Kernmengenbasiertes Clustering von Datenströmen auf Android-basierten Smartphones

Eike Ahmels

18.03.2015

Erstgutachterin: Prof. Dr. Katharina Morik
Zweitgutachter: Nico Piatkowski

Eike Ahmels
Adresse: Rüsingstraße 73, 44894 Bochum
Telefonnr.: 0234/45975434

Matrikel-Nr.: 133599
Fachsemester: 11
E-Mail: eike.ahmels@tu-dortmund.de

Inhaltsverzeichnis

Algorithmenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Smartphones werden in der heutigen Zeit überall eingesetzt. Sie werden nicht nur zur Kommunikation an sich, sondern auch für Freizeitaktivitäten, Büroapplikationen und industrielle Zwecke eingesetzt. Diese Nutzung erzeugt ein großes Maß von Daten, zum einem durch die gegenwärtig ausgeführten Applikationen (*Apps*), zum anderen hinsichtlich des aktuellen Systemzustands. Dieser beinhaltet die aktuelle Nutzung der verfügbaren Ressourcen des Smartphones wie des Hauptprozessors, des Speichers und der Netzwerkverbindungen, wie z.B. *Wireless Local Area Network* (WLAN) oder *Long Term Evolution* (LTE). Darüber hinaus wird der Status diverser Hardwarezusätze und Sensoren wie Beschleunigungssensoren oder *Global Positioning System* (GPS) Empfänger erfasst. Eventuell vorhandene Muster in diesen Daten könnten dabei helfen die Nutzungsbedingungen sowie die Energieverwaltung des Smartphones automatisch zu verbessern.

In dieser Bachelorarbeit werden aktuelle Algorithmen für das Auffinden solcher Muster in Datenströmen vorgestellt und mit Blick auf die Ausführung auf Smartphones untersucht. Eine Auswahl dieser Algorithmen wird anschließend so implementiert, dass sie direkt auf Android-basierten Smartphones ausgeführt werden können. Die implementierten Algorithmen werden bezüglich Speicherverbrauch, Rechenzeit und Batterieverbrauch verglichen, bei letzterem muss die Annahme genügen, dass hohe Rechenleistungen über die Zeit zu größerem Batterieverbrauch führt, da eine genaue Auswertung des Batterieverbrauches pro Applikation auf Android nicht möglich ist.

In Abschnitt 2 werden die Grundlagen für den Rahmen dieser Bachelorarbeit erklärt, die Reihenfolge der Unterabschnitte von Abschnitt 2 ist wichtig, da der Kernpunkt dieser Arbeit das Gruppieren von Daten auf Datenströmen ist, daher wird in Unterabschnitt 2.2 zuerst auf Datenströme eingegangen und danach in Unterabschnitt 2.3 das Gruppieren von Daten erläutert.

Danach wird der Algorithmus selber vorgestellt werden, dabei halte ich mich strikt an das Dokument der Ersteller des Algorithmus. Darauf folgt eine Beschreibung des Entwicklungsprozesses. Dieser teilt sich auf in die Erklärung der Grundlagen (2), dies heißt im einzelnen, ich gehe mit UML-Diagrammen auf die Struktur der Implementierung ein und erläutere wie der Algorithmus in der Objekt-Hierarchie vorgeht. Danach gehe ich in Unterabschnitt 4.2 auf die Zielfunktionen ein, da diese eine Besonderheit aufweisen müssen, um zusammen mit dem Algorithmus dieser Arbeit zu funktionieren. Es folgt in Unterabschnitt 4.3 eine Beschreibung der einzelnen Stadien während der Entwicklung. Die Stadien werden anhand von Bildern verdeutlicht und erklärt.

Darauf wird in Abschnitt 5 der programmierte Algorithmus evaluiert, dies passiert anhand von zwei Datensätzen in den Unterabschnitten 5.2 und 5.4. Weiter wird in Unterabschnitt 5.3 untersucht welche Auswirkungen die Reihenfolge, in welcher die Daten aus dem Datenstrom verarbeitet werden, auf die Erstellung der Datenstruktur hat.

Zum Schluss schließe ich in Abschnitt 6 mit einem Fazit zur Arbeit ab und weise auf Probleme und Untersuchungen hin, die noch gemacht werden müssen.

Ziel dieser Bachelorarbeit ist es zuerst den Algorithmus zu implementieren, sodass er auf Datenströmen und auf Android-Smartphones benutzt werden kann. Außerdem soll der Unterschied zwischen Nachbarschaftssuchen untersucht werden, dies geschieht anhand zweier Datensätze. Weiter untersuche ich, welche Ergebnisse der für k-means geschaffene Algorithmus mit einer anderen Zielfunktion erzielt.

2 Grundlagen

In den Grundlagen sollen die Begriffe und Themen um die es in der Arbeit geht erklärt werden, damit die nachfolgenden Texte ohne weiteres verstanden werden können.

2.1 Android Smartphones

Was Smartphones angeht, gibt es drei große Unternehmen, die Betriebssysteme für diese herstellen. Apple mit iOS, Microsoft mit Windows 8.1 und Google mit Android. Android hat sich sehr viel weiterentwickelt und befindet sich mittlerweile in der *major version* 5.

Man kann die Betriebsysteme unter vielen Aspekten vergleichen, für diese Arbeit ist der Aspekt der eigenen Programm-Entwicklung wichtig. Denn Android basiert auf einem Unix System, so, dass man eigene Programme und Entwicklungen benutzen kann, sofern nur die im Smartphone verwendete Chip-Architektur unterstützt. Deshalb gibt es viele Erweiterungen für Android-Smartphones, es gibt ganze Frameworks, die eine bekannte Linux-Bash-Shell-Umgebung auf einem Android Smartphone ermöglichen usw.

Ein weiterer Vorteil von Android ist, dass es fertige Java-Umgebungen gibt, die fast die komplette Bandbreite der Java Möglichkeiten unterstützten.

Ein möglicher Nachteil könnte sein, dass mobile Geräte keine herkömmlichen Architekturen für die Chips verwenden, bei Android ist die ARM-Architektur am verbreitetsten. Wenige Firmen bieten direkt für diese Architektur vorbereitete Software an, daher müssen Enthusiasten oft Software für ihre mobilen Endgeräte selber übersetzen. Wenn Software für Java entwickelt wurde, stellt sich dieses Problem jedoch nicht.

2.2 Streams

Um Datenstromalgorithmen so modular zu implementieren, dass sie sowohl eigenständig in einer App als auch als Teil einer größeren Datenanalyse eingesetzt werden können, kann ein *Datenstrom-Framework* verwendet werden. Solche Frameworks erlauben eine einfache Anbindung von Algorithmen an eine Vielzahl strömender Datenquellen. Bekannte Datenstrom-Frameworks wie Twitters Storm sind freilich auf die Ausführung in großen Rechenzentren bzw. der Cloud ausgelegt und sind nicht ohne Weiteres auf aktuellen Smartphones ausführbar.

2.2.1 Streams-Framework

Das Datenstrom-Framework *Streams* [3] erlaubt es, die Ausführungseinheit auszutauschen und ist daher sowohl auf Rechnerclustern, als auch auf Workstations und sogar Smartphones lauffähig. Mit dem Streams-Framework lassen sich die Datenstromverarbeitungsprozesse flexibel in einem XML-Format definieren, wodurch es sich für eine

Vielzahl von Anwendungsfällen verwenden lässt. Darüber hinaus ist es quelloffen, frei erweiterbar und wird in mehreren internationalen Forschungsprojekten eingesetzt.

Außerdem gibt es schon Implementierungen um mit Android zu kommunizieren, so können bspw. Sensoren ausgelesen, oder mit einem Adapter auf Apps zugegriffen werden; auch die Verwendung von anderen Daten aus dem Smartphone ist möglich.

2.2.2 ARM Architektur und Java

Wie die meisten mobilen Geräte haben die meisten Chips von Android-Geräten ARM-Chips.

Für diesen Chipsatz funktionieren die meisten Programme nicht ohne weiteres und man muss extra für ARM übersetzten. Außerdem sind nicht alle Funktionen für die Chips von Smartphones möglich. Also wäre man mit nativer Programmierung hier schnell am Ende.

Wie schon erwähnt, laufen viele Programme nicht »out of the box« auf Android Smartphones. Es gibt aber von Oracle eine Java-Runtime-Umgebung für die ARM-Architektur. Dies bedeutet, dass (fast) jedes in Java geschriebene Programm auf Android laufen wird (als Konsolen-applikation). Da das Streams-Framework ebenso auf Java basiert, setzen wir hier auf Java als Grundlage für die Entwicklung des Algorithmus.

Außerdem ist der Algorithmus so komplett plattformübergreifend und kann für verschiedene Anwendungsbereiche benutzt werden.

2.3 Maschinelles Lernen

Seit der Vernetzung der Welt gibt es zu vielen Sachverhalten große Datenmengen. Diese Daten können dabei alles sein, es könnten z.B. Daten über Schlaganfälle bei Patienten sein oder handgeschriebene Postleitzahlen. Mit den Daten zu den Schlaganfällen könnte man versuchen zu berechnen ob die Gefahr für einen zweiten besteht. Eine solche Auswertung der Daten würde die Möglichkeiten eines Menschen weit überschreiten, daher nimmt man lernfähige Computer zur Hilfe. Diese Computer lernen anhand von bestehenden Daten, die Wahrscheinlichkeit für ein Ereignis eines neuen Datums vorherzusagen. Je mehr Daten er zum Lernen hat, desto genauer werden die Vorhersagen für neue Daten werden.

In diesem Vorgehen unterscheidet man grob in zwei unterschiedliche Modelle [10].

Überwachtes Lernen Hier wird der Lernvorgang in dem Sinne überwacht, dass die Lösung zu einem Problem vorhanden ist und der Computer während des Lernens ständig prüfen kann, ob seine Ergebnisse stimmen. Das bedeutet, es gibt zu jedem Datum eine *Label* (vgl. Hastie/Tibshirani/Friedman 2003, S. 12).

$$\mathcal{D}_{\mathcal{S}} = \left\{(\boldsymbol{x}^{(1)}, y^1), (\boldsymbol{x}^{(2)}, y^2), \ldots, (\boldsymbol{x}^{(N)}, y^N)\right\}$$

Dieses ist das gängigste maschinelle Lernverfahren und das am meisten erforschte.

Unüberwachtes Lernen Dies ist genau das selbe wie überwachtes Lernen, nur dass es zu den Daten keine Labels gibt und somit keine Lösung während des Lernens vorhanden ist. Daraus ergibt sich auch die Folgerung, dass eine optimale Lösung nicht bekannt ist und eine berechnete Lösung nicht unbedingt mit einer optimalen verglichen werden kann (vgl. Hastie/Tibshirani/Friedman 2003, S. 12).

$$\mathcal{D}_{\mathcal{U}} = \left\{ \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)} \right\}.$$

Dieser Art gehört z.B. das Clustering an, denn es gibt vor der Berechnung bzw. dem Lernenn keine Möglichkeit, das Ergebnis zu messen.

2.3.1 Clustering

Ein Hauptthema dieser Arbeit ist das *Clustering*, denn der Algorithmus ist genau dafür entworfen worden.

Diest ist eine Möglichkeit, Muster in Daten zu finden. Dabei werden beliebige Daten nach vorgegebenen Kriterien in sogenannte *Cluster* gruppiert. Die Kriterien werden durch ein *Distanzmaß* repräsentiert, welches den *Abstand* zweier Datenpunkte darstellt. Das Ziel einer optimalen Gruppierung ist es, den Abstand der Punkte innerhalb eines Clusters zu minimieren und den Abstand zwischen Punkten verschiedener Cluster zu maximieren. Das Clustering findet in verschiedensten Bereichen Anwendung, so wurde z.B. gezeigt, dass es drei Gattungen der Iris Blume gibt (in der Bachelorarbeit wird dieser Datensatz zur Veranschaulichung des Algorithmus verwendet). Doch gibt es noch sehr viele andere Anwendungsmöglichkeiten, man könnte diese in der Werbung nutzen oder berechnen an welchem Standpunkt ein neuer Supermarkt die effektivste Position hat. Auch anderen Datenzusammenhänge können mit Clustering gefunden werden.

2.3.2 Kernmengen

Als eine Kernmenge (coreset) bezeichnet man eine Teilmenge eines Datensatzes, anhand dessen die Lösung für den gesamten Datensatz annäherungsweise gelöst werden kann.

Kernmengen sind abstrakte Aggregationen von Daten, die gewisse Eigenschaften der Rohdaten bewahren. Welche Eigenschaften das sind, hängt von der zu lösenden Aufgabe ab. Kernmengen für das Clustering fassen die Daten so zusammen, dass die Güte eines optimalen Clusterings der Daten in der Kernmenge höchstens um den Faktor ϵ von der Güte eines optimalen Clusterings der Rohdaten abweicht. Der Qualitätsverlust durch die Aggregation der Daten ist also beschränkt.

Das erste Mal wurden Kernmengen in einer Arbeit ([9]) zu k-means/k-median verwendet, hier wurde eine Kernmenge berechnet um die k-means-Zielfunktion als eine $(1+\epsilon)$-Approximation zu lösen, jedoch nur für statische Datensätze.

In diesem Fall ist die Kernmenge auf dem Datenstrom eine Menge von *coreset features*1.

Definition 1. Ein *coreset feature* ist ein Quadrupel $(n, s_1, S_2, \mathbf{s_0})$. Es wird aus einer Teilmenge $M \subseteq \mathcal{D}_U$ des Datensatzes berechnet, wobei $n = |M|$ die Anzahl der im Feature aggregierten Punkte berzeichnet, s_1 die Summe über die Punkte und S_2 die Summe aller quadrierten Längen der Punktvektoren. Der Vektor $\mathbf{s_0}$ ist der Referenzvektor der coreset features.

$$\begin{aligned} s_1 &= \sum_{\boldsymbol{x} \in M} \boldsymbol{x} \\ S_2 &= \sum_{\boldsymbol{x} \in M} \|\boldsymbol{x}\|_2^2 \end{aligned}$$

Die Motivation ist es, mit einem coreset feature mehrere Punkte durch einen Punkt zu repräsentieren und dabei in qualitativ nur um den Faktor ϵ von den originalen Daten abzuweichen; das bedeutet, wenn eine Lösung aus den original Daten berechnet wird, dann weicht die Lösung, die durch Benutzung der Kernmenge berechnet wurde, höchstens um den Faktor ϵ ab.

