




University of  
Stavanger

Faculty of Science and Technology

## MASTER'S THESIS

Study program/Specialization:  Petroleum Engineering / Natural Gas Technology	Spring semester, 2016  Open
Writer: Lars Kartevoll	 ..... (Writer's signature)
Faculty supervisor: Rune W. Time	
Thesis title:  Using machine learning to identify flow regimes from capacitance sensor data	
Credits (ECTS): 30	
Key words:  Multi phase flow Flow regime identification Machine Learning k-means algorithm Neural network	Pages: 41  + enclosure: 20 + program on USB  Stavanger, 29.06.2016

Using machine learning to identify flow regimes  
from capacitance sensor data

University of Stavanger



Lars Kartevoll

June 2016

# Using machine learning to identify flow regimes from capacitance sensor data

Lars Kartevoll

## Abstract

In this thesis the k-means clustering and a neural network is developed and used to classify capacitance data from multi phase flow in a horizontal tube.

Theoretical background for the unsupervised machine learning algorithm: k-means clustering and for the supervised machine learning algorithm: Neural network with one hidden layer is presented. Data acquisition method and analysis of the multi-phase flow data is discussed. The machine learning algorithms are created in Matlab in a general manner so that the programs will work for input of varying sizes. The k-means algorithm is used as a method for clustering provided data examples in flow regime clusters. The algorithm fails to provide rigid clusters which match observations at phase transitions, but works well as a general indicator of flow regime clusters. Classifications from the k-means algorithm and a set of manual classifications is used as input in the neural network for training and testing. The neural network provides overall good results, and shows its ability to detect complex patterns.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Thesis statement . . . . .	1
1.2	Approach and comments . . . . .	2
<b>2</b>	<b>Theory</b>	<b>4</b>
2.1	Flow regimes and identification . . . . .	4
2.2	Capacitance sensors and data . . . . .	7
2.3	Machine learning . . . . .	11
2.3.1	Unsupervised learning: k-means clustering . . . . .	11
2.3.2	Supervised learning: Neural network . . . . .	12
<b>3</b>	<b>Method</b>	<b>21</b>
3.1	Extracting and plotting information . . . . .	21
3.2	k-means program . . . . .	25
3.2.1	kmain.m . . . . .	25
3.2.2	randomInit.m and runkmeans.m . . . . .	26
3.2.3	assignClosestCentroid.m and computeCentroids.m . . . . .	26
3.2.4	computedist.m and plotKmeans.m . . . . .	26
3.3	Neural network . . . . .	27
3.3.1	NNmain.m . . . . .	28
3.3.2	nnRandInit.m and randInitializeWeights.m . . . . .	29
3.3.3	nnCostFunction.m and Predict.m . . . . .	29
3.3.4	plotNN.m . . . . .	30
<b>4</b>	<b>Results</b>	<b>32</b>
4.1	Clustering with k-means . . . . .	32
4.2	Assigning regimes to examples . . . . .	34
4.3	Running the neural network . . . . .	36
<b>5</b>	<b>Conclusion</b>	<b>40</b>

<b>Bibliography</b>	<b>42</b>
<b>Nomenclature</b>	<b>45</b>
<b>Appendix</b>	<b>1</b>
<b>A k-means Matlab code</b>	<b>1</b>
A.1 kmain.m . . . . .	1
A.2 randomInit.m . . . . .	3
A.3 runkmeans.m . . . . .	3
A.4 assignClosestCentroid.m . . . . .	4
A.5 computeCentroids.m . . . . .	5
A.6 computedist.m . . . . .	5
A.7 plotKmeans.m . . . . .	6
<b>B Neural network Matlab code</b>	<b>8</b>
B.1 NNmain.m . . . . .	8
B.2 nnRandInit.m . . . . .	11
B.3 randInitializeWeights.m . . . . .	11
B.4 nnCostFunction.m . . . . .	12
B.5 sigmoid.m . . . . .	14
B.6 sigmoidGradient.m . . . . .	14
B.7 plotNN.m . . . . .	14
<b>C Plotting data from capacitance sensors</b>	<b>16</b>

# Chapter 1

## Introduction

Machine learning is the science of getting computers to learn, without being explicitly programmed [10]. The process of machine learning is used more and more around us, and surrounds us in our daily lives. Everything from spam filters, to deep space analysis. Machine learning algorithms are capable of doing a lot of complex tasks. Multi phase flow is a study of the flow regimes that occur when matter of different phases flow together. The flow regimes that occur are of a complex nature, and should be a great target for machine learning.

This thesis started with a provided set of data, which was collected many years ago. The capacitance sensor data is a measurement of the flow regimes on a horizontal tube. The data was collected by Time [24] and Eeg [3]. This data would serve as the basis for the machine learning application.

Although machine learning is everywhere around us, I had very little knowledge about the subject. More information was needed about methods. Choosing a good programming environment important. Finding methods which would be able to provide correct regime classifications. The programs should also be made in such a way that they can be utilized later by myself or other students. These questions lead to the following thesis statement.

### 1.1 Thesis statement

These are the main points that were set at the beginning and throughout the working process of this thesis,

- Do a literature study of machine learning used for flow regime identification
- Look for additional datasets comparable to the provided datasets

- Create a machine learning program for classifying flow regimes
- The machine learning program should be of a general nature, so that they can be utilized later or for other projects.
- Run data from the time series through the machine learning algorithms
- Use different input information to find a good flow regime identifier

## 1.2 Approach and comments

From the beginning, literature search was focused on machine learning utilized with flow regime identification. This yielded a handful of papers which had used different types of data analysis before inserting it into the machine learning algorithms. The papers were often relatively short in explaining how to build a machine learning algorithm, as this was not their main purpose.

This quickly led to the discovery that a more general search for machine learning algorithms, and how to build one was required. This opened up a vast jungle of: Different types of machine learning, different programs used for implementation, different explanations of machine learning principles, and many, many diffuse articles about machine learning mechanics and implementations. A lot of time was spent reading articles written above my understanding of programming and being left with very little new knowledge. In the end I came across a MOOC (Massive Open Online Course) on Coursera.org called "Machine Learning". This course was an inspiration, and made this thesis possible. This MOOC was completed and its teachings were utilized when applying machine learning. Some of the sources used from the Machine Learning courses wiki-page are not available online, without signing in. Screen-shots of these web-pages have been included with the digital content of this thesis after clarifying with Coursera via email.

Searching for data similar to the data provided was also time consuming. There were some articles and studies which seemed promising, but in the end, the time series were not available. This seemed to be the case for the most of available articles. I was also in contact with institutions which perform measurements on multiphase flow, but email contact was very slow. In the end, no additional data was acquired.

Two approaches were chosen as methods to classify the flow regimes from the data files. One unsupervised machine learning method, the k-means method. The

other, a supervised machine learning method, the neural network. The unsupervised method does not require a classification of the sensor data samples as input, and can be used to try and classify the data flow regimes given the sensor data. The supervised method does, however require a classification of the data samples. This is because a neural network uses the classification of the datasets to train itself to recognize patterns in the input data. For a long time it was believed that the data from Time [24] could be easily identified using an available diagram, but this turned out to be wrong. The only classified data was therefor from Eeg [3].

The neural network therefore got two sets of classified input. This mostly just as a test that the neural network works, since there was no absolute or visual classification of the examples available. One set was taken from the k-means algorithm. The other was based on the observations done by Eeg [3] and some manual interpretation.



# Chapter 2

## Theory

The main theory surrounding the subjects is presented in this chapter. Many subjects will be touched upon, and some will be examined more deeply than others.

Firstly flow regimes and identification will be discussed. The factors which define, and which can be used to determine a flow regime are many. The three main areas used in this thesis are density functions, spectral analysis, and cross correlation.

A quick look at the capacitance sensor, their setup, and the data output from the sensors will also be presented. An understanding of the data is essential for interpretation of the output related to the input in the machine learning part.

Machine learning will be separated into two main parts. The first will be unsupervised learning. Here the k-means algorithm will be discussed and used as a method for clustering the examples. The data inserted into the algorithm are features gathered from the data sets. The other method is supervised learning. Here the features of an example will be grouped with a classifier and inserted into a neural network. The network will then train based on the features and classifiers and produce a hypothesis for predicting a classification based on only features.

### 2.1 Flow regimes and identification

A lot of study has been done in identifying flow regimes. One of the most known regime maps for horizontal flow is probably the theoretical and experiment based maps from Taitel and Dukler [22], and Mandhane [8]. Their models have been used for many years, and the regime maps are a function of the superficial flow velocity.

$$U_{LS} = \frac{q_L}{A} \tag{2.1}$$

$$U_{GS} = \frac{q_G}{A} \tag{2.2}$$

Here  $U_{LS}$  and  $U_{GS}$  are the superficial velocities of liquid and gas respectively.  $A$  is the cross-sectional area of the tube.  $q_L$  and  $q_G$  are the volumetric flow rates of liquid and gas respectively.

The model of plotting the flow regime based on the superficial velocities is well known. The problem with the model is however that it is dependant on the inner diameter of the tube. As the cross section of the tube is dependant on the diameter. Of course also other factors which would affect the fluids in the tube will also distort the regime maps. The flow regime maps done by Taitel and Dukler, and Mandhane are therefor not universal [8] [22]. An example of the Taitel-Dukler model can be seen in figure 2.1.

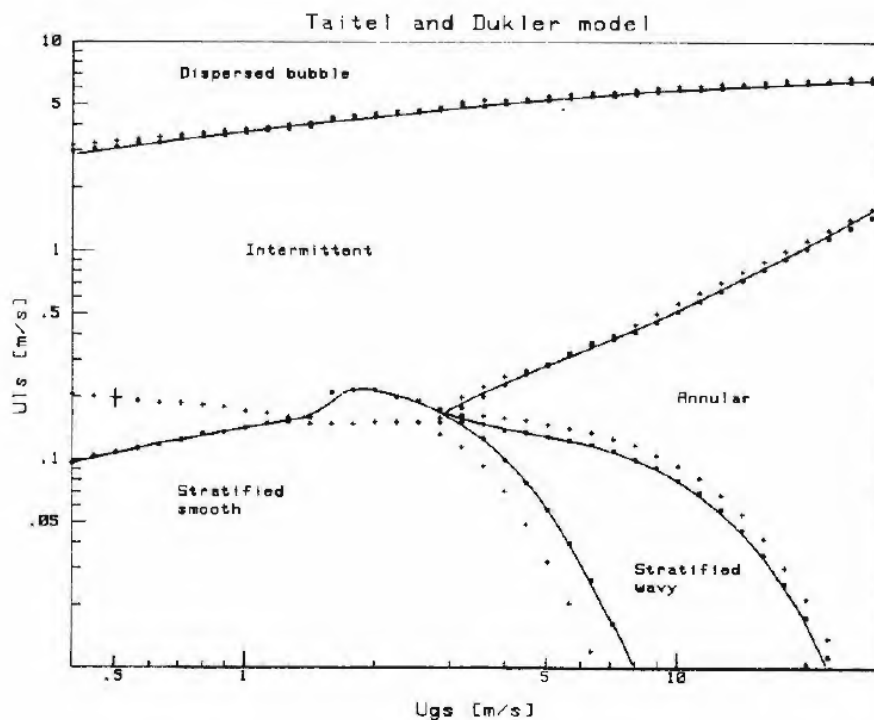


Figure 2.1: An example of a Taitel Dukler model done by Time [24]. The model is based on a tube diameter of 4 centimetre.

Well known methods for extracting information from time series for identification purposes are: Probability density models, and frequency analysis. There are three probability density models applied in this thesis. The standard probability density function, PDF, the cumulative density function, CDF, and the probability distribution function PDSF [6]. The probability density function applied to time series shows the probability of a value being measured.

The discrete version of this is to count the number of times a number, or a number in a set interval or "bin" occurs in a time series. For multi phase flow the PDF will vary depending of the type of regime in the tube.

The CDF is very similar to the PDF. Instead of a bin only holding the value for the given interval, it holds the sum of the given interval and all before it. The CDF also rescales the function value so that it goes from 0 to 1. A measurement with mostly low values, like annular or dispersed bubble will reach 1 quickly while a measurement with high values will reach 1 slowly.

The PDSF or probability distribution function, as it is named by Lee. et al [6], is all the measurements sorted by size. The features of the PDSF have also been rescaled by dividing it by 10 000. This way it is in the same size scale as the CDF. The main reason for using CDF and PDSF over PDF is that the discrete version of the PDF can seem quite "choppy" and uneven, as seen in figure 3.3. This can lead to the neural network having problems with detecting characteristics.

Spectral analysis of the sensor data can be done using the Fourier Transform. The fourier transform converts the signal from the time-domain to the frequency-domain. [2] This makes it possible to detect the peak frequencies in the flow regimes. Some flow regimes can be distinguished by their peak frequency, like slug and stratified wavy. Dispersed and annular however share a similar fourier transform output [23].

