City University London

MSc Data Science

Project Report

2015

Representation Learning for Structural Music Similarity Measurements

Author: Vahndi Minah

Supervisor: Tillman Weyde

Co-supervisor: Daniel Wolff

Submitted: 25th September 2015

ABSTRACT

The Project has been carried out to determine whether representation learning can be used to improve the performance of a state-of-the-art structural music similarity system. Representation learning has been carried out using denoising autoencoder neural networks on a number of common audio features. Relevant components of existing toolkits written by members of the MIR community in Matlab have also been adapted for the Python 2.7 programming language. Finally, new types of features have been developed incorporating the combination of representation learning and energy-normalised statistics calculations. These features have shown to be perform favourably in comparison with highly engineered features derived from chroma vectors. Future work is identified to further investigate the development of these features using higher level representation learning and to improve upon the overall system performance.

Keywords: Audio signal processing, music information retrieval (MIR), music structure analysis, representation learning.

TABLE OF CONTENTS

Abstract 2

1 Introduction and Objectives 1

1.1 Formatting 1

1.2 Purpose, Products and Beneficiaries 1

1.3 Objectives 2

1.4 Methods 2

1.4.1 Porting of toolboxes 2

1.4.2 Structural Similarity System 2

1.4.3 Representation Learning 2

1.5 Work Plan 2

1.6 Background - The Need for Music Similarity Systems 3

1.6.1 Piracy and file sharing 3

1.6.2 Broadcasting royalties 3

1.6.3 Music Information Retrieval (MIR) systems 3

1.6.4 Recommender systems 3

1.6.5 Music Research 4

1.7 Changes in Goals and Methods 4

1.8 Structure of the Report 4

2 Context 5

2.1 The State of the Art in Structural Music Similarity Measurement Systems 5

2.1.1 Types of Feature 5

2.1.2 Feature down-sampling 5

2.1.3 Recurrence plots 5

2.1.4 Normalised Compression Distance 7

2.1.5 Mean Average Precision 7

2.2 Types of Neural Network Investigated 8

2.2.1 Denoising Autoencoders (Vincent, et al., 2008) 8

2.2.2 Restricted Boltzmann Machines 9

3 Methods 11

3.1 Emulation of Existing System 11

3.1.1 Programming languages and tools 11

3.1.2 Communication with the author of the existing system 11

3.1.3 Datasets 11

3.1.4 Feature down-sampling 12

3.1.5 Recurrence plot numerical parameters 12
3.1.6 Recurrence plot methods 12
3.1.7 Normalised Compression Distance 12
3.2 Training, Validation and Testing 13
3.2.1 Training / validation / testing methodology 13
3.2.2 Number of experiments and MAP calculations 13
3.3 Feature Learning 14
3.3.1 Types of features 14
3.3.2 Feature distributions 14
3.3.3 Types of neural network 14
3.3.4 Creation of training files 15
3.3.5 Input feature range standardisation 15
3.3.6 Network dimensions 16
3.3.7 Batch size 17
3.3.8 Learning rate 17
3.3.9 Corruption level 17
3.3.10 Incorporation of temporal effects 18
3.4 Creation of new CENS-like features 19
3.4.1 Number of features 19
3.4.2 Input feature transformation 19
3.4.3 Normalisation threshold 20
3.4.4 Quantisation 20
3.4.5 Down-sampling 20
3.5 Training Algorithms 21
3.5.1 Greedy, hill-jumping algorithm for optimal parameter search 21
3.5.2 Fine-tuning of feature transformation weights 21
3.5.3 Coercion of FENS features into a Gaussian distribution 22
4 Results 23
4.1 Effects of Parameter Settings on MAP results 23
4.1.1 Feature Down-sampling vs. NCD Sequence Length 23
4.1.2 Recurrence Plot Parameters 24
4.1.3 Number of hidden units 25
4.1.4 Neural network training batch size 25
4.1.5 Frequency standardisation / number of input features 26
4.1.6 Time-stacking 27
4.1.7 Fine-tuning of feature weights 28

4.1.8 FENS transformation functions 28
4.1.9 FENS down-sampling rate and window length 29
4.1.10 FENS quantisation steps 30
4.1.11 FENS quantisation weights 31
4.1.12 Coercion of FENS features into a normal distribution 32
4.2 Training Results 33
4.2.1 Features without energy-normalised statistics 33
4.2.2 Features using energy-normalised statistics 33
4.3 Validation Results 34
4.4 Test Results 34
4.5 Analysis 34
4.5.1 Results without Energy-Normalised Statistics 34
4.5.2 Results with Energy-Normalised Statistics 35
4.5.3 Why did CENS perform better than CQSFENS? 35
4.5.4 Comparison of Results with (Bello, 2011) 35
5 Discussion 36
5.1 Original Objectives 36
5.1.1 Porting of toolboxes from *Matlab* to *Python* 36
5.1.2 Structural similarity system 36
5.1.3 Representation learning 37
5.2 New Findings 37
5.3 Answer to the Research Question 37
6 Evaluation, Reflection and Conclusions 38
6.1 Adequacy of Project Plan 38
6.2 Choice of Objectives 38
6.3 Recommendations for further work 38
6.3.1 Structural similarity system 38
6.3.2 Representation learning 39
6.3.3 Interface of representation learning with the structural similarity system 39
6.3.4 Learning from more data 39
7 Glossary 40
8 References 41
Appendix A Descriptions of Common Audio Feature Types 44
A.1 Spectrogram 44
A.2 Log -frequency spectrogram 44
A.3 Constant-Q spectrogram 44

A.4 Mel-Frequency Cepstral Coefficients (MFCCs) 44
A.5 Chroma 45
A.6 Chroma Energy Normalised Statistics 45
A.7 Chroma DCT-reduced log pitch 46
Appendix B Analysis of the Greedy, Hill-Jumping Algorithm 47
Appendix C Tables 48
C.1 Toolkits used in the original study (Bello, 2011) 48
C.2 Risk register 48
C.3 Factors influencing the choice of programming language 48
C.4 Numbers of combinations of run settings 49
C.5 Neural network runs 50
C.6 Record of Training Runs 51
Appendix D Original Planned Project Timeline 53
Appendix E Communication with Dr. Norbert Marwan 54
Appendix F Example Code 56
F.1 CENS and FENS feature generation functions 56
F.2 Calculation of Recurrence Plots 60
F.3 Calculation of the Normalised Compression Distance 64
F.4 Calculation of Mean Average Precision from NCD pairs 65
F.5 Calculation of Mean Average Precision for a Single Parameter Setting 67
F.6 Optimiser Class used in Greedy Hill-Jumping Algorithm 70
F.7 Stepper and Adjuster classes used in Weights fine-tuning and Coercion of FENS Distributions 73
F.8 Learned Weights Fine-tuning Algorithm 77
F.9 FENS Distribution Coercion Algorithm 81
F.10 MAP Training Algorithm (with or without neural network weights) 86
F.11 MAP Training Algorithm (with FENS parameters) 89
F.12 Training of Denoising Autoencoders (adapted from example Theano code) 94
F.13 Training of RBMs (adapted from example Theano code) 101

1 INTRODUCTION AND OBJECTIVES

1.1 FORMATTING

Capitalisation is used to identify the Author (Vahndi Minah), the Project (Representation Learning for Structural Music Similarity Measurements) and the Report (this report).

Quotations from external sources are presented in blue italic font.

Names of software are written in *black italic font*.

The first reference to a piece of software is given with a *clickable hyperlink*, for readers interested in using or reading more about the software.

1.2 PURPOSE, PRODUCTS AND BENEFICIARIES

The purpose of the Project is to determine whether audio features produced using representation learning can outperform or enhance the performance of engineered features when used as part of a structural music similarity measurement system. The performance of the learned features has been measured against commonly used engineered audio features using an implementation of the system proposed in (Bello, 2011) and (Bello, 2009).

The products that have been generated are as follows:

- Neural network models and associated training algorithms, which produce learned features to accomplish the purpose of the Project.
- An implementation of the system described in (Bello, 2011) and (Bello, 2009), using version 2.7 of the *Python* programming language.
- A port of relevant code from existing *Matlab* Toolboxes into the *Python* programming language.
- New algorithms used in training the implemented system and producing new types of features, based on a combination of representation learning and energy-normalised statistics (Muller, et al., 2005).

The beneficiaries of the work comprise three groups who benefit from an improved ability to identify structural similarity in music:

- Members of the Music Information Retrieval (MIR) community who are interested in furthering the goals of MIR, one of these being effective methods for measuring structural music similarity.
- Music copyright owners such as artists and record labels, who stand to gain financially from increased protection of their intellectual copyright.
- Users of online music distribution services who will benefit from better content-based recommendations.

The question posed for the research project was as follows:

Can representation learning improve the performance of structural music similarity measurements in comparison with engineered features?

1.3 Objectives

In order to answer the research question, a number of objectives were set out:

- Porting of relevant code from *Matlab* toolkits used in (Bello, 2011) to the *Python* programming language, i.e.
  - The *CRP Toolbox* (Marwan, 2015), used to calculate Recurrence Plots.
  - The *Chroma Toolbox* (Muller & Ewert, 2011), used to calculate CENS features from chroma features.
  - The *CompLearn Toolkit* (Cillibrasi, et al., 2008), used to calculate the Normalised Compression Distance.
- Develop a structural similarity system in *Python* to reproduce the system described in (Bello, 2011).
- Use representation learning to learn new features and compare their performance in the structural similarity system with traditional engineered features.

1.4 Methods

1.4.1 Porting of toolboxes

The required functionality from each of the toolboxes was translated by hand from *Matlab* into *Python 2.7*, making extensive use of the *numpy*, *scipy* and *pandas* modules. A/B testing of every function was carried out to ensure that the *Python* functions produced exactly the same results as the original *Matlab* code. The author of the *CRP Toolbox*, Dr. Norbert Marwan, was contacted to clarify the workings of the recurrence plot code, leading to resolution of a potential bug in the original code (see Appendix E).

1.4.2 Structural Similarity System

As the code for the system used in (Bello, 2011) was not available, it was implemented from the ground up, incorporating the ported code from the toolboxes and using the system description in (Bello, 2011) as a basis for the design.

Additional training algorithms and classes were developed using object-oriented design principles to deal with the vastly increased parameter space introduced by the incorporation of neural networks and new types of features.

1.4.3 Representation Learning

The *Theano* machine learning library (Bergstra, et al., 2010) was used for the representation learning. Tutorial scripts which were designed to work with the *MNIST handwritten digits dataset* (National Institute of Standards and Technology, 1998) were adapted to work with audio features, and some additional functionality was introduced to optimise the training for the audio features.

1.5 Work Plan

The original Work Breakdown Structure is shown in Figure 1. The associated planned timeline is given in Appendix D. After an initial two week period of testing of neural networks, it was decided that the highest risk item would be implementation of the structural similarity system. Therefore this was brought forwards in the plan of work, and the neural network development was postponed until after this had been accomplished. While the order of the tasks changed, the tasks themselves remained largely as had been planned, with some additional work put into post-processing the learned neural network features using energy-normalised statistics. This meant that some

compromises had to be made on the variety of neural networks that could be used for representation learning.