2.4 Zielfunktion

Für das Clustering gibt es verschiedene Zielfunktionen. Eine Zielfunktion beschreibt, was optimiert werden muss, um zur Lösung zu gelangen. In dieser Bachelorarbeit wird die k-means-Zielfunktion behandelt, da dieser Algorithmus für diese Art von Problem geschaffen wurde. Theoretisch ist der Algorithmus aber auf jede andere Zielfunktion adaptierbar. Aber die Berechnung der coreset features wird anhand der k-means-Kostenfunktion getätigt.

Im Folgenden werden verschiedene Algorithmen zum k-means-Problem beschrieben und gezeigt, welcher Algorithmus in dieser Arbeit Anwendung findet und warum.

2.4.1 BIRCH

Eine Möglichkeit, ein Clustering zu lösen, ist mit Hilfe von Heuristiken. BIRCH ist ein Algorithmus der auch auf Kernmengen basiert, aber die Zuweisung der Datenpunkte zu den Kernmengen mit Heuristiken löst. Die Ausführungszeit von BIRCH ist dementsprechend gering, jedoch lässt sich keine Aussage über die Qualität der Lösung treffen, also kann diese beliebig schlecht sein. Experimente haben gezeigt, dass BIRCH zu einem großen Teil sehr schlechte Lösungen liefert.

In der Abbildung2.1 sieht man die Lösung die BIRCH berechnet hat als »X«. Hier ist deutlich zu sehen, dass die berechneten Mittelpunkte keine hohe Qualität aufweisen, denn die von BICO berechneten Mittelpunkte (Kreise mit Punkt), teilen den Datensatz in zwei sinnvolle Teilmengen.

2.4.2 Neural Gas

Ein weitere Ansatz des Clusterings auf Datenströmen heißt »G-Stream: Growing Neural Gas over Data Stream«[7]. Bei diesem Ansatz ist ein einmaliges Iterieren über dem Datensatz ohne Parameter für die Mittelpunktsanzahl nötig, damit alle Mittelpunkte gefunden werden.

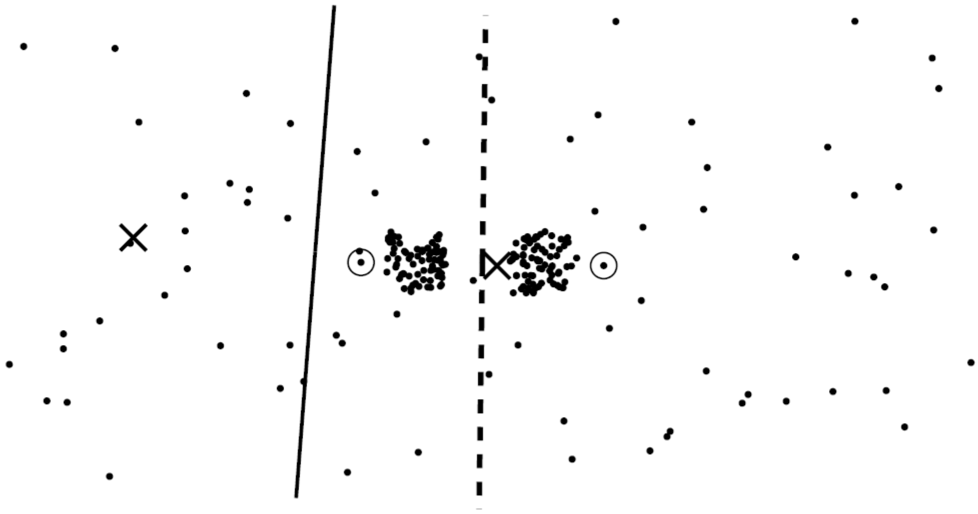


Abbildung 2.1: Vergleich einer Berechnung von BIRCH und BICO[6]

Die Vorgehensweise ist dabei, dass ein Graph erzeugt wird, in welchem jeder Punkt im Graphen mit seiner Nachbarschaft verknüpft ist und die Entfernung gespeichert wird. Außerdem werden zu jedem Punkt die »weit entfernten« Punkte festgehalten. Dies stellt sicher, dass man mit einer Iteration alle Mittelpunkte findet.

In der Praxis ist dieser Algorithmus nicht ohne weiteres anwendbar, da unbekannte Parameter eingestellt werden müssen und die Anwendung nicht praktikabel ist. Außerdem ist die Algorithmus Beschreibung undurchsichtig.

2.4.3 *k*-means

k-means ist eine Zielfunktion für das Clustering-Problem.

Definition 2. k-means versucht den Datensatz in k Gruppierungen zu unterteilen, sodass die Summe der quadrierten Abweichungen von den Gruppenmittelpunkten minimal ist. Dazu wird die Funktion J optimiert, mit dem Datensatz S und den Mittelpunkten μ_i und $1 \leq i \leq k$.

$$J = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|\mathbf{x}_j - \mu_i\|^2$$

Es wurden schon viele Algorithmen zur Lösung dieses Problems entwickelt. Der erste k-means-Algorithmus (Lloyds Algorithmus [12]), die erste Weiterentwicklung von MacQueen [13], die Erweiterung mit durch Wahrscheinlichkeiten clever gewählten Startlösungen k-means++ [2] sowie der dichtebasierte Algorithmus DBSCAN [5] gehören zu den bekanntesten Algorithmen für das Clustern statischer Datenbestände, wobei DBSCAN allerdings nicht die k-means-Zielfunktion minimiert. Hierbei wird jedes Element im Datensatz $\mathcal{D}$ mit $|\mathcal{D}| = N$ einem *Cluster* zugeordnet. Sollen stattdessen die Elemente eines Datenstroms gruppiert werden, so ist N monoton wachsend, d.h. die Anzahl der zu gruppierenden Objekte ist potentiell unendlich, und bestehende Cluster-Algorithmen können nicht ohne Weiteres angewendet werden, denn da zu keinem Zeitpunkt auf »alle« Daten zugegriffen werden kann, kann z.B. die Distanz zwischen einem Datenpunkt und allen anderen Datenpunkten nicht berechnet werden.

2.4.4 DBScan[5]

Wie schon erwähnt gibt es noch weiter Ähnlichkeitsstrukturen in Datensätzen. Eine Möglichkeit ist, zwei Datenpunkte dann als ähnlich anzusehen, wenn die Entfernung zueinander kleiner als ein wählbares λ, und wenn dann genug Datenpunkte sich nacheinander ähnlich sind, dann kann ist diese Gruppierung ein Cluster. Hierbei gibt es keine Zentren mehr, da hier tatsächlich geometrische Strukturen im Datensatz erkannt werden und es keinen Mittelpunkt in dieser Variante gibt. Die Anzahl der Cluster in einem Datensatz ist unbekannt und wird nicht als Parameter benötigt.

Definition 3. DBScan sucht nach Gruppen C_i von Datenpunkten in einem Datensatz D, sodass jeder Datenpunkt in dieser Gruppe zu mindestens einem anderen Datenpunkt maximal den Abstand λ hat. Die Idee ist, dass man Erreichbarkeiten definiert und sagt, dass Punkte auch transitiv über andere Punkte erreichbar ist, falls der Abstand $d_{dbscan}(d_i, d_j)$ immer kleiner als λ ist. Seien $x, y, z \in D$ und $d_{dbscan}(x, y) \leq \lambda \wedge d_{dbscan}(y, z) \leq \lambda \wedge d_{dbscan}(x, z) > \lambda$, dann ist x transitiv über y von z erreichbar (vgl. Cao et. al., 2006, S. 328-339). Die Transitivität ist mit beliebig vielen Punkten erweiterbar.

Wenn eine solche Gruppe (alle Punkte untereinander erreichbar) mehr als m Punkte aufweist, ist diese Gruppe ein Cluster.

Denstream[4]

Es existiert bereits ein Algorithmus der auf einem Datenstrom DBScan löst. Dieser Algorithmus erstellt während Daten aus dem Stream kommen »*micro-cluster*«, aus der Menge an micro-clustern wird dann das Clustering auf dem ganzen Datensatz berechnet.

2.5 Stream-Clustering

Aus diesem Grund werden Algorithmen entwickelt, die speziell auf das Gruppieren strömender Daten ausgelegt sind [14]. Direkte Datenstrom-Varianten von klassischem k-means sind schnell und Ressourcensparend, jedoch kann theoretisch und empirisch gezeigt werden, dass die Güte des so berechneten Clusterings vergleichsweise gering ist. Gleiches gilt für den Datenstromalgorithmus BIRCH [15]. BIRCH repräsentiert Punktmengen über so genannte *Clustering Features* innerhalb einer balancierten *Baumstruktur*. Sowohl die Baumstruktur als auch die Clustering Features können beim Eintreffen neuer Daten aus dem Datenstrom effizient aktualisiert werden.

Bei den Algorithmen StreamKM++ [1] und StreamLS [8] werden die Daten des Stroms dagegen in Blöcken fester Länge eingelesen. Aus jedem Block wird eine so genannte *Kernmenge* berechnet.

2.6 BigData

Im Zeitalter der Supercomputer und des Internets sind Daten der Schlüssel zu vielem, jedoch ist es nicht möglich alle diese Daten zu analysieren und zu speichern. Als Beispiel könnte hier Google dienen, denn Google hat Rechenzentren mit Datenspeichern von Petabyte Größe. Daten sind aber interessant für viele Bereiche, vor allem auch Daten von Smartphones. Nimmt man die Gesamtheit aller von Smartphones produzierten Daten, kann man hier ohne Probleme von BigData sprechen. Und BigData effizient und effektiv analysieren zu können wäre ein Meilenstein der modernen Informatik.

3 Clustering auf Smartphones

Die Daten, die ein Smartphone produziert, können für verschiedenste Zwecke sehr interessant sein. Um diese Daten zu verarbeiten, gibt es drei Möglichkeiten. Erstens, alle Daten werden an einen Server geschickt und dort weiter verarbeitet. Zweitens, die Daten werden direkt auf dem Smartphone verarbeitet und das Ergebnis wird verschickt. Und Drittens, die Daten werden aggregiert und dann weiter verarbeitet, dabei könnte die Verarbeitung auf dem Smartphone, oder auf einem entferntem System passieren.

Wenn die Daten auf dem Smartphone verarbeitet werden, und das würde im Kontext dieser Bachelorarbeit ein k-means++-Clustering der gesamten Daten bedeuten, dann würde dies sehr lange dauern und mit sehr großer Wahrscheinlichkeit auch die Kapazität des Akkus und bzw. oder die des Arbeitsspeichers überschreiten. Die zweite Möglichkeit wäre das Hochladen einer großen Datenmenge bedeuten und dies kostet wiederum sehr viel Akku-Leistung. Würde man die Daten aggregieren und dann weiter verarbeiten, würden beide Prozesse, k-means++ und Hochladen, deutlich weniger Ressourcen verbrauchen, da nur ein Bruchteil der Ursprungsdaten verarbeitet oder hochgeladen werden.

Somit ist eine ressourcen-sparende und für den Alltag brauchbare Methode, Clustering auf dem Smartphone zu betreiben, die Aggregierung der Daten. Um dies zu erreichen gibt es verschiedene Möglichkeiten. Man könnte nur jedes 10te Datum nehmen und hätte so nur 10 % der Größe. Jedoch würde dies keine gute Repräsentation des Ursprungsdatensatzes bedeuten, da sozusagen wahllos Daten gelöscht würden und so signifikante Daten verloren gingen. Man könnte herkömmliche Komprimierungsmechanismen vor dem Hochladen verwenden. Da die Daten aber im Datenstrom vorliegen, würde während des Komprimieren die Datenaufnahme gestoppt werden müssen. Außerdem kann kein Grad der Komprimierung garantiert werden. So würden im schlechtesten Fall der gesamte Datensatz in voller Größe verarbeitet werden.

Eine weiter Möglichkeit sind *Kernmengen*. Der Ansatz von Kernmengen ist, die Daten zusammenzufassen, deren Aussage dabei aber nur bis zu einem gewissen Prozentsatz zu verfälschen. Das bedeutet, dass wir einen großen Datensatz auf wenige Datenpunkte reduzieren und mit dem reduzierten Datensatz dann trotzdem die Kernaussage approximieren können.

Für Clustering auf Smartphones bedeutet das, dass ein viel kleinerer Datensatz bearbeitet werden muss. Heißt es wird nur ein Bruchteil der Datenmenge geclustert oder verschickt. Dabei reduziert sich der Ressourcenverbrauch und die Zeit die in Anspruch genommen wird, egal für welchen Weg man sich entscheidet.

Die Kernmenge, die in diesem hier entwickelten Algorithmus, verwendet wird, verwendet zur Aggregation sogenannte *coreset features.*

3.1 BICO

Die Idee ist es die Struktur von *BIRCH* zu übernehmen und eine gewisse Qualität und Güte zu garantieren. Dazu werden pro coreset feature (Def. 1) die Kosten begrenzt und es gibt pro coreset feature eine Kindsmenge von weiteren coreset features.

Trifft ein neues Datum ein, wird zu diesem Datenpunkt erst das am nächsten liegende coreset feature im Level 1 gesucht und dann getestet, ob das Datum im Radius dieses coreset features liegt. Falls nicht, wird ein neues coreset feature geöffnet. Sollte das Datum im Radius liegen, wird weiterhin geprüft ob das coreset feature das Datum aufnehmen kann, ohne die maximalen Kosten pro coreset feature zu überschreiten. Im ersten Fall ist klar, das Datum wird ins coreset feature aufgenommen, die Anzahl der Elemente wird erhöht, zur linearen Summe wird das Datum hinzugefügt und auf die Summe der quadrierten Länge wird die quadrierte Längen des Datums addiert. Sollten die Kosten jedoch überschritten werden, wird das Level um eins erhöht und der Einfügealgorithmus wird mit der Kingsmenge des dieses coreset features ausgeführt. Die maximale Anzahl an coreset features ist begrenzt. Das ganze Verfahren ist in Algorithmus 1 zu sehen.

