01SOVBH - Statistical learning and neural networks

Statistical Signal Processing
Computer Lab 1: K-NN classifier

Group Members
Lorenzo Bellone
Victor De Castro Morini

Turin, February 5, 2019

Contents

List of Figures

Acronyms

KNN K-Nearest Neighbors.

lab Laboratory Lesson.

MAP *Maximum a Posteriori*.

MN Machine Learning.

RSSI Received Signal Strength Indicator.

SW Software.

1 Introduction

The purpose of this laboratory is to implement MATLAB scripts in order to explore the k-NN classifier method.

The main two types of Machine Learning (MN) are: 1) **predictive or supervised**, which is the mostly widely used in practice and it is used for solving prediction problems, and 2) **descriptive or unsupervised**, for finding good representations. On this lab we are exploring the first, therefore the goal is to learn a mapping from inputs **x** to outputs **y**, given a labeled set of inputs-output pairs (x,y) where, $D = (x_i, y_i)_{i=1}^N$.

D is called the **training set**, **N** is the **number of samples**, **x** is the **regressors/features/attributes/covariates**, where each **xi** is a **D**-dimensional vector of numbers, representing, say, the height and weight of a person; and **y** is the **output/regressand/label or response variable**.

The output can be divided in two kinds of variables: 1) **categorical/nominal variable** from some finite set, as male and female or any other finite set where there is no intrinsic ordering, and 2) when the output is a real value, as income level. When **categorical**, the problem is known as **classification or pattern recognition**, otherwise, **regression**.

The core of this lab is to implement K-Nearest Neighbors (KNN) algorithm with euclidean distance, which is a **predictive or supervised** MN type. For the implementation, the following SW were used:

- Software MATLAB 2018b;

  [Toolbox] Parallel Computing Toolbox;

2 Exercise 1 – Synthetic dataset

In the first Statistical Signal Processing Lab it is required to apply a specific classification method, called KNN (non-parametric classifier/algorithm), in different contexts. The aim of the method is to classify a test input given the class the **K** nearest points of a train set belong to. The distance between different points can be evaluated in many ways, in these exercises the Euclidean distance will be always applied. Once that the value of **K** has been chosen, the probability that a point belongs to a certain class is given by the fraction between the number of the **K** nearest training points belonging to that class and the value of **K**.

Two datasets are given in the first exercise: the first one represents the training points while the second one the test data. The data can belong to two possible classes that will be identified as "Class 1" and "Class 2", therefore we can classify this problem as: **supervised learning - binary classification**, as the output, y ϵ {1,...,C}, with C being the number of classes and equal to 2.

The aim of the exercise is to classify the test and the training data through the KNN classifier method and to derive the accuracy of the method (Figure 1) using the real class of each sample. For each value of K the error committed will be evaluated by observing how many times the prediction of the class was wrong.

It is useful to apply the KNN classifier also on the train set in order to avoid overfitting. Thus, when **K**=1, the method makes no error in the predictions related to the training part because each point is compared with itself, but it could lead to a large error in the prediction of the test data. On the contrary, a value of K that is closer to the total number of samples leads to underfitting, which means that the method is taking into account not only the closest points. This causes a larger misclassification rate.

The distribution of the samples for both the training and the test points (Figure 2) shows that the clusters of the test set are more well defined with respect to the train set. This could

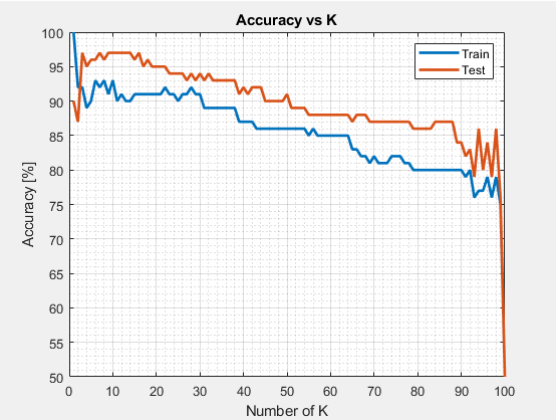

Figure 1: Accuracy of the KNN classifier as a function of K evaluated for the train and the test sets.

be the reason of the larger misclassification rate obtained through the application of the KNN classifier on the train set.

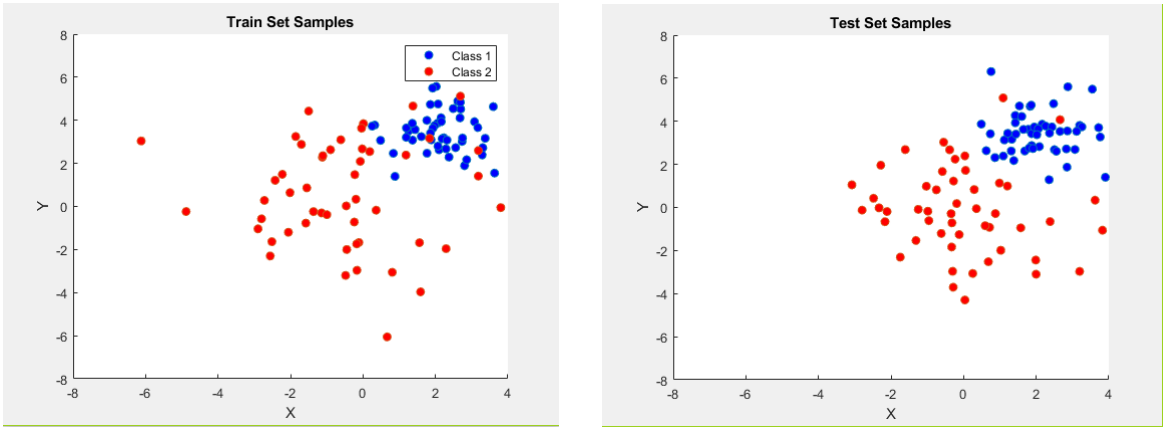

Figure 2: The distribution of the training set (on the left) and the test set (on the right) data in a 2-D space, according to their features (X & Y), and each correspondent class.

3 Exercise 2 – Phoneme recognition

In the second exercise, again a real dataset is used, but this time containing 5404 speech samples. Each one of these samples is characterized by five features that consist on the normalized amplitudes of the five first harmonics of the speech signal. The aim of this exercise is to distinguish between the nasal and the oral sounds given the features of each sample, therefore we can characterize this problem as **supervised learning - binary classification**.

The classification method applied is the KNN (non-parametric classifier/algorithm) previously explained. The dataset is divided into a training and a test set. The samples of the training set are placed in a 5-dimensional space according to their features. The class every training point belongs to is already known. Then the Euclidean distance between each point of the test set and the **K** nearest neighbors is evaluated. The class to which the majority of the K nearest points belongs is defined as the predicted class of the test point.

Then, it is necessary to evaluate the accuracy of the method, which means to assess how

many times the predicted class corresponds to the real one given by the test set. The accuracy strictly depends on the chosen value of K. **In this case the accuracy of the KNN classifier is slightly worse with respect to the previous exercise since this method does not assure optimal performances with a high dimensional input. This problem is known as curse of dimensionality.**

The accuracy of the method for this exercise is shown in Figure 3. The optimum value of **K** appears to be around thirty-five because, for the test set, the accuracy stops to increase at this point. The values of K bigger than fifty have not been considered due to the heavy computations, which can be soften using the parallel toolbox of MATLAB, e.g. **parfor** command[1].