The last data output used in this thesis comes from the use of the cross correlation function on a sensor pair. The signals are compared by taking the product of each signal. One of the sensor signals is then displaced by one element, and the function value is calculated again. This is done for the entire length of the sensor vector. The plotted function values will then have a peak where the two sensors' signal values are most like each other. The number of element steps to reach the peak is then correlated with the scan delay of the sensors, and the slug velocity is calculated [1]. The difference in slug velocity from the three sensor pairs may serve as a characteristic for flow regime in the neural network. An example of the cross correlation principle can be seen in figure 2.2

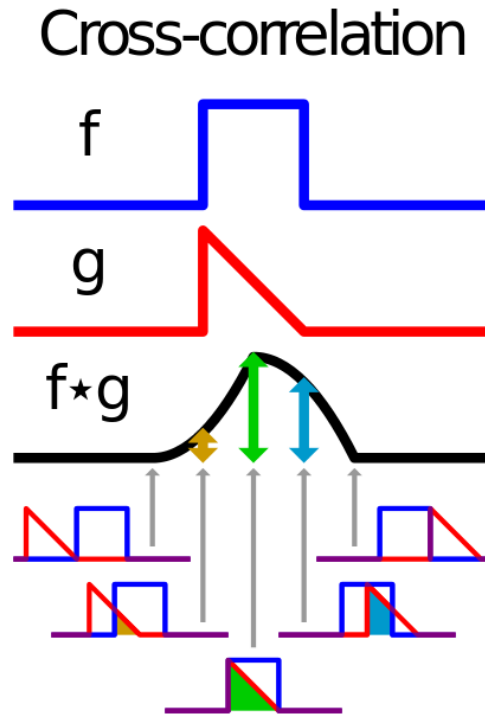


Figure 2.2: The figure illustrates the concept of the cross correlation function. [18]

## 2.2 Capacitance sensors and data

The data used in this project was collected in 1991 - 1992 and was collected by Rune W. Time and Ole Eeg for their doctoral thesis [24] and master's thesis [3] respectively. The sensors themselves were built and developed by Time. The capacitance sensors are set up in three pairs. The sensors are numbered from 1 to 6, and are paired according to capacitor orientation. The pairs are: pair 1: Sensor 1 and 6, pair 2: Sensor 2 and 5 and pair 3: Sensor 3 and 4. The sensors were mounted on the outside of a horizontal tube which was part of a flow loop. The flow passed through the sensors in the direction from sensor 6 to sensor 1. The sensor pairs are different from each other. The first sensor pair is mounted horizontally, the second pair is mounted vertically. The third pair has one large capacitance sensor on the bottom, and one small at the top of the pipe to ensure great sensitivity at the top. An illustration of the sensor pair orientations can be seen in figure 2.3

The capacitance sensors are connected to a voltage source which alternated between negative and positive. At first the supply imposes a positive voltage, and the capacitors charge. When the capacitors reach a set charge, the voltage is switched to negative, and the capacitors discharge, before charging again. The time it takes for the capacitors to reach the amount of charge where the voltage swaps is propor-

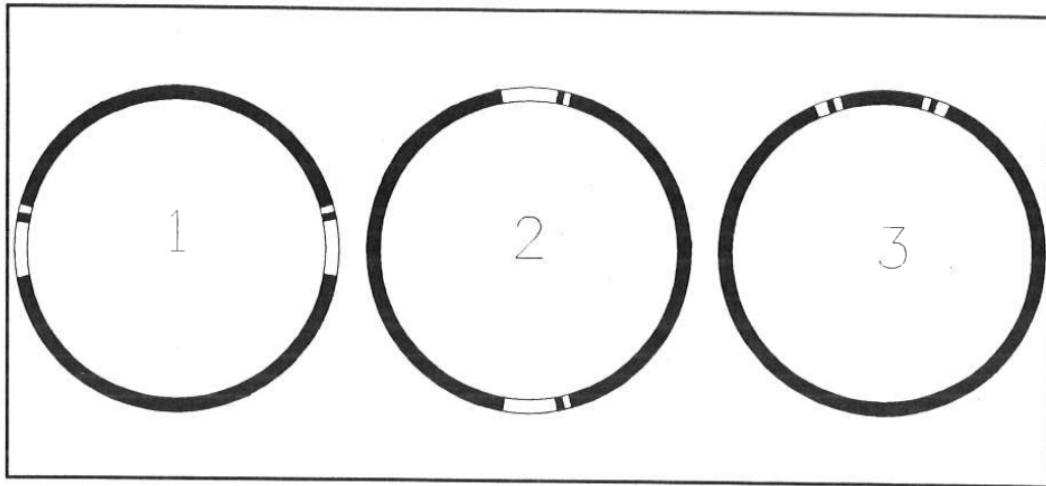


Figure 2.3: An illustration of the three orientations of the sensor pairs done by Eeg. [3]. The sensor pairs will have different sensitivities in different parts of the tube.

tional to the capacitance. Because of this the frequency of the voltage becomes a expression for the capacitance. [24] [3]. The voltage signal is then sent to a frequency converter which transfers its signal to the data acquisition device.

The data acquisition device then outputs the data through a program made by Time, and outputs a standardized output for all of the sensors, ranging from 0 to 10000. As seen in the figure, a low number represents a low liquid height, and a high number represents a high liquid height. The standardized signal and the representative liquid height in the tube can be seen in figure 2.4. It shows very well that the sensors have different sensitivity-areas in the tube cross section.

The data files output from the program contain a lot of data. The first 40 lines are strings of data, many not viable to this project, and some not used by Time at the time of recording [24]. The data used from the comment section of each file are the superficial velocities from line 32 and 34, and the scan delay for the recorded run from line 4. Figure 2.5 show a crop of the top of a datafile. The capacitance sensor output is listed in columns numbered by sensor number V1 to V6. Each file holds a total of 5000 capacitance measurements per sensor, and the recording time is calculated from the scan delay.

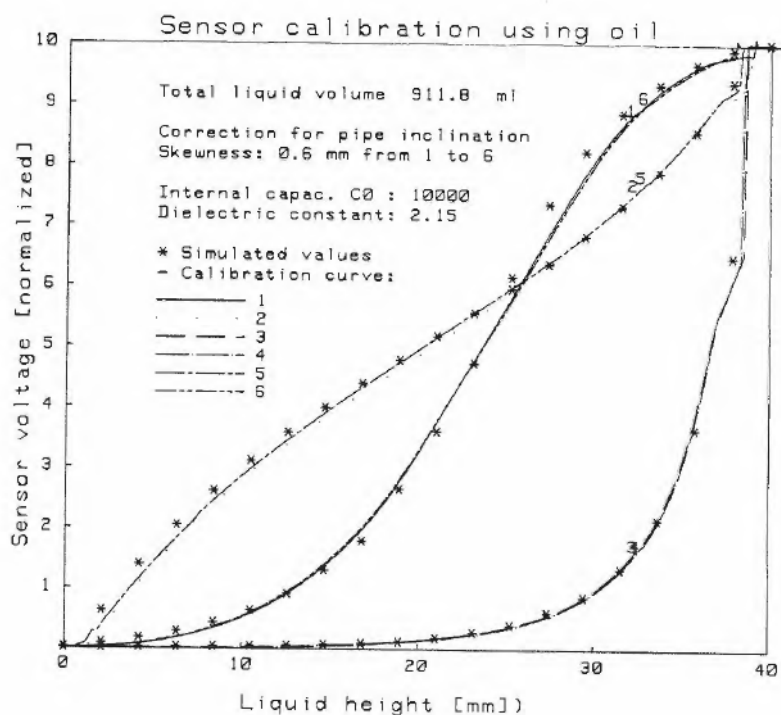


Figure 2.4: A plot showing the sensor calibration with oil from Time [24]. The sensor pairs behave differently to different liquid heights, because of their respective sensitive areas in the pipe.

```

1      DATE:  3 Oct 1991 13:44:44
2      Visual flow regime:DB - GAS AT TOP,INHOMGS
3      Sampling speed      : 1.E-6
4      Scan delay         : .002
5      Sonic nozzles disabled
6      Gas flowmeter is used
7      Reference frequency (Hz - read from oscilloscope): 33.0
8      Reference Vp-p     (Volts - read from oscilloscope): 2.36
9      Uls= .516576507971 m/s  Ugs= 2.46438672486 m/s
10     Rho_l= 782.798299705 kg/m^3  Rho_g= 1.29526410967 kg/m^3
11     My_l= .00266311614173 Pa.s  My_g= 1.84069168307E-5 Pa.s
12     Inlet pressure= 1.0900511811 Pressure at sensors= 1.04748228346
13     Gas temp. at nozzles: 21.9119094488
14     Oil temp. at inlet  : 21.6289370079
15     Superficial gas velocity at inlet  2.47647321318 [m/s]
16     Superficial gas velocity at sensor  2.57711523489 [m/s]
17
18
19
20     ORIG. FILE =NYMAL05_24
SENSOR:  1      VMAX:  6.12778  +/- .0021  VMIN:  1.72437  +/- .0022
SENSOR:  2      VMAX:  6.09562  +/- .0015  VMIN:  1.26536  +/- .0017
SENSOR:  3      VMAX:  6.32835  +/- .006   VMIN:  .9552   +/- .0019
SENSOR:  4      VMAX:  6.64075  +/- .0023  VMIN:  1.35181  +/- .0027
SENSOR:  5      VMAX:  6.52246  +/- .0019  VMIN:  1.64895  +/- .0023
SENSOR:  6      VMAX:  6.32951  +/- .0019  VMIN:  1.55071  +/- .0023
Time between measurements .002
Oscilloscope readings:
Frekvens:
Vpp :

ULS : .516576507971
UGS : 0
UGS at capacitance sensors: 2.57711523489
Oil density: 0
Gas density: 0
Oil viscosity: .00266311614173
Gas viscosity: 1.84069168307E-5
Inlet pressure: 1.0900511811
Oil temperature: 21.6289370079
  V1      V2      V3      V4      V5      V6
8682     8539     7505     7153     8748     8871
8733     8544     7518     7092     8764     8871
8767     8560     7481     7068     8723     8913

```

Figure 2.5: Here is a crop of the top part of one of the data files. The data files include many lines of information, for example: Superficial velocities at the sensor and scan delay, before the capacitance log starts.

## 2.3 Machine learning

Machine learning is a science where you want a machine to solve a problem without specifically programming it [10]. There are countless implementations of machine learning surrounding you everywhere at this day of age. Some examples are spam filters, "auto complete" functions, image recognition and self driving cars. The aim will be to use the outputs from the clustering algorithm together with my own intuition to produce the inputs for the neural network.

The formulations used in this chapter are heavily influenced by the "Machine Learning" MOOC from Stanford University, which is available online at Coursera [10].

### 2.3.1 Unsupervised learning: k-means clustering

The first use of the name k-means was by James MacQueen [7] in his paper where he: "*Described a process for partitioning an  $N$ -dimensional population into  $k$  sets on the basis of a sample.*"

A quick explanation of the k-means clustering algorithm is that it takes an unlabeled dataset and groups the data into a predefined set of clusters. The algorithm is an iterative process which consists of two parts. The first part is cluster assignment. The algorithm goes through every input example and assigns it to one of the predefined cluster centroids. The second part is reassigning of centroids. The centroid is moved to the mean point of all its assigned examples [10]. This also is where its name comes from.

The k-means algorithm in its basics only require two types of input:

- $K$  - total numbers of clusters
- A set of examples  $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$

Here  $x^{(i)}$  represents an example vector, and  $m$  is the total number of examples. The first step for the algorithm is the random initialization of the cluster centroids. There are many possible ways of doing this, but one especially has been used for a long time, and is very effective [7]. A number of clusters  $K$  is manually chosen.  $K$  training examples,  $x^{(i)}$ , are then randomly picked and assigned to the cluster centroids  $\{\mu_1, \mu_2, \dots, \mu_K\}$  so that  $\{\mu_1 = x^{(i)}, \mu_2 = x^{(j)}\}$  and so forth.

The next step goes to the inner loop of k-means. The first step in the loop is often called the cluster assignment step. Here each example is assigned to the closest



cluster. The distance used is often the squared distance [10] [11].

$$c^{(i)} = \operatorname{argmin}_k [x^{(i)} - \mu_k]^2 \quad (2.3)$$

Here  $c$  is the index of cluster (1, 2, ... ,  $K$ ) to which the  $x^{(i)}$  example is assigned.

The second step in the inner loop of k-means is often called the move centroid step. Here the average value of points assigned to cluster  $k$  is set as the new cluster centroid.

$$\mu_k = \frac{1}{n} [x^{(k_1)} + x^{(k_2)} + \dots + x^{(k_n)}] \quad (2.4)$$

Here  $n$  is the total number of examples assigned to a cluster. These two steps are iterated until the algorithm converges. Additional iterations will then no longer do anything to the cluster centroid or assignment of examples.

The optimization objective of the algorithm can be defined from the cost function, often called distortion when used for k-means. Calculating the distortion for k-means is a method for comparing the input of the algorithm to the output. For the k-means purpose we define it as the sum of the square distances from the examples to their assigned centroids [10] [11]. The distortion can be expressed as:

$$J(c^{(i)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m [x^{(i)} - \mu_{c^{(i)}}]^2 \quad (2.5)$$

$J$  represents the cost function, or distortion. The objective is then to minimize all the parameters using this distortion function. Or in other word: Find the values in the sets of clusters  $c$ , and the centroids  $\mu$  which will minimize the average distance of every example to the cluster centroid. [11]

Because of the nature of randomized initializations and possible outlying data points, there might be problems with finding local minima when utilizing the k-means algorithm. This can be circumvented by running a number of times, and in the end comparing the computed distortion for each run. For a large number of runs compared to the data samples, a near global maxima will be found [10]. Figure 2.6 shows a simple example of global and local minima and maxima.

### 2.3.2 Supervised learning: Neural network

The neural network is often compared to neurons in your brain, and the ability to mimic the brain was the inspiration for the neural network algorithm. An often cited inspiration comes from psychologist Donald O. Hebb and his postulate about

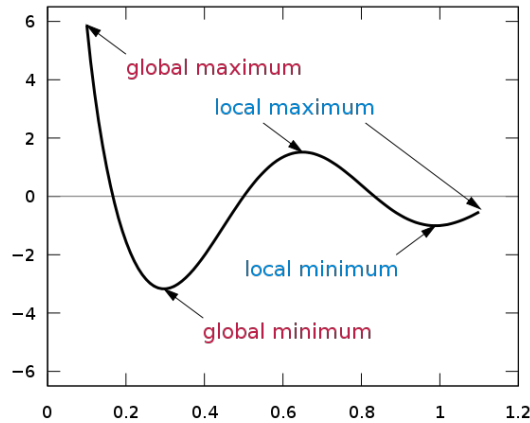


Figure 2.6: A simple illustration of global and local maxima on a graph [20]. Depending on where you start on the graph and travel down the slope, you could end up at either the local or global minima.

a method for learning: *Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability... When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.* [4].

Neural networks are an old idea, which have fallen in and out of popularity for some time, but is now one of the "state of the art" techniques for many machine learning applications [10]. One of the main reasons for the rising popularity in later years is the increase of computational power in modern machines. Now it is possible to run large and complex neural network in a moderate amount of time. Figure 2.7 shows a simple illustration of a neural network with one input layer, one hidden layer and one output layer.

A neural network can learn complex non-linear hypotheses even when the number of features is very large. Before the neural network can be discussed, logistic regression which is a stepping stone to the neural network will be presented.

### **Supervised learning: logistic regression**

Supervised learning has a very simple basis. Given a training set input to the learning algorithm, the algorithm then produces a hypothesis  $h$ . Using this hypothesis, a set of features of the same kind as in the training example can then be input to the hypothesis, and it will output an estimation or prediction.

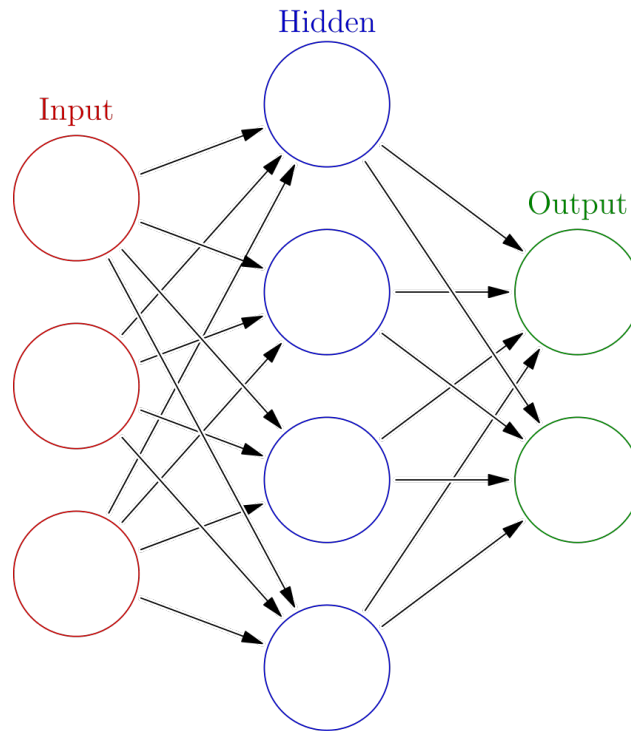


Figure 2.7: An illustration of a neural network [19]. This neural network has three input nodes, one hidden layer with four nodes and two outputs.

