Verslag Research & Development 2 (R&D2):

Instinct-Based Mating in Genetic Algorithms
Applied to the Tuning of 1-NN Classifiers

Naar het gelijknamige artikel van:
Thiago Quirino, Miroslav Kubat en Nicholas J. Bryan

Robin Munsterman, Wesley Janssen en Joost Rijneveld
s4070968, s4037332, s4048911

Tweedejaars Informatica
Radboud Universiteit Nijmegen

25 maart 2012

Samenvatting

In dit verslag bespreken we de methodiek en de testresultaten van het gebruik van een genetisch algoritme om een optimale k-NN classifier te vinden voor een gegeven dataset. Het beschreven en geïmplementeerde algoritme wordt uitgelegd in het artikel *Instinct-Based Mating in Genetic Algorithms Applied to the Tuning of 1-NN Classifiers* van *Thiago Quirino, Miroslav Kubat* en *Nicholas J. Bryan*. We laten zien dat het algoritme het beste presteert bij een middelgrote dataset. Gezien de complexiteit van de derde orde zijn grote datasets een probleem, terwijl kleine datasets te snel gereduceerd worden. Verder bespreken we de mogelijke verbeteringen op het standaardalgoritme dat in het artikel wordt aangedragen en het resultaat dat deze verbeteringen teweeg kunnen brengen.

In het laatste hoofdstuk van dit verslag bespreken we de verbeteringen die we hebben doorgevoerd in de tweede fase van dit project en de hierdoor behaalde resultaten.

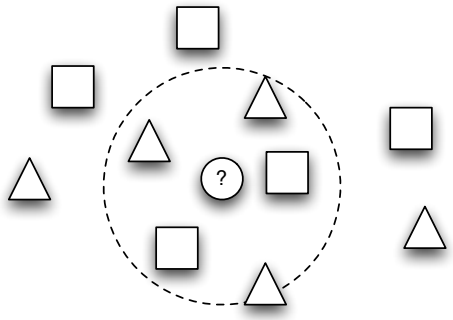
Inhoudsopgave

Lijst van figuren

1 Inleiding

We hebben getracht een algoritme te implementeren wat een verbetering is op het KNN-algoritme. KNN (K-nearest neighbour) is een classificatie-algoritme. Classificatie is het hangen van een label (een klasse) aan een bepaalde input. Als je bijvoorbeeld twee dozen hebt, één met het label 'kleine spullen' en eentje met 'grote spullen', bedoelen classificeren het stoppen van spullen 'de input' in één van de twee dozen.

KNN probeert dit door te kijken naar wat de 'buren' van de test-sample zeggen. Iedere input krijgt de label wat het meest voorkomt onder de k punten die het dichtst bij deze input zitten (zie figuur 1). De waarde van K is dus belangrijk voor de nauwkeurigheid van het algoritme. Als k = 1, zal de input i de label krijgen van zijn dichtstbijzijnde buur. Deze methode is erg simpel om te implementeren en te begrijpen, en wordt daarom dus redelijk veel ingezet voor simpele classificatieproblemen.



Figuur 1: Classificatie op basis van K-NN

K-NN is jammer genoeg niet een erg efficiënt algoritme, en er zitten nog wat andere probleempjes aan die het wat trager maken. Twee manieren om het classificeren te verbeteren zijn 'condensen' of te 'editen'.

