим, че оптималният брой на клъстерите е 5.

11) Стандартизираме данните

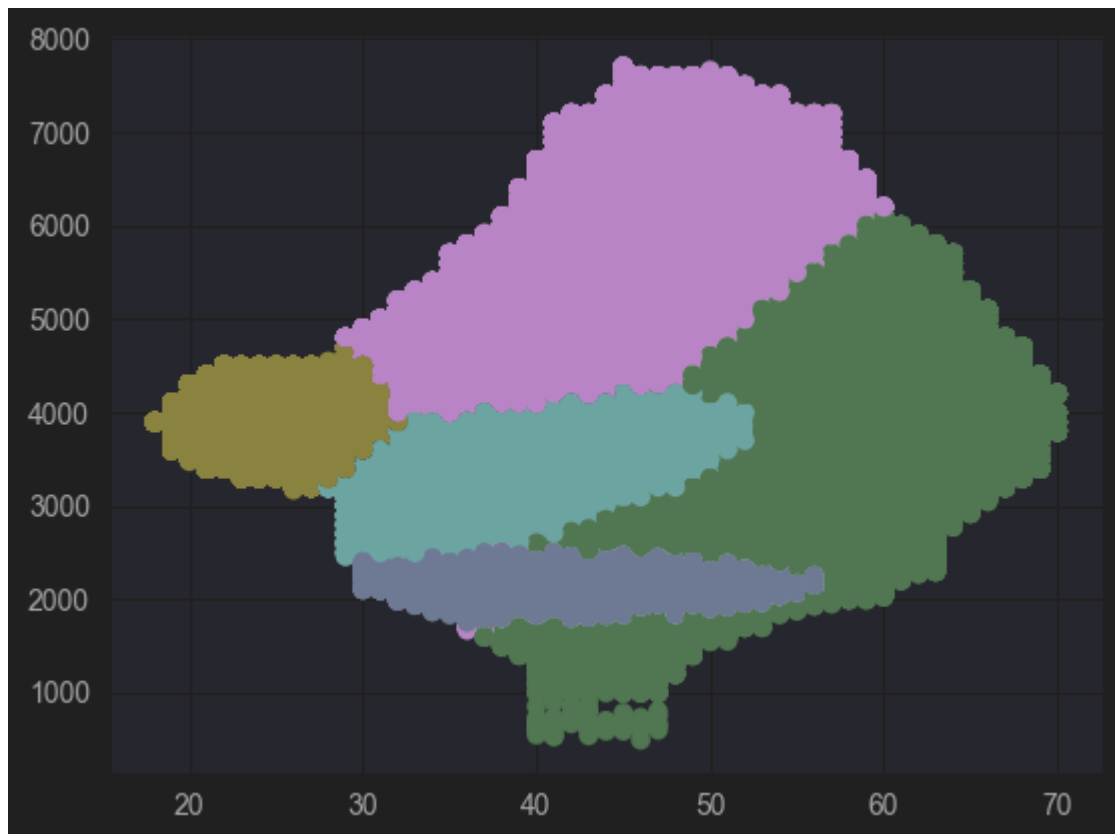
```
from sklearn.preprocessing import StandardScaler
numeric_columns = cleaned_df.select_dtypes(include=['number']).columns
scaler = StandardScaler()
data_standartized = scaler.fit_transform(cleaned_df[numeric_columns])
```

12) Извършваме клъстеризацията

```
n_clusters = 5
gmm = GaussianMixture(n_components=n_clusters)
gmm.fit(data_standartized)
labels = gmm.predict(data_standartized)
```

13) Визуализираме

```
data_standartized=cleaned_df.copy()
data_standartized['Cluster']=labels
plt.scatter(data_standartized['Скорост'],data_standartized['Дължина'],c=
data_standartized['Cluster'],cmap= 'viridis')
plt.show()
```



Тук вече говорим за ясно изразени и добре оформени клъстери и съответно можем да направим изводи за категориите клъстери, които се оформят. Имаме категория с автомобили с дължина между 3 и 5м, които се движат с ниска скорост (до 35км/ч), имаме категория с автомобили с дължина около 2м, които се движат със скорост от 30 до 58км / ч, имаме клъстер за автомобилите, които са около 3м и се движат със скорост от 30 до 50 км / ч. Покриващо очакванията, най-големите клъстери са с автомобили с ширина от 2 до 5м., които се движат със скорост от 40 до 70 км / ч, както и клъстер с автомобили с дължина от 2 до 7 м, движещи се със скорост от 30 до 50 км / ч. Можем да говорим за зависимост от една страна между скоростта и дължината в контекста на това, че по-малките автомобили се движат средно с по-ниска скорост. Също така можем да кажем, че по-голямата част от трафика (независимо от размера на превозното средство) се движи със скорост съобразена с ограниченията за градски условия, а именно 50 км/ ч.