Algorithmus 1 Basis Algorithmus

```
 1: Input: $p \in \mathbf{R}^d$
 2: Set $f = p$, $S = \text{children}(CF(p))$ and $i = 1$;
 3: if $S = \emptyset$ or $\|p - \text{nearest}(p, S)\| > R_i$ then
 4:   Open new CoresetFeature with reference point $p$ in level $i$ as child of $CF(f)$
 5: else
 6:   Set $r := \text{nearest}(p, S)$;
 7:   if $\text{cost}_{CF}(CF(r) \cup p) \leq T$ then
 8:     Insert $p$ in $CF(r)$;
 9:   else
10:     Set $S := \text{children}(CF(r))$;
11:     Set $f := r$ and $i := i + 1$;
12:     Goto line 3;
13:   end if
14: end if
```

Der Radius wird pro Level berechnet. Zum ersten Level gehören alle coreset features der ersten Ebene. Pro Menge an Kind-Coreset-Features wird das Level um eins erhöht.

Der Radius hängt stark vom *threshold* T ab und dieser ist definiert wie folgt:

$$OPT/(k * f(\epsilon)) \leq T \leq 2 * OPT/(k * f(\epsilon)).$$

Da wir wärend der Ausführung die Werte für T nicht haben, können wir diesen am Anfang nicht setzen. Im ursprünglichen Algorithmus wird anhand der ersten Daten »trainiert«, dass bedeutet es werden die Abstände der ersten Daten zueinander getestet und aus dem minimalen Abstand wird der *threshold* gebildet. Dies ist nicht optimal und führt dazu, dass der threshold zu klein ist, so dass die maximale Anzahl an coreset features überschritten wird. Jetzt kann der Algorithmus die Anzahl der coreset features verringern ohne Qualität (unter dem threshold) zu verlieren.

Der Neubildungsteil arbeitet so, dass erst der threshold verdoppelt wird, um die möglichen Kosten pro coreset feature zu vergrößern, da sich mit größerem threshold der Radius vergrößert und mehr Punkte einschließt. Da es aber keine Punkte in dem Sinne mehr gibt, denn diese werden in den coreset features nicht mehr einzeln repräsentiert, wird getestet ob sich zwei Coresetfeatures vereinigen lassen, also ob die Kosten der Vereinigung zweier coreset features auf selbem Level unter den Kosten, die durch den threshold definiert sind, bleiben. Außerdem wird versucht aus den Kindsmengen der coreset features, welche mit dem Elternknoten zu vereinigen. Dies geschieht nach den selben Kriterien wie das Vereinigen auf dem selben Level.

Dies wird so lange wiederholt bis die maximale Anzahl an coreset features unterschritten ist (Algorithmus 2). Danach werden wieder Daten aus dem Datenstrom der Struktur hinzugefügt.

Algorithmus 2 Neubildungs Algorithmus, wenn Anzahl an Coresetfeatures zu groß wird

```
Input: Set of CoresetFeature (root set) $R_{CFs}$
Set $\text{OPT}_{est} := 2 * \text{OPT}_{est}$;
Create a new empty level 1 (increase every number of all existing levels in $R_{CFs}$)
$S := \emptyset$ (empty set of ClusteringFeatures in level 1;
$T := R_{CFs}$
for all CFs $X \in T$ with reference point $p$ do
  if $S = \emptyset$ or $> R_1$ then
    Move $X$ from $T$ to $S$;
    comment: implicitly moves all children of $X$ one level up
  else
    if $\text{cost}(X \cup CF(\text{nearest}(p, S))) \leq T_i$ then
      Insert $X$ into $CF(\text{nearest}(p, S))$;
    else
      Make $X$ a child of $CF(\text{nearest}(p, S))$;
    end if
  end if
end for
comment: Traverse through the CoresetFeature tree and, if possible merge CoresetFeatures into parent CoresetFeatures
```

Mathematisch gibt es einen optimalen Threshold, dieser ist durch die Gegebenheit des Datenstroms nicht berechenbar, denn dafür benötigt man alle Daten.

Wenn aber ein Threshold $T \leq OPT/(k{*}f(\epsilon))$, durch das Training der Daten berechnet wird, wird dieser durch die Neubildung immer verdoppelt. Es kann gezeigt werden, dass dadurch der Threshold auf jeden Fall in dem Intervall $[OPT/(k{*}f(\epsilon)), 2{*}OPT/(k{*}f(\epsilon))]$ liegt. Zu finden ist dieser Beweis im Anhang des BICO-Papiers.

4 Entwicklung

Der Anfang der Entwicklung gestaltet sich als sehr schnell, da die mathematischen Definitionen übernommen werden konnten und so der Kern des Algorithmus schnell programmiert wurde.

4.1 Grundlagen

Zum besseren Verständnis, werde ich die Klassen-Struktur mit Hilfe eines UML-Diagrams zeigen und die einzelnen Bestandteile erklären.

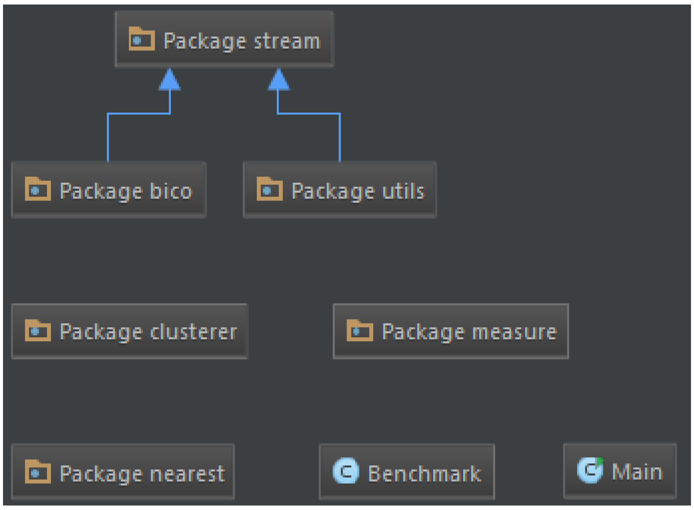


Abbildung 4.1: Klassenstruktur

Im Packet »bico« gefindet sich das Streams-Framework. Im »utils« Paket befinden sich Hilfsklassen und andere nützliche Funktionen, sodass diese bei Bedarf schnell aufgerufen werden können. Das Paket »nearest« enthält das Interface »INearest«, auf diesem bauen alle Nachbarschaftssuchen auf. Ähnlich wie im Paket »measure«, hier befindet sich das Interface »IMeasure« um die Ähnlichkeit zweier Datenpunkte zu messen. Das letzte Paket »clusterer« enthält Wrapper-Klassen für Clustering-Implementierungen, oder auch eigene Implementationen. Weiter werde ich jetzt genauer auf die einzelnen Pakete, deren Klassen und die wichtigen Funktionen eingehen.

Paket »bico« Im Paket »bico« (Abbildung(4.2)) befinden sind alle Objekte, die benötigt werden, um die *Coreset-Feature*-Struktur aufzubauen.

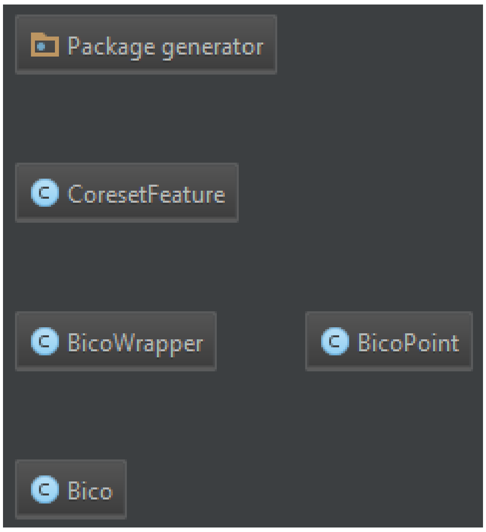


Abbildung 4.2: Paket »bico«

Als Objekt für einen Vektor habe ich die Klasse »BicoPoint« erstellt. Das Interface »Clusterable« muss für die Apache-Implementationen der Zielfunktionen eingebunden werden.

Diese ist eine Repräsentation eines Vektors mit Zahlen aus $\mathbb{R}$. Die Funktion »Dimensions« gibt die Anzahl der Dimensionen des Vektors zurück, hierbei wird keine extra Zahl gespeichert, sondern die Größe des internen Vektors abgefragt. Mit der Funktion »getEntry« kann man auf die einzelnen Werte der Dimensionen zugreifen. Für einige Berechnungen muss man den ganzen Vektor durch eine Zahl teilen, dies übernimmt die Funktion »div«, hierbei wird das aktuelle Objekt nicht verändert, sondern ein neuer »BicoPoint« mit den neuen Werten zurückgegeben. Die Funktion »Add« (»Del«) fügt dem aktuellen Objekt ein anderes Objekt vom Typ »BicoPoint« hinzu (bzw. entfernt es), hierbei müssen die Dimensionen übereinstimmen. Mit der Funktion »dot« wird das Vektorprodukt mit einem anderen »BicoPoint« ausgerechnet. Die Länge des »BicoPoints« wird mit der Funktion »length« berechnet, wobei hier die Wurzel des Ergebnis der Funktion »sqLength« zurückgegeben wird, denn diese berechnet die quadrierte 2-Norm des »BicoPoints«. Die letzte wichtige Funktion ist »clone«, denn hier wird eine echte Kopie des aktuellen Objekts erzeugt, diese Funktion ist Teil der Lösung des Referenzproblems, dass ich früher erwähnt habe.

Als nächstes kommt die Klasse »CoresetFeature«, diese Klasse repräsentiert das Quadrupel von 1 auf Seite 5.

In der Klasse »CoresetFeature« (Abbildung 4.4) befinden sich als private Objekte die vier Elemente, die notwendig sind, um ein coreset feature zu bilden. Ein CoresetFeature-Objekt kann man auf vier verschiedene Arten erstellen, als erstes kann man ein leeres Objekt erstellen, hier sind alle privaten Objekte initialisiert aber ohne Wert, die zweite Variante verwendet ein Objekt der Klasse »BicoPoint« und erstellt ein neues coreset feature der Größe eins mit diesem Objekt als Referenzpunkt. Die letzten beiden Varianten erzeugen ein neues Objekt aus einem vorhandenen CoresetFeature-Objekt, dies

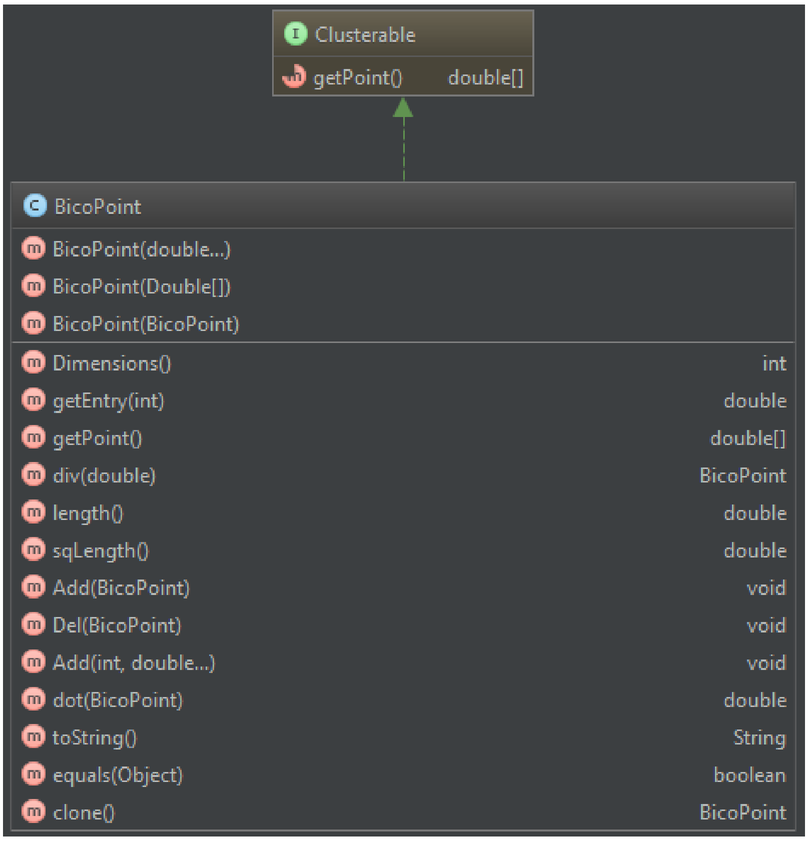


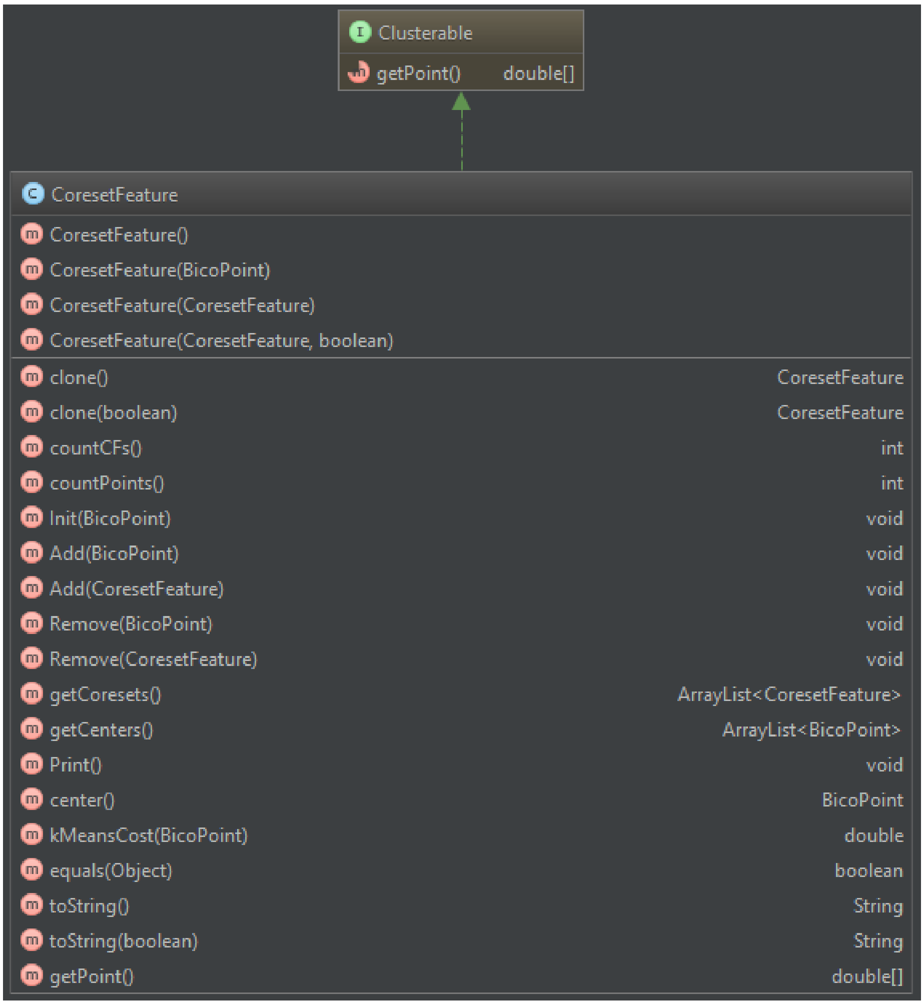
Abbildung 4.3: Klasse »BicoPoint«

sind Kopierkonstruktoren, dabei lässt die erste Varianten, die Kindsmenge von coreset features weg und kopiert diese nicht mit, die andere Variante kopiert diese rekursiv mit in das neue Objekt. Diese Unterscheidung ist notwendig für viele Szenarien, denn um mit dem coreset feature alleine zu rechnen, wird die Kindsmenge nicht benötigt und das Kopieren dieser Menge würde viel Zeit beanspruchen. Die beiden Überladungen der Funktion »clone« sind äquivalente Funktionen dieser Konstruktorvarianten.

