

Storingen voorkomen met slimme algoritmen

Modellenpracticum, Radboud Universiteit Nijmegen

Opdrachtgever:

Sander Rieken (Alliander N.V.)

Begeleider:

Wieb Bosma (Radboud Universiteit Nijmegen)

Uitgevoerd door:

Fons van der Plas
Matthijs Neutelings
Sebastiaan van Krieken
Dennis Geelhoed
Rick Koenders

Inhoudsopgave

1	Inleiding	2
2	SCG	2
I	Analyse SCG data	3
3	De data	3
4	Opvallende kenmerken	5
II	Cluster-Algoritmes	8
5	Een cluster-algoritme algemeen	8
6	Poisson-algoritme	9
7	Pinta-algoritme	17
8	DBSCAN-algoritme	19
9	Ensemble-model	23
10	Vergelijking	31
III	Tot slot	36
11	Conclusie	36
12	Ideeën voor de toekomst	36
IV	Appendix	38
A	Monte Carlo-algoritme	38

1 Inleiding

Voor het vak Modellenpracticum uit de Bachelor Wiskunde van de Radboud Universiteit is het de bedoeling om in een groepje van circa 5 studenten een praktische opdracht uit te voeren. De studenten kunnen hierbij kiezen welke opdracht ze willen doen. De opdrachten worden op hun beurt weer aangeboden door bedrijven. Dit verslag gaat over de opdracht *Storingen voorkomen door slimme algoritmen* van het bedrijf Alliander. Deze opdracht werd uitgevoerd door de studenten Fons van der Plas, Matthijs Neutelings, Sebastiaan van Krieken, Dennis Geelhoed en Rick Koenders.

Het bedrijf Alliander bestaat uit een aantal bedrijfsonderdelen, waaronder Liander. Liander beheert in grote delen van Nederland, waaronder de provincies Gelderland en Noord-Holland, het elektriciteitsnet. In de opdracht vroeg Alliander om zwakke punten in de elektriciteitskabels op te sporen. Op deze manier kan men de kabels al repareren voordat er een storing heeft plaatsgevonden.

1.1 Middenspanningsnetwerk

Het Nederlandse elektriciteitsnet is opgedeeld in drie sectoren: het hoogspanningsnet, het middenspanningsnet en het laagspanningsnet. Het hoogspanningsnet verbindt de grote elektriciteitscentrales met elkaar over zeer lange afstanden. Er staat een spanning van 220kV of 380kV over kabels in dit netwerk. Het laagspanningsnet wordt direct gebruikt door huishoudens. 230 Volt is de spanning over dit netwerk. Het middenspanningsnetwerk (vaak 10kV, 15kV of 20kV) zorgt voor de overbrugging van het hoogspanningsnet (in Nederland 110kV, 150kV, 220kV of 380kV) naar het laagspanningsnet (230V). Alliander beheert in delen van Nederland het middenspanningsnetwerk en het laagspanningsnetwerk. Hogespanningskabel zijn vaak bovengronds, maar middenspanningskabels liggen in Europa meestal onder de grond. (In de VS zijn ze bovengronds, dit zijn de iconische houten masten.) De middenspanningskabels onder de grond leggen meestal een afstand af van minder dan 10km. Ze bestaan niet uit één stuk, maar uit verschillende onderdelen die aan elkaar vastzitten met zogenoemde *moffen*.

Deze moffen blijken zwakke punten van de kabels te zijn. Vaak is een storing te wijten aan een defect van één van de moffen. Aangezien de kabels zich doorgaans ondergronds bevinden en omdat het middenspanningsnet soms flink verouderd is, weet Alliander niet altijd waar de moffen zich precies bevinden. De klassieke manier om een fout op te sporen is om halverwege de draad een teststroom aan te sluiten, en het effect te meten. Door dit te herhalen kom je steeds dichterbij de locatie van de fout. Dit kost echter veel tijd en geld, de kabels liggen immers ondergronds.

Een middenspanningskabel bestaat doorgaans uit vier delen, die geïsoleerd van elkaar moeten zijn: de aarddraad, en de drie fasen van wisselspanning. Een *fout* kan een verbreking van een van de vier verbindingen zijn, maar is doorgaans juist een kortsluiting: een van de fasen komt in (elektrisch) contact met de aarddraad, of met een andere fase. Dit gebeurt wanneer het isolatiemateriaal tussen de geleiders niet zijn taak doet, omdat het is gescheurd, gesmolten, geplet, als test doorgeknipt, of iets dergelijks. In dit geval slaan de zekeringen van de draad over, en is de draad onbruikbaar tot de fout is opgelost. Moderne zekeringen worden kort na een fout automatisch een aantal keer gereset, en als de fout dan weg is (dit heet een *dovende* fout) is de kabel weer in gebruik.

Vaak wordt een fout voorafgegaan door *ontladingen*: vonkjes binnen de draad, bijvoorbeeld tussen de twee fasen, op plekken in de kabel waar het isolatiemateriaal aan het degraderen is.

2 SCG

Om zwakke plekken in de kabel op te sporen heeft het bedrijf DNV GL de zogenoemde *Smart Cable Guard* (SCG) ontwikkeld.¹ De SCG detecteert ontladingen (vonken) die in de kabel optreden en slaat de locatie, de ladingsgrootte en het tijdstip van de ontlading op in een bestand. De ontladingen worden in het Engels ook wel *Partial Discharges* (PD's) genoemd. Er treden per toeval altijd wel een paar PD's op in een willekeurig stuk van de kabel. Als er meer PD's dan normaal optreden in een bepaald stuk van de kabel is er iets met dat stukje kabel aan de hand. Op die manier kan de netbeheerder zien waar ze de kabel moet vervangen.

De data-analyse wordt gedaan door het externe bedrijf DNV GL, die de SCG ook ontwikkelt. Dit bedrijf stuurde waarschuwingen naar Alliander als er te veel ontladingen in een stukje kabel plaatsvonden. Deze analyse is gedeeltelijk geautomatiseerd: zeer sterke veranderingen in PD-gedrag worden automatisch herkend, en (in dit geval) Alliander ontvangt hier dan direct bericht van. Toch wordt alle data ook

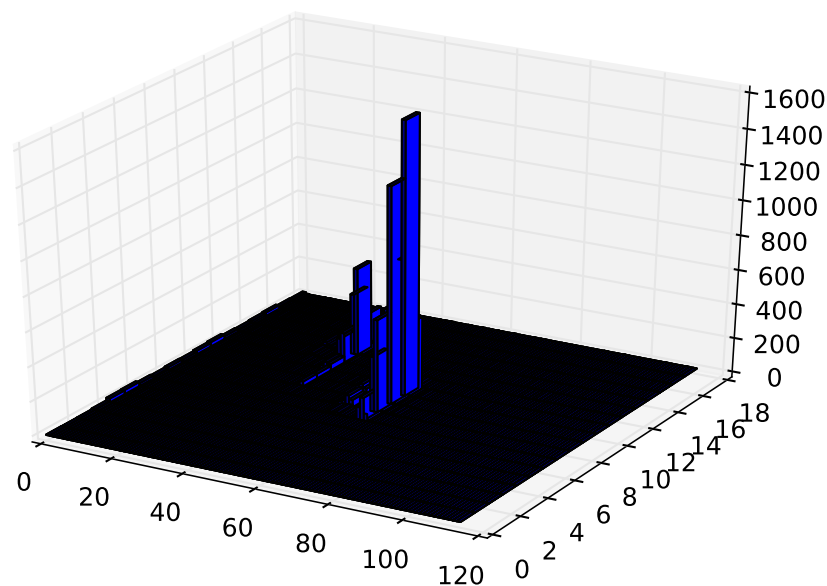
¹De echte geschiedenis van de ontwikkeling is minder beknopt.

handmatig doorgenomen (waarschijnlijk met behulp van dezelfde visualisaties als in dit verslag), en kleinere pieken in PD-gedrag worden aan het einde van de maand naar Alliander gestuurd. Dit zijn dan plekken in de kabel waar ze ‘op moeten letten’, en misschien vervangen *voordat* een fout heeft plaatsgevonden. Handig!

Deel I

Analyse SCG data

3 De data



Figuur 1: PD-dichtheid in een segment van circuit 2063. Op het vlak zijn locatie (links naar rechts) en tijd (in het papier) te zien, tegenover PD-aantallen per bakje (hoogte).

Per circuit (kabel) hebben wij twee of drie bestanden met data: één met de gemeten gedeeltelijke ontladingen, één met algemene eigenschappen van het circuit, en indien relevant één met ‘warnings’. Dit zijn .csv-bestanden (tabellen, *comma separated values* (hoewel ze niet kommagescheiden maar puntkommagescheiden zijn)). Deze bestanden zijn (na enige verwerking door onze opdrachtgever) de bestanden die Alliander *aan het einde van de maand* van DNV GL krijgt. Het ziet er echter naar uit dat Alliander een overeenkomst gaat sluiten om deze data direct te krijgen, en dan zal de analyse van dit verslag dus extra relevant worden.

3.1 Kabeleigenschappen.csv

In het bestand met algemene data staan de locaties van de middenspanningsruimtes en van bekende moffen voor dit circuit. Bij de moffen staat ook wat voor type mof dit is. Ook staat de lengte van de kabel in dit bestand.

Niet alle moffen staan in dit bestand, en de *moftypes* kloppen ook niet allemaal. De reden voor deze discrepantie is dat de kabels ouders zijn dan de huidige ambities van Alliander om data te verzamelen, en de kabels soms een levendig verleden hebben gehad. Niet alle reparaties zijn goed geregistreerd. Zie Figuur 2 voor een voorbeeldbestand.

```

Component type;Length (m);Cumulative length (m)
RMU;;0.0
Termination (unknown);;0.0
Cable (PILC, 3 cores, belted);18.21;18.21
Joint (oil);;18.21
Cable (PILC, 3 cores, belted);329.9;348.11
Joint (unknown);;348.11
Cable (PILC, 3 cores, belted);25.42;373.53
...

```

Figuur 2: Een cableconfig bestand

3.2 Gedeeltelijke-ontladingen.csv

Het tweede bestand is dat van de gemeten PD's. Hij bestaat uit drie kolommen: de tijd, de locatie en de grootte van de ontlading. Voor elke minuut dat er gemeten is heeft het bestand minstens één regel, ook als er geen PD heeft plaatsgevonden. Op sommige minuten vinden meerdere PD's plaats, dan heeft elke PD zijn eigen regel. Dit bestand is het meest relevant voor ons algoritme.

```

...
2016-08-04 13:29:00;;
2016-08-04 13:30:00;;
2016-08-04 13:31:00;;
2016-08-04 13:32:00;;
2016-08-04 13:33:00;1310.25107883513;2737.0
2016-08-04 13:34:00;;
2016-08-04 13:35:00;;
2016-08-04 13:36:00;;
2016-08-04 13:37:00;;
2016-08-04 13:38:00;;
2016-08-04 13:39:00;;
2016-08-04 13:40:00;1305.80955400445;7097.0
2016-08-04 13:41:00;;
2016-08-04 13:42:00;;
2016-08-04 13:43:00;;
2016-08-04 13:44:00;;
...

```

Figuur 3: Een bestand met gedeeltelijke ontladingen (PD's)

3.3 Warnings-van-DNV-GL.csv

Het derde bestand bevat de 'warnings' die DNG VL naar Alliander heeft doorgestuurd. Deze waarschuwingen stuurt DNV GL naar Alliander als uit de metingen blijkt dat er op een locatie van de kabel een grote kans is dat er iets niet goed gaat of niet goed dreigt te gaan. Elk warningsbestand heeft een locatie, een grootte (1,2,3 of Noise) en een starttijd en eindtijd.

```

Location in meters (m);SCG warning level (1 to 3 or Noise);Start Date/time (UTC);End Date/time (UTC)
818;3;2016-08-11 14:55:18;2016-08-31 23:59:00
818;N;2016-08-11 14:55:18;2017-02-28 00:00:00
1309;1;2017-01-26 15:13:37;2017-02-07 09:14:41
1309;N;2016-12-11 21:14:22;2017-01-26 15:13:37
1313;3;2016-09-26 12:39:46;2016-11-28 20:45:44

```

Figuur 4: Een warnings-bestand

De data-analyse is uitgevoerd met Python 3.7, met name gebruikmakend van pandas, numpy,

`matplotlib`, `printipigeon`, `jupyter` en `scikit-learn`. We hebben GitHub gebruikt voor *version control*, hiermee hebben we samen aan het project gewerkt.² Het project is *openbaar*³ (exclusief data), en is online beschikbaar op:

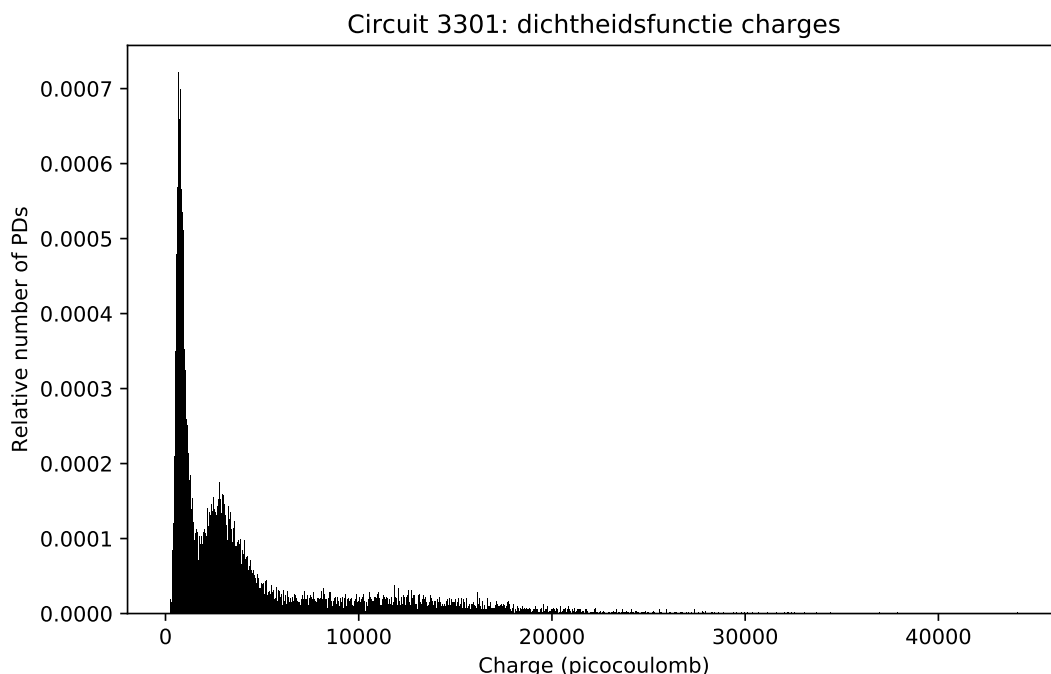
<https://github.com/fonsp/SCG-analyse>

Deze repository bevat onder andere een **importeerbare, algemene module**. Dit betekent dat het door iedereen gemakkelijk gebruikt kan worden, net als populaire modules zoals `numpy` of `printipigeon`. De module bestaat uit een aantal submodules, en alle klassedefinities en algemene functies hierin zijn **uitgebreid en duidelijk gedocumenteerd**. Dat wil zeggen: wanneer iemand de ingebouwde hulp-functies van `Python` oproept op iets van de module, komt de werking en beschrijving van parameters in beeld.

De repository bevat ook zogenaamde *notebooks*: deze documenten zijn een mengelmoes van tekst, code en haar output. Hierin worden alle algoritmes, vergelijkingen en verdere analyses gedaan en beschreven. De belangrijkste module is `/notebooks/Voorbeeld clusterizer submodules.ipynb`, hierin wordt de **volledige module** voorgedaan en uitgelegd. Dit verslag dient ook niet als technische beschrijving van de implementatie van de algoritmes: dit zit al in de module zelf.

4 Opvallende kenmerken

De metingen van de circuits verschillen onderling aanzienlijk. Bij sommige circuits is de hoeveelheid PD's veel hoger, ook in gebieden waar niks aan de hand lijkt te zijn. Ook zijn er grote verschillen in ladingen. Vaak zie je wel dat de verdeling van de ladingen twee pieken heeft. Dit is te zien in figuur 5. Een van onze theorieën hierover is dat het ruis zijn eigen verdeling heeft en de PD's waar daadwerkelijk iets gebeurt ook hun eigen verdeling hebben.

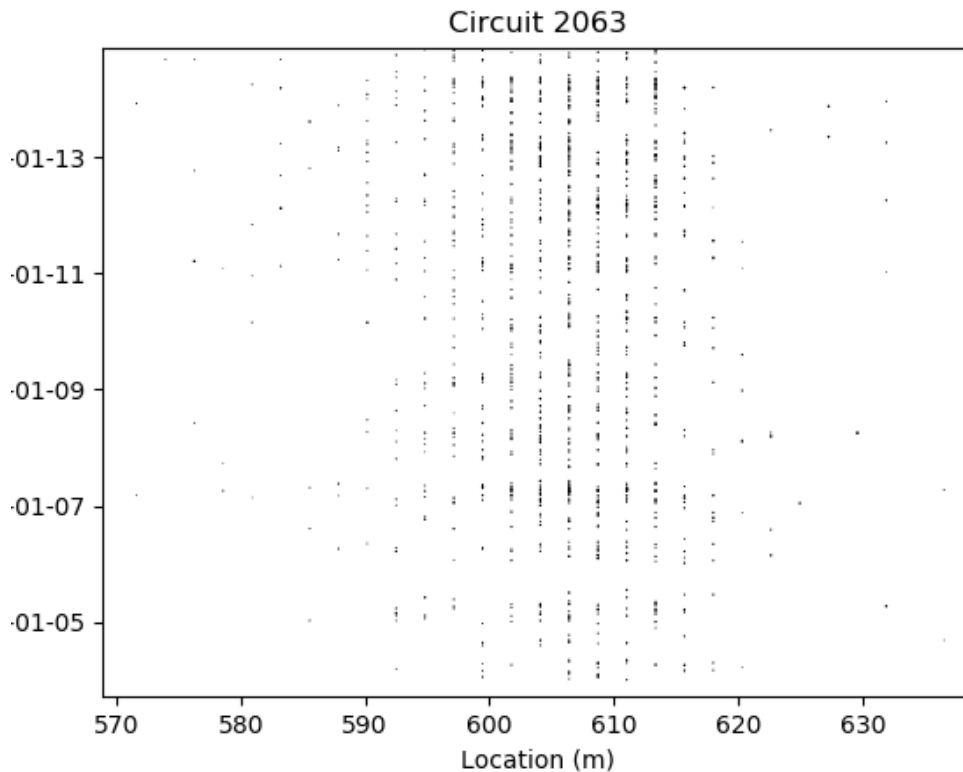


Figuur 5: Er is een combinatie van twee verdelingen te zien.

²Het leek ons daarom ook niet nodig om een tweede 'logboek' bij te houden. De volledige geschiedenis, met beschrijvingen, is te vinden op <https://github.com/fonsp/SCG-analyse/commits/master>.

³We hebben geen precieze open source-licentie met onze begeleider overlegd.

Als je heel erg inzoomt op de locaties van de PD's dan zie je dat de locaties in verticale banden lijken te liggen. Dit is in figuur 6 te zien. Helaas zijn die banden niet helemaal recht, je ziet namelijk alsnog kleine fluctuaties in locatie. Wij hebben een theorie dat de locaties gediscrètiseerd zijn en dus eigenlijk wel precies op een verticale lijn liggen, maar dat de leverancier van de data, DNVGL, ze opzettelijk 'jittert'. Het onhandige aan deze verticale banden is dat als je bijvoorbeeld een histogram van de locaties wil maken, het zou kunnen dat er in de ene bin precies twee van zulke banden zitten en in de ander drie. Helaas is de afstand tussen de banden niet helemaal consistent en verschilt het ook per circuit.



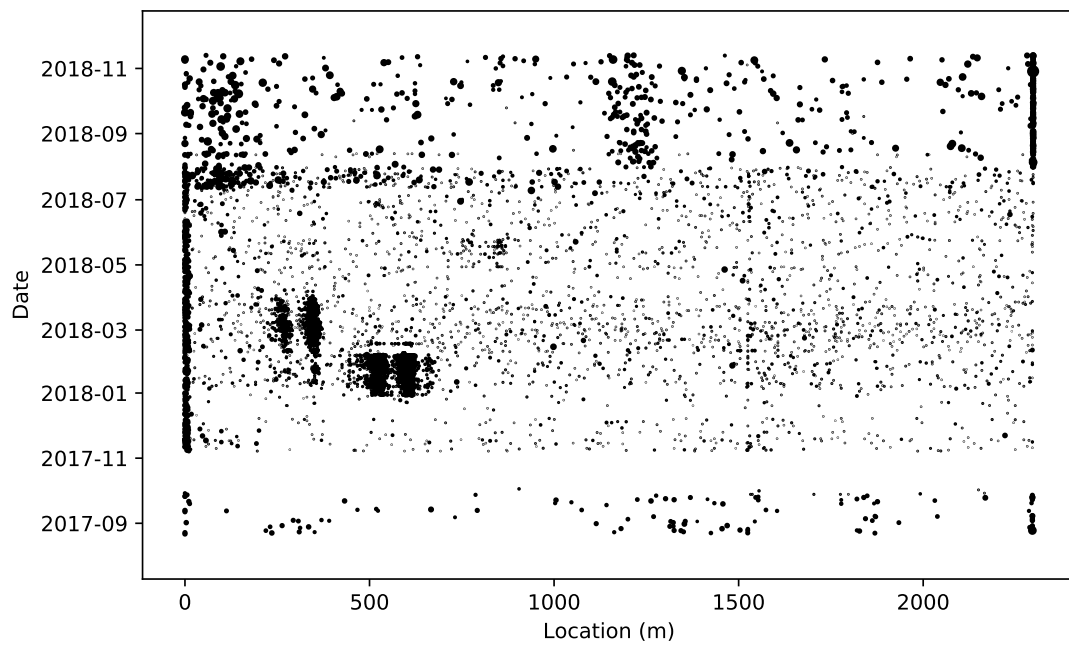
circuit.png

Figuur 6: Een ingezoomde scatter plot. De banden van PD's zijn hier duidelijk zichtbaar.

Een ander opvallend fenomeen is dat van de spiegelclusters. Bij een aantal van de circuits zie je duidelijk twee PD-clusters die op verschillende locaties zitten, maar precies dezelfde vorm lijken te hebben. Het zou interessant zijn om uit te zoeken waar dit precies door veroorzaakt wordt. De gangbare theorie is dat het komt doordat sommige moffen in de kabel signalen kunnen reflecteren. Dit heeft invloed op de tijd tussen het moment dat de linkerkant van de SCG het ontvangt en dat de rechterkant het ontvangt. Dit resulteert in een verkeerde locatie.