The hypothesis can be presented in its basic form for linear regression as:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (2.6)$$

Here  $n$  is the total number of features in an example, and  $\theta_0$  is weight of the "bias unit"  $x_0$  and is equal to one.  $\theta_{(i)}$  is the weight given a feature in an example. As the hypothesis stands now, it is a series of sums. Using matrix multiplication, this can be written as [10] [12]:

$$h_{\theta}(x) = \begin{bmatrix} \theta_0 & \theta_1 & \dots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} = \theta^T x \quad (2.7)$$

Here T indicates the transpose of the theta vector which is a row vector of all the weights. This representation only handles one example and can be further extended

by storing all example and the weights row-wise [10] [12]:

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \quad (2.8)$$

Here there are three examples, each with one bias and one feature and a total of two weights, one accompanying each feature. The hypothesis can then be simplified as:

$$h_{\theta}(X) = X\theta \quad (2.9)$$

The neural network wants to predict a certain identification. In this case determine the type of flow regime. The hypothesis therefore has to be modified into logistic regression. For logistic regression you want to interpret the output as either a "yes" or a "no", a 0 or a 1. This is done by running the output through the sigmoid function, also called the "Logistic Function" [10] [13]. A plot of the sigmoid function can be seen in figure 2.8. Redefining the hypothesis to include the sigmoid function, it becomes:

$$h_{\theta}(x) = g(\theta^T x) \quad (2.10)$$

$$z = \theta^T x \quad (2.11)$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad (2.12)$$

The decision boundary of the hypothesis will now be that all outputs from the sigmoid function larger than or equal to 0.5 will give  $y = 1$  and all outputs smaller than 0.5 will give  $y = 0$ . This handles decisions for only one class. To apply it to classification with multiple possible classes, use the "one vs all" method. When evaluating one category, set all other categories into a separate category. This way, each classification will become a binary classification problem. The hypothesis which then returned the highest result is then chosen as the prediction [13].

The cost function is then applied to the logistic regression. The goal of the cost function for logistic regression in neural networks is to evaluate the hypothesis output and compare it to the classification for that training example. The general version of the cost function for logistic regression can be expressed as:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m Cost(h_{\theta}(x^{(i)}), y^{(i)}) \quad (2.13)$$

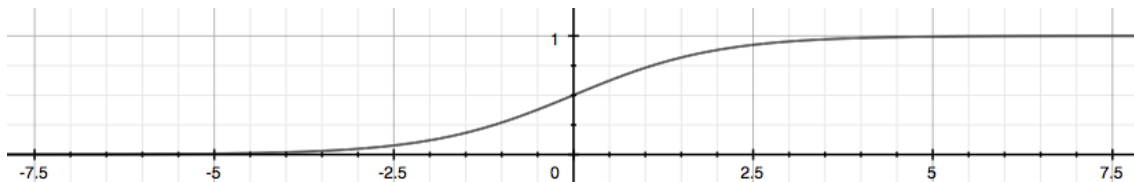


Figure 2.8: A plot of the sigmoid function [25]. The sigmoid function outputs a number between 0 and 1, and works well with translating an arbitrary value output into a classifier.

For logistic regression to avoid the problem of local minima, the cost function is defined differently given the input  $y = 1$  or  $y = 0$  [13] [10].

$$J(\theta) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases} \quad (2.14)$$

Input into one equation which tackles both possibilities simultaneously and is implementable without an if function.

$$J(\theta) = -y \log(h_{\theta}(x)) - (1 - y) \log(1 - h_{\theta}(x)) \quad (2.15)$$

The last addition to the cost function is then to battle the problem of overfitting. Overfitting is when the algorithm fits the available training data too well, but performs poorly on other test data input into the model. There are two easy ways to combat this problem. One is to reduce the number of features, the second is to implement regularization. Regularization is favoured when we have a lot of slightly useful features [16] [10].

Regularization alters the weights of the hypothesis. It smoothes out the hypothesis function as a means to reduce overfitting [9]. In logistic regression, regularization is applied to all weights except for the bias [10]. The cost function with logistic regression applied is [16]:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y \log(h_{\theta}(x)) + (1 - y) \log(1 - h_{\theta}(x))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2 \quad (2.16)$$

The  $\lambda$  in the regularization term is the degree of regularization applied to the function and is manually set. The summation of the regularization term from  $j$  to  $n$  is because the bias unit is not included in the regularization.

The final step of the process is to minimize the cost function by making changes to the weights. One way of doing this, which is easily explained, is using gradient descent which calculates the gradient, or slope, of the cost function and updates

the weights by taking a step in that direction [13]. The weights must be updated simultaneously for each step [10]. A learning curve can be plotted from the cost versus each iteration. When the plotted slope flattens out, further iterations will not achieve a significantly better result.

$$\theta_j = \text{theta}_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad (2.17)$$

Here  $\alpha$  represents the step size for each repeat of the gradient descent. This process requires heavy computing, and will not be utilized. The method utilized is an advanced optimization algorithm of a more advanced nature, and not covered by the thesis. Matlab has available advanced optimization algorithms.

### Neural Network with one hidden layer

Logistic regression will now be extended to the neural network. Instead of having a straight step from input to output via calculation, the neural networks adds a "hidden layer". The name hidden layer is because the values calculated and output through the hidden layer are not shown during the calculation. The hidden layer has randomly generated weights and because of this has the ability to detect non-linear characteristics in the input data.

When talking about neural networks it is common to talk about layers and nodes. The first layer is the input layer, containing all of the examples and features. The second layer is the hidden layer. Here the different nodes provide output information which rely information about characteristics in the input. The last layer is the output layer. Here the output from the hidden layer is gathered in each output node, and the one firing the most is chosen. An example of the neural network can be seen in figure 2.9.

In a neural network each node in the hidden layer acts as a single logistic regression function, outputting a value from 0 to 1 depending on how well a feature it has been tuned for is prominent in an example. The output from the hidden layer is then passed forward to the output layers and give an output hypothesis [15].

$$\begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \dots \\ a_m^{(2)} \end{bmatrix} \rightarrow h_\theta(x) \quad (2.18)$$

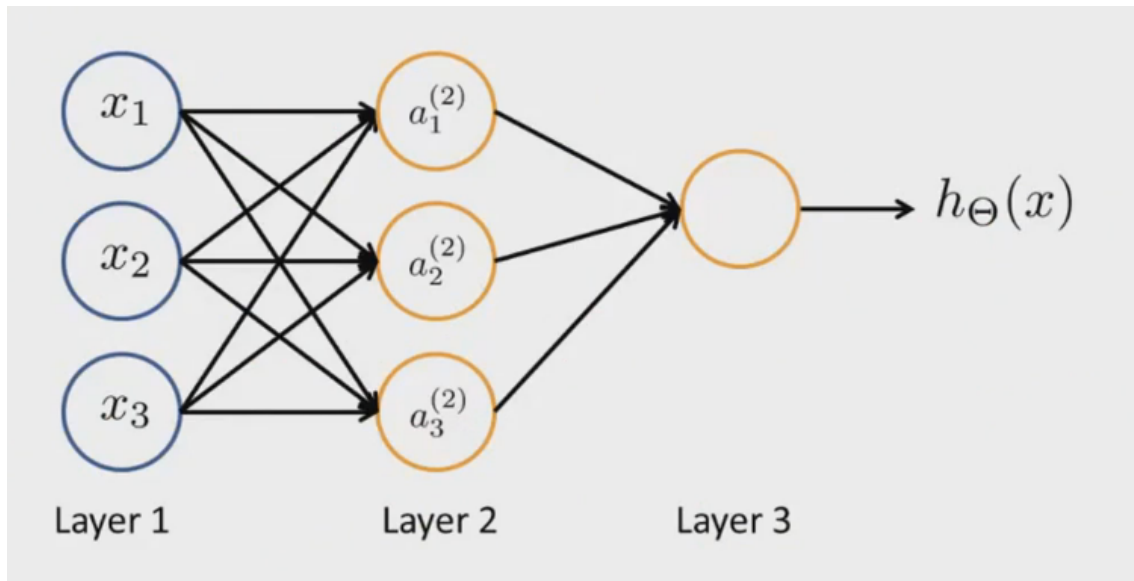


Figure 2.9: Here is an illustration of a neural network with one hidden layer [10]. Both the input layer and the hidden layer contain three nodes. The biases are not included in the illustration.

Here  $n$  is the number of features in and example the input layer,  $m$  is the number of nodes in the hidden layer. One node in the hidden layer is represented by  $a_i^{(j)}$  where  $i$  represents the node number and  $j$  represents the layer. Using this subscript, the weight acting on a layer can be written as  $\Theta_{in}^{(j)}$ . The way of calculating  $a_1^{(2)}$  and  $a_2^{(2)}$  would then be [15]:

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \dots + \Theta_{1n}^{(1)}x_n) \quad (2.19)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \dots + \Theta_{2n}^{(1)}x_n) \quad (2.20)$$

The equation can be simplified by setting:

$$z_1^{(2)} = \Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \dots + \Theta_{1n}^{(1)}x_n \quad (2.21)$$

$$z_2^{(2)} = \Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \dots + \Theta_{2n}^{(1)}x_n \quad (2.22)$$

so that :

$$a_1^{(2)} = g(z_1^{(2)}) \quad (2.23)$$

$$a_2^{(2)} = g(z_2^{(2)}) \quad (2.24)$$

The equation for calculating the hypothesis output for 1 output as above would then be [15]:

$$h_{\Theta}(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \dots + \Theta_{mn}^{(2)} a_m^{(2)}) \quad (2.25)$$

For multi class classification, and not single class like above, the hypothesis output becomes a column vector of zeros and a 1 for the right classifier. For example here the hypothesis shows the third classification:

$$h_{\Theta}(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (2.26)$$

Initializing all the weights in a neural network with the same value does not work [10]. This will cause all of the nodes in a hidden layer to update to the same value repeatedly. A method of random initialization which ensures a quick neural network and the ability to detect complex features is [17]:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{L_{output} + L_{input}}} \quad (2.27)$$

$$\Theta^{(l)} = 2 \epsilon \text{rand}(L_{output}, L_{input} + 1) - \epsilon \quad (2.28)$$

Here  $\epsilon$  is a value for setting the range interval of the random initialization of weights.  $L_{input}$  and  $L_{output}$  are the sizes of the input and output layers the weights are defined for.

The last part of the neural network is applying the cost function, and calculating the gradient. The cost function for a neural network is [14]:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K [y_k^{(i)} \log((h_{\Theta}(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2 \quad (2.29)$$

A quick explanation without going too much into detail. Compared to the logistic regression cost the first part of the equation has a nested sum over the total number of output nodes  $K$ . In the second part, multiple weight matrices are taken account for. The number of columns in the current theta matrix is equal to the number of nodes in the current layer (including bias). The number of rows in the current theta matrix is equal to the number of nodes in the next layer (excluding bias unit).  $s_l$  is the number of nodes in a layer excluding the bias unit.  $L$  is the total number of layers. The tripple sum just sums up all of the individual weights except for the bias, which is not regularized [14].



The last part of the neural network is the to calculate the gradient. The gradient of a neural network is calculated from the back-propagation algorithm and is very complex. What it does is sums up the error over every individual node, so that the error reduction can be traced back to the nodes that produces the greatest errors. The errors are then propagated backwards through the neural network. The process can be shortened to six steps [10] [14].

First: Perform a calculation through the neural network.

Second: For each output unit  $k$  in the output layer set

$$\delta_k^{(3)} = (a_k^{(3)} - y_k) \quad (2.30)$$

Where  $y_k$  is either 0 or 1, and indicates if the current training example belongs to class  $k$ , or to a different class.

Third: For the hidden layer, set:

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)}) \quad (2.31)$$

(.\* denotes element-wise multiplication, and  $T$  the transpose.)

Fourth: Accumulate the gradient from this example:

$$\Delta^{(l)} = Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T \quad (2.32)$$

Fifth: Obtain the un-regularized gradient for the neural network cost function from the accumulated gradients and divide by number of examples:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \quad (2.33)$$

Sixth: Include the regularization for all terms except the bias:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \quad (2.34)$$

The cost and gradient of the neural network is then used together with an advanced optimization function to train the neural network for a given number of iterations until the cost versus iterations graph flattens out.

# Chapter 3

## Method

A program for extracting and plotting information from the data files had already been developed in beforehand of the thesis. During the thesis this program was edited and improved on to fit the needs of the project. This program was also ported into python, but due to time constraint, more of the project was not.

Matlab [5] was used as the development environment for the machine learning algorithms. The main reason for this is that Matlab offers a very quick and easy way of handling vector and matrix multiplications. The language is built with this in mind, so the code will more often be easily read. Perfect for a low-level programmer. In other languages, many of the features in Matlab are not built in, and requires calling of additional packages, which often can lead to a code which is not as easily read.

The Matlab programs made for the machine learning algorithms are heavily influenced by the teachings of Andrew Ng from Stanford University, and their available course in machine learning on Coursera.org [10]. The sections covering the different Matlab programs for machine learning are meant as a explanation of the process the programs go through, and is best read together with the corresponding program code available in the appendix.

### 3.1 Extracting and plotting information

From the start of the project, a program for exporting the sensor data from the data files was provided called `readColCapData.m`, created by Time. Additionally a program for plotting: the sensor data versus time, Probability Density Function, Single sided amplitude spectrum FFT, and cross correlation plot with Uslug calculation. The last three functions were only implemented based on one sensor, or sensor pair.

The plotting program was extended to apply the last three function to all sensor pairs, and was also used as the main basis for a data-extraction program for the thesis. For the data extraction, sections of the provided program was changed, just to fit its purpose. It has not been included with the thesis, a short description will however be given.

The first output of the plotting program was not altered. It plots the time when a measurement was recorded on the x-axis, and plots the recorded capacitance on the y-axis. The sensors are grouped by pairs and plotted in the same sub-plot in a 3x1 plot as shown in figure 3.1. The time plotted on the x-axis is not stored by itself, but is based on the number of times the capacitance sensors logged data, and the scan delay between each recording. The scan delay is also extracted from the file.

