Automatically Estimate the Best K for K-Means Clustering with WhizzML

by hangartnerr on June 16, 2016
(Thanks to Alex Schwarm of **dnb.com** (http://www.dnb.com) for bringing to our attention **the Pham, Dimov, and Nguyen paper (https://www.ee.columbia.edu/~dpwe/papers/PhamDN05-kmeans.pdf)**, which is the subject of this post.)

The BigML platform offers a robust **K-Means Clustering** (https://bigml.com/api/clusters) API that uses the **G-Means algorithm (https://blog.bigml.com/2015/02/24/divining-the-k-in-k-means-clustering/)** for determining K if you don't have a good guess for K. However, sometimes you may find that the divisive top-down approach of the G-Means algorithm does not always yield the Best-K for your dataset. After a little experimentation, you may also discover that the G-Means algorithm does not choose a value of K that makes sense based on your knowledge of your dataset (see the *"k"* and *"critical_value"* arguments in the **Cluster Arguments (https://bigml.com/api/clusters#cl_cluster_arguments)** section). You could manually try running the cluster operation on your dataset for a range of K, but that approach does not inherently include a way to recognize the best K. And it can be very time consuming!

The Pham, Dimov, and Nguyen Algorithm and the K-means Algorithm in BigML

Fortunately, **WhizzML** (https://bigml.com/whizzml) allows us to easily implement another approach for choosing K using an algorithm by Pham, Dimov, and Nguyen. (D.T. Pham, S. S. Dimov, and C. D. Nguyen, "**Selection of *K* in K-means clustering (https://www.ee.columbia.edu/~dpwe/papers/PhamDN05-kmeans.pdf)**". *Proc. IMechE, Part C: J. Mechanical Engineering Science*, v. 219, pp. 103-119.) Pham, Dimov, and Nguyen define a measure of concentration *f(K)* on a K-means clustering and use that as an evaluation function to determine the best K. In this post, we show how to use the Pham-Dimov-Nguyen (PDN) algorithm in WhizzML to calculate *f(k)* over an arbitrary range of *Kmin* to *Kmax*. You can then consider using the *k* that yields the optimum (minimum) value of *f(k)* as the best *K* for a K-means clustering of your dataset.

Before jumping into the WhizzML code, we first note the clustering functions WhizzML provides via the BigML API calls:

(create-and-wait-cluster …): Using this function we can create a BigML **Cluster (https://bigml.com/developers/clusters)** object for a BigML **Dataset (https://bigml.com/developers/datasets)** object using K-means or G-means clustering.
(create-and-wait-centroid …): Once we have a BigML Cluster for a BigML Dataset we can create a BigML **Centroid (https://bigml.com/api/centroids)** object for a row in the dataset using this function.
(create-and-wait-batchcentroid …): Given a Cluster object and a Dataset object, we can use this function to create a BigML **Batch Centroid (https://bigml.com/api/batchcentroids)** object and a new Dataset that labels every row with the number of the cluster centroid to which the row is assigned.

(create* "cluster" …): With this function we can initiate the creation of a sequence of BigML Cluster objects on the BigML platform **in parallel**.
(wait* …): Although not a clustering function, this synchronization function re-establishes serial program flow in WhizzML after **(create* …)** initiates parallel creation of BigML objects.

We'll use the latter two parallel operations to increase the speed of our WhizzML script that implements the PDN algorithm.

Our WhizzML script in the BigML gallery uses the PDN concentration function *f(k)* and finds the best *K* in several steps. Given a BigML Dataset object, the steps of the generic algorithm are:

1. Compute a sequence of Bigml Cluster objects for *k* ranging from *Kmin* to *Kmax*.
2. Evaluate *f(k)* for each cluster in the sequence of BigML Cluster objects.
3. Choose the *k* with the optimum (minimum) value of *f(k)* as the best *K*.
4. Finally, if desired, create a BigML Batch Centroid object from the best *K* Cluster object and the source Dataset object.

