Introduzione

1. Problema del clustering

Il problema del clustering è un problema nato nell'ambito dell'unsupervised learning (ovvero il learning su dati senza label) e consiste nel partizionare un insieme di esempi senza label in sottoinsiemi disgiunti (i cluster) in modo tale da massimizzare la somiglianza tra gli elementi all'interno dei cluster (infra-cluster similarity) e massimizzare la differenza tra gli elementi di cluster differenti (inter-cluster dissimilarity); può essere applicato praticamente ad ogni campo, per esempio in ambito commerciale potremmo dividere i clienti secondo le loro similarità per poi proporre prodotti più appropriati alle categorie di clienti dedotte.

Gli algoritmi di clustering si dividono in gerarchici e diretti:

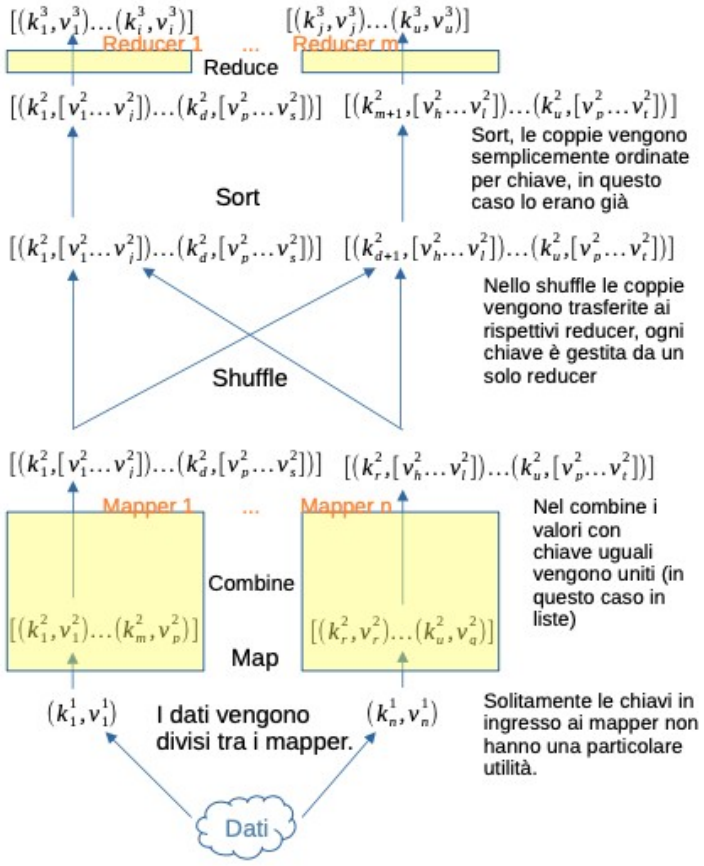
- gli algoritmi gerarchici hanno una maggiore complessità computazionale in quanto producono un albero che ha come foglie i singoli elementi e come nodi le categorie (un problema differente, invece, è quello di nominare queste categorie!); per esempio un algoritmo di questo tipo applicato ad un insieme di esseri viventi dovrebbe produrre una vera e propria tassonomia.
- gli algoritmi diretti hanno complessità minore perché non devono costruire un albero ma dato k il numero di cluster (cioè la granularità con cui osserveremo il dataset) producono direttamente una (sola) clusterizzazione e non a più livelli come nel caso precedente. Il problema principale di questi algoritmi è la scelta dei k cluster iniziali (o equivalentemente I rispettivi centroidi) in quanto potrebbe non essere noto né quante sono le categorie attese (quindi il k ideale) né l'appartenenza degli elementi a ciascun cluster; in effetti quest'ultima può incidere così tanto sul risultato che essa può venire chiamata anche scelta del seed.

Le differenze tra gli algoritmi diretti riguardano solitamente il tipo di distanza usata (si possono usare similarità ma devono essere "riportate" a comportarsi come una distanza, ad esempio due elementi uguali hanno distanza 0 ma similarità 1) e il criterio di stop (per esempio: nessuna o minima riassegnazione di esempi e nessun o minimo cambio nell'insieme dei centroidi).
Nel caso specifico useremo (come richiesto) l'algoritmo diretto **K-Means** originale che per la distanza utilizza quella euclidea e per il criterio di stop l'uguaglianza dei centroidi rispetto all'iterazione precedente.

2. Modello di Programmazione MapReduce

Si ispira alle funzioni di programmazione funzionale map e reduce.
Il map applica una certa funzione singolarmente ad ogni elemento di una lista; il reduce invece applica una funzione a tutti gli elementi della lista insieme. Si intuisce quindi che la funzione map è in grado di effettuare un mapping tra input e output e la funzione reduce di ridurre la dimensione dell'input.

Il MapReduce quindi si compone di due fasi che corrispondono alle due funzioni da cui deriva il suo nome; la sua funzione principale è di consentire di distribuire la computazione di grandi quanità di dati efficientemente, la "magia" è data dall'utilizzo, in entrambi le fasi, di coppie chiave valore (invece che solamente valori come le funzioni map e reduce originali). In particolare la chiave tra l'output della fase di map e l'input della fase di reduce. Di seguito una rappresentazione dello stack che rappresenta un sistema MapReduce.



Si può notare che gli apici cambiano tra ingresso e uscita del mapper, e ingresso e uscita del reducer; ho aggiunto questo particolare per mettere in evidenza che tra input ed output di essi non ci sono le stesse chiavi (o comunque non hanno necessariamente la stessa semantica).
Tra l'output del mapper e l'input del reducer, invece, le chiavi sono sempre le stesse e servono proprio a riunificare il lavoro svolto indipendentemente da ciascun mapper pur mantenendolo diviso in chiavi e quindi riuscire a "dominare" la grandezza dei dati.

3. Programmi utilizzati

Vim, go, ansible, docker-compose, aws-cli, Makefile, terraform, cURL, git, ssh, gnumeric.
Per le macchine virtuali ho utilizzato Debian 10.13 (perché è la distribuzione che uso di solito).

4. Deployment

Attenzione, queste sono le versioni del software che ho usato, non posso garantire il funzionamento con versioni differenti:

- Terraform 1.2.9
- Ansible 2.10.8
- Go 11.1
- Docker-compose 1.21

Creare un cartella per il progetto, al suo interno clonare i seguenti repository:

- github.com/sgaragagghu/sdcc-go
- github.com/sgaragagghu/sdcc-ansible
- github.com/sgaragagghu/sdcc-docker
- github.com/sgaragagghu/sdcc-terraform

Dopodiché:

1. Configurare AWS.
2. Configurare terraform.project (profile, region, vpc_security_group_ids, subnet_id).
3. Creare la cartella keypair (nella directory radice del progetto) e inserirvi la coppia di chiavi per poterci connettere via SSH alla macchina virtuale.
4. Mandare in run “terraform apply” per creare la macchina virtuale.
5. Recuperare l’ip della macchina virtuale ed inserirlo nel file hosts.ini (nella directory sdcc-ansible).
6. Configurare la quantità di mapper e reducer in deploy.yaml (ultime due righe).
7. Utilizzare lo script deploy.sh (contiene semplicemente il comando per avviare ansible) per fare in modo che ansible configuri in modo adeguato la macchina virtuale.
8. Connettersi alla macchina virtuale via ssh… (utilizzavo i comandi contenuti in un set di script github.com/sgaragagghu/sdcc-ssh)
9. Inviare i task via HTTP con cURL, c’è un insieme di esempi nella directory sdcc-go/tasks
10. Leggere i log con il comando “docker-compose logs master” (cambiare master con mapper o reducer per leggere i rispettivi log).