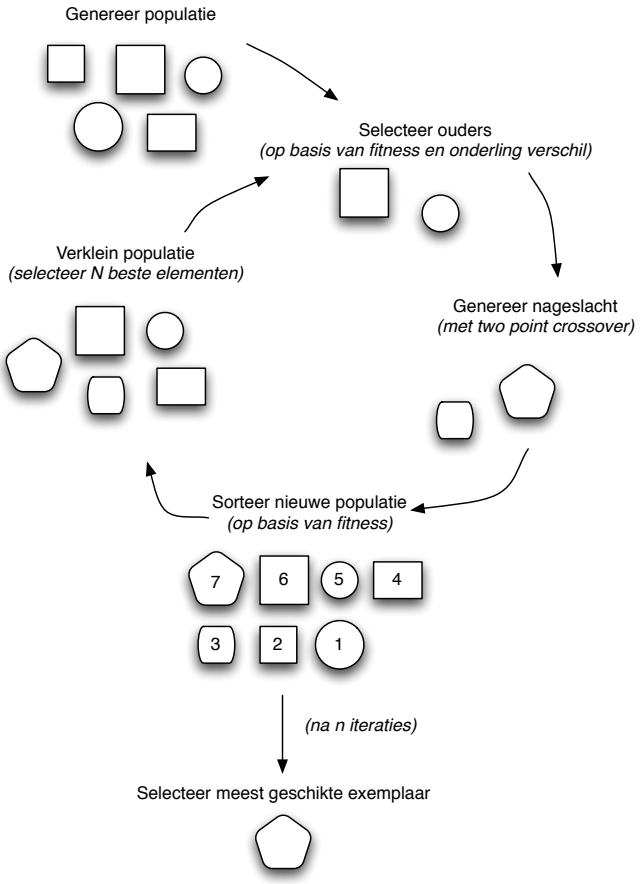
2 Probleemstelling

2.1 Condensing en Editing

Condensing is het verkleinen van de trainingset, zodat er maar naar minder samples hoeft worden gekeken. Door een deel van de set buiten beschouwing te laten, hoeft het K-NN algoritme minder vaak uitgevoerd te worden. Dit wordt uigedrukt in 'reductie'. Editing probeert ruis van de trainingset weg te halen. Ruis zijn samples die verkeerd zijn geclassificeerd. Deze wil je natuurlijk niet gebruiken om de testset te valideren. Editing kan ook vele manieren gebeuren, maar zorgt er dus voor dat de nauwkeurigheid stijgt.

2.2 Onze methode

Ons artikel (Instinct-Based Mating in Genetic Algorithms Applied to the Tuning of 1-NN Classifiers) is gebaseerd op het artikel van A. Rozsypa en M. Kubat. Zij probeerden met behulp van een genetisch algoritme (zie figuur 2 voor de algemene structuur van een genetisch algoritme) 1-NN te implementeren. We zullen dit vanaf nu RK-GA noemen.



Figuur 2: De algemene structuur van een genetisch algoritme

Dit algoritme zoekt een geschikte oplossing door, zoals altijd bij een genetisch algoritme, een grote populatie bij te houden, waarbij ieder exemplaar uit deze populatie twee rijen met getallen (chromosomen) heeft: één chromosoom om te bepalen welke voorbeelden uit de trainingsset worden gebruikt, de ander kijkt welke attributen er worden gebruikt om deze voorbeelden te beschrijven. Ons artikel gebruikt dit als basis. Hij krijgt eerst een aanpassing (van RK-GA naar modified RK-GA) aan de manier waarop de fitness wordt uitgerekend. Dit zorgt er onder andere voor dat de parameters die in RK-GA worden gebruikt, uitgerekend kunnen worden. Hierdoor hoeven we niet meer op deze parameters te letten, omdat ze toch al optimaal worden gezet. Zie voor de initiële en veranderde fitness functie figuur 3.

De fitnessfunctie in het normale RK-GA-algoritme ziet er (na wat algebraïsche manipulatie) als volgt uit, met N_{ET} en N_{AT} het totale aantal examples en attributen, N_E en N_A het aantal bewaarde examples en attributen en E_R het aantal verkeerd geclassificeerde samples.

$$f_{RK} = c_1 \cdot \left(\frac{N_{ET} - E_R}{N_{ET}}\right) + c_2 \cdot \left(\frac{N_{ET} - N_E}{N_{ET} - 1}\right) + c_3 \cdot \left(\frac{N_{AT} - N_A}{N_{AT} - 1}\right)$$

Vervolgens kiest de auteur bij Modified RK-GA de volgende waarden voor c_1, c_2 en c_3, waardoor er niks meer ingesteld hoeft te worden.