Bij sommige PD-metingen zie je dat het gedrag van de PD's en de ladingen opeens heel sterk verandert. Het lijkt dan alsof de SCG op een nieuw circuit is geplaatst of dat de gevoeligheid is aangepast.

Helemaal links en helemaal rechts in het circuit is altijd een buitensporig hoge dichtheid van PD's. Hier moet rekening mee worden gehouden als men naar verdachte locaties zoekt.



Figuur 7: Bij deze scatter plot zijn aantal van deze kenmerken duidelijk te zien: De twee clusters rond het begin van 2018 hebben allebei een spiegelcluster ernaast. Vanaf augustus 2018 gedraagt de scatter plot zich opeens heel anders. Aan de zijanten zitten zwarte stroken van PD's.

Deel II

Cluster-Algoritmes

5 Een cluster-algoritme algemeen

Het vinden van verdachte locaties in een ondergrondse elektriciteitskabel is van groot belang, omdat hierdoor foutieve moffen op tijd kunnen worden gevonden en mogelijke calamiteiten verholpen kunnen worden. Onze opdracht luidt om een algoritme te ontwikkelen dat automatisch verdachte locaties vindt. Om verdachte locaties te vinden zoeken wij naar clusters. De precieze definitie van een cluster ligt niet vast en verschilt per algoritme, maar het komt neer op een gebied in de locatie-tijd grafiek met buitensporig veel PD's waarbij soms ook naar de ladingen wordt gekeken. Zo'n cluster is niet per definitie een reden om actie te ondernemen, maar het geeft aan dat daar mogelijk een mof ligt die kapot dreigt te gaan. Dan kan vervolgens een expert handmatig inschatten of het iets is om je zorgen over te maken.

5.1 Voorwaarden cluster-algoritme

In principe zijn wij helemaal vrijgelaten in onze implementaties van een algoritme dat clusters zoekt en zelfs in de precieze definitie van een cluster. Echter hebben wij gestreefd om te voldoen aan een aantal zelfopgelegde voorwaarden.

Snelheid: Vanzelfsprekend geldt, hoe sneller de algoritmes hoe beter. Hoewel langer ook zou volstaan, was ons streven om per circuit minder dan een seconde bezig te zijn.

Parameters: Elk van onze algoritmes is afhankelijk van parameters. Het aantal parameters per algoritme proberen we zo klein mogelijk te houden. Ook heeft elk algoritme een aantal standaardparameters waarvoor het algoritme goed werkt. Deze hebben we door met verschillende circuits te experimenteren handmatig bepaald.

Onafhankelijkheid: We willen dat de parameters zo veel mogelijk onafhankelijk zijn van het circuit. Een voorbeeld hiervan is vergelijken met de gemiddelde dichtheid van PD's in plaats van met een absoluut aantal. Een ander voorbeeld is bakjes maken van een bepaalde lengte in plaats van een percentage van de totale lengte van de kabel.

Output: Omdat we uiteindelijk de verschillende algoritmes willen kunnen combineren, hebben we besloten dat de output van de algoritmes van een bepaald type object moet zijn. Dit object is geïmplementeerd als Python-klasse in onze module en wordt later in het verslag verder toegelicht. Onze clusters zijn altijd gebieden in de vorm van een rechthoek. Een voordeel hiervan is dat het combineren van de verschillende algoritmes makkelijker maakt. Het uiteindelijke doel van verdachte clusters is om kapotte moffen op te sporen. Dus dan geeft een rechthoek een veel handigere indicatie van de locatie dan een of andere vreemde vorm.

5.2 beoordeling cluster-algoritme

Aanvankelijk dachten wij dat wij de warningbestanden handig konden gebruiken om de nauwkeurigheid van de algoritmes te bepalen. Helaas werkt dit niet goed, omdat bij veel gebieden die op het oog overduidelijk clusters zijn, geen warning is geweest. Ook is de warning altijd verschoven in de tijd, namelijk een stuk later dan het cluster daadwerkelijk begon. En ook van het cableconfig bestand hebben wij niet veel gebruik kunnen maken. De clusters liggen weliswaar rond moffen, maar helaas zijn er meestal te veel moffen op het circuit en zelfs dan is er nog een groot aantal onbekende moffen. Uiteindelijk hebben we dus alleen met onze eigen ogen kunnen bepalen hoe goed de algoritmes werken.

6 Poisson-algoritme

Het eerste algoritme dat we hebben ontwikkeld is zo simpel mogelijk begonnen, en is steeds een beetje aangepast om aan alle eisen van 5.1 te voldoen. Het algoritme bestaat nu uit twee delen: eerst worden ééndimensionale *locatieclusters* gevonden (alleen begin- en eindlocatie op de lijn) en vervolgens wordt dit verfijnd tot tweedimensionale clusters (een begin- en eindlocatie, en een begin- en eindtijd).

We zullen het volledige algoritme uitleggen aan de hand van twee voorbeeldcircuits, en daarbij de tussenresultaten laten zien. De gekozen voorbeeldcircuits zijn 2063 en 3010. Deze zijn gekozen omdat 2063 een aantal duidelijk zichtbare clusters bevat (die ook door DNV GL als *warnings* zijn aangegeven) van verschillende sterktes, en 3010 bevat geen clusters. In 2063 is de hoeveelheid PD-ruis ook niet constant, en hier moet het algoritme goed mee om kunnen gaan.

Deel 1: locatieclusters

Omdat clusters gedurende hun levensduur op dezelfde locatie zitten, is een logische eerste stap om enkel de locaties van PD's te analyseren, en zo te zoeken naar *lijnsegmenten met abnormaal hoog PD-gedrag*. Dit is misschien ook wel de belangrijkste eigenschap van het cluster voor verdere analyse, en voor reparatiewerk.

Deel 1A: Discretisatie

Als eerste worden de PD-locaties gediscetiseerd: de lijn wordt opgedeeld in *bakjes* van constante breedte. Voor een lijn van lengte L komt dit overeen met de opsplitsing

$$[0, L] = [0, d) \sqcup [d, 2d) \sqcup \dots \sqcup [Nd, (N+1)d),$$

met $N = \lfloor L/d \rfloor$ het aantal bakjes.

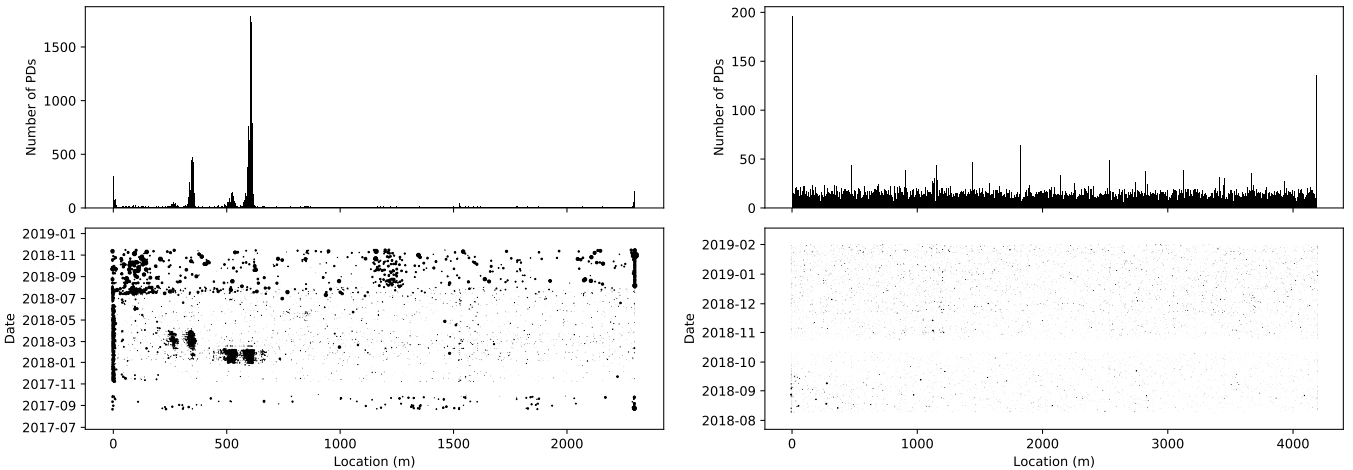
Als $PD \subseteq \{(x, t, c) \in [0, L] \times [0, T] \times \mathbb{R}_{\geq 0}\}$ de verzameling PD's is, dan is de functie

$$i \mapsto \sum_{(x,t,c) \in PD} \mathbb{1}_{[i \cdot d, (i+1)d)}(t)$$

een *discretisatie* van de PD-locaties. Het is ook mogelijk om niet het *aantal* PD's per bakje tellen, maar de *totale ontladingsgrootte* van de PD's in een bakje. Zo'n *gewogen discretisatie* is de functie

$$i \mapsto \sum_{(x,t,c) \in PD} c \cdot \mathbb{1}_{[i \cdot d, (i+1)d)}(t).$$

Met een van de parameters wordt bepaald of de ladingen wel of niet worden meegeteld (standaard: niet). Het resultaat van de discretisatie is gegeven in 8. Men ziet in de linkerfiguur de clusters al duidelijk terug. In de rechterfiguur zijn uitschieters minder klein ten opzichte van de hoeveelheid PD-ruis (let op de schaal van de verticale as), en houden minder lang aan.



Figuur 8: Eéndimensionale discretisatie (boven) van aantallen PD's in het circuit (onder).

Deel 1B: Model nominaal PD-gedrag

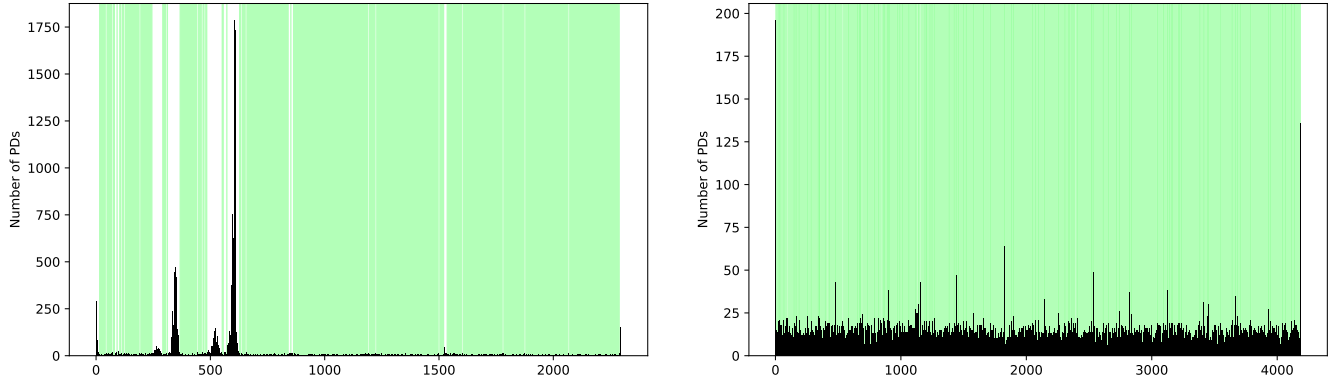
Gebaseerd op eerder onderzoek maken we de aanname dat *ten minste 80% van de lijn* nominaal PD-gedrag vertoont, i.e. geen clusters bevat.

In Deel 1A hebben we per lijnsegment (bakje) het aantal PD's in dat bakje geteld. Als we het 80%-kwantiel van deze bakjes bepalen, noem dit M_η , dan zegt onze aanname: alle lijnsegmenten waarop het aantal onder M_η ligt, bevat geen cluster, en vertoont nominaal PD-gedrag (PD-ruis). We modelleren PD-ruis als een Poissonproces,⁴ en deze bakjesinhouden dus als realisaties van dezelfde Poisson-verdeelde stochast:

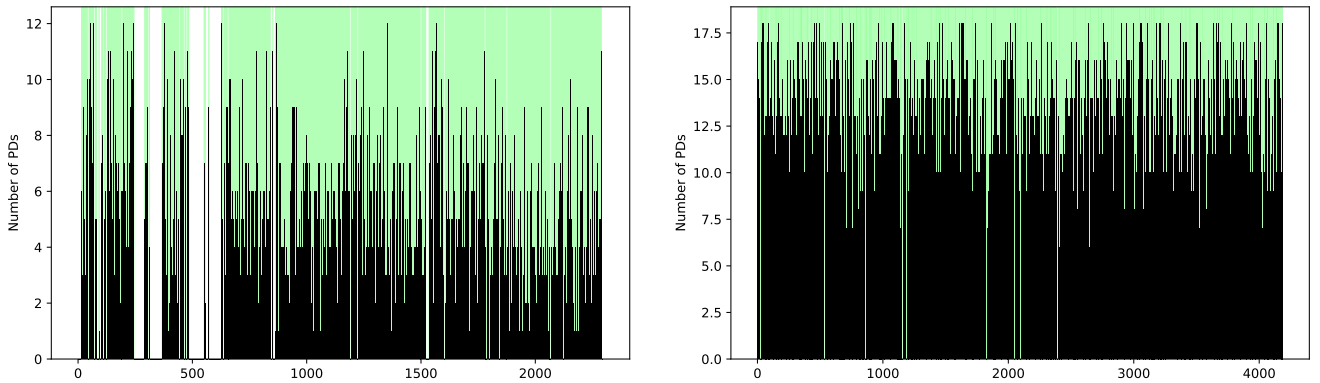
$$X_{\text{ruis}} \sim \text{Poisson}(\lambda)$$

oftewel, voor $k \in \mathbb{Z}_{\geq 0}$:

$$\mathbb{P}[X = k] = \frac{\lambda^k}{k!} e^{-\lambda}.$$

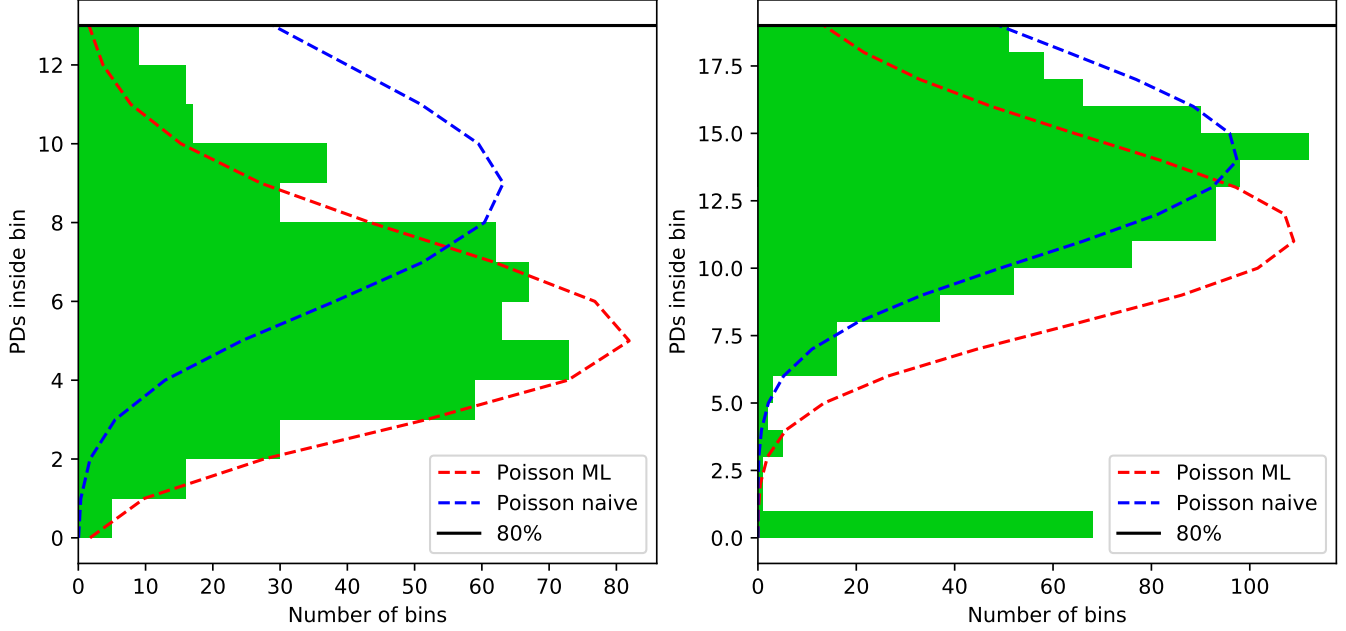


Figuur 9: Lijnsegmenten met PD-dichtheid onder het 80%-kwantiel zijn ingekleurd. Deze segmenten worden gebruikt om nominaal PD-gedrag te modelleren.



⁴Hoewel de fysische oorsprong van de PD-ruis ons niet precies duidelijk is, is het geen vreemde aanname dat de willekeurige vonken op onafhankelijke locaties en tijdstippen plaatsvinden. De frequentie van PD-gedrag is niet tijdsafhankelijk (zoals te zien in circuit 2063), maar lijkt wel constant te zijn gedurende korte periodes (waarschijnlijk tussen firmware-updates in). Omdat we, tijdens de eendimensionale discretisering, de tijdstippen weglaten, komt dit neer op het sommeren van Poisson-verdeelde stochasten, wat weer een Poisson-verdeelde stochast oplevert.

Figuur 10: Dezelfde figuur als hierboven, met alleen de lijnsegmenten onder het 80%-kwantiel. De verticale as is herschaald.



Figuur 11: Histogram van PD-dichtheden (staafhoogtes in vorige figuur). Beide Poisson-fits zijn weer-gegeven.

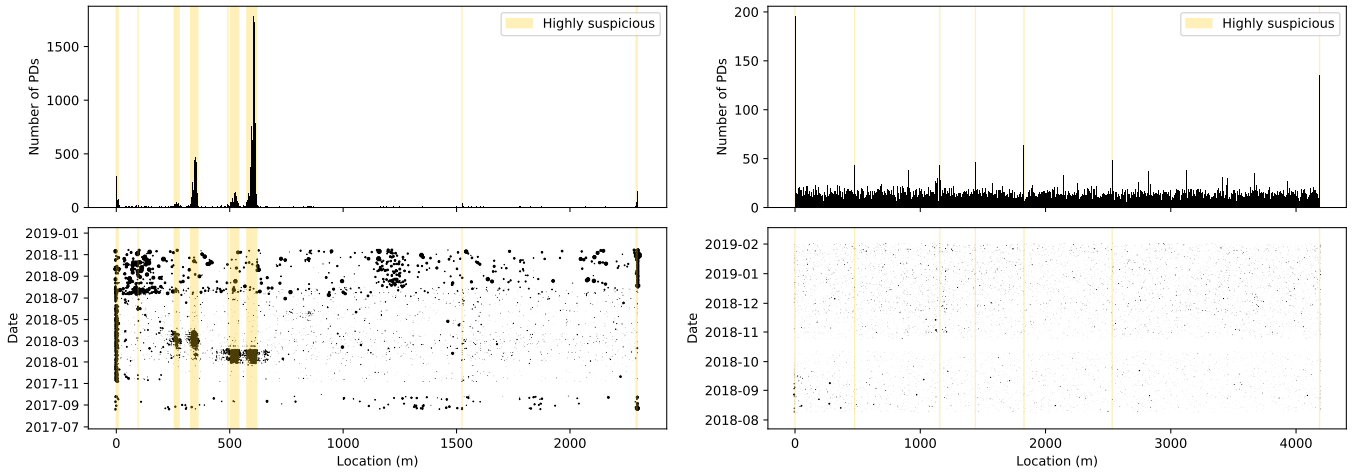
Gegeven de verdeling van PD-dichtheid, onder het 80%-kwantiel M_η (zie Figuur 11) willen we een Poissonverdeling (i.e. de parameter λ) schatten die deze verdeling modelleert. De meest voor de hand liggende schatter is de Maximum-Likelihood-schatter. (In het geval van de Poisson-verdeling is dit simpelweg het gemiddelde.) De Poisson-verdeling volgens de ML-geschatte parameter is gegeven in Figuur 11.

Hoewel dit bij circuit 2063 een goede fit geeft, werkt deze methode niet in het algemeen. In circuit 3010, waar geen clusters in zitten, heeft de selectie van lijnsegmenten onder het 80%-kwantiel ervoor gezorgd dat de verdeling *getrunceerd* is. In het ergste geval (wanneer er geen clusters zijn, zoals in 3010), is precies de 20% hoogste waarden afgekap. Dit motiveert de tweede schatter voor λ :

We bepalen de parameter λ zodat het 80%-kwantiel van $\text{Poisson}(\lambda)$ precies op M_η ligt.

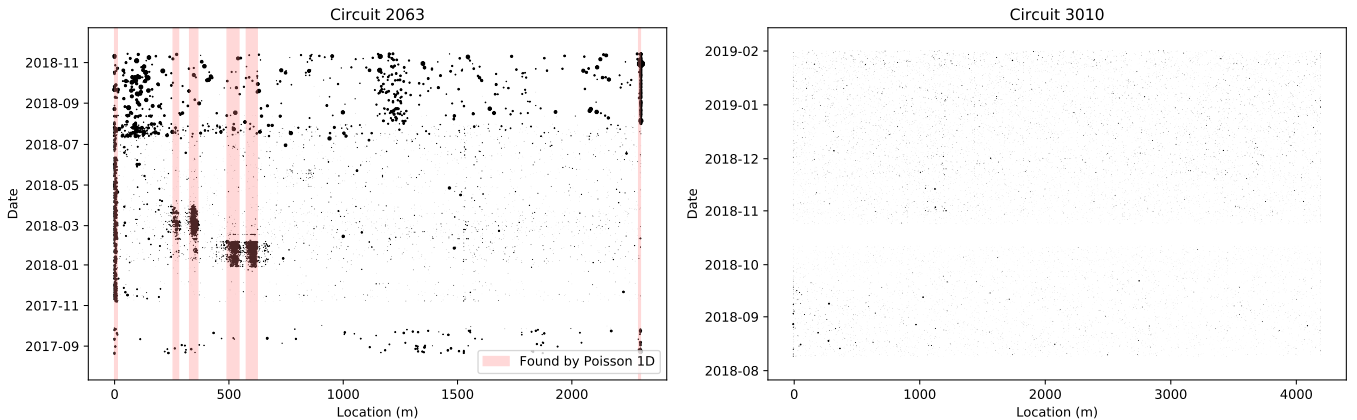
Deze naïve schatter zal altijd iets hoger zijn dan de ML-schatter, omdat het de juiste schatter is wanneer er geen clusters zijn. Wanneer er wel clusters zijn, dan is een kleiner gedeelte van de ruis afgekap, en zal de naïve schatter dus λ te hoog schatten. Toch is voor deze schatter gekozen, omdat de naïve verdeling de PD-dichtheid nooit *onderschat*.