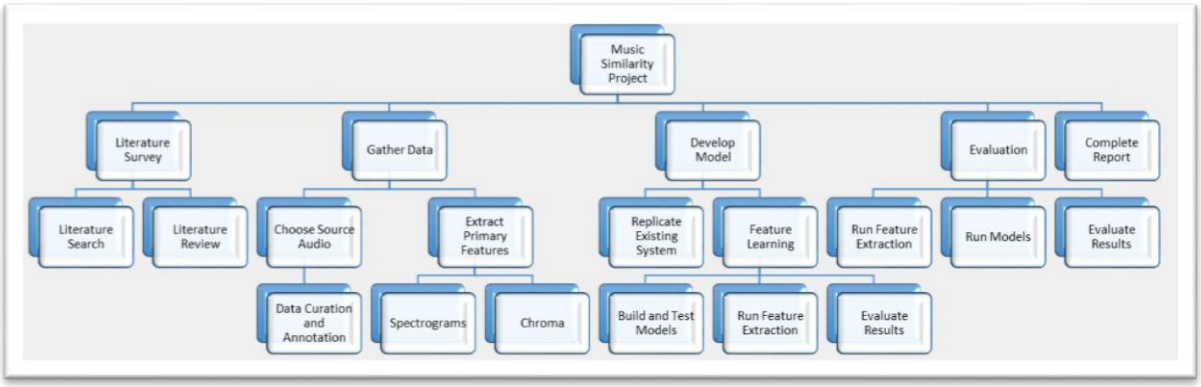


Figure 1 - Original Work Breakdown Structure from the Project Plan

1.6 Background - The Need for Music Similarity Systems

1.6.1 Piracy and file sharing

Efficient audio compression algorithms in combination with faster home internet speeds and increased numbers of personal devices have enabled more widespread piracy of music (Popham, 2011). Many popular peer-to-peer file-sharing services such as Pirate Bay have now fallen foul of copyright law. However, online sharing of music has been demonstrated to constitute a large loss of revenue to the music industry (Butler, 2007); one estimate is that *28.2% of the population participates in illegal file sharing* (Bustinza, et al., 2013). The ability to recognise music which has been slightly modified to evade piracy detection measures (for example by adding random low-level noise) would be a valuable tool in reducing lost revenues.

1.6.2 Broadcasting royalties

Every time a song is broadcast on the radio, television or online, the broadcaster has a duty to provide a royalty payment to the copyright owner. Although most broadcasters and large online providers such as YouTube have agreements with performing rights organisations, an increase in the number of online broadcasters has the potential to overwhelm the organisations responsible for collecting these royalties and revenue may be lost.

1.6.3 Music Information Retrieval (MIR) systems

The field of MIR system research tackles myriad problems, including genre, composer and artist classification, and extraction of other metadata such as key, meter, tempo and instrumentation which can aid in applications such as automatic transcription, cover song identification and compression. An accurate structural similarity measurement system could, for example, assist in making more accurate transcriptions by identifying similar known pieces of music with verified metadata such as time signature and tempo, and using these as a basis for the new transcription.

1.6.4 Recommender systems

Probably the most famous recommender system problem was posed in 2006 by Netflix, who offered a prize of $1M to anyone who could increase the quality of their user recommendations by 10% (Koren, 2015). Many recommender systems use collaborative filtering to assist in making recommendations. Collaborative filtering works by identifying similar users by existing preferences, and making recommendations to one user based on a similar user's likes.

However, the collaborative filtering approach to recommendation suffers from a number of drawbacks, such as the “cold-start” problem. For music recommendation systems, such as those used at online music distributors like Google Play, iTunes and Spotify, the cold-start problem occurs when an artist releases a new song and it has not yet been rated by any users. While tracks by popular artists are likely to be listened to regardless of existing listens, this is not the case for tracks from new or niche artists, due to the long-tail distribution in popularity of songs (Herrada, 2008). This is where content-based recommendation systems using audio similarity measures are most useful. These are sometimes known as latent factor models as they use a hidden representation of the content to measure similarity; these models have also been applied in the domain of music-based recommendation (van den Oord, et al., 2013).

1.6.5 Music Research

Another application of structural music similarity systems is music research. Musicologists are interested in making structural comparisons on large collections of music. This is often difficult or impossible to do manually, due to the vast volumes of material involved.

1.7 Changes in Goals and Methods

The goals of the Project did not changed substantially from the Project Plan. However, as the Project developed, an insight into the use of energy-normalised statistics to improve the performance of conventional chroma inspired the Author to borrow from and adapt the process for calculating energy normalised statistics to improve the performance of learned features on the task of structural similarity measurements.

1.8 Structure of the Report

Section 2 sets out the context for the work, giving an overview of the state of the art in structural music similarity measurements, and provides a description of the types of neural network which were investigated for the purpose of feature learning.

Section 3 describes in detail the methods used in the Project, including how each step of the original system was emulated, how the neural networks were trained for the representation learning, how the new types of features were developed based on existing CENS features, and the algorithms that were written to train the system.

Section 4 gives intermediate and final results of the training and testing, exploring some of the more important interactions between the various system parameters.

Section 5 discusses the results obtained, and makes a qualitative comparison between the system developed and the system in (Bello, 2011), outlining some potential reasons for differences in performance between them.

Section 6 provides an evaluation of the results obtained in the Project, reflects on what has been learned and lists some conclusions and suggestions for further work that could lead to a better system performance in the future.

Section 7 provides a glossary of some technical terms used in the Report.

Finally, Section 8 provides a list of References used throughout the Report.

2 CONTEXT

2.1 THE STATE OF THE ART IN STRUCTURAL MUSIC SIMILARITY MEASUREMENT SYSTEMS

As state of the art, we use the system described in (Bello, 2011), which uses an information-based distance metric, the Normalised Compression Distance, to classify the pair-wise similarity of musical recordings which have been converted into recurrence plots using time-delay embedding. The overall accuracy of the system proposed is measured using a ranking method known as the Mean Average Precision. The signal chain of the system is illustrated in Figure 2.

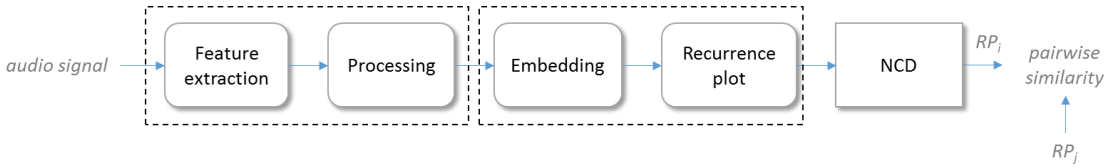


Figure 2 - Illustration of the signal chain described in (Bello, 2011)

The Project seeks to augment the performance of this system by the addition of feature learning. This section gives a brief description of the main components of the system, so that the later enhancements can be discussed with more clarity in Section 3.

2.1.1 Types of Feature

As inputs into the recurrence plots, the author uses three different types of musical audio features which have been extracted from the original audio signal: chroma, chroma energy-normalised statistics (CENS) and chroma DCT-reduced log pitch (CRP). The chroma feature and its variants have proved useful in a number of MIR applications such as cover song identification (Ellis & Poliner, 2007), audio thumb-nailing (Bartsch & Wakefield, 2005) and audio synchronisation (Muller, et al., 2005). The CENS and CRP features, which are derived from chroma, are less widely used than conventional chroma, but showed significant improvements in performance in comparison with the standard chroma features using the system described in (Bello, 2011).

2.1.2 Feature down-sampling

Before the recurrence plots which will be described in 2.1.3 were calculated, down-sampling of the input features was performed, to achieve overall feature rates of ⅓, ½, 1, 1¼, 2½, 5 and 10 features per second. This step can have a number of benefits:

- Down-sampling reduces the number of features which must be processed, therefore decreasing the storage space and increasing the calculation times required for recurrence plots.
- Since the similarity being measured is structural, very short timescale information is superfluous for the task.
- It is stated in (Bello, 2011) that both the use of chroma features and down-sampling *help to smooth out the effect of short-term events such as transients and different types of pitch simultaneities (e.g. arpeggios, walking bass lines, trills).*

An additional, variable feature rate was also implemented in (Bello, 2011) using beat-tracking of the individual pieces. However, it was shown that there was a relatively low upper bound to the overall performance that could be achieved using this method.

2.1.3 Recurrence plots

A recurrence plot (RP) is a two-dimensional graphical illustration of the similarities between two time-series (a cross-recurrence plot) or the self-similarity of a single time-series (an auto-recurrence

plot). Auto-recurrence plots are similar to self-similarity matrices (SSMs), where for a given time series $\boldsymbol{x}(t)$, the value of the element at row i and column j of the SSM represents the distance between the value of $\boldsymbol{x}$ at time $t = i$ and the value at time $t = j$. Note that the term *distance* is used rather than *difference* because in the case of musical time series, $\boldsymbol{x}(t)$ is usually a two-dimensional vector. Therefore a method such as the Euclidean distance is used to compute a single figure distance value between the multi-dimensional points in space.

Recurrence plots differ from self-similarity matrices in that they are calculated from a time-delay embedded representation of the time-series, rather than from the raw time series. Also, instead of using the actual distance between the values of two elements i and j of the matrix to represent a lack of similarity, a threshold distance ϵ is chosen such that for any i, j with a distance less than ϵ, the similarity is set to one, and for elements with a distance greater than ϵ, the similarity is set to zero. Thus, recurrence plots are typically binary rather than real-valued. Example recurrence plots of a Mazurka from the training set, calculated at a number of different resolutions (sequence lengths) are shown in Figure 3.

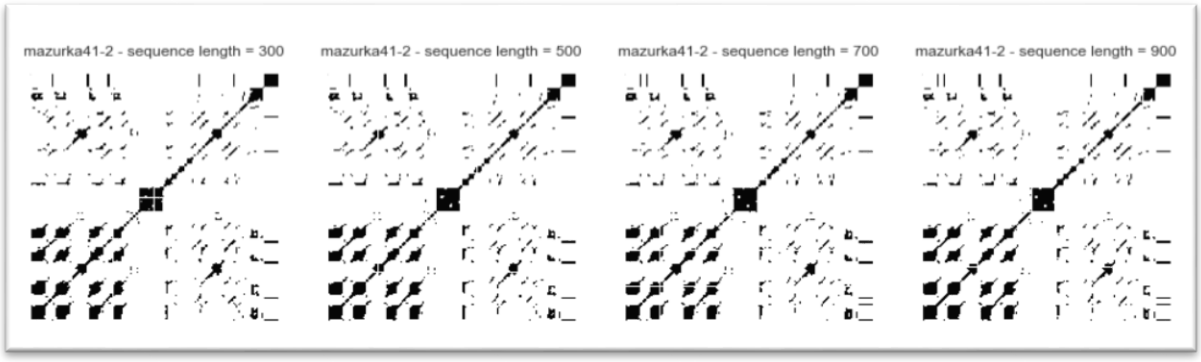


Figure 3 - Recurrence Plots calculated for a Mazurka from the Mazurka Dataset, resized to fixed resolutions (sequence lengths)

To calculate the time-delay embedded representation for a time series of length N, two additional parameters are required, the time delay τ and the embedding dimension m. These parameters allow repeating sequences of temporal elements to be found, rather than individual elements. This is particularly relevant for structural similarity, since different performances of the same piece would be expected to exhibit repetitions of the same phrases, motifs and sections, even if other facets of the performances, such as instrumentation or musical key are different.

The time delay τ can be thought of as controlling the expected typical temporal separation of the information which is being characterised, for example the time delay between two notes in a musical piece. The embedding dimension m can be thought of as controlling the length of sequences for which repetitions which are sought. The time-delay embedded representation $\boldsymbol{x_c}(n)$ for a β-dimensional input time series $\{\boldsymbol{c}_b(n)\}$ where $b \in [1, \beta]$ can be obtained by concatenating elements of $\boldsymbol{c}(n)$ thus:

$$x_c(n) = \begin{matrix} [c_1(n), c_1(n-\tau), c_1(n-(m-1)\tau)\,, \\ \cdots, \\ c_\beta(n), c_\beta(n-\tau), c_\beta(n-(m-1)\tau)]^T \end{matrix}$$

In (Bello, 2011), three different methods of computing the recurrence plot were tested. Two of these were selected for use in the Project and will be described in more detail later in the report. For

further information on these two methods, and details of the method not used in the Report, please refer to (Bello, 2011).