$$c_1 = c_2 = \frac{N_{ET} + 1}{2 \cdot N_{ET} + N_{AT} + 3}$$
$$c_3 = \frac{N_{AT} + 1}{2 \cdot N_{ET} + N_{AT} + 3}$$

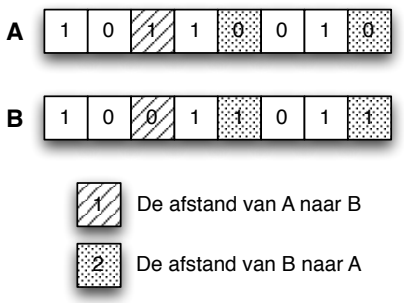
Figuur 3: Fitnessfuncties

Modified RK-GA wordt gebruikt als uitgangspunt voor drie voorgestelde aanpassingen. De drie aanpassingen zijn bedacht rondom het uitzoeken van een partner. Dit gebeurde hiervoor alleen maar op basis van fitness, maar in de biologie is dit ook niet altijd het geval; er komt vaak veel meer bij kijken. Het artikel beschrijft drie manieren om deze instinct-gebaseerde partnerkeuzes in een GA te implementeren: Correct-My-Wrongs, Hamming-Distance en Multi-Population.

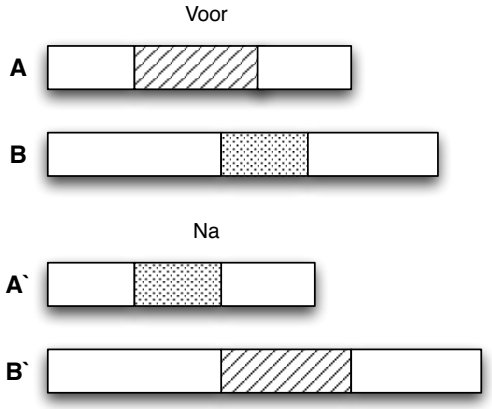
In de eerste vergadering werd duidelijk dat slechts één van deze drie nodig was om te implementeren. Voor ons viel de derde meteen af: hij is veel moeilijker om te maken en had een lagere nauwkeurigheid en reductie. Hoewel de eerste en de tweede erg op elkaar lijken zijn we uiteindelijk voor de eerste gegaan. Deze sprak ons meer aan en was iets nauwkeuriger dan de andere twee.

In Correct-My-Wrongs worden allereerst alle specimens gesorteerd op basis van hun fitness. De eerste helft worden als 'eerste ouders' gelabeld. Daarna wordt voor iedere specimen een 'error-vector' gemaakt en opgeslagen. In deze vector, die de grootte heeft van de trainingset, staat op plaats i een "0" als de specimen het i-de element een goed label heeft gegeven, en een "1" als dit niet het geval was. Daarna wordt een zogenoemde 'Supplementary Distance Matrix' afgeleid van de error-vectoren. In deze matrix staat op de i-de rij en de j-de kolom de CMW-waarde van exemplaar i naar exemplaar j. Dit is het aantal keren waar de error-vector van i waarde "0" heeft en j waarde "1". Zie figuur 4 ter verduidelijking:

Deze matrix is dus asymmetrisch, en wordt gebruikt bij het kiezen van de partners van de 'eerste ouders'. De probabilistische functie kiest namelijk een partner waarbij de CMW-waarde hoog is. Hierdoor worden kinderen dus, zoals de naam suggereert, beter doordat i en j elkaars 'fouten herstellen'. De ouders worden als laatste met behulp van two-point crossover 'voortgeplant' (zie figuur 5). Dit wordt voor alle examples uit de populatie gedaan. Daarna worden willekeurig gekozen samples op willekeurige manier gemuteerd. Zo ontstaat de nieuwe populatie. Deze populatie wordt vervolgens teruggebracht tot de originele grootte door de 'slechtste' samples weg te gooien.