Nu we een model voor PD-ruis hebben gevonden, kunnen we per lijnsegment bepalen wat de kans is dat de PD-dichtheid zo hoog is, *als er geen cluster zou zijn*. (Dit is dus een hypothesetoets: de nulhypothese is dat het lijnsegment geen cluster bevat, en dat de PD's afkomstig zijn van ruis.) We kunnen dan lijnsegmenten verwerpen met een significantie (standaard 95%). Deze lijnsegmenten zijn gegeven in Figuur 12. Dit is natuurlijk een deelverzameling van de lijnsegmenten die boven het 80%-kwantiel liggen (omdat de significantie groter is dan 80%).



Figuur 12: Lijnsegmenten waarop PD-dichtheid *volgens de Poisson-verdeling* buiten het 95%-kwantiel ligt.

Ten slotte worden alle aangrenzende, zeer verdachte hokjes samengenomen tot een locatiecluster, door middel van een *groepeeralgoritme*. Twee locatieclusters die maximaal 2 hokjes uit elkaar liggen, worden als één locatiecluster beschouwd. Er geldt ook beperking dat een locatiecluster minimaal 3 verdachte hokjes moet bevatten, zoniet wordt het locatiecluster genegeerd. De precieze implementatie van dit algoritme is in de module te lezen: `clusterizer.algoritms.group_boolean_series`.



Figuur 13: De bovenstaande lijnsegmenten worden gegroepeerd tot *locatieclusters*, door korte sprongen op te vullen, en te kleine gebieden te laten vervallen.

In circuit 2063 zijn alle gewenste locatieclusters gevonden. In Deel 2 zullen deze locatieclusters worden verfijnd tot tweedimensionale locatie-tijdclusters. In circuit 3010 worden geen locatieclusters gevonden, zoals gehoopt, en het algoritme is na Deel 1 al klaar.

Deel 2: locatie-tijdclusters

Als resultaat van Deel 1 hebben we een verzameling locatie-clusters. Dat wil zeggen, de begin- en eindlocatie van verdachte gebieden is gevonden. We willen dit verfijnen tot een verzameling tweedimensionale clusters: voor elk locatie-cluster zoeken we de begin- en eindtijd.

Ter vergelijking zijn dezelfde figuren gegeven, toegepast op een circuit met en zonder clusters. In het geval zonder clusters worden er in Deel 1 helemaal geen clusters gevonden, en wordt Deel 2 dus niet uitgevoerd. Om toch te kunnen zien hoe Deel 2 werkt zonder de aanwezigheid van clusters, zijn de figuren ook gegeven voor het circuit zonder clusters. Hierbij is kunstmatig een locatiecluster toegevoegd,

voordat Deel 2 is toegepast. Het resultaat is dat de verhouding van PD-dichtheid binnen en buiten het 'cluster' gedurende het volledige meetinterval rond 1 zit, en nooit de ondergrens (10) overschrijdt.

⁵

Een mogelijke aanpak is om begin- en eindtijden te zoeken met precies dezelfde methode als in Deel 1: we maken een histogram van PD-tijdstippen, en gaan dan op zoek naar pieken. Deze methode is echter problematisch om een aantal redenen:

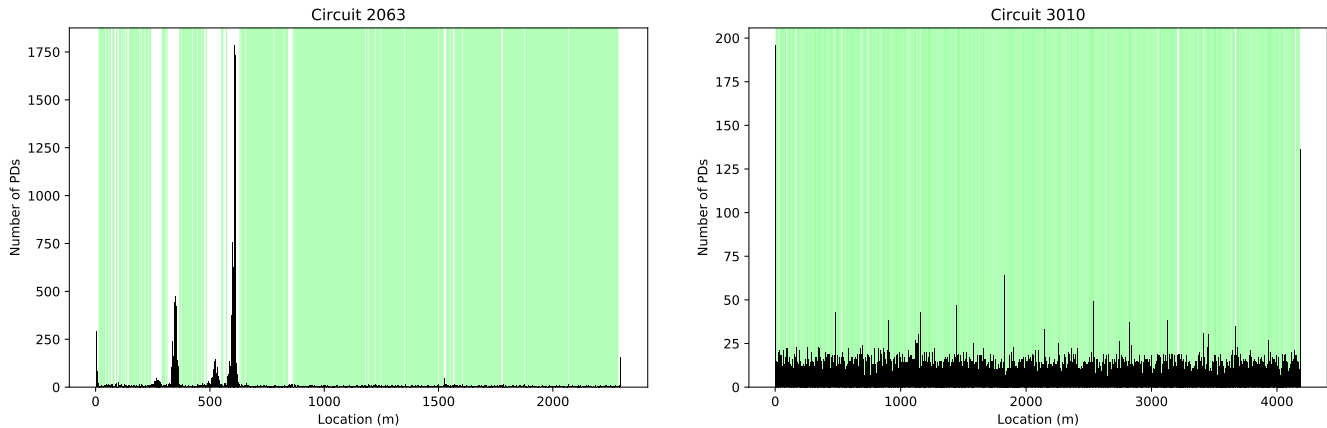
- De aanname dat er 80% van de *tijd* niets aan de hand is, lijkt niet te kloppen. Vaak duren clusters juist erg lang. Sterker nog, het komt voor dat een cluster tijdens de volledige meting aanwezig is. In een tijd-histogram zou er dan geen opvallend gebied zijn.
- Het PD-gedrag varieert sterk gedurende de tijd. In tegenstelling tot Deel 1 is het dus niet logisch om een statistisch model te fitten voor de PD-frequentie (aantal PD's per seconde) *op een willekeurig tijdstip*.

Gelukkig weten we al op welke lijnstukken de clusters zitten, en ook waar géén clusters zitten! Dit motiveert de aanpak van dit algoritme: Het meetinterval wordt opgedeeld in weken. Voor elk cluster worden deze intervallen vervolgens afgegaan, en in elk interval wordt de PD-dichtheid *binnen het locatiecluster* vergeleken met de dichtheid *op lijnsegmenten waarin geen clusters zitten*, de zogenaamde *nusters*.

Als de PD-dichtheid binnen dit interval, binnen het locatiecluster, niet relatief hoog is, dan ligt deze verhouding naar verwachting rond 1. Als de PD-dichtheid wel hoog is (omdat een gedeelte van een cluster in het interval ligt), dan is deze verhouding veel groter. Door een ondergrens te kiezen voor deze verhouding, vinden we precies alle intervallen terug waarop een cluster ligt, en deze intervallen vormen uiteindelijk de tweedimensionale clusters door aangrenzende intervallen samen te voegen.

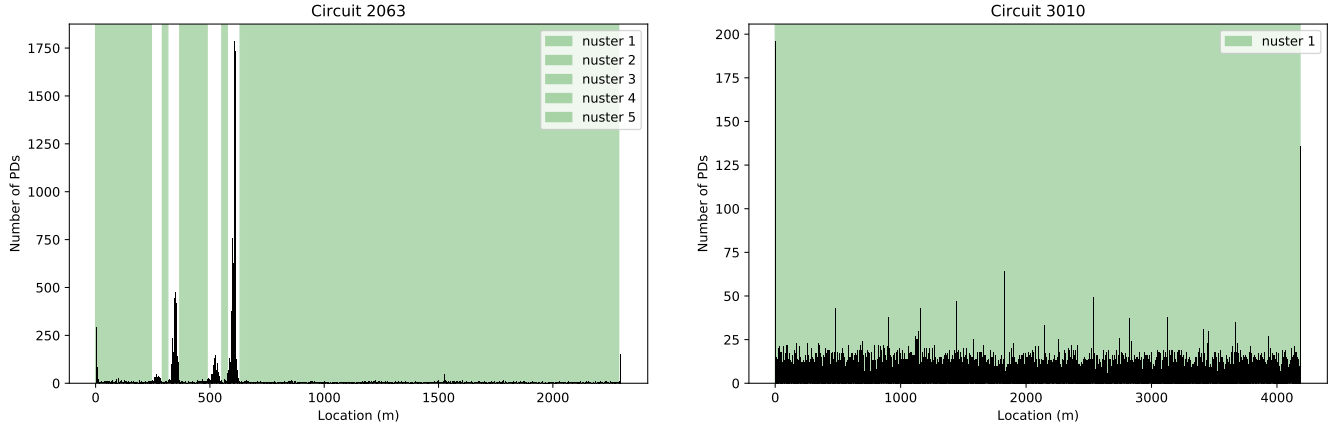
Deel 2A: Nusters

Met hetzelfde groepeer algoritme als in Deel 1 kunnen lijnsegmenten worden gevonden waarop de PD-dichtheid *onder* het 80%-kwantiel ligt. (Let op, in Deel 1 zijn niet de overige 20% van de lijnsegmenten tot clusters vervormd, maar slechts het kleine gedeelte ervan dat significant afwijkt van de Poisson-verdeling!) Dit zijn lijnsegmenten waarop per aanname *geen clusters* liggen. We noemen zo'n lijnsegment daarom een 'nuster' (*not a cluster*).



Figuur 14: Lijnsegmenten waarop het aantal PD's onder het 80%-kwantiel ligt.

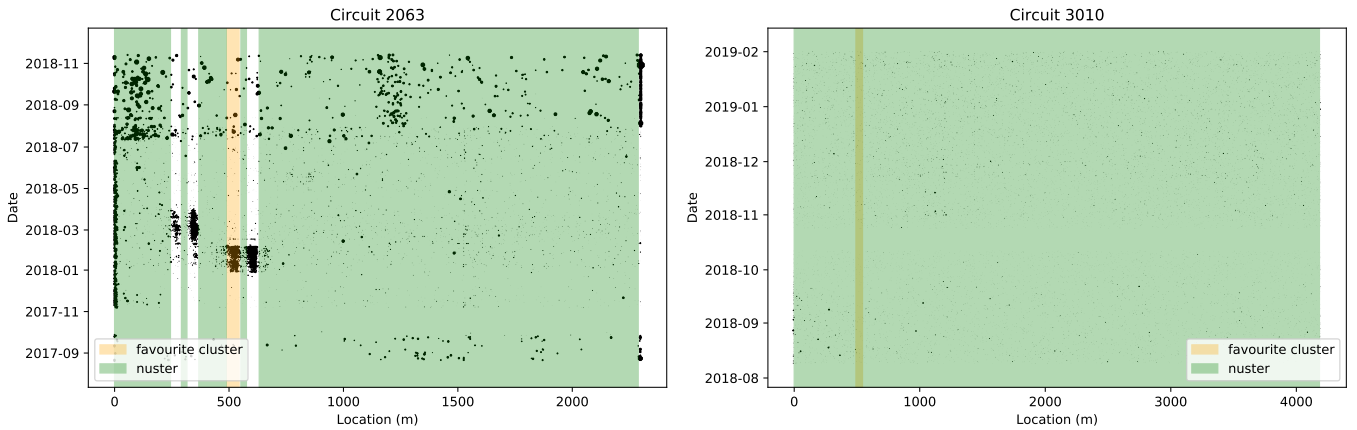
⁵In sommige circuits komt een cluster op dezelfde locatie enkele keren terug. In het algemeen kan één locatiecluster dus meerdere clusters bevatten, en hier moet rekening mee worden gehouden.



Figuur 15: De bovenstaande lijnsegmenten worden gegroepeerd tot *nusters*.

Deel 2B: Tijdsverloop

Elk eendimensionaal cluster wordt nu verfijnd tot een verzameling tweedimensionale cluster. Als voorbeeld kiezen we een van de eerder gevonden clusters uit.



Figuur 16: We kiezen een locatiecluster uit. Voor circuit 3010 is een kunstmatig locatiecluster toegevoegd, ter illustratie van de verdere analyse.

We delen het meetinterval op in deelintervallen van constante duur (dit is standaard 7 dagen). We maken nu *twee* discretisaties: per interval tellen we het aantal PD's binnen het cluster, en het aantal PD's dat in één van de nusters ligt. Dit tweede aantal geeft schatting voor de verwachte PD-dichtheid dat wordt veroorzaakt door ruis.

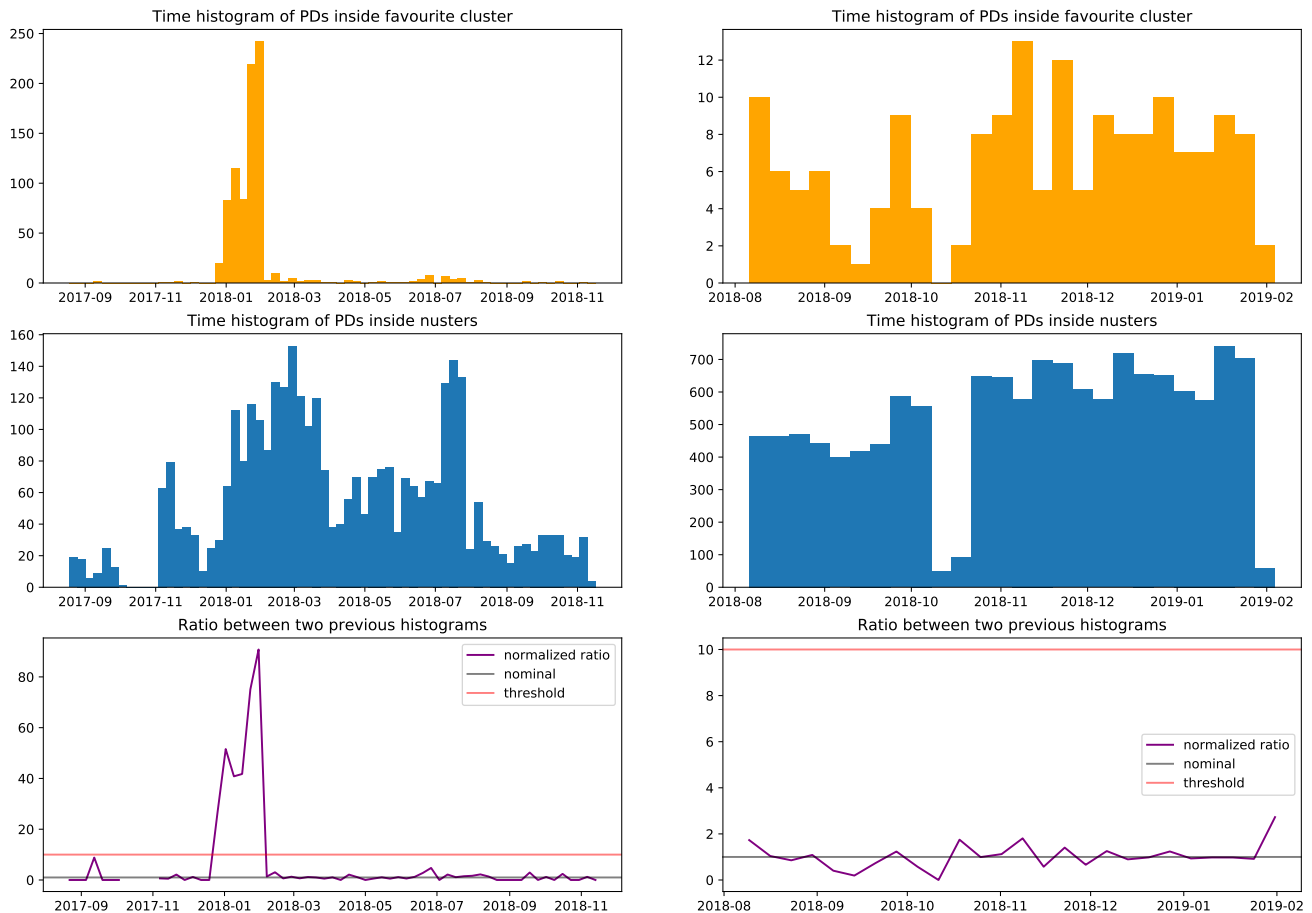
De belangrijke observatie is hier dat PD-gedrag *niet afhankelijk is van locatie*, maar in veel gevallen wel van de tijd. Door PD-dichtheid te meten op andere locaties, maar wel binnen hetzelfde tijdsinterval, krijgen we dus een goede schatting voor de verwachte PD-dichtheid. We nemen namelijk aan dat dit niet locatie-afhankelijk is.

Het resultaat van deze twee discretisaties is gegeven in Figuur 17. In de eerste figuur is het cluster duidelijk zichtbaar, en de begin- en eindtijden van de piek komen overeen met Figuur 16. De tweede figuur toont aan dat nominaal PD-gedrag sterk afhankelijk is van de meettijd. (Anders zou de grafiek redelijk constant zijn.) Ook is te zien dat de hoeveelheid achtergrondruis in de eerste grafiek (binnen het cluster) hiermee overeenkomt. Zo is er in beide figuren in juli 2018 een piek te zien.

Wanneer we de verhouding tussen deze twee reeksen berekenen (Figuur 17), is het cluster duidelijk te identificeren.⁶ Zoals verwacht, schommelt deze verhouding voor en na het cluster rond de 1. Eventuele

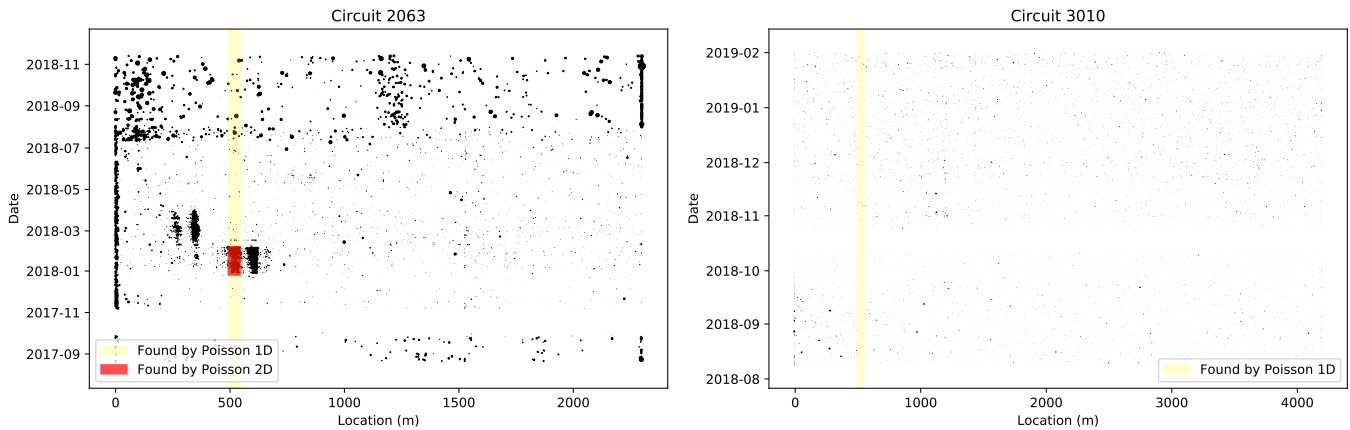
⁶In feite wordt niet de directe verhouding berekend, maar de verhouding tussen de PD-dichtheden (aantal PD's *per meter, per dag*). Dit is locatieinvariant!

afwijkingen gebeuren natuurlijk, en worden versterkt wanneer de PD-dichtheid binnen de nusters laag is. (Er wordt dan gedeeld door een laag getal.) Dit is het geval bij de uitschieter aan het begin van de meetreeks.

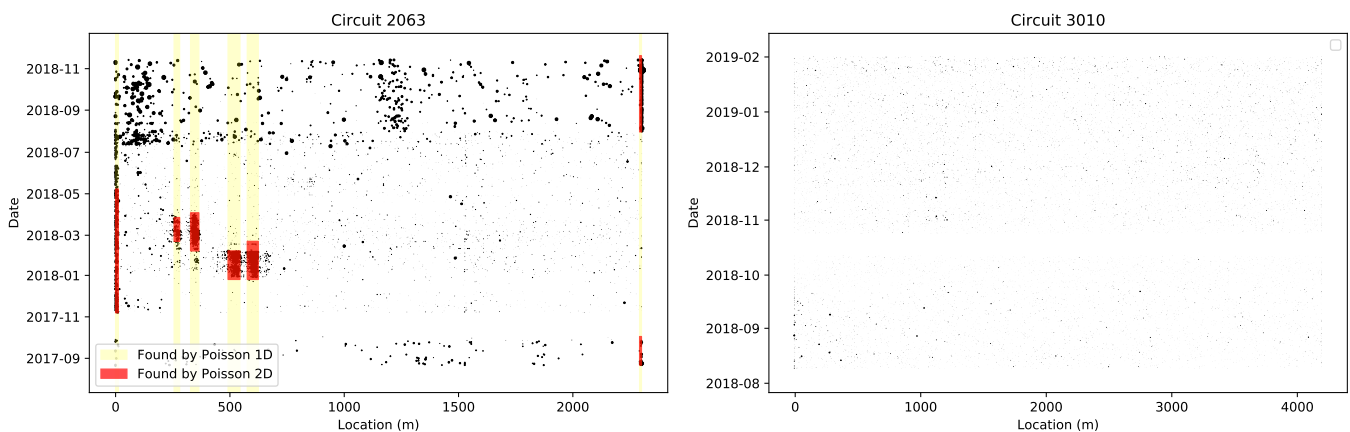


Figuur 17: Voor het gekozen locatiecluster (boven) en alle nusters (midden) is het aantal PD's per tijdsinterval geteld. De genormaliseerde verhouding tussen deze twee (onder) verklapt de begin- en eindtijd van alle clusters in het locatiecluster.

Door een ondergrens te kiezen voor deze verhouding (standaard 10) kunnen we de intervallen selecteren waar het cluster in ligt. Met hetzelfde groepeer algoritme als gebruikt in Deel 1 worden de gevonden intervallen samengevoegd tot clusters. (Standaardparameters: minimaal 2 intervallen, maximaal 1 overgeslagen interval.) In het gekozen voorbeeldcluster wordt op deze manier maar één cluster gevonden, te zien in Figuur 18. Op dezelfde manier worden alle locatieclusters verfijnd, en het resultaat is gegeven in Figuur 19.



Figuur 18: We groeperen de tijdsintervallen waarbij de dichtheid-verhouding groter is dan de ondergrens. Dit verfijnt het locatiecluster tot locatie-tijdcluster.



Figuur 19: Het eindresultaat: de verfijnde, tweedimensionale clusters.

6.1 Runtime

Na verschillende optimalisaties is het Poisson-algoritme erg snel: de runtime is bij de meeste circuits minder dan 10ms. (Dit is gemiddeld 100 maal sneller dan het *inladen* van de circuit-data!) Een volledige vergelijking van runtimes wordt later in dit verslag gegeven. Een aantal redenen waarom dit algoritme zo snel is:

- Er wordt geen 2D-discretisatie uitgevoerd. Er is een 1D-discretisatie om locatieclusters te vinden, en dan per cluster en nuster weer een 1D-discretisatie.
- Wanneer het circuit geen clusters bevat, wordt Deel 2 nooit uitgevoerd, wat weer een tijdsbesparing oplevert. Hierdoor is het een **snelle manier om te testen of een circuit nominaal is**, wat bij de meeste circuits het geval is.
- Gebruikmakend van de constante bakjesgrootte bij discretisatie, hebben we een eigen discretisatie-algoritme kunnen maken dat gemiddeld **3 maal sneller** is dan `numpy.histogram`.
- Dankzij verdere optimalisaties zijn alle stappen *behalve* de discretisatie slechts 10% van de totale runtime.