Die Funktionen »countCFs« und »countPoints« zählen einmal coreset features und Punkte dieses CoresetFeature-Objekts unter Berücksichtigung der Kindsmenge. Mit der Funktion »getCoresets« wird die Menge an CoresetFeature-Objekten mit Kindsmenge zurückgegeben. Die Funktion »getCenters« gibt eine Menge an »BicoPoint« Objekten mit den jeweiligen Mittelpunkten der coreset features zurück. Der Mittelpunkt eines coreset features wird mit der Funktion »center« berechnet.

Die »Init« Funktion ist wichtig, falls man das Objekt ohne Eingabe erzeugt hat, denn dann funktioniert keine der Berechnungen, die »Init« Funktion stellt desshalb sicher, dass dann eine Exception geworfen wird, denn nur, wenn die »Init« Funktion mit Eingabe aufgerufen wird, werden alle weiteren Möglichkeiten freigeschaltet.

Es besteht mit der »Add« (»Remove«) Funktion die Möglichkeit einen weiteres BicoPoint- oder ein CoresetFeature-Objekt dem coreset feature hinzuzufügen (oder zu entfernen). Ob ein Punkt in das coreset feature aufgenommen wird oder in die Kindsmenge einge-



Clusterable	
getPoint()	double[]

CoresetFeature	
CoresetFeature()	
CoresetFeature(BicoPoint)	
CoresetFeature(CoresetFeature)	
CoresetFeature(CoresetFeature, boolean)	
clone()	CoresetFeature
clone(boolean)	CoresetFeature
countCFs()	int
countPoints()	int
Init(BicoPoint)	void
Add(BicoPoint)	void
Add(CoresetFeature)	void
Remove(BicoPoint)	void
Remove(CoresetFeature)	void
getCoresets()	ArrayList<CoresetFeature>
getCenters()	ArrayList<BicoPoint>
Print()	void
center()	BicoPoint
kMeansCost(BicoPoint)	double
equals(Object)	boolean
toString()	String
toString(boolean)	String
getPoint()	double[]

Abbildung 4.4: Klasse »CoresetFeature«

gliedert wird, entscheidet unter anderem die Funktion »kMeansCost«.

Im Package »generator« gefindet sich eine Hilfsklasse eines Generators, der zufällige Datenvektoren generiert, dabei kann man die Anzahl an Mittelpunkten, deren Koordinanten, die Anzahl der Dimensionen und die Varianz um den Mittelpunkt als Parameter angeben. Dies verwende ich, um einen Datenstrom zu simulieren und damit den Algorithmus zu testen.

In der Klasse »BicoWrapper« (Abbildung 4.5) befindet sich die Logik für das Erstellen der einzelnen coreset features und die Zuweisung neuer Punkte aus dem Datenstrom.

Wie schon erwähnt, benutze ich mehrere Prozessstränge (*Threads*), um die Arbeit der Prozessoren des Streams-Frameworks nicht zu behindern. Dafür hat die »BicoWrapper«-Klasse zwei Listen, in denen ankommende Punkte gespeichert werden. Einmal pro Sekunde wird die Funktion »run« aufgerufen, um die Punkte der Liste abzuarbeiten. Um den Fluss in dieser Zeit nicht zu behindern, wechseln sich die zwei Listen mit dem Vorhalten der Punkte ab. So wird sichergestellt, dass die Klasse »BicoWrapper« weiterhin

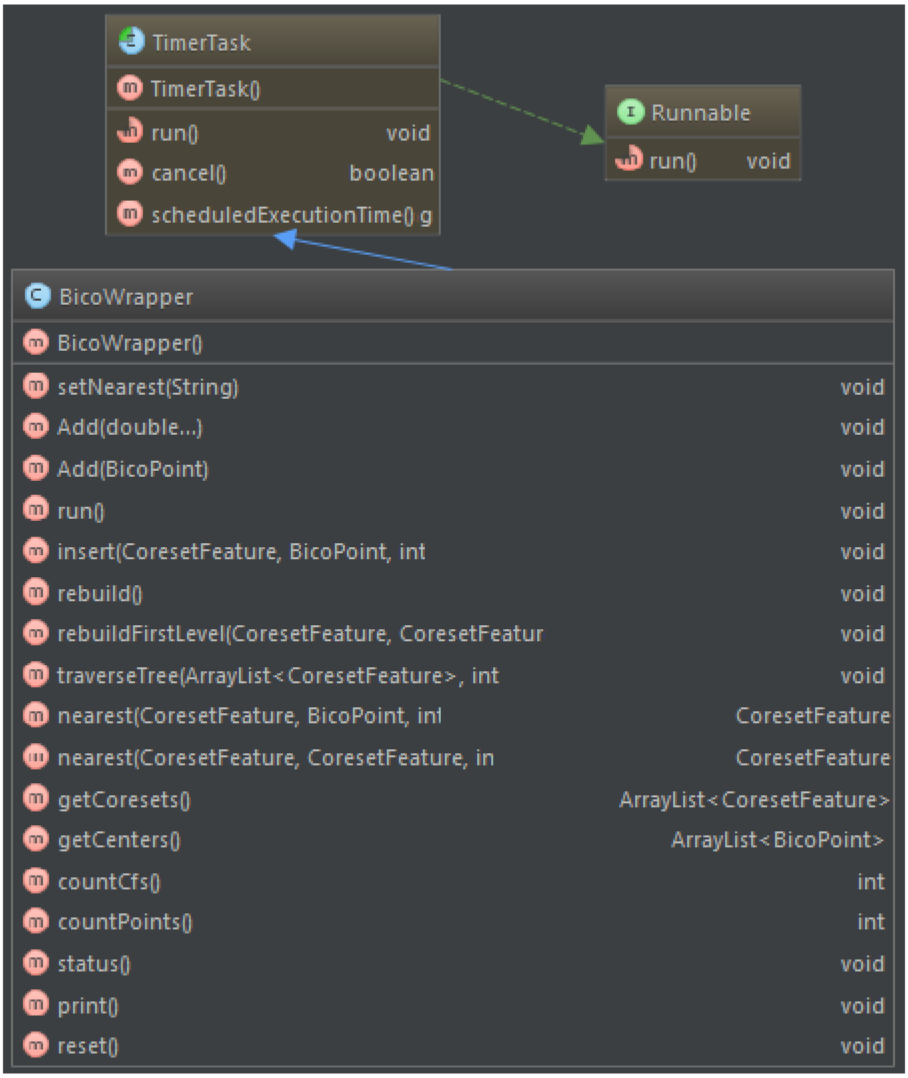


Abbildung 4.5: Klasse »BicoWrapper«

Punkte aufnehmen kann, obwohl der Algorithmus gerade Punkte in die Datenstruktur eingliedert. Weiter benötigt der Algorithmus eine Festlegung der Nachbarschaftssuche. Dafür muss eine Klasse, welche das Interface »INearest« implementiert, initialisiert und der »BicoWrapper«-Klasse zur Verfügung gestellt werden. Dies passiert in der Funktion »setNearest«, wobei diese Funktion alles übernimmt und man ihr nur den Namen der Variante übergeben muss. Falls es keine Implementation mit dem angegeben Namen findet, wird die Standard-Variante ausgewählt, welche aus der naiven Nachbarschaftssuche besteht.

Um der Warteschlange Punkte hinzuzufügen wird die Funktion »Add« aufgerufen, hier gibt es zwei Überladungen, einmal kann ein einfaches »double«-Array übergeben werden, oder ein vorher erzeugtes BicoPoint-Objekt. In der Funktion »run« befindet sie die Logik des Algorithmus, diese Funktion wird nur vom Thread selber und niemals von Benutzer aufgerufen. Ein Punkt wird mit der Funktion »insert« in die Struktur eingegliedert, dabei übergibt man das CoresetFeature-Objekt, welches durch die Nachbarschafssuche gefunden wurde (oder die Wurzel der Struktur), den einzugliedernden Punkt und das Level in dem sich die Eingliederung gerade befindet.

Falls die maximale Anzahl an coreset features überschritten wird, muss die Struktur neu gebildet werden, dies passiert in der Funktion »rebuild«, wobei dies eine Wrapper-Funktion für zwei Funktionen ist, denn die Neubildung besteht aus zwei Schritten, in Schritt eins (Funktion »rebuildFirstLevel«) wird der threshold verdoppelt und es werden mögliche Zusammenführungen von »CoresetFeature« Objekten im Level eins durchgeführt. Mit Schritt zwei (Funktion »traverseTree«) wird durch die ganze Datenstruktur iteriert und, wenn möglich, ein Kindsobjekt mit seinem Elternobjekt zusammengeführt.

Die zwei Überladungen der Funktion »nearest« sind ein Aufruf der Nachbarschaftssuche, mit »BicoPoint« als Parameter wird ein CoresetFeature-Objekt gesucht, das der Entfernung genügt; mit Übergabe eines CoresetFeature-Objekts, wird der Referenzpunkt als Grundlage für die Suche benutzt. Um eine Liste mit allen berechneten coreset features zu bekommen, ruft man die Funktion »getCoresets« auf. Um eine Implementierung zu einem Clustering-Problem auszuführen, werden die Mittelpunkte benutzt, diese liefert die Funktion »getCenters«. Um Statusberichte anzuzeigen, ruft die Funktion »status« die Funktionen »countCfs« und »countPoints« auf.

Um den Algorithmus im Streams-Framework einbinden zu können, habe ich die Klasse »Bico« (Abbildung 4.6) geschrieben.

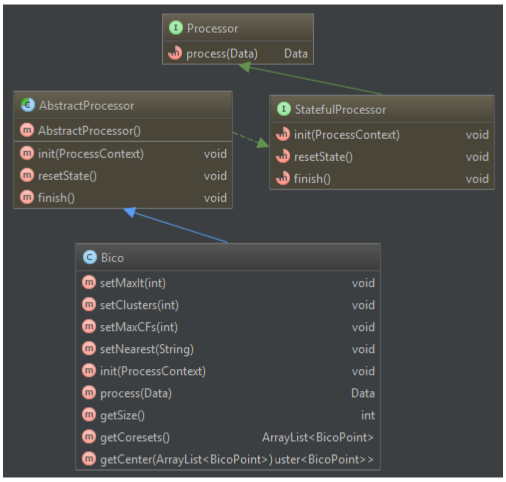


Abbildung 4.6: Klasse »Bico«

Wie man in Abbildung 4.6 sehen kann, erweitert die Klasse »Bico« die vom Streams-Framework stammende Klasse »AbstactProcessor«. Dadurch kann diese Klasse vom Streams-Framework als Prozess benutzt werden. Für die Bildung der Coresetfeature Struktur benötigt der Algorithmus eine maximale Anzahl an Coresetfeatures, dies wird über die Funktion »setMaxCFs« realisiert. Für eine spätere Ausführung einer Zielfunktion, in diesem Fall K-Means, setzten wir die maximalen Interationen (»setMaxIt«) und die Anzahl an zu berechnenden Mittelpunkte (»setClusters«) fest. Welcher Algorithmus zur Nachbarschaftssuche benutzt werden soll, wird mit der Funktion »setNeareset« festgelegt, dieser kaskadiert weiter zur Funktion »setNearest« in der Klasse »BicoWrapper«, zu sehen in Abbildung 4.5.

Paket »clusterer« Als nächstes Kommt das Paket »clusterer« (Abbildung 4.7), in welchem sich Implementierungen verschiedener Clustering-Algorithmen befinden.