5. Sviluppo

1. Sebbene avessi conoscenza del problema del clustering, tutto il resto mi era nuovo.
Studiare Go, MapReduce, ansible, terraform, docker-compose, aws.
(dal 6 al 19 settembre, 50% del giorno. Nelle successive fasi ho usato il 100% del tempo giornaliero (mediamente 7 ore) in quanto mi sono reso conto che altrimenti non avrei fatto in tempo)

2. Memore di problemi, in altri progetti, riguardanti il deployment, questa volta ho voluto sin da subito cercare di automatizzare il deployment per sviluppare il programma direttamente nell’ambiente dove avrebbe poi lavorato.
Configurazione di terraform, ansible, docker-compose e vari parametri di aws con awscli.
(dal 20 al 27 settembre)

3. Organizzazione dei file sorgenti e sviluppo di funzionalità di supporto essenziali quali heartbeat e logging.
(dal 28 settembre al 4 ottobre)

4. Sviluppo del task manager sia lato master che lato mapper (servono entrambi per poterne testare almeno uno!)
Sviluppo delle funzioni di task managing(master, mapper e si include gestione di crash del mapper), parser, comunicazione, partizione del carico e preparazione di un task di prova.
(dal 8 ottobre al 18 ottobre)

5. Sviluppo del task manager sia lato master che reducer (sostanzialmente un adattamento di quello per il mapper) e della comunicazione diretta tra reducer e mapper.
(dal 19 al 27 ottobre)

6. Sviluppo della funzionalità per permettere di iterare (necessario per gli algoritmi di clustering)
Inoltre è stato necessario risolvere alcuni dei debiti tecnici accumulati in

tutte le precedenti fasi (per fretta), in quanto il codice risultava quasi incomprensibile e non era possibile continuare.
(dal 28 ottobre al 2 novembre)

7. Mi sono reso conto di aver sbagliato* (un caso di errore non era coperto!), ho dovuto modificare abbastanza pesantemente lo scheduler lato master.
Overhaul scheduler e diminuzione debito tecnico e testing con più mapper e reducer insieme
(dal 3 al 7 novembre)

8. Funzione per ricevere nuovi task da eseguire e per richiedere il risultato, tramite JSON RPC
(dal 8 al 12 novembre)

9. Pulire il codice e risolvere molti dei debiti tecnici accumulati (controllo overflow, gestione di alcuni casi di errore, armonia tra i nomi delle variabili... etc)
(dal 13 al 17 novembre)

10. Scrittura relazione, realizzazione e processamento di task per testare sia l'algoritmo di clustering che il programma e correzione dei bug.
(dal 18 al 30 novembre)

6. Spirito e idea iniziale del programma

1. Thread e comunicazione tra essi
Solitamente nei miei programmi la comunicazione tra thread* avviene tramite memoria condivisa e uso di lock; avevo da tempo desiderio di provare un approccio diverso, ovvero pochi thread e lascamente sincronizzati, il caso ha voluto che fosse anche il "mantra" di Go: "Do not communicate by sharing memory; instead, share memory by communicating".
In Go, quindi, si fa largo uso dei canali di comunicazione (è una primitiva del linguaggio); non è magia… comunque sono implementati con lucchetti e memoria condivisa! Però nascondere tutto ciò rende il lavoro del programmatore (almeno inizialmente), meno oneroso.

*In go abbiamo le goroutine, implementazione differente dai pthreads.

2. Disaccoppiare la logica di MapReduce dalla funzione di Map e di Reduce
L'idea iniziale prevedeva di fornire il codice della funzione Map e della funzione Reduce come due campi testo dell'oggetto JSON rappresentante il task.
Go però è un linguaggio compilato e non interpretato, quindi implementare questa funzionalità avrebbe richiesto troppo tempo; ho dovuto ripiegate sul fornire solo il nome della funzione tra quelle già disponibili all'interno del programma.

3. Siccome MapReduce si utilizza solitamente per dati così grandi da non poter nemmeno essere ospitati su un solo server, nessun elemento nel sistema ha mai l'intera mole di dati ma solamente una porzione di essi.

4. Un obiettivo è stato il risparmio di banda, infatti un collo di bottiglia dei sistemi MapReduce è la trasmissione dati.
P.S. Durante il testing mi sono accorto che ad ogni iterazione i mapper devono scaricare nuovamente la porzione di dati ad essi assegnata, sarebbe furbo che dal lato master si assegnino i job in modo tale che i server possano riulizzare la ciò che hanno scaricato all'iterazione precedente. (Non sono stati implementati nè la cache lato mapper, nè questa politica di assegnazione lato master)

Componenti principali del programma

1. Heartbeat

Sia i mapper che i reducer comunicano la loro presenza tramite heartbeat, un prolungato mancato heartbeat verrà inteso dal master come un leave (volontario o no). L'indirizzo del master è quindi hard-coded nel codice dei worker.

2. Map algorithm

Deve essere scelto nella definizione del task e svolge diversi compiti:

1. Recupero della porzione di dati assegnata dal master; nel caso specifico è prevista come sorgente un url (pensiamo ad eempio ad un file di diversi terabyte ospitato su S3) associato ad un range di byte.
2. Il parsing dei dati scaricati.
3. La funzione map.
4. Combinamento.
5. Encoding del risultato (deve essere contenuto in una variabile `map[string]interface{}`).

3. Reducer algorithm

Deve essere specificata nella definizione del task e svolge la funzione reduce; è preceduta da una funzione a parte per recuperare i dati dai rispettivi mapper.

4. Join algorithm

Siccome la codifica del risultato dipende dall'algoritmo map utilizzato, è necessaria una funzione join per poter unire più coppie chiave valore (per esempio nella funzione del reducer che recupera i dati dai mapper è necessario effettuare il combining delle coppie con stessa chiave ma provenienti da diversi mapper).

5. Iteration algorithm

Anch'esso specificato nella definizione del task, viene svolto al termine di ciascun task e decide se generare il nuovo task che rappresenta l'iterazione successiva.

6. Initialization algorithm

Scelto nella definizione del task, funzione necessaria ad inizializzare le strutture dati per l'esecuzione del task.

7. Map scheduler lato master

Svolge le seguenti funzioni:

1. Tenere traccia dei mapper in funzione.
2. Tenere traccia dei job ed a quali mapper essi fossero stati assegnati.
3. Creare i job a partire dalla definizione del task.
4. Assegnare i job ai mapper.
5. Ricevere, da parte dei mapper, la notifica di terminazione dei job.
6. Capire quando un task è terminato, ovvero che tutti i suoi job siano stati completati.
7. Ri-schedulare i job che erano stati assegnati ad un server rimosso (eg. crashato).
8. Tenere traccia della posizione (cioè in quali mapper) dei risultati per ciascuna chiave.
9. Avvertire il reduce scheduler qualora un mapper avesse crashato e, quindi, la posizione di una o più chiavi fosse variata.

8. Reduce scheduler lato master

Svolge le seguenti funzioni:

1. Tenere traccia dei reducer in funzione.
2. Tenere traccia dei job ed a quali reducer essi fossero stati assegnati.
3. Creare i job a partire dalla definizione del task.
4. Assegnare i job ai reducer.
5. Ricevere, da parte dei reducer, la notifica di terminazione dei job.
6. Capire quando un task è terminato, ovvero che tutti i suoi job siano stati completati.
7. Ri-schedulare i job che erano assegnati ad un server rimosso.
8. Notificare lo scheduler mapper della terminazione dei task in modo tale che esso possa eliminare tutte le strutture dati annesse e che a sua volta possa avvertire i mapper (così che possano eliminare i risultati del task)

9. Scheduler lato mapper

Esegue i job e notifica il master della terminazione allegando la lista di chiavi risultanti (senza i valori).
Gestisce le richieste dei valori per talune chiavi, da parte dei reducer.