Figuur 4: De CMW-afstand op basis van error-vectoren



Figuur 5: Met two-point crossover ontstaan nieuwe afstammelingen

Als het algoritme klaar is met itereren zal het specimen met de hoogste fitness gekozen worden als classifier. De examples in deze classifier vormen de prototypes. Met deze classifier zal vervolgens het K-NN algoritme worden uitgevoerd om de samples van de testset te kunnen classificeren.

2.3 Keuzes

We hebben dit artikel gekozen omdat op het eerste zicht genetische algoritmes ons erg aansprak. Tijdens een ander college van vorig jaar werden ze al ter sprake gehaald, ze zijn erg intuïtief en vanuit de biologie lijken ze erg op de natuurlijke evolutie, die ons als mensen überhaupt in staat stelt erover na te denken. We hebben voor Correct-My-Wrongs gekozen omdat deze ons het beste leek. Multi-Population leek te moeilijk om te implementeren, zonder daar accuracy voor terug te geven. Hij was minder goed in het classificeren, volgens het artikel. Hamming-Distance leek erg op CMW, maar sprak ons wat minder aan en was gemiddeld iets minder accuraat.

We hebben besloten de implementatie in Java te maken, omdat we allen al genoeg (recente) ervaring met deze taal hebben. Ook is Java snel, en leek een objectgeoriënteerde aanpak voor dit probleem wel een goede keuze.

3 Experimenten

In de onderstaande tabel staan de resultaten van het uitvoeren van ons algoritme op de testsets van zowel de oefencompetities als de diabetes-competitie, zoals ook te vinden op de websites van de competities. Het variëren van de parameters zorgde nauwelijks voor een afwijking van deze resultaten; ze bieden een goed beeld van de gemiddelde executieresultaten van het algoritme. Wel kwam er tijdens het testen natuurlijk af en toe een classifier voorbij die uitzonderlijk goed of slecht presteerde, wat een direct resultaat is van de willekeur die aanwezig is in de mutatiemethode.

Testset	**# Attr.**	**# Klassen**	**Accuracy**	**Reduction**	**Attr. reduc.**
XOR	2	2	72.0%	95.0%	0%
MINST	196	10	78.2%	87.7%	84.7%
Diabetes	8	2	76.6%	81.1%	75.0%

Figuur 6: Tabel met testresultaten

Ons algoritme werkt beter op trainsets met veel attributen: er is in onze metingen een lichte stijging in accuracy te zien. Dit komt omdat attributen met minder invloed op de classificatie worden verworpen en zo alleen naar de cruciale attributen gekeken wordt, waardoor de classificatie nauwkeuriger wordt. Bij classificatie van bijvoorbeeld de XOR-set merken we dat een groot deel van de samples uit de populaties een van de attributen weglaten, waardoor deze classifiers bij voorbaat al slecht scoren. Hierdoor worden de prestaties van het algoritme nogal vertroebeld, waardoor ook het uiteindelijke resultaat tegen valt.

Aangezien het algoritme een vervelende complexiteit (kwadratisch over het aantal samples en lineair over de attributen), konden we bij de grotere datasets (met name MNIST) maar een beperkt aantal iteraties uitvoeren en is het onduidelijk of het algoritme uiteindelijk een betere classifier zou hebben gevonden. Een grotere populatie en een groter aantal iteraties is over het algemeen altijd beter, maar bij MNIST bleven we steken op een populatie van 20 samples en een 40tal iteraties: het duurde een ruime nacht voordat het algoritme deze populatie doorgerekend had. Bij Diabetes en XOR konden we dit wel opvoeren tot enkele honderden respectievelijk duizenden iteraties met een populatie van rond de 200.

We zijn over het algemeen tevreden over de resultaten. Het genetische algoritme staat erom bekend nooit de meest optimale resultaten te vinden in de meest optimale tijd, maar door het vaste stramien kan het algoritme relatief eenvoudig en foutloos geïmplementeerd worden en heel divers worden ingezet. Ten opzichte van de andere teams presteren we (op moment van schrijven) in de middenmoot, wat voor een genetisch algoritme zeker niet slecht is.