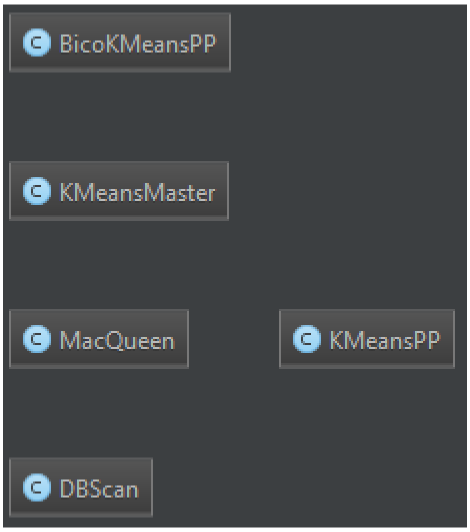


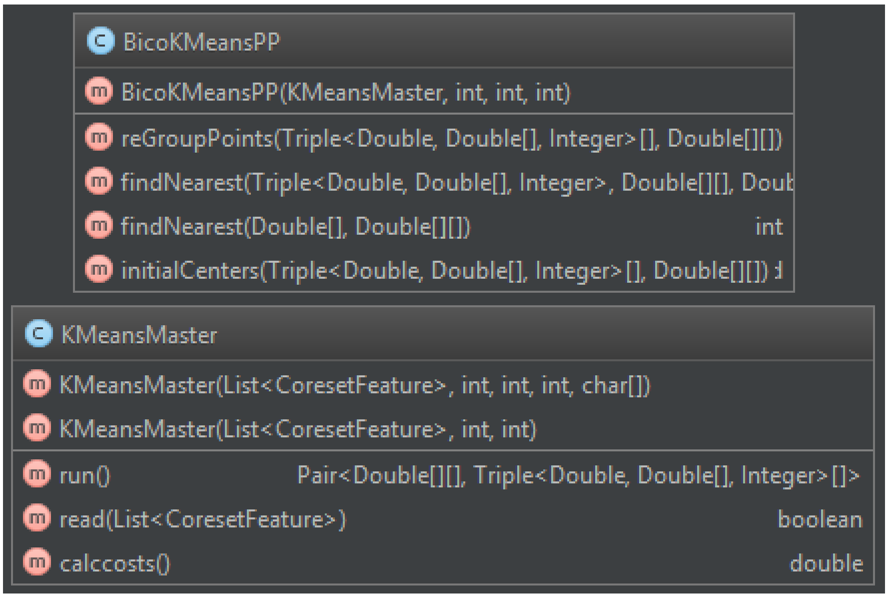
Abbildung 4.7: Paket »clusterer«

Die übernommene Implementierung der Entwickler von BICO von k-means++ befindet sich in der Klasse »BicoKMeansPP« (Abbildung 4.8).

Ich führe in der Abbildung 4.8 noch die Klasse »KMeansMaster« auf, da diese eine Wrapper-Klasse für den eigentlichen Algorithmus ist und dessen Funktionen verwendet, um ein Clustering zu berechnen. Das Tripel hat drei Bestandteile, zuerst die Gewichtung des Punktes, danach den Vektor und zuletzt die Zugehörigkeit zu einem Mittelpunkt. Die Zugehörigkeit wird mit -1 initialisiert.

Der k-means++-Algorithmus funktioniert so, dass er den ersten Punkt als ersten Mittelpunkt wählt und dann randomisiert und unter Berücksichtigung der Entfernung der anderen Punkte zu diesem die nächsten Mittelpunkte wählt. Dies stellt sicher, dass die initialen Mittelpunkte maximale Kosten untereinander garantieren, und dies ist ein Schwerpunkt der k-means-Zielfunktion (Funktion »initialCenters«).

Mit der Funktion »reGroupPoints« werden alle restlichen Punkte im Datensatz den Mittelpunkten zugeordnet, dazu verwendet diese Funktion die Funktion »findNearest«, welche den Index des Mittelpunktes zurückgibt, der am nächsten an dem Punkt ist. Hiernach werden die Mittelpunkte neu berechnet, und falls eine bessere Lösung als zuvor gefunden wurde, wird die Iteration neu begonnen. Im Falle von Konvergenz, d. h., dass sich die Mittelpunkte nicht mehr verändern, ist die Lösung gefunden und wird zurückgegeben.

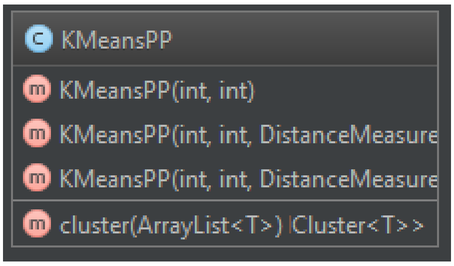


BicoKMeansPP
BicoKMeansPP(KMeansMaster, int, int, int)
reGroupPoints(Triple<Double, Double[], Integer>[], Double[][])
findNearest(Triple<Double, Double[], Integer>, Double[][], Dou
findNearest(Double[], Double[][]) int
initialCenters(Triple<Double, Double[], Integer>[], Double[][])

KMeansMaster
KMeansMaster(List<CoresetFeature>, int, int, int, char[])
KMeansMaster(List<CoresetFeature>, int, int)
run() Pair<Double[][], Triple<Double, Double[], Integer>[]>
read(List<CoresetFeature>) boolean
calccosts() double

Abbildung 4.8: Klasse »BicoKMeansPP«

Eine weitere Implementierung des k-means++ Algorithmus, ist in der Bibliothek von Apache zu finden.



KMeansPP
KMeansPP(int, int)
KMeansPP(int, int, DistanceMeasure
KMeansPP(int, int, DistanceMeasure
cluster(ArrayList<T>) Cluster<T>>

Abbildung 4.9: Klasse KMeansPP

In Abbildung 4.9 ist eine Wrapper-Klasse für diese Implementierung aus der Bibliothek zu sehen. Die Vorgehensweise ist analog zum vorherigen Algorithmus, mit dem Unterschied, dass alle Punkte im Datensatz die selbe Gewichtung haben. Aus diesem Grund, berechnet dieser Algorithmus auf der Kernmenge eine deutlich schlechtere Lösung, als die gewichtetet Implementierung.

Um im späteren Verlauf die Daten der Struktur abzurufen, ruft man die Funktionen »getCoresetes« oder »getCenter« auf, je nach Anwendung, benötigt man die kompletten coreset features oder nur deren Mittelpunkte.

Für Untersuchungen hinsichtlich Qualität und Laufzeit habe ich eine weiter Implementierung der k-means-Zielfunktion vorgenommen, diese basiert auf einem einfachen Prinzip und hat eine Laufzeit von $O(n)$, denn es wird nur ein einziges Mal über den Datensatz iteriert. Die Qualität der Lösung hängt sehr stark davon ab, in welcher Reihenfolge die Punkte vom Algorithmus verarbeitet werden. Dieser Algorithmus ist von

MacQueen [13] entwickelt worden und nach seinem Entwickler benannt.

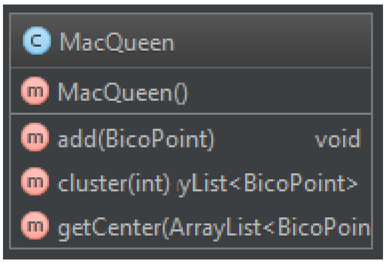

MacQueen
MacQueen()
add(BicoPoint) void
cluster(int) yList<BicoPoint>
getCenter(ArrayList<BicoPoin

Abbildung 4.10: Klasse »MacQueen«

Das Vorgehen dieses Algorithmus legt die ersten k Punkte des Datensatzes als Mittelpunkte fest und fügt dann darauf folgende Punkte dem Mittelpunkt zu, der dadurch geringste Varianzänderung erfährt, danach wird der Mittelpunkt neu gesetzt und es wird mit dem nächsten Punkt fortgefahren. Die Anzahl der Mittelpunkte wird mit der Funktion »cluster« festgelegt. Die Berechnung der Mittelpunkte findet durch die Funktion »getCenter« statt.

Da in dieser Arbeit untersucht werden soll, ob die Kernmenge, die für k-means berechnet wurde, auch für DBScan verwendet werden kann, habe ich eine weitere Wrapper-Klasse (Abbildung 4.11) geschrieben, diesmal für die Apache-Implementierung von DBScan.

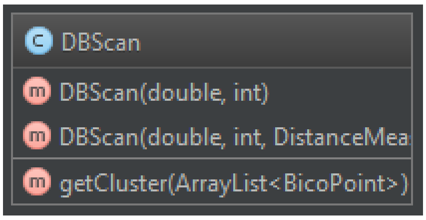

DBScan
DBScan(double, int)
DBScan(double, int, DistanceMea
getCluster(ArrayList<BicoPoint>)

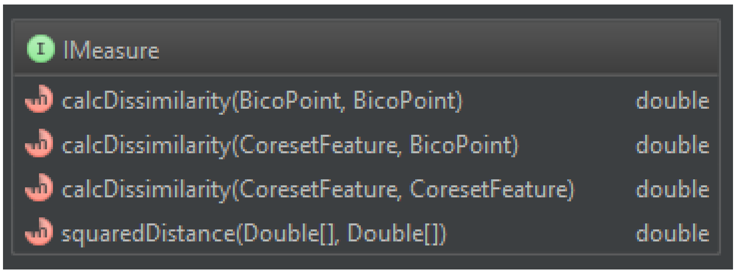
Abbildung 4.11: Klasse »DBScan«

Die Klasse ist einfach gehalten, da nicht mehr benötigt wird, als ein Mindestabstand ϵ, so dass zwei Punkte nur dann als zusammengehörig erkannt werden, wenn ihr Abstand kleiner als ϵ ist. Außerdem eine Zahl m, so dass eine Punktmenge, die in Reihe maximal den Abstand ϵ habe, mindestens die Größe m haben muss. Wichtig ist außerdem, dass man dasselbe Distanzmaß nimmt, welches man auch zum Berechnen der Kernmenge benutzt hat.

Paket »measure« Die Ähnlichkeit zweier Punkte kann verschieden aufgefasst werden, daher gibt es verschiedene Varianten. Die bekanntesten sind hier die Euklidische Norm oder die Manhattan-Distanz.

Für diesen Algorithmus wurde in der Kernmengen-Berechnung und für k-means die quadrierte 2-Norm angewendet, daher gibt es hierzu eine Implementation. Es ist dem

Anwender aber freigestellt, welches Distanzmaß er verwendet will, er kann weitere entwickeln und benutzen. Dazu muss die Klasse nur das Interface »IMeasure« (Abbildung 4.12) implementieren.



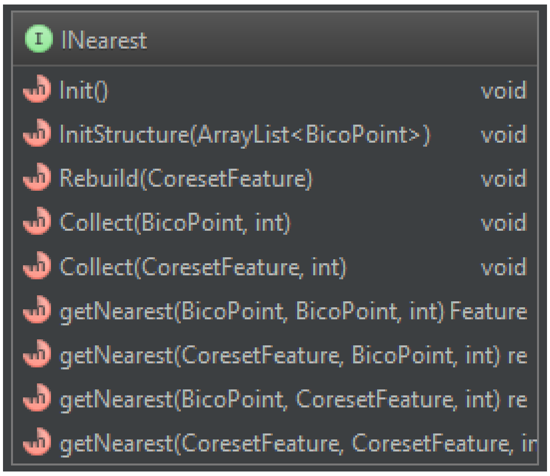
IMeasure

calcDissimilarity(BicoPoint, BicoPoint)	double
calcDissimilarity(CoresetFeature, BicoPoint)	double
calcDissimilarity(CoresetFeature, CoresetFeature)	double
squaredDistance(Double[], Double[])	double

Abbildung 4.12: Interface »IMeasure«

Dieses Interface bietet verschiedene Varianten zum bestimmen der Ähnlichkeit (Unähnlichkeit).

Paket »nearest« Für die Berechnung der Kernmenge muss eine Nachbarschaftssuche vollzogen werden, auch hier gibt es verschiedene Möglichkeiten. Die einfachste ist der *Nearest-neighbour*-Algorithmus, dieser sucht in einer Menge anhand eines Distanzmaßes den nächsten Punkt zu einem Eingabepunkt. Eine weiter Variante ist Filtering, dieser ist analog zum »NearestNeighbour«, jedoch wird die Suche auf eine Dimension beschränkt; dadurch spart man sehr viel Ressourcen und gewinnt damit Zeit, leider büßt man umso mehr Qualität ein, je mehr Dimensionen ein Punkt hat.



INearest

Init()	void
InitStructure(ArrayList<BicoPoint>)	void
Rebuild(CoresetFeature)	void
Collect(BicoPoint, int)	void
Collect(CoresetFeature, int)	void
getNearest(BicoPoint, BicoPoint, int)	Feature
getNearest(CoresetFeature, BicoPoint, int)	re
getNearest(BicoPoint, CoresetFeature, int)	re
getNearest(CoresetFeature, CoresetFeature, ir	

Abbildung 4.13: Interface »INearest«

Das Interface »INearest« (Abbildung 4.13) legt die Rahmenbedingungen für eine Nachbarschaftssuche fest. Der Kern liegt in der Funktion »getNearest«, jedoch habe ich Vorbereitungen getroffen um jede Art von Nachbarschaftssuche zu ermöglichen, denn im Algorithmus sind die Funktionen »InitStructure«, »Collect« und »Rebuild« an den richtigen Stellen verankert. So kann man z.B. auch eine Nachbarschaftssuche wie Projektion realisieren und muss dazu nur die Projektion selbst implementieren und muss den anderen Programmcode nicht anfassen.

Paket »utils« Viele der vorgestellten Klassen müssen immer wieder die selben Funktionen aufrufen und verschiedene Instanzen von Objekten teilen sich Werte. Um dies so einfach und effizient wie möglich zu realisieren, habe ich im Paket »utils« Hilfsklassen geschrieben und Methoden wie **Caching** benutzt um Ressourcen zu sparen.

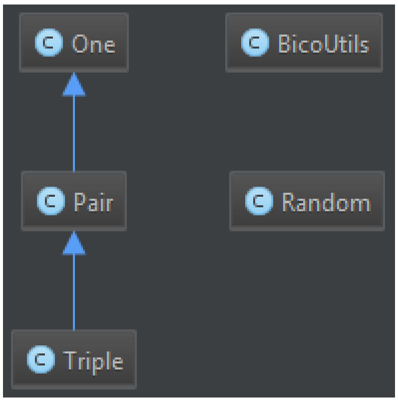


Abbildung 4.14: Paket »utils«

Die Klassen »One«, »Pair« und »Triple« bilden eine Objekt Hierarchie zum Speichern von Objekten mit mehreren Einträgen ab (Abbildung 4.15).