10. Scheduler lato reducer

Esegue i job e notifica il master della terminazione allegando la lista di coppie risultante.

11. Gestione errori

Il programma di per sé presenta una elevata complessità (rispetto alla mia conoscenza degli strumenti, del problema ed il tempo che ho avuto a disposizione) quindi ho cercato di semplificare al massimo la gestione degli errori:

- I mapper e reducer avranno un errore fatale in qualsiasi caso di malfunzione (compreso errori di comunicazione).
- Il master invece, in caso di errore di comunicazione con un mapper o reducer lo considererà crashato, nel resto dei casi, anch’esso terminerà l’esecuzione.

12. Comunicazione

Per mantenere lasca sincronizzazione tra le componenti (che voleva essere parte della filosofia del programma), la comunicazione tra mapper, reducer e master avviene tramite RPC (livello trasporto (tcp) e asincrono, le risposte vengono comunicate tramite callback ed immesse in rispettive code); questa scelta comporta una complessità maggiore nella gestione della comunicazione ma grazie alla gestione degli errori estremamente semplice sono riuscito a mantenere la complessità della comunicazione (soprattuto nei casi di errori) accettabilmente bassa.

La comunicazione con un eventuale front-end (non ho fatto in tempo) avviene tramite RPC a livello applicativo (HTTP) utilizzando il formato JSON, sincrona differita. Quindi può interfacciarsi facilmente con un client HTTP, es. cURL.

Problemi riscontrati

1. Complessità dello scheduler

Inizialmente lo scheduler mapper e scheduler reducer (lato master) dovevano essere completamente disaccoppiati in quanto avevo considerato che una volta che i mapper avevano completato avrei potuto disinteressarmi di loro (rispetto al task in corso).
Probabilmente nella fase iniziale, in cui pensavo come realizzare il programma, ero propenso a scrivere i risultati dei mapper su uno storage esterno es. S3 ma poi devo aver cambiato idea (un collo di bottiglia dei sistema MapReduce è il trasferimento dati) e mi sono dimenticato che avevo bisogno di questo vincolo per mantenere completamente disaccoppiati i due scheduler!

2. RPC asincrona

Ho utilizzato l’implementazione di RPC contenuta nella libreria standard ed ho avuto difficoltà a gestire la richiesta, da parte dei reducer, e successiva ricezione di dati, da parte dei mapper.
Avendo avuto più tempo avrei probabilmente valutato le capacità del “done channel” (è una funzione dell’ implementazione di RPC standard Go) e se non sufficiente avrei valutato di cambiare con gRPC (che sembra decisamente più sviluppato, per quanto riguarda l’asincronia).

3. Gestire la modularità

Ovvero rendere la struttura del programma disaccoppiata dall’implentazione del map e reduce. Il risultato finale, però, sembra apprezzabile; necessita sicuramente di ulteriore polishing (es. armonia tra le strutture dati, la loro posizione e i loro nomi).

4. Gestire i dati e le relazioni

Mantenere le informazioni e relazioni riguardo server, task, job, key è stato dispendioso. Probabilmente se avessi saputo a cosa sarei andato in contro avrei probabilmente usato un DB relazionale.

5. Tracing degli errori

Non ho fatto in tempo ma in un programma che andasse in produzione sarebbe sicuramente necessario gestire questo aspetto.

Test di performance

1. Introduzione

I task ed i loro risultati si trovano nella directory sdcc-go/tasks.

Mi sono limitato a misurare il tempo solo per il primo run, avrei dovuto creare una feature per fare molti run e poi calcolare la media del tempo ma sarebbe stato dispendioso e poco utile in questo contesto qualitativo.
Oltretutto quello che è interessante è il tempo medio per iterazione, quindi è già di per se una media su più run (sebbene pochi).

2. 2 dimensioni e 4 classi, versione 4, 100MB (6200000 entry)

Identificativo del task: *2d-4c-v4-100MB*
(Mi sono limitato a 100MB perché è il limite di Github ed a 2 vCPU perché non sono riuscito ad avviare tipi di istanze con più vCPU; suppongo sia un limite della versione studenti)

L'obiettivo di questo test è mostrare che effettivamente il programma distribuisce il carico.

a. t2.small (1 vCPU)

- Con 1 mapper e 1 reducer ha impiegato 94 secondi e 7 iterazioni, ovvero 13,43 secondi/iterazione.
- Con 2 mapper e 2 reducer ha impiegato 246 secondi e 18 iterazioni, ovvero 13,66 secondi/iterazione.
- Con 4 mapper e 4 reducer ha impiegato 83 secondi e 6 iterazioni, ovvero 13,8 secondi/iterazione.

Come volevo mostrare, visto che la macchina virtuale ha solo 1 vCPU, aumentare i thread non aumenterà le performance... (anzi, si nota un leggero peggioramento delle performance dovuto all'overhead di avere più thread)

b. t2.large (2 vCPU)

- Con 1 mapper e 1 reducer ha impiegato 120 secondi in 10 iterazioni, ovvero 12 secondi/iterazione.
- Con 2 mapper e 2 reducer ha impiegato 37 secondi in 5 iterazioni, ovvero 7,4 secondi/iterazione.
- Con 4 mapper e 4 reducer ha impiegato 44 secondi in 7,33 secondi.

Qui si nota che le prestazioni migliorano dal primo al secondo caso perché sfruttiamo entrambe le vCPU, ma non migliora dal secondo al terzo caso perché ci sono più thread che vCPU.

Test di clustering

1. Introduzione

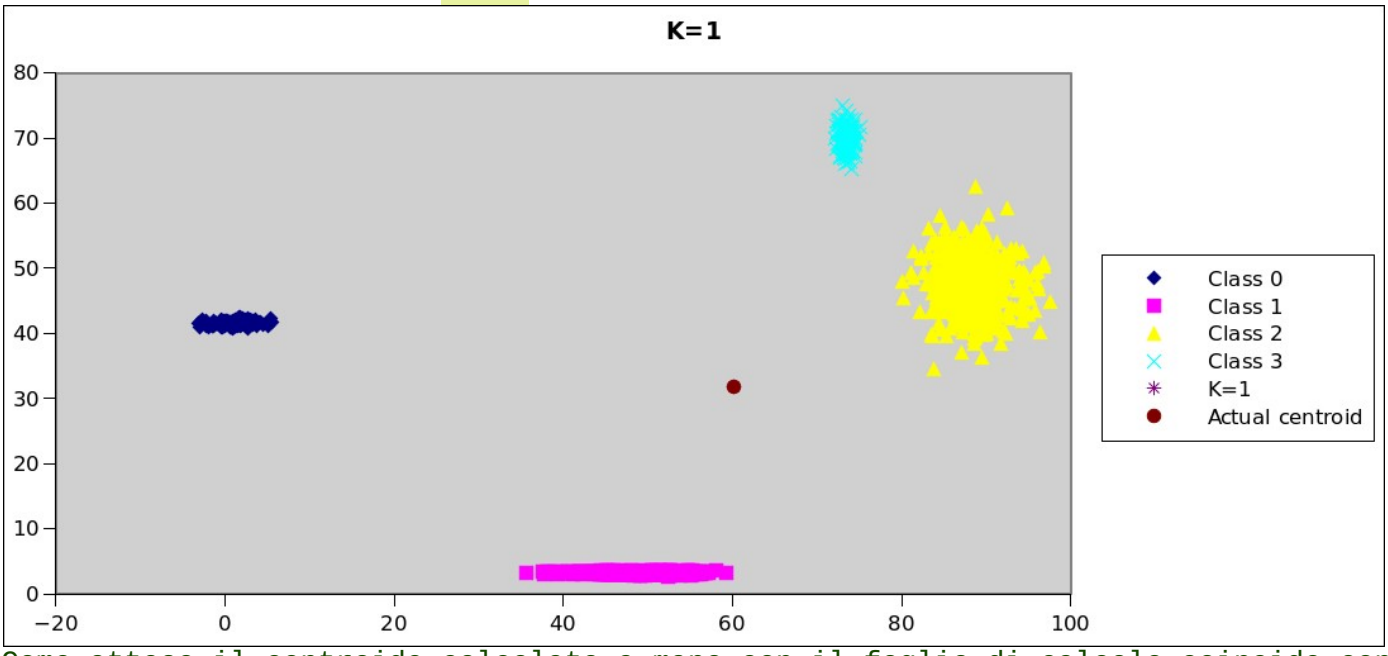
I task ed i loro risultati si trovano nella directory sdcc-go/tasks.
I dataset, fogli di calcolo ed i grafici invece in sdcc-clustering-datasets/sdcc (repository: github.com/sgaragagghu/sdcc-clustering-datasets).
Ad ogni run i risultati possono variare, ma dalla disposizione e forma delle classi si riesce a capire, tendenzialmente, in quali casi funzionerà spesso "bene" ed in quali altri spesso male.