7 Pinta-algoritme

Het Pinta algoritme vindt de clusters met behulp van het verschil in verdeling van de PD's in de ruis en in de clusters zelf. Het algoritme is gebaseerd op eigen observaties van de verdeling van de datasets en is bedoeld om relatief snel te werken.

7.1 Definities

We beginnen met de dataset in twee dimensies discretiseren. Zij

$$X = \{(t, x) \mid \text{er is een partial discharge op tijdstip } t \text{ en locatie } x\} \subseteq \mathbb{R}^2$$

de dataset. Zij $\pi_1, \pi_2 : \mathbb{R}^2 \rightarrow \mathbb{R}$ de projectieafbeeldingen. Noem $T_{\max} = \max\{\pi_1(X)\}$, $T_{\min} = \min\{\pi_1(X)\}$, $L_{\max} = \max\{\pi_2(X)\}$ en $L_{\min} = \min\{\pi_2(X)\}$. Dan definiëren we

$$T = T_{\max} - T_{\min}$$

als het *tijdsinterval* van de dataset en

$$L = L_{\max} - L_{\min}$$

als de *lengte* van de dataset. Gegeven een tijdsinterval van lengte $0 < l_T < T$ en een plaatsinterval van lengte $0 < l_L < L$ en zij $N_T = \lfloor \frac{T}{l_T} \rfloor$ en $N_L = \lfloor \frac{L}{l_L} \rfloor$. Dan kan weet men dat

$$X \subseteq \bigcup_{\substack{i \in \{0, 1, 2, \dots, N_T\} \\ j \in \{0, 1, 2, \dots, N_L\}}} I_{T_i} \times I_{L_j}$$

met

$$I_{T_i} = [T_{\min} + i * l_T, T_{\min} + (i + 1) * l_T], I_{L_j} = [L_{\min} + j * l_L, L_{\min} + (j + 1) * l_L]$$

Notatie: We geven de doos $I_{T_i} \times I_{L_j}$ ook wel aan als doos D_{ij} . Dan is het aantal partial discharges in een "hokje" gegeven door:

$$P_{ij} = \#(X \cap D_{ij})$$

Het Pinta algoritme zoekt naar alle paren i en j waarvoor P_{ij} 'abnormaal' hoog is. We zeggen dan dat de doos D_{ij} tot een cluster behoort. We leggen een equivalentierelatie op op de dozen:

$$D_{ij} \sim D_{kl} \iff P_{ij} = P_{kl}$$

Dit is inderdaad een equivalentierelatie, zoals men gemakkelijk nagaat. We doen de volgende aanname:

- Als doos D_{ij} niet tot een cluster behoort, dan is de kans zeer groot dat geldt: $\#[D_{ij}] > 1$.

Dit valt te motiveren: het willekeurig optreden van een partial discharge als gevolg van ruis is een zeldzame gebeurtenis. Het is dus niet onredelijk te veronderstellen dat de P_{ij} Poisson verdeeld zijn, als geldt dat er niets aan de hand is met de kabel. Er geldt:

$$\mathbb{P}(P_{ij} = k) = \frac{\lambda^k}{k!} e^{-\lambda}$$

met

$$\lambda = \mathbb{E}(P_{ij})$$

Uit analyse van de data blijkt dat λ klein is. De kans dat men een doosje vindt met slechts een paar partial discharges is dus zeer groot. Gezien de data wordt opgedeeld in erg veel hokjes moet dus wel gelden dat $\#[D_{ij}] > 1$ voor de meeste hokjes met ruis. Merk hierbij op dat het om de *meeste* hokjes met ruis gaat. Het algoritme houdt maar in beperkte mate rekening met hokjes die per toeval de eigenschap hebben dat $\#[D_{ij}] = 1$.

7.2 Algoritme

De eerste stap van het algoritme is het discretiseren van de data volgens bovenstaande methode. Daarna kijkt het algoritme naar de grootte van de equivalentieklassen $[D_{ij}]$. Hoe groter de equivalentieklasse, hoe zekerder het algoritme is dat de bijbehorende dozen tot ruis behoren. Aan de andere kant, als $\#[D_{ij}] = 1$, dan zit het doosje hoogstwaarschijnlijk in een cluster. Hoe meer partial discharges er in dat doosje zitten, hoe zekerder we er van zijn dat het doosje ook echt tot een cluster behoort. Om dit preciezer te maken sorteert het algoritme de doosjes op aantal PD's. De gesorteerde dozen geven we aan met D_k , waarbij een paar (i,j) dus correspondeert met een rangnummer k . We definiëren P_k op de volgende wijze:

$$P_k := \#(X \cap D_k)$$

Dan geldt dus dat $P_1 \leq P_2 \leq P_3 \leq \dots \leq P_{N_T * N_L}$. Voor elk rangnummer k definiëert het algoritme de volgende score S_k :

$$S_k = P_k - C * k$$

waarbij C een door de gebruiker gekozen gevoeligheidsconstante is. We bekijken het verschil in S_k tussen twee opeenvolgende rangnummers:

$$S_k - S_{k-1} = (P_k - P_{k-1}) - C$$

We zien dat als $P_k = P_{k-1}$, dan is $S_k > S_{k-1}$. S daalt dus als er veel dozen zijn met hetzelfde aantal PD's. Als het verschil $P_k - P_{k-1}$ groot is, stijgt S . Als de equivalentieklasse $[D_k]$ dan niet te groot is, zal S daarna dus ook niet ver meer dalen. Het blijkt dat het minimum van S een goede maat geeft voor de grens tussen clusters en ruis. De dozen na het minimum zullen de eigenschap hebben dat niet veel andere dozen hetzelfde aantal PD's hebben, en bovendien dat het verschil tussen P_k en P_{k-1} groot is. De dozen voor het minimum hebben de eigenschap dat er veel andere dozen zijn met hetzelfde aantal PD's, anders zou de grafiek niet dalend zijn. Het algoritme markeert de dozen D_k met $k > \text{argmin}(S)$ als clusters. Vervolgens markeert het dozen die naast elkaar liggen, en beide tot een cluster behoren, als één cluster.

8 DBSCAN-algoritme

Een van de eerste dingen die we hebben gedaan na ons eerste gesprek met Sander, was onderzoek doen naar bestaande cluster-algoritmes op het internet. De meeste algoritmes vielen gelijk af, omdat deze als parameter het aantal te vinden clusters hebben, terwijl we dat van tevoren niet weten. Eén algoritme trok echter gelijk onze aandacht, omdat het aan de volgende cruciale eigenschappen voldoet:

- het aantal te vinden clusters is geen parameter
- het algoritme kan clusters van verschillende vormen vinden
- ruis wordt gemakkelijk gedetecteerd

De naam van dit algoritme is DBSCAN, wat staat voor 'Density-Based Spatial Clustering of Applications with Noise'. Voordat de stappen waarin het algoritme te werk gaat worden uitgelegd, eerst 2 definities:

- epsilon-omgeving van een datapunt x : een klein gebiedje rondom het datapunt x . In het algemeen is dit een cirkel met straal epsilon, en middelpunt x . Hierbij zijn epsilon en de vorm van het gebied parameters van DBSCAN
- kernpunt: een datapunt x heet een kernpunt als er in de epsilon-omgeving van x minstens y punten liggen. Hierbij is y een parameter van DBSCAN

Het algoritme gaat volgens de volgende stappen te werk:

1. Kies een willekeurig datapunt dat nog niet eerder gekozen is. Kijk of er in de epsilon-omgeving van dat punt minimaal y punten liggen. Als dit niet het geval is, label het punt dan als ruis (het punt later nog een randpunt van een cluster worden), en herhaal stap 1. In het andere geval, begin een nieuw cluster en label het punt als kernpunt van dit cluster, en ga naar stap 2
2. Voeg alle datapunten uit de epsilon-omgeving van het kernpunt toe aan het cluster. Als een punt eerder was gelabeld als ruis, label het dan nu als randpunt. Check voor al deze punten of het kernpunten zijn en zo ja, label ze als kernpunt
3. Voor ieder nieuw kernpunt, herhaal stap 2
4. Herhaal stap 2 en 3 totdat ieder punt uit het cluster een label heeft
5. Het cluster is nu af. Ga weer terug naar stap 1, of stop als alle datapunten zijn gelabeld als kernpunt, randpunt of ruis.

8.1 Eerste versie van het algoritme

Ons eerste idee was om de stappen zoals ze hierboven vermeld staan, te vertalen naar code. Dit was echter lastig, onder andere omdat je in Python niet zomaar labels aan punten toe kan kennen. Om het simpel te houden, begonnen we met een versie die alleen de locatie van de ontladingen beschouwde. Deze versie deed ongeveer 10 minuten over het scannen van één circuit, en dat is zo langzaam dat we de resultaten van het algoritme verder niet meer hebben onderzocht. Op dit punt hadden we besloten dat DBSCAN waarschijnlijk te lastig was om te implementeren.

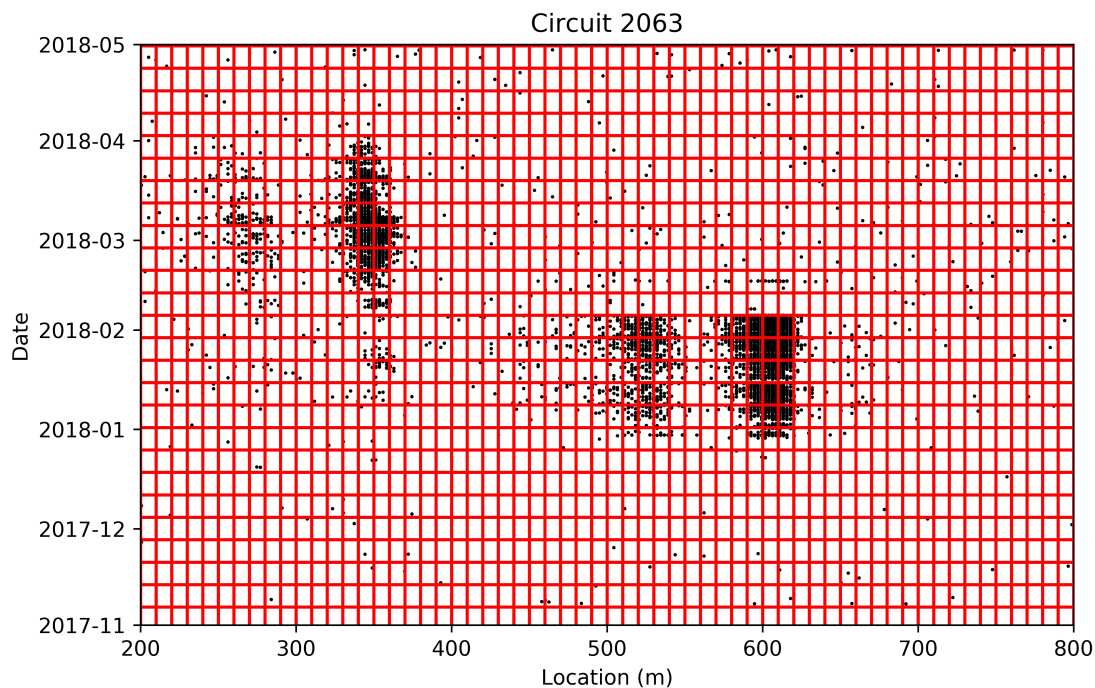
In ons gesprek bij Alliander had Sander ook scikit-learn genoemd, een Python module voor machine learning. (Pedregosa et al. (2011)) Een paar weken na de DBSCAN mislukking zijn we begonnen met onderzoek naar scikit. Deze module bleek ook een DBSCAN algoritme te bevatten. Om bekend te worden met scikit, leek het ons een goed idee om te kijken of we het DBSCAN algoritme konden implementeren op onze data. Hoewel we dit vooral hadden gedaan om wat meer over scikit te leren, bleek het nieuwe algoritme verassend goed te werken. Niet alleen was het veel sneller dan onze eerste poging (nu ongeveer 10 seconden per circuit), ook was het nu eenvoudig om van één- naar tweedimensionale clustering te gaan. Ook bleken er parameters te bestaan waarop het algoritme op onze gelimiteerde hoeveelheid circuits resultaten gaf waar wij tevreden mee konden zijn. Dit deed ons besluiten om bij ons tweede bezoek aan Alliander dit algoritme te laten zien aan Sander.

Bij het gesprek met Sander bleek dat hij ook al onderzoek had gedaan naar DBSCAN, wat ons het gevoel gaf dat we op het goede spoor zaten. We lieten de plaatjes zien die ons algoritme gegenereerd had,

en Sander zij dat het fijn zou zijn als we dit algoritme in de module zouden plaatsen die we uiteindelijk aan Alliander leveren. Ook had hij nog de suggestie gegeven dat als we eerst de data discretiseren, het algoritme waarschijnlijk sneller is. Het idee hierachter is dat je punten die dicht bij elkaar liggen samenvoegt tot één punt met een gewicht dat correspondeert met het aantal samengevoegde punten. Vervolgens pas je een gewogen DBSCAN toe op deze nieuwe verzameling punten, en omdat dit er minder zijn zou dit sneller moeten gaan.

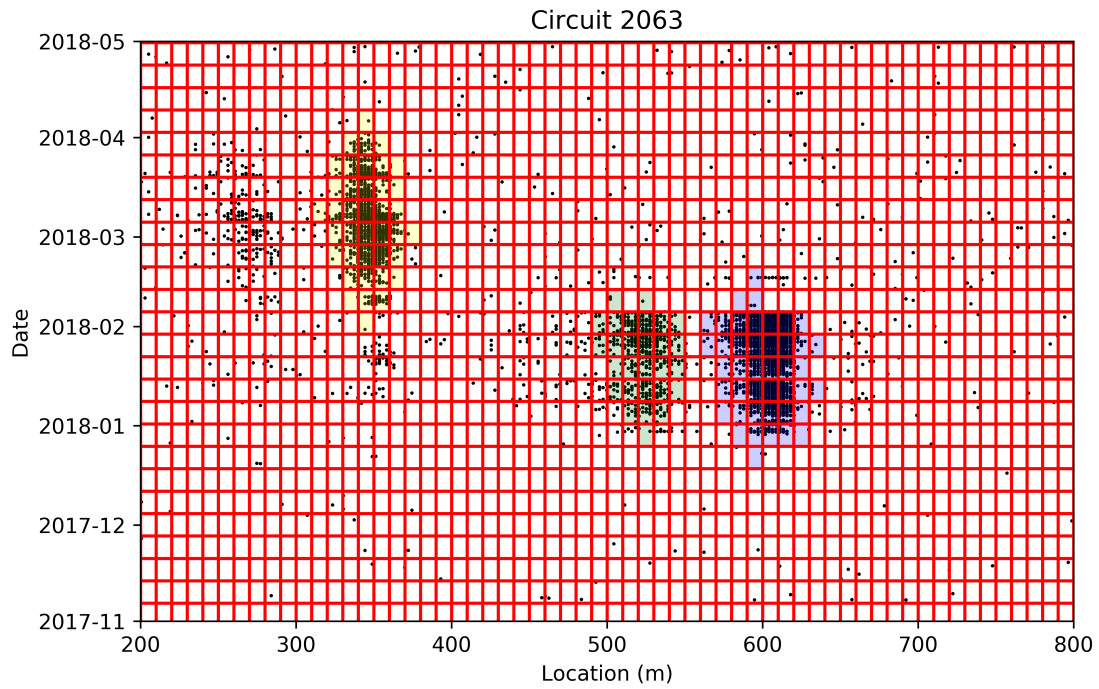
8.2 Optimalisatie

In de weken na ons tweede gesprek bij Alliander zijn we bezig geweest met optimalisatie van het DBSCAN-algoritme. De belangrijkste stap was hierbij het discretiseren van onze data voordat de DBSCAN hierop wordt toegepast. We hebben ervoor gekozen om de data te discretiseren door er een histogram van te maken. Ieder bakje krijgt dan een gewicht gelijk aan het aantal datapunten in dat bakje, en vervolgens worden de bakjes omgezet tot punten die in de DBSCAN gaan. Dit wordt gedaan door een rooster van punten over de bakjes heen te leggen, waarbij ieder bakje precies één roosterpunt bevat. We kwamen er al snel achter dat het algoritme niet meer nauwkeurig is als we te weinig bakjes gebruikten. We kregen pas weer goede resultaten als het aantal bakjes ongeveer net zo groot was als het aantal datapunten. Omdat hier natuurlijk geen tijdswinst mee geboekt wordt, moesten we iets anders verzinnen. Toen we beter naar het histogram gingen kijken, merkten we op dat meer dan 90 procent van de bakjes leeg was als we een groot aantal bakjes namen. Door eerst deze bakjes, die als datapunt met gewicht nul in de DBSCAN gingen, weg te gooien, boekten we een aanzienlijke tijdswinst: het algoritme duurde nu gemiddeld nog maar één seconde per circuit.

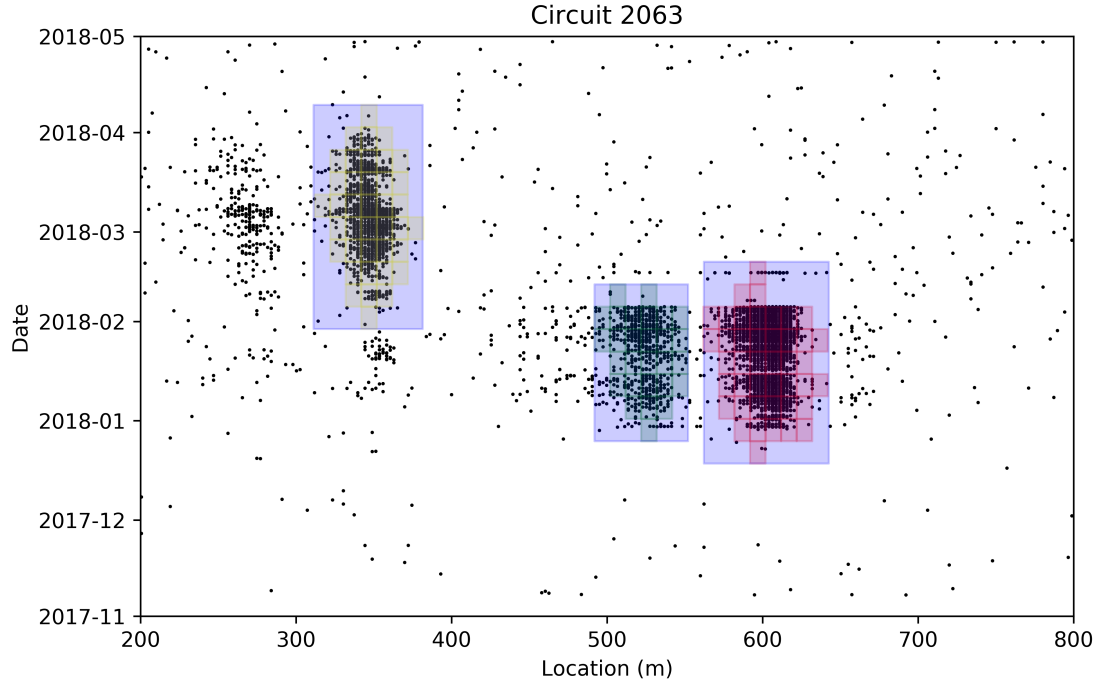


Figuur 20: histogram maken van de data

De resultaten van de DBSCAN zijn niet direct bruikbaar. Voor ieder rechthoekje in het histogram wordt aangegeven of het in een cluster zit en zo ja, in welk cluster. Zo zijn de clusters dus een verzameling kleine rechthoekjes. Wij hebben echter de keuze gemaakt om onze clusters te definiëren als één rechthoek, omdat dit overzichtelijker is. Om geen informatie te verliezen, wordt de 'bounding box' van de rechthoekjes genomen. Dit is de kleinste rechthoek die alle kleine rechthoekjes omvat.



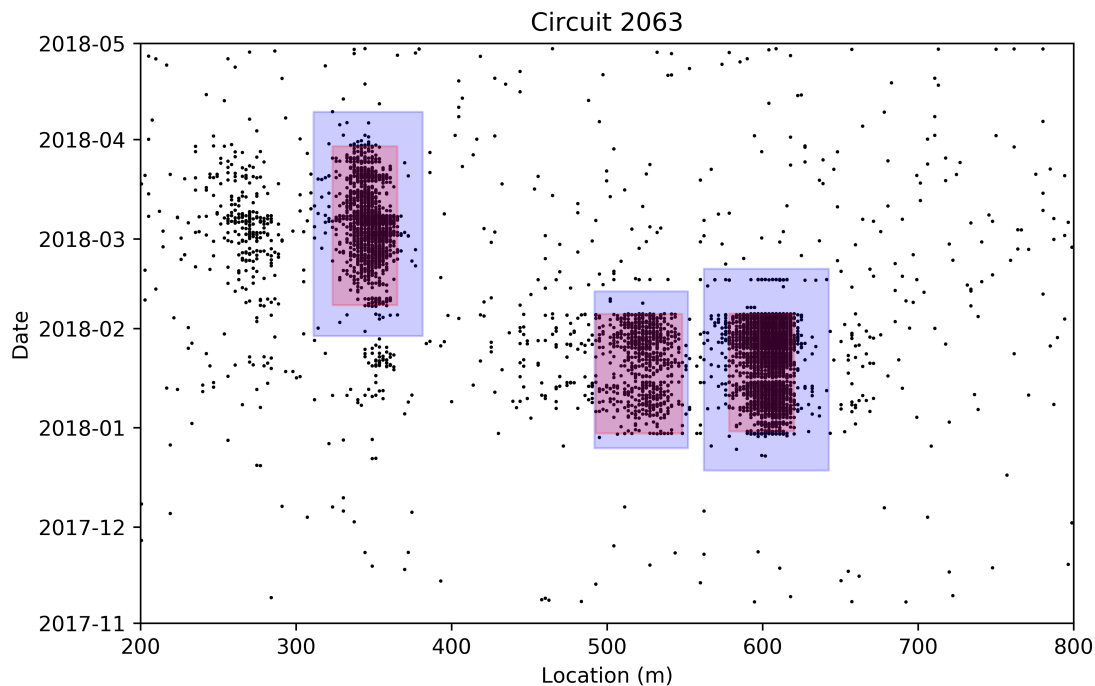
Figuur 21: resultaten van DBSCAN



Figuur 22: de 'bounding box'

Ondanks dat er op deze manier geen informatie wordt weggegooid, is deze methode niet perfect. Als er bijvoorbeeld een kabel moet worden gerepareerd, dan is de bounding box groter dan je zou willen. Om het cluster kleiner te maken en daarbij het informatieverlies te minimaliseren, hebben we de 'shave' parameter bedacht. Als deze niet op 0 staat, zoekt het algoritme naar alle datapunten in de bounding

box. Daarna haalt het aan iedere kant van de rechthoek een percentage van de punten weg, afhankelijk van de grootte van de shave parameter. met de shave parameter op één procent worden de clusters al beduidend kleiner, zonder dat er veel informatie verloren gaat.