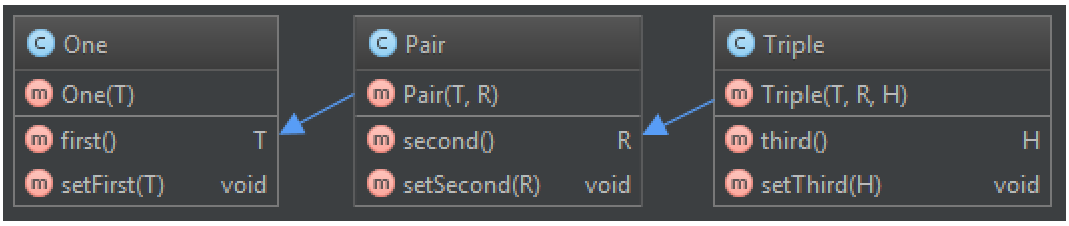


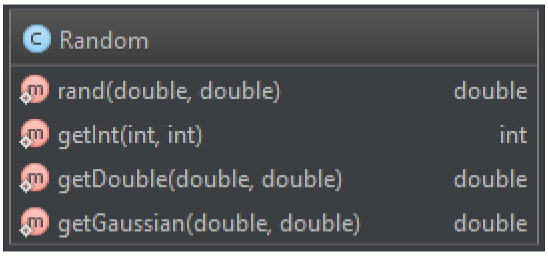
Abbildung 4.15: Klassen »One«, »Pair«, »Triple«

In der Klasse »Random« (Abbildung 4.16) bieten statische Funktionen an, welche in Zahlenintervallen Zufallszahlen berechnet.

In der Klasse »BicoUtils« (Abbildung 4.17) befinden sich die geteilten Objekte und Hilfsfunktionen für den Algorithmus.

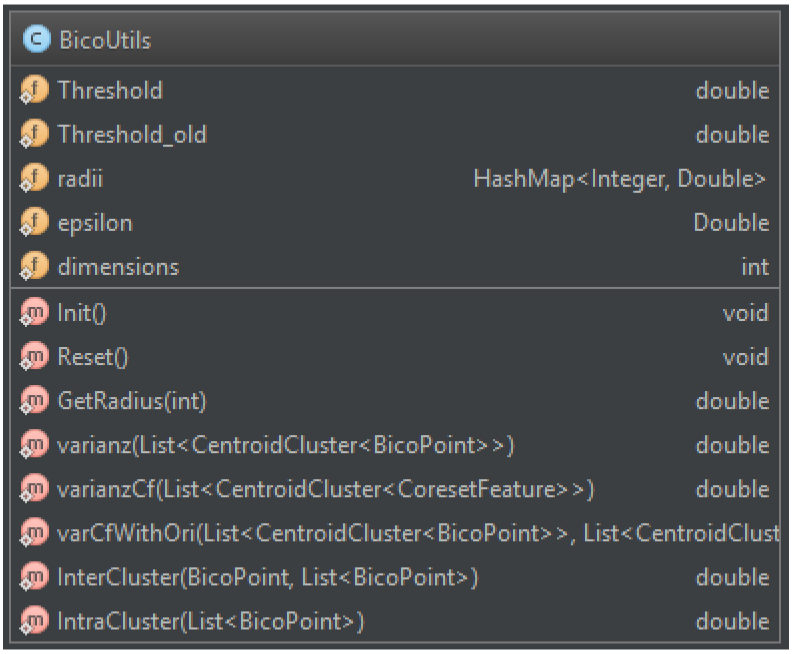
In der Berechnung der Kernmenge wird der Radius benötigt um zu entscheiden, ob ein Datum in ein coreset feature hinzugefügt wird, oder in eine Kindsmenge dieser coreset features, und ob ein neues coreset feature in der Kindsmenge oder im Level 1 erzeugt wird. Der Radius berechnet sich direkt aus dem threshold.

Der threshold ist ein Wert und wird oft abgerufen, hierfür muss nichts berechnet werden, der Radius hingegen muss für jedes Level berechnet werden. Daher benutzte ich



Random	
rand(double, double)	double
getInt(int, int)	int
getDouble(double, double)	double
getGaussian(double, double)	double

Abbildung 4.16: Klasse »Random«



BicoUtils	
Threshold	double
Threshold_old	double
radii	HashMap<Integer, Double>
epsilon	Double
dimensions	int
Init()	void
Reset()	void
GetRadius(int)	double
varianz(List<CentroidCluster<BicoPoint>>)	double
varianzCf(List<CentroidCluster<CoresetFeature>>)	double
varCfWithOri(List<CentroidCluster<BicoPoint>>, List<CentroidClust	
InterCluster(BicoPoint, List<BicoPoint>)	double
IntraCluster(List<BicoPoint>)	double

Abbildung 4.17: Klasse »BicoUtils«

hier eine »HashMap«, um schon berechnete Radien zu memoisieren (*cachen*). Dies spart Ressourcen beim Berechnen der Radien. Wenn sich jedoch der threshold ändert, müssen alle gecachten Radien verworfen werden, dazu speichere ich den aktuellen threshold zweimal (»Threshold« und »Threshold_old«), falls sich der threshold beim Neubilden ändert, sind die Variablen »Threshold« und »Threshold_old« unterschiedlich und die Funktion »GetRadius« weiß, dass die gecachten Radien veraltet sind und löscht diese, darauf hin werden die Radien mit dem neuen threshold neu gecacht.

Für qualitative Auswertungen bietet die Klasse Funktionen für die Varianzberechnung und Inter- sowie Intraclustervarianz an.

4.1.1 Wrapper und Streamsframework

Die Konvention des Umgangs mit Daten im Streams-Frameworks ist, dass das Ursprungs-Datum nicht verändert wird. Das bedeutet, ein Datum gelangt genauso aus einem Prozess heraus wie es hineingegangen ist. Dies stellt sicher, dass mehrere unterschiedliche

Prozesse unabhängig auf einem Datenstrom arbeiten können und die Daten verarbeiten. Dieses Vorgehen habe ich im BICO-Algorithmus adaptiert und lasse zudem die Arbeit des Algorithmus in einem extra Thread laufen, so gelangt das Datum schnell aus dem eigentlichen Prozess heraus und kann weiter benutzt werden. So werden jede Sekunde alle gesammelten Daten in die Struktur eingearbeitet. Während dessen können weitere Daten gesammelt werden, diese werden beim nächsten Auslösen abgearbeitet.

4.2 Implementierung der Zielfunktion

Es gibt in sehr vielen Frameworks schon vorhandene Implementationen für die k-means Zielfunktion, leider gibt es nur in der Bibliothek von Apache für Java die k-means++ Variante und außerdem wird für eine kernmengen-basierte Berechnung eine Abwandlung des Algorithmus benötigt, da die einzelnen coreset features in der Kernmenge gewichtet sind und dies die Optimierung der Zielfunktion verändert, da nicht jedes coreset feature gleichviel Kosten zur Varianz beiträgt.

Daher bin ich dazu übergegangen und habe die k-means++-Implementierung für Kernmengen selber vorgenommen. Als Vorlage diente mir dazu die C++-Implementierung von den Erstellern dieses Algorithmus.

Dass eine Gewichtung der coreset features im k-means-Algorithmus unumgänglich ist, habe ich erkannt, als ich k-means-Implementierungen erst auf die originalen Daten und danach auf die Kernmenge angewendet habe. Eine der Implementierungen war die der Ersteller des BICO-Algorithmus. Die vorhandenen Implementierungen (Apache, WEKA, Rapidminer) lieferten auf der Kernmenge immer eine deutlich schlechtere Lösung, im Vergleich mit den Originaldaten, als der gewichtete Algorithmus (die Gewichtung war mir zu dem Zeitpunk nicht bewusst). Dies führte zu der Annahme, dass dieser Algorithmus anders funktioniert. Beim Betrachten des Programmcodes ist mir die Gewichtung aufgefallen und ich habe eine eigene Implementierung vorgenommen und die Gewichtung mit einbezogen. Die neuen Ergebnisse mit diesem Algorithmus waren deutlich besser und stimmten mit den Ergebnissen des Algorithmus der Ersteller von BICO überein.

Trotzdem möchte ich verschiedene Frameworks und Implementierungen miteinander vergleichen um Ausführungszeiten zu analysieren, die Qualität der Lösung spielt bei dieser Untersuchung keine Rolle. Die vorgestellten Werte sind Mittelwerte von mindestens 100 Durchläufen mit geschüttelten Eingangsdaten der Algorithmen.

4.2.1 Rapidminer

Rapidminer ist eine Toolbox mit sehr vielen verschiedenen Werkzeugen für Diverses. Unter anderem gibt es Implementierungen von der k-means Zielfunktion. Eine Implementation von DBScan liegt ebenfalls vor.

4.2.2 WEKA

Auch Weka ist eine Toolbox von verschiedenen Werkzeugen. Doch hier liegt der Fokus darauf, dass die Werkzeuge im Programmcode verwendet wird. Leider muss man auch hier die komplette Bibliothek einbinden.

4.2.3 Apache

Apache bietet ein Vielzahl von Bibliotheken an. Unter anderem eine für die Berechnung der Zielfunktion k-means. Die einzelnen Bibliotheken können unabhängig voneinander eingebunden werden und die Abhängigkeiten sind auch modular gehalten.

	Rapidminer	WEKA	Apache	BICO-k-means
k-means	3.0s	2.09s	0.131s	0.125s

Tabelle 1: Vergleich der Ausführungszeit verschiedener Implementationen

Die Evaluation hat mit einer Datenmenge von $n = 100000$*, einer Clustergröße* $k = 3$ *und von maximal* $it = 20$ *Iterationen statt gefunden. Die Qualitäten der Lösungen weichen kaum voneinander ab.*

Ich konnte beobachten, dass die Iterationsbegrenzung eine starke Auswirkung auf die Implementationen von Rapidminer und WEKA hat, denn bei kleinem it, wird die Ausführungszeit auch kleiner, dabei sinkt die Qualität der Lösung aber stark. Vergrößert manit, steigt die Qualität der Lösung, aber auch die Ausführungszeit.

Die Apache-Implementation braucht im Schnitt deutlich weniger Iterationen zur Lösung, als andere Implementation.

4.3 Stadien der Entwicklung

Während der Entwicklung gab es Stadien der Entwicklung. Das erste Stadium war eine rudimentäre Implementation der mathematischen Definition ohne deren Hilfsfunktionen. Dort waren Radius und threshold festgesetzt. Außerdem gab es noch keinen Neubildungsalgorithmus (Abbildung 4.18). Hier kann man sehen, dass die Mittelpunkte der coreset features sich nur im Mittelpunkt der Punktwolken befinden, dies kommt daher, dass die Nachbarschaftssuche und damit verbunden der Radius festgesetzt sind. Außerdem fehlt der Neubildungs-Teil.

Am nächsten Beispiel kann man sehen, dass der Radius fest gesetzt ist und es keine Neubildung gibt. Denn die Coresetfeatures sind um den Mittelpunkt der Punktwolken fokussiert. Dieses Verhalten wird durch die festen Parameter noch unterstützt (Abbildung 4.19).

Die Weiterentwicklung bestand darin, jetzt den Radius, wie mathematisch vorgeschrieben, zu implementieren. Außerdem ist jetzt die Neubildung bei zu vielen coreset features aktiv. Mit den errechneten coreset features wurden die Mittelpunkte berechnet. Einmal aus den Rohdaten, einmal aus den coreset features und einmal durch eine BICO-Implementation der Verfasser des Algorithmus.

Die Punkte aus dem Datensatz »km_out1.txt« sind dabei Mittelpunkte aus Coresets die der Algorithmus dieser Bachelorarbeit berechnet hat. Im Datensatz »km_out2.txt« sind die Mittelpunkte vom Algorithmus der Verfasser des Algorithmus berechnet worden.

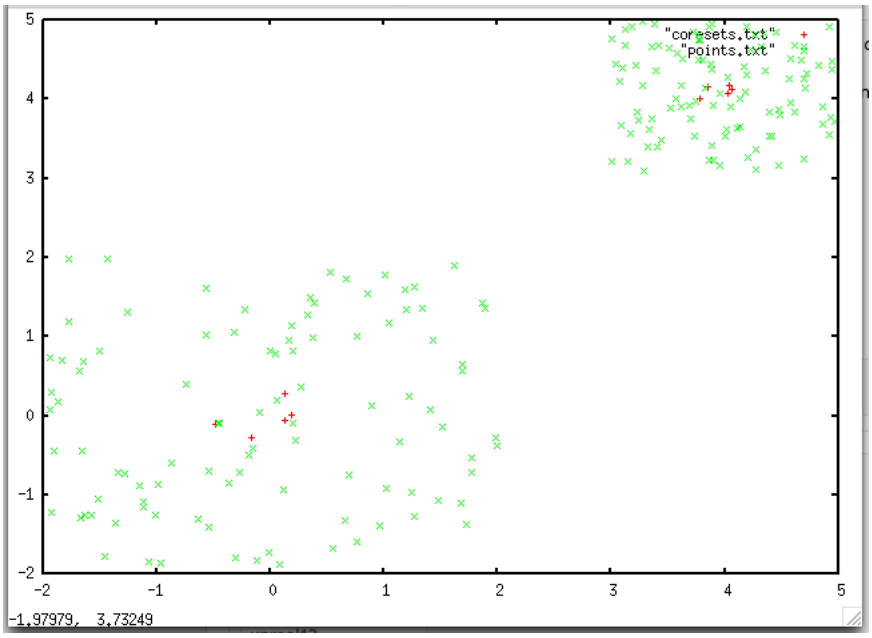


Abbildung 4.18: Bico Test 1

Und im Datensatz »km_out3.txt« sind die Mittelpunkte von einer k-means++-Implementation auf dem Originaldatensatz berechnet worden. Jede BICO-Implementation hat neun Coresets berechnet und aus den neun Coresets wurden die Mittelpunkte durch die k-means++-Implementation berechnet.

Das bedeutet die Mittelpunkte wurden aus einer Menge berechnet, die kleiner als 10 % des Ursprungsdatensatzes ist. Erhöht man die maximale Anzahl der coreset features, so nähern sich die Mittelpunkte denen der Originaldaten an (Abbildung 4.20).

An dieser Stelle ist der Algorithmus fertig und die Berechnung der coreset features ist erfolgreich.