Ricordo che nel caso "reale" non sappiamo quante classi sono presenti e se sono state "azzeccate". E' necessario uno o più criteri di valutazione (interno, esterno o indiretto…).

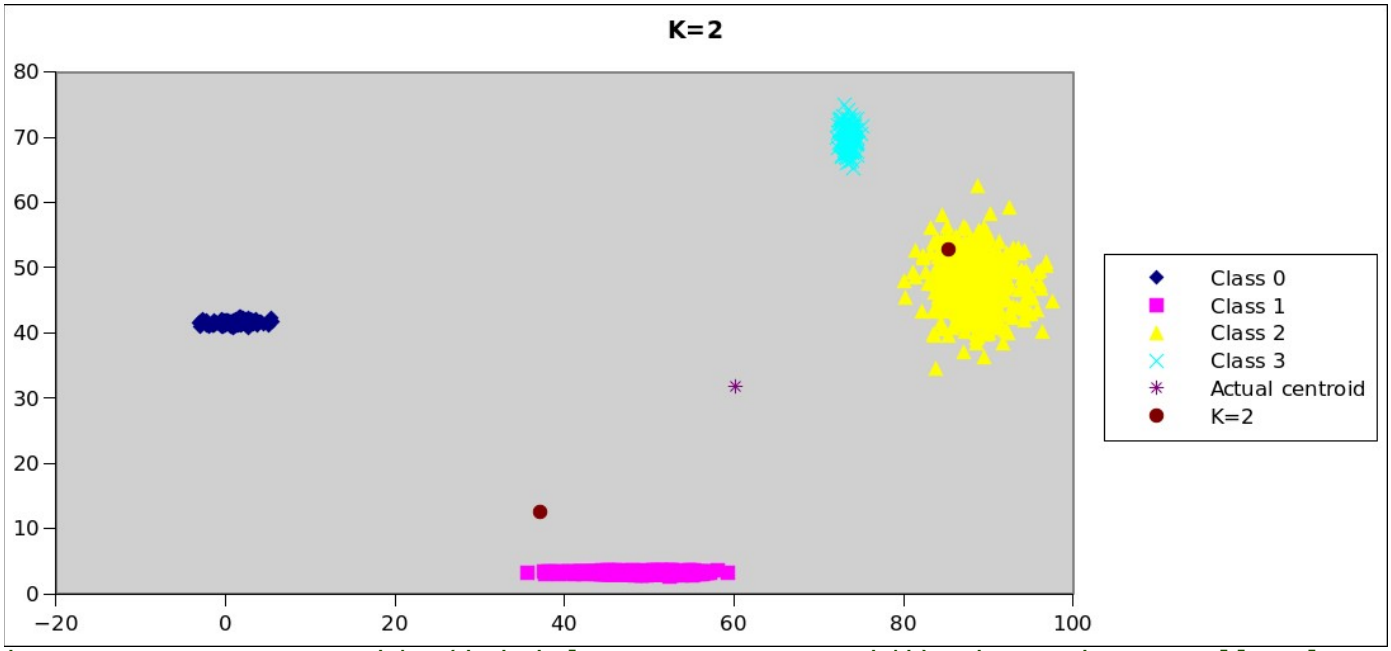
Avevo pianificato di implementare anche k-means++ e k-means** ma non ho avuto tempo… come vedremo i risultati non sono molto buoni e sarebbe stato interessante confrontarli con gli altri algoritmi di k-means (oltre che diverse distanze).