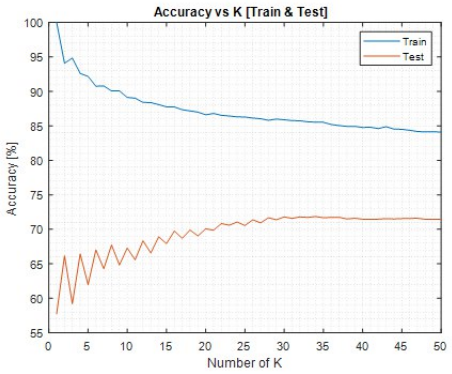

Figure 3: Accuracy of the KNN classifier as a function of K evaluated for the train and the test sets.

4 Exercise 3 - User localization from RSSI

Consider a 2D space divided in 24 sections/cells surrounded by 7 base stations/sensors and a transmission device in the center of one of the cells. Each sensor provides a Received Signal Strength Indicator (RSSI) measurement for each transmission device, therefore 7 RSSI measurements, called data packet, to describe the position of each transmission device. In order to make it more accurate, 5 data packets, were obtained for each transmission device/cell. The Figure 4 describes this idea schematically.

Two datasets are provided, train and test. The first one contains 5 data packets for each cell, and the second provides data packets of random cells. The task is to create a fusion center (also called **sensor fusion**), which will apply a KNN classifier and return the probability of which cell each data packets from test dataset belongs to.

The train dataset (D) is provided and it contains a labeled set of inputs-output pairs (x,y), where **x** is the data packets and **y** the cells, hence this problem is **predictive/supervised MN type**. In addition, the output can only belong to 24 classes, meaning that this is a **multiclass classification** problem.

The performance was evaluated in two ways: 1) Accuracy[%] when comparing the output of the evaluated set with the given train dataset output; 2) Accuracy[%] when comparing the output of the evaluated set and its **neighbors**[3x3] with the given train dataset output. The Figure 5 shows the performance of the classification, when varying **K**, the considered number of nearest neighbors.

[1]https://it.mathworks.com/help/distcomp/parfor.html

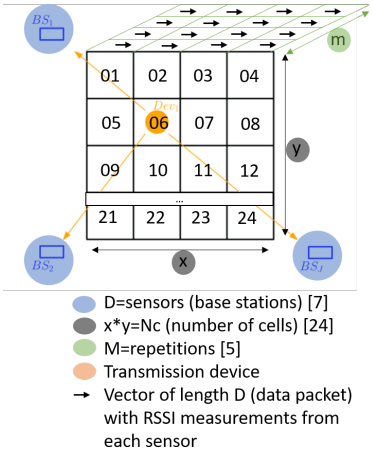

Figure 4: Exercise 3 schematic.

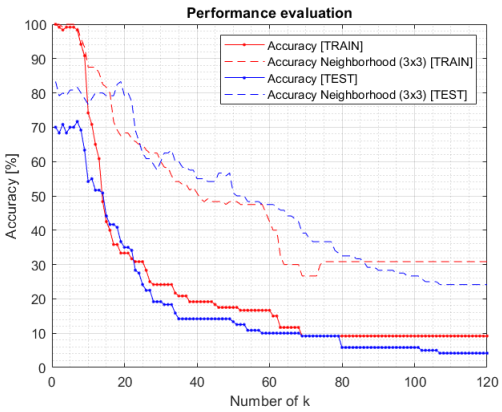

Figure 5: Performance evaluation of the KNN classifier for varying K value.

References

[1] Statistical Signal Processing A.A. 2017/2018 Computer Lab 1 – k-NN classifier - Statistical learning and neural networks

01SOVBH - Statistical learning and neural networks

Statistical Signal Processing Computer Lab2 – Model fitting and classification

Group Members
Lorenzo Bellone
Victor De Castro Morini

Turin, February 5, 2019

Contents

List of Figures

Acronyms

BoW Bag of Words.

lab Laboratory Lesson.

LDA Linear Discriminand Analysis.

MAP Maximum A Posteriori Estimation.

MLE Maximum Likelihood Estimation.

MN Machine Learning.

MVN Multivariate Normal Distribution, also known as Multivariate Gaussian Distribution or Joint Normal Distribution.

NBC Naive Bayes Classifier.

PDF Probability Density Function.

QDA Quadratic Discriminant Analysis.

1 Introduction

In the second Statistical Learning lab it is required to deal with continuous distribution of data. The aim of the Lab is to classify data that are characterized by many continuous features. In order to do this, these features have to be considered Gaussian distributed, characterizing the exercise as Multivariate Normal Distribution (MVN).

- SOFTWARE MATLAB 2018b;

2 Exercise 1 - Model fitting for continuous distributions: Multivariate Gaussian (MVN)

In the first exercise a real dataset is employed: it contains labeled data for two classes (male and female). Each sample is composed by three columns: the height(cm), the weight(kg) and the class(1=Male, 2=Female). Height and weight are considered Gaussian variables, and the class is a discrete random variable.

The task of this exercise is to fit a class-conditional Gaussian multivariate distribution to the data in order to plot two unique Probability Density Function (PDF), one for each class (male and female).

At the beginning it was useful to separately plot the histogram of the height and weight features of each gender, in order to compare the data of both classes and **visually check** if they are suitable to be fitted in a multivariate Gaussian distribution (Figure 1).

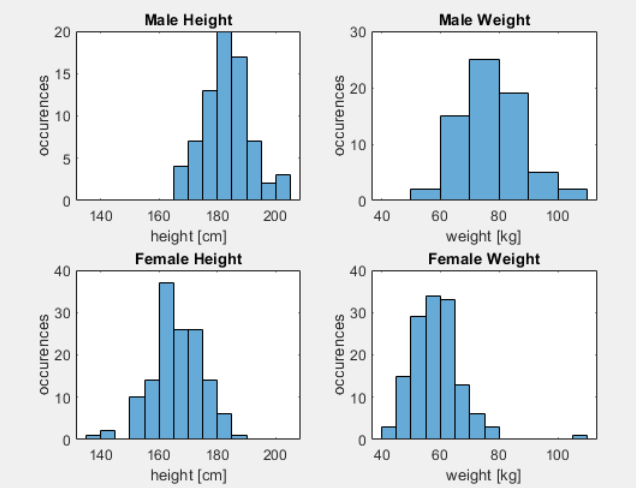

Figure 1: Histograms of the occurrences of the feature among the samples, according to the gender.

Then, it was necessary to evaluate the Maximum Likelihood Estimation (MLE) of the mean, μ_{MLE}, and the covariance matrix, $\sum_{MLE}$. The equations for that are depicted in Figure 2.

$$\boldsymbol{\mu}_{MLE} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x_i}$$

$$\sum\nolimits_{MLE} = \frac{1}{N}\sum_{i=1}^{N} (\boldsymbol{x_i} - \boldsymbol{\mu}_{MLE})(\boldsymbol{x_i} - \boldsymbol{\mu}_{MLE})^T$$

Figure 2: Maximum Likelihood Estimation equations.

Once these parameters are known, the joint probability density function of height and weight can be shown (Figure 3).