4 Conclusie deel 1

Wij zien genetische algoritmen, met daarin in het bijzonder het Modified RK-GA algoritme met Correct-My-Wrongs strategie, als een goede methode om datasets te classificeren. Het algoritme is gemakkelijk te begrijpen en er kunnen redelijk goede resultaten mee behaald worden. Het grootste probleem is echter dat het niet erg efficiënt is. Met een complexiteit van ongeveer $\Theta(n^3)$ (aangezien we K-NN voor elk sample uitvoeren en deze samples vervolgens moeten sorteren op basis van fitness) hebben we erg veel tijd nodig om honderden keren te itereren. Ook zijn genetische algoritmes altijd heel willekeurig in hun keuzes, hierdoor is de output (zelfs na honderden iteraties) iedere keer altijd wel verschillend. Bij de XOR-set wisselde het vaak zelfs van 60% tot 85%.

Als vervolgonderzoek zou nog geëxperimenteerd kunnen worden met populatie- en iteratiegrootte. Om deze groter te krijgen, zou er gekeken kunnen worden of de complexiteit van het algoritme verkleind kan worden. Ook zien we dat het algoritme niet al te goed functioneert op een dataset met weinig attributen. Hierdoor hadden we wat moeite met de XOR-set. Het probleem is dat het algoritme niet kijkt hoeveel attributen er over zijn, en gooit te gemakkelijk een attribuut weg, zelfs als dit het laatste attribuut is. We proberen in het tweede deel van de cursus een oplossing hiervoor te verzinnen, tevens kijken we nog of sommige delen efficiënter kunnen worden uitgevoerd.

5 Verbeteringen, deel 2

Een voordeel van een genetisch algoritme is dat het een redelijke oplossing biedt voor veel uiteenlopende problemen omdat het zich door middel van een natuurlijke vorm van evolutie langzaam maar zeker steeds dichter naar een oplossing toe werkt. Hier zit ook het grote nadeel; *langzaam*. Tijdens het classificeren van de testsets in deel 1 liepen we hier al tegenaan, vooral bij sets als MNIST. Dit was dus zeker iets dat we aan moesten gaan pakken tijdens het tweede deel.

Het mooie van een genetisch algoritme is dat gewoonweg meer iteraties uitvoeren (voor grote sets) meestal leidt tot een betere oplossing. Door de rekentijd te verbeteren slaan we dus meteen twee vliegen in één klap, aangezien we hierdoor ook meer iteraties kunnen uitvoeren en dus de resultaten verbeteren. Voor de snelheidsverbeteringen hebben we twee plannen uitgewerkt. Allereerst bleek een analyse van de variantie van attributen een goed idee, aangezien we hiermee het aantal attributen al kunnen terugbrengen voordat we ons genetisch algoritme op de dataset loslaten. Verder ligt het natuurlijk voor de hand om op een gedetailleerd niveau de code te doorlopen en te optimaliseren.

5.1 Algoritmische optimalisaties

Allereerst viel het ons op dat we gedurende het uitvoeren van het algoritme regelmatig elementen (zij het samples uit de trainingsset of attributen binnen een specimen) toevoegen aan een lijst, om deze vervolgens te sorteren. Het toevoegen gebeurt in $\Theta(n)$, en sorteren gaat in $\Theta(nlogn)$. Door hier gebruik te maken van een `PriorityQueue` kunnen we elementen toevoegen én sorteren in $\Theta(nlogn)$ – an sich geen complexiteitsverbetering (want $\Theta(nlogn) = \Theta(n + nlogn)$), maar voor relatief kleine n kan dit natuurlijk behoorlijk veel verschil maken. Vooral omdat dit gesorteerd bij elke classificatie uitgevoerd werd (en dus duizenden keren per iteratie) schelen dit soort aanpassingen behoorlijk.