2. 2 dimensioni e 4 classi

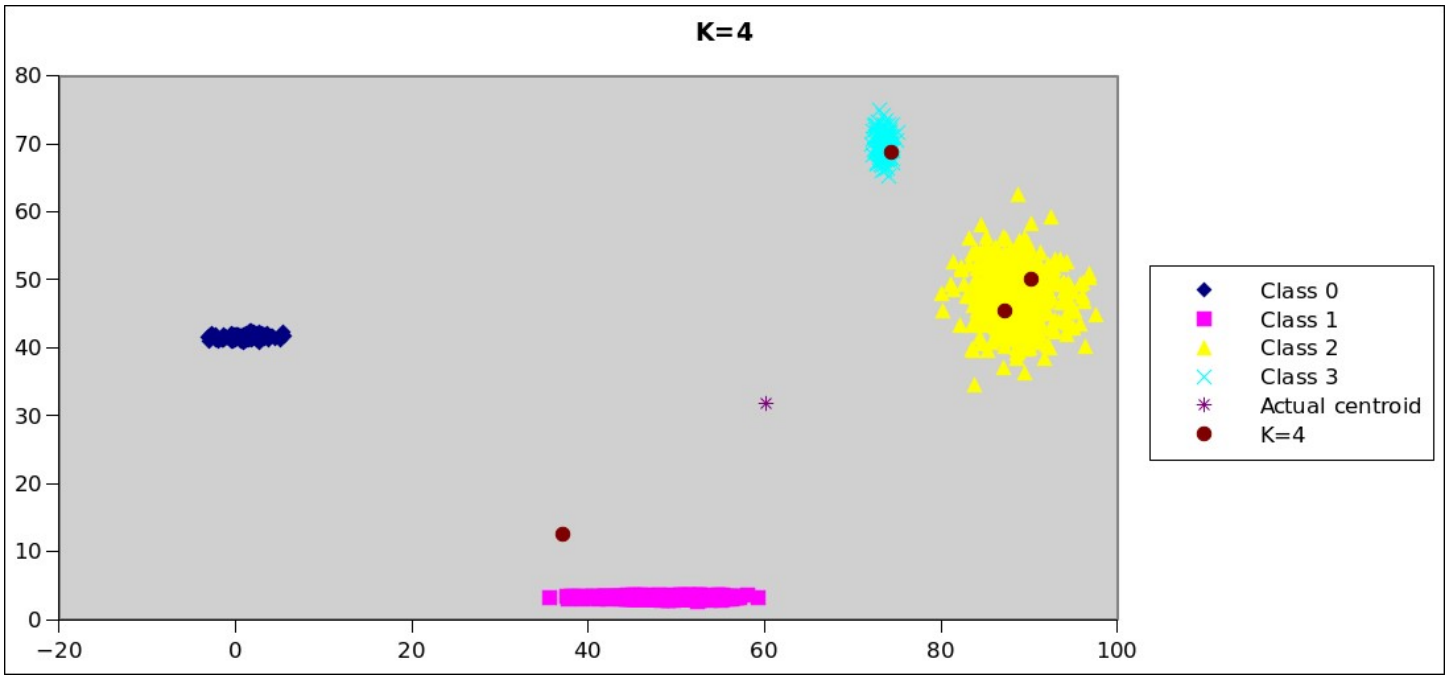
Identificativo del task: *2d-4c*



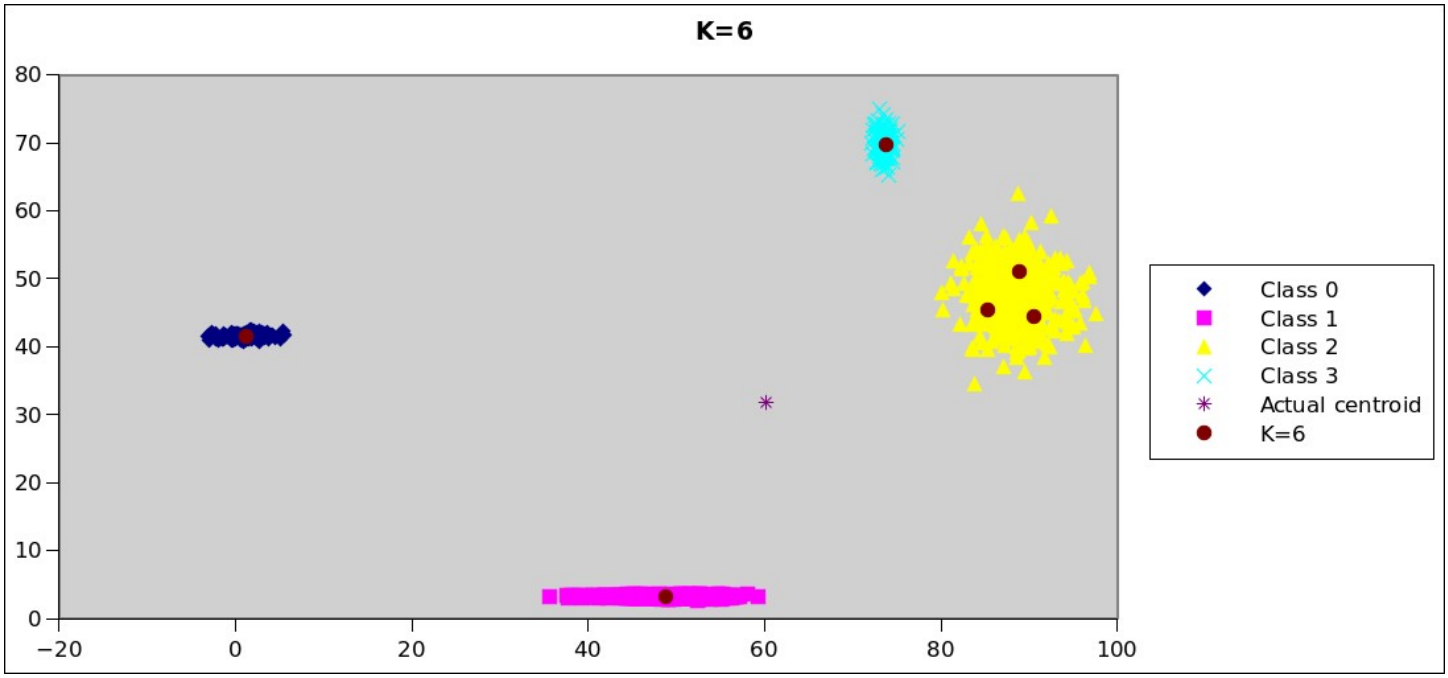
Come atteso il centroide calcolato a mano con il foglio di calcolo coincide con il risultato del clustering con k = 1.



Si nota come un centroide (inizialmente sono meotoidi) sia capitato nella Classe 1 ed uno nella Classe 2, successivamente essi sono stati “attratti” rispettivamente dal Cluster 0 e dal Cluster 3 (perché risultavano i più vicini ad essi).



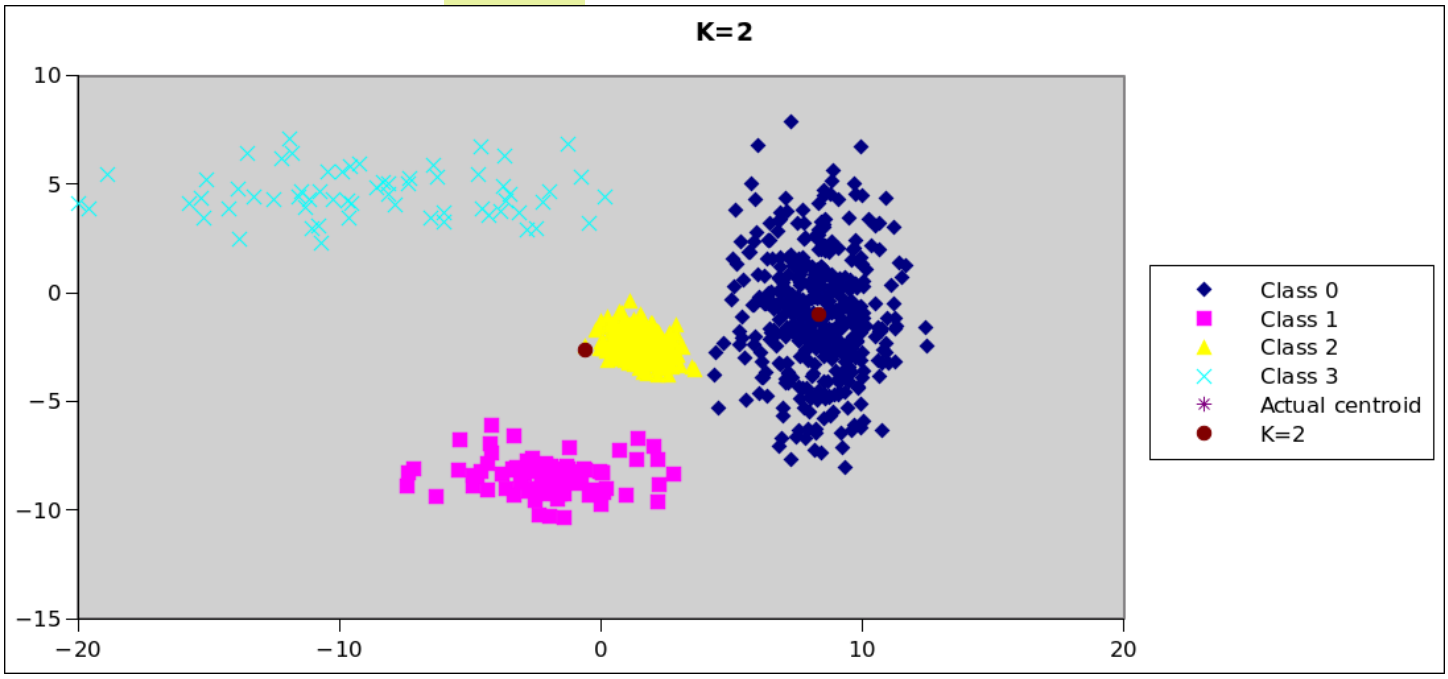
Quì si inizia a capire quanto, in effetti, la scelta dei centroidi iniziali sia cruciale per il risultato finale. Vista la lontananza tra le classi, centroidi rimangono “ancorati” alle classi iniziali.