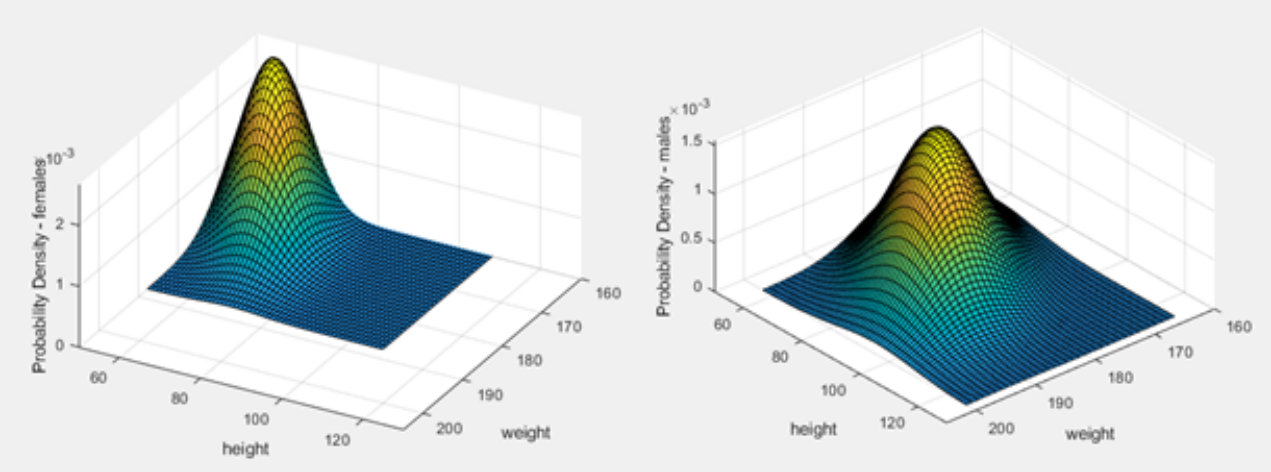

Figure 3: Joint Multivariate Gaussian Distribution of weight and height according to the gender.

By comparing the histograms in Figure 1, it can be noticed that the higher numbers of occurrences match with the point in which the joint probability density function (PDF) has its maximum. In conclusion, the continuous features of these samples are well-suited in a multivariate Gaussian distribution.

3 Exercise 2 – Model fitting for discrete distribution: Bag of Words (BoW)

The task of the second exercise is to implement a Naïve Bayes Classifier (NBC) in order to classify a dataset called XwindowsDocData.mat, which contains features for two classes. In the dataset there are documents concerning Microsoft Windows and others concerning X Windows. Each feature represents the presence or absence (1 or 0) of a word in a document. In the training set there are 900 documents and 600 features (words) for each document.

The Naïve Bayes Classifier is an approach to classify D-dimensional vectors of discrete-valued features. Technically, to do so, the class-conditional distribution is needed but, in this case, an assumption is made: all the features are statistically independent (Bayes' theorem). According to this assumption, the parameters of the Naïve Bayes Classifier are:

- The prior probabilities: the probability of each class regardless of any prior information. This is evaluated as depicted on Figure 4.

$$\pi_j = {}^{Nc}/_{N}$$

Figure 4: Prior probability(π_j), where Nc is the number of samples belonging to class C while N is the total number of samples

- The probability of each feature (f) given the class (c), θ_{jc}, is depicted on Figure 5.

$$\theta_{jc} = p(x_j = 1|y = c)$$

Figure 5: Probability of each feature (f) given the class (c), θ_{jc}.

In order to calculate it, an approximation of θ_{jc}, given the type of exercise, is shown on Figure 6, where N_{jc} is the number of features belonging to class "present" or "absent" and N the number of samples.

$$\theta_{jc} = \frac{N_{jc}}{N}$$

Figure 6: Approximation of the probability of each feature (f) given the class (c), θ_{jc}, where N_{jc} is the number of features belonging to class "present" or "absent" and N the number of samples.

Once these parameters are evaluated, the class-conditional densities can be displayed (Figure 7), which means to plot the θ_{jc} for both classes to have an idea of which words are more likely in a document rather than the other.

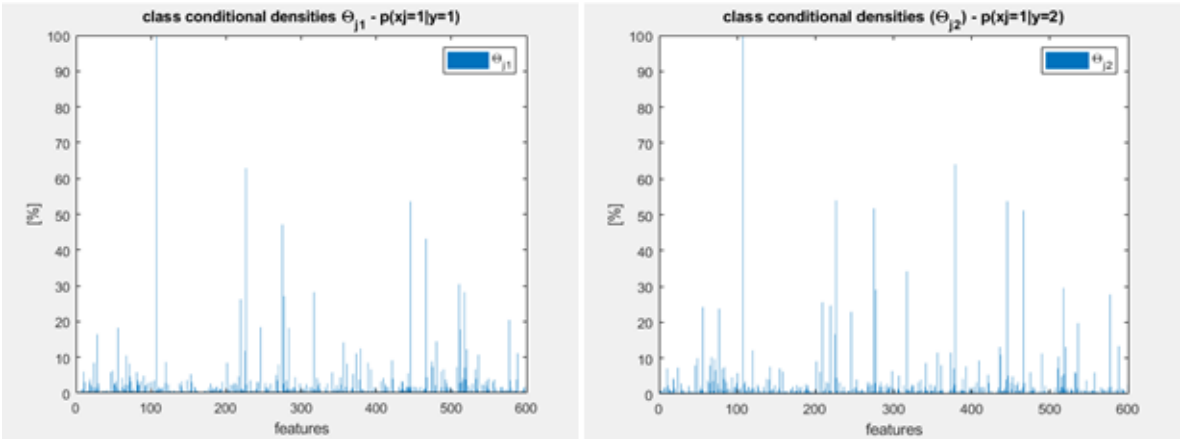

Figure 7: Class-conditional densities for both the classes (male and female).

It is also important to recognize the uninformative features, which are words like "and" or "the" that are equally likely in both the type of documents. The classifier will work better by removing these types of words, since the probability of finding that word in a document is the same for both the cases, and hence, it does not bring information about the possible class of the document. Sometimes it can be helpful to perform a sort of feature selection, in order to remove irrelevant features that do not help in the classification.

The relevance of each feature must be evaluated separately, and then the most discriminative features are taken (based on some trade-off between accuracy and complexity) while the other are discarded.

4 Exercise 3 – Classification - discrete data

This exercise is a continuation of Section 3, which the following was calculated from the **train data**:

- The prior probabilities of each class. $\pi_c = p(c)$;
- The class-conditional probabilities of each feature, $\theta_{jc} = p(x_j = 1|c)$.

The focus now rests on the **test data**, which will be classified according to the NBC model, previously parametrized with the train data on Section 3.

The next step is to calculate the Maximum A Posteriori Estimation (MAP) of the class, of each document contained in the test data. The MAP classifier details are presented on Figure 8.

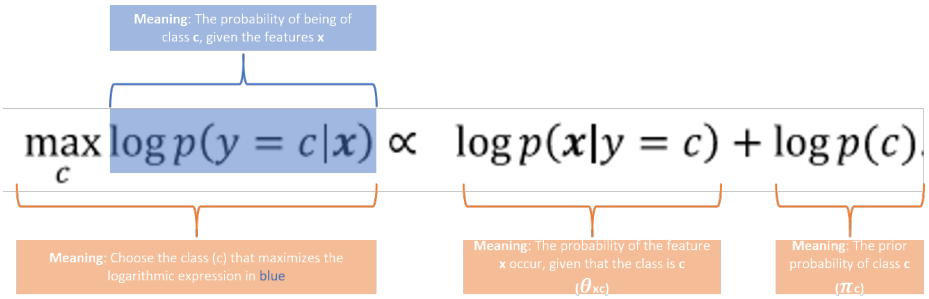

Figure 8: MAP classifier detailed.