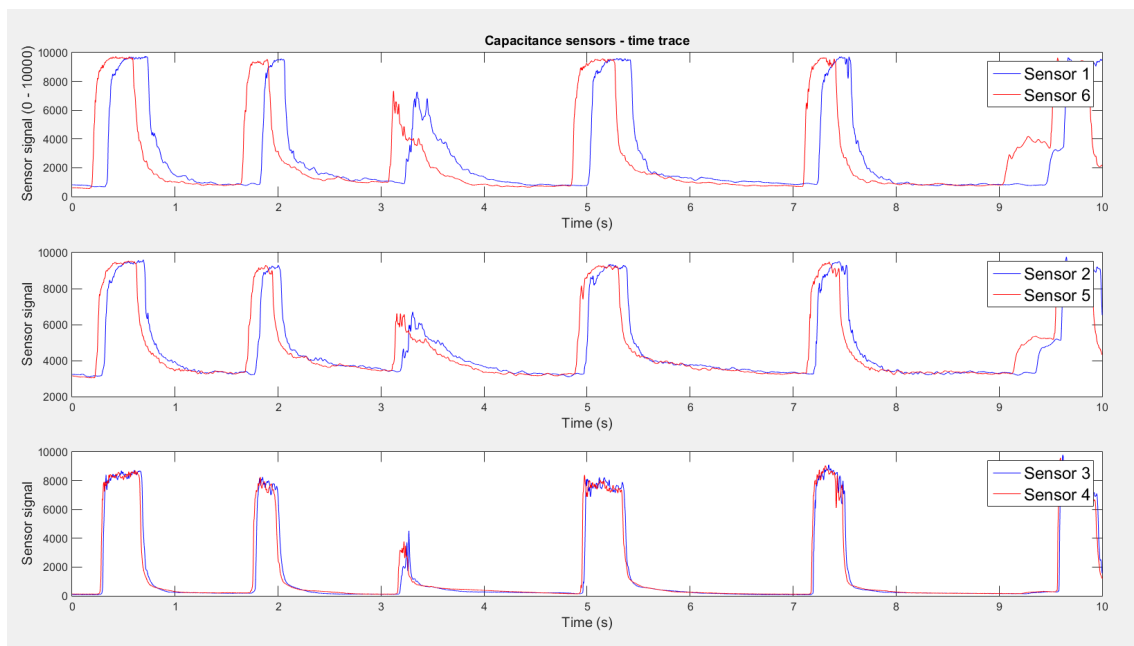


Figure 3.1: Here is an example of the first output figure from the plotting program. The plot is a time trace of the capacitance sensor readout.

The next part of the program computes and plots the single-sided fast Fourier transform of the data from the capacitance sensors. The program originally only plotted this for one sensor, but was extended to include all sensors and paired in groups similar to the capacitance time trace plot. An example can be seen in Figure 3.2

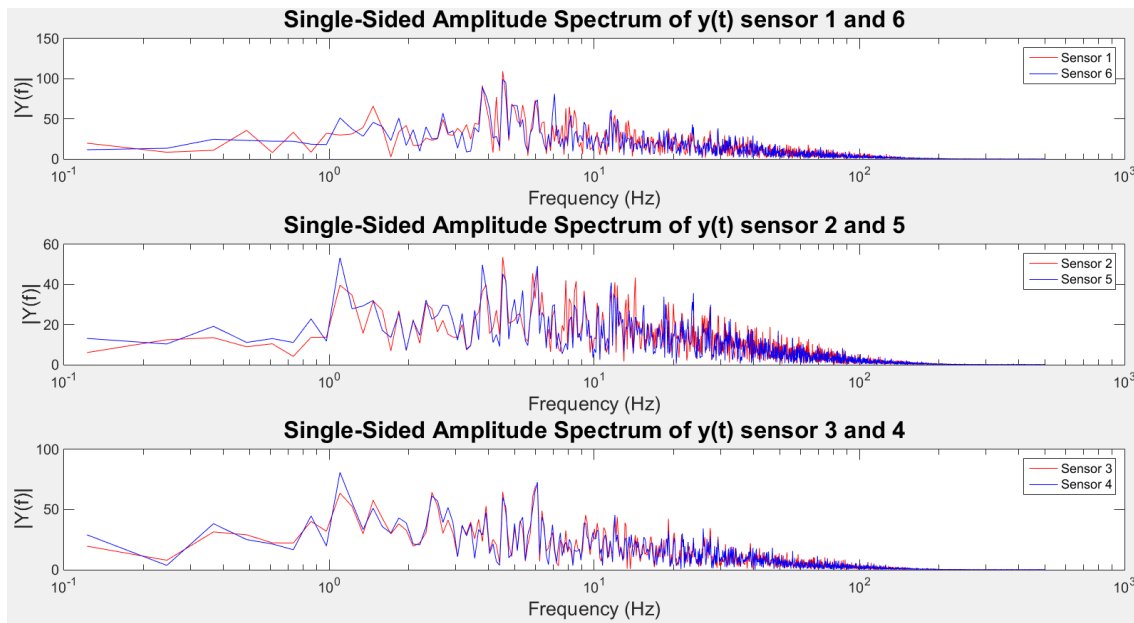


Figure 3.2: Here is an example of the second output figure from the plotting program. shows the single sided fast Fourier transform of the capacitance sensor data, displaying frequency data.

The third output of the program is a histogram of the capacitance sensor data. This serves as a plot of the probability density function, as it counts the number of occurrences of numbers within a set "distance". The tops of the bins in the histogram will then represent a PDF plot. An example can be seen in figure3.3.

The last part of the program does a cross correlation of the capacitance sensor pairs and calculates the slug speed based on this cross correlation. The process is done by using the Matlab *circshift* function. One of the capacitance sensor time series is kept the same through the whole process, the other one of the pair is displaced one by one measurement at a time. The displacement is done by taking the last measurement, and putting it at the start of the time series, and moving all other measurements one space. The logic is that this will produce the greatest peak when the measurement tops match each other. The slug speed ( $U_{slug}$ ) is then calculated based on the number of steps to the peak of the cross correlation function and the scan delay of the sensor. The cross correlations function is also plotted, but only serves as a visualization of the process, and holds little other significance.

The program also outputs a mixture velocity ( $U_{mix}$ ) and a ration between  $U_{slug}$  and  $U_{mix}$  based on the superficial velocities recorded alongside the capacitance recordings. The superficial velocities are assumed as unknown in the machine learn-

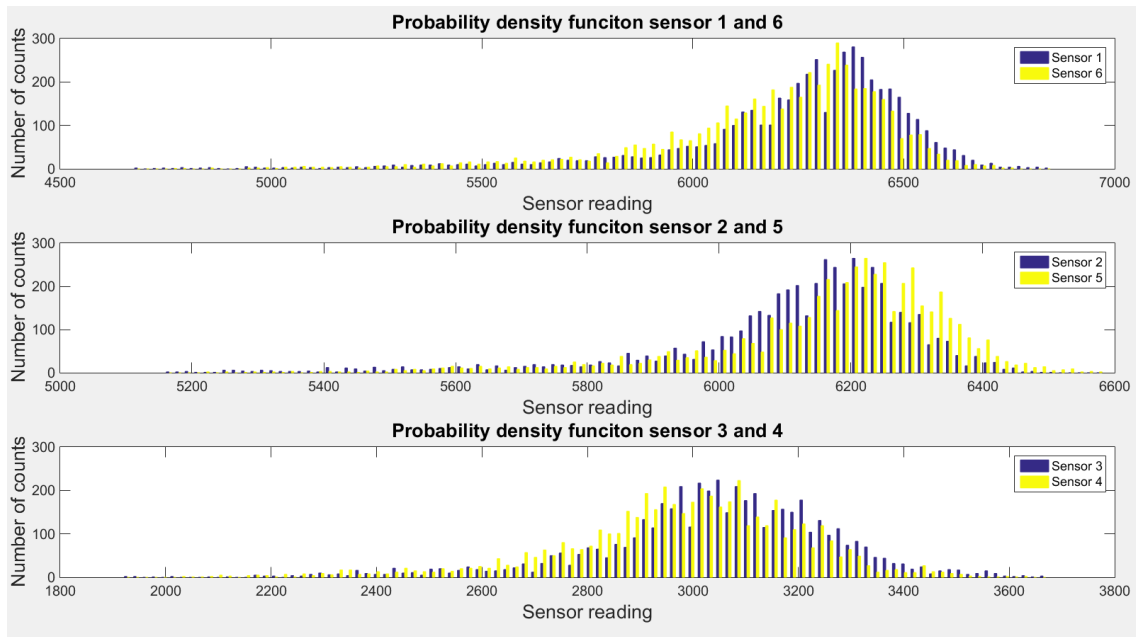


Figure 3.3: The third figure output from the program is a discrete probability density function.

ing process, and will not be used. An alternate version of this program also exists. It produces a single figure containing the top three outputs, capacitance time series, Fourier transform and probability density function in a 3x3 plot. These figures were saved with for each data-file and used together with the k-means program to try and manually interpret a classification of the flow regime.

## 3.2 k-means program

The k-means program bases itself on the theory presented in chapter 2.3.1 Unsupervised learning: k-means clustering. Some of the text might be a bit repetitive, as it goes through much of the same objective.

The program is a combination of 7 files. One main program for running the full algorithm, and functions for calling different parts or "sub algorithms". This arrangement makes it easier to spot mistakes, and test different parts of the algorithm, as you can call one of the at a time. The full code for the program can be seen in the appendix.

<code>kmain.m</code>	The main program for running the whole algorithm.
<code>randomInit.m</code>	Random initialization of centroids.
<code>runkmeans.m</code>	Runs the k-means part of the algorithm.
<code>assignClosestCentroid.m</code>	Assigns data sets to the closest centroid.
<code>computeCentroids.m</code>	Computes new centroids based on the assigned data sets.
<code>computedist.m</code>	Computes the distortion of the k-means run.
<code>plotKmeans.m</code>	Plots the assigned clusters in a Uls-Ugs diagram.

### 3.2.1 kmain.m

The `kmain.m` program is the main hub of the algorithm, and this is where the user input is controlled. The program initializes at the start with clearing all previous stored information in the matlab memory, and then initiates a counting variable. Following the user input is listed: "K", the number of clusters, `totalloops`, the number of randomly initialized loops, and lastly, `max_iterate`, the number of iteration to define the cluster centroid per initialization.

The input data values are then loaded, and assigned to the X matrix. In the X matrix each row represents a different measurement. Each column represents a type of data from that measurement. There is also a choice to enable calculation of the standard score, a form of normalization. This can be applicable if many types of data of different magnitudes are used.

The rest of `kmain.m` is just the complete run of the K-means algorithm, calling the other functions, and lastly finding data associated with the global minima (or

in most cases a near global minima) and plotting it with the `plotKmeans.m`.

### 3.2.2 `randomInit.m` and `runkmeans.m`

The `randomInit.m` function secures the randomized initialization of each run of the k-means algorithm. The program randomly rearranges the rows of the  $X$  matrix and then chooses the first number of rows from the rearranged matrix equal to the number of centroids. This would mean that each centroid will start of equal to a point defined by one of the samples. This will help prevent unassigned centroids, and should be very rear with this initialization.

The next program runs the main par of the k-means algorithm. The k-means algorithm relies on two other functions. This setup is for a hierarchical and easier management. For every iteration from 1 to `max_iterate`, `runkmeans.m` will call on `assignClosestCentroid.m` for assigning the different experiments to the nearest centroid. Afterwards the `computeCentroids.m` is called to calculate new values for the centroids based on the assignments. For each iteration, the centroid will move closer to its local optima based on the randomized initial values.

### 3.2.3 `assignClosestCentroid.m` and `computeCentroids.m`

The assignment of each example to its closest centroid is done by looping over the number of centroids, and computing the squared distance from each example to the looping centroid. Each example is then assigned to the centroid with the least squared distance.

The computation of new centroids is then done be taking the mean value of all examples which are assigned to the same cluster. The `computeCentroids.m` also checks if one of centroid has become unassigned and does not perform the computation if this is true, as this would lead to errors.

### 3.2.4 `computedist.m` and `plotKmeans.m`

The first of the final functions, `computedist.m`, computes the distortion of the k-means run. This is the squared distance from the examples within a cluster, to that cluster centroid. Then the sum of all of the squared distances is stored and used for comparison after all random initializations have been run.

Before `plotKmeans.m` handles the plotting, the main program finds the cluster assignment which yields the lowest distortion. These cluster indexes are then fed to the plotting function, and each example is plotted on a Uls-Ugs plot. Each example is represented as a point using Uls and Ugs data collected alongside the capacitance data. The cluster to which the data has been assigned is shown with a marker. This will help to correlate the assignments with often used models, like Taitel and Dukler [22] or Mandhane [8] flow regime maps. In Figure 3.4 you can see an example of figure output from `plotKmeans.m`.

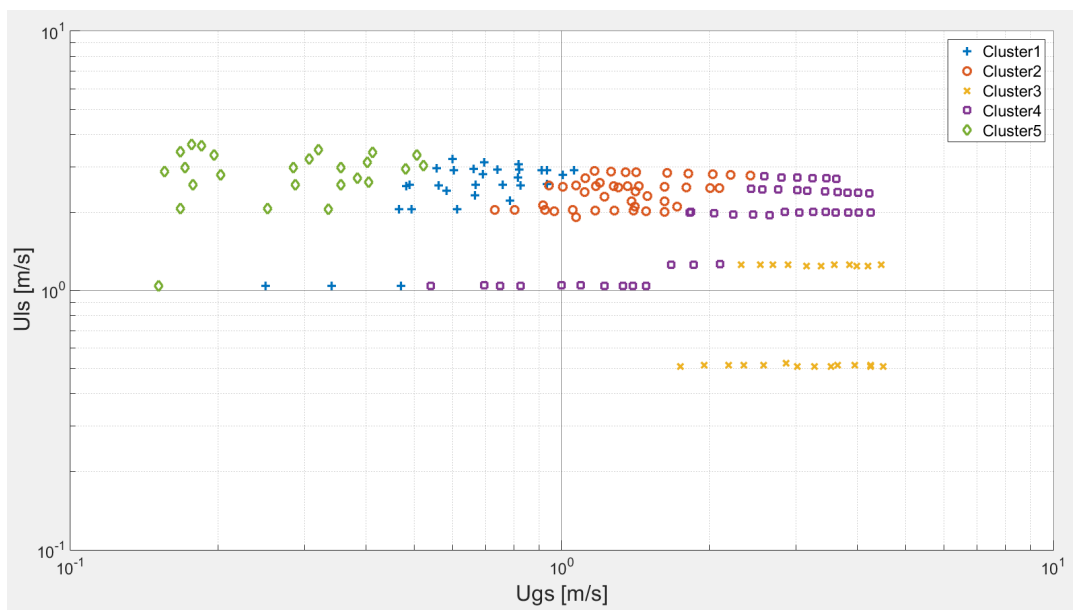


Figure 3.4: This is an example of the output from the plotting function after a k-means run. Here there are five clusters, based on CDF of data from sensor 1.