IX. Четвърти опит за клъстеризация

При четвъртия опит за клъстеризация, подходът е същият като при третия опит, с тънката разлика, че колоните по които клъстеризираме за Скорост и Тегло.

Програмна реализация:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture

import warnings
warnings.filterwarnings("ignore")

data = 'result_comma.csv'
df = pd.read_csv(data)
df.head()

selected_columns = ['Скорост', 'Тегло']
data_selected=df[selected_columns]
data_selected=data_selected[data_selected['Тегло']>0]
data_selected.head()

missing_values = data_selected.isnull().sum()
print(missing_values)

data_selected = data_selected.replace(0, np.nan)
data_selected.head()

missing_values = data_selected.isnull().sum()
print(missing_values)

data_selected = data_selected.dropna()
data_selected.head()

missing_values = data_selected.isnull().sum()
print(missing_values)

from sklearn.ensemble import IsolationForest

# Създаване на модел за Isolation Forest
model = IsolationForest(random_state= 10,contamination=0.1) #
contamination е долята на аномалиите в данните

# Подготовка на данните X, които съдържат вашите данни
model.fit(data_selected) # Трениране на модела
predictions = model.predict(data_selected) # Предсказване на аномалиите

cleaned_df = data_selected[predictions == 1]

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

```

# Примерни данни
# предполагаме, че X е вашият набор от данни
X = cleaned_df

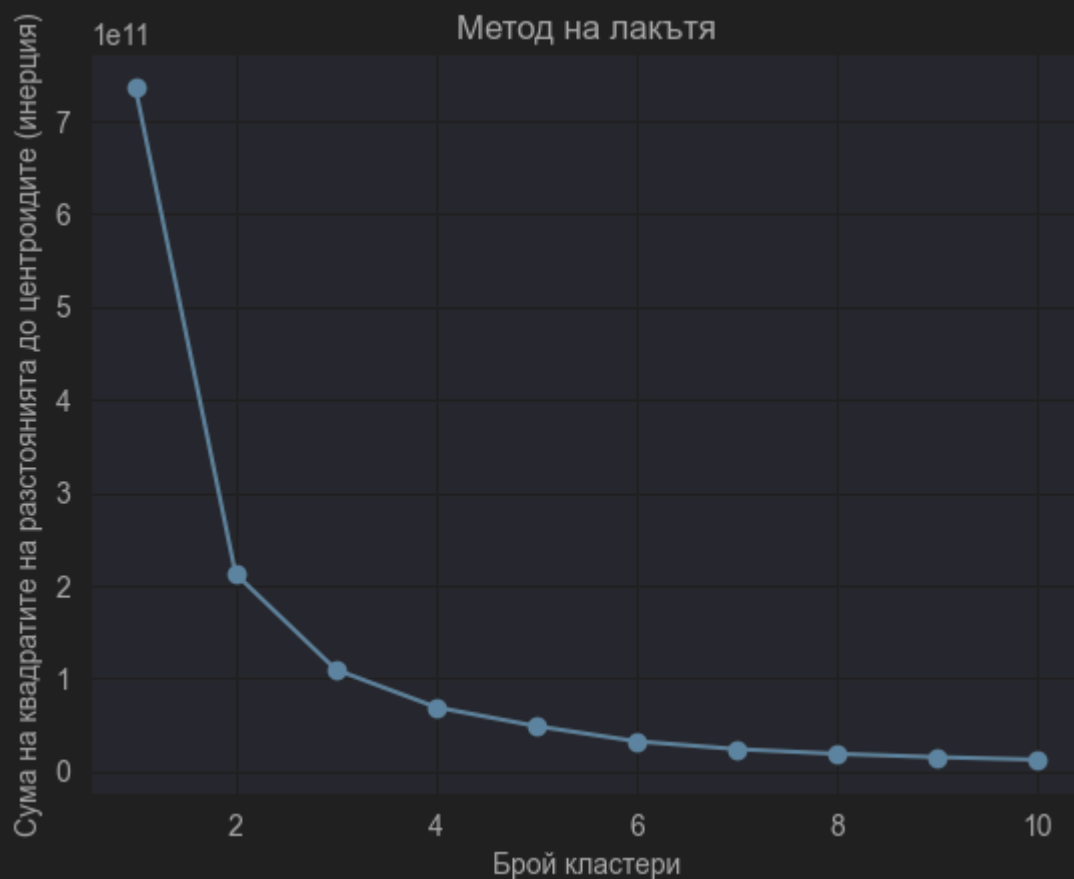
# Създаване на списък, в който ще съхраним стойности на вариацията за
# всяко k
variations = []

# Ползваме k от 1 до 10 за пример
for k in range(1, 11):
    # Създаване на k-means модел с k кластера
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)

    # Записване на вариацията за текущото k
    variations.append(kmeans.inertia_)

# Визуализация на метода на лакътя
plt.plot(range(1, 11), variations, marker='o')
plt.xlabel('Брой кластери')
plt.ylabel('Сума на квадратите на разстоянията до центроидите (инерция)')
plt.title('Метод на лакътя')
plt.show()