Verder hebben we verschillende kleine optimalisaties uitgevoerd. Zo berekenen we niet langer de Euclidische afstand, maar alleen het kwadraat hiervan - dit bespaart bij elke vergelijking een wortelberekening, en aangezien deze in $\Theta(n)$ wordt uitgevoerd (want computers moeten wortels benaderen) maakt dit flink uit, mede omdat we de afstand natuurlijk vele duizenden malen berekenen. Het onderling vergelijken van kwadraten levert natuurlijk hetzelfde resultaat op als het vergelijken van de echte afstand.

Tijdens het schrijven van het algoritme zijn we begonnen met het implementeren van K-NN. Achteraf blijkt dat we hier meer gedaan hebben dan nodig: het artikel betreft namelijk uitsluitend 1-NN. Conceptueel gezien betekent dit dat ons algoritme minder flexibel hoeft te zijn – er is immers geen instelling voor de waarde van K. Vanuit praktisch oogpunt betekent dit dat we de classificatie enorm kunnen specialiseren. Voorheen zetten we alle punten in een `PriorityQueue` en ordende deze door middel van een vergelijking van de afstand tot het te classificeren punt. Hierna namen we de K eerste elementen, en bepaalden welke klasse het vaakst voorkwam. Op zich een logische aanpak, ware het niet dat de afstandsberekening nu wel erg vaak wordt uitgevoerd elke

keer dat een punt aan de queue wordt toegevoegd, moeten (met behulp van een binaire zoekboom) $logn$ vergelijkingen worden gemaakt om de juiste plaats in de queue te vinden. Elk van deze vergelijkingen heeft twee afstandsberekingen nodig; van beide te vergelijken punten. Al met al een dure grap. Aangezien we eigenlijk alleen geïntresseerd zijn in het punt met de kleinste afstand, volstaat het ook om de afstand van alle punten één keer te berekenen en gewoon bij te houden welke tot dusver de kleinste afstand heeft. De code in figuur 7 illustreert dit verschil (boven oude code, onder nieuwe code).

```
public int computeClassification(Sample sample) {
      ArrayList<Sample> referenceSet = new ArrayList<Sample>();
      for (int example:examples) {
            Sample t = samples.get(example);
            referenceSet.add(t);
      }
      Collections.sort(referenceSet, new DistanceComparator(sample));
      referenceSet.subList(K, referenceSet.size()).clear();
       //Code om klasse te bepalen uit K dichstbijzijnde buren
      return votedClassification;
}
```

O(n)

Berekent afstand 2n log n keer

O(n log n)
O(n)
O(K)

```
public int computeClassification(Sample sample) {
      Sample minsample = null;
      double minDistSq = Double.MAX_VALUE;
      for (int example:examples) {
            Sample t = samples.get(example);
            double d = sample.distanceSquared(t, attributes);
            if (d < minDistSq) {
                  minDistSq = d;
                  minsample = t;
            }
      }
      return minsample.getClassification();
}
```

1x per example
O(n)

Berekent afstand 1x per example

Figuur 7: Optimalisatie van code, na K = 1

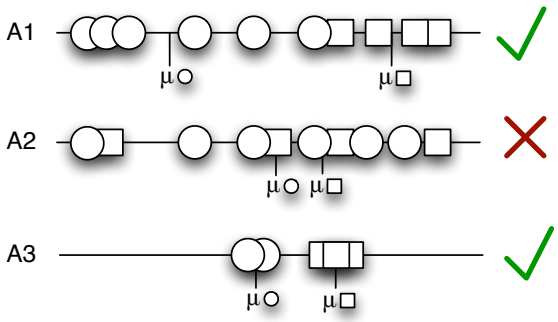
Zoals in de figuur al is aangegeven, is dit wel degelijk een verschil dat echt meespeelt in de complexiteit. In eerste instantie werd de afstand maar liefst $2 \cdot n \cdot logn$ keer berekent, terwijl dit na optimalisatie nog maar n keer gebeurt. Het verschil in tijd is spectaculair bij grote sets levert dit al snel een verschil op van een factor 10!

5.2 Analysis of Variance (ANOVA)

Het algoritme hangt of staat met het veelvuldig berekenen van de afstand tussen twee punten. Met veel attributen wordt dit typisch snel zwaar – de zogenaamde *curse of dimensionality*. Als we dit probleem aan kunnen pakken, versnellen we 't algoritme bij de bron van alle ellende. Enter *Analysis of Variance*.