### 3.3 Neural network

The full code of the neural network can be seen in Appendix B, with the exception of the `fminunc` function. This is created by Rebello and is a function minimization routine for logistic regression similar to the Matlab function `fminunc`. The `fmincg.m` is available online [21].

It is common practice to check the implementation of the back propagation algorithm by doing gradient descent. This requires some time consuming programming and testing. The program presented in the Machine Learning course came with a gradient checker, but required modification to work with this program. As an alternative the program provided was therefore tested with a sample of data used



in the Machine Learning course [10]. This test data had the same outputs for both implementations, and the back propagation was seen as correctly implemented.

The program is a combination of 10 files. The files are here listed in chronological working order.

<code>NNmain.m</code>	The main program for running the whole algorithm.
<code>nnRandomInit.m</code>	Randomly chooses examples for training and test set.
<code>randInitializeWeights.m</code>	Randomly initializes the weights in the neural network.
<code>nnCostFunction.m</code>	The cost function for the neural network computes both forward propagation and the back propagation through the network.
<code>sigmoid.m</code>	Computes the sigmoid function.
<code>sigmoidGradient.m</code>	Computes the gradient of the sigmoid function.
<code>fmincg.m</code>	A function minimization routine [21].
<code>predict.m</code>	Predicts the classification of an input based on the trained neural network.
<code>plotNN.m</code>	Plots the predictions and specifies false predictions.

### 3.3.1 NNmain.m

The `NNmain.m` acts in a similar way as the main file for the k-means algorithm. It is the main hub for the algorithm, inputting user data and choosing parameters. The assignment of the example matrix,  $X$ , and choosing which features to use is the first input. the  $y$  variable is set to the matrix containing the classification for each example.

It is then possible to choose a randomized initialization where a number of the examples are picked for training, and the rest are picked for testing the algorithm. This will lead to variances in the result, because of the possibility of a large portion of the test series being outlier which are hard to identify from the example set. The other option is to test the set on itself, but can lead to misleading prediction information due to over fitting.

The sizes of the different layers are then chosen. The input layer size is automatically assigned, as its size is determined by the amount of training examples. The number of hidden layers can be manually set, and the number of labels is set by checking the maximum value (and should work automatically given the labelling method given earlier). Lambda is also set for use of regularization.

### 3.3.2 `nnRandInit.m` and `randInitializeWeights.m`

The function for randomly selecting examples for training and testing, `nnRandInit.m`, bases itself on the method of the `k-means randomInit` function. The number of examples used for training is based on the training size input. This should be a fraction between 0 to 1. A common practice is to use 70 percent for training and the rest for testing. The function multiplies the examples with the fraction and rounds to the nearest whole number. These are chosen as a training set and the rest as a test set. The `y` column vector is added to the end of the `X` matrix to secure that the example factors and classifier stays connected. The rows are then resorted in a random pattern, and the randomized index is stored. The training sets are then selected from the randomized list by extracting rows from the top to the size of the training set, and then the rest of the rows as test sets. The last column is then separated from the matrix and again stored as the classifier for the train and test set separately.

The randomized initialization of weights is based on the method proposed by Nguyen et. al which is discussed in chapter two. The function randomly initialises the weights based on the sizes of the input layer, the hidden layer and the output layer [17].

### 3.3.3 `nnCostFunction.m` and `Predict.m`

The cost function for logistic regression is implemented in `nnCostFunction.m`. Before the initial values for the weights ( $\Theta$ ) are input into the algorithm, they are unrolled. This is to make it work with the `fmincg` function. Inside they are then reshaped before proceeding. The function then does three things: A forward propagation of the neural network, calculates the cost of the neural network with regularization, and then does a back propagation to find the gradient of the weights. The outputs from this function is the cost and the gradient of the weights. The `fmincg` function is then used as an advanced optimizer to iterate over the `nnCostFunction`

and minimizing the error caused by the weights and the cost as an indicator for it. The cost for each iteration is logged, and can be plotted to check if the number of training iterations are enough. Figure 3.5 shows that the graph flattens out as the cost approached 100 iterations. More iterations will not reduce the cost significantly.

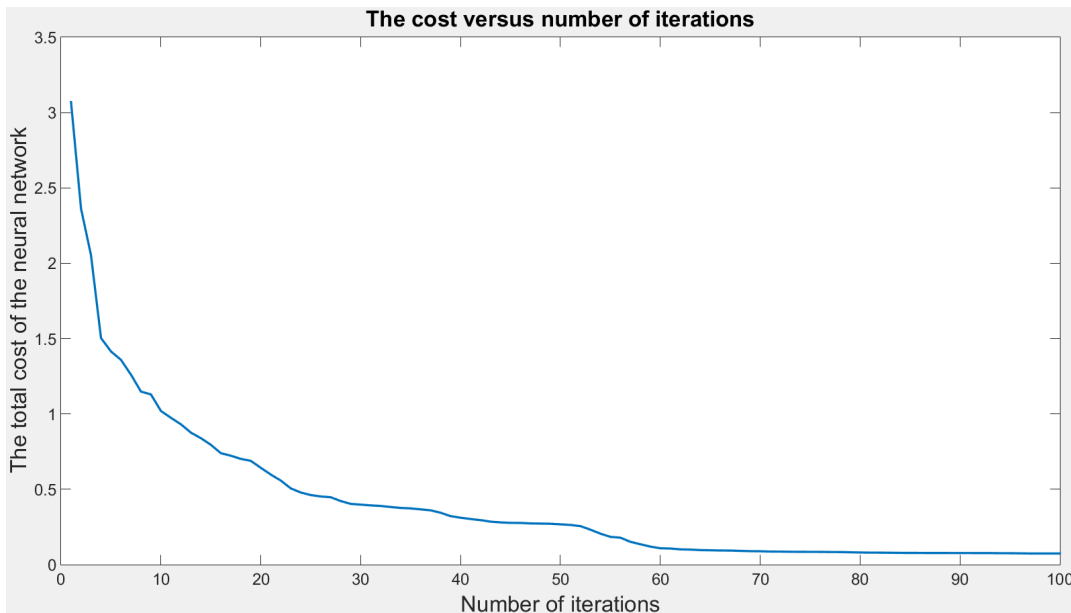


Figure 3.5: Here the cost of a run on the Neural Network is plotted against the number of iterations. It serves as a measurement of how well the training is working.

The `nnCostFunction` also calls the `sigmoid.m` and `sigmoidGradient.m`. These are a Matlab implementation of the sigmoid function, and the gradient calculation of the sigmoid function, and follows the theory from chapter 2.

The `predict` function is used for predicting the classification of data based on the trained network. It takes the input of the weight calculated from the training step and the features from new example(s). The function then outputs the predicted classification for either one example or as a column vector for many examples.

### 3.3.4 `plotNN.m`

The plotting program borrows a lot from plotting function for k-means. It takes the prediction, the number of labels, the index numbers of the test set and the test set itself as input. The prediction is then plotted, following which the wrongly predicted examples are crossed over by a red x. The function also returns the index

of the miss-predictions so that it is possible to check which ones fail more easily. An example of the output plot can be seen in figure 3.6

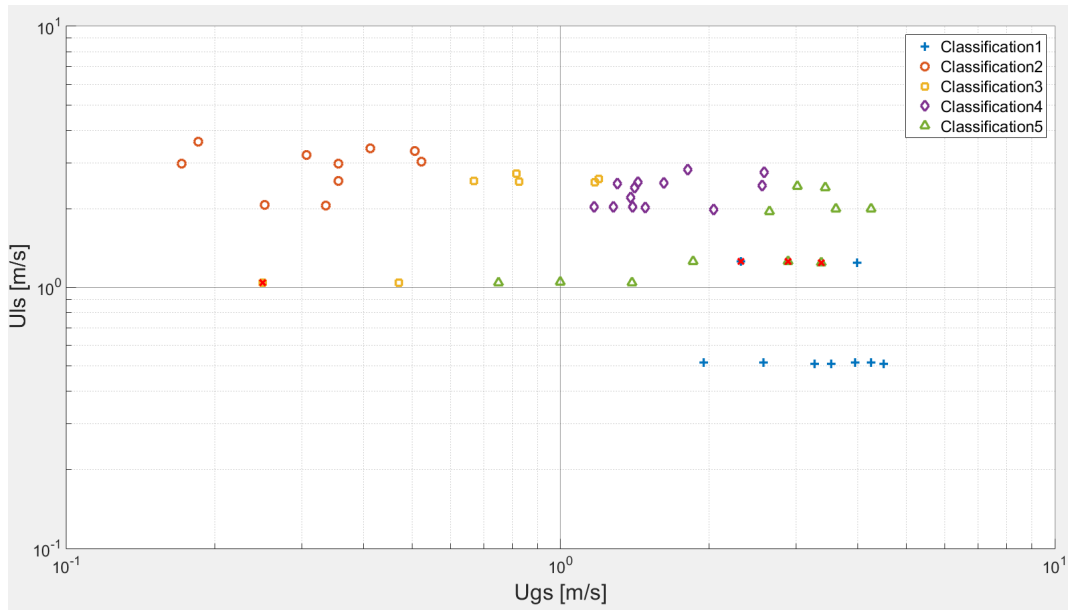


Figure 3.6: Here is an example of the output from the plotNN function. The run uses the classification from figure 3.4, and the PDSF from sensor 1 to train on a random 70 percentage of the data. The wrongly predicted examples are marked with a red x.

# Chapter 4

## Results

Firstly this section will discuss the clusters output from the k-means algorithm. This section will also contain the methodology and process used to try and manually assign the different data-recordings to the flow regime it represents. The supervised neural network needs the classifier as an input to its training set, and also as a way to check the validity of the predictions on the test set. Afterwards the outputs and the neural network will be presented and discussed.

A problem with the data provided is that it only covers two zones in the Taitel and Dukler model, see figure 4.3. According to this model, there should only be two clusters. On the other hand, the intermittent flow regime covers a large area, and the flow regime may cover many smaller sub regimes which are somewhat distinguishable.

### 4.1 Clustering with k-means

The k-means algorithm was used to run a series of tests with varying input and cluster centroids. The idea behind this process is to see if some cluster borders on the Uls-Ugs diagram will remain, or shift for different data input and number of clusters. This can then be an indicator if a cluster is very rigid and separated by some distance in vector space, or if the examples are in an "oblong cloud" of points and clusters can move greatly based on the number of clusters and data. It can also be a pointer to whether the data input serves as a good indicator for flow regime.

For the figures discussed in this section, the k-means algorithm was run with 40 iteration per initialization to properly center the centroids, and 200 random initializations were run to minimize the chance of missing the global minima, or at

least hitting a very close to global minima. The run-times of the k-means algorithm becomes very long when the input matrices are large. For example, a run with all sensor data, and four clusters took about 21 minutes to finish.

The first runs was done using four clusters and the capacitance sensor data as input. The runs were repeated with a number of different data inputs. First one run for each sensor, then one run for each sensor pair. After that a run for sensor 1, 2 and 3, and 4, 5 and 6. Lastly a run with all sensors as input. A comparison between the runs shows that sensor 1 and 6 and the pair as input produces the same result (or very close). The same goes for the rest of the sensors. The runs with 1, 2, and 3 and 4, 5, and 6 also produces the same result as the run with all sensors. This will reduce the number of runs required significantly to get the information. For the rest of the runs, only individual runs with sensor 1, 2 and 3 and a run with all three will be used. As the clusters output from other inputs will be assumed the same. A full test was also done with three clusters. The same observations were made.

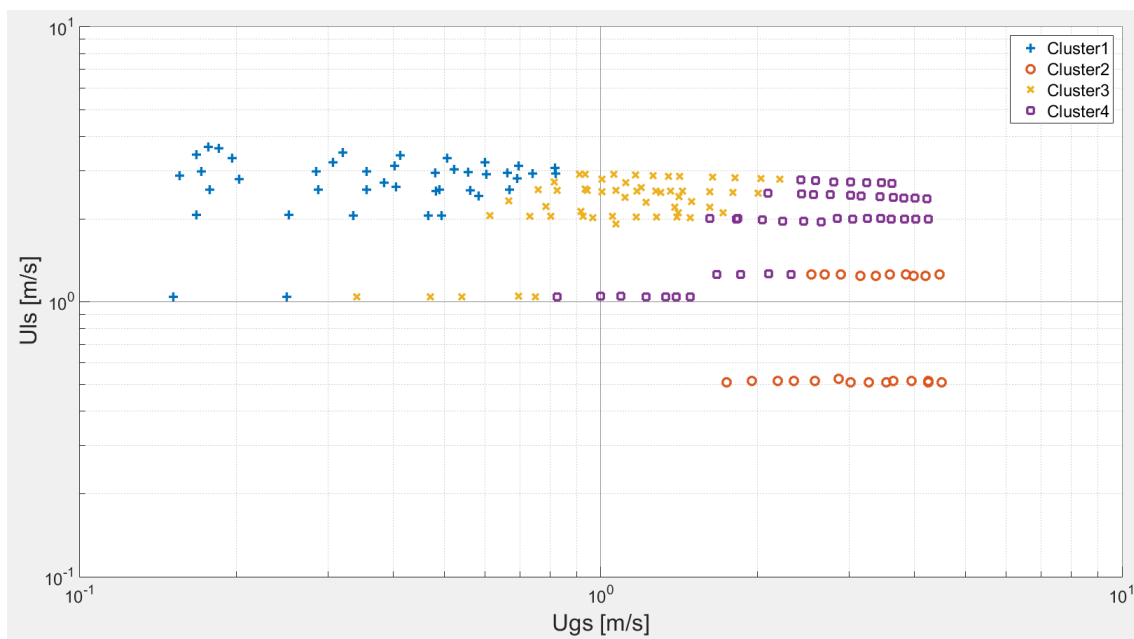


Figure 4.1: The k-means algorithm run with 200 random initializations based on sensor 6 capacitance sensor data.

The algorithm was run for clusters ranging from 2 to 5, and for data types: capacitance sensor data, Uslug. It was also tested with the CDF and the sorted data PDSF. Since the PDSF holds exactly the same values as the capacitance sensor data, but sorted, it yields the same results in the k-means algorithm. It can however be noted that the PDSF retains its "clustering quality" even when "compressed" a lot.

If for example only every 100th value is taken from the PDSF, and then run in the k-means, it will produce the same result as the full vector, but in a much shorter time. Figure 4.2 shows the output from the k-means algorithm based on capacitance sensor data from sensor 6.

The k-means algorithm produces somewhat stable cluster segregation for the sensor pairs as the number of clusters increase. The k-means algorithm did not fare well with the Uslug data, nor the FFT. This may be because of the implementation of the FFT data, which is the amplitude data put in, and should rather be linked to tops and locations. Both the sensor data, PDF and CDF seem to produce somewhat similar clusters. The k-means algorithm seems like it is better suited for a general assessment of flow regimes, but not good for finding clear borders between flow regimes.