One of the characteristics of the NBC is that the features are **assumed** to be **statistically independent** from each other, therefore: $p(x|y = c) = \prod_{j=1}^{D} p(x_j|y = c)$. Parsing it and putting it in layman's terms, $p(x|y = c)$ is the probability of the set of features **x** to occur given that the class is **c**; $\prod_{j=q}^{D}$ is the loop multiplication of the term in green, starting on 1 and finishing on **D**(total number of features); $p(x_j|y = c)$ is the θ_{jc} calculated on Section 3, which is defined as the class-conditional probabilities of each feature.

In order to avoid underflow[1], the logarithm was applied in this calculation. Working without logarithm, makes necessary to multiply probabilities θ_{jc}, and as these probabilities are usually less than 1, it can lead to underflow.

Applying the classifier(trained with the train data) in the test data, and comparing it with the provided true class of test data, it was obtained an accuracy of 68.2%. Applying it to the train data, 92.8%.

Applying the classifier(trained with the test data) in the train data, and comparing it with the provided true class of train data, it was obtained an accuracy of 73.2%. Applying it to the test data, 91.4%.

5 Exercise 4 – Classification - continuous data

This exercise employs the height/weight data already employed on Exercise 1.

[1]condition in a computer program where the result of a calculation is a number of smaller absolute value than the computer can represent.

The entire data set containing 210x3 matrix, where 210 are the samples, and each sample contains the height, weight and class (male or female) columns.

This data set was divided into test data and train data. The test data contains 25 males and 40 females from the data set, and the train data contains the rest of data set, i.e. $210-(25+40) = 145$ samples.

In this exercise, the test data was fit into three versions of Gaussian Discriminative Analysis[2], classified accordingly and evaluated the accuracy.

The three evaluated Gaussian Discriminative Analysis are: 1) Two-class Quadratic Discriminant Analysis (QDA); 2) Two-class Quadratic Discriminant Analysis (QDA) with diagonal covariance matrices and 3) Two-class Linear Discriminant Analysis (LDA), also knows as Fisher discriminant, after its inventor Sir R. A. Fisher.

The Discriminant analysis is a classification method which assumes that different classes (Male/Female) generate data based on different Gaussian distributions. In other words, the model assumes that the data has a gaussian mixture distribution (also knows as Gaussian Mixture Model (GMM)).

Two Discriminant analysis are exploited in this exercise: LDA and QDA. The first one, the model has the same covariance matrix for each class, only the means vary. And for the second one, both means and covariances of each class vary.

5.1 Two-class Quadratic Discriminant Analysis (QDA)

For the Two-class Quadratic Discriminant Analysis, it was computed the sample mean of each class (μ_c), the sample covariance of each class (Σ_c) and the prior probabilities of each class ($\pi_c = p(c)$), since on this exercise the prior probability are not considered uniform. With these information, it was possible to fit in the Gaussian density yields, depicted on Figure 9.

$$p(y = c|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{p(y = c)p(\boldsymbol{x}|y = c, \boldsymbol{\theta})}{\sum_{c'} p(y = c')p(\boldsymbol{x}|y = c', \boldsymbol{\theta})}$$

$$= \frac{\pi_c |2\pi\boldsymbol{\Sigma}_c|^{-1/2} \exp[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_c)]}{\sum_{c'} \pi_{c'} |2\pi\boldsymbol{\Sigma}_{c'}|^{-1/2} \exp[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_{c'})^T \boldsymbol{\Sigma}_{c'}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_{c'})]}$$

Figure 9: Full Posterior Distribution.

After classifying each of the tests samples, by making a comparison between $p(y = Male|x, \theta)$ and $p(y = Female|x, \theta)$ and choosing the maximum among both (MAP), it was compared with the true class of the tests samples. The accuracy of this classifier was: 89.2%.

5.2 Two-class Quadratic Discriminant Analysis (QDA) with diagonal covariance matrices

The same steps of Subsection 5.1 were used, but this time the sample covariance matrices were transformed in diagonal matrices.

[2]Discriminant analysis is a classification method. It assumes that different classes generate data based on different Gaussian distributions. More precisely, each class (Male/Female), generates data using a Multivariate Normal Distribution (MVN).

After classifying each of the tests samples, by making a comparison between $p(y = Male|x, \theta)$ and $p(y = Female|x, \theta)$, it was compared with the true class of the tests samples. The accuracy of this classifier was: 87.7%.

5.3 Two-class Linear Discriminant Analysis (LDA)

For this task, the sample covariance matrices are the same ($\Sigma_c = \Sigma$), and different means (μ_1 and μ_2, or μ_{male} and μ_{female}, respectively). This time, the accuracy was: 89.2%.

References

[1] Statistical Signal Processing A.A. 2017/2018 Computer Lab 2 – Model fitting and classification

01SOVBH - Statistical learning and neural networks

Statistical Signal Processing Computer Lab3 – Principal component analysis (PCA)

Group Members
Lorenzo Bellone
Victor De Castro Morini

Turin, February 5, 2019

Contents

List of Figures

Acronyms

lab Laboratory Lesson.

LDA Linear Discriminand Analysis.

MSE Mean Squared Error.

PCA Principal Component Analysis.

1 Introduction

In the third Statistical Learning lab[1] it is required to apply Principal Component Analysis (PCA) to a given dataset. The aim of PCA is to find a small set of features that explains well the data, i.e. compressing the data and hence reducing its dimension. This technique if often used in image compression.

In more details, the given dataset consists of a hyper-spectral image of a field, the first one sizing 145x145x220 (x, y, z axis respectively), i.e. an image with 145x145 pixels, where each pixel contains 220 spectral bands. In additional, each pixel belongs to 1 out of 16 classes, and each class represents 1 type of crop.

On the overview, there are $145x145 = 21025$ samples, where each sample contains 220 features and 1 class out of 16 possible ones. Applying the PCA will reduce the amount of features, leaving only the principal ones, i.e. the ones that **clusters** best the classes or separate best the classes. After reducing the features, it should be possible to obtain a good approximation of the original data using them.

• Software MATLAB 2018b;

Live Editor

2 Exercise 1 – PCA

A given dataset: *IndianPines*, containing a hyper-spectral image of dimensions 145[pixels]x145[pixels]x220[spectral bands] was employed.

Statistically there are $145x145 = 21025$ samples, where each sample contains 220 features and 1 class out of 16 possible ones.

For this exercise only 2 classes were randomly chosen: 2 and 14, to perform the PCA, and hence reduce the dimensionality. Therefore all the pixels classified as 2 and 14 composed a new dataset(called "whole") of 2856 samples x 220 features, being 1428 samples of class 2 and 1428 of class 14.

From this new dataset(called "whole"), it was calculated the sample covariance matrix (with class 2 and 14 together), obtaining a 220x220 matrix. From this square matrix, it was obtained 220 eigenvalues and 220 eigenvectors.

The final number of dimensions, K, needs to obey 0¡K¡220(number of dimensions of the dataset), and for this example, K value was randomly chosen 50.

The most important eigenvectors are those with higher eigenvalues, therefore a new dataset, W, was created only with the K eigenvectors with the largest magnitude eigenvalues. The matrix W is called Feature Vector, and the eigenvector with largest magnitude, First Principal Component.