Figuur 23: de bounding box en het bijgeschaafde cluster

8.3 Parameters

Het DBSCAN algoritme gebruikt de volgende parameters:

- epsilon: de grootte van de epsilon-omgeving die bij de DBSCAN gebruikt wordt. De standaard waarde is 3. Een grotere epsilon zorgt ervoor dat er meer datapunten onderdeel zijn van een cluster, en clusters die dicht bij elkaar liggen worden aan elkaar vast gemaakt
- minpts: het aantal punten in de epsilon-omgeving van een punt om aangemerkt te worden als kernpunt. De standaardwaarde is 125. Als minpts groter is, zijn er minder datapunten onderdeel van een cluster. Dit is de parameter waarmee gevoeligheid het makkelijkst kan worden ingesteld
- binlengthx: de lengte van de bakjes van het histogram in de x-richting, in meters. De standaardwaarde is 2. Een grotere binlengthx zorgt voor bredere clusters
- binlengthy: de lengte van de bakjes van het histogram in de y-richting, in weken. De standaardwaarde is 1. Een grotere binlengthy zorgt voor langere clusters
- shave: het percentage van punten dat aan iedere kant (onder, boven, links, rechts) weg wordt gehaald van de oversized clusters. De standaardwaarde is 0.01. Een grotere shave zorgt voor kleinere clusters

9 Ensemble-model

We willen de algoritmes combineren in een *ensemble-model*. Dit model moet op basis van de output van de andere algoritmes een nieuwe output geven. Tot nu toe zagen we dat clusters bestaan uit rechthoeken. De output van de andere algoritmes is hier ook op gebaseerd. Om de verschillende algoritmes te kunnen combineren, moeten we twee dingen doen. Namelijk:

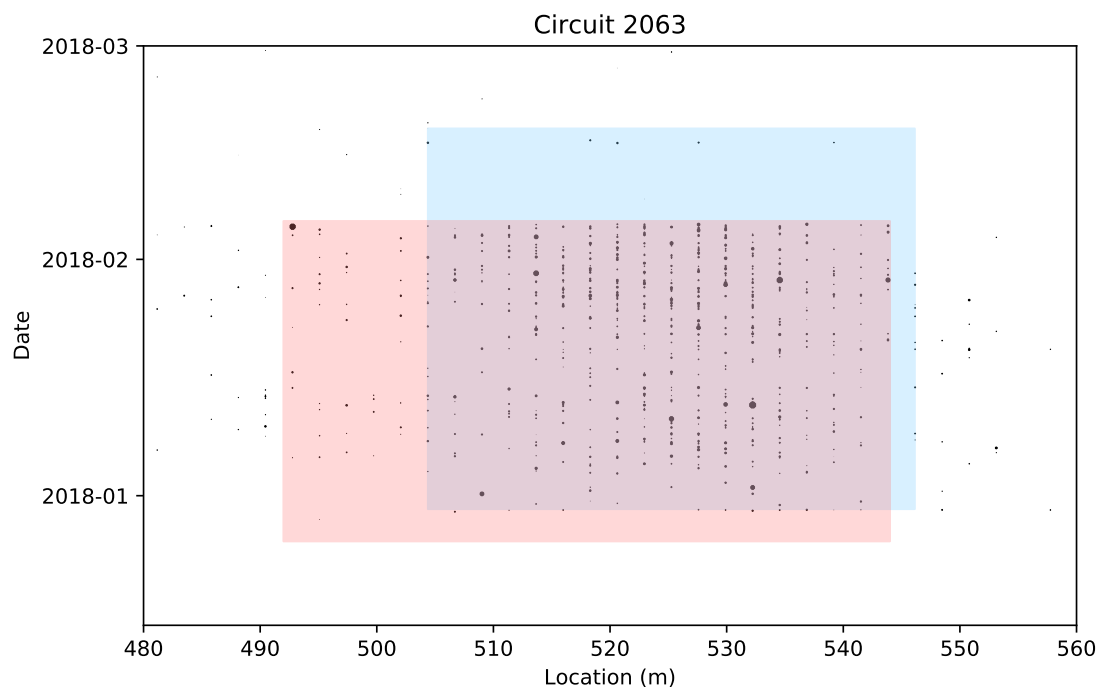
1. Rechthoeken combineren
2. De combinatie betekenis geven
3. Clusters aangeven

Tot nu toe hebben we steeds gezegd dat een cluster een rechthoek is in de tijd en afstand. Maar door de verschillende manieren waarop we rechthoeken kunnen combineren zal dit niet meer genoeg blijken. Hiervoor hebben we dus een nieuwe definitie van een cluster nodig.

9.1 Combineren van rechthoeken

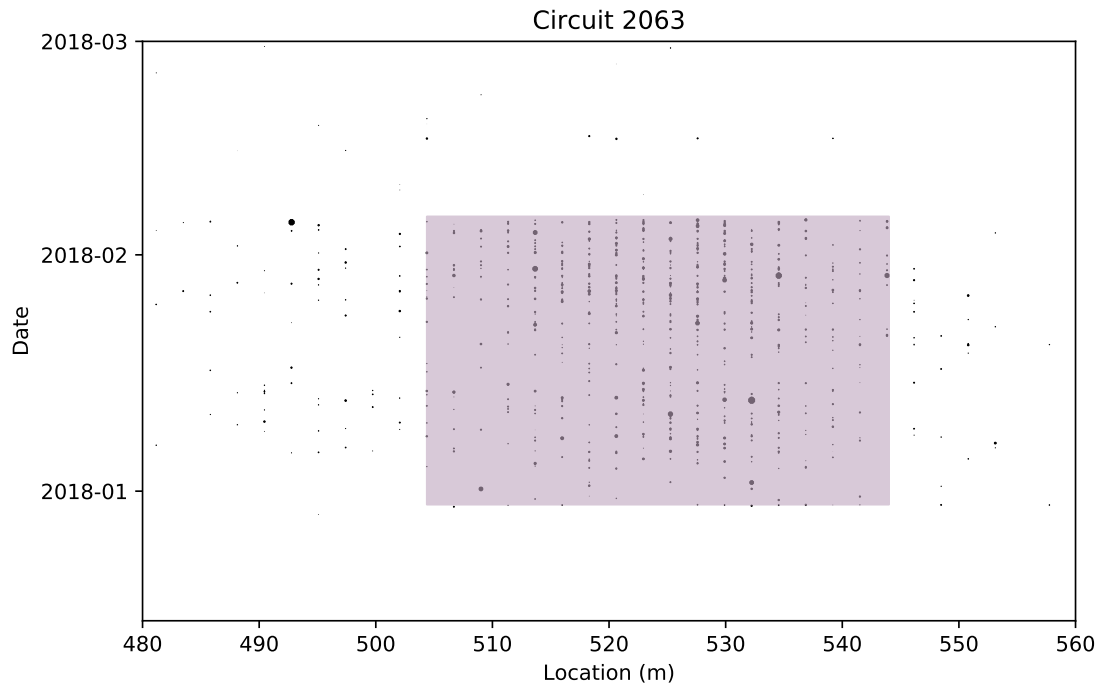
Om rechthoeken te combineren voeren we een nieuwe klasse in: `Rectangle`. Deze staat in `rectangle.py`. Een `Rectangle` heeft in beginsel twee eigenschappen: `location_range` en `time_range`. `location_range` is een tuple van een beginlocatie en eindlocatie. Tussen deze twee locaties ligt dan (als de algoritmes goed werken) ongewoon veel partial discharges. `time_range` is precies hetzelfde, maar dan in de tijd. De Python klasse stelt ons ook in staat om nieuwe functies te definiëren, zoals `__str__`, wat een mooie textrepresentatie van een `Rectangle` geeft, en `get_partial_discharges`, wat van een input circuit zegt welke partial discharges binnen de `Rectangle` liggen.

We hebben een aantal methoden bedacht om `Rectangles` te combineren. Namelijk: `&` (`--and--`), `|` (`--or--`) en `+` (`--add--`, maar uitgesproken als: plus). De werking van deze methoden valt het best te begrijpen door te kijken naar plaatjes van de resultaten.



Figuur 24: Rechthoeken van Poisson en DBSCAN

In 24 zien we de rechthoeken van Poisson (Rood) en DBSCAN (Blauw) die gevonden worden voor een bepaald cluster. Zoals te zien is, overlappen ze, maar hebben ze ook beide een gedeelte wat niet overlapt met de andere rechthoek. We kunnen op deze rechthoeken nu onze combinatiemethoden loslaten om te laten zien wat ze doen.



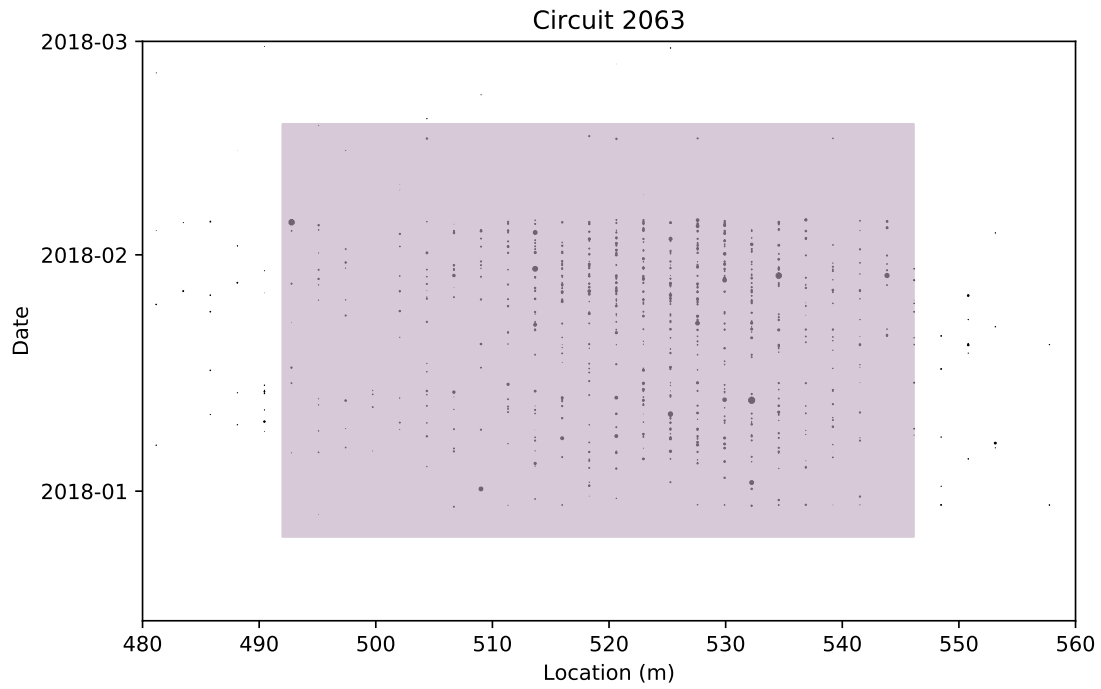
Figuur 25: Resultaat van &

In 25 zien we wat er gebeurt nadat we $\&$ toepassen op de rechthoeken. Alleen de plekken waar de rechthoeken overlappen blijft nog over. Strikt gesproken: Voor de linkergrens in de locatie-dimensie nemen we het maximum van de twee linkergrenzen van de rechthoeken, terwijl we voor de rechtergrens het minimum nemen van de rechtergrenzen van de rechthoeken. Hetzelfde doen we in de tijd met begin- en eindtijd.

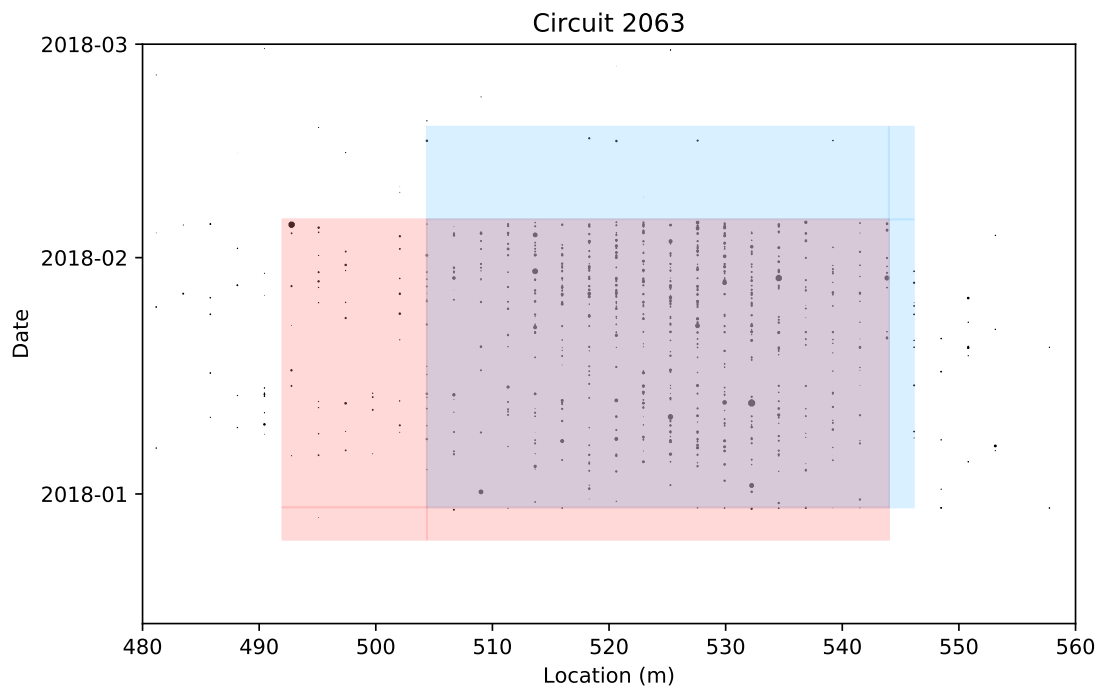
In 26 zien we wat er gebeurt nadat we $|$ toepassen op de rechthoeken. Als resultaat krijgen we de zogenaamde bounding box van de rechthoeken waarmee we begonnen. Deze rechthoek is dus altijd groter dan het resultaat van de $\&$, en ook dan de input-rechthoeken. Voor de linkergrens nemen we dit keer het minimum, terwijl we voor de rechtergrens het maximum nemen.

In 27 zien we wat er gebeurt nadat we $+$ toepassen op de rechthoeken. In het midden zien we de overlap, met daaromheen de kleinere rechthoeken die nergens mee overlappen. Het resultaat is een soort Venn diagram van rechthoeken. De rechthoekjes die niet overlappen zijn ook opgesplitst in kleinere rechthoeken, terwijl ze gewoon bij elkaar horen. Dit is gedaan om zodat ze een rechthoekige vorm houden. Er zijn nog rechthoekjes die we samen zouden kunnen voegen (bijvoorbeeld de twee meest linkse rechthoeken), omdat ze een gemeenschappelijke `location_range` of `time_range` en in de andere dimensie elkaar raken, maar we hebben ervoor gekozen om dit niet te doen, omdat het niet veel toevoegt aan het eindresultaat. Het zou wel een goede manier zijn om het aantal rechthoeken te minimaliseren. Iets anders wat opvalt is dat het resultaat bestaat uit meerdere rechthoeken. Bij de $\&$ en de $|$ is het resultaat steeds maar 1 nieuwe rechthoek. In het geval van de $+$ kunnen er maximaal 9 rechthoeken ontstaan. In 9.3 wordt besproken hoe dit probleem wordt opgelost.

De $+$ is van de 3 methoden het moeilijkst om te berekenen. Bij $\&$ en $|$ hoeven we alleen maar minima en maxima van de grenzen te nemen als nieuwe grenzen. Bij $+$ hangt het resultaat echter sterk af van de manier waarop de rechthoeken overlappen. Is er een compleet bevat in de ander? Of is er een kleine overlap in één van de hoeken? Of misschien snijdt de ene rechthoek de andere rechthoek doormidden. Om dit probleem op te lossen introduceren we een hulpfunctie: `-(__sub__,` maar uitgesproken als: `min`). In 28 en 29 zien we respectievelijk de resultaten van DBSCAN - Poisson en Poisson - DBSCAN. `-` is dus niet commutatief zoals de andere methoden om rechthoeken te combineren. De $+$ maken is nu makkelijk: Neem de twee mogelijke versies van `-`, en voeg daar de overlap ($\&$) aan toe. Maar nu is het probleem verschoven: We moeten de `-` implementeren. Stel we hebben rechthoeken A en B en we willen `A - B` berekenen. Een eerste observatie die ons helpt is dat we bij niet rekening hoeven te houden met heel B. De delen van B die niet overlappen met A, hebben namelijk geen effect op het resultaat. Dus:

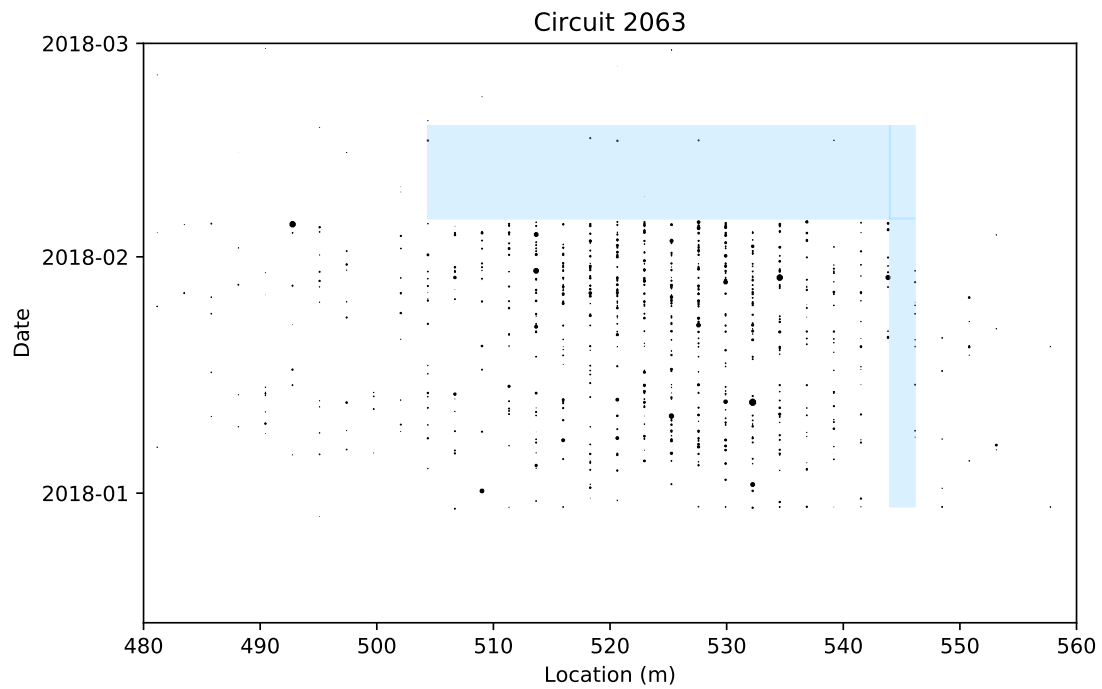


Figuur 26: Resultaat van $|$

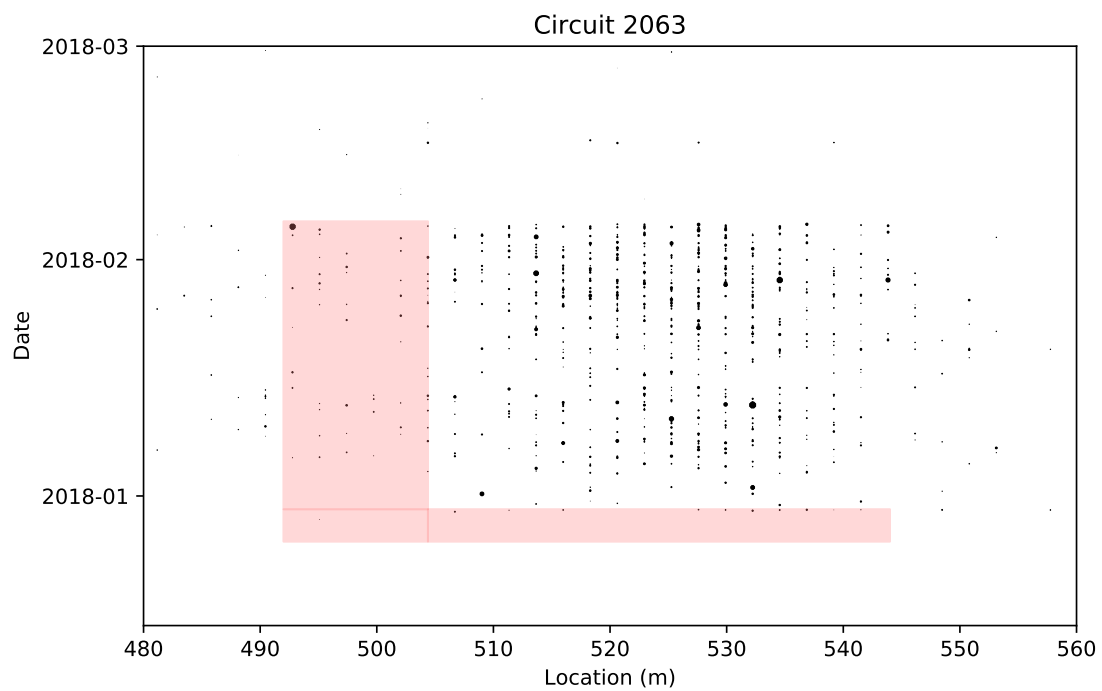


Figuur 27: Resultaat van $+$

$A - B$ berekenen is equivalent aan $A - (A \& B)$ berekenen. Nu is wanneer de meerdere manieren waarop $A - (A \& B)$ kunnen overlappen om de hoek kunnen kijken: In een hoek, aan een zijde, van boven naar onder, van links naar rechts, of alleen in het midden. Deze verschillende manieren zijn allemaal echter een speciaal geval van de laatste manier: $(A \& B)$ is compleet bevat in A . We kunnen dus de 9 rechthoeken berekenen die voor dit ene geval nodig zijn (hiervoor gebruiken we weer de minima en



Figuur 28: Resultaat van -, optie 1



Figuur 29: Resultaat van -, optie 2

maxima van de grenzen), en vervolgens kunnen we kijken in welk geval we zitten. Het handige is: Als de overlap ergens een zijde raakt, dan is de rechthoek die we berekend hebben aan die kant van het midden leeg. Neem bijvoorbeeld het geval dat (A & B) in de rechterbovenhoek van A zit. (A & B) raakt dan de rechts A. De rechthoeken aan de rechterkant van het midden hebben als locatie-dimensie allemaal als linkergrens de rechtergrens van (A & B) en als rechtergrens de rechtergrens van A. In dit

geval komen deze overeen, dus de resulterende rechthoeken zijn leeg. Op analoge wijze geldt dit ook voor de rechthoeken boven het midden. We houden dus nog vier rechthoeken over: De overlap ($A \& B$), een stukje links van de overlap, een stukje onder de overlap, en de hoek links-onder. De rest is allemaal leeg. Bij de $-$ zijn we niet geïnteresseerd in de overlap, dus als resultaat hebben we nu de 3 overgebleven rechthoeken die niet leeg zijn. Als A en B op een andere manier overlappen, dan geldt op analoge wijze dat we ook alleen maar geïnteresseerd zijn in de niet-lege rechthoeken, met uitzondering van de overlap.