2.1.4 Normalised Compression Distance

The distance metric used in (Bello, 2011) to compute the similarity between two recurrence plots is the Normalised Compression Distance (NCD). The NCD is an information-theoretic distance metric that measures the similarity of two objects by comparing differences in the compressed sizes of their joint and individual information. It is based on the theoretical Information Distance (ID):

$$\mathrm{ID}(o_1, o_2) = \max\{K(o_1|o_2), K(o_2|o_1)\}$$

where: o_1, o_2 are the objects between which the Information Distance is sought, and
$K(x|y)$ is the conditional Kolmogorov complexity, the resources needed by a universal machine to specify x, given that it already knows y.

Since the Information Distance does not consider the sizes of o_1 and o_2, then in order to make useful comparisons between different IDs, it is necessary to normalise the measure, giving the Normalised Information Distance (NID):

$$\mathrm{NID}(o_1, o_2) = \frac{\max\{K(o_1|o_2), K(o_2|o_1)\}}{\max\{K(o_1), K(o_2)\}}$$

Also, since a universal machine does not exist, then the Kolmogorov Complexity is not computable, and therefore for practical purposes, it is necessary to use an approximation, the NCD:

$$\mathrm{NCD}(o_1, o_2) = \frac{C(o_1 o_2) - \min\{C(o_1), C(o_2)\}}{\max\{C(o_1), C(o_2)\}}$$

where: $C(o_1 o_2)$ is the size in bytes of the compressed concatentation of o_1 and o_2,
$C(o_1)$ is the size in bytes of the compressed representation of o_1, and
$C(o_2)$ is the size in bytes of the compressed representation of o_2.

It can be seen from the numerator that intuitively, the compressed size of both o_1 and o_2, minus the smaller of each one's compressed size, should roughly equate to the Kolmogorov Complexity, given a sufficiently efficient compression algorithm that can exploit similarities between o_1 and o_2 to reduce the size of their combined, compressed representation.

2.1.5 Mean Average Precision

Precision, also known as the positive predictive value, is a widely used measure in the machine learning community. It is often used in preference to the accuracy measure when it is important to find the proportion of true positives identified relative to the total number of true and false positives. In the context of a ranking system, for example in internet search results, precision is often used because typically the user does not care as much about the irrelevant pages that were not shown, as the proportion of pages that were shown which were relevant. This can be extended to the field of music similarity, where a user may only want to find pieces of music which are most similar to the query piece, and doesn't care as much about the system's accuracy on successfully putting the least relevant queries to the bottom of the pile.

Given a ranked list of M items for a given query q, where q is a member of both the set of K relevant items, and the set of all M relevant and irrelevant items, the Precision $P(r)$ of the ranked list up to a given position r can be defined as:

$$P(r) = \frac{1}{r}\sum_{j=1}^{r} \Omega(j)$$

where: $\Omega(j)$ is an indicator function which is equal to 1 if the item at rank j is relevant or 0 otherwise.

The Average Precision AP for a query q is then defined as the sum of the Precision values at the position of each relevant item, divided by the total number of relevant items:

$$AP_q = \frac{1}{K}\sum_{r=1}^{M} P(r)\,\Omega(r)$$

The Mean Average Precision (MAP), is then the mean value of AP for all queries $q \in M$:

$$MAP = \frac{1}{M}\sum_{q=1}^{M} AP_q$$

The MAP score is widely used in Information Retrieval (Inkpen, 2014), and is the measure used in (Bello, 2011) to compare different configurations of the structural similarity system. In this case the queries are the recurrence plots of the individual performances and the ranking is determined by sorting the NCDs between each recurrence plot and the query recurrence plot, in ascending order. The value of the relevance indicator function is set to one if the two recurrence plots are different performances of the same piece, and zero otherwise.

2.2 Types of Neural Network Investigated

2.2.1 Denoising Autoencoders (Vincent, et al., 2008)

An autoencoder is typically a three-layer unsupervised feed-forward network which is trained produce a reconstruction $\hat{\boldsymbol{x}}$ of a D-dimensional input vector $\boldsymbol{x}$ at the output layer via an intermediate, H-dimensional hidden layer $\boldsymbol{h}$. The combination $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b})$ of the weights $\boldsymbol{W}$ between $\boldsymbol{x}$ and $\boldsymbol{h}$, and the biases $\boldsymbol{b}$ of $\boldsymbol{h}$, computes the latent representation of the data and is known as the encoder. The decoder $\boldsymbol{\theta}' = (\boldsymbol{W}', \boldsymbol{b}')$ from the hidden layer to the output layer transforms this representation back into the reconstructed output vector $\hat{\boldsymbol{x}}$.

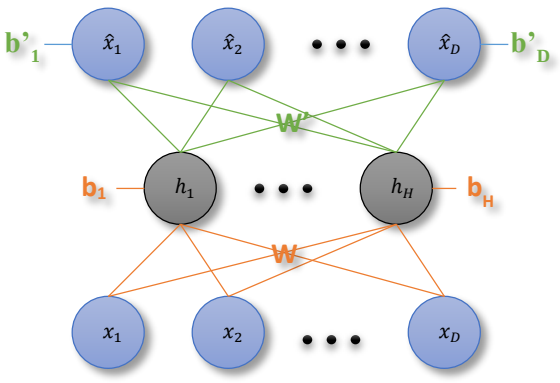


Figure 4 - An autoencoder

If the hidden layer is made smaller than the input and output layers, then an autoencoder can be used to compress the data using independent components of its distribution. The parameters of the

autoencoder are optimised to minimise the *reconstruction error* ε, which is the average over all n inputs in the training set of a loss function L:

$$\varepsilon = \underset{\theta, \theta'}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(x_i, \hat{x}_i)$$

The loss function L may be a common measure such as the mean-squared error or, alternatively, the reconstruction cross-entropy:

$$L = -\sum_{d=1}^{D} [x_d \log \hat{x}_d + (1 - x_d) \log(1 - \hat{x}_d)]$$

Substituting this loss function into the equation for the reconstruction error ε produces an expression which is differentiable, and therefore the autoencoder can be trained by stochastic gradient descent on $\frac{d\varepsilon}{dL}$.

A denoising autoencoder can improve upon the accuracy of the standard autoencoder by introducing a stochastic corruption process, which pseudo-randomly sets each of the input features to zero with a given probability known as the corruption level. Doing this trains the autoencoder to reproduce the full input from a partially destroyed version of it. This has the benefits of preventing autoencoders with enough hidden units from learning an identity function (i.e. learning a combination of ones and zeros that directly map each visible unit to a single hidden unit), and also allows more hidden units than visible units to be used (Vincent, et al., 2008). It also avoids the need to add a regularisation term to the process to avoid overfitting.

2.2.2 Restricted Boltzmann Machines

The Restricted Boltzmann Machine (RBM) is a two-layer unsupervised network that is trained to learn a probability distribution over the training data, using neurons in the hidden layer. It is called restricted because unlike regular Boltzmann Machines, it only has weights linking units in the visible layer to units in the hidden layer and has no weights linking units within the layers themselves. The RBM tries to learn a joint configuration of visible units $\boldsymbol{v}$ and hidden units $\boldsymbol{h}$ with an energy function E given by

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i \in \boldsymbol{v}} a_i v_i - \sum_{j \in \boldsymbol{h}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

where: a_i and b_j are the biases applied to visible unit i and hidden unit j
v_i and h_j are the values of visible unit i and hidden unit j
w_{ij} is the symmetric weight between visible unit i and hidden unit j.

The hidden units of the RBM are sometimes known as *feature detectors* (Hinton, 2010) because they activate when certain combinations of the input units are present, and can be used to generate new synthetic reconstructions.

RBMs cannot practicably be trained by gradient descent on the derivative of the log-likelihood of the model, because it is very difficult to obtain an unbiased sample of estimate of the expected distribution of the visible and hidden units in the model, since this would require an infinite number of steps of Gibbs sampling. Therefore an approximation is used where the expected distribution of the model is replaced with the expected distribution of a reconstruction vector. This algorithm is known as *Contrastive Divergence*.

To implement Contrastive Divergence, firstly the values of the visible units are set to the values of a training vector. Then the values of the hidden units are calculated from the visible units (this can be done in parallel for computational efficiency) using:

$$p(h_j = 1|\boldsymbol{v}) = \sigma\left(b_j + \sum_i v_i w_{ij}\right)$$

where: the logistic sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

$p(h_j = 1|\boldsymbol{v})$ can be interpreted as the probability that h_j is 1 for binary data, or the actual value of h_j for real-valued data.

Then the values of the visible units are updated using the mirror of the previous update equation:

$$p(v_i = 1|\boldsymbol{h}) = \sigma\left(a_i + \sum_j h_j w_{ij}\right)$$

Then the first update equation is applied again. This is known as CD-1. Another application of the hidden and visible updates is known as CD-2 and so on. In practice a low value of x for CD-x often works well enough. Contrastive divergence is theoretically problematic as it does not approximate the gradient of any function, but is efficient and has been shown to work well in practice so is widely used.

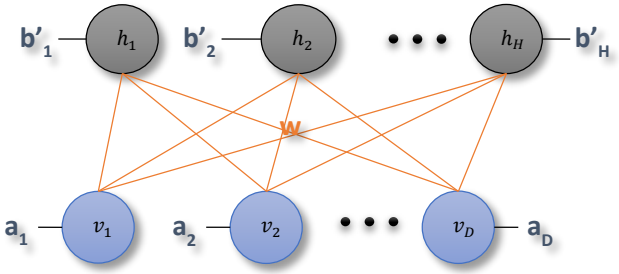


Figure 5 - A restricted Boltzmann machine

3 METHODS

3.1 EMULATION OF EXISTING SYSTEM

3.1.1 Programming languages and tools

The system used in (Bello, 2011) was developed in *Matlab* using various toolboxes and plug-ins (see Appendix C.1). During the project planning phase, it was identified that there would be several advantages and disadvantages to reimplementation of the system using Python (see Appendix C.3). On balance of these considerations, and in the context of the risk register (see Appendix C.2), it was decided that the advantages to reimplementation in *Python* outweighed the disadvantages. Therefore all development work was carried out using *Python 2.7*. The relevant components of existing *Matlab* toolkits were ported from *Matlab* into *Python*.

3.1.2 Communication with the author of the existing system

A number of attempts were made to communicate with Dr. Juan Pablo Bello, the author of (Bello, 2011). Unfortunately no responses were received to requests for information, datasets or code. Therefore it was necessary to re-implement the existing system and make certain necessary assumptions on its details.

3.1.3 Datasets

Dr. Bello used two distinct datasets for training and testing. The training set consisted of 123 recordings of 19 classical works written by eight composers. Just under half of these are piano recordings and the remainder are symphonic works. The study's test set was 2919 recordings of Frederic Chopin's Mazurkas for solo piano, known as the Mazurka Project (CHARM, 2009), which were collected at the Centre for the History and Analysis of Recorded Music. The motivation given for choosing a relatively monolithic test set is to better discern the method's ability to discern between different works in a similar style.