The PCA coefficients were computed using W(Feature Vector) and the dataset "whole", using the equation depicted on Figure 1.

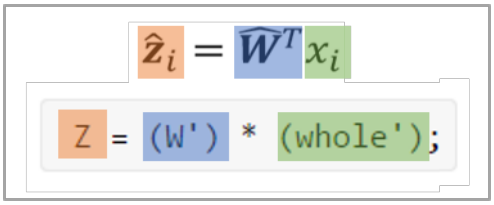

Figure 1: Calculation of PCA coefficients. The above equation is on algebraic form, where the orange variable represents the final dataset, i.e. the original dataset in terms of the K eigenvectors, instead of in terms of 220 features. Below the algebraic equation, it is represented the same but in Matlab language. The colors of the terms indicate the correlation between the algebraic equation and the Matlab equation.

Similarly it is possible to get the old data back, but it is only going to be identical to the previous matrix "whole" if all the eigenvectors of its sample covariance matrix were copied to Feature Vector W. Hence, as only K eigenvectors were considered, some information were lost, therefore it is not possible to retrieve exactly the same data. From Figure 1, it is possible to infer the back way in Matlab: $Z = (W)' * (whole)' \rightarrow \widehat{whole}' = (W')' * Z \rightarrow \widehat{whole}' = W * Z$; remembering that $W' = W_T$ in Matlab.

From the data retrieved, is was possible to calculate the Mean Squared Error (MSE) between the original matrix $whole$ and $\widehat{whole}$, by the formula

$$MSE = \frac{\left\| whole - \widehat{whole} \right\|^2}{N}$$

where N is the total number of subtracted pairs on the numerator. The MSE for K=50 is 252.91.

The same previous steps were executed in order to get all possible K and their correspondent MSEs. The result can be seen on Figure 2.

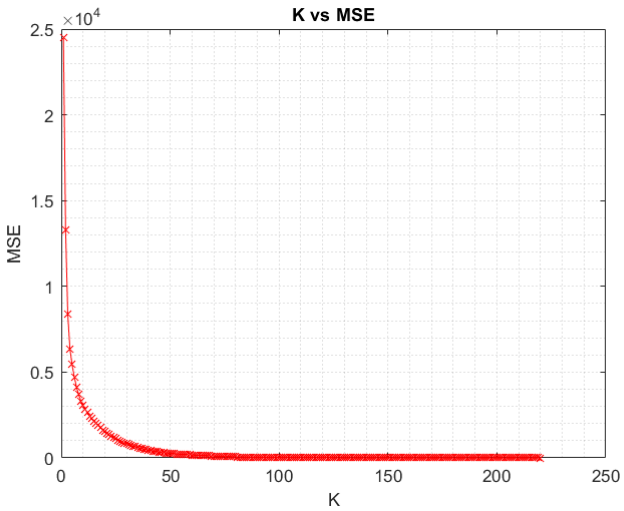

Figure 2: MSEs and respective Ks.

For the last topic, the eigenvectors corresponding to the 3 largest eigenvalues were plotted in 2D plot, depicted on Figure 3.

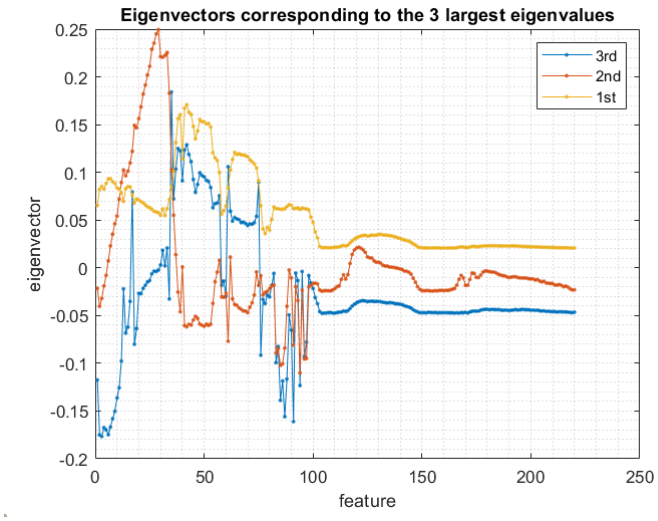

Figure 3: MSEs and respective Ks.

References

[1] Statistical Signal Processing A.A. 2017/2018 Computer Computer Lab 3 – Principal component analysis.

01SOVBH - Statistical learning and neural networks

Computer Lab – Statistical Learning and Neural Networks

Group Members
Lorenzo Bellone
Victor De Castro Morini

Turin, February 5, 2019

Contents

List of Figures

Acronyms

lab Laboratory Lesson.

1 Part 1: Tensow Flow

TensorFlow was used in the lab concerning the training of Neural Networks.

TensorFlow is an open-source library developed by Google for numerical computations. It is mainly used to build computational graphs, which represent the mathematical operations that are performed. A computational graph is a sequence of mathematical operations connected to each other as a graph of nodes.

The goal of this lab is to build a neural network using TensorFlow to perform a handwritten digit recognition. The input of the network is the MNIST dataset, composed of images of handwritten digits. More precisely, there are 60000 training images and 10000 testing images. Every image is grayscale with size 28x28. The desired output "y" is a ten-dimensional vector called "one-hot vector". This means that the vector has ten entries, as the number of possible classes (from 0 to 9), and they are all zero except for one entry that is set to one in the position corresponding to the correct digit.

Another useful tool of TensorFlow is Tensorboard. This is a graphical interface that helps to understand, debug and optimize the TensorFlow program. Thanks to this tool, it can be possible to keep track of many parameters, for example the loss function during the training or the change of weights and biases during the algorithm.

In the next paragraph the most useful functions of TensorFlow will be explained and applied to the proposed task of this lab.

1.1 The program

The first thing to do is always to define the "placeholders". The placeholders are used to provide the inputs of the neural network that will be filled by some values provided during the runtime. Since this is a supervised classification problem, two kinds of inputs will be necessary: the images that have to be classified and the labels corresponding to every possible class of the images.

```
x = tf.placeholder(tf.float32, shape = (None, NUM_PIXELS), name = 'input_images')

y = tf.placeholder(tf.float32, shape = (None, NUM_CLASSES), name = 'hot_vector')
```

The first is the "x" placeholder with shape $(None, NUM_PIXELS)$, where NUM_PIXELS is the number of pixels in a single image (in this case equal to 784) and None allows to use the code for any arbitrary value of a batch size, which is the number of images that will be picked at every iteration of the training phase.. For the other placeholder "y", the label has size $(None, NUM_CLASSES)$, where $NUM_CLASSES$ is the number of the values that each possible digit can assume.

Once that the inputs are defined, it is necessary to define the variables, which are the weights and the biases used in the layers of the network (in this case the number of layers required is 3). These variables can be defined in the following way:

```
W1 = tf.Variable(tf.truncated_normal(shape = (NUM_PIXELS, HID_1),
stddev = 1/NUM_PIXELS), name = 'trainable_weights1')

b1 = tf.Variable(tf.constant(0.1, shape = (HID_1,)),
name = 'trainable_biases1')

W2 = tf.Variable(tf.truncated_normal(shape = (HID_1, HID_2),
stddev = 1/NUM_PIXELS), name = 'trainable_weights2')
```

```
b2 = tf.Variable(tf.constant(0.1, shape = (HID_2,)),
name = 'trainable_biases2')


W3 = tf.Variable(tf.truncated_normal(shape = (HID_2, NUM_CLASSES),
stddev = 1/NUM_PIXELS), name = 'trainable_weights3')


b3 = tf.Variable(tf.constant(0.1, shape = (NUM_CLASSES,)),
name = 'trainable_biases3')
```

W1 is a variable with a gaussian distribution (called with the function tf.truncated.normal) and with a small standard deviation in order to have a stable initialization and to avoid the divergence of the network.