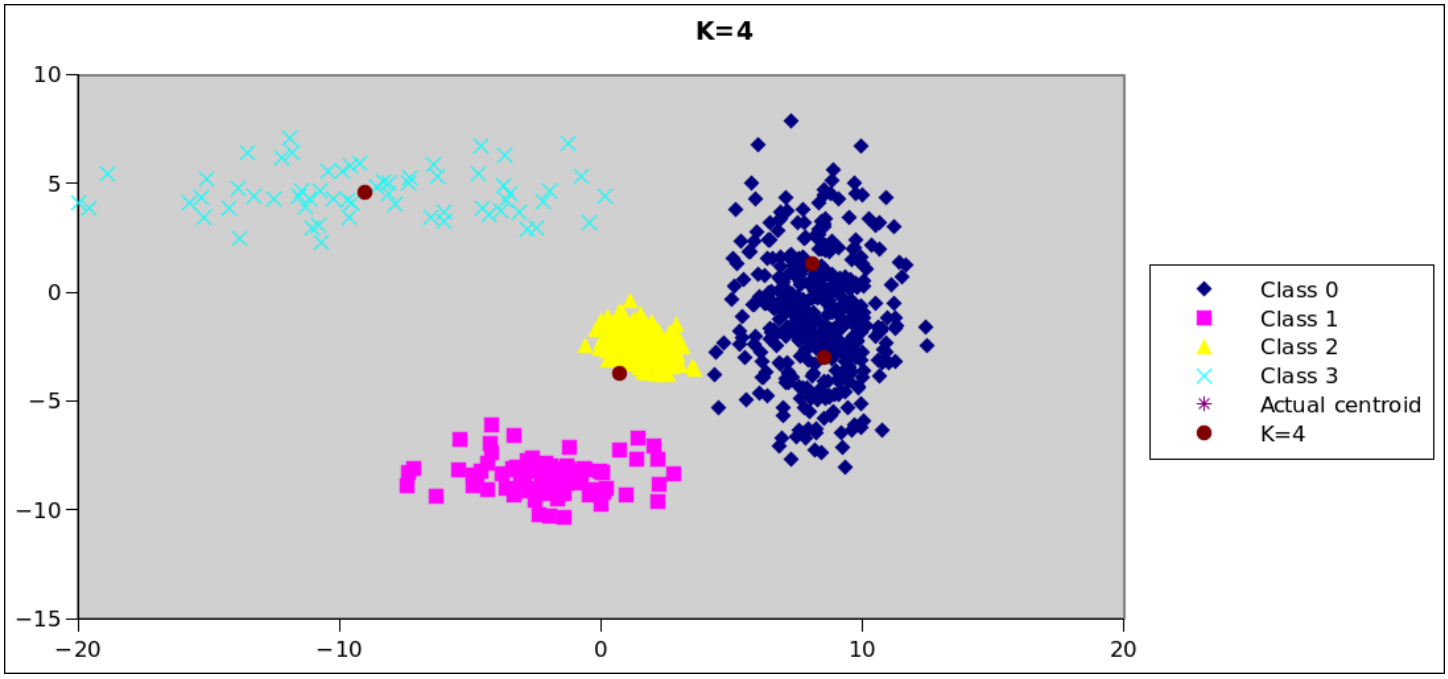


Test aggiuntivo, non troppo significativo.

3. 2 dimensioni e 4 classi, versione 2

Identificativo del task: *2d-4c-v2*

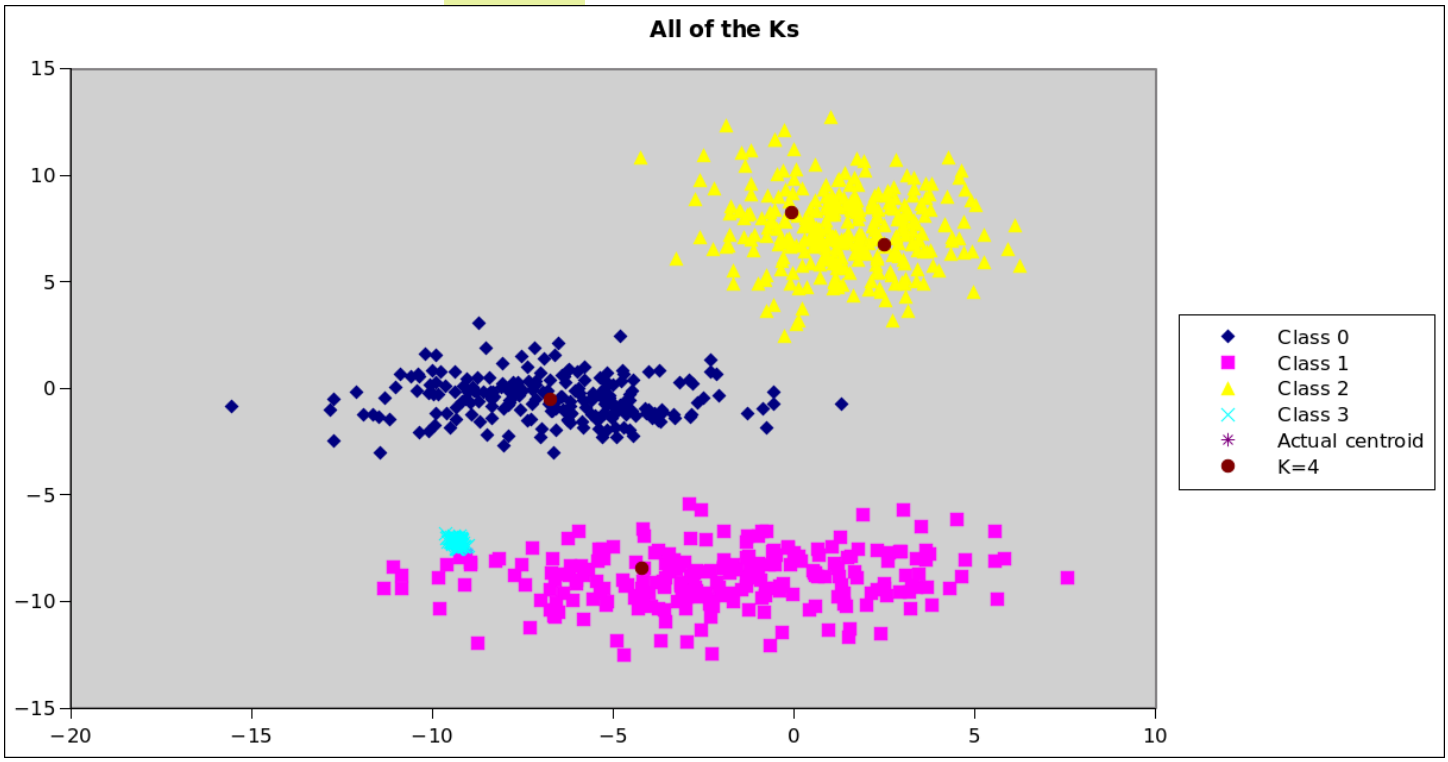




Anche se la distanza tra i cluster è inferiore, abbiamo gli stessi comportamenti del caso precedente.