Deze bovenstaande functies zijn in Python allemaal geïmplementeerd met operator overloading. Je kunt dus om de overlap tussen rechthoeken A en B te berekenen in Python gewoon `overlap = A & B` typen, en de $+$ kun je toepassen met `plussed = A + B`.

`Rectangle` heeft ook nog een handige hulpfunctie `disjunct`. Deze geeft aan wanneer rechthoeken disjunct zijn; wanneer ze niet overlappen. `A.disjunct(B)` is daarom equivalent aan `(A & B) is None`, waarbij `None` het staat voor een lege rechthoek.

9.2 Betekenis van de combinatie

Nu we de operatoren `&`, `|` en `+` op rechthoeken hebben gedefiniëerd, moeten we er een betekenis aan toekennen. Hiervoor hebben we 3 mogelijkheden bedacht:

1. Een gewicht, waarbij hoger beter is
2. Een kans, bijvoorbeeld op basis van Bayesiaanse statistiek
3. Een lijst met de algoritmes waardoor de rechthoeken gevonden zijn

Uiteindelijk hebben we ervoor gekozen om de laatste optie te implementeren. De andere twee vielen af. Dit was omdat:

Gewicht) Dit is misschien de meest voor de hand liggende manier om rechthoeken betekenis te geven. Als het door één algoritme gevonden is, geef het gewicht 1; bij twee algoritmes geef je het gewicht 2, enzovoort. Echter is een lijst van algoritmes beter: Het geeft meer informatie. Een lijst met algoritmes kun je namelijk altijd omzetten naar een gewicht door de lengte van de lijst op te vragen. Andersom kun je een gewicht niet omzetten naar een lijst met algoritmes (tenzij je het gewicht op een manier berekent waardoor het de facto een lijst algoritmes wordt). Een gewicht toekennen viel dus af omdat een lijst met algoritmes altijd beter is.

Kans) Een kans is in feite ook een soort gewicht, maar het geeft net iets meer informatie. Het probleem met deze manier van combineren is dat de onderliggende algoritmes ook een kans moeten bedenken voor hun clusters. In het geval van het Poisson algoritme lukt dat, maar bijvoorbeeld DBSCAN heeft geen manier om de kans dat een cluster correct is weer te geven. Hierdoor zou je iets artificiëls moeten bedenken om dit soort algoritmes toch een kans toe te kennen, maar dit zal waarschijnlijk nooit een goede schatting worden. Een kans viel dus af omdat het onpraktisch en bijna niet te implementeren is.

Uiteindelijk hebben we er dus voor gekozen om een lijst van algoritmes bij te houden bij elk cluster. Ieder algoritme heeft hiervoor een nieuwe, optionele parameter `name` gekregen, die de naam van het algoritme aangeeft. Ieder algoritme heeft hiervoor een standaardwaarde (Poisson 1D, Poisson 2D, DBSCAN, Pinta). De klasse `Rectangle` heeft hiervoor een nieuw attribuut `found_by`, wat aangeeft door welke algoritmes het gevonden is. `found_by` is een gewoonlijk een set van strings, waarbij iedere string de naam van een algoritme is. Tijdens het berekenen van de `&` en `|` worden die sets bij elkaar opgeteld. Bij de `+` krijgt de overlap beide de som de `found_bys` als eigen `found_by`, terwijl alle losse rechthoekjes hun eigen `found_by` behouden.

Indien een nieuwe algoritme wordt ontwikkeld, moet deze dus ook een eigen `name` meegeven aan alle rechthoeken die worden gevonden. Indien een een algortime wordt gemaakt door de standaardparameters van een oud algoritme aan te passen, kun je dit voor de duidelijkheid ook aangeven met een andere `name`. Bijvoorbeeld:

```
lambda circuit: clusterize_pinta(circuit,
                                sensitivity=100,
                                name="Pinta high sensitivity")
```

is hoe je een Pinta algoritme kunt maken met een hogere sensitivity. De clusters die gevonden worden door dit algoritme zullen dan ook als `found_by` de set `{"Pinta high sensitivity"}` hebben. Als dit cluster overlap heeft met een ander cluster dat gevonden is door de normale pinta, wordt de `found_by`:

`{"Pinta", "Pinta high sensitivity"}`. Hierdoor kun je dus aan de `found_by` zien dat het gevonden is door twee verschillende versies van het Pinta algoritme. Als de `name` parameter niet wordt aangepast, dan zou de combinatie de `found_by`: `{"Pinta"}` hebben en kun je dus onterecht denken dat het door maar één algoritme gevonden is. Denk er dus altijd aan om een goede `name` parameter mee te geven, zodat de output van de combinatie van clusters zinvol is.

9.3 Clusters aangeven

In het stukje over rechthoeken combineren zagen we al dat er 3 manieren zijn om dat te doen: `&`, `|` en `+`. Bij `&` en `|` ontstaan er na de combinatie van 2 rechthoeken steeds 1 nieuwe rechthoek. Bij `+` is dit niet het geval: in het ergste geval kunnen er 9 verschillende rechthoeken ontstaan. Deze 9 rechthoeken horen echter nog wel steeds allemaal bij elkaar: ze vormen samen één cluster. We introduceren hierom een nieuwe definitie van wat een cluster is: een verzameling rechthoeken. In Python is dit geïmplementeerd als de klasse `Cluster`, die als attribuut een `set` van `Rectangle` objecten bevat. In het eenvoudigste geval bevat een `Cluster` object dus een `set`, met daarin één `Rectangle`. Verder is er de aanname dat alle `Rectangles` in het `Cluster` onderling disjunct zijn.

9.3.1 cluster.py

De `Cluster` klasse implementeert zelf ook de operatoren `&`, `|` en `+`. Als deze operatoren op twee `Clusters` worden aangeroepen, dan worden alle `Rectangles` in de twee `Clusters` met elkaar gecombineerd met dezelfde operator. De manier om `Clusters` met bijvoorbeeld de `+` te combineren is dus:

```
added_cluster = poisson_cluster + dbscan_cluster + pinta_cluster
```

De implementatie van `&` is vrij eenvoudig. Om de clusters `c` en `d` te combineren, loop je eerst over alle `Rectangles` in `c`, en dan over alle `Rectangles` in `d`, en pas je de `&` operator toe op de `Rectangles`. Dit kan makkelijk, omdat de overlap tussen twee rechthoeken altijd kleiner is dan de rechthoeken waarmee je begint, en we aannemen dat alle `Rectangles` binnen een `Cluster` disjunct zijn. Een bijkomend voordeel hiervan is dat de `&` erg snel is. Een test op een representatief circuit geeft een runtime van ongeveer 500 microseconden.

Bij `|` en `+` is het iets ingewikkelder. Om te beginnen met `|`: Bij het berekenen van de `|` tussen twee `Rectangles` is het resultaat groter. Hierdoor kan het dus voorkomen dat je een `Rectangle` al bekeken hebt, maar nadat je twee andere `Rectangles` combineerd, kan het grotere resultaat (de bounding box) plotseling toch overlap hebben met het eerste `Rectangle`. Hierdoor moeten alle `Rectangles` nog een keer geïnspecteerd worden. Bij de implementatie is er nog een complicerende factor: Je mag een object niet aanpassen terwijl je met een `for` loop over zijn elementen loopt. Hierdoor wordt de implementatie gecompliceerd met `breaks` en `for-else` statements. Het idee achter de implementatie is als volgt:

Je begint met twee `Clusters` (sets van `Rectangles`): `result` en `helper`. Het uiteindelijke doel is dat `result` het eindresultaat bevat, en `helper` kan daar bij helpen. We beginnen een loop over de elementen in `helper`. Bij iedere loop halen we een `Rectangle`, genaamd `helpercur`, uit `helper`. We weten dat we klaar zijn als `helper` leeg is. Nu gaan we voor iedere `Rectangle` in `result` kijken of ze overlappen met `helpercur`. Indien niet, dan is `helpercur` een losstaande `Rectangle` en kan deze dus toegevoegd worden aan `result`. Als er echter een `Rectangle` in `result` is die wel overlapt met `helpercur`, dan berekenen we de `|` van de twee en verwijderen we de oorspronkelijke `rectangle` uit `result`. Dit resultaat is groter dan de twee originele `Rectangles`, dus kan het overlappen met `Rectangles` uit `result`. Hierom voegen we de nieuwe `Rectangle` toe aan `helper` en beginnen we het algoritme opnieuw.

Dit algoritme geeft het gewenste resultaat. Het enige probleem dat we nu hebben is de vraag: Stopt dit algoritme ook? We gaan namelijk door tot `helper` leeg is, maar in het geval dat twee `Rectangles` overlappen, voegen we het resultaat juist toe aan `helper`. Het antwoord op de vraag is gelukkig: Ja, het stopt altijd. Dit kunnen we bewijzen met inductie:

Noem het aantal `Rectangles` in `result` `n` en het aantal `Rectangles` in `helper` `m`.

We beginnen met inductie naar `m`. Als `m` gelijk is aan 0, dan zijn we klaar: Het algoritme stopt omdat `helper` leeg is.

In de inductiestap mogen we aannemen dat het algoritme stopt als er `m-1` `Rectangles` in `helper` zitten, en moeten we bewijzen dat het ook stopt als we beginnen met `m` `Rectangles` in `helper`. Kijk nu wat er gebeurt in het algoritme: We halen een `Rectangle` uit `helper`, deze heeft hierdoor nog maar `m-1` `Rectangles`.

Als de `Rectangle` geen overlap heeft met een `Rectangle` uit `result`, dan voegen we hem toe aan `result`.

Hierdoor heeft **result** nu $n+1$ Rectangles en heeft **helper** $m-1$ Rectangles. Per inductie stopt het algoritme altijd.

Als de Rectangle wel overlap heeft met een Rectangle uit **result**, dan verwijderen we de Rectangle uit **result** en voegen we het resultaat van een `|` toe aan **helper**. Hierdoor heeft **result** $n-1$ Rectangles en heeft **helper** weer m Rectangles. We kunnen daarom geen gebruik maken van de inductiehypothese. Maak nu de volgende observatie: Het is onmogelijk dat er oneindig lang overlap gevonden wordt een Rectangle uit **result**. Iedere keer dat er een overlappende Rectangle gevonden wordt, wordt het aantal Rectangles in **result** namelijk 1 minder. Dat aantal is eindig, dus na een eindig aantal stappen zijn er 0 Rectangles in **result** over. Op dat moment is het onmogelijk dat er nog een Rectangle in **result** is de overlap heeft met een Rectangle uit **helper**. Het aantal Rectangles verandert daardoor van 0 in **result** en m in **helper** naar 1 in **result** en $m-1$ in **helper**. Vanwege de inductiehypothese weten we nu dat het algoritme stopt.

Als je deze twee resultaten samenvoegt kom je tot de conclusie dat per inductie het algoritme altijd stopt. We hoeven ons dus geen zorgen te maken over oneindige loopjes als resultaat van het aanpassen van de Clusters waarover we lopen.

Een nadeel is wel dat het algoritme langzamer wordt door het opnieuw moeten lopen over Rectangles die al bekeken zijn. Hierdoor is de `|` minder snel dan de `&+` met een runtime van ongeveer 2 milliseconden op hetzelfde circuit waar de `&` ongeveer 500 microseconden over deed. Dat is dus 4 keer langer.

Als laatste hebben we de `+`. Deze heeft een vergelijkbaar probleem met de `|`: Als twee Rectangles overlappen, dan kan het resultaat van de `+` nog steeds overlap hebben met een andere Rectangle, waarvan sommige kunnen overlappen met **result** en andere kunnen overlappen met **helper**. Hierom gebruiken we eenzelfde opzet als voor de `|`. Het enige verschil is dat er nu meerdere Rectangles zijn als resultaat van de `+` op twee Rectangles. Dit mag de pret niet drukken, want het algoritme werkt nog steeds en met een vergelijkbaar argument als hierboven kunnen we ook bewijzen dat het altijd stopt. Een nadeel is wel dat de groei van het aantal Rectangles zorgt voor een langzamere runtime. `+` doet over hetzelfde circuit als we hiervoor gebruikten 28 milliseconden, wat significant langzamer is dan `&` en `|`. Als we meer algoritmes toepassen en dus vaker de `+` toepassen, wordt deze nog langzamer. Ons advies is om het gebruik van `+` te beperken tot het combineren van 2 algoritmes. `+` heeft wel een voordeel ten opzichte van `&` en `|`: Er gaat geen informatie verloren. Bij `&` kijk je alleen naar de overlap, dus de informatie waar de algoritmes niet overlappen gaat verloren. Bij `|` kijk je naar de bounding box, waardoor de exacte vorm van de onderliggende clusters verloren gaat. `+` heeft deze problemen niet, aangezien het een strikte combinatie van de onderliggende clusters is.

Dat was alles over `&`, `|` en `+`. De Cluster klasse implementeert ook nog een paar handige andere functies die gebruikt kunnen worden:

- `__str__` en `__repr__`: Geven een mooie manier om Clusters om te zetten naar text.
- `__bool__`: Geeft aan of een Cluster leeg is. `bool(cluster)` geeft False als het cluster leeg is (dus als het geen Rectangles bevat), en True als het cluster niet leeg is. Analoog aan het gedrag van `__bool__` op andere Python objecten zoals sets en lists.
- `__len__`: Geeft het aantal Rectangles in een Cluster aan. Analoog aan het gedrag van `__len__` op andere Python objecten zoals sets en lists.
- `__iter__`: Verandert een Cluster in een iterable. Hierdoor kun je het gebruiken bij alle dingen in Python die een iterable nodig hebben, zoals for loops. Je kunt dus `for r in my_cluster` gebruiken om over de Rectangles in een cluster te lopen.
- `__hash__`: Geeft een hash van een Cluster. Dit is een integer die geassocieerd wordt met een Cluster. Dit wordt waarschijnlijk nooit aangeroepen door de gebruiker, maar het is toch handig, omdat het een Cluster "hashable" maakt. Hierdoor kan het gebruikt worden als key in dictionaries (wat eigenlijk een hash table is) en sets.
- `get_partial_discharges`: Functie met als input een circuit en als output alle partial discharges in dat circuit die in het Cluster liggen.
- `most_confident`: Functie die de Rectangles in het Cluster geeft met het hoogste aantal `found_by`. Als je twee Clusters hebt gecombineerd met de `+`, kun je op deze manier de overlap vinden. (Het resultaat lijkt op het resultaat van de `&`)

- **get_bounding_rectangle**: Functie die de bounding box van alle Rectangles in Cluster aangeeft. Als je twee Clusters hebt gecombineerd met de `+`, kun je op deze manier het resultaat reduceren naar 1 Rectangle. (Het resultaat lijkt op het resultaat van de `|`)
- **location_range**: Eigenschap van een Cluster. Geeft de begin- en eindlocatie van de bounding box van alle Rectangles in het Cluster. Als het Cluster maar 1 Rectangle bevat, kun je op deze manier makkelijk de **location_range** van die Rectangle opvragen. (Anders zou je dat moeten doen met `list(cluster.rectangles)[0].location_range`. De nieuwe optie `cluster.location_range` is dus een stuk eenvoudiger en leesbaarder)
- **time_range**: Eigenschap van een Cluster. De tijd-versie van **location_range**.
- **found_by**: Eigenschap van een Cluster. De **found_by**-versie van **location_range**.

9.3.2 ensemble.py

Uiteindelijk was het de bedoeling om de resultaten van algoritmes te combineren. De algoritmes vinden echter niet één Cluster per circuit, maar kunnen er meerdere vinden. Hiervoor is het `ClusterEnsemble` bedacht. Dit is een verzameling Cluster objecten, die samen een compleet beeld geven van wat de algoritmes als output geven bij een circuit. De algoritmes geven dus als output een `ClusterEnsemble` object, met daarin (indien er iets aan de hand is) meerdere Cluster objecten, elk met 1 Rectangle object. Er kunnen ook meerdere Rectangles in ieder Cluster in het `ClusterEnsemble` zitten. Deze gelaagde structuur geeft dan aan welke Rectangle objecten bij elkaar horen: ze zitten samen in een Cluster. `ClusterEnsemble` zelf implementeert ook veel van de algoritmes van Cluster. De `&`, `|` en `+` zijn compleet analoog, maar dan een stap hoger. De output van twee algoritmes kun je dus makkelijk combineren dus `&`, `|` of `+` aan te roepen op de resultaten. Bijvoorbeeld:

```
pinta_ensemble = clusterize_pinta(circuit)
poisson_ensemble = clusterize_poisson(circuit)
combined_ensemble = pinta_ensemble + poisson_ensemble
```

Ook `__str__`, `__repr__`, `__bool__`, `__len__`, `__iter__`, `__hash__` en `get_partial_discharges` werken hetzelfde als bij de Cluster klasse.

most_confident werkt niet helemaal hetzelfde als bij Cluster. **most_confident** op een `ClusterEnsemble` geeft namelijk de **most_confident** per Cluster als output. Dit maakt **most_confident** verschillend van de `&`:

Als er twee algoritmes zijn, waarbij er 1 Cluster is met overlap en 1 zonder in de eerste, dan geeft `&` als output alleen de overlap. Het losstaande Cluster wordt dus genegeerd. Als je de `+` toepast, dan wordt het Cluster wel onthouden, en na een **most_confident** is bij het Cluster met overlap alleen nog de overlap over, terwijl het losstaande Cluster in zijn geheel bij de output hoort. Ieder afzonderlijk Cluster in het `ClusterEnsemble` heeft een andere grens voor hoe groot `len(rectangle.found_by)` moet zijn. In het voorbeeld hierboven is dat bij de overlap 2, terwijl het bij het losstaande Cluster slechts 1 is. De `&` vindt alleen het de Rectangle met `len(rectangle.found_by)` gelijk aan 2.

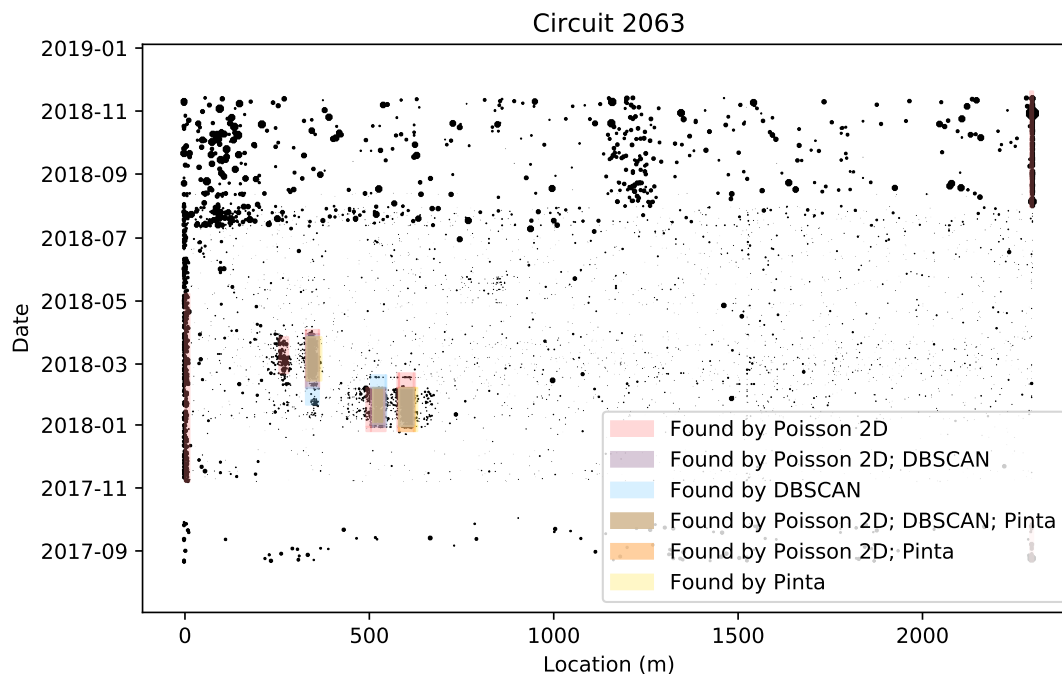
10 Vergelijking

Over het algemeen zijn de algoritmes veel sneller dan we hadden verwacht. Ons streven van 1 seconde per circuit wordt meestal ruim gehaald. Alleen bij circuits met uitzonderlijk veel PD's doet bijvoorbeeld de DBSCAN er iets langer dan een seconde over. Dit is natuurlijk helemaal afhankelijk van de parameters die gekozen zijn, zoals de grootte van de bakjes bij het discretiseren. De resultaten van de algoritmes komen voor een groot deel overeen, maar verschillen ook wel aanzienlijk. Soms worden clusters niet door alle algoritmes gevonden, omdat sommige algoritmes gevoeliger zijn. Dit is geen ramp, want in het ensemblemodel geeft het juist een indicatie van hoe waarschijnlijk het is dat daar daadwerkelijk iets gevaarlijks aan de hand is.