The shape is composed by the number of pixels and the number of hidden neurons in the first layer, which in this case is set to 100. W1 takes as input a row vector with 784 entries and it maps this row vector into a new vector that has 100 entries. Then there is the first bias, b1, which has to be summed to the result of the multiplication between the weight and the inputs. For this reason, it must have the same shape of the number of hidden layers. The logic behind the other variables is the same. The last layer takes an input that has size HID_2, in this case is set equal to 50, and it will produce an output, which is the number of classes in the classification problem.

The next step is to define the mathematical operations performed by the network. This network is simply a sequence of matrix-vector multiplications followed by a non-linearity:

```
h1 = tf.matmul(x, W1) + b1

h1 = tf.nn.relu(h1)

h2 = tf.matmul(h1, W2) + b2

h2 = tf.nn.relu(h2)

h3 = tf.matmul(h2, W3) + b3
```

The non-linearity applied to every output is the "relu" application function. The output of each layer must be used as the input of the next layer. There is not non-linearity in the last layer because it will use a special non-linearity, which is the softmax.

At this moment, it is necessary to define the loss function that this network will use. In this case it is required to perform the multinomial logistic regression using softmax cross-entropy as loss function:

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=h3))
```

The loss function will compute the softmax activation with the cross-entropy cost between the output of the last layer (h3) and the true labels (y). This function will return a vector that has one scalar value of this cost for each of the entries in the batch. Since the loss function must be a scalar value, it is required to compute the mean of all these values.

Given the node that will hold the values of the loss function, it is possible to use Tensorboard to track these values by attaching a scalar summary to the loss:

```
optimizer = tf.train.GradientDescentOptimizer(0.5)
train_step = optimizer.minimize(loss)
```

The used learning rate of the gradient descent is set to 0.5. This value is sufficiently small to allow the network to converge to the minimum of the loss function, but it is also large enough to avoid an excessively slow process. Once the optimizer is defined, the minimize method is called to tell the optimizer which loss function has to be minimized.

The next step provides a measure of the accuracy of the neural network, so how many time it gets the prediction of the digit correct. The network will consider correct the digit that corresponds to the position of the maximum in the output vector. This output is compared with the labels:

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(h3, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

The *argmax* function returns the position of the maximum and then "tf.equal" is used to check for equality between two tensors. It will return a Boolean value that is 1 if they match or otherwise 0. "Correct_prediction" is a Boolean tensor, which must be converted in another numerical type, through the function "tf.cast", in order to evaluate the mean among all the predictions that the network performed in every single batch. That is why "tf.reduce_mean" was used.

Thanks to another scalar summary, the value of the accuracy is tracked with Tensorboard. Now all the operations of the computational graph are defined. It is possible to create a session to run this graph through the function "tf.InteractiveSession()". Then, all the created summaries are merged into a single node: "merged = tf.summary.merge_all()". Before starting the training, all the variables must be initialized using:

```
tf.global_variables_initializer().run()
```

At this point, the training of the neural network can start. To perform the training of the neural network it is necessary a loop that iterates TRAIN_STEPS times (TRAIN_STEPS = 1000 in this case):

```
for i in range(1, TRAIN_STEPS):
        batch_xs, batch_ys = mnist.train.next_batch(BATCH_SIZE)
        summary_train, _ = sess.run([merged, train_step], feed_dict={x: batch_xs, y: batch_ys})
        train_writer.add_summary(summary_train, i)
```

$TRAIN\_STEPS$ should be large enough that the network sees the whole dataset many times. It means that the network sees a batch of 100 images at every iteration, but the dataset has 60000 images in total, so it will take 600 iterations to see the whole dataset once (one epoch). Typically, during the training phase, it is better if the network sees more than one epoch.

Inside the loop the "mnist" object is called. It returns a matrix that has size $BATCH\_SIZE$ by NUM_PIXELS for *batch_xs* and $BATCH\_SIZE$ by ten for *batch_ys*. These data will be used as input for running the code. A feed dictionary is created, which has *batch_xs* and *batch_ys*. It fills the placeholders x and y with *batch_xs* and *batch_ys*.

Then "sess.run()" will run the computations in the computational graph until it reaches the merged and *train_step* nodes. "sess.run()" will return the values of the summaries for Tensorboard in *summary.train*; the other node is *train_step* because it is necessary to perform one iteration of gradient descent, but it does not return any useful output. Thanks to this loop the gradient descent will be performed as many times as $TRAIN\_STEPS$. At the end of the loop the network should be trained, so the accuracy on the test set has to be computed.

"mnist.test.next_batch(10000)" returns the whole test set, and "sess.run()" is called again on the accuracy node, with test set as the input. The output will return the test accuracy, that is, in this case, around 98%.

```
batch_xs, batch_ys = mnist.test.next_batch(10000)
test_accuracy = sess.run(accuracy, feed_dict={x: batch_xs, y: batch_ys})
print('Test accuracy: %.4f' % test_accuracy)
```

In Tensorboard, Figure 3 and Figure 4 will be displayed:

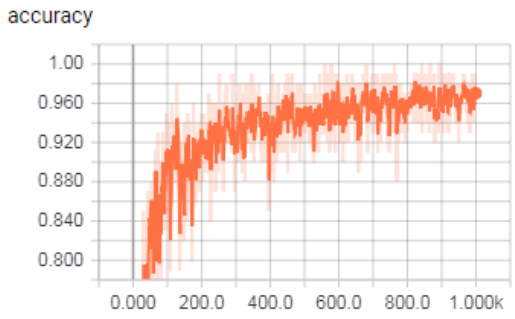

Figure 1: The curve of the accuracy as a function of the number of iterations in the training procedure.

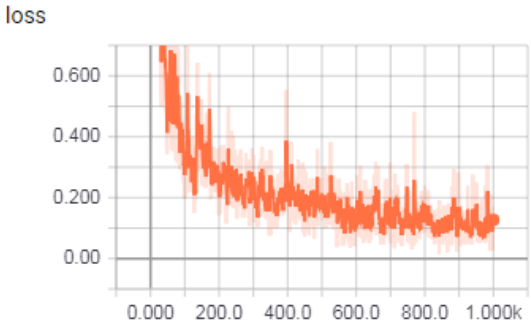

Figure 2: The curve of the loss as a function of the number of iterations in the training procedure.

Figure 3 shows the accuracy evaluated at each iteration in the training procedure. It can be seen that the function grows until it saturates around 98%. Figure 4 shows the loss function, which conversely decreases until it reaches a minimum.

2 Part 2: Keras

The second exercise of the neural networks lab required the application of Keras. Keras is a high-level application programming interface (API) that runs on top of Tensorflow. Keras allows to code neural networks more easily through its integration into the native Tensorflow flow.

In Keras, it is not necessary to define the variables because there are high-level definitions for many common neural network layers. Then, everything works as in pure Tensorflow, as it will be explained later.

The task of the second exercise is to use Keras in order to define a convolutional neural network that classifies the MNIST images, which are the same data already used in exercise 1.

2.1 The program

As in the previous exercise, it firstly becomes necessary to define the placeholders that hold the inputs of the neural network. Two placeholders are defined: the first one will contain the pixels of the images for every batch, while the second one will contain the true labels.

Then, the Keras model is defined through a function: "classifier_model()". This function defines the sequential model, that in this case is a linear stack of layers, which means that the output of a layer is the input of the next layer. The "add" method of the model is exploited to include additional layers. The first layer, as specified by the requirements, is a two-dimension convolutional layer. The filter that is going to be used has a spatial dimension of 3x3 (it is common in neural networks to use small filters), then the strides are set equal to 2 in both x

and y directions; this means that the filter is moving by two pixels at a time, so the spatial resolution will be halved at each layer. The first parameter of the "add" method is the number of filters, because at each convolutional layer it is not applied a single filter but many of them. This parameter is arbitrary chosen, the more filters are applied, the higher the capacity of the network is, which means that more complicated models can be learned but the neural network becomes harder to train.

The first convolutional layer has an "input shape" parameter because the first layer in a sequential model requires that the dimension of the input is manually specified. There is also an "activation" parameter that specifies which kind of non-linear activation function must be applied at the layer. As in the previous exercise, the non-linearity applied to every output is the "relu" application function.

The neural network must have two 2D convolutional layer built in this way. After the second layer, the output has shape (BATCH_SIZE, 7, 7). The last layer must be a fully connected layer through the "Dense" layer of Keras, but this layer only accepts an input that is vectorized. In order to solve this issue, a "Flatten" layer is added between the second convolutional layer and the dense layer. The flatten layer converts the shape (BATCH_SIZE, 7, 7) into (BATCH_SIZE, 49). At this point, the only parameter required by the dense layer is the number of output neurons, that corresponds with the number of possible classes. The code for the obtained sequential model is:

```
def classifier_model():
        model = models.Sequential()
        model.add(layers.Conv2D(NUM_FILTER_1, kernel_size=3, strides=(2, 2),
        activation='relu', input_shape = (NUM_PIXELS, NUM_PIXELS, 1)))
        model.add(layers.Conv2D(NUM_FILTER_2, kernel_size=3, strides=(2, 2),
        activation='relu'))
        model.add(layers.Flatten())
        model.add(layers.Dense(10))
return model
```

This sequential model is applied to the placeholder x and it will return the output h, which is the output of all the layers that are specified in the model.

```
my_net = classifier_model()
h = my_net(x)
```

After this step, the remaining passages are the same of the previous exercise:

1. Define the loss function, which as usual is the softmax activation with cross-entropy cost;
2. Use the scalar summary to keep track of the loss in tensorboard.
3. Identify the optimizer, which is the gradient descent optimizer with a learning rate set to 0.5. The optimizer is used to minimize the loss function previously defined, by creating "train_step", which is a node in the graph that performs one iteration of gradient descent.
4. Compute the accuracy and keep track of it in tensorboard.
5. Create the session
6. Merge all the summaries into one single node to keep track of them

7. Initialize the variables, even if these variables have not been explicitly declared in the code, because when a sequential model is created, internally Keras allocates all the variables, without the need of keeping track of them manually.

8. Run the training loop defined in the previous exercise up to a certain number of training steps.

9. The test phase can start and the accuracy on the test set can be evaluated after the training phase. Also this step is equivalent to the testing phase of the exercise 1.

In this case, the training loop is iterated 1000 times. As figure 3 shows, the accuracy converges very quickly, after 800 iterations more or less. The training phase should not be run for too long, because otherwise there is the risk of overfitting. Overfitting occurs when the training accuracy gets very close to 100%, while the testing accuracy decreases. In this case the testing accuracy is around 98.5%. It means that overfitting does not occur, since the accuracy gets high values both in the training and test set. Figure 4 shows the loss function, which decreases until it gets a minimum around 0.1, as in the previous exercise.

By a comparison of the number of parameters for this model with the one implemented in the first exercise, it turns out that this model has much fewer parameters. This is the advantage of a convolutional network with respect to a fully connected network, which makes the network easier to train.

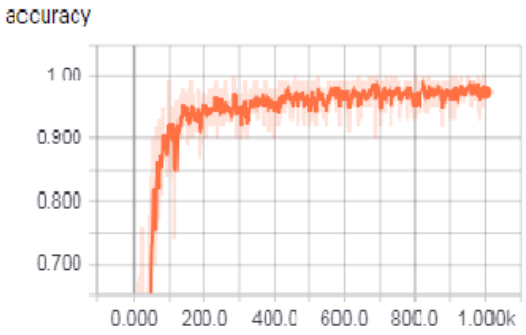

Figure 3: The curve of the accuracy as a function of the number of iterations in the training procedure.

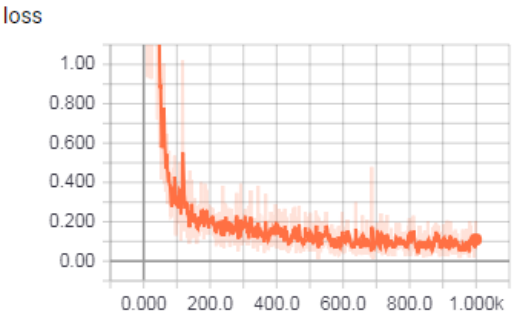

Figure 4: The curve of the loss as a function of the number of iterations in the training procedure.

References

[1] Statistical Signal Processing A.A. 2017/2018 Statistical Learning and Neural Networks.

01SOVBH - Statistical learning and neural networks

Statistical Signal Processing Computer Lab – Kalman Filter

Group Members
Lorenzo Bellone
Victor De Castro Morini

Turin, February 5, 2019

Contents

List of Figures

Acronyms

GPS Global Position System.

lab Laboratory Lesson.

LG-SSM Linear Gaussian - State Space Model.

SSM State Space Model.

1 The Kalman Filter

The Kalman filter is an algorithm that uses a series of observations corrupted by statistical noise to make estimations of unknown variables.

In order to have a better idea of what the Kalman filter does, let's consider a system that can be in different "states", where the system behavior depends on the state in which the system is. The sequence of the states assumed by the system can be defined through a function:

$$z_t = g(u_t, z_{t-1}, \epsilon_t) \tag{1}$$

Equation (1) describes how the system is going to evolve with the time. At every instant of time, the state in which the system is depends on: some optional inputs u_t, the previous state z_{t-1} and noise ϵ_t. It is important to say that the observer will never see these states since they are hidden. The output of the system, which is the only observation that comes out from the system, also can be defined by a function:

$$y_t = h(z_t, u_t, \delta_t) \tag{2}$$

Equation (2) shows that the output depends on: the state in which the system is, the optional inputs and some observation noise.

The Kalman filter has two main objectives:

1. To make some predictions on the outputs, given all the previous observations of the system;
2. To make some inference on the statistical distribution of the states given the outputs. This is the most powerful characteristic of the Kalman filter, since it is possible to say something about variables that will never be directly observed.

The Kalman filter works with linear dynamical system. This means that the equations (1) and (2) are linear functions, which can be rewritten as:

$$z_t = A_t z_{t-1} + B_t u_t + \epsilon_t \tag{3}$$

$$y_t = C_t z_t + D_t u_t + \delta_t \tag{4}$$

The Equation (3) is called "State Equation" while the equation (4), "Observation Equation". The noises are assumed to be Gaussian distributed:

$$\epsilon_t \sim N(0, Q_t) \tag{5}$$

$$\delta_t \sim N(0, R_t) \tag{6}$$