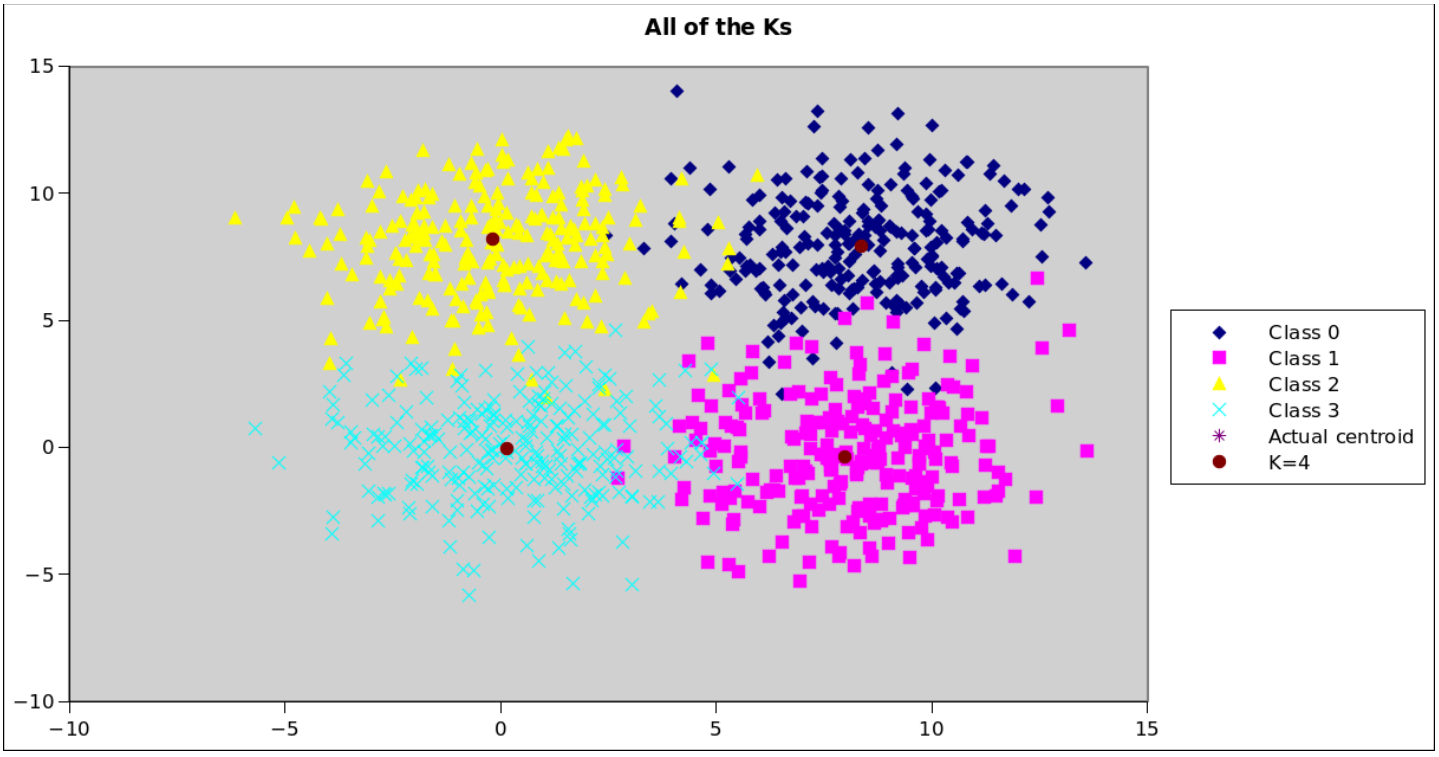
4. 2 dimensioni e 4 classi, versione 3

Identificativo del task: `2d-4c-v3`



5. 2 dimensioni e 4 classi, versione 4

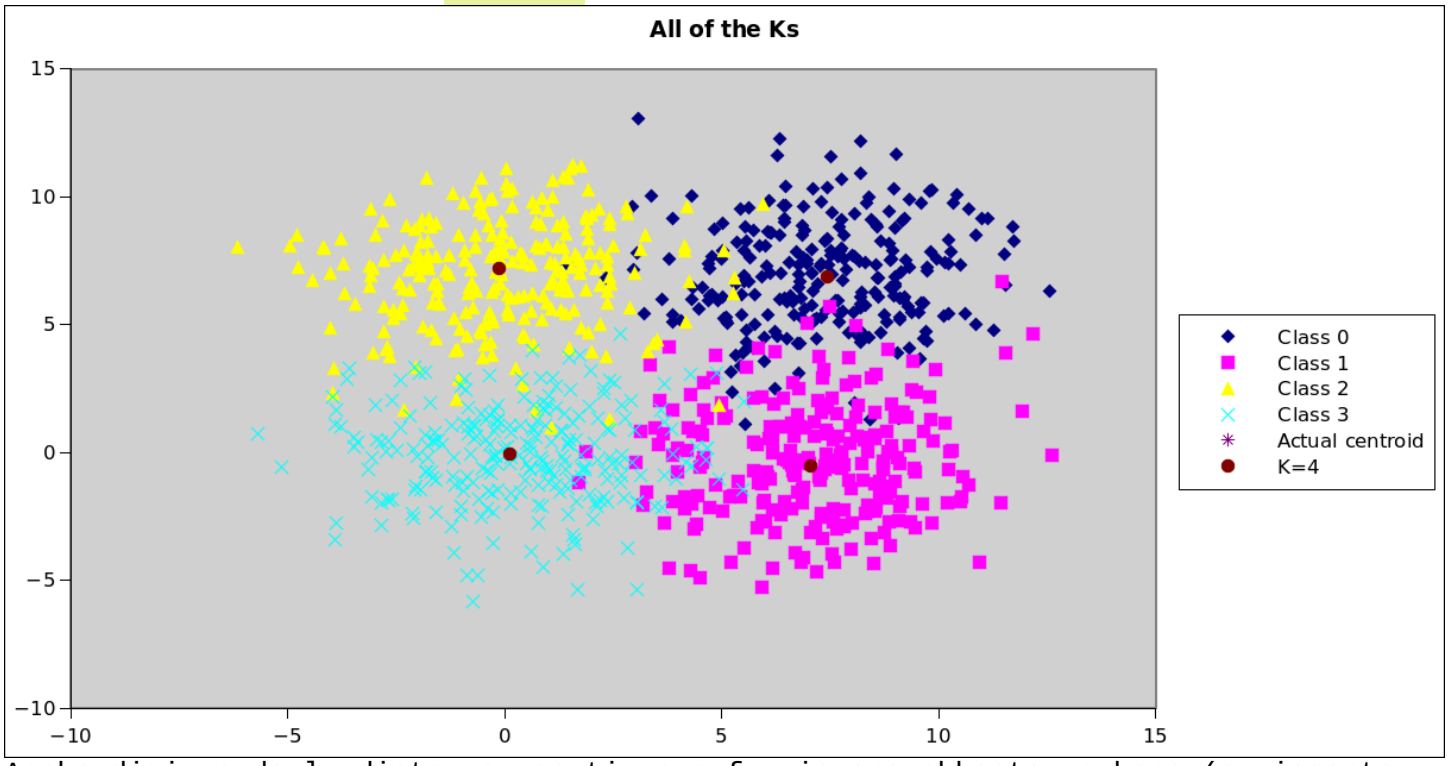
Identificativo del task: `2d-4c-v4`



In questo caso l'algoritmo funziona bene perché le classi sono abbastanza vicine (ed hanno forme circolari).