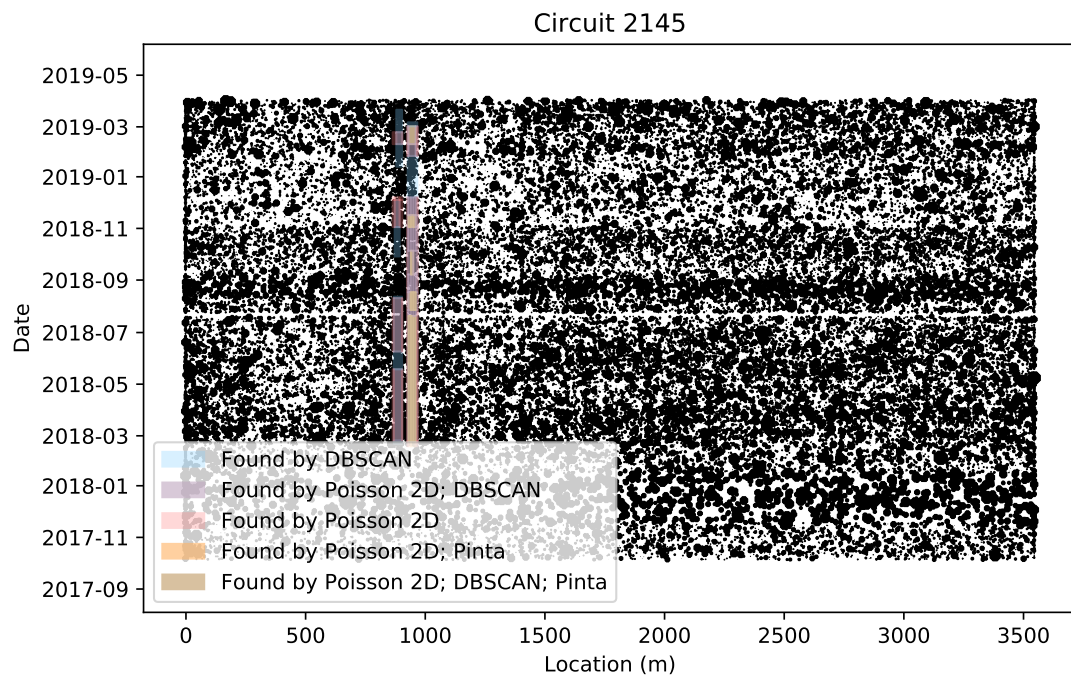
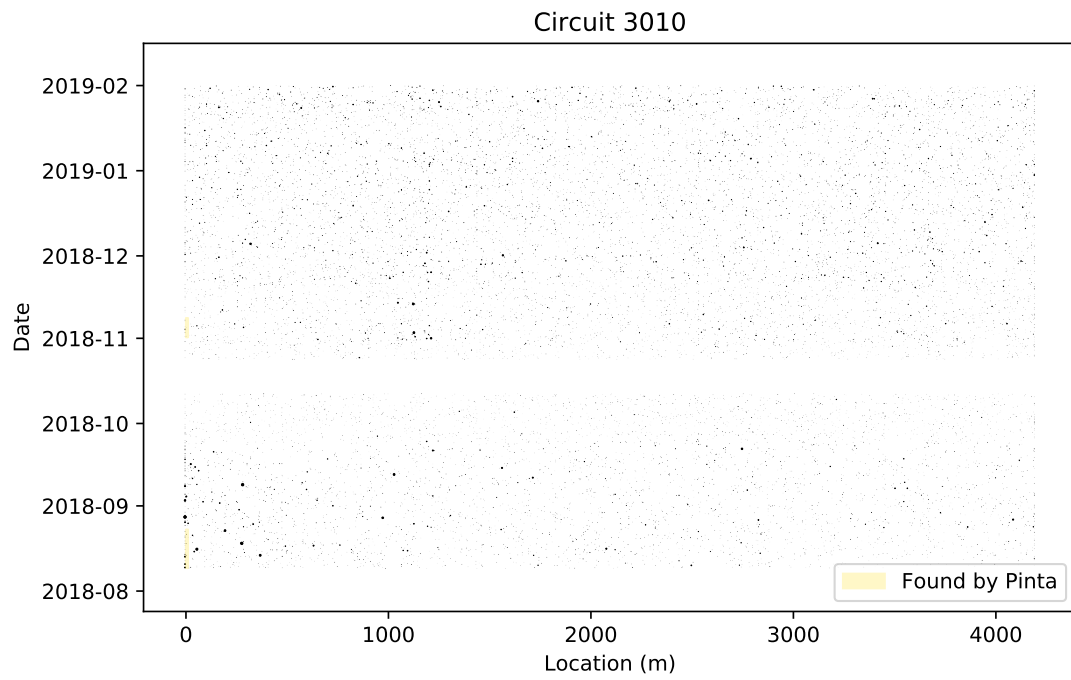
Hieronder volgen de resultaten van een aantal willekeurige circuits (en onze favoriet 2063).

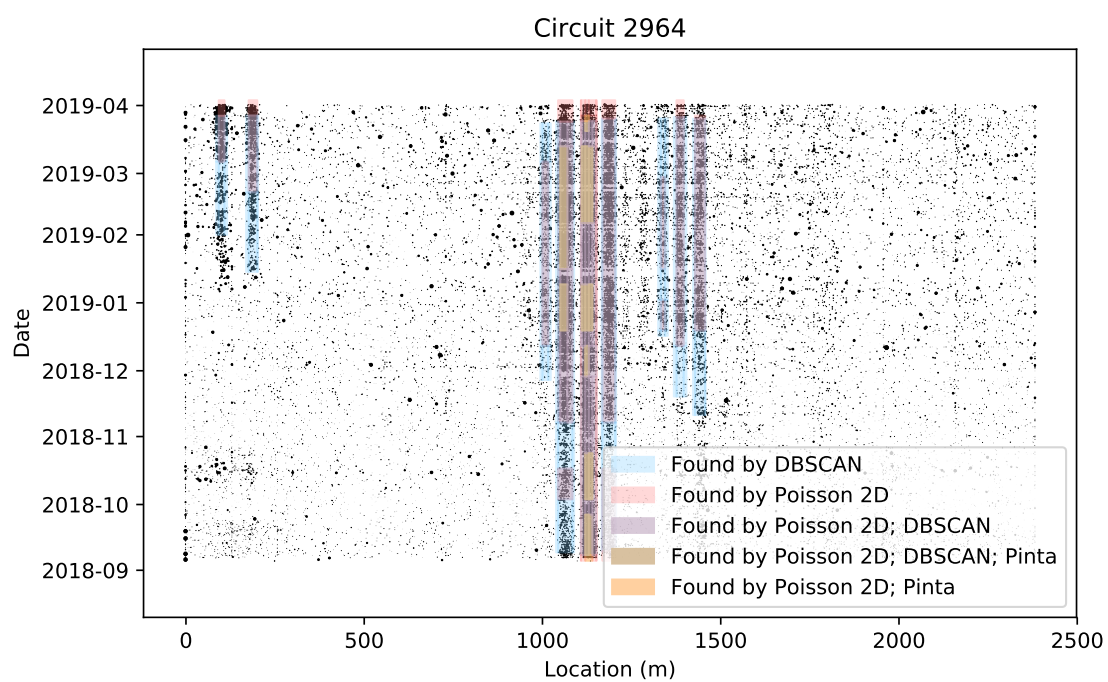
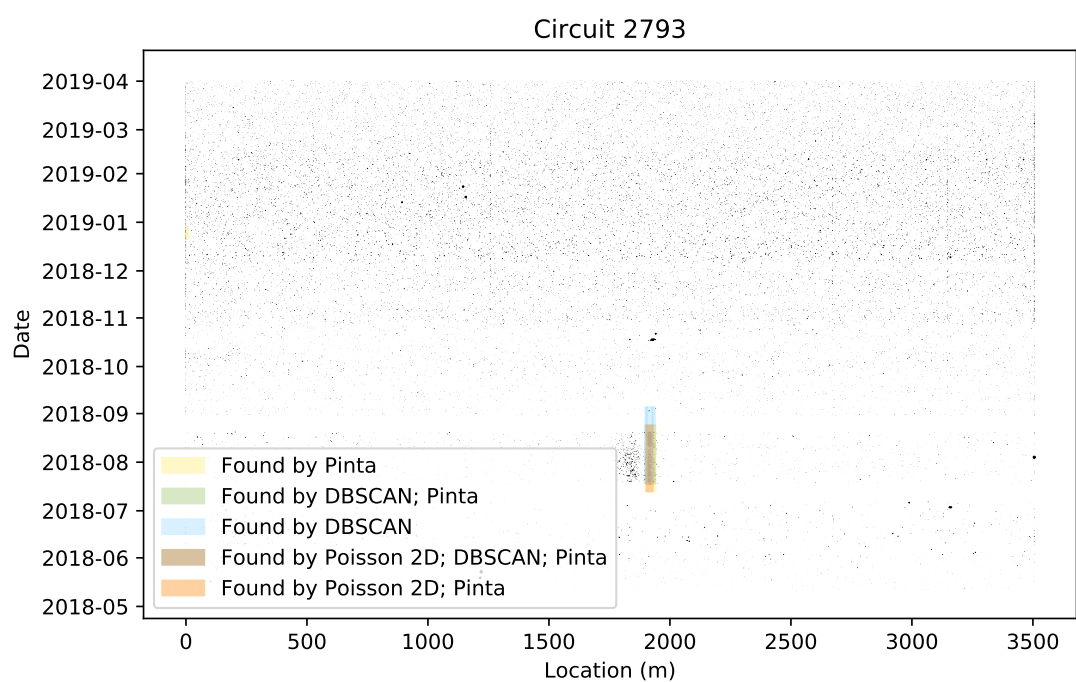
	Poisson	DBSCAN	Pinta
Circuit 2063 (gemiddeld circuit)	7.86 ms	338 ms	27.1 ms
Circuit 3010 (circuit zonder clusters)	5.1 ms	329 ms	22.8 ms
Circuit 2145 (een circuit met veel PD's)	17.6 ms	2240 ms	95.8 ms
Circuit 2793	16.2 ms	1190 ms	53.6 ms
Circuit 2964	12.7 ms	1060 ms	34.9 ms
Circuit 3598	6.04 ms	261 ms	11.8 ms
Circuit 1615	6.44 ms	189 ms	11.7 ms
Circuit 2000	7.01 ms	495 ms	16.5 ms

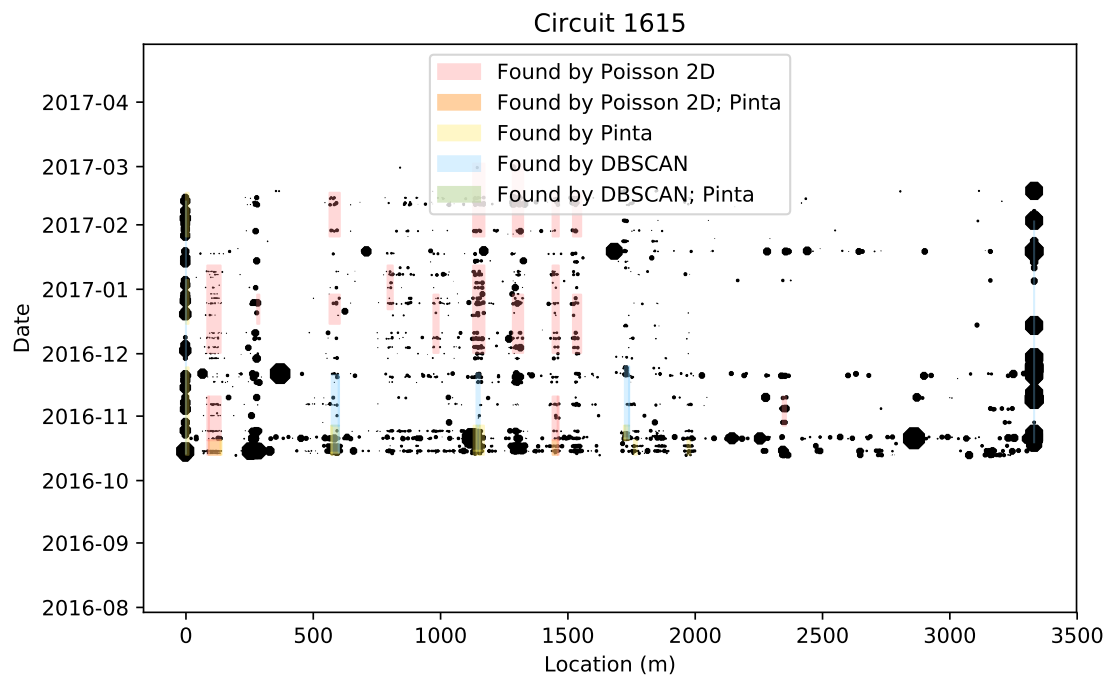
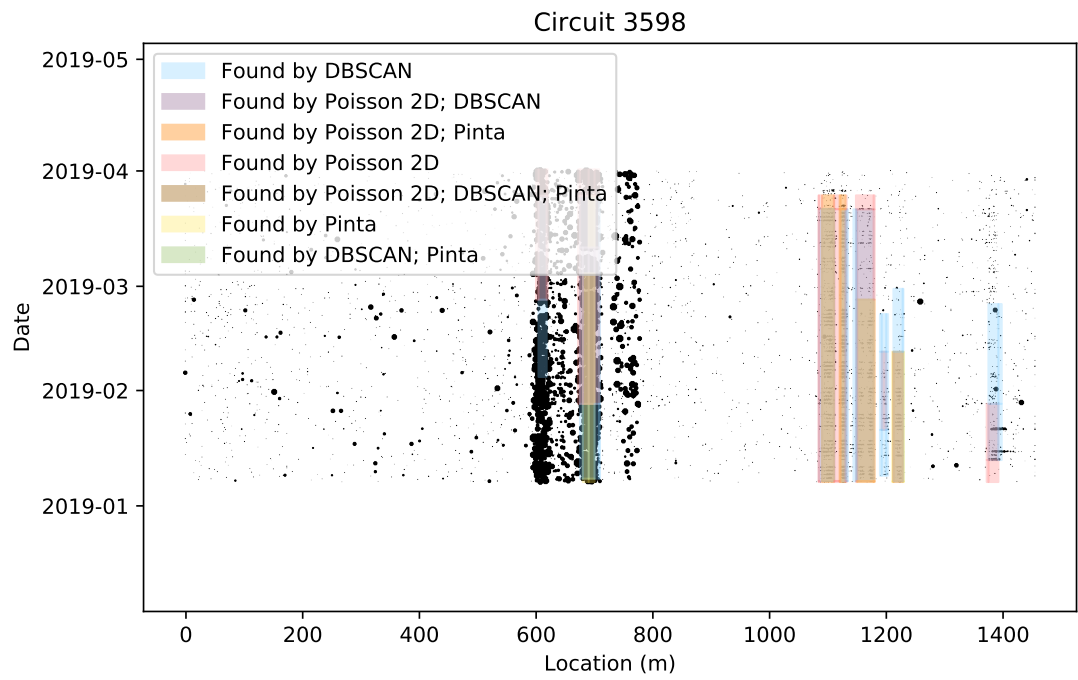
Tabel 1: Enkele tijdsduren voor het uitvoeren van de algoritmes.

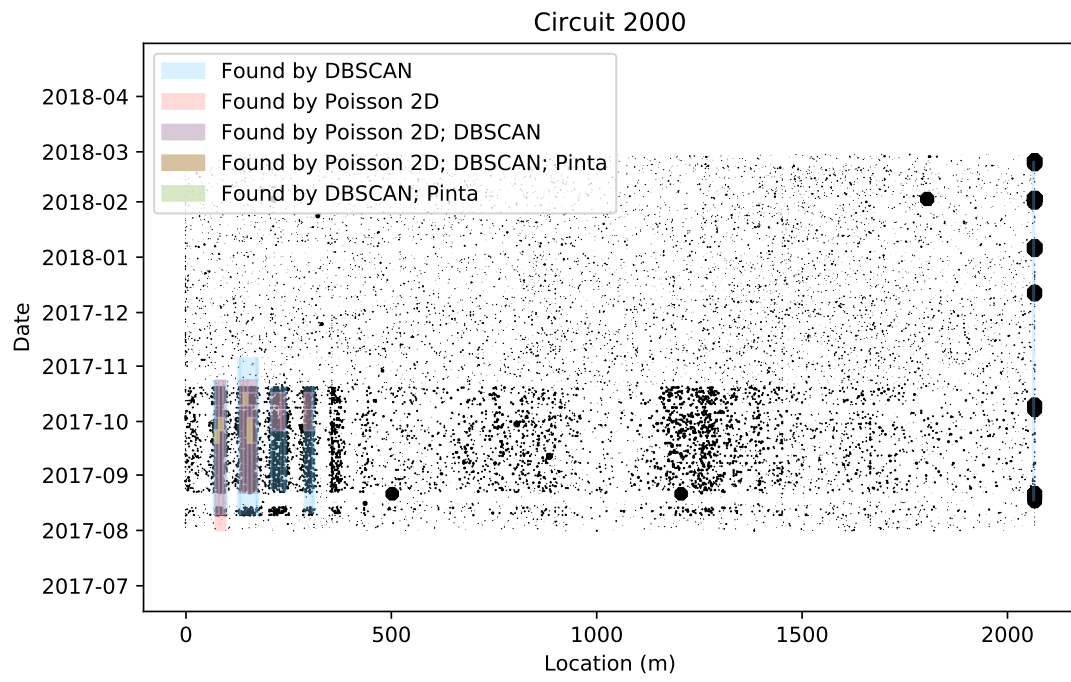


Figuur 30: De clusters gevonden door de verschillende algoritmes.









Deel III

Tot slot

11 Conclusie

11.1 Het resultaat

Wij zijn trots op ons resultaat. Onze eerste opdracht, het maken van een cluster-algoritme, hadden we binnen de eerste weken volbracht. In de tijd die over is hebben we niet alleen nóg twee fundamenteel nieuwe algoritmes geïmplementeerd, maar ook een systeem ontwikkelend om de resultaten van de algoritmes zinvol te combineren. Het systeem is algemeen genoeg voor alle voorziende doeleinden, en biedt ook de mogelijkheid om hetzelfde algoritme meerdere keren toe te passen met verschillende parameters (dit heet een *embedded method* in de numerieke wiskunde).

De algoritmes, samen met het ensemblemodel, vormen de kern van ons resultaat. We vinden het belangrijk dat onze algoritmes ook echt bruikbaar zijn voor Alliander. Daarom hebben we veel tijd geïnvesteerd in het verwerken van de algoritmes tot een goed gedocumenteerde Python-module die we leveren aan Alliander, zodat zij met één regel code aan de slag kunnen.

Ten slotte zijn we veel bezig geweest met het optimaliseren van de algoritmes. Het Poisson- en Pinta-algoritme zijn zo ver geoptimaliseerd dat de *bottleneck* nu het discretiseren is. Gebruikmakend van de constante discretisatieafstand hebben dit zelfs 3 keer tot 10 keer sneller kunnen doen dan de standaardmethode van `numpy`. Ter illustratie van het resultaat: het 2D-clusteren met het Poisson-algoritme (de snelste van de drie) is gemiddeld **90 keer sneller** dan inladen van de `.csv`-bestanden van hetzelfde circuit, en gemiddeld **3 keer sneller** dan het inladen (SSD naar RAM) van een voorverwerkt (*gepickled*) circuit.

De snelheid van de algoritmes is niet alleen prettig, maar het biedt ook nieuwe mogelijkheden. In het bijzonder is het toepassen en combineren van meerdere algoritmes met verschillende parameters nu rendabel, waardoor gevonden clusters een zekerheidsgraad krijgen. Ook denken wij dat autonoom *real-time clusteren* hiermee mogelijk is, zonder nauwkeurigheid te moeten inleveren om de gewenste snelheid te behalen. Het is daarmee een verbetering op het bestaande algoritme van DNV GL.

11.2 De samenwerking

De samenwerking verliep goed. Natuurlijk was er sprake van verschil in programmeerniveau, maar degenen met veel ervaring hebben de rest veel geleerd en goed geholpen. De begeleiding vanuit de Radboud Universiteit en Alliander was op maat, en over het algemeen werden onze vragen helder beantwoord.

12 Ideeën voor de toekomst

Er zijn een aantal onderwerpen die ons interessant of nuttig leken om in de toekomst uit te pluizen. Naar sommige hiervan hebben we al wat gekeken en naar sommige nog helemaal niet.

12.1 Realtime clusterizen

Op dit moment zoeken onze algoritmes naar clusters in data van enkele maanden of zelfs jaren. Het uiteindelijke doel is natuurlijk om, meteen als er iets mis dreigt te gaan in de kabel, een warning te kunnen geven. De volgende stap zou dus zijn om live de verdachte gebieden te kunnen vinden.

In principe werken alle algoritmes al realtime. Als je ze bijvoorbeeld toepast op de 6 maanden meest recente data, zal het ook clusters vinden die aan het eind van de dataset voorkomen, en dit is dan dus *realtime* herkenning van huidige clusters. We hebben dit niet verder onderzocht (je zou historische data natuurlijk kunnen afkappen om realtime data te simuleren). Verder is de runtime van de algoritmes zó laag dat dit ongetwijfeld realtime toepasbaar is. (Zo is Poisson, op een *moving window* van 6 maanden, in theorie 100 keer per seconde toe te passen op de nieuwste data.)

12.2 Banden

Zoals eerder gezegd, blijkt dat als je heel erg inzoomt, de PD's in verticale banden liggen. Binnen deze banden fluctueert de locatie van de PD's alsnog enigszins. Ten eerste zou het interessant zijn om erachter

te komen waarom dit zo is. Ons vermoeden is dat de SCG de locaties eigenlijk discreet meet, maar dat DNV GL ze opzettelijk een beetje jittert. En ten tweede zou het handig zijn als deze banden automatisch kunnen worden bepaald. Dan zou bijvoorbeeld voorkomen kunnen worden dat bij het maken van een histogram sommige bakjes twee banden bezitten en sommige drie. Of je zou de banden verder kunnen jitteren zodat de PD's weer wat organischer verspreid zijn.

12.3 Dubbele clusters

In sommige circuits zie je dat veel clusters een soort *kopie* hebben op een net iets andere locatie. Voorheen werd gedacht dat dit komt doordat sommige moffen in de kabel signalen gedeeltelijk reflecteren, waardoor de SCG dezelfde PD's op verschillende locaties meet. Omdat de mof een constante locatie heeft, wordt het volledige cluster *reflecteerd*: alle PD's uit het cluster worden gespiegeld vanaf hetzelfde punt. Na een beraadslaging met DNV GL blijkt deze hypothese waarschijnlijk niet te kloppen. Het bepalen van de precieze oorzaak vereist betere kennis van de werking van de SCG-firmware en de fysische eigenschappen van PD's; dit is een onderwerp voor later onderzoek. Onze algoritmes, of de SCG-firmware, zou moeten worden aangepast om onderscheid te kunnen maken tussen echte cluster en hun *kopieën*.

12.4 Tijdperken

Bij sommige circuits zie je dat het gedrag door de tijd heen opeens heel erg kan veranderen. Bijvoorbeeld dat er in een keer veel minder ruis is en dat de ruis gemiddeld een significant hogere lading heeft. Dit zou kunnen komen door een verplaatsing van de SCG of door een software of hardware update. Dit heeft nadelige effecten op de algoritmes: Een cluster dat significant meer punten bevat dan de rest van de kabel rond die tijd, maar vergeleken met de rest van de tijd niet zo veel, wordt niet gevonden. Het zou erg handig zijn voor de algoritmes als dit soort tijdperken automatisch gevonden zouden kunnen worden. Dit zou misschien kunnen door te kijken naar de gemiddelde lading op een tijdstip of naar momenten dat er überhaupt langere tijd niet gemeten wordt.

12.5 Indicatie van warnings of bepalen moffen

Een ultiem doel zou zijn dat de algoritmes ook het warningsniveau kunnen vaststellen. Als de dataset groot genoeg is, zou wellicht met machine learning kunnen worden bepaald wanneer welk warningsniveau toepasselijk is. Op dezelfde wijze zou ook het type mof kunnen worden gevonden.