Als nächstes habe ich zwei andere Varianten zur Nachbarschaftssuche implementiert, die beiden Varianten dabei sind zuerst **Filtering**, dabei wird die Nachbarschaftssuche auf nur eine Dimension begrenzt, der Gewinn hierbei ist, dass die Zeit für die Berechnung des Abstandes zweier Datenpunkte atomar ist, da nur eine Dimension verglichen wird. Es ist logisch, dass hierbei große Fehler passieren können, da zwei Punkte in einer Dimension weit auseinander liegen können, in anderen Dimensionen aber sehr nah beieinander. Je mehr Dimensionen ein Datum hat, desto größer wird der durch Filtering erreichte Ressourcengewinn und gleichzeitig auch der Fehler.

Die zweite Variante ist das **Hashing**, hierbei wird der Raum, in dem sich der Datensatz befindet, in Abschnitte unterteilt und Punkte in den Abschnitten durch einen Referenzpunkt abgebildet. Somit suchen wir nach dem nächsten Referenzpunkt im Raum um den nächsten Nachbarn zu suchen. Hauptgewinn ist, dass sich die Suche auf einen kleinen Teil der Daten beschränkt. Auch hier kann man den Grad des Fehlers bestimmen, denn je feiner die Unterteilung ist, desto größer der Fehler. Wenn große Areale abgedeckt werden, ist der Fehler kleiner.

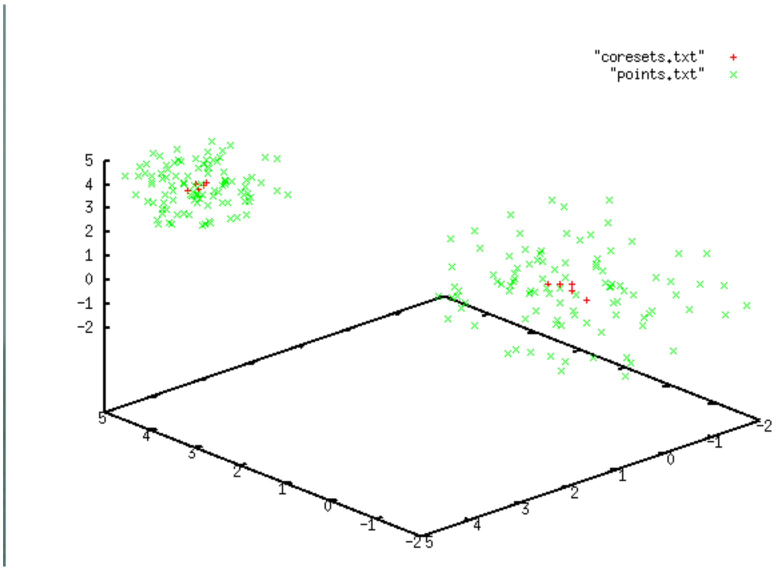


Abbildung 4.19: Bico Test 2 (feste Parameter, keine Neubildung)

5 Evaluation

Zur Evaluation verwende ich zuerst den selben Covertype-Datensatz, den auch die Entwickler des Algorithmus verwendet haben, um die Ergebnisse vergleichen zu können.

Der MacQueen-k-means-Algorithmus ist sehr von der Reihenfolge der Daten im Datensatz abhängig; um zu testen, wie robust BICO ist, werde ich hier einen Datensatz mit 100 unterschiedlichen Reihenfolgen der Daten untersuchen.

Danach wende ich DBScan mit Kernmenge und ohne auf verschiedene »Two Moon«-Datensätze an, dabei wird das Experiment einmal mit und einmal ohne Rauschen durchgeführt.

5.1 Nachbarschaftssuchen

Die zwei implementierten weiteren Nachbarschaftssuchen (neben der naiven Nachbarschaftssuche), sind Filtering und Hashing. Dazu werden auf dem Iris-Datensatz 10 Durchläufe gemacht und die Varianzen der Ergebnisse mit der Varianz auf den Originaldaten verglichen, hierbei ist die maximale Anzahl an coreset features auf 20 Begrenzt. Außerdem untersuche ich Selbiges auf einem von mir generierten Datensätze mit 500000 Punkten und 5 Mittelpunkten, die Anzahl der Dimensionen steigt dabei pro Datensatz. Die coreset features sind auf 5000 beim Filtering und 50000 beim Hashing begrenzt, der Unterschied resultiert daraus, dass Hashing bei weniger coreset features schlechter wird, da pro die Anzahl der Abschnitte aus der maximalen Anzahl an coreset features berechnet wird und dadurch die Menge an Punkten pro Abschnitt sinkt. Daraus resultieren sehr viele Neubildungen der Struktur, was wiederum in einer höheren Laufzeit resultiert.

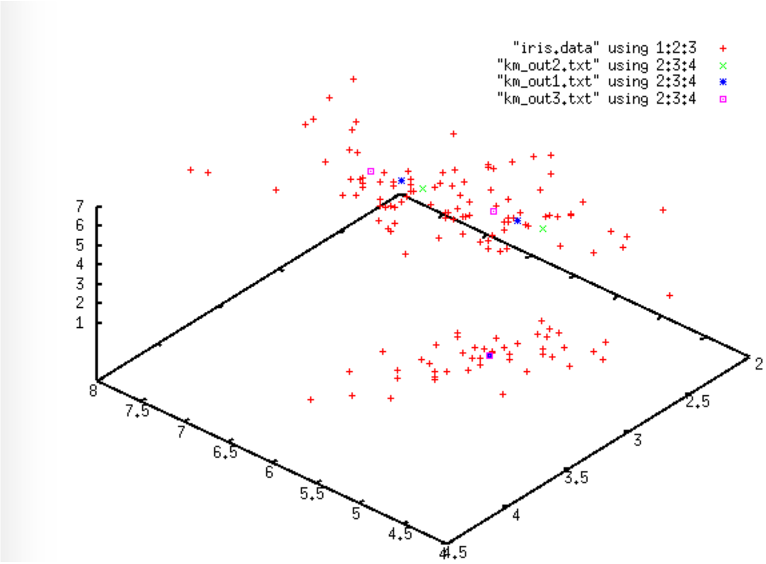


Abbildung 4.20: Bico Test 3 (dynamischer Radius, mit Neubildung)

5.1.1 Filtering

Wie schon erwähnt reduziert Filtering die Nachbarschaftssuche auf eine Dimension. Dadurch wird die Berechnung der Entfernung zweier Punkte stark beschleunigt.

In Tabelle 2 sind die Durchschnittswerte aus den Läufen zu sehen.

Datensatz (Dimensionen)	Bico (ms)	k-means++ (ms)	k-means++ (ms)	Koeffizient
Iris (4)	14.5	2.4	23	0.588
Eigener (2)	3584.3	120.3	26971	0.964
Eigener (4)	4799.4	122.3	33055	20.631
Eigener (10)	6680.3	206.3	49243	0.986

Tabelle 2: Filtering Untersuchung (Kosten der optimalen Lösung sind immer gleich)

5.1.2 Hashing

Beim Hashing wird der Raum in Abschnitte unterteilt und ein Punkt sucht nur in seinem eigenen Abschnitt nach einem Nachbarn. Daher reduziert sich die in der zu suchenden Menge.

Auch hier in Tabelle 3 die Durchschnittswerte der Läufe.

Wie man in den beiden Tabellen 2 und 3 sehen kann, sind die Zeitersparnisse sehr groß und die Qualität der Lösung sehr gut.

Datensatz (Dimensionen)	Bico (ms)	k-means++ (ms)	k-means++ (ms)	Koeffizient
Iris (4)	16.9	5.4	31.5	0.984
Eigener (2)	2774.6	2154.5	26971	0.999
Eigener (4)	3702.9	3148.8	29608	1.0
Eigener (10)	5995	4656.1	46176	1.0

Tabelle 3: Hashing Untersuchung (Kosten der optimalen Lösung sind immer gleich)

5.2 Covertype

Der Covertype Datensatz [11] besteht aus $n = 581012$ Vektoren mit jeweils $d = 55$ Dimensionen. Dieser Datensatz wurde schon mit BICO untersucht, ich möchte herausfinden ob meine Implementierung dieselben Ergebnisse, heißt die selben Kosten, liefert.

Dafür sind zehn, in der Reihenfolge randomisierte, Datensätze erzeugt worden und zu diesen wurde jeweils eine Kernmenge mit einer maximalen Größe von 50000 berechnet. Als Nachbarschaftssuche wurde die Hashing-Variante gewählt, da eine naive Nachbarschaftssuche bei so großen Datensätzen zu viel Ressourcen benötigt. Außerdem haben die Ersteller dieses Algorithmus auch eine Art Hashing verwendet und die naive Nachbarschaftssuche zu lange dauern würde.

In Tabelle 4 sieht man 10 Durchläufe auf Covertype mit randomisierten Reihenfolgen. Der Kosten-Koeffizient bildet dabei ab, wie gut der Algorithmus, der im Rahmen dieser Bachelorarbeit programmiert wurde, im Vergleich zum Algorithmus der Ersteller von BICO ist, auf dem jeweiligen Durchlauf, ist.

Das Ergebnis dieser Untersuchung lautet, dass der hier entwickelte Algorithmus nahezu die selben Ergebnisse liefert, wie der von den Entwicklern von BICO. Die Unterschiede rühren daher, dass die Entwickler von BICO Projektion benutzen und diese randomisiert ist.

Durchlauf	BICO (s)	k-means++ (s)	Kosten-Koeffizient
1	273.717	48.608	0.988
2	288.616	46.323	0.995
3	245.658	44.167	0.905
4	203.797	45.655	0.887
5	193.451	43.648	0.837
6	209.233	50.778	0.861
7	240.808	45.347	0.844
8	213.078	51.573	0.857
9	244.938	51.526	0.833
10	264.052	37.999	0.832

Tabelle 4: BICO auf Covertype

5.3 Reihenfolge (BICO gegen MacQueen)

In welcher Reihenfolge die Daten vom MacQueen-Algorithmus verarbeitet werden, hat einen sehr großen Einfluss auf die Qualität der Lösung. Daher wird jetzt anhand vom Iris-Datensatz [11] mit 100 unterschiedlichen Reihenfolgen untersucht, wie sich die Kernmengen der unterschiedlichen Reihenfolgen unterscheiden. Als Referenz wird dazu auf jeweils denselben Datensätzen der MacQueen-Algorithmus ausgeführt, um zu verdeutlichen, welche Auswirkung unterschiedliche Reihenfolgen haben können, der Durchschnitt aus 10 Durchläufen sind in Abbildung 5.1 zu sehen.

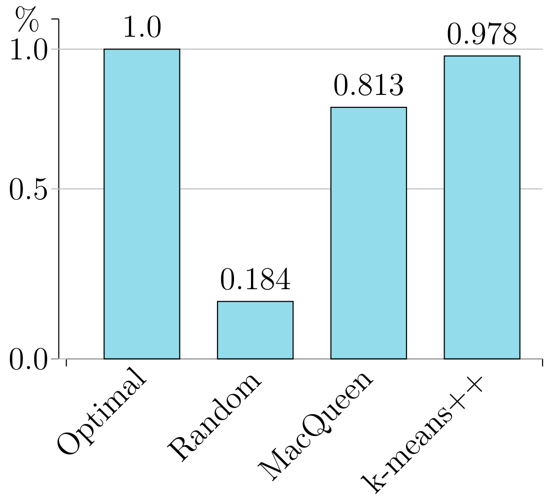


Abbildung 5.1: Kosten-Koeffizienten verschiedener Algorithmen

Die Experimente haben gezeigt, dass die Reihenfolge beim MacQueen-Algorithmus eine große Rolle spielt und die Ergebnisse sich deutlich voneinander unterscheiden, von sehr hohen Kosten (Maximal: 94.5), bis hin zur Lösungen eines k-means++ Algorithmus (Kosten: 48.38).

Die Experimente haben außerdem gezeigt, dass auch BICO abhängig von der Reihenfolge ist, dies prägt sich sich nicht so stark aus wie beim MacQueen-Algorithmus, jedoch sind die berechneten Mittelpunkte aus Kernmengen verschiedener Reihenfolgen signifikant unterschiedlich. Sogar die Kernmengen an sich unterscheiden sich, es gibt Redundanzen, jedoch sind große Teile unterschiedlich. Auch die berechneten Mittelpunkte aus den Kernmengen unterscheiden sich. Minimale Kosten aus den Kernmengen lagen bei 48.42 und ein Maximum der Kosten ergab einen Wert von 51.33, damit ist die Abweichung nicht so groß wie bei MacQueen, jedoch wurde keine optimale Lösung berechnet.

5.4 Two Moons

Für ein dichte basiertes Clustering ist die beste Trainingsmethode, den »Two Moons«-Datensatz zu verwenden, denn dieser besteht aus zwei ineinander verschlungenen Halbmonden. Die

Untersuchung findet einmal auf dem Datensatz mit Grundrauschen statt und einmal ohne Rauschen, dies bedeutet nur die zwei Halbmonde.

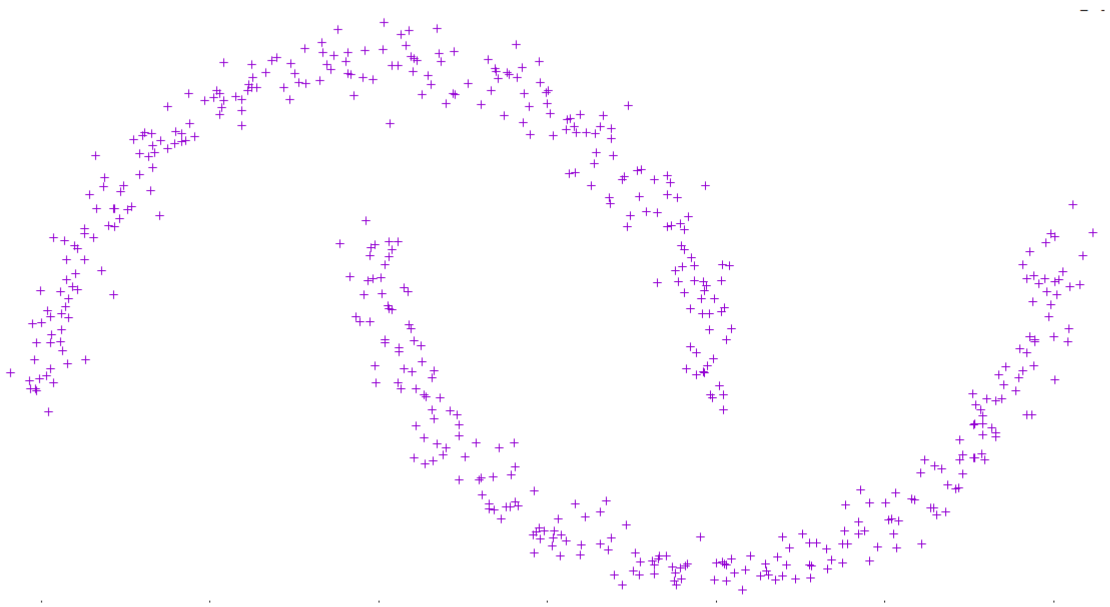


Abbildung 5.2: »Two Moons«

5.4.1 Ohne Rauschen

Die Untersuchung ohne Rauschen soll sicherstellen, dass überhaupt Strukturen gefunden werden können. Für die Evaluierung verwende ich als ϵ den Radius des ersten Levels in verschiedenen Multiplikationen.