It turns out that our example WhizzML script implements a sequence of component WhizzML functions that aren't quite one-to-one with the steps in this generic algorithm. The functions in our script are organized into three layers: The base layer are foundation functions to enable computation of the PDN concentration function *f(k)*. The functions in the middle layer use these foundation functions to implement our algorithm to find the best *k* for K-Means clustering of a dataset. The top layer are WhizzML functions that provide examples of different ways to use our best *k* implementation of K-Means clustering in your own workflows.

Foundation Functions for a PDN-based Approach to Finding the Best *k*

Our WhizzML script begins with a set of four simple foundation functions **(generate-clusters …)**, **(extract-eval-data …)**, **(alpha-func …)** and **(evaluation-func …)**. The **(generate-clusters …)** function implements the first step in the generic algorithm we outlined. Given a BigML dataset ID and a range for values of *k*, this script creates a sequence of BigML Cluster objects:

```
(define (generate-clusters dataset cluster-args k-min k-max)
  (let (dname (get (fetch dataset) "name")
        fargs (lambda (k)
                (assoc cluster-args "dataset" dataset
                                    "k" k
                                    "name" (str dname " - cluster (k=" k
        clist (map fargs (range k-min (+ 1 k-max)))
        ids (create* "cluster" clist))
    (map fetch (wait* ids))))
```

In addition to the *"dataset"* ID and range for *k* specified by *"k-min"* and *"k-max"*, the function accepts a map *"cluster-args"* of arguments for the BigML API to create Cluster objects. This base *"cluster-args"* map is expanded to a map for a specific value of *k* by the function *fargs(k)* created as a lambda function.

The rest of the function creates the *clist* of argument maps for each value of *k* and uses the WhizzML **(map …)** function. The WhizzML **(create* …)** and **(wait* …)** functions are then used to create the BigML Cluster objects for *k* in *"k-min"* to *"k-max"* in parallel. The function then returns a list of the metadata for the resulting clusters on the BigML server.

As we will explain subsequently, the PDN concentration function *f(k)* for a given *k* is computed from certain members of the metadata map for the cluster object for *k*. To illustrate this and simplify the code, the next helper function **(extract-eval-data …)** in the script encapsulates the required values from the metadata map in a separate map:

```
(define (extract-eval-data cluster)
  (let (id (get cluster "resource")
        k (get cluster "k")
        n (count (get cluster "input_fields"))
        within_ss (get-in cluster ["clusters" "within_ss"])
        total_ss (get-in cluster ["clusters" "total_ss"]))
    {"id" id "k" k "n" n "within_ss" within_ss "total_ss" total_ss}))
```

In addition to the BigML cluster *"id"* and *"k"*, this smaller map includes the number *"n"* of fields in the dataset that are actually considered when doing the clustering. The *"within_ss"* property is the total sum-squared distance between every dataset row in the cluster and the centroid of the cluster. Similarly, *"total_ss"* is the total sum-squared distance between every row in the entire dataset and the global centroid of the dataset. Therefore, it will be the same value for each cluster.

The next two functions **(alpha-func …)** and **(evaluation-func …)**, are actually factory functions that together create the PDN concentration function *f(k)* for a clustering. This function includes an internal weighting function *a(k)* parameterized on the number *n* of input fields considered in clustering the dataset. WhizzML does not provide an equivalent to the LISP **(apply …)** or the Clojure **(partial …)** for creating partial function evaluations, but it does create standard closures. This allows us to use Javascript methods based on lambda functions and closures to build the PDN concentration function *f(k)* parameterized on *n* in WhizzML. We do this by using a factory function **(alpha-func …)** that returns the weighting function *a(k)*, and a factory function **(evaluation-func …)** that returns a custom version of the concentration function *f(k)*.