```



```

from sklearn.preprocessing import StandardScaler
numeric_columns = cleaned_df.select_dtypes(include=['number']).columns
scaler = StandardScaler()
data_standartized = scaler.fit_transform(cleaned_df[numeric_columns])
data_standartized

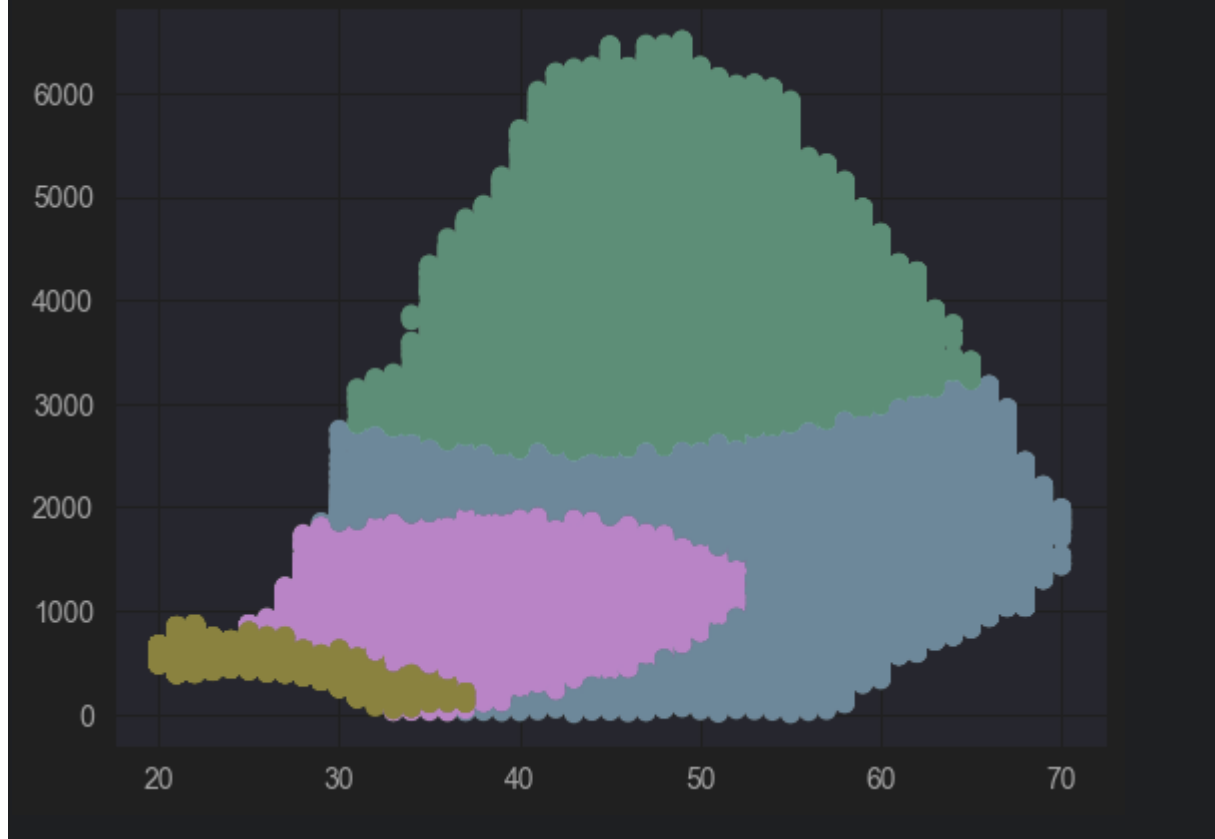
```

```

n_clusters = 4
gmm = GaussianMixture(n_components=n_clusters)
gmm.fit(data_standartized)
labels = gmm.predict(data_standartized)

data_standartized=cleaned_df.copy()
data_standartized['Cluster']=labels
plt.scatter(data_standartized['Скорост'],data_standartized['Тегло'],c=data_
standartized['Cluster'],cmap= 'viridis')
plt.show()