The test set used for the Project was also a dataset of performances of Mazurkas by Frederic Chopin, originating from the Mazurka Project (CHARM, 2009), and provided by the British Library. The original training set was requested from Dr. Bello but were not made available. Therefore a subset of all performances of twenty of the 49 mazurkas was partitioned for training and validation and the remainder were set aside for testing. Five performances of each of the mazurkas in the training set were used for iterative training. This was done for several reasons, these being:

- The shape of the training set was similar to that of the original study, in terms of the number of pieces and the number of performances of each piece.
- This size of training set allowed feature learning to be performed in a reasonable time using denoising autoencoders.
- This size of training set allowed the intermediate files used for ongoing system analysis and development to be stored in reasonable space on the network; there were not many recurrence plot files required for training but they could be in the order of hundreds of megabytes each. Also, while the sizes of files required to store NCDs were not significant, there were over 5,000 files required, even for a small training set of 100 performances. This represented a significant amount of disk I/O.
- Initial tests carried out using the partitioned training and validation set indicated that better results were obtained when twenty performances of each of five pieces were used rather than five performances of each of twenty pieces. However, the decision was taken to use more pieces

and fewer performances in order to try to minimise the risk that the feature learning would over-fit to relatively homogeneous training data.

The test set used was therefore all performances of each of the remaining 29 mazurkas, which had not been used in training or validation.

3.1.4 Feature down-sampling

The base feature rate used in (Bello, 2011) before down-sampling was 10 features per second. The features available for the Project had a rate of 22 features per second. It was considered impracticable both in terms of computational time and storage capacity to re-extract features from the source audio at the same rate as the (Bello, 2011). The difference in the original feature rates was therefore addressed by using higher feature down-sampling rates to achieve comparable rates in frames per second.

The method of down-sampling of input features were not specified in (Bello, 2011); therefore a simple arithmetic average was used.

3.1.5 Recurrence plot numerical parameters

To maintain consistency with (Bello, 2011), the values used for the time delay were {1, 2, 3, 4, 5}. The values used for the embedding dimension were also the same, i.e. {1, 2, 3, 4, 5, 6, 7}. The minimum and maximum values used in (Bello, 2011) for the neighbourhood size were 0.05 and 0.95. It was not clear what the step values used were, so in some cases where a larger range of values was being used for training, the step value was 0.1, and in cases using a smaller range a step value of 0.05 was used.

3.1.6 Recurrence plot methods

Three methods of generating recurrence plots were used in (Bello, 2011). The first, known as NEUC (normalised Euclidean), defines an additional threshold value called the neighbourhood size $\theta = \frac{\varepsilon}{2}$ where $0 \leq \theta \leq 1$. This choice for the relationship between θ and ε is described as helping *to maintain the range of θ constant across thresholding methods*; however the method was not in the original *CRP Toolbox*, and no clarification on its working was received so it was not used in the Project.

The second method, known as the *Fixed Amount of Nearest Neighbours* (FAN) method varies the distance threshold ε for for every time frame n so that the $\theta \times N$ closest points to any given point in the similarity matrix are set to one.

The third method, known as the *Fixed Recurrence Rate* (RR) method adjusts the distance threshold ε so that a fixed proportion θ of the points in the matrix are set to one, effectively defining the reciprocal of the Recurrence Plot sparsity.

The FAN and RR methods were both used in early experiments. After some initial comparisons the FAN method was found to give better results so was used for the remainder of the experiments.

3.1.7 Normalised Compression Distance

In an earlier publication (Bello, 2009), Dr. Bello tested two different compression algorithms to calculate the Normalised Compression Distance between each pair of recurrence plots, the *bzip2* (bzip2, 2010) and *PPMd* (Moffat, 1990) algorithms, included as part of the *CompLearn Toolkit* (Cillibrasi, et al., 2008). *bzip2* was shown to perform better, and was used exclusively in (Bello, 2011), so has also been used for the Project.

The best performance was shown to occur when the dimensions of the recurrence plots were equal. This was achieved in (Bello, 2011) by down-sampling the feature sequences input into the recurrence plots to a fixed size before compression. For reasons which will be discussed in section 5, the recurrence plots were resampled to a fixed size after calculation for the Project implementation. The value of this parameter is known as the NCD sequence length and takes values in the set {300, 500, 700, 900, 1100}.

3.2 Training, Validation and Testing

3.2.1 Training / validation / testing methodology

For problems with a short run-time, it is common to use cross-validation on the training set. This involves training an algorithm on one part of the data, and testing on a part that has not been used in the training. Cross-fold validation, typically using ten folds, iteratively trains parameters using 90% of the data and tests these parameters on an unseen 10%. This is repeated ten times, using a different 10% for the test set each time. The cross-validated result is then the average of the ten test results. However, it was found that a single iteration of an early implementation of the training algorithm used for the Project system could take several minutes to complete, depending on the choice of parameters. Therefore, it was decided not to use cross-fold validation, but instead to test some of the best-performing training settings on the larger validation set. Finally the best performing settings on the validation set were tested on the held-out test set for the best performing features.

3.2.2 Number of experiments and MAP calculations

Over 60 different experiments were conducted on the training set, most using the hill-jumping algorithm, which will be described in section 3.5.1, and some using another permutation algorithm to conduct a standard grid-search over more limited ranges of parameters. Each experiment ran for around 8 – 12 hours; typically daytime runs were 12 hours because there was more load from other users on the server during the daytime so it took longer to complete the same number of iterations.

Early versions of the system took up to ten minutes per run as the results of the recurrence plots and the NCDs were saved and loaded to disk at each iteration to allow inspection and analysis of intermediate results files. Towards the end of the experimentation, the amount of disk access was minimised so that there were only two significant disk accesses required per iteration, i.e. loading the features and at the start of the iteration and saving the MAP results and ranking table at the end. Where neural network parameters were included in the experiment, then weights were also loaded at the start of each iteration, but this did not constitute a significant overhead.

Despite the relatively long run-time of each MAP calculation, which was mainly due to the pairwise compression of recurrence plots, it was possible to achieve over 30, 500 MAP calculations by leveraging multithreading and running experiments on nearly every day of the Project execution stage. Appendix C.6 shows the best configuration and corresponding MAP result for each experiment.

Each MAP calculation on the training set involved 100 features files, and 5,049 NCD calculations, including the associated compression operations. For the validation set this increases to around 1000 feature files and half a million NCD calculations. For the test set this increases further to around 1600 feature files and 1.3 million NCD calculations.

Most of the more time-consuming code was parallelised to run on multiple processors using *Python*'s *multiprocessing* library. This allowed for more intensive calculation during periods when others were not using the server.

3.3 Feature Learning

3.3.1 Types of features

A number of different feature types were tried for representation learning: MFCCs, chroma, log-frequency spectrograms, constant-Q spectrograms and CENS features, which were calculated from the chroma features. Descriptions of these feature types for unfamiliar readers can be found in Appendix A. Early experiments using MFCC features confirmed the finding in (Bello, 2009) that they did not perform at all well for structural similarity measurements so MFCCs were excluded from the remainder of the project in order to focus efforts and computational time on more promising features.

3.3.2 Feature distributions

Histograms of the overall distributions of each of the main types of feature in the training set are shown in Figure 6(a). Upon first inspection, the chroma, constant-Q spectrogram and log-frequency spectrogram features look to be similarly distributed, apart from a higher number of features at the very low end (the scale of this difference is understated by the logarithmic axis). However, when the distributions are viewed in terms of their different frequency components, it becomes apparent that whilst the chroma distributions are approximately equal within each chroma bin, the frequency bins of the spectrograms are distributed in very different ranges, as shown in Figure 6(b).

(a) *(b)*

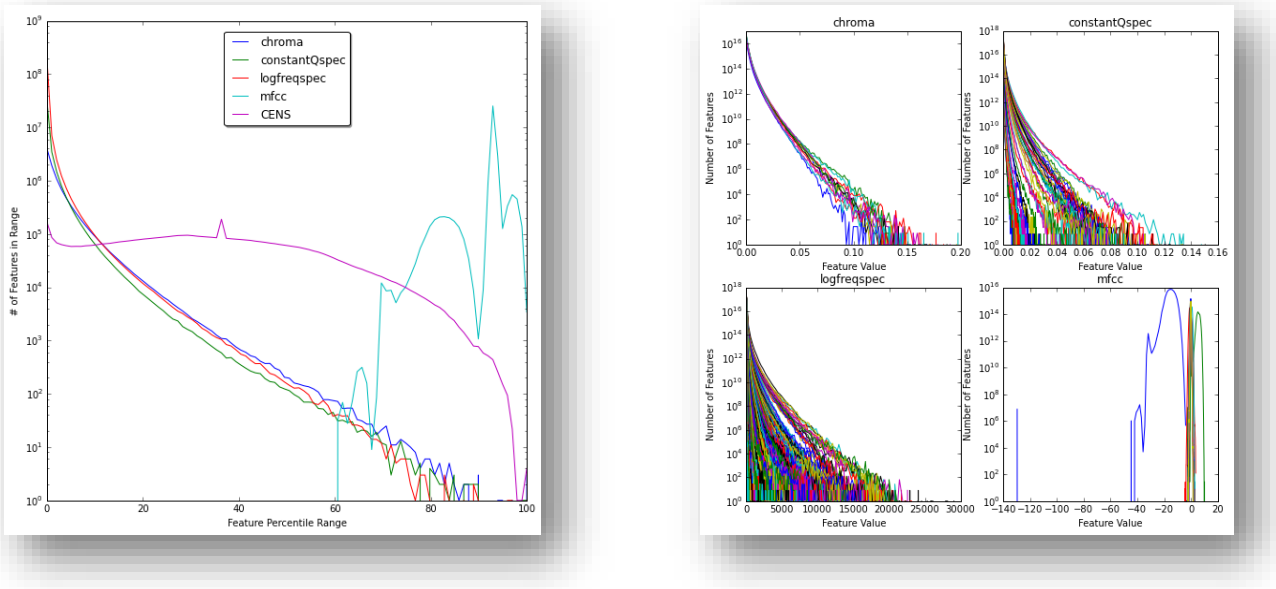


Figure 6 - Distributions of basic feature types in each of their range percentiles (a) for all components (b) in individual bins or frequencies.

3.3.3 Types of neural network

The original intention was to use as many types of neural network as possible. However, after initial experimentation the decision was taken to use only a denoising autoencoder, for the following reasons:

- This would allow for more exploration of hyper-parameters related to the neural networks.
- Early experimentation indicated that the denoising autoencoders were performing better at the structural similarity task than the restricted Boltzmann machines.
- The autoencoder has a simple error measure so it is more straightforward to compare networks and estimate their performance at the structural similarity task, as well as to know when to stop training them.
- The autoencoders were found to be approximately five times faster than the restricted Boltzmann machines for training purposes, allowing more networks to be trained.
- Replication of the existing similarity system was taking longer than had been planned to implement.

3.3.4 Creation of training files

The neural network library used was *Theano* (Bergstra, et al., 2010), which is a *Python* library optimised to carry out matrix multiplications using the GPU rather than CPU. Initial tests on a sample dataset indicated that the GPU learned from a dataset in approximately 5% of the time of the CPU. The input features had been extracted from the original source audio to *.csv* files using *Sonic Annotator* (Cannam, et al., 2010). The features were read into main memory, shuffled ten times, and then written to a training file. For the training stage, which used 5 performances of each piece, 70% of the features were used for training.

The original intention was to perform more representation learning on the validation set for the validation tests. Unfortunately, an unforeseen limitation with the size of arrays which are stored in the standard *Theano* file format was encountered so this additional learning was not possible.

3.3.5 Input feature range standardisation

To effectively train a neural network, a common practice is to standardise the range of the input data before feeding it into the network, typically to the range from 0 to 1. Figure 7 shows the maximum values for each frequency component of the training set constant-Q-spectrograms and log-frequency spectrograms. It can be seen that different frequency components of the features occupy different ranges. Therefore it is necessary to decide whether to standardise all frequency components using the overall range of values or to standardise each component to its own range. Standardising them all together preserves the inequality in the ranges; however, this may not be desirable if frequencies in the upper range of components are important for the subsequent similarity measurements. Both methods were tried during the training stage to determine which was better.