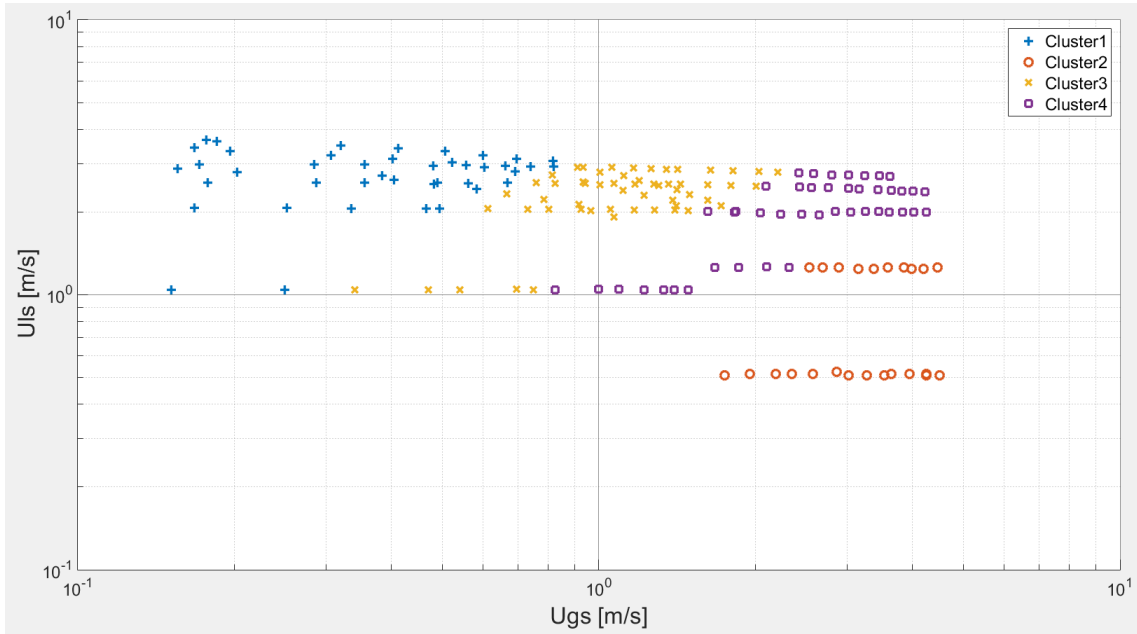


Figure 4.2: The k-means algorithm run with 200 random initializations based on sensor 6 capacitance sensor data.

## 4.2 Assigning regimes to examples

Throughout the thesis it was believed that previous interpretation of the examples could be used as a guideline for assigning flow regimes to the different data examples before inputting into the neural network. This, however, was not feasible. Therefore a manual approach was chosen, based on the outputs of the k-means algorithm, the figures of sensor data, PDF and FFT, as well as a comparison with the Taitel and

Dukler model.

When the Uls-Ugs plot of the data was compared with the Taitel and Dukler model by Time, figure 2.1,

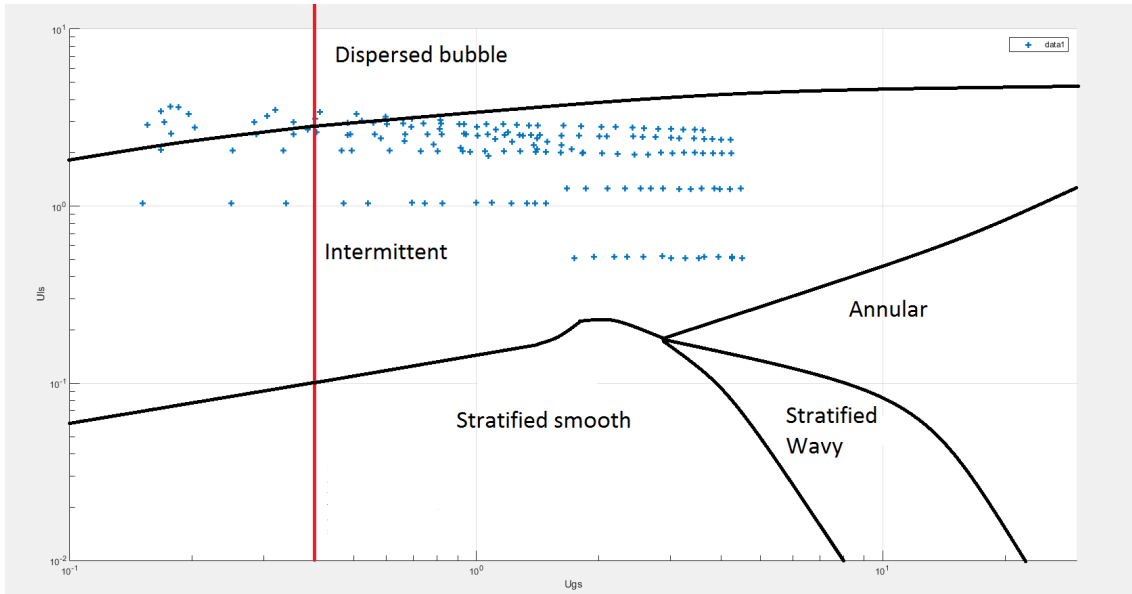


Figure 4.3: Here the straight lines from the Taitel and Dukler model from figure 2.1 has been roughly traced on top of a plot of all examples in Uls - Ugs space. The red line marks the end of the original figure, and the lines to the left have been assumed to follow the same trend.

The only classification available was from the test runs done by Eeg [3]. He had noted the observed regime as either: Dispersed bubble, Intermittent or slug. The definition of the three being:

- Dispersed bubble: Dispersed bubble flow, both visually confirmed and from the oscilloscope.
- Intermittent: Visually looks like dispersed bubble flow, but sensor sensor 3 and 4 show great fluctuations on the oscilloscope.
- Slug: Visually observed as slug slow, and the oscilloscope clearly shows a slug pattern.

A plot of the observations done by Eeg, can be seen in figure 4.4, also with the Taitel and Dukler model traced on it. As you can see, the model does not fit the observations done by Eeg, and an for the model to fit, an adjustment is required. One other problem with the Eeg observations is the intermittent between slug and dispersed bubble. These can be very hard to determine manually afterwards, and



most likely a line will have to be set based on a "hunch".

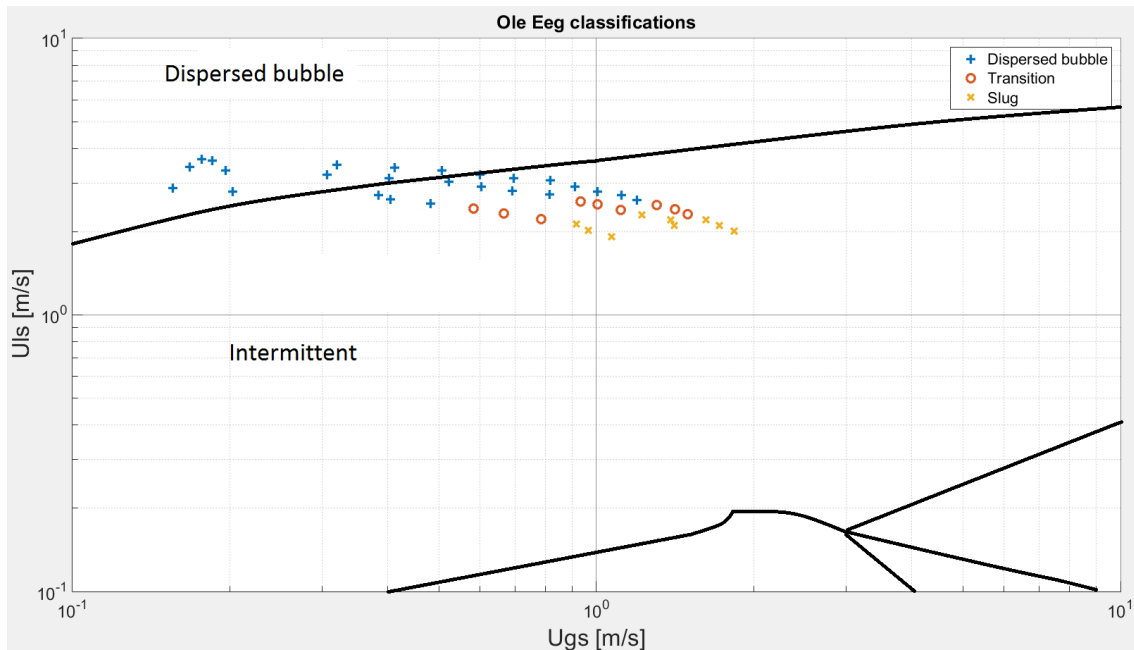


Figure 4.4: A plot showing the observed classifications of Eeg [3] and the Taitel and Dukler model from figure 2.1 roughly traced on top.

After reviewing the data output from the k-means algorithm and also manually checking the PDF and FFT data of some of the data point close to the border. Then trying to correlate this data with the observation from Eeg, it was hard to find a solid "border marker" for identifying the flow regimes. A choice was made to have one set of identifiers just based on sensor data run through the k-means algorithm with four clusters, which means four classifiers. The other was based on the observation by Eeg, and trying to adjust the Taitel and Dukler to group into dispersed bubble, intermittend and slug. Two data sets were chosen, to see if different classification lines in the data would affect the prediction values of the neural network.

### 4.3 Running the neural network

After labelling the examples, the neural network can be trained and used to predict flow regime on test examples. The program itself randomly selects training examples, and the rest as a test set. It is also possible to train the neural network on all of the examples, and test it on itself, but this will not reflect how well it will behave with outside data. Therefore a test set will better illustrate this. The randomness of the test set will cause the prediction accuracy to vary somewhat based on which ex-

amples get trained and tested. If it were to achieve 100 percent accuracy on the test set each time, this would mean the input information and hypothesis would easily distinguish flow regimes. This will probably not be the case, but a high prediction accuracy should be possible, at least with the k-means classifiers.

The k-means classifiers were run first through the neural network. The network was first tested on itself, to see if the prediction percentage with different input would yield results differing a lot from each other. All identifiers gave outcome of 100 percent when testing on themselves, except for Uslug. The Uslug speed is therefore not a good identifier by itself, but might help in some situations. Another way of pretesting the neural network is to check prediction precision when only prediction one classification. In the k-means set, this is classifier 2 which yields 39 percent.

The random initialization was then run and tested with different identifiers. For each test, a total of 20 random initializations were run, and the maximum, minimum, and average value of the prediction is presented. 20 initialization might not be enough to the real maximum, minimum and average values for the neural network, but it gives an indication of how well the data performs.

Test data	Max	Min	Average
Sensor 1 CDF	100	92	97
Sensor 2 CDF	100	90	96
Sensor 3 CDF	96	76	86
All sensor CDF	98	82	92
Sensor 1 data	84	70	76
Sensor 3 data	80	56	66
Sensor 1, 2, 3 data	86	62	76
Sensor 1 PDSF	100	92	96
Sensor 3 PDSF	94	78	87
Sensor 1, 2, 3 PDSF	100	80	94
Sensor 1 PDSF 10 values	100	90	97
Sensor 1 FFT	90	72	81
Uslug	70	46	57

*The PDSF 10 values are extracted taking every 50th value from the PDSF examples.* The Neural network overall performs very well on the k-means classifiers, scoring averages as high as 94 percent correct prediction. The PDSF is able to retain its prediction qualities for flow regime classifications, even though only 10 values are

used to represent the function. This lets you do a very quick training of the network if you have a lot of examples.

The Eeg-based identifications were the next to be run through the algorithm. More problems is to be expected, as the manual classification proved difficult. When testing on the whole training set, and using CDF there were a lot 23 miss-predictions. These can be seen in figure 4.5. if you were to only predict one classification, the maximum hit in the Eeg-based classifiers is classification 3, "slug", and yield 53 percent hit.

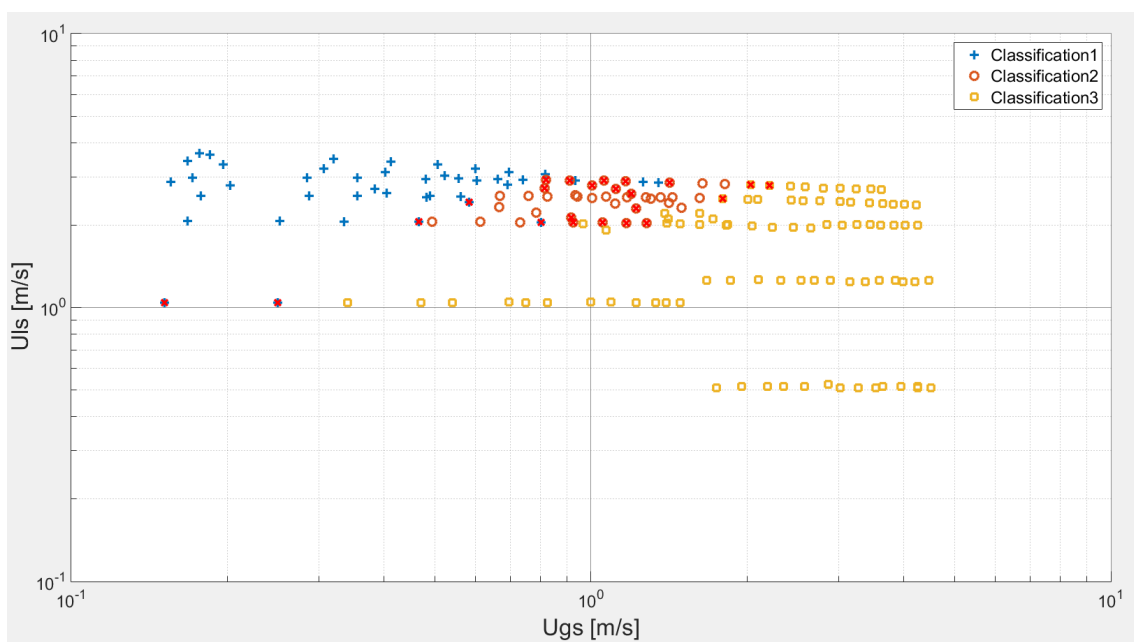


Figure 4.5: The output plot from the neural network testing on the whole training set using CDF from sensor 1.

The data was then run through the randomly initialized test in the same manner as for the k-means classifiers. The output can be seen in the table below.

Test data	Max	Min	Average
Sensor 1 CDF	98	84	90
Sensor 2 CDF	88	78	80
Sensor 3 CDF	98	86	91
All sensor CDF	96	86	94
Sensor 1 data	86	66	78
Sensor 3 data	86	48	73
Sensor 1, 2, 3 data	94	82	87
Sensor 1 PDSF	94	78	85
Sensor 3 PDSF	98	82	91
Sensor 1, 2, 3 PDSF	98	90	94
Sensor 1 PDSF 10 values	90	72	82
Sensor 1 FFT	92	72	81
Uslug	76	54	64

Overall, the neural network predictions worked very well with many different inputs. The capacitance sensor data, CDF, PDSF were the ones that yielded the highest results. Even a very reduced version of the PDSF, only represented by ten points, yielded good results. The k-means classification yielded better results than manual / Eeg-based identifications did. The latter were probably poorly classified by me, as this turned out to be hard to do without observations.