```



Тук отново имаме ясно изразени клъстери, като вече (посредством метод на лакътя), виждаме, че оптималният им брой е 4. Тази клъстеризация отново показва дадени зависимости между скоростта и теглото, които може да се направят. Вижда се, че се образува един клъстер на автомобилите с по-малко тегло и по – ниска скорост, също така има клъстер на автомобилите с тегло до 2 тона и скорост до 55 км / ч. Образуват се и два големи клъстера на автомобилите, които са с по-голямо тегло от 3 до 6.5 тона, които се движат със скорост от 35 до 65 км / ч, можем да предположим, че става въпрос за товарни автомобили. Друг голям клъстер, който се образува е на автомобили до 3 тона, движещи се със скорост от 30 до 70 км / ч. Изводът, който можем да си направим е, че по – леки автомобили (може би мотори, мотопеди и мотоциклети) са по-рядко срещани, за разлика от товарните автомобили и леките коли, които се срещат

приблизително равно на брой пъти и се движат с близки скорости от 30 до 70 км / ч, като товарните автомобили средно се движат с по-ниска скорост от леките коли.

Общият извод, който може да бъде направен за GMM алгоритъма е, че той е изключително подходящ за голям обем от данни, като това по никакъв начин не затруднява алгоритъма и скоростта на изпълнение е изключително добра. Също така този алгоритъм е много подходящ за клъстеризация на данни, които са с различна форма от сферична. GMM е много гъвкав модел, който може да моделира сложни форми на разпределение на данните. Той може да се справи с разнообразие от форми на клъстери, включително елипсовидни, сферични или неправилни форми. Може да се справи с клъстери с различни размери и дисперсии. Негов основен минус е необходимостта броят на клъстерите да бъде зададен предварително.

Х. Клъстеризация на малък обем от данни

За да бъде направено общо сравнение и извод между отделните алгоритми, които използваме, то извода до който стигнахме е, че това трябва да се случи върху по-малък обем от данни тъй като данните, с които разполагаме, не позволяват на алгоритми, които не са подходящи за голям обем от данни, да бъдат използвани.

За целта от файла с данни използваме данните само за една пътна лента, извършваме пречистване на несъстоятелните данни и извършваме клъстеризация с няколко различни алгоритъма - K – means, GMMs, HDBSCAN и Agglomerative clustering. Така експериментално можем да стигнем до извода, че не всеки алгоритъм е подходящ за всякакъв вид данни и, че отделните алгоритми имат различни преимущества и недостатъци, които трябва да бъдат съобразени в зависимост от конкретната цел на изследването, което правим, както и резултатите, които гоним.

Програмна реализация:

1) Импортване на необходимите библиотеки и четене на файла

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture

import warnings
warnings.filterwarnings("ignore")
```

```
data = 'result_comma.csv'
df = pd.read_csv(data)
df.head()
```

- 2) Избиране на една лента, която да остане, както и на колоните, по които ще клъстеризираме.