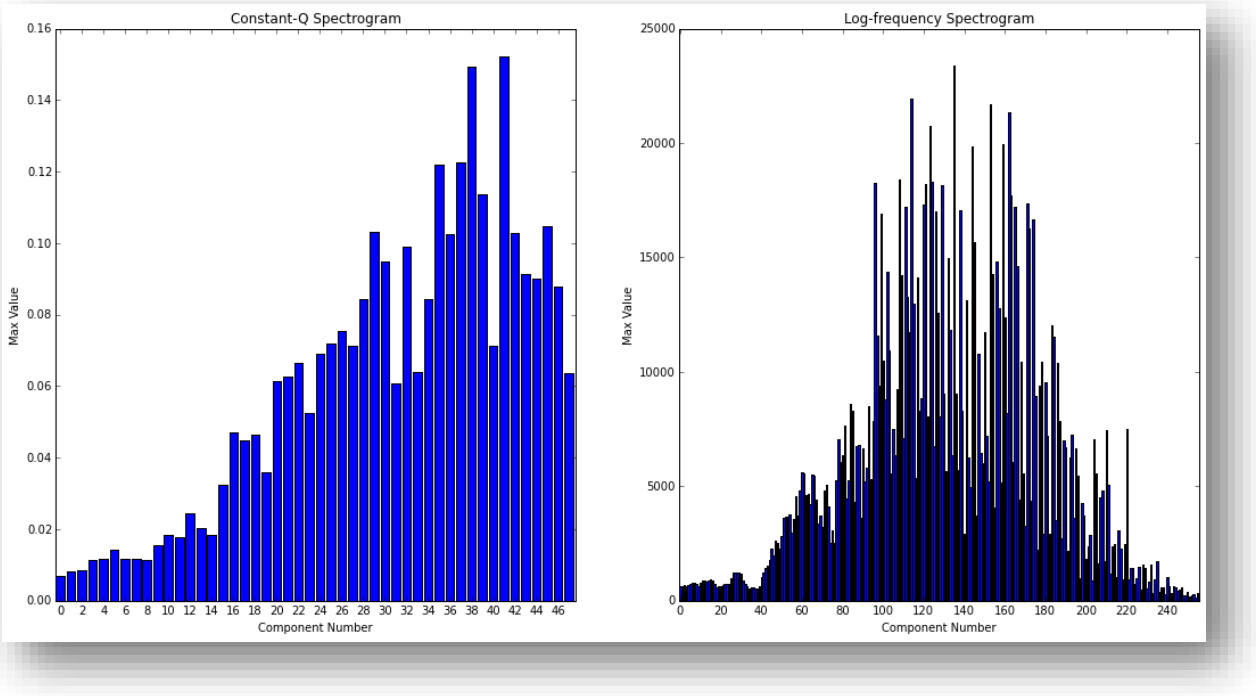


Figure 7 - Maximum values for each frequency component in the training set for the constant-Q spectrograms and log-frequency spectrograms.

3.3.6 Network dimensions

There are two dimensions which can be set in a neural network with a single hidden layer. The first is the input layer, which is simply the number of features used from the input dataset. For the chroma-based features and constant-Q spectrograms, the frequency components are already closely related to the range of musical frequencies (see Table 1). Therefore using fewer features as visible units would be likely to discard valuable information that could be used in the structural similarity calculations.

Parameter	Chroma	Constant-Q Spectrogram
"minpitch" parameter	36 (C2)	36 (C2)
"maxpitch" parameter	96 (C7)	84 (C6)

Table 1 - pitch settings parameters and associated musical notes used in the calculation of the input features using Sonic Annotator.

For the log-frequency spectrograms, which have 256 frequency components, training was conducted using the full range of frequencies and also using only the lower half of the frequency range. The decision to do this was based on the hypothesis that the upper half of the frequency range, corresponding approximately to the range from 10 kHz to 20 kHz may not be particularly useful for the similarity measurements, since higher frequencies were more likely to contain information related to the instrumental timbre such as transients and other noise. This type of information would be less useful than the harmonic musical information in the lower half of the range, and it would perhaps be detrimental to include it. This hypothesis was partially informed by domain knowledge, and also by the result in (Bello, 2009) that MFCCs, which use higher frequency spectral content, were not useful for structural similarity measurements.

3.3.7 Batch size

The fastest way to train a feed-forward neural network is to update the weights after one or more epochs of the training examples being presented to the network; the slowest but potentially safest way is to update the weights after each individual example has been presented. For practical purposes, it is desirable to strike a favourable balance between speed of training and accuracy of the trained network. A recipe for setting the batch sizes for training RBMs is given in (Hinton, 2010). The document recommends the following:

> *For datasets that contain a small number of equiprobable classes, the ideal mini-batch size is often equal to the number of classes and each mini-batch should contain one example of each class to reduce the sampling error when estimating the gradient for the whole training set from a single mini-batch. For other datasets, first randomize the order of the training examples then use minibatches of size about 10.*

Batch sizes of 10, 20 and 40 were used in early experiments. After some experimentation it was decided to use 40 for the remainder of experiments for reasons of consistency, training time and the approximate number of classes that would be used in the final test.

3.3.8 Learning rate

The learning rate is the rate at which the weights and biases of a network are adjusted to minimise the loss function of the network. One common training methodology used is to begin with a high learning rate and gradually reduce it towards the end of training so that minima are not missed by taking too large a step towards them and stepping over them. However, when more time is available, it can be advantageous to use a lower learning rate throughout the training as this can find deeper minima in the landscape of the objective function. This was the approach taken to train the denoising autoencoders on the feature data.

In the early stages of training, inspection of the final network reconstruction errors showed that the minima reached in training by networks of different architectures were inconsistent. This was found to be due to the size of the decrease in the reconstruction error being less than the stopping criterion for certain configurations. Simply lowering the reconstruction error stopping criterion did not resolve this problem and only led to longer training times (some networks were already taking several hours to train, depending on the settings used). Therefore an additional parameter was added called the learning rate boost factor, which is the amount by which the learning rate was increased towards the end of training, when the change in reconstruction error criterion was not met after an epoch of training. This modification was able to find deeper minima, and inspection of the training results confirmed that there was a better correlation between the number of epochs required and the reconstruction error. This implied that the training method was more consistent than without increasing the learning rate towards the end of training.

3.3.9 Corruption level

The corruption level is the probability that any individual input unit will be set to zero (corrupted) in the network training process. Initial experiments used a corruption level at steps of 10% from 0% to 90%. After a few experiments it was observed that corruption levels above 70% were producing very poor results so the range was decreased to 0% to 70% to reduce network training times and focus the parameter searches. The increase in reconstruction error at a 70% corruption level (shown on the vertical axis as 0.7) can be seen in Figure 8.

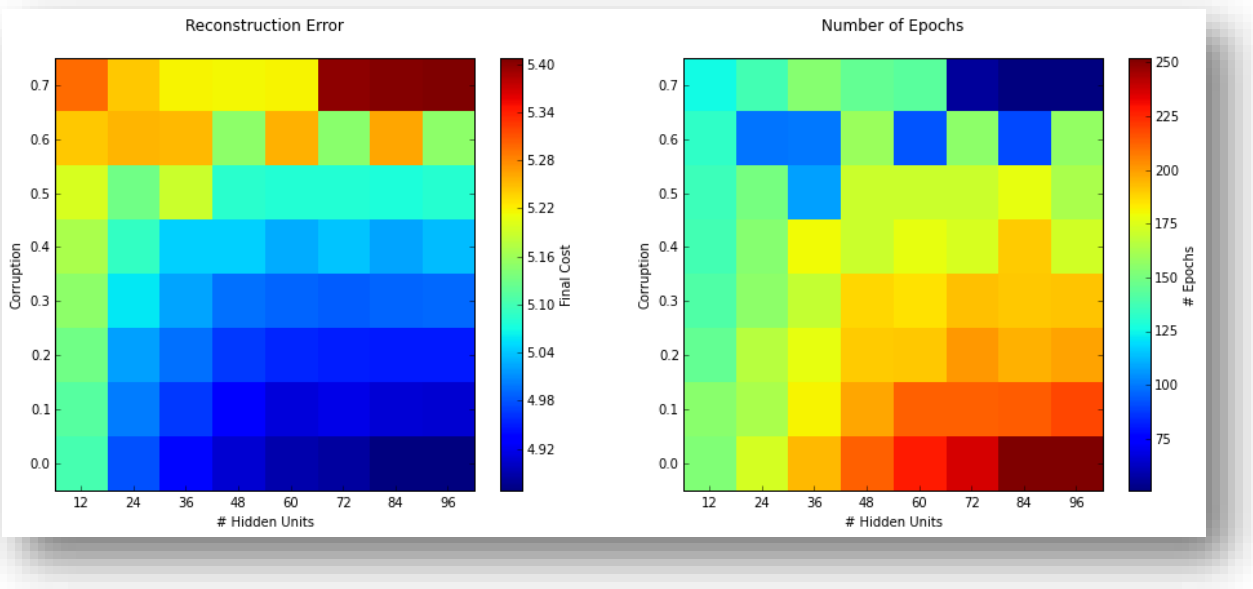


Figure 8 - Illustration of the final reconstruction errors (cost function) and numbers of epochs taken to train a denoising autoencoder on the log-frequency spectrum training set (256 visible units).

3.3.10 Incorporation of temporal effects

In (Bello, 2011), the CENS features used, which incorporate time averaging of several frames of input features, showed a significant improvement over standard chroma features. Therefore it was decided to test the effects of training the neural network on the concatenation of several frames of input features at a time, known in the context of the Report as *time-stacking*. The weights learned from this learning were then applied in the structural similarity calculations to time-stacked versions of the input features. This introduced a significant overhead into both processes, in terms of training time and storage for the neural network training files, and in terms of training time for the structural similarity training. The concept is illustrated in Figure 9.

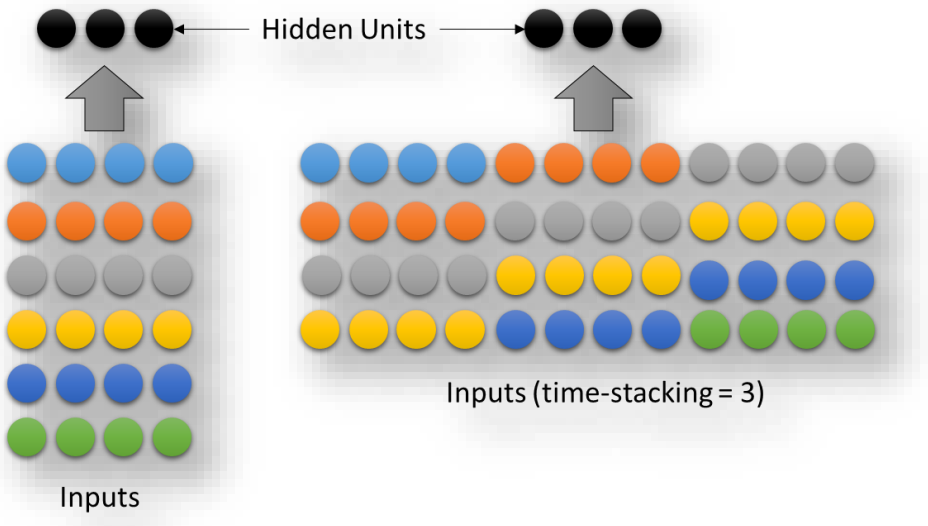


Figure 9 - An illustration of learning from a four-component input feature vector, with and without time-stacking applied. N.B. in reality, inputs are shuffled before being presented to the network.