# Chapter 5

## Conclusion

Two machine learning programs were created. The unsupervised k-means clustering algorithm and the supervised neural network. The k-means clustering algorithm was able produce clusters based on the input data fed into it. The clustering did, however, not match up in a satisfactory way to the visual classifications of examples that were available. The algorithm can serve as a general indication towards which flow regime an example should be grouped. Precision at flow regime borders is where the algorithm falls short. This is at least true for the data used in this thesis. Other methods for analysing time series exist. These might be better suited for use with the k-means clustering algorithm.

The neural network is a supervised learning algorithm. The neural network therefore requires a classification of the input examples. This was only available for a fraction of the data examples utilized in this thesis. This meant that the data had to be classified manually before being entered into the neural network. This was where the k-means algorithm was going to be applied, but it did not produce optimal results. The neural network was therefore tested on two classification sets. One set produced by the k-means algorithm and one set based on the available visual classifications and some manual interpretation. The manual interpretation was not easy, as interpreting the flow regimes without visual observation was not an easy task. This lead to classifications of the examples which were not optimal. Even so, the neural network was, with a high success rate, able to predict the classification of the example data sets. This shows that the neural network truly is a powerful tool in analysing complex non-linear hypothesis.

The two machine learning algorithms were build in Matlab. They are build in a general manner, so that they can be used again in the future and for other tasks as

well. This as long as the input data is arranged in the same order. Hopefully these programs will be utilized, not only by me, but by anyone who wants to try out a k-means algorithm or a neural network.

The task as presented in the introduction was partly solved. Machine learning was applied as a method for identification of multi phase flow regimes. The results were okay, and at the time expected. They were partly impacted limitations of the example data.

# Bibliography

- [1] John P Bentley. *Principles of measurement systems*. Pearson Education India, 1995.
- [2] E Besalú. A graphical representation to teach the concept of the fourier transform. *J. Chem. Educ*, 83(12):1795, 2006.
- [3] Ole S. Eeg. Undersøgelse af gasfraktions fordeling ved dispergeret boblestrøm i horisontale rør. Master's thesis, Høgskolecenteret i Rogaland, July 1992.
- [4] Donald Olding Hebb. *The organization of behavior: A neuropsychological approach*. John Wiley & Sons, 1949.
- [5] The Mathworks Inc. Matlab 8.4 release 2014b. Natic, Massachusetts, United States.
- [6] Jae Young Lee, Mamoru Ishii, and Nam Seok Kim. Instantaneous and objective flow regime identification method for the vertical upward and downward co-current two-phase flow. *International Journal of Heat and Mass Transfer*, 51(13):3442–3459, 2008.
- [7] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.
- [8] JM Mandhane, GA Gregory, and K Aziz. A flow pattern map for gas—liquid flow in horizontal pipes. *International Journal of Multiphase Flow*, 1(4):537–553, 1974.
- [9] Arnold Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM review*, 40(3):636–666, 1998.
- [10] Andrew Ng. Machine learning. <https://www.coursera.org/learn/machine-learning>. Date accessed: 2016.04.29.

- [11] Andrew Ng. Machine learning wiki and lecture notes: Clustering. <https://share.coursera.org/wiki/index.php/ML:Clustering>. [Online; accessed: 2016.05.06].
- [12] Andrew Ng. Machine learning wiki and lecture notes: Linear regression with multiple variables. [https://share.coursera.org/wiki/index.php/ML:Linear\\_Regression\\_with\\_Multiple\\_Variables](https://share.coursera.org/wiki/index.php/ML:Linear_Regression_with_Multiple_Variables). [Online; accessed: 2016.05.10].
- [13] Andrew Ng. Machine learning wiki and lecture notes: Logistic regression. [https://share.coursera.org/wiki/index.php/ML:Logistic\\_Regression](https://share.coursera.org/wiki/index.php/ML:Logistic_Regression). [Online; accessed: 2016.05.10].
- [14] Andrew Ng. Machine learning wiki and lecture notes: Neural networks: Learning. [https://share.coursera.org/wiki/index.php/ML:Neural\\_Networks:\\_Learning](https://share.coursera.org/wiki/index.php/ML:Neural_Networks:_Learning). [Online; accessed: 2016.05.12].
- [15] Andrew Ng. Machine learning wiki and lecture notes: Neural networks: Representation. [https://share.coursera.org/wiki/index.php/ML:Neural\\_Networks:\\_Representation](https://share.coursera.org/wiki/index.php/ML:Neural_Networks:_Representation). [Online; accessed: 2016.05.11].
- [16] Andrew Ng. Machine learning wiki and lecture notes: Regularization. <https://share.coursera.org/wiki/index.php/ML:Regularization>. [Online; accessed: 2016.05.10].
- [17] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 21–26. IEEE, 1990.
- [18] Cmglee of Wikipedia the free encyclopedia. Comparison convolution correlation. [https://commons.wikimedia.org/wiki/File:Comparison\\_convolution\\_correlation.svg](https://commons.wikimedia.org/wiki/File:Comparison_convolution_correlation.svg), 2014. [Online; accessed 2016.06.26].
- [19] Glosser.ca of Wikipedia the free encyclopedia. Colored neural network. [https://commons.wikimedia.org/wiki/File:Colored\\_neural\\_network.svg](https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg), 2016. [Online; accessed 2016.05.27].
- [20] KSmrq of Wikipedia the free encyclopedia. Extrema example. [https://commons.wikimedia.org/wiki/File:Extrema\\_example.svg](https://commons.wikimedia.org/wiki/File:Extrema_example.svg), 2016. [Online; accessed 2016.04.30].



- [21] Jason Rebello. Logistic regression with regularization used to classify hand written digits. <https://www.mathworks.com/matlabcentral/fileexchange/42770-logicistic-regression-with-regularization-used-to-classify-hand-written-digits/content/Logicistic>[Online; accessed: 2016.05.15].
- [22] Yemada Taitel and AE Dukler. A model for predicting flow regime transitions in horizontal and near horizontal gas-liquid flow. *AIChE Journal*, 22(1):47–55, 1976.
- [23] Rune Time. *Two-Phase Flow in Pipelines: Course Compendium*. Department of Petroleum Engineering, Faculty of Science and Technology, University of Stavanger, 2009.
- [24] Rune W. Time. *Analysis of space and time structures in two-phase flow using capacitance sensors*. PhD thesis, Høgskolesenteret i Rogaland, December 1993.
- [25] Neylon Tyler. Logistic function. [https://share.coursera.org/wiki/index.php/File:Logistic\\_function.png](https://share.coursera.org/wiki/index.php/File:Logistic_function.png), 2015. [Online; accessed 2016.05.27].

# Nomenclature

$\Delta$	The accumulated gradient from backwards propagation
$\delta$	The error of a node in a layer
$\epsilon$	A value for setting the random initialization of weights
$\lambda$	The regularization factor. regulates the degree of regularization
$\mu$	A cluster centroid. The vector representing the center of a cluster
$\theta$	A vector containing the weight vectors for all features
$\Theta_{in}^{(j)}$	the weight acting on layer $j$ , for node $i$ and feature $n$
$\theta_{(i)}$	The weight for one particular feature in an example.
$\theta_{(i)}$	The weight given a feature in an example
$A$	The cross-sectional area of a tube
$a_i^{(j)}$	The activation value of node number $i$ in layer $j$
$c^{(i)}$	The index of a cluster, $k$ , to which an example, $x^{(i)}$ , is assigned
$D$	The accumulated gradient divided by the amount of examples from backwards propagation
$g(z)$	The sigmoid function
$h_\theta$	The hypothesis produced by a learning algorithm
$J$	The cost function
$K$	The total number of clusters defined in the k-means algorithm
$k$	a specific cluster in the k-means algorithm
$L_{input}$	Size of the input layer for random initialization of weights

*Loutput* Size of the output layer for random initialization of weights

$q_G$  The volumetric gas flow rate

$q_L$  The volumetric liquid flow rate

$U_{GS}$  The superficial gas velocity

$U_{LS}$  The superficial liquid velocity

$X$  A matrix containing all examples and features

$x^{(i)}$  An example vector containing features

$y$  A vector containing the classification of all examples

$z_n^{(j)}$  The value of node  $n$  in layer  $j$  before the sigmoid function

# Appendix A

## k-means Matlab code

Presented here is the Matlab code for k-means listed per file, and in a chronological working order. The code is based heavily on the teachings of Andrew Ng and his course on Coursera. [10]

### A.1 kmain.m

```
%Clears everything as an initialization of the program
clc, clear, close('all')
tic %for timing the program

%Main program for running the K-means clustering.

%initiate counting of cycles
cycles = 0;

%===== User input=====
%Number of clusters
K = 1;
%Total of random initialized loops
totalloops = 200;
%Number of iterations to find the optimal cluster per loop
max_iterat = 40;
%Choose wether or not to count cycles; 1 = yes
countcycles = 1;

%Load values
%forstetall = loadforstetall();
```

## APPENDIX A. K-MEANS MATLAB CODE

---

```
load('../flowdata/CDF_500space.mat');
load('../flowdata/PDF_500space.mat');
load('../flowdata/per_sensor_data.mat');
load('../flowdata/per_sensor_data_sorted.mat');
load('../flowdata/Uslug_data.mat');
load('../flowdata/FFT_data.mat');
load('../flowdata/sensor_shift_diff.mat');
X = [sensor1cdf500];
m = size(X,1);
n = size(X,2);

%Standard score, if wanted. Applicable when using
%data of different sizes or nature
[X, mu, sigma] = zscore(X);

%%
%===== The K-means algorithm =====
%Apply K-means a number of times to avoid local
%minima and find global minima
for i = 1 : totalloops

    %Randomly initialize centroids
    initial_centroids = randomInit(X, K);

    %Run the kmeans algorithm for this set of initialization centroids
    [centroids, idx] = runkmeans(X, initial_centroids, max_iterat);

    %Remember the centroids and assignments for each run
    centroidsroll(i,:) = centroids(:)';
    idxroll(:,i) = idx;

    %compute the distortion (cost function for k-means)
    J(i) = computedist(X, centroids, idx);

    if countcycles == 1
        cycles = cycles + 1
    else
        end
end

end
```

```
%Find minimum value for distortion, and it's index
[distortion, index] = min(J);
least_dist_idx = idxroll(:,index);
least_dist_centroids = reshape(centroidsroll(index,:), [K,n]);

distortion;

%plot the measurements in an Uls Ugs diagram and show the assigned
%centroids.
plotKmeans(least_dist_idx,K);
toc
```

## A.2 randomInit.m

```
function centroids = randomInit(X, K)
%This function creates a random initialization
%of centroids based on examples Given in X

% initialize values
centroids = zeros(K, size(X, 2));

% Randomly reorder the indices of examples
randomindex = randperm(size(X, 1));

% Take the first K examples as centroids
centroids = X(randomindex(1:K), :);

end
```

## A.3 runkmeans.m

```
function [centroids idx] = runkmeans(X, initial_centroids, max_iters)

% Each row of X represents an example, and each column of X
% is a feature of X. runkmeans runs the K-means algorithm on
% the X matrix, with the given initial centroids, for a given
```

```
% max iterations.

%initializing values
[m n] = size(X);
K = size(initial_centroids, 1);
centroids = initial_centroids;
idx = zeros(m, 1);

for i = 1:max_iters

    %Assign the closest centroid for each example in X
    idx = assignClosestCentroid(X, centroids);

    %Given assignment to centroids, compute new centroids
    centroids = computeCentroids(X, idx, K);

end

end
```

## A.4 assignClosestCentroid.m

```
function idx = assignClosestCentroids(X, centroids)
% assignClosestCentroid computes the centroid memberships for
% every example

%Initialize values
% Set K
K = size(centroids, 1);
idx = zeros(size(X,1), 1);

distance = zeros(size(X,1),K);
%Find the distance from each point to the initialized centroids
for i = 1:K
    diffs = bsxfun(@minus, X, centroids(i,:));
    distance(:,i) = sum(diffs.^2, 2);
end
```

```
% index the column which holds the minimum distance.
[fodder index] = min(distance, [], 2);

idx = index;

end
```

## A.5 computeCentroids.m

```
function centroids = computeCentroids(X, idx, K)
%This function computes new centroids based on the
%samples assigned to each centroid.
%It should also deal with the unlikely situation of an
%unassigned centroid.

[m, n] = size(X);
centroids = zeros(K, n);

for i = 1:K
    select = find(idx==i);
    if isempty(select) == 0
        centroids(i, :) = mean(X(select, :));
    else
        end
end
end

end
```

## A.6 computedist.m

```
function J = computedist(X, centroids, idx)

K = size(centroids, 1);
m = size(X, 1);
```



```
for i = 1:K

    diffs = bsxfun(@minus, X(i == idx,:), centroids(i,:));
    distortion = sum(sum(diffs.^2, 2));

    holder(:,i) = distortion;

end

J = (1/m) * sum(holder);

end
```

## A.7 plotKmeans.m

```
function plotKmeans(idx, K)

figure; hold on;
Mrkr = '+oxsd^v><ph';
load('../flowdata/Ugs_Uls_speeds.mat');
for i = 1:K

    cluster = find(idx==i);
    plot(Ugs_Uls_speeds(cluster, 1), ...
         Ugs_Uls_speeds(cluster, 2), Mrkr(i), ...
         'LineWidth', 2.5, 'MarkerSize', 9)

end

xlabel('Ugs [m/s]', 'FontSize', 25)
ylabel('Uls [m/s]', 'FontSize', 25)
set(gca, 'xscale', 'log')
set(gca, 'yscale', 'log')
set(gca, 'XLim', [0.1 10])
set(gca, 'YLim', [0.1 10])
set(gca, 'XGrid', 'on')
set(gca, 'XMinorGrid', 'on')
set(gca, 'YGrid', 'on')
set(gca, 'YMinorGrid', 'on')
set(gca, 'GridLineStyle', '-')
set(gca, 'fontsize', 18)
set(gca, 'GridAlpha', 0.50)
set(gca, 'MinorGridAlpha', 0.40)
```

```
for i = 1:K
    legendnames{i} = ['Cluster' num2str(i)];
end
legenden = legend('show', legendnames);
set(legenden, 'FontSize', 18);

hold off;

end
```

# Appendix B

## Neural network Matlab code

Here is the matlab code for the neural network developed and used during the project. It is heavily influenced by the course: "Machine Learning" available on Coursera.org [10] The program also uses the function `fmincg.m` which is an advanced optimization function. It has not been included in the text version of the thesis, but it is available online. [21]