```
selected_columns = ['Лента']
file_lenta = df[selected_columns]

columns = ['Тегло', 'Скорост']
data_selected = df[df['Лента'] == 'L4'][columns]
```

- 3) Игнориране на резултатите, по-малки от 0, тъй като ги приемаме за несъстоятелни

```
data_selected = data_selected[data_selected['Тегло'] > 0]
```

- 4) Прилагане на IsolationForest

```
from sklearn.ensemble import IsolationForest

# Създаване на модел за Isolation Forest
model = IsolationForest(random_state=10, contamination=0.1) #
contamination е долята на аномалиите в данните

# Подготовка на данните X, които съдържат вашите данни
model.fit(data_selected) # Трениране на модела
predictions = model.predict(data_selected) # Предсказване на
аномалиите

cleaned_df = data_selected[predictions == 1]
```

- 5) Прилагане на метода на лакътя

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Примерни данни
# предполагаме, че X е вашият набор от данни
X = cleaned_df

# Създаване на списък, в който ще съхраним стойности на вариацията за
всяко k
variations = []

# Ползваме k от 1 до 10 за пример
for k in range(1, 11):
    # Създаване на k-means модел с k кластера
    kmeans = KMeans(n_clusters=k)
```



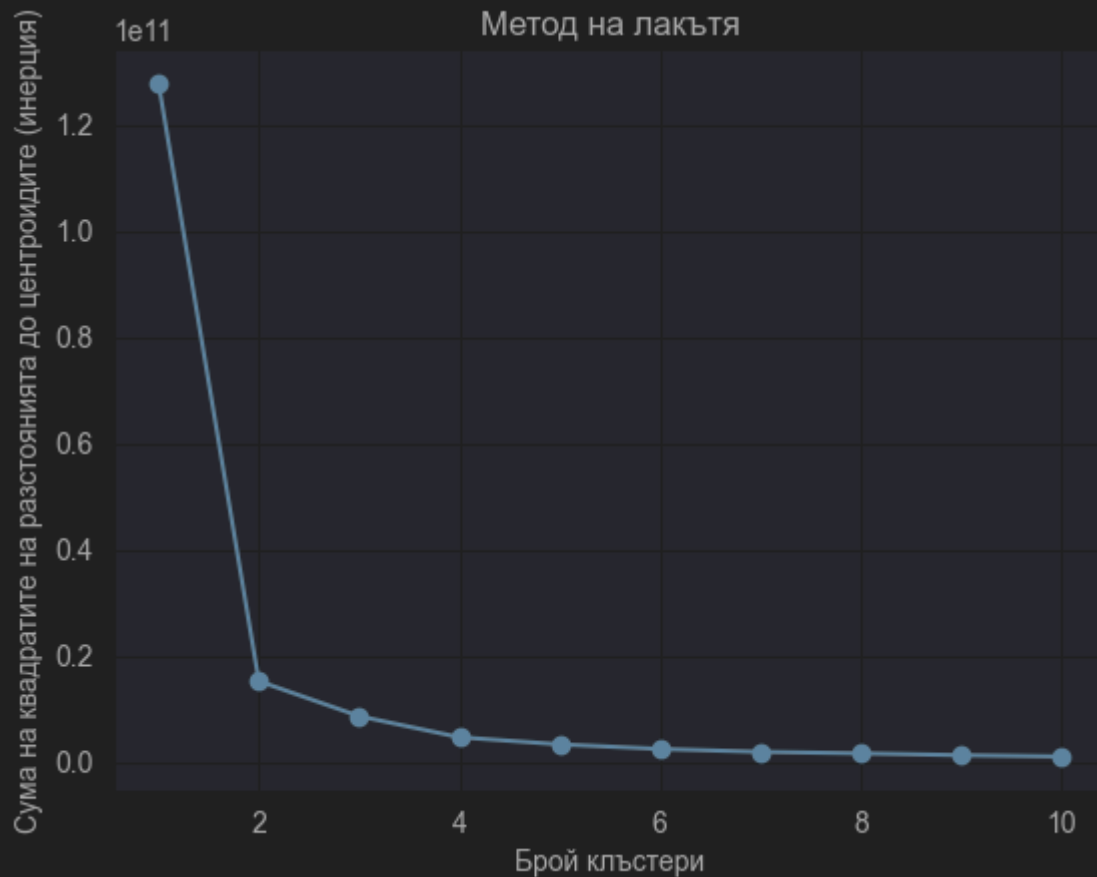
```

kmeans.fit(X)

# Записване на вариацията за текущото k
variations.append(kmeans.inertia_)

# Визуализация на метода на лакътя
plt.plot(range(1, 11), variations, marker='o')
plt.xlabel('Брой клъстери')
plt.ylabel('Сума на квадратите на разстоянията до центроидите (инерция)')
plt.title('Метод на лакътя')
plt.show()