3.4 CREATION OF NEW CENS-LIKE FEATURES

In order to incorporate the time-averaging properties of CENS features whilst avoiding the costly process of time-stacking, yet still taking advantage of the neural network's ability to reduce the number of features, new features were developed inspired by CENS. These were created by emulating the process used to create CENS features from chroma features, but using machine-learned compressed representations of the original base features, instead of the features themselves. For the purposes of the Report these will be known as FENS (Feature Energy Normalised Statistics) as they use and build upon the existing algorithm for creation of CENS features and apply it to any two-dimensional feature time series. The new and borrowed FENS creation steps are illustrated in Figure 10. The code written for creating CENS and FENS features is listed in Appendix F.1. It is recommended that readers unfamiliar with CENS features refer to Appendix A.6 before reading the remainder of this section.

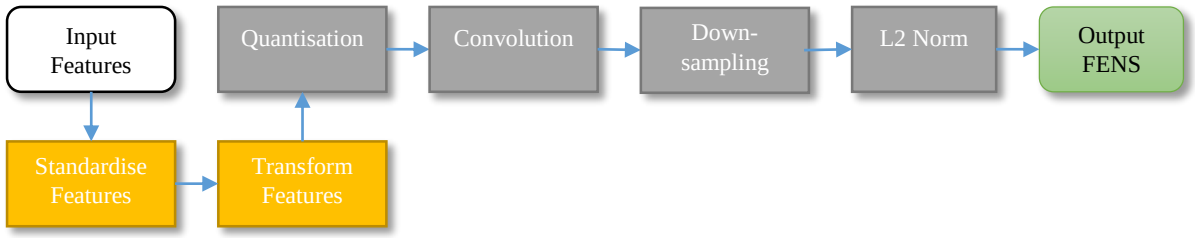


Figure 10 - Illustration of the steps involved in calculation of Feature Energy-Normalised Statistics. Additional steps not used in the CENS calculation process are shown in yellow.

3.4.1 Number of features

For calculation of CENS from chroma the number of feature components is fixed to twelve. However, for practical purposes this can be adjusted to fit the number of dimensions of another type of feature with different dimensionality. For learned features, the number of components, equating to the number of hidden units in the autoencoder, was varied with the values which had been run in the network training (see Appendix A.1).

3.4.2 Input feature transformation

An optional transformation function can be applied at the input stage to modify the distribution of the data into a shape where it is likely to be distributed more favourably amongst the quantisation buckets. After rescaling the features to the range from 0 to 1 to ensure that range errors were not encountered during execution of the transformation functions (e.g. taking the square root of a negative number), some common variable transformations were applied to the feature data, as illustrated in Figure 11.

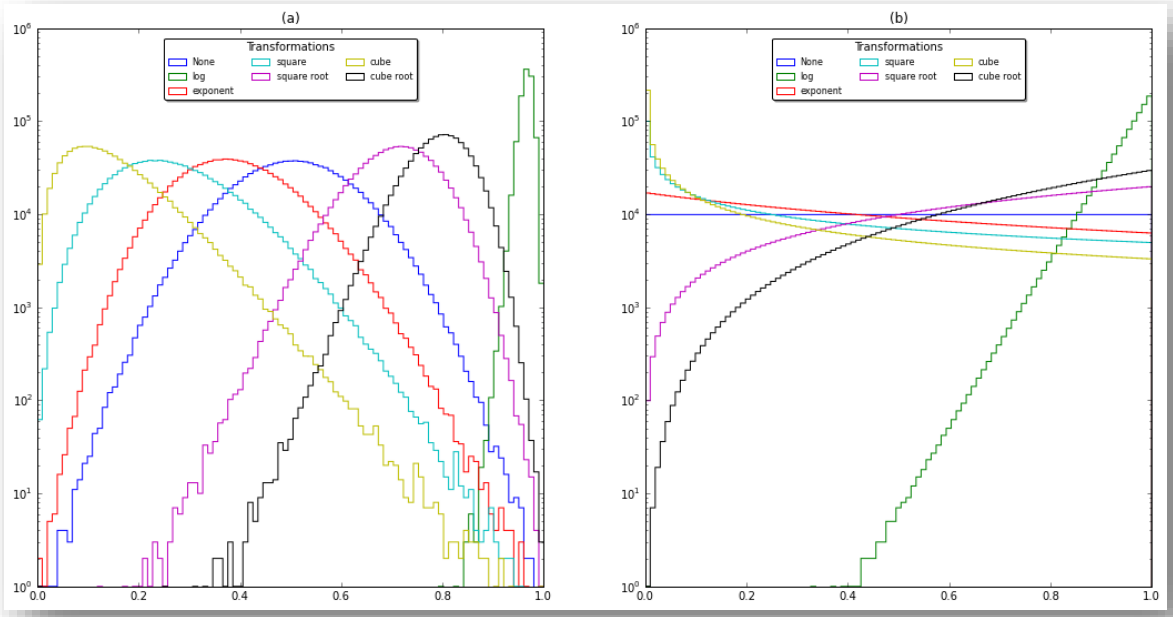


Figure 11 - Common variable transformations applied to (a) a normal distribution and, (b) a uniform distribution.

3.4.3 Normalisation threshold

All feature frame components with a level below a small absolute threshold value are excluded from the subsequent energy calculation. This can be likened to the subtraction of a noise floor, below which features are not considered useful for the subsequent quantisation and down-sampling.

Each feature component is then normalised to the sum of the values within its time frame such that it expresses the relative proportion of that component relative to the entire frame. Since the ranges of values of the individual feature components are very different (see the x-axes of Figure 6(b)), a standardisation function is applied to reduce the ranges of the features and their associated noise floors to a similar level to those of chroma features (the input features in the training set were distributed in the range from 0 to 0.2).

3.4.4 Quantisation

Higher dimensional features may require lower quantisation threshold steps in order to adequately differentiate between different feature components in the energy calculation. Feature components with a level above each of the quantisation step values are then assigned the value of the product of the step value and its associated quantisation weights.

The defaults found in the original *Chroma Toolbox Matlab* code for creating CENS features from chroma features are [1, 1, 1, 1] for the weights and [0.4, 0.2, 0.1, 0.05] for the threshold steps. These quantities were varied in the training to find the best settings for the given base feature.

3.4.5 Down-sampling

The feature sequence is forward-backward filtered with a Hann window of a given length and then down-sampled to a specified factor. The values for calculation of CENS features from chroma features given in (Muller, et al., 2005) are a window length of 41 and a down-sampling factor of 10. These parameters were also varied in the training to find their best settings.

3.5 TRAINING ALGORITHMS

3.5.1 Greedy, hill-jumping algorithm for optimal parameter search

The number of possible combinations of settings of parameters used in (Bello, 2011) is approximately 2400 (see Appendix C.4). With all the additional settings added for the Project due to the introduction of neural networks and energy normalised statistics calculations, the number of possible settings increases exponentially, to over 4×10^{12}, making it impossible to test all combinations. This was the motivation to develop an efficient algorithm to search over the parameter space.

A widely-used machine learning algorithm is gradient ascent (or descent, depending on whether the objective function is to be maximised or minimised). Gradient ascent works by measuring the difference in the objective function for neighbouring regions of settings, and taking a step in the direction with the steepest gradient, i.e. choosing the nearby settings that most increase the value of the objective function.

For problems involving discrete parameter settings, gradient ascent suffers from the problem that it can be slow for high dimensional spaces, as the number of different possible settings in the neighbourhood of the current settings increases exponentially (for a problem with categorical parameters, then assuming the algorithm must search in two directions in each dimension $d \in D$ to determine the gradient, then the number of settings it must test at each decision stage increases with D^2). It also makes an assumption that the objective function space is monotonically increasing, and therefore when this assumption does not hold, it is prone to the problem of locating local maxima in the objective function.

The greedy, hill-jumping algorithm developed for the Project searches across a whole single dimension of parameters and picks the one which increases the value of the objective function by the most. If no increase in the objective function was found in the chosen dimension, then the next dimension is tried, and so on until the algorithm has searched across all of the dimensions without finding an improved value for the objective function. At this point it has found a local optimum and the training begins again at a random location in the parameter space. A fuller analysis of the algorithm is given in Appendix B.

3.5.2 Fine-tuning of feature transformation weights

The most common method of fine-tuning the weights and biases of neural networks is to use the backpropagation algorithm. This is performed by feeding an error signal back through the network starting at the output units and ending at the weights between the input layer and the first hidden layer. In the case of an autoencoder, the initial error is calculated by comparing the output of the trained network with its input. The difference between these two quantities is called the reconstruction error.

In the case of structural music similarity measurements, the goal is to maximise the overall MAP performance of the system, rather than minimising the reconstruction error of the network, (although it is assumed that in general, a system with a lower reconstruction error is the latter is more likely to produce a lower MAP score, so long as the network has not over-trained). Therefore, it is not possible to use the common method of backpropagation to fine-tune the weights learned during feature learning because the ground truth (the target values for the hidden representational layer) is not known.

However, another method of fine-tuning weights exists, which is not commonly used in neural network training due to its relative inefficiency, randomness and lack of elegance in comparison with backpropagation. This is the method of adding small amounts of random noise to the weights and comparing the output with the objective function of the network. In the case of the Project, once initial weights and biases have been learned from unsupervised learning, small random adjustments can be made to these weights and biases. If the MAP performance of the structural similarity system improves then the adjustments can be retained, otherwise they are discarded.

The code written for the fine-tuning algorithm is given in Appendix F.8. There is a parameter to restart with the initial weights once a certain number of adjustments has been made to the weights without any increase in the MAP score, which was set to 50.

3.5.3 Coercion of FENS features into a Gaussian distribution

Inspection of the distribution of the CENS features for a down-sampling rate of 10 and a Hann window length of 41 (see Figure 12) illustrates that the values of each CENS bin resemble a Gaussian distribution centred around 0.22, with a distinctive spike at the value 0.28, and a sharp increase in the number of values towards 0. CENS features for other settings of down-sampling rate and window length. This implies that the thresholds and weights applied to the input chroma features, which follow an exponential distribution, have efficiently and evenly scattered the values within each chroma bin to a representation with higher entropy.

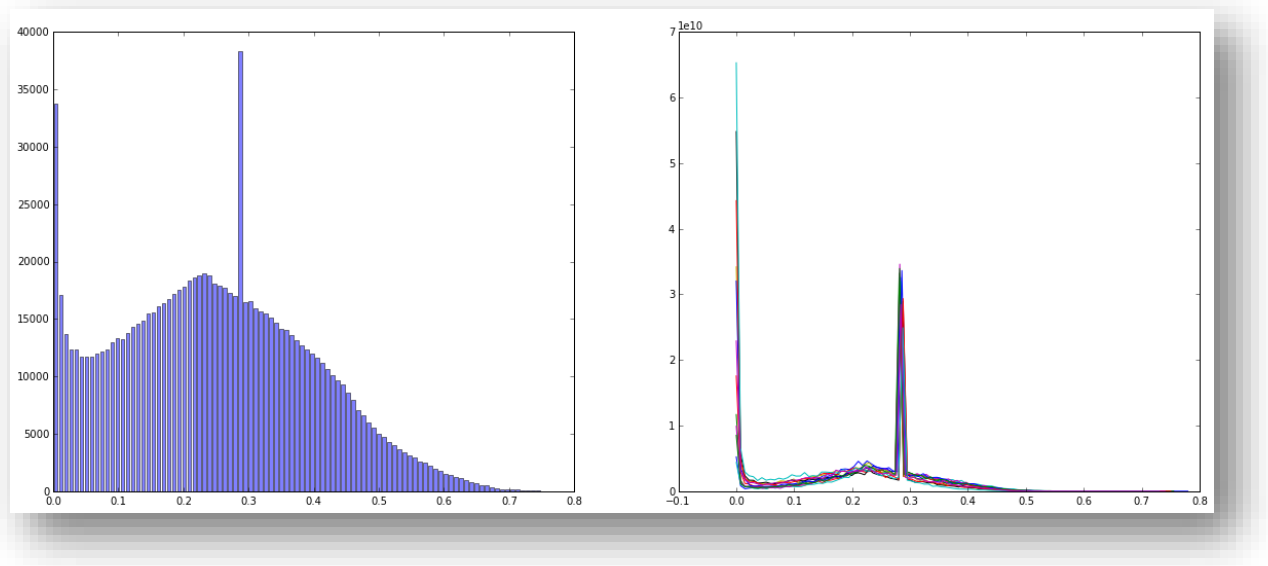


Figure 12 - Joint and individual chroma bin distributions of CENS features calculated from the training set using the settings suggested in (Muller, et al., 2005).