### B.1 NNmain.m

```
%%
%Initialization and choosing variables
clc, clear, close('all')
tic

%Main program for running the neural network with one hidden layer

%load data features and classifiers.
load('kmeans_id.mat');
load('eeg_id.mat');
%load('../flowdata/forstetall.mat');
load('../flowdata/CDF_500space.mat');
load('../flowdata/PDF_500space.mat');
load('../flowdata/per_sensor_data.mat');
load('../flowdata/per_sensor_data_sorted.mat');
load('../flowdata/Uslug_data.mat');
load('../flowdata/FFT_data.mat');
load('../flowdata/sensor_shift_diff.mat');

%Choose wether to test on train whole set, or random init
```

## APPENDIX B. NEURAL NETWORK MATLAB CODE

---

```
% 1 = random init, 0 = train on whole set
random_init = 0;

%X is the data chosen for training the algortihm
X = [Uslug_data];

%y is the identifiyer vector containing predetermined
%classification data
y = kmeans_id;
%y = Eeg_id;

%set lambda for regularization
lambda = 1;

%feature normalization, standard score.
%Makes is easier for the weights to adapt.
[X, mu, sigma] = zscore(X);

if random_init == 1
    %randomly chooses training and test data given a fraction
    %to be amount of training data.
    %The randomized index is also extracted for camparison
    [X ,y ,X_test ,y_test, ...
     train_randindex, test_randindex] = nnRandInit(X,y,0.70);
else
    %Train the whole set, then test it on itself
    X_test = X;
    y_test = y;
    test_randindex = [1 : size(X,1)]';
end

%Set the number of learning iterations
learning_iterations = 100;

%set the different layer sizes
input_layer_size = size(X,2);
number_hidden_layer = 50;
num_labels = max(y);

%%
%random initialization
initial_Theta1 = randInitializeWeights(input_layer_size, ...
```

```
        number_hidden_layer);
initial_Theta2 = randInitializeWeights(number_hidden_layer, ...
        num_labels);

%unroll parameters
initial_nn_params = [initial_Theta1(:); initial_Theta2(:)];

% ===== Training the Neural Network =====

options = optimset('MaxIter', learning_iterations);

costFunction = @(p) nnCostFunction(p, ...
        input_layer_size, ...
        number_hidden_layer, ...
        num_labels, X, y, lambda);

[nn_params, cost] = fmincg(costFunction, initial_nn_params, ...
        options);

Theta1 = reshape(nn_params(1:number_hidden_layer * ...
        (input_layer_size + 1)), ...
        number_hidden_layer, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (number_hidden_layer * ...
        (input_layer_size + 1))):end), ...
        num_labels, (number_hidden_layer + 1));

%%
% ===== Predict using the trained neural network =====

pred = predict(Theta1, Theta2, X_test);

Prediction_percentage = mean(double(pred==y_test))*100;

%This plots the test set, and marks the one that
%it failed to identify correctly.
%the index of the missed file is also output
miss = plotNN(pred,num_labels,test_randindex,y_test)
Prediction_percentage

toc
```

## B.2 nnRandInit.m

```
function [train_X, train_y ...
         test_X, test_y, train_randindex, ...
         test_randindex] = nnRandInit(X,y,trainsize)
%This randomly picks examples for training and test set

%Gets the sizes of the input matrix and vector
[Xm, Xn] = size(X);

%The number of examples used for training
train_amount = round(trainsize*Xm);

%consolidates X and Y for randomization, so that the
%indexes of X and y will remain the same after randperm
allData = [X y];

%Randomizes the index of the data
randomindex = randperm(size(allData, 1));

%gets the randomized data in a train and test set
train_set = allData(randomindex(1:train_amount), :);
test_set = allData(randomindex(train_amount+1:end), :);

%Gets original indexes of the training and test set
train_randindex = randomindex(:,1:train_amount)';
test_randindex = randomindex(:,train_amount+1:end)';

%extracts the X and y values from train and test sets
train_X = train_set(:,1:Xn);
train_y = train_set(:,Xn+1);
test_X = test_set(:,1:Xn);
test_y = test_set(:,Xn+1);

end
```

## B.3 randInitializeWeights.m

```
function W = randInitializeWeights(L_in, L_out)
% Randomly initializes weights based on the size of the
% input layer and the hidden layer
W = zeros(L_out, 1 + L_in);
```

```
epsilon_init = sqrt(6) / sqrt(L_in + L_out);  
W = (rand(L_out, 1 + L_in) * 2 * epsilon_init) - epsilon_init;
```

```
end
```

## B.4 nnCostFunction.m

```
function [J grad] = nnCostFunction(nn_params, ...  
                                   input_layer_size, ...  
                                   hidden_layer_size, ...  
                                   num_labels, ...  
                                   X, y, lambda)  
  
%Implementation of the cost function for this neural network  
  
% Reshape nn_params back into the parameters  
% Theta1 and Theta2, the weight matrices  
% for our 2 layer neural network  
Theta1 = reshape(nn_params(1:hidden_layer_size * ...  
                        (input_layer_size + 1)), ...  
                  hidden_layer_size, (input_layer_size + 1));  
  
Theta2 = reshape(nn_params((1 + (hidden_layer_size * ...  
                        (input_layer_size + 1))):end), ...  
                  num_labels, (hidden_layer_size + 1));  
  
% Setup some useful variables  
m = size(X, 1);  
  
%--- Part 1: implement forward propagation and calculate cost----  
  
% --- Forward propagation through the network ---  
% Convert the classifier (y) values into matrixes  
% containing 0 and ones for the correct indexes  
eye_matrix = eye(num_labels);  
y_matrix = eye_matrix(y,:);  
  
% Add a column of ones to X (the bias)  
a1 = [ones(m,1) X];
```

```
%Calculate the values in the hidden layer
z2 = Theta1 * a1';
a2 = [sigmoid(z2)]';
%Add the bias to the hidden layer
a2 = [ones(size(a2,1),1) a2];

%Calculate the values in the output layer
z3 = Theta2 * a2';
a3 = [sigmoid(z3)]';

h = a3;

%---- cost function calculation----
%Calculate the cost for the neural network with regularization

mpart1 = (- y_matrix .* log(h));
mpart2 = (1 - y_matrix) .* log(1 - h);
cost = ( (1/m) * sum(sum(mpart1 - mpart2, 2)) );

%compute regularization
regur = (lambda / (2*m) ) * ( sum(sum(Theta1(:,2:end).^2, 2)) ...
    + sum(sum(Theta2(:,2:end).^2, 2)) );

J = cost + regur;

% ----- Backpropagation-----%

%compute the error term in the output layer.
d3 = a3 - y_matrix;

%Compute the error term in the hidden layer.
d2 = ( d3 * Theta2(:,2:end) ) .* sigmoidGradient(z2');

%Compute accumulated gradient
Delta1 = d2' * a1;

Delta2 = d3' * a2;

Theta1_grad_un = Delta1 * (1/m);
Theta2_grad_un = Delta2 * (1/m);
```



```
%Regularization of the gradient-----
Theta1(:,1) = 0;
Theta2(:,1) = 0;
Theta1 = Theta1 * (lambda / m);
Theta2 = Theta2 * (lambda / m);

Theta1_grad = Theta1_grad_un + Theta1;
Theta2_grad = Theta2_grad_un + Theta2;

% -----

% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:)];

end
```

## B.5 sigmoid.m

```
function g = sigmoid(z)
% computes the sigmoid function of z

g = 1.0 ./ (1.0 + exp(-z));
end
```

## B.6 sigmoidGradient.m

```
function g = sigmoidGradient(z)
% Computes the gradient of the sigmoid function

g = sigmoid(z) .* (1 - sigmoid(z));

end
```

## B.7 plotNN.m

```
function miss = plotKmeans(pred, num_labels, test_randindex, y_test)
```

```
figure; hold on;
Mrkr = '+osd^xv><ph';
load('../flowdata/Ugs_Uls_speeds.mat');

Ugs_Uls_speeds_hit = Ugs_Uls_speeds(test_randindex,:);

for i = 1:num_labels

    label = find(pred==i);
    plot(Ugs_Uls_speeds_hit(label, 1), ...
         Ugs_Uls_speeds_hit(label, 2), Mrkr(i), ...
         'LineWidth', 2.5, 'MarkerSize', 9)

end

miss = test_randindex(pred~=y_test);

plot(Ugs_Uls_speeds(miss,1), ...
     Ugs_Uls_speeds(miss,2), 'xr', 'LineWidth', 2.5, ...
     'MarkerSize', 9)

xlabel('Ugs [m/s]', 'FontSize', 25)
ylabel('Uls [m/s]', 'FontSize', 25)
set(gca, 'xscale', 'log')
set(gca, 'yscale', 'log')
set(gca, 'XLim', [0.1 10])
set(gca, 'YLim', [0.1 10])
set(gca, 'XGrid', 'on')
set(gca, 'XMinorGrid', 'on')
set(gca, 'YGrid', 'on')
set(gca, 'YMinorGrid', 'on')
set(gca, 'GridLineStyle', '-')
set(gca, 'fontSize', 18)
set(gca, 'GridAlpha', 0.50)
set(gca, 'MinorGridAlpha', 0.40)

for i = 1:num_labels
    legendnames{i} = ['Classification' num2str(i)];
end
legenden = legend('show', legendnames);
set(legenden, 'FontSize', 18);

hold off;

end
```

# Appendix C

## Plotting data from capacitance sensors

This following matlab code is based upon a program originally written by Time, and was provided at the start of the project. The program is discussed in Chapter 3.1

```
%=====
%           SLUG FLOW (PART 1)
%=====
% Made from "Slug_flow_1.m"

clc
clear
close all

% ***** SENSOR DISTANCES:*****
    L16=.102+.081+.062+.082+.102;
    L25=.081+.062+.082;
    L34=.062;
% *****

load('scan_frequency.mat');
origfolder = cd
cd('Sensor data')
filenames = dir('*.txt');

%Chose file number from the list of files in Sensor data
%from 1 to 166
filenumber = 60
fname =filenames(filenumber).name
```

```
nhead = 40; % number of lines of header information
ncols = 6; % number of columns in the data file

[labels,V] = readColCapData(fname,ncols,nhead);
M = length(V(:,1))

Tid = 1:M;

dt = scan_frequency(filename)

Tid = Tid*dt;

L1 = 1
L2 = 5000 % number of points

figure(1) % Plot Capacitance sensors - time trace

subplot(3,1,1); plot(Tid(L1:L2),V(L1:L2,1),'b-');
hold on
plot(Tid(L1:L2),V(L1:L2,6),'r-');
xlabel('Time (s)','FontSize',15); % add axis labels and plot title
ylabel('Sensor signal (0 - 10000)','FontSize',15);
title('Capacitance sensors - time trace','FontSize',14);
legend('Sensor 1','Sensor 6',1);
set(gca, 'fontsize', 12)

subplot(3,1,2); plot(Tid(L1:L2),V(L1:L2,2),'b-');
hold on;
plot(Tid(L1:L2),V(L1:L2,5),'r-');
xlabel('Time (s)','FontSize',15);
ylabel('Sensor signal','FontSize',15);
legend('Sensor 2','Sensor 5',1);
set(gca, 'fontsize', 12)

subplot(3,1,3); plot(Tid(L1:L2),V(L1:L2,3),'b-');
hold on;
plot(Tid(L1:L2),V(L1:L2,4),'r-');
xlabel('Time (s)','FontSize',15);
ylabel('Sensor signal','FontSize',15);
legend('Sensor 3','Sensor 4',1);
set(gca, 'fontsize', 12)
```

```
% ***** FFT ANALYSIS *****
%
% *****
for i = 1 : 6
ISE=i; %Sensor number
L = M;
sensordata(:,i) = V(:,ISE);
y(:,i) = sensordata(:,i) - mean(sensordata(:,i));

Fs = 1/dt;
NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y(:,i) = fft(y(:,i),NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2);

end

for i = 1:3

figure(2) % Plot Single-sided amplitude spectrum.

sens16 = [1,6];
sens25 = [2,5];
sens34 = [3,4];
if i == 1
    sensor = sens16;
elseif i == 2
    sensor = sens25;
else
    sensor = sens34;
end

subplot(3,1,i); semilogx(f,2*abs(Y(1:NFFT/2,i)),'-r')
hold on
semilogx(f,2*abs(Y(1:NFFT/2,7-i)),'-b')
hold off
title(['Single-Sided Amplitude Spectrum of y(t) sensor ' ...
    num2str(sensor(1,1)) ' and ' num2str(sensor(1,2))])
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
legend(['Sensor ' num2str(sensor(1,1))], ['Sensor ' ...
    num2str(sensor(1,2))],1)
end

%*****
% PDF histogram
```

```
%*****  
  
for i = 1:3  
figure(4) % Histogram Probability density function  
sens16 = [1,6];  
sens25 = [2,5];  
sens34 = [3,4];  
if i == 1  
    sensor = sens16;  
elseif i == 2  
    sensor = sens25;  
else  
    sensor = sens34;  
end  
  
subplot(3,1,i); hist(sensordata(:,sensor),100)  
title(['Probability density function sensor ' ...  
    num2str(sensor(1,1)) ' and ' num2str(sensor(1,2))])  
xlabel('Sensor reading')  
ylabel('Number of counts')  
legend(['Sensor ' num2str(sensor(1,1))], ['Sensor ' ...  
    num2str(sensor(1,2))],1)  
end  
  
%*****CROSS CORRELATION FUNCTION*****  
%  
%*****  
for j = 1:3  
x(:,j)=V(L1:L2,j); % 1 function (1 sensor)  
y(:,j)=V(L1:L2,7-j); % 2 function (6 sensor)  
  
N=length(Tid(L1:L2));  
yy(j,:)=y(:,j)';  
ntau=200  
for i=1:ntau  
yy(j,:) = circshift(yy(j,:), [0,+1]);  
CCF(i,j) = yy(j,:)*x(:,j)/N;  
end  
  
figure(5) % Plot Cross correlation function  
  
plot(CCF)  
xlabel('Time (ms)'),  
ylabel('Correlation')
```

```
title('Cross correlation function')
legend('sensor 1&6', 'sensor 2&5', 'sensor 3&4',1)

maxccf=max(CCF(:,j))

for i=1:ntau
    if CCF(i,j)==maxccf
        tshift(1,j)=i        % Time shift
    end;
end

if j == 1
    sensor = L16;
elseif j == 2
    sensor = L25;
else
    sensor = L34;
end

Uslug(1,j)=sensor/(tshift(1,j)/(1/dt)) % Slug velocity

fin = fopen(fname,'r');
for i=1:9, buffer = fgetl(fin); end

Uls=str2double(buffer(16:28)) % Superficial liquid velocity
Ugs=str2double(buffer(40:52)) % Superficial gas velocity
Umix=Uls+Ugs;                % Mixture velocity
Ratio(1,j)=Uslug(1,j)/Umix;
end

Uslug
Ratio
Umix
dt
cd(origfolder);
```