Es wurden 10 Versuche durchgeführt und die Struktur wurde zuverlässig gefunden.

5.4.2 Mit Rauschen

Mit der Untersuchung mit Grundrauschen im Datensatz, möchte ich feststellen, ob der Algorithmus in einem realistischen Datensatz die gesuchten Strukturen erkennt. Auch hier wird für das ϵ der Radius des ersten Levels in verschiedenen Multiplikationen verwendet.

Wichtig ist, dass Rauschen als solches erkannt wird und keinem Cluster zugeordnet wird, dafür verwende ich den selben Datensatz, nur es wurde Rauschen hinzugefügt. Wenn hier die selbe Konfiguration angewendet wird, wie bei der Untersuchung ohne Rauschen, erzielt man kein Ergebnis, denn das Rauschen wird in der Kernmenge auch abgebildet. Das bedeutet, man benötigt eine deutlich größere Kernmenge um gutes Ergebnis zu erhalten.

Das Ergebnis auf dem Datensatz mit Rauschen und 100 coreset features sieht man in Abbildung 5.6.

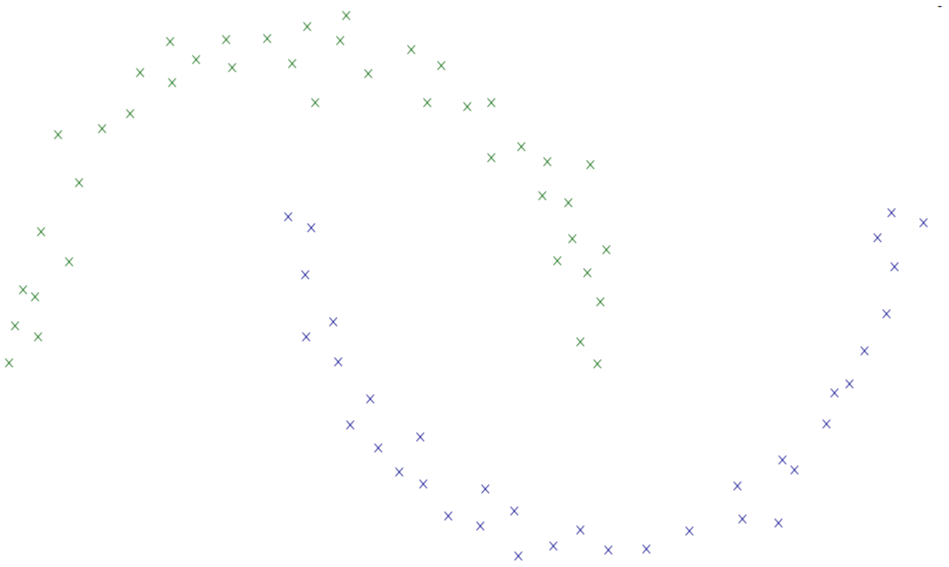


Abbildung 5.3: Ohne Rauschen mit 100 coreset features

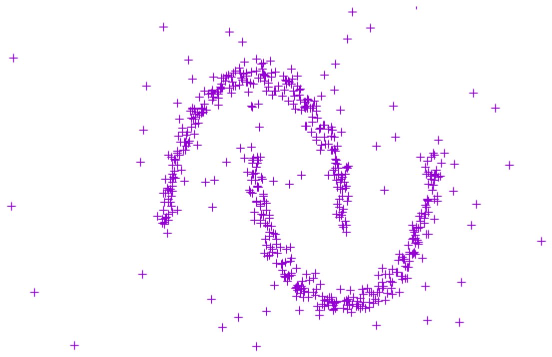


Abbildung 5.4: »Two Moons« mit Rauschen

Da für den Datensatz maximal 100 coreset features erlaubt sind und der Datensatz aus 600 Punkten besteht (davon 100 Punkte Rauschen), sind die coreset features alleine mit dem Rauschen schon aufgebraucht. Wie man sehen kann, sind die zwei Halbmonde in keiner Weise mehr zu erkennen.

5.5 Auf dem Smartphone

Eine Anforderung an die Entwicklung ist, dass dieser auf einem Android Smartphone lauffähig ist. Tests hat gezeigt, dass dem so ist. Leider mit ein paar Einschränkungen, denn ein Smartphone hat weniger Ressourcen zur Verfügung als ein Desktop Computer. Die einfachen Tests mit dem Iris-Datensatz haben gezeigt, dass alle programmierten Algorithmen funktionieren, sowie Hashing und Filtering. Die Ausführungszeit sind natürlich erheblich länger.

Im der beigefügten DVD befinden sich, im Ordner »OnAndroid«, Log-Dateien, in

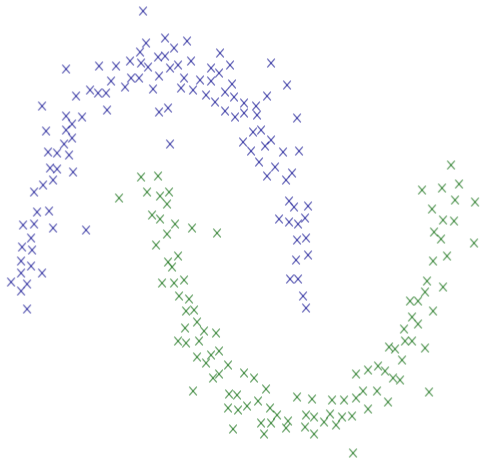


Abbildung 5.5: Rauschen mit maximal 300 coreset features

denen die Ausgabe des Algorithmus auf einem Android-Smartphone gespeichert sind.

6 Fazit

Dieser Algorithmus ist eine große Innovation für das Cluster-Problem, da sehr große Datenmengen effizient berechnet werden können und der Fehler beschränkbar ist. Außerdem ist der Algorithmus durch die Einbindung in das Streams-Framework sehr einfach und komfortabel auf einen Stream aufsetztbar. Die Rahmenbedingungen lassen es außerdem zu, dass dieser Algorithmus so auf einem Android-Smartphone lauffähig ist und das Streams-Framework bietet von Haus aus an, Ausgaben des Smartphones abzurufen. Somit ist dem Anwender freie Hand gelassen, wie er ein Smartphone oder was er analysieren möchte.

Für die k-means-Zielfunktion kann im Schnitt die Ursprungsmenge auf 10 % durch eine Kernmenge reduziert werden und das Ergebnis ist maximal 10 % schlechter als die Lösung auf den Originaldaten (aus den Experimenten). Zu erst die Kernmenge zu berechnen und darauf k-means zu optimieren ist zusammen Ressourcensparender als nur k-means auf den Originaldaten zu benutzen.

Was noch nicht untersucht wurde und unter Umständen zu Problemen führen könnte, ist die Situation, wenn die Daten des Datenstroms tatsächlich sehr groß werden. Denn die Begrenzung der maximalen coreset features ist zu jedem Zeitpunkt im Algorithmus fest und wird nicht geändert. Da in einem potentiell unendlichen Datenstrom die maximale Anzahl immer irgendwann überschritten wird und die resultieren threshold-Verdopplung den Radius vergrößert, haben wir irgendwann im ersten Level ein einziges coreset feature, dessen Radius alle Punkte einschließt und dessen Radius in keinem Zusammenhang mehr mit dem Datensatz steht. Die coreset features der Kindsmenge bilden trotzdem noch den Datensatz ab. Es können aber Szenarien generiert werden, in denen dieses Verhalten dazu führt, dass die Kernmenge keine gute Repräsentation des Datensatzes mehr darstellt.

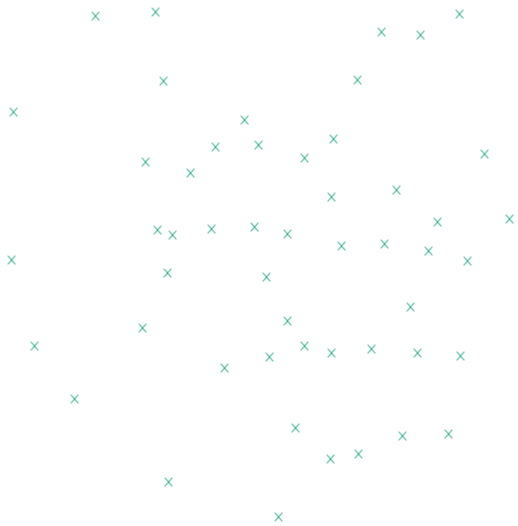


Abbildung 5.6: Rauschen mit maximal 100 coreset features

Und das Berechnete Clustering auf der Kernmenge eine schlechte Qualität aufweist.

Möglichkeiten, um diesem Verhalten entgegenzuwirken, wärn z. B., wenn man coreset features mit einem Zeitstempel versieht und ein coreset feature verwirft, wenn es sehr lange nicht betrachtet wurde, hierbei muss nur garantiert werden, dass die Punkte in dem coreset feature weiter eine Rolle spielen und komplett verworfen werden. Oder ein Punkt wird nur dann in die coreset feature-Struktur aufgenommen, wenn die Änderung am Mittelpunkt eines coreset features größer als ein Parameter ist, und andernfalls verworfen.

Somit könnte man dem ständigen Neubilden und Threshold-Verdoppeln entgegenwirken. Dies gilt es noch weiter zu untersuchen und zu belegen.

Die Untersuchungen haben gezeigt, dass die berechnete Kernmenge für k-means durchaus auch für DBScan benutzt werden kann, jedoch ist die angenommene einfache Verwendung des Radius als ϵ nicht möglich, da DBScan auf der 2-Norm arbeitet und die Kernmengen-berechnung auf der quadrierten 2-Norm, daher bilden die Radien nicht die Euklidsche-Distanz ab.

Wenn man mit den Parametern spielt, kann man eine gute Einstellung finden um eine Lösung zu erhalten. Weiter haben die Experimente gezeigt, dass man die Größe der Kernmenge auf deutlich mehr als 10 % setzen muss, da Rauschen hier ein wichtiger Gesichtspunkt ist. Daher gilt noch zu untersuchen, ob die Zeit für Berechnung der Kernmenge und danach die Ausführung von DBScan effizienter ist, als nur die Ausführung von DBScan.

Was jedoch noch weiterer Experimente und Untersuchungen bedarf, sind bei DBScan die unterschiedlichen Radien der verschiedenen Level in der Datenstruktur. So wie jetzt implementiert, benutzt DBScan immer den Radius aus Level 1 und multipliziert diesen mit einem konfigurierbaren Parameter. Bei einer klaren Datenstruktur (ohne Rauschen) funktioniert dies sehr gut. Wenn jedoch Rauschen dazukommt oder zwei Strukturen, die eigentlich ein Cluster bilden, mit ihren Grenzen sehr nah zusammen liegen, wird DBScan mit dieser Kernmenge an seine Grenzen stoßen und keine klaren Strukturen mehr erkennen. Um dies zu lösen, muss man im ersten Schritt ein *dynamisches Epsilon* implementieren, so dass nicht anhand eines festenϵ gerechnet wird, sondern, dass der Abstand der coreset features und ob diese zusammengehören mit ihren zugehörigen Radien berechnet wird. Im Schritt zwei würde man ganz von einem Radius abweichen und eine Varianzmatrix implementieren. Eine solche Matrix legt keinen festen Radius in allen Dimensionen fest, sondern es wird ein Körper um das coreset feature gelegt. Nun würde man nicht den Abstand zweier coreset features berechnen, sondern den Abstand der Varianz-Matrizen der coreset features. Somit würde man mit DBScan immer ein gutes Ergebnis berechnen. Jedoch muss man zu jedem coreset feature eine Varianz-Matrix speichern. Die Kosten um diese zu berechnen und zu speichern stehen evtl. nicht im Verhältnis zu Ertrag. Es könnte sein, dass die Berechnung mit DBScan auf den originalen Daten genauso viele Ressourcen, Speicher und Rechenzeit, braucht wie die Bildung der Kernmenge mit Varianz-Matrix und dann eine Anwendung von DBScan. Dies gilt untersucht zu werden.

Literatur

[1] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. StreamKM++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[2] David Arthur and Sergei Vassilvitskii. K-Means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1027–1035, Philadelphia, PA, USA, 2007.

[3] Christian Bockermann and Hendrik Blom. The streams framework. Technical Report SFB876 (5), TU Dortmund, 2012.

[4] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *In 2006 SIAM Conference on Data Mining*, pages 328–339, 2006.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231. AAAI Press, 1996.

[6] Hendrik Fichtenberger, Marc Gillé, Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. BICO: BIRCH meets coresets for k-means clustering. In *Algorithms–ESA 2013*, pages 481–492. Springer Berlin Heidelberg, 2013.

[7] Mohammed Ghesmoune, Hanene Azzag, and Mustapha Lebbah. G-stream: Growing neural gas over data stream. In *ICONIP (1)*, pages 207–214, 2014.

[8] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, 2003.

[9] Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 291–300, New York, NY, USA, 2004. ACM.

[10] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, corrected edition, August 2003.

[11] M. Lichman. UCI machine learning repository, 2013.

[12] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. Former Bell Laboratories paper from 1957.

[13] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[14] Jonathan A. Silva, Elaine R. Faria, Rodrigo C. Barros, Eduardo R. Hruschka, André C. P. L. F. de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Computing Surveys*, 46(1):13:1–13:31, 2013.

[15] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.