The distributions of the learned neural network features do not follow a similar distribution to chroma features however, so a method was developed to coerce the features into a form closer to a Gaussian distribution to determine if this would improve the performance of the overall system (see F.9 for code).

The method used begins with a predetermined distribution of weights, and threshold and normalisation values, and varies each by an amount randomly drawn from a uniform distribution within a user-specified range until the mean squared error between the distribution of the FENS features and an ideal normal distribution is minimised.

4 RESULTS

4.1 EFFECTS OF PARAMETER SETTINGS ON MAP RESULTS

There was found to be a high level of interaction between the set of parameters tested, i.e. different parameters were generally not independent to each other with respect to their effects on the overall MAP performance of the system. This subsection illustrates and describes some of the more important interactions that were found, and are illustrated by varying two to four parameters of a fairly well performing configuration of the overall system on the training set. This base system configuration is set out in Table 2. The log-frequency spectrum was chosen for illustrative purposes because it allows variations of most of the parameters to be tested.

Parameter	Setting
Feature Type	Log-frequency spectrogram
Feature Down-sampling	4 (5.5 frames per second)
Number of Visible Units	128 (half of full frequency range)
Number of Hidden Units	12
Learning Rate	0.05
Corruption Level	10%
Time Stacking	1 (no stacking)
Frequency Standardisation	No
RP Method	FAN
RP Embedding Dimension	1
RP Time Delay	4
RP Neighbourhood Size	0.05
NCD Sequence Length	500

Table 2 - Base system configuration for exploration of parameters.

4.1.1 Feature Down-sampling vs. NCD Sequence Length

The feature down-sampling and NCD sequence length operations are closely related because they both involve setting the size of the recurrence plot in different ways. The feature down-sampling sets the effective frame rate at the start of the process and therefore has a downstream impact on not only the NCD calculation but also the form of the Recurrence Plots; the setting of the NCD sequence length is intended to equalise the sizes of the recurrence plots so that the NCD measure is as reliable as possible, but how much the plot changes from this down-sampling is dependent on the difference between the number of features that result from the down-sampling and the fixed sequence length.

Figure 13 shows the MAP results for different feature down-sampling rates and sequence lengths; there is a hot-spot where the chosen sequence length matches well with the down-sampling factor at sequence length = 500, down-sampling = 4 (5.5 frames per second).

Concretely, the higher the value for input feature down-sampling, and the lower the consequent feature rate, then the smaller the dimensions of the resulting recurrence plot would be without further changes to the sequence length. Smaller recurrence plots will be altered less by lower values for the NCD sequence length and visa-versa.

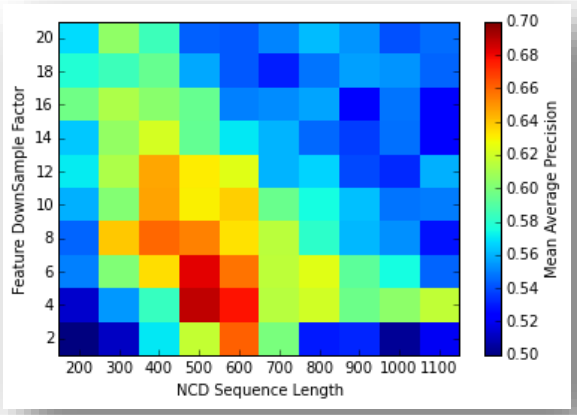


Figure 13 - MAP results for different combinations of NCD sequence lengths and feature down-sampling rates.

4.1.2 Recurrence Plot Parameters

A full treatment of the parameters of the Recurrence Plots is given in (Bello, 2011). A number of important properties of the interactions between the neighbourhood size, embedding dimension and time delay parameters can be observed in Figure 14.

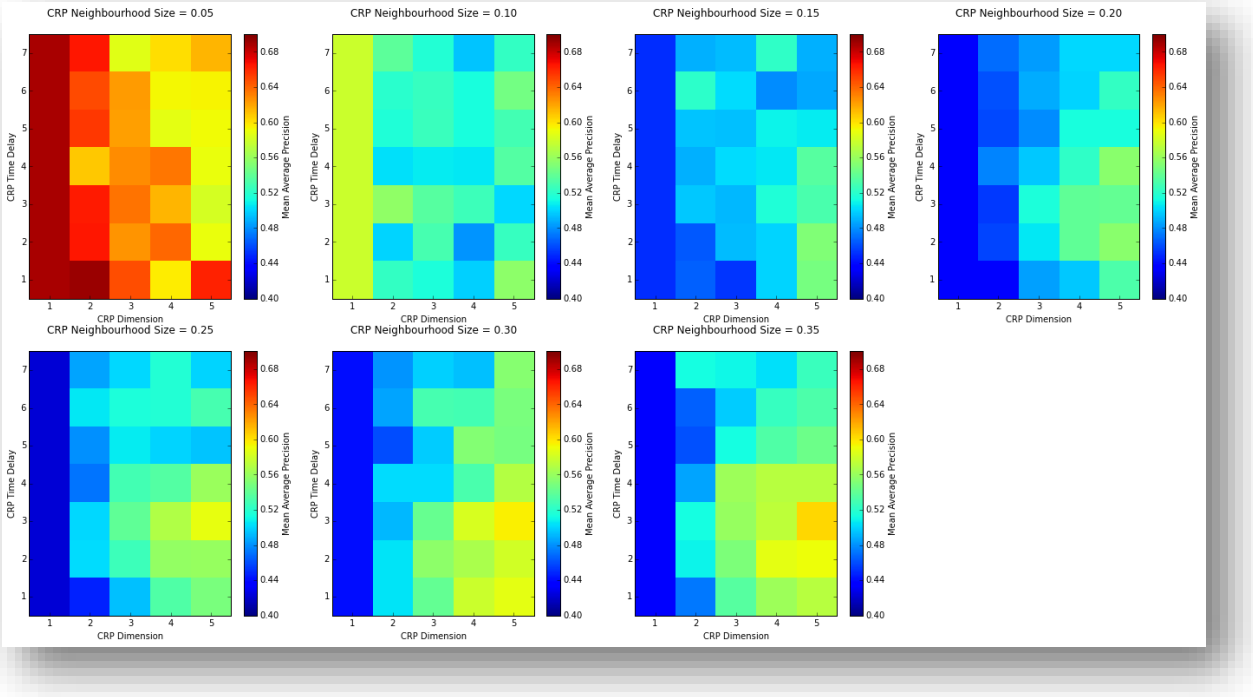


Figure 14 - Summary of the MAP performance of the system from varying the main recurrence plot calculation parameters.

The first thing to note is the vertical bars of identical values along the left-hand side of each plot, corresponding to an embedding dimension of 1. When the embedding dimensionality of the plot is 1, then the recurrence plot represents only the similarity between corresponding points of time within the piece, not corresponding sequences of points. As the neighbourhood size is increased, the MAP performance of the parameter space represented by the bar inverts from the highest performing region of the settings space to the lowest performing region.

As the neighbourhood size and embedding dimension are increased, an area of higher performance emerges to the right of the plots. This indicates that allowing for a larger amount of nearest neighbours to be assigned as similar is better able to identify repeating sequences of notes. However, for this particular configuration, the effect is not strong enough to outperform the single embedding-dimension, low neighbourhood size bar. It is possible that using much higher embedding dimensions would allow for detection of a stronger region; however this was not within the parameter range used in (Bello, 2011), and would have greatly increased the calculation time and memory requirements.

4.1.3 Number of hidden units

To optimise the speed of the structural similarity system, it was desirable to use as few hidden units as possible to learn representations of the base features. However, this needed to be balanced against using sufficient hidden units to capture enough of the useful information in the distribution to adequately represent the similarities and differences between features from different performances. Experiments demonstrated that a small level of corruption (10% - 20%) generally performed better than higher levels of corruption or using no corruption at all (Figure 15a). Also the assumption that networks that could be trained to have a lower reconstruction error would produce a better MAP result held for different numbers of hidden units (Figure 15b).

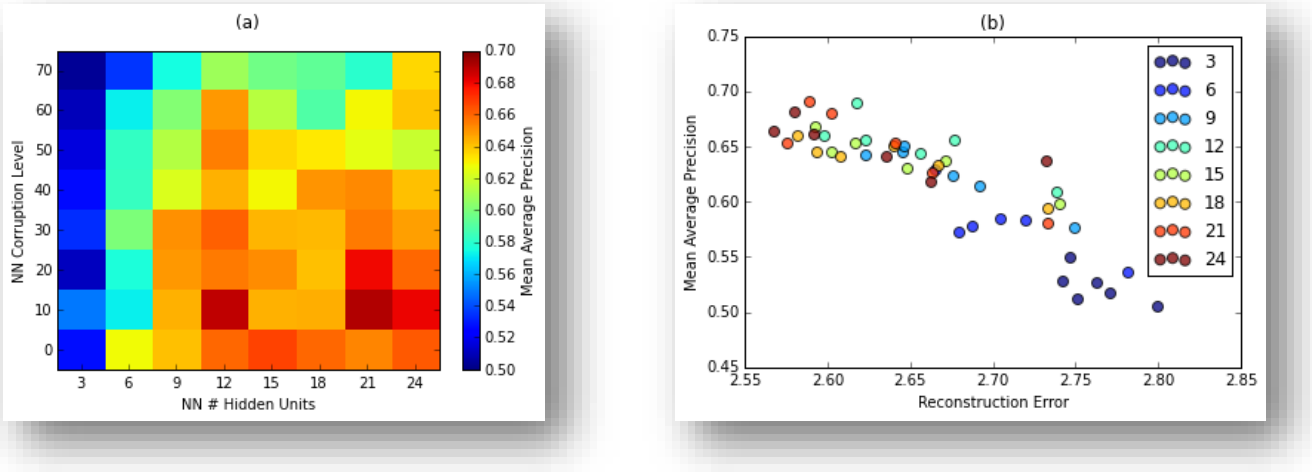


Figure 15 - MAP training results for a fixed configuration of parameters (a) varying the number of hidden units and corruption level, and (b) the same results set plotted against the final network reconstruction errors, coloured by number of hidden units used.

4.1.4 Neural network training batch size

Figure 16 shows the MAP results found across different embedding dimensions and time delays using neural networks trained on batch sizes of 10 and 40 units. The network trained on 10 inputs at a time is able to find localised settings which give a higher MAP value, and has higher mean and median values. This indicates that using a small batch size, perhaps equal to the number of classes, as suggested in (Hinton, 2010), may yield better overall system performance. Unfortunately, there was not sufficient time to train the variety of networks architectures using such a size.