The concentration function *f(k)* in the PDN paper incorporates a weighting function *a(k)* that is recursive in *k* and parameterized on *n* (eqns. (3a) and (3b) in the paper). Because we want to evaluate *f(k)* over an arbitrary range of *k*, we need a closed form expression for *a(k)*. We can't go through the derivation here, but the closed form we need is:

```
        | 1 - 3/4n                                          k=2
a(k) =  |
        | (5/6)^(k-2) a(2) + [1 - (5/6)^(k-2)]    k>2
```

We could write our factory function **(alpha-func …)** in multiple ways. The implementation in our WhizzML script follows a simple Javascript pattern that returns an anonymous function:

```
(define (alpha-func n)
  (let (alpha_2 (- 1 (/ 3 (* 4 n)))
        w (/ 5 6))
    (lambda (k)
      (if (<= k 2)
        alpha_2
        (+ (* (pow w (- k 2)) alpha_2) (- 1 (pow w (- k 2))))))))
```

This factory function implicitly creates a closure that captures the input parameter *"n"* and then returns a lambda function that computes *a(k)*.

We next use **(alpha-func …)** in our factory function **(evaluation-func …)** that creates the concentration function *f(k)*. As with the weighting function *a(k)*, since we want to evaluate *f(k)* over an arbitrary range of *k* we need to slightly transform *f(k)* in the PDN paper (eqn. (2)):

```
                        | 1                    k=1
f(k, S(k), S(k-1)) =    | 1                    S(k-1) undefined or S(k-1)=0
                        | S(k)/[a(k)S(k-1)]    otherwise
```

where *S(k)* is the *"within_ss"* property in the map returned by the **(extract-eval-data …)** function we described above. Our factory function implements the simple Javascript pattern that returns an anonymous function:

```
(define (evaluation-func n)
  (let (fa (alpha-func n))
    (lambda (k sk skm)
      (if (or (<= k 1) (not skm) (zero? skm))
        1
        (/ sk (* (fa k) skm))))))
```

This factory function accepts the single input parameter *"n"*, implictly creates a closure that includes an instance of the weighting function *a(k)*, and then returns an anonymous instance of our modified concentration function *f(k, S(k), S(k-1))*.

At this point it's worth recapping the functions we've built so far. In just a few lines of WhizzML code, we've implemented four routines that form the foundation layer of the Best-K script in the WhizzML script gallery and illustrate the power of WhizzML. The **(generate-clusters …)** function orchestrates a potentially large amount work on the BigML backend to create a sequence of BigML

cluster objects for K-means clusterings of our dataset over a range of *k*. Each BigML cluster object itself embodies a large amount of data and metadata, so we've defined a function **(extract-eval-data ...)** that you could customize further in your own WhizzML scripts to extract just the metadata we'll need. Finally, we've implemented two factory functions **(alpha-func ...)** and **(evaluation-func ...)** that together generate a version of the Pham-Dimov-Nguyen concentration function *f(k)* suitable for our needs.

Using Our Foundation Functions to Implement a Best *k* Algorithm

We next combine our foundation functions with other WhizzML built-in functions in a set of three functions at the heart of our implementation of the PDN algorithm for choosing the best K-means clustering. The first function **(evaluate-clusters ...)** accepts a list of clusters created by **(generate-clusters ...)** and returns a corresponding list of metadata maps:

```
(define (evaluate-clusters clusters)
  (let (cmdata (map extract-eval-data clusters)
        n (get (nth cmdata 0) "n")
        fe (evaluation-func n))
    (loop (in cmdata
           out []
           ckz {})
      (if (= [] in)
        out
        (let (ck (head in)
              ckr (tail in)
              k (get ck "k")
              within_ss (get ck "within_ss")
              within_ssz (if (<= k 2) (get ck "total_ss") (get ckz "wit
              cko (assoc ck "fk" (fe k within_ss within_ssz)))
          (recur ckr (append out cko) ck))))))
```

Each metadata map in the returned list includes a property *"fk"* that is the value of the PDN function *f(k)* for the corresponding K-means clustering.