```



6) Стандартизация

```

from sklearn.preprocessing import StandardScaler
numeric_columns =
cleaned_df.select_dtypes(include=['number']).columns
scaler = StandardScaler()
data_standartized = scaler.fit_transform(cleaned_df[numeric_columns])
data_standartized

```

7) Клъстеризация

```

n_clusters = 3
gmm = GaussianMixture(n_components=n_clusters)

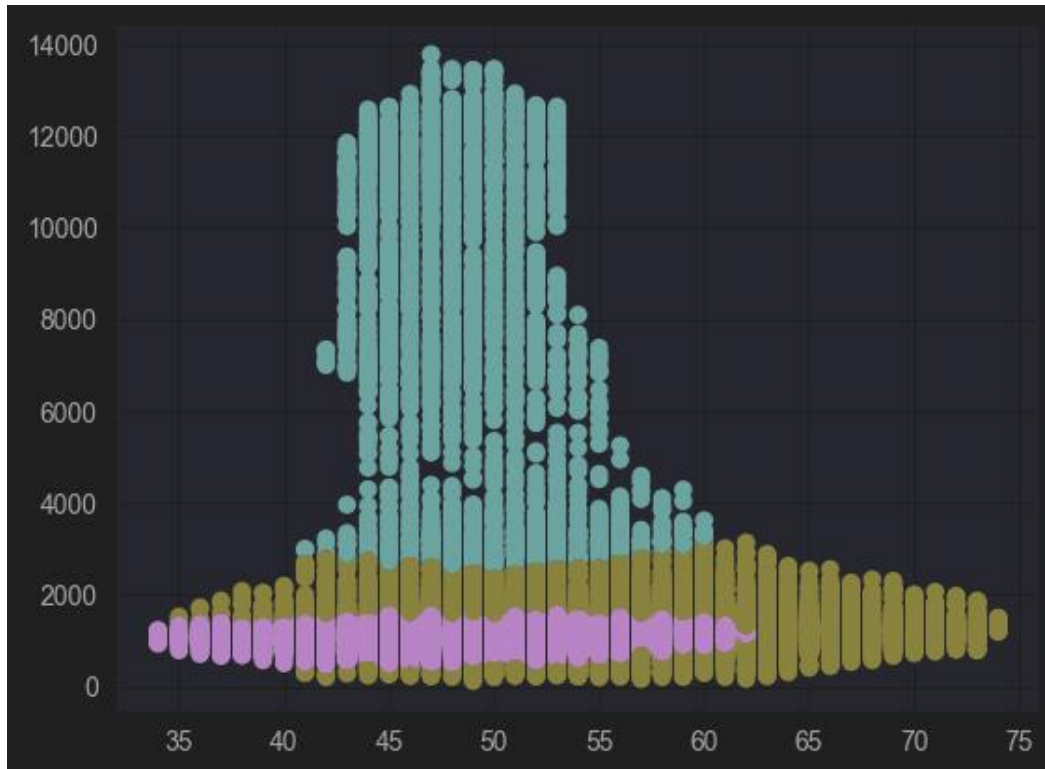
```

```
gmm.fit(data_standartized)
labels = gmm.predict(data_standartized)
```