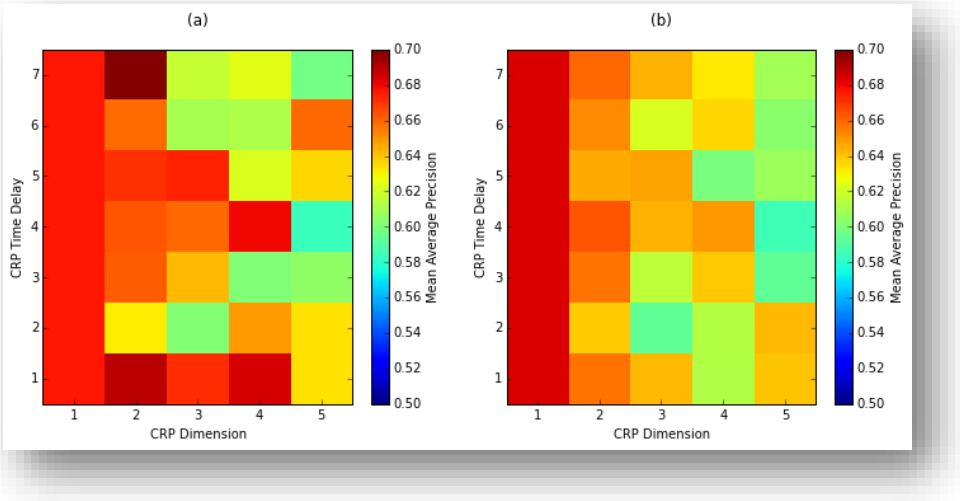


Figure 16 - MAP results found for batch sizes of (a) 10 and (b) 40 inputs.

4.1.5 Frequency standardisation / number of input features

Figure 17 shows the effect on calculated MAP values, obtained across embedding dimensions and time delays, of using individual frequency standardisation and also using the full range and half the range of features for the log-frequency spectrogram. While the embedding dimension and time delay parameters appear to be independent of variations in frequency range and standardisation, it is clear that for both ranges of frequencies, standardising the individual frequencies has a detrimental effect on the performance of the system. This implies that frequency standardisation, which amplifies the smaller ranges of components shown in Figure 7, does not assist in the similarity measurements.

The detrimental effect is greater for the full range of frequencies than when only using the lower half. This lends further weight to the hypothesis that the upper range of frequencies are less important for the task of structural similarity than the lower half. It should also be noted that the variation across the range of frequency components in Figure 7 is not a smooth curve, and this is likely to lead to overfitting when used with small training sets. It is possible that low pass filtering the curve and standardising to the smoothed curve could work better.

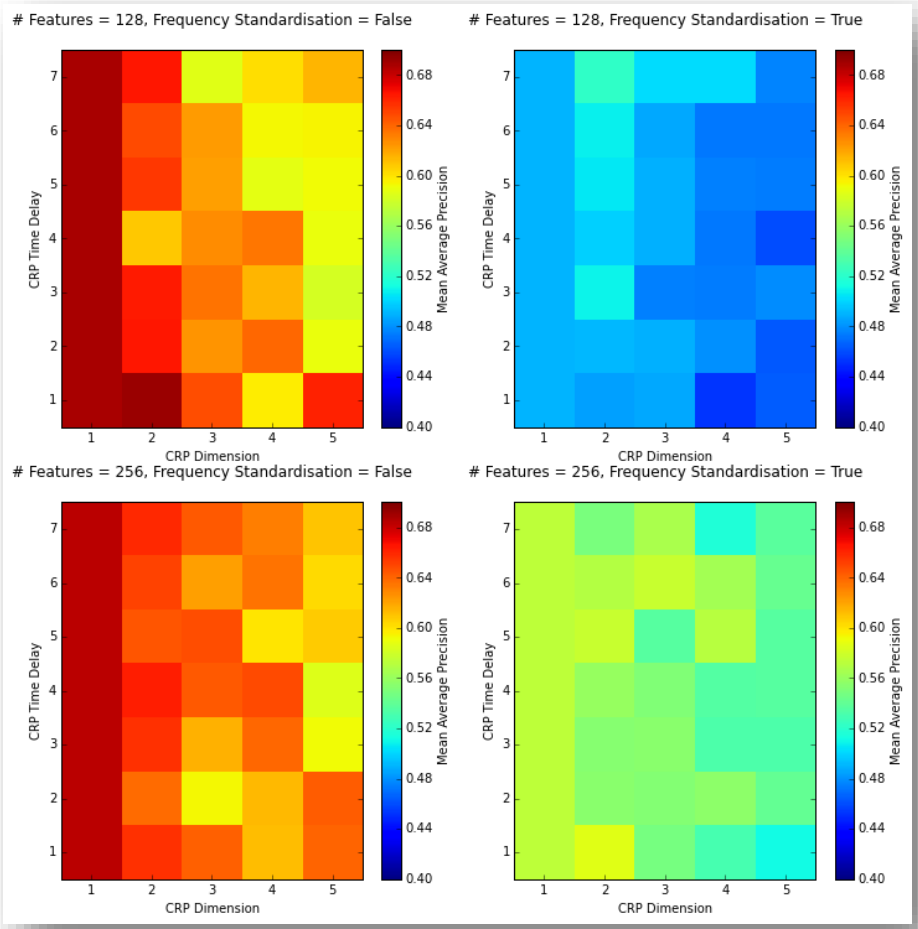


Figure 17 - MAP results using all and half of the frequency range, with and without frequency standardisation.

4.1.6 Time-stacking

Limitations in memory and disk-space prevented time-stacking of the log-frequency spectrograms. Figure 18 shows a comparison of no time stacking (48 features) with 10x time stacking (480 features) for the constant-Q spectrogram, using the base configuration settings for the other parameters. While MAP results are generally higher for the 1x setting, the familiar bar for the embedding dimension parameter of 1 is actually higher for the 10x setting, implying that the incorporation of more than one time frame helps the calculation for the case of m = 1 for the constant-Q spectrogram. However, it should be noted that there are higher MAP values calculated for the 1x time-stacking case at other recurrence plot embedding dimensions.

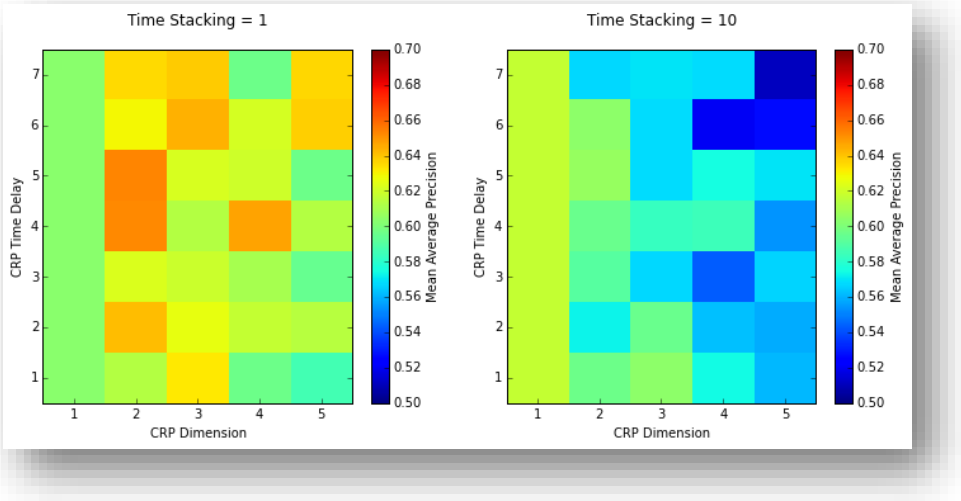


Figure 18 - MAP results for two time-stacking settings, 1x and 10x

4.1.7 Fine-tuning of feature weights

The weights fine-tuning algorithm was run on two configurations of weights learned from autoencoders on CENS features with a down-sampling rate of 2 and a window length of 41. The first run increased the score from 0.707 to 0.727. The second run increased the score from 0.754 to 0.764.

The improvements found on the training set did not translate to improvements on the validation set, indicating that the fine-tuning process may be overfitting to peculiarities of the training set. Therefore the weights learned from the fine-tuning were not included in the final tests.

4.1.8 FENS transformation functions

Figure 19 shows the MAP values calculated for different input transformation functions applied to the weight-transformed log-frequency spectrogram training set. Blank values indicate regions where a result could not be calculated. Since the input features have been standardised to the range from 0 to 1 before application of the transformation function, the *cube*, *square* and *exponent* functions have the effect of pushing the distribution of input features towards zero, whereas the *square root*, *cube root* and *log* functions push the distribution closer towards the value of one (see Figure 11). The plots in Figure 19 have been ordered so that the functions which push the value closer to zero are in the top row and those which push the values towards one are in the bottom row. Those which push the distribution closer to zero result in a higher MAP performance, with the *cube* function performing best. This is because as the features are pushed towards zero, they begin to more closely resemble the distribution of chroma features shown in Figure 6(b), which is what the original CENS algorithm was designed for.

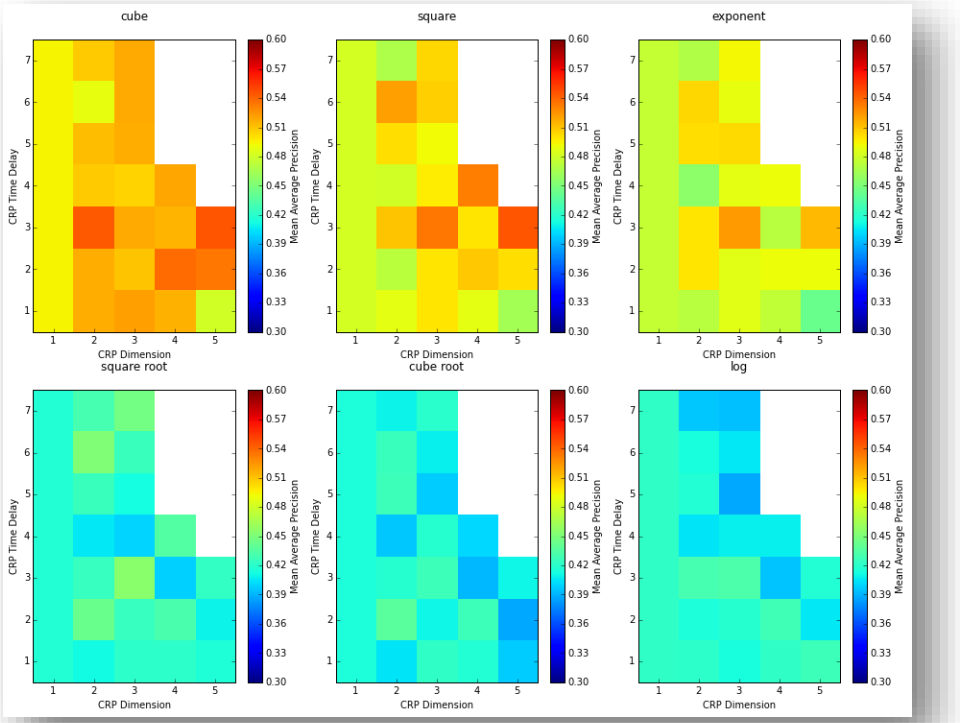


Figure 19 - Comparison of MAP values calculated on FENS features with various transformation functions applied.

4.1.9 FENS down-sampling rate and window length

A down-sampling rate of 10 and window length of 41 are recommended in (Muller, et al., 2005) for the calculation of CENS features from chroma. However, these are not necessarily the best-performing settings for the calculation of energy-normalised statistics from features learned from neural networks.

Figure 20 shows comparisons of calculated MAP results for different FENS down-sampling rates and window lengths on the weight-transformed log-frequency spectrogram training set. The best value for the down-sampling rate is either 2 or 4 depending on the sequence length.