6. 2 dimensioni e 4 classi, versione 5

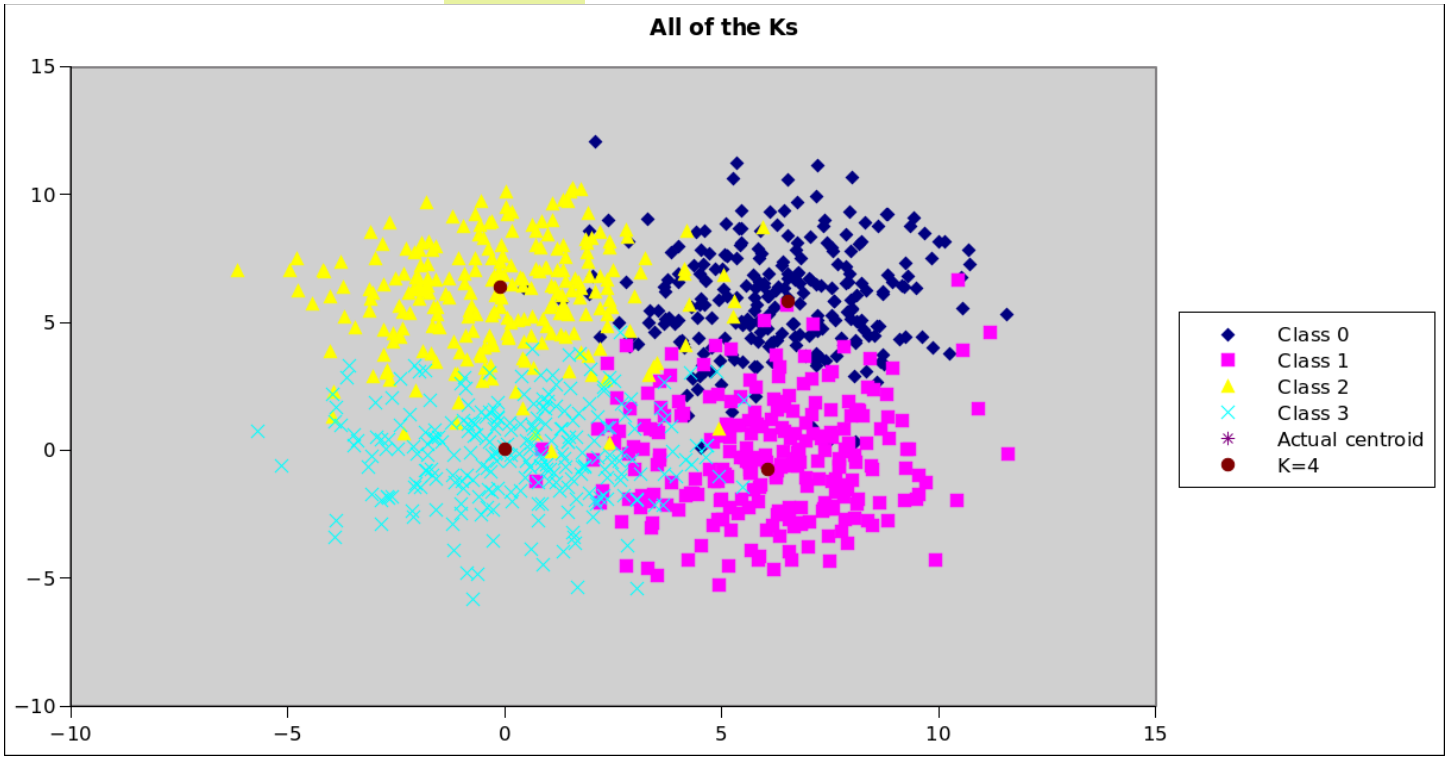
Identificativo del task: *2d-4c-v5*



Anche diminuendo la distanza, continua a funzionare abbastanza bene (ovviamente gli outlier comportano errori..)

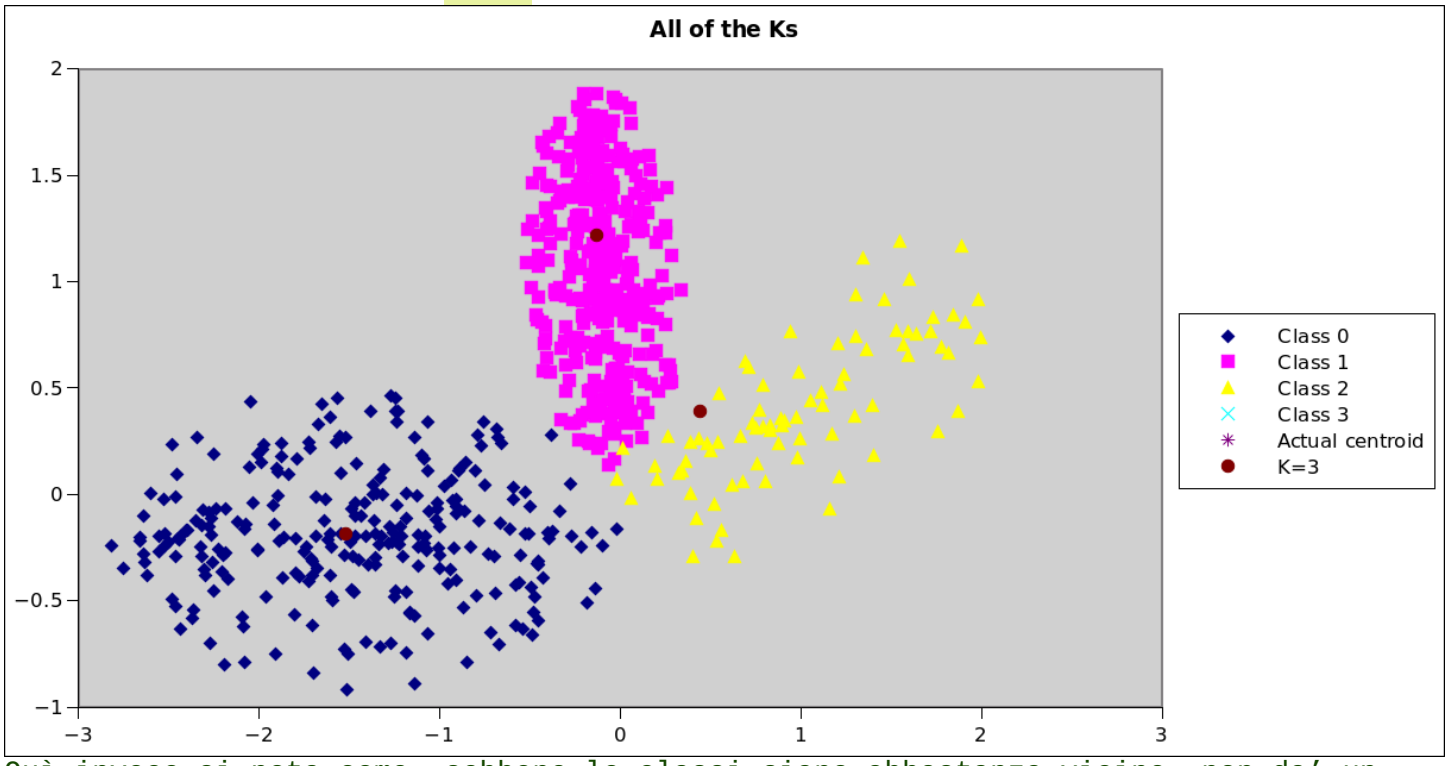
7. 2 dimensioni e 4 classi, versione 6

Identificativo del task: *2d-4c-v6*



8. 2 dimensioni e 3 classi

Identificativo del task: *2d-3c*



Qui invece si nota come, sebbene le classi siano abbastanza vicine, non da' un buon risultato a causa proprio della forma non circolare delle classi.

8. 2 dimensioni e 20 classi

Identificativo del task: *2d-20c*