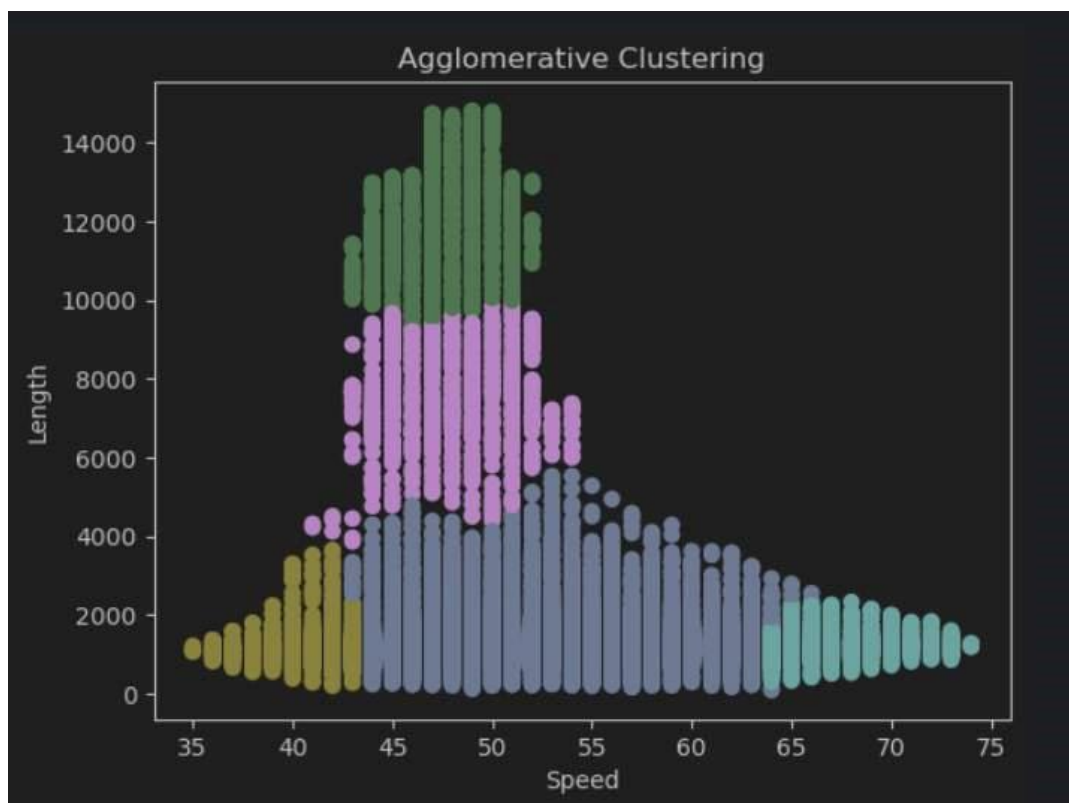
8) Визуализация

```
data_standartized=cleaned_df.copy()
data_standartized['Cluster']=labels
plt.scatter(data_standartized['Скорост'],data_standartized['Темло'],c=data_
standartized['Cluster'],cmap= 'viridis')
plt.show()
```

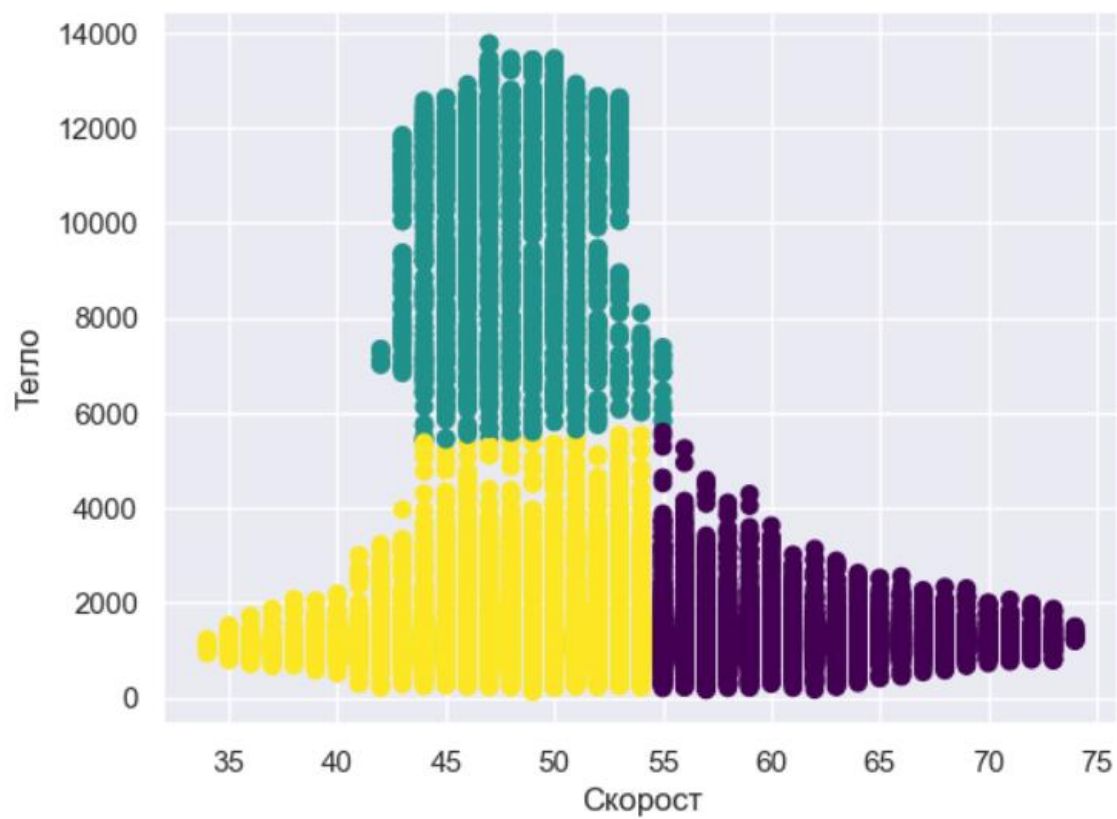
Резултат от GMM:



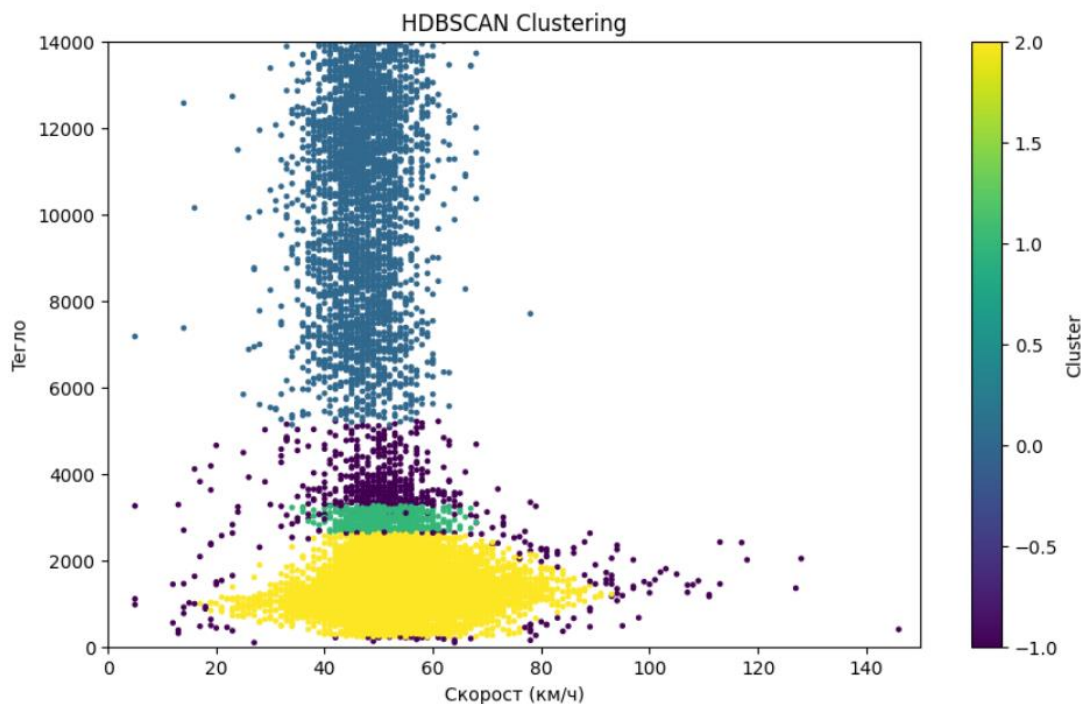
Резултат от Agglomerative Clustering:



Резултат от K – Means:



Резултат от HBDSCAN:



Извод:

В периодът на изследване на алгоритмите установихме, че Йерархичният метод и DBSCAN са прекалено тежки за такъв обем от данни и затова решихме да филтрираме данните като изследваме само едната лента (L4) от данните предоставени ни за пътното движение и по този начина намалихме резултатите до малко над 30 000. По този начин успяхме да съпоставим четирите алгоритъма. Изводите, които направихме са че GMM и K-means са бързи алгоритми подходящи за обработване на голям обем от данни или обработка в реално време. И при двата трябва броят клъстери трябва да бъде зададен предварително, което не винаги е много удачно и изпълнимо. Изборът на алгоритъм изцяло зависи от целта на изследването, за което ще бъде използван. DBSCAN е удобен, защото сам определя броя клъстери и засича outlier-ите след което ги отделя в отделен клъстер. Подходящ е за малък набор от данни. Йерархичния метод се използва при неизвестен брой клъстери и е интуитивен, но е бавен и затова не се използва в практиката. След като разгледахме четирите алгоритъма установихме, че изборът на всеки от тях може да се бъде направен в зависимост от данните с които разполагаме, производителността на компютъра върху който ще реализираме алгоритмите, както и целта на изследването ни.