Het idee is eenvoudig: attributen waarvan de waarde weinig spreiding vertoont (en de gemiddelden van de klassen dus weinig verschillen) zeggen weinig over de klasse zelf. Deze attributen leveren alleen maar vertraging op en vertroebelen de afstandsfunctie – genoeg reden om ze eruit te filteren. Een goed voorbeeld is attribuut 0 in de laatste test-set: de linker bovenhoek van een te classificeren afbeelding. In elk van de voorbeelden in de trainset was dit attribuut 0, want de linker bovenhoek is altijd leeg typisch overbodige informatie.

Variantieanalyse gaat voor twee klassen redelijk eenvoudig door middel van een zogenaamde t-test. Bij deze test wordt de gemiddelde waarde van een attribuut berekend voor beide klassen, en wordt bekeken of deze gemiddelden significant (lees: ver genoeg) uit elkaar liggen. Zie figuur 8 Het probleem is alleen dat we meestal geen twee klassen hebben, maar soms enkele tientallen. Om dit probleem op te lossen is de zogenaamde Analysis of Variance (ANOVA) ontwikkeld, in de simpelste vorm als generalisatie van de t-test (*one-way* ANOVA).



Figuur 8: De gemiddelden verschillen bij attribuut 1 en 3 significant, bij nummer 2 niet. Vanwege de t-test reduceren we dit attribuut.

Gelukkig bestaat er een handige implementatie van one-way ANOVA. In het statistiek-package van Apache treffen we een functie aan met de geschikte signatuur en omschrijving: `public boolean anovaTest(Collection<double[]> categoryData, double alpha)`: *Performs an ANOVA test, evaluating the null hypothesis that there is no difference among the means of the data categories.* Dit is precies wat we zochten. Het enige wat ons nu rest is het in de goede vorm gieten van de data, aangezien deze functie lijsten van attribuutwaarden per klasse verwacht. Met wat handig gebruik van `HashMaps` lukt ook dat op een efficiënte manier.

Door gebruik te maken van ANOVA kunnen we een groot deel van de attributen weggooien – bij de laatste competitieset bijna de helft. Hiermee reduceren we een significant deel van de rekentijd die nodig is om de onderlinge afstand van punten te bepalen, waardoor ons algoritme sneller uitgevoerd kan worden.

5.3 Resultaten na verbetering

Natuurlijk zijn de resultaten uiteindelijk het belangrijkst. Na de bovenstaande optimalisaties kon ons algoritme grote datasets binnen afzienbare tijd ook classificeren, waardoor we de onderstaande resultaten konden behalen. Zonder de verbeteringen zou vooral het correct classificeren de laatste dataset onhaalbaar zijn geweest – toen haalden we scores van slechts 80 procent, waar 97 procent een minimum was. De resultaten van de spaties- en nieuwsgroepensets zijn overigens behaald met een gedeelte van de verbeteringen (aangezien die deadlines inmiddels voorbij zijn), dus vooral de karakterherkenning biedt een goede indicatie.

Testset	**Setgrootte**	**# Klassen**	**Accuracy**	**Reduction**	**Attr. reduc.**
Spaties	22x15000	2	75.6%	99.4%	36,4%
Nieuwsgroepen	20x1000	3	84.1%	97.4%	35.0%
Karakters	400x9876	26	97.9%	85.9%	80.1%

Figuur 9: Tabel met testresultaten na verbetering

Aangezien het algoritme nog altijd non-deterministisch is, hebben we om deze resultaten te behalen het programma een aantal keren moeten uitvoeren. Omdat het verkrijgen van resultaten bij een grote set nog altijd even duurt, was het handig om dit synchroon te kunnen doen. Bij deze spreken we daarom onze waardering uit voor de terminalkamers in het Huygensgebouw, die op een warme vrijdagmiddag zo heerlijk leeg zijn.