Project in Practical Machine Learning

Radar image clustering

Marko Hassinen
014029585

Helsinki 20.03.2016

University of Helsinki
Department of Computer Science

Terms of Reference

A report submitted in fulfilment of the requirements for Course 582739 Project in Practical Machine learning, Deparment of Computer Scinece, University of Helsinki.

Abstract

Weather Radar image clustering is the topic of this project. This report explains which tools and methods were used in the project. Report also investigates the results of the project.

Table of Contents

Introduction

The goal of this project is to find clusters within weather radar images and find these clusters with different clustering methods. Also an additional goal is to compare the results of different clustering methods and evaluate the performance of the clustering methods. Most of the clustering methods are predictive clustering methods. This means that the aim of the clustering is to predict possible clusters rather than observe the structure and relation within the dataset. Clustering relates to machine learning because clustering is considered to be part of unsupervised machine learning. Unsupervised machine learning differs from supervised machine learning such that no labels are attached to the data by user. The results of this project were gathered around late winter and early spring of 2016.

Methods

Tools used in this project were programming language Python 3.4.3 (lähde), web application framework Django (version + lähde), Platform-as-a-Service(viite) provider Heroku(lähde).

Data Collection

Data analysed in the project was provided by Finnish Meteorological Institute (FMI) and retrived from FMIs Web Map Service. FMI has 10 different weather radars located around Finland which provide the data for WMS. WMS can also merge all of the weather radars data into a single image covering almost all of Finland. Only single radar images were used in the project, since they all shared the same shape. Every radar also has four different measures, radar reflectivity (dbzh), radial velocity (vrad), rain classification (hclass) and cloud top height (etop_20).

Modules

A module *datagathering-py* was implemented to connect to FMIs WMS server. This module is able to retrieve 40 different weather radar images and store them on disk. *datagathering.py* is module that is able to pull all of the different kinds of images from every FMIs weather radar in real time. The initial resolution of the images was 720x480 (480p). Image resolution had different options, but 720x480 because it's a stardard size. Image resolution affects greatly the clustering algorithms performance. Also larger image resolution was considered, but the algorithm runtime on a local machine suggested that smaller resolution would be better for further cluster and algorithm analysis. Because of the poor performance with 480p images, a smaller resolution was chosen of size 300x250 and new dataset was gathered from FMI. New resolution was almost half of the size of the original resolution and thus the algorithm runtime improved greatly. Faster algorithm performance with smaller resolution means also loss of detail in images so it's a trade-off between image detail and clustering algorithm runtime.

The File *preprocessor.py* is used to read images into numpy arrays. The File defines a class *preprocessor*, which has three local attributes *data, amount* and *path*. *Data* datatype is pythons built-in

list, because with large amounts(i.e. >1000) of images *data*s conversion into numpy array would take a lot of time. *Amount* holds the information about the amount of images read with *preprocessor*. Directory path to the images read by *preprocessor* has been saved into *path*. *preprocessor* has three public functions and one local. The local function is called *read_data()* and it has no parameters, but the convetional *self* parameter. It uses the module *os* to read traverse through the filesystem. The function also assumes that the files within the root are images, which the module *matplotlib.image* can read. *matplotlib.image.imread* returns *numpy* arrays with the shape (480, 720, 3) in this case. r*ead_data()* appends the class attribute *data* and returns it. Functions *count_data()* and *get_path()* work as typical getter-functions(viite). *get_data()* has one paratemeter *path*, which is a string that represents the directory path to the data. *get_data()* uses *read_data()* to get the data and stores *path, amount* and the *data* itself. If *get_data()* is called twice with same *path*, then *read_data()* is called only once. *get_data()* returns a four dimensional matrix, which contains all the images as pixel arrays. *get_data_as_2d()* has also one parameter *path* and it uses *get_data()*, but returns data as two dimensional matrix instead of four, where all single images have been reshaped into one dimension with *numpys* rehape function.

All clustering related code has been encapsulated in *my_models.py* file. The file has a class *my_models* that includes every machine learning algorithm used in the project. K-means (Flach, 2012), k-metoids (Flach, 2012) algorithms were selected for cluster analysis and they are implemented as functions inside *my_models*. K-means and k-metoids require initial cluster kernels and local function *randomly_init_cluster_kernels()* does exactly that. It has two parameters *data* and *k,* where k is the desired amount of clusters. *randomly_init_cluster_kernels()* uses *numpy.random.randint()* to get a random value between 0 and n, where n is the amount of data. Then it stores the random index(key) and coresponding pixel array and repeats this k times. Finally it returns two *numpy.array*s first one containing new random kernels and second one containing their keys. *clusterify()* is also local function that returns a dictionary containing every single pixel arrays index. Dictionarys keys values represent the indexes of kernel array. *Clusterify()* is a bottleneck, because it's running time is one of the worst of the components of k-means and -metoids O(n*k*O(dist_2)). This means that by improving *clusterify()* k-means and -metoids running time would improve.

Machine learning algorithms

All of the machine learning algorithms have been implemented into the file *my_models.py.* Module also contains some functions that some of the machine learning algorithms share for example *randomly_init_cluster_kernels().*

K-means (Flach, 2012) is very well known predictive clustering algorithm. Projects version follows heavily the pseudo-code in (Flasch, 2012). Initial resolution of data files also drove towards some optimization in terms of calculations and storage of raw data.

K-means++(Arthur & Vassilvitskii, 2007) is an improved version of k-means. K-means++ works exactly the same as k-means, but it has improved way to select initial cluster centers.

k-medoids (Flach, 2012)

fuzzy c-means

k-modes

k-prototypes

hierarchical clustering

Results

I made an experiment with image size (480x720x3), amount of images n = 1142, amount of clusters k = 10 and the averidge running time for clusterify was 69 seconds. So if k-means runs at least 69 seconds and converges after 100 rounds, it would require at least 6900 seconds approx 2 hours to complete. After this experiment I decided to reduce the number of clusters to 4 and then clusterify ran 27 seconds. Also I decided to run k-means with less data (n<1000).

Discussion

Conclusion

Appendices

References

Arthur, D. and Vassilvitskii, S. (2007). k-means++ : The Advantages of Careful Seeding. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp.1027-1035.

Flach, P. (2012). *Machine learning*. Cambridge: Cambridge University Press.

Project repository:

https://github.com/mihassin/ubiquitous-spoon

Project schedule:

https://docs.google.com/spreadsheets/d/1SMEjF9h94tKEg0vpC7j1yXE0jVpetmsGcNhnBnqYh8M/edit#gid=0

Hour accounting chart:

https://docs.google.com/spreadsheets/d/1SMEjF9h94tKEg0vpC7j1yXE0jVpetmsGcNhnBnqYh8M/edit

#gid=2095313416.