This function uses **(extract-eval-data ...)** to build a list *cmdata* of metadata maps for the list of K-means clusterings, and the factory function **(evaluation-func ...)** to create a function *"fe"* that is our version *f(k, S(k), S(k-1))* of the PDN concentration function *f(k)*. The body of the function is a WhizzML **(loop ...)** function that steps through the input list *"in"* of metadata maps (initially the *cmdata* list) to sequentially generate the output list *"out"* of metadata maps. The loop body operates on the head metadata map of the *"in"* list and the metadata map from the head member of the last iteration *"ckz"* to compute the *"fk"* property, and then appends an augmented metamap map to the output list *"out"*. We note that the input list of *"clusters"* spans an arbitrary range of *k* and that the computation to generate *"within_ssz"* generates the initial value for *S(k-1)* required by our concentration function *f(k, S(k), S(k-1))* for the first cluster in the *"clusters"* list.

Our next two functions are helper functions used by our top level functions we describe next. The first function **(clean-clusters …)** just deletes unneeded BigML Cluster objects created by our PDN-based algorithm:

```
(define (clean-clusters evaluations cluster-id logf)
  (for (x evaluations)
    (let (id (get x "id")
          _ (if logf (log-info "Testing for deletion " id " " cluster-id
      (if (!= id cluster-id)
        (prog (delete id)
              (if logf (log-info "Deleted " id))))))
  cluster-id)
```

We note that this function includes an input parameter *"logf"*. When this parameter is *true,* the function logs information about the delete operation to the BigML logging system. The function is intended to be a base example you could expand with additional logging information in your own version of the script.

The other function **(best-cluster …)** generates a new BigML Cluster object:

```
(define (best-cluster dataset cluster-args k)
  (let (dname (get (fetch dataset) "name")
       ckargs (assoc cluster-args "dataset" dataset
                                  "k" k
                                  "name" (str dname " - cluster (k=" k
    (create-and-wait-cluster ckargs)))
```

This helper function is intended to increase the flexibility of our WhizzML script. In the initial evaluation stage we generate a list of BigML Cluster objects using the **(generate-clusters …)** function using an arbitrary map *"cluster-args"* of values for the BigML **clustering operation arguments (https://bigml.com/api/clusters# cl_cluster_arguments)**. Using this helper function, we can generate a final version of the BigML Cluster object for a given *k* using a different *"cluster-args"* map.

Before introducing the final top level functions in our example WhizzML script, we can add a few additional notes. First note that our middle level functions only access data in WhizzML to do their work, they don't need to access the BigML Cluster objects in the BigML system after we created the BigML Cluster objects with the **(generate-clusters …)** function. Correspondingly, our example **(clean-clusters …)** WhizzML function queues the object deletion requests to the BigML platform but doesn't need to wait for them to complete. Finally, although the sample **(best-cluster …)** function allows us to regenerate the K-means clustering for the best *k* and waits for BigML to complete, you could just queue the request to create the BigML Cluster object in your own custom WhizzML script and check if it is complete with the **(wait …)** function when you need it. The

BigML platform takes care of all the cumbersome work of creating and deleting objects, and just provides our WhizzML code with the small out of data we need. This greatly simplifies orchestrating and optimizing the performance of our workflows.

Functions that Illustrate Several Applications of the PDN Best *k* Approach

The final group of functions in our example WhizzML script are three simple top level functions that provide us with a stack of operations relevant to different applications. We step through them in order. We then provide example WhizzML calls of each function.

The first top-level function **(evaluate-k-means …)** just creates the list of BigML Cluster objects for K-means clustering for *k* ranging from *"k-min"* to *"k-max"* and returns the list of metadata maps that includes the value of the PDN concentration function *f(k)* as the property *"fk"*:

```
(define (evaluate-k-means dataset cluster-args k-min k-max clean logf)
  (let (clusters (generate-clusters dataset cluster-args k-min k-max)
        evaluations (evaluate-clusters clusters))
    (if clean
      (clean-clusters evaluations "" logf))
      evaluations))
```

In addition to the basic input parameters *"dataset"*, *"k-min"*, and *"k-max"*, the function allows us to specify a WhizzML map *"cluster-args"* of our choice of arguments for the BigML cluster operation. When the "clean" parameter is true, it causes the function to call the (clean-clusters …) function to optionally delete the BigML Cluster objects on the BigML platform before returning the result list. In this example function, the value of the parameter "logf" is just passed on to the (clean-clusters …) function. In your own custom version of this WhizzML script you can use this parameter to control whatever additional logging you might want.