where ϵ_t is the system noise, which means, the level of confidence that the Equation (5) properly represents the behavior of the system. δ_t is the observation noise, which means how much the Equation (6) describes the sensing system.

In the overall, the parameters of a linear dynamical system are the four matrices A_t, B_t, C_t and D_t that describe the relationship between the input, the state and the output, and the covariance matrices of the system noise Q_t and of the measurement noise R_t.

The algorithm is made of two steps: the prediction step and the update step. It works in the following way: at time t-1, all the inputs u_t and the outputs y_t are known. When the system sees a new input at time t, an intermediate estimate of the probability of the state can be done:

$$p(z_t|y_{1:t-1}, u_{1:t}) \sim N(z_t|\mu_{t|t-1}, \Sigma_{t|t-1}) \tag{7}$$

The Equation (7) is an intermediate estimate, since the output y_t has not yet been observed. A Gaussian assumption is made, with a mean value $\mu_{t|t-1}$, which indicates that the outputs have been observed up to time $t-1$, while the inputs up to time t. $\Sigma_{t|t-1}$ is the corresponding covariance matrix. Furthermore, it is possible to evaluate the distribution of y_t, $p(y_t|y_{1:t-1})$, which allows to come up with an estimation of the future value of the observation.

$$\widehat{y}_t = E[y_t|y_{1:t-1}, u_{1:t}] \tag{8}$$

Equation (8) shows the estimation of the observation as the expectation of this statistical distribution, since the distribution is Gaussian. This calculation ends the prediction step. In the update step, a new output y_t is observed and the temporary estimate of the state is updated:

$$p(z_t|y_{1:t}, u_{1:t}) \sim N(z_t|\mu_t, \Sigma_t) \tag{9}$$

Also this estimation is Gaussian distributed. The shapes of all the estimated probability density functions are never going to change. What instead changes is the value of the mean and the covariance matrix of these distributions, which will be iteratively updated by the Kalman filter.

Prediction:

$$\mu_{t|t-1} = A_t \mu_{t-1} + B_t u_t$$

$$\Sigma_{t|t-1} = A_t \Sigma_{t-1} A_t^T + Q_t$$

Update:

$$\widehat{y}_t = C_t \mu_{t|t-1} + D_t u_t$$

$$K_t = \Sigma_{t|t-1} C_t^T (C_t \Sigma_{t|t-1} C_t^T + R_t)^{-1}$$

$$r_t = y_t - \widehat{y}_t$$

$$\mu_t = \mu_{t|t-1} + K_t r_t$$

$$\Sigma_t = (I - K_t C_t)\Sigma_{t|t-1}$$

(10)

2 The Exercise

Given an object moving along a x,y axis(2D), the task was to design a Kalman Filter algorithm (iterative) to track the object. The state vector $z_t \in \mathbb{R}^4$, is represented by:

$$z_t^T = (z_{1t}, z_{2t}, \dot{z_{1t}}, \dot{z_{2t}}) \tag{11}$$

Where z_{1t}, z_{2t}, are the horizontal and vertical locations of the object, and $\dot{z_{1t}}$, $\dot{z_{2t}}$ the respective velocities.

The model parameters are described as $\Theta_t = (A_t, B_t, C_t, D_t, Q_t, R_t)$, and the exercise states that:

$$A = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{12}$$

A is a square matrix **nxn** which contains constant coefficients that describe the system. **n** is the number of state variables, or the length of state vector z_t.

$$B = 0 \tag{13}$$

B is a **nxr** matrix, which contains constant coefficients that weight the input. **n** is the number of state variables, and **r** is the number of inputs.

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{14}$$

C is a **mxn** matrix, which contains constant coefficients that weight the state variables. **m** is the number of outputs, and **n** is the number of state variables.

$$D = 0 \tag{15}$$

D is a **mxr** matrix, which contains constant coefficients that weight the system inputs. **m** is the number of outputs, and **r** is the number of inputs.

From Equation 3, it was possible to infer the transitional/system model, depicted on Figure 1. Δ given is 1.5 and represents the sampling period, and ϵ_t represents the system noise vector with standard deviation σ_{Qx} for the positions and σ_{Qv} for the velocities.

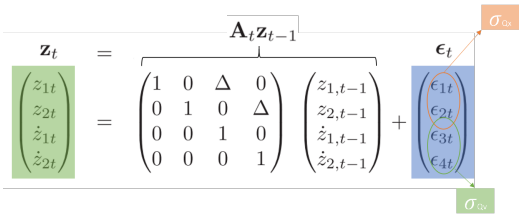

Figure 1: Transitional/System Model: In green, z_t, the state vector. A, the square matrix **4x4** which contains constants that describe the system, multiplied by the state vector of the previous step, z_{t-1}. ϵ_t is the random Gaussian noise of the system, as for example, the wind, with standard deviation σ_{Qx} for the positions and σ_{Qv} for the velocities.

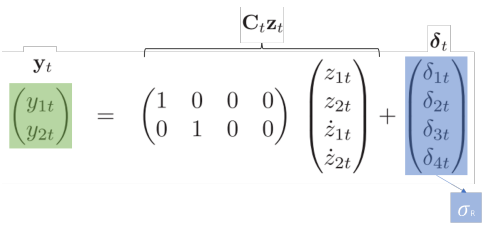

Figure 2: Observation Model: In green, y_t, the output column vector of the output variables. C, the **2x4** matrix which contains constant coefficients that weight the state variables, multiplied by the state vector, z_t. δ_t is the observation noise vector, a Gaussian with diagonal covariance matrix representing the noise, as for example, the GPS error, with standard deviation σ_R on each component.

Accordingly, from Equation 4, it was possible to infer the observation model, where it is possible to observe the location of the object, but not the velocity. Considering the output vector $y_t \in \mathbb{R}^2$, where $y_t = (y_{1t}, Y_{2_t})$ and δ_t the observation noise vector with standard deviation σ_R on each component, the observation model is depicted on Figure 2.

Considering the steps provided on Equation 1, an iterative Matlab script was made in order to apply the Kalman filter and show its results. Consider that initial estimates for μ and Σ were chosen randomly. The result is presented on Figure 3.

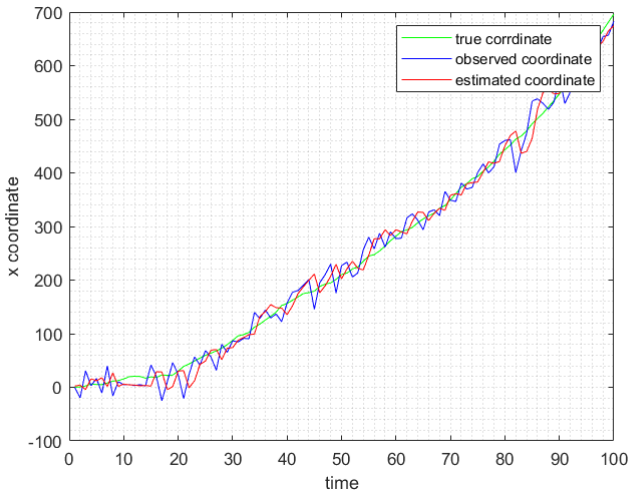

Figure 3: Plot of the estimated position of the object over time with respect to the true position and the observed position.