12.6 De grootte van de ladingen

Op dit moment maken de algoritmes (op uitzondering van Poisson met `weigh_charges=True`) geen gebruik van de grootte van de ladingen van de PD's. De ladingen bij een kapotte mof zijn over het algemeen hoger dan die van ruis. Er zouden dus nog algoritmes ontworpen kunnen worden die wel naar de lading kijken. Een voorbeeld zou zijn om een soort drempelwaarde in te voeren, zodat alleen de ladingen hoger dan die drempelwaarde worden meegenomen. Vervolgens zouden gewoon dezelfde algoritmes op deze nieuwe verzameling PD's kunnen worden losgelaten.

12.7 Clusterbreedtes

We hebben wat gekeken naar de 'breedtes' van clusters en hoe de PD's binnen een cluster verdeeld zijn. Een van de vragen is: waar komt deze breedte vandaan? Komt dit doordat de PD's daadwerkelijk zo verspreid rond de mof plaatsvinden, of komt dit door een meetfout van de SCG? In het laatste geval zou de breedte van de clusters per circuit ongeveer gelijk moeten zijn. En is deze breedte dan recht evenredig met de lengte van het circuit? Dit is iets wat verder uitgezocht zou kunnen worden.

Referenties

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay
2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Deel IV

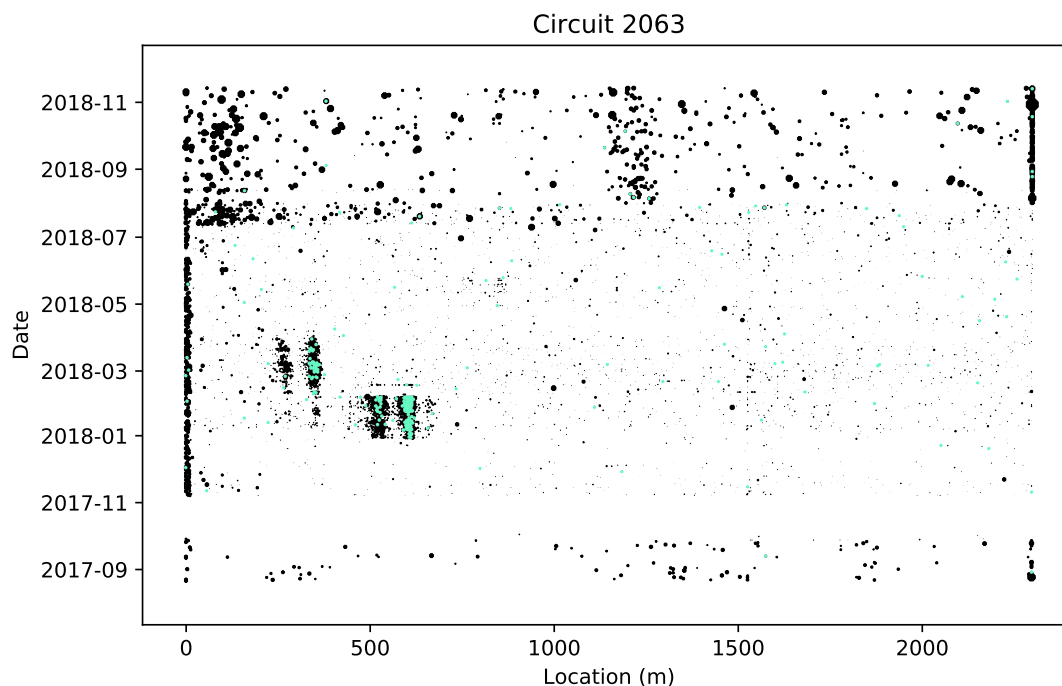
Appendix

A Monte Carlo-algoritme

Als uitsmijter nog een laatste algoritme. *Monte Carlo* is een naam die veel gebruikt wordt voor randomized algoritmes. Monte Carlo-clustering is dus randomized clustering. Het hoofdidee is het volgende stappenplan:

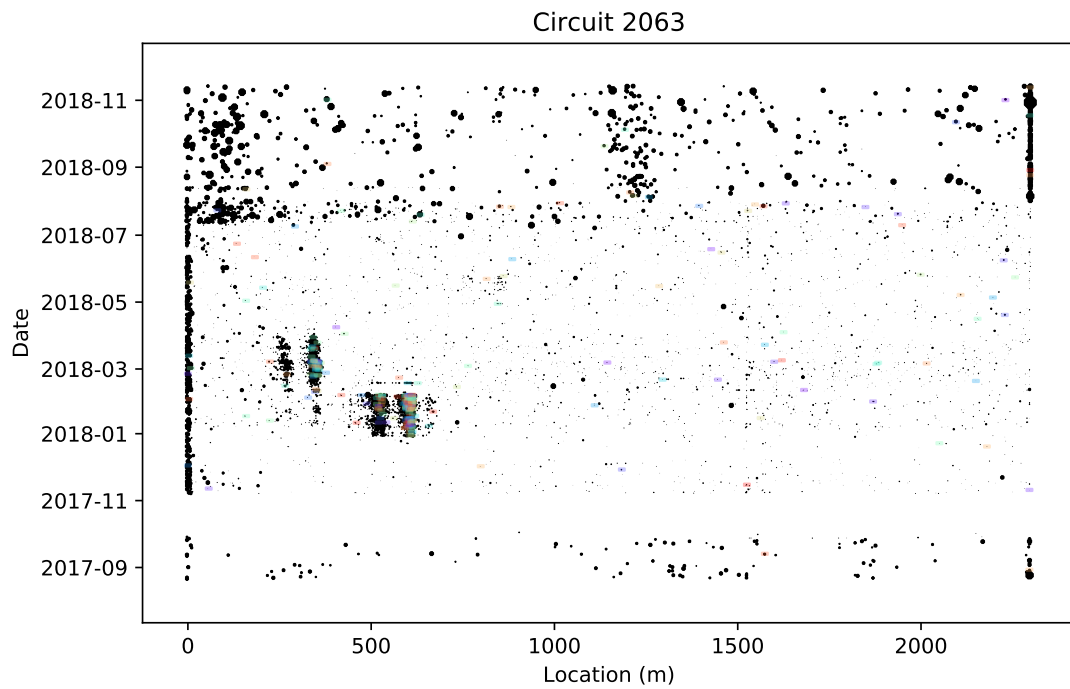
1. Kies een heleboel willekeurige partial discharges
2. Teken een rechthoek om de gekozen partial discharges
3. Combineer de rechthoekjes met de | (or)
4. Bestempel de rechthoekjes met de meeste rechthoekjes waardoor ze gevonden zijn als Cluster

Om deze stappen wat inzichtelijker te maken staan hieronder een aantal plaatjes.



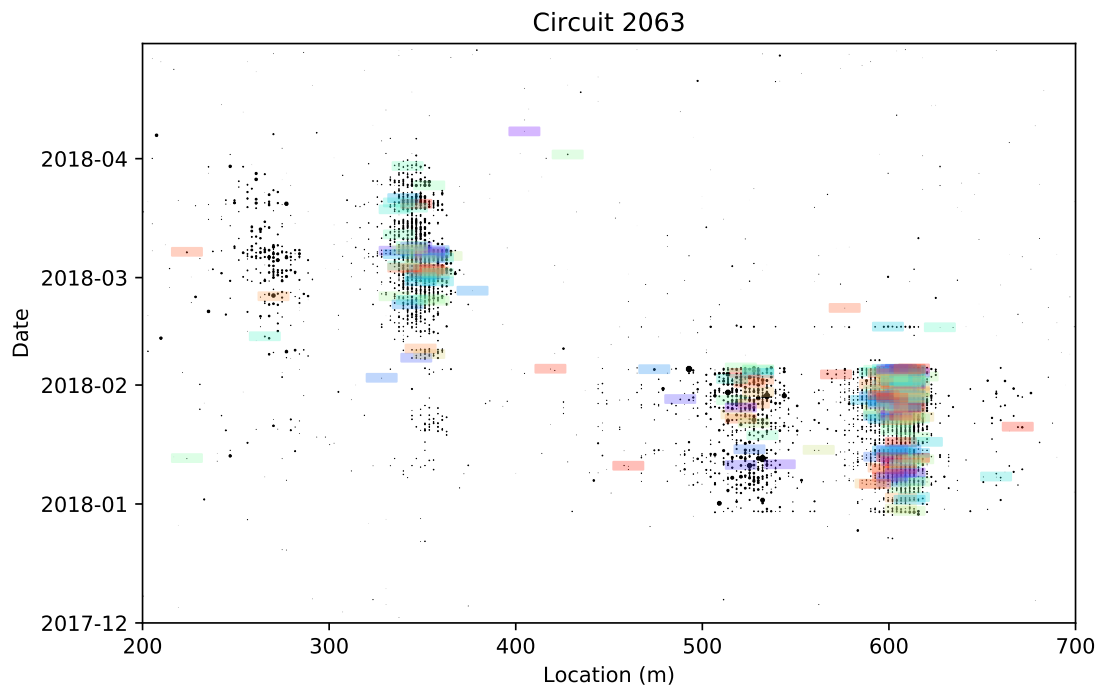
Figuur 31: Stap 1: Kies een heleboel willekeurige punten

Het aantal punten dat we kiezen kan op twee manieren worden bepaald. De eerste is door een parameter `choices_exact`, waarbij de gebruiker een exact aantal punten dat moet worden gekozen kan aangeven. Als deze parameter op `None` staat (de standaardwaarde), dan wordt er op een dynamische manier een geschikt aantal punten gekozen. Het aantal hangt in dit geval af van de lengte van het circuit (in meters) en de tijdsduur van de metingen (in maanden). De gebruiker kan dit nog iets sturen door de parameter `choices_div` te veranderen: Het aantal gekozen punten wordt uiteindelijk gedeeld door `choices_div`.



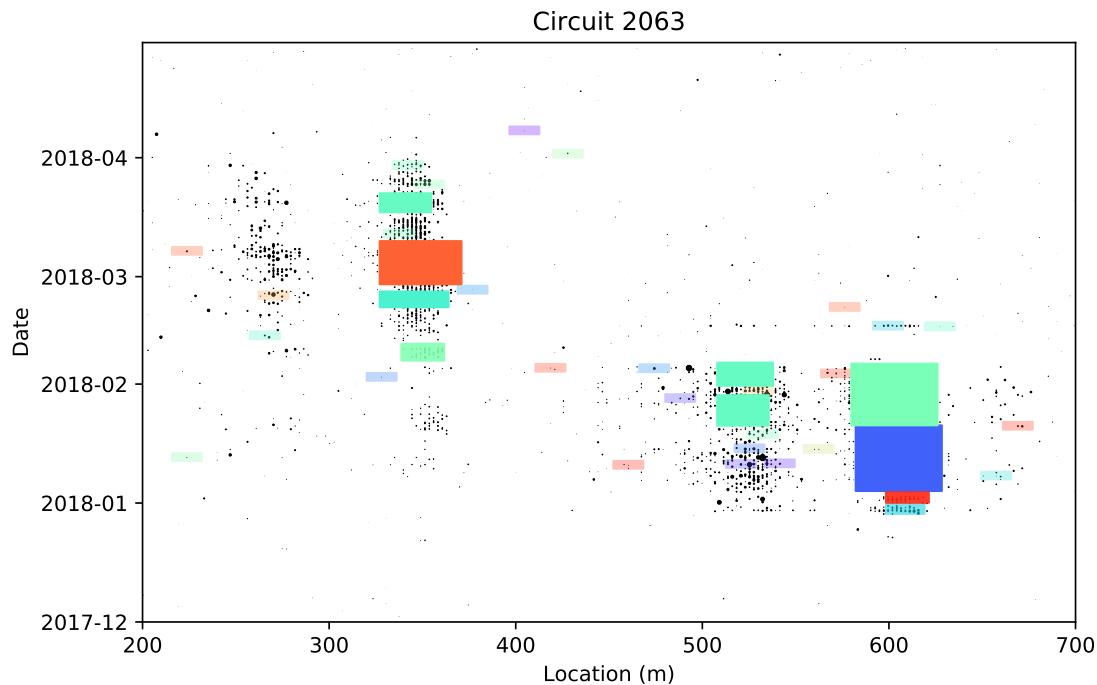
Figuur 32: Stap 2: Teken rechthoekjes om de gekozen punten

De grootte van de rechthoekjes is standaard ingesteld op 32 meter breed en 6 dagen lang. De gebruiker kan dit zelf aanpassen door de parameters `loc_rect_size` en `time_rect_size` aan te passen. Met name `time_rect_size` groter maken kan interessant zijn, aangezien clusters vaak langer langwerpig zijn. Grotere rechthoeken betekent natuurlijk ook dat ze sneller overlappen. Hier moet eventueel rekening mee worden gehouden door andere parameters aan te passen; dit wordt niet dynamisch gedaan.



Figuur 33: Stap 2: Ingezoomd op een verdacht gebied

Op het ingezoomde plaatje is goed te zien dat er erg veel rechthoeken liggen in het meest rechtse cluster, terwijl andere plekken minder rechthoeken hebben. Dit komt doordat het rechtse cluster veel partial discharges bevat, waardoor de kans dat een van de PD's willekeurig gekozen wordt relatief groot is. Op deze manier is goed terug te zien dat Monte Carlo-clustering zich goed houdt aan de definitie dat een cluster een rechthoek is met uitzonderlijk veel partial discharges.

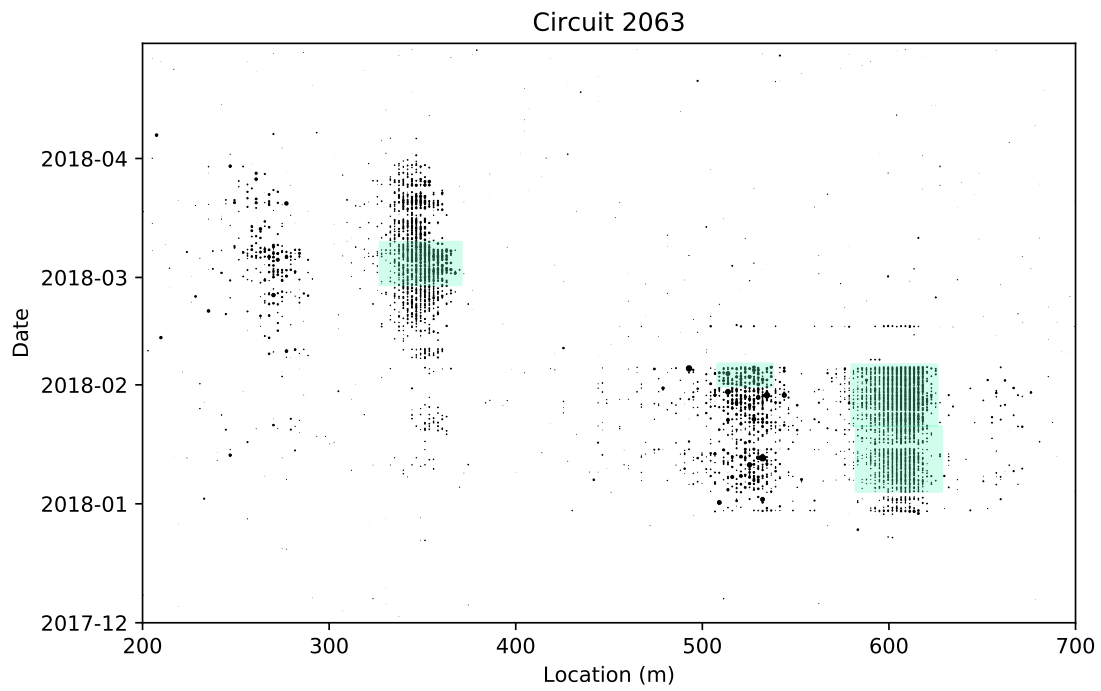


Figuur 34: Stap 3: Combineer de rechthoekjes

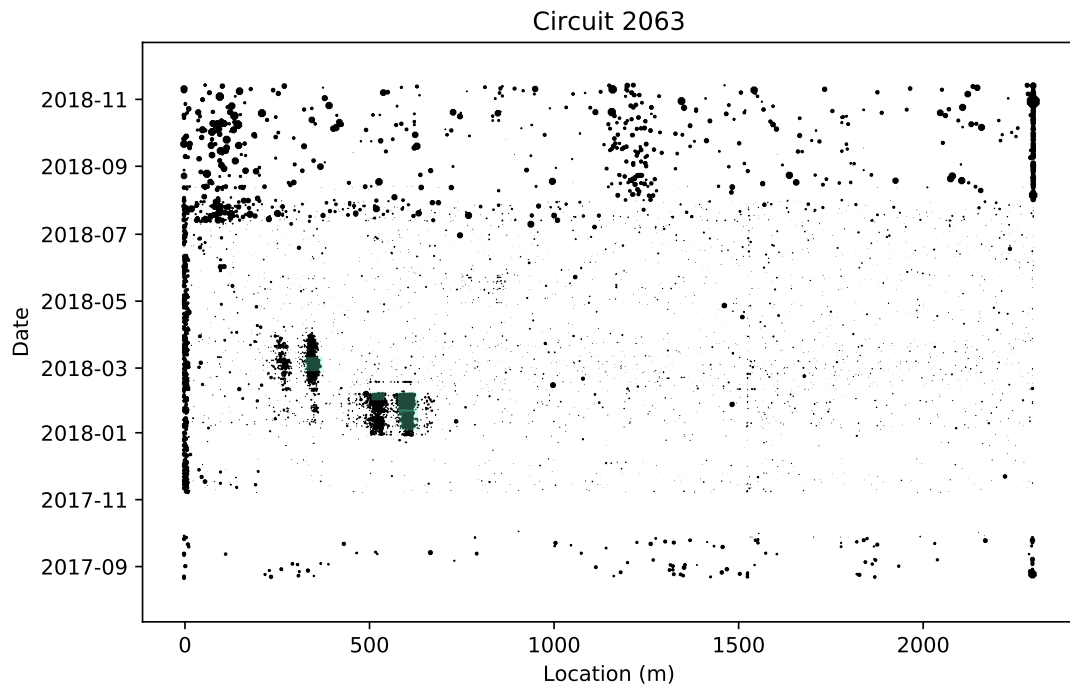
Stap 3 zou gewoonlijk een moeilijke stap zijn om te programmeren, maar gelukkig kunnen we hiervoor gebruik maken van de functie `|` die al gedefinieerd is voor `Rectangle` en `Cluster`. Hierdoor kunnen we deze stap in 2 regels opschrijven:

```
clusters = {Cluster({r}) for r in rectangles}
reduced = functools.reduce(operator.__or__, clusters)
```

Tijdens het tekenen van de rechthoeken hebben we ze allemaal een uniek nummer gegeven en deze opgeslagen in de `found_by` property van `Rectangles`. Tijdens het combineren zijn dus al deze nummers bijgehouden. Bij een grotere `Rectangle` kunnen we nu makkelijk kijken hoeveel `Rectangles` uit stap 2 overlap hebben met deze grote `Rectangle`. Dat is namelijk: `len(rectangle.found_by)`. We kiezen nu een minimum waarboven dit aantal moet zitten voordat we een `Rectangle` toelaten als `Cluster`. Net zoals bij het aantal willekeurige punten kan dit weer op twee manieren: Een exact aantal gekozen met de parameter `found_exact`, of dynamisch op basis van de circuitlengte, tijdsduur, en de parameters `choices_div` en `found_div`.



Figuur 35: Stap 4: Maak clusters



Figuur 36: Eindresultaat

Het eindresultaat ziet er redelijk uit. De belangrijkste clusters zijn gevonden, al worden ze niet helemaal correct weergegeven: grote delen van de clusters vallen buiten de gekozen Rectangles. Het resultaat is waarschijnlijk beter geweest als we `time_rect_size` groter maken. Ook is het algoritme randomized, dus door het algoritme opnieuw uit te voeren, vinden we een andere uitkomst, die wellicht beter is.

De runtime van het Monte Carlo algoritme hangt heel erg af van hoeveel willekeurige punten er worden gekozen. Nu is het punten kiezen zelf niet het probleem: het probleem is dat we deze de rechthoekjes om deze punten heen allemaal moeten combineren met de $\mid$. Deze methode is erg langzaam als we veel rechthoekjes combineren. Hier een tabel van runtimes:

Aantal punten	Runtime
100	70 ms
300	520 ms
500	1.23 s
1000	5.12 s

Ter illustratie: bij het circuit hierboven hebben we ongeveer 300 punten gebruikt om het eindresultaat te bereiken. Voor de runtime geldt dus: meer punten is slechter. Het algoritme is dus langzamer dan Poisson en Pinta. Dat $\mid$ zo langzaam is bij veel rechthoeken maakt bij Ensemble clustering niet uit, omdat we daar hooguit 20 Rectangles hebben. Die kunnen dus snel gecombineerd worden. Dat we bij Monte Carlo-clustering 300+ clusters gebruiken, is dus een probleem als je naar de runtime kijkt.

De nauwkeurigheid van het resultaat hangt erg af van hoeveel punten er worden gekozen. In dit geval is het dus: meer punten is beter. Er moet dus een afweging worden gemaakt tussen runtime en nauwkeurigheid. Toch is Monte Carlo-clustering minder nauwkeurig dan de andere 3 algoritmes, tenzij een groot aantal punten wordt gebruikt. Ruis kan Monte Carlo nog wel goed herkennen; de willekeurige rechthoekjes liggen dan zo verspreid dat ze waarschijnlijk nooit overlappen. Ook een circuit met een klein aantal clusters (1 of 2) lukt goed. Er komen wel problemen om de hoek kijken als het aantal clusters groot wordt. Dan raken de willekeurige rechthoeken verspreid over de verschillende clusters, waardoor geen enkel cluster een groot aantal willekeurige rechthoeken heeft. Daarom overlappen de rechthoeken niet, en worden er geen rechthoeken als verdacht bestempeld. Monte Carlo-clustering kan dus niet het verschil zien tussen een circuit met 0 clusters en een circuit met bijvoorbeeld 10 clusters, iets wat de andere algoritmes wel kunnen. Ook worden clusters soms in meerdere verdachte rechthoeken opgesplitst die per toeval net niet overlappen. Dit is bijvoorbeeld ook te zien in 35. Het rechtse cluster wordt aangegeven met 2 Rectangles die net niet overlappen.

Dit alles bij elkaar maakt dat Monte Carlo-clustering een aantal grote zwaktepunten heeft die de andere algoritmes niet hebben. Hierom hebben we ook besloten om het als ‘bonus-algoritme’ te beschouwen. Het concept erachter vinden we leuk en dat een randomized algoritme soms goede resultaten geeft is interessant, maar in de praktijk denken we dat de nadelen van Monte Carlo-clustering niet opwegen tegen de voordelen, zeker als we het vergelijken met algoritmes als Poisson, DBSCAN en Pinta.