Our next function **(best-k-means …)** builds on **(evaluate-k-means …)** to return a BigML Cluster object for the best *k*:

```
(define (best-k-means dataset cluster-args k-min k-max bestcluster-args
  (let (evaluations (evaluate-k-means dataset cluster-args k-min k-max f
        _ (if logf (log-info "Evaluations " evaluations))
        besteval (min-key (lambda (x) (get x "fk")) evaluations)
        _ (if logf (log-info "Best " besteval))
        cluster-id (if (= cluster-args bestcluster-args)
                     (get besteval "id")
                     (best-cluster dataset bestcluster-args (get besteva
    (if clean
      (clean-clusters evaluations cluster-id logf))
    cluster-id))
```

After we generate the list *evaluations* of metadata maps with the PDN concentration function values, we used the WhizzML **(min-key …)** built-in function to find the metadata map for the best *k*. We then check if the *"cluster-args"* map used in the first stage when we find k differs from the *"bestcluster-args"* map. If the two maps don't agree, we generate a new BigML Cluster object for the best *k*. Regardless, if *"clean"* is specified as *true*, we direct the BigML platform to asynchronously delete the BigML Cluster objects on the platform that we don't need. Finally, we return the ID of the BigML Cluster object for the best *k* the routine found.

Our last routine **(best-batchcentroid …)**, uses the BigML Cluster object created by the **(best-k-means …)** function and the input BigML Dataset object to create a BigML Batch Centroid object:

```
(define (best-batchcentroid dataset cluster-args k-min k-max bestcluster
  (let (cluster-id (best-k-means dataset cluster-args k-min k-max bestcl
        batchcentroid-id (create-and-wait-batchcentroid {"cluster" clust
                                                         "dataset" datas
                                                         "output_dataset
                                                         "all_fields" tr
    batchcentroid-id))
```

Because the argument map in our call to the WhizzML **(create-and-wait-batchcentroid…)** function includes the *"all_fields"* snf the *"output_dataset"* properties, the function also creates a BigML Dataset object that includes all columns in the input *"dataset"* and an extra column that specifies the cluster number to which the dataset row was assigned.

Using the Best k Algorithm Implementation

All of our top 3 WhizzML functions have the same parameters, so we can call them in the same way:

```
(define bestk-evaluations (evaluate-k-means dataset cluster-args k-min k

(define bestk-cluster (best-k-means dataset cluster-args k-min k-max bes

(define bestk-batchcentroid (best-batchcentroid dataset cluster-args k-m
```

These three examples illustrate how we compute a list of PDN concentration function *f(k)* evaluations, the BigML Cluster object for the best *k*, and the BigML Batch Centroid object for the best *k*, respectively.

In your application, you might have a guess for the best *k*. In that case, you might want to specify a range *"k-min"* to *"k-max"* that brackets that *k* value. You could then use the first call to the **(evaluate-k-means …)** function above, examine the results, and choose the best *k*. Alternatively you could use **(evaluate-k-means …)** in a loop to test a series of intervals [*k_1*,*k_2*], [*k_2*,k_3] …

[*k_N-1*, *k_N*], and then choose the best *k* from all of those tests. Finally, if you know a range *"k-min"* to *"k-max"*, you can use **(best-k-means ...)** or **(best-batchcentroid ...)** to generate the BigML Cluster object or BigML Batch Centroid object, respectively, for the best *k*.

From → Algorithms, Clusters, Data, WhizzML

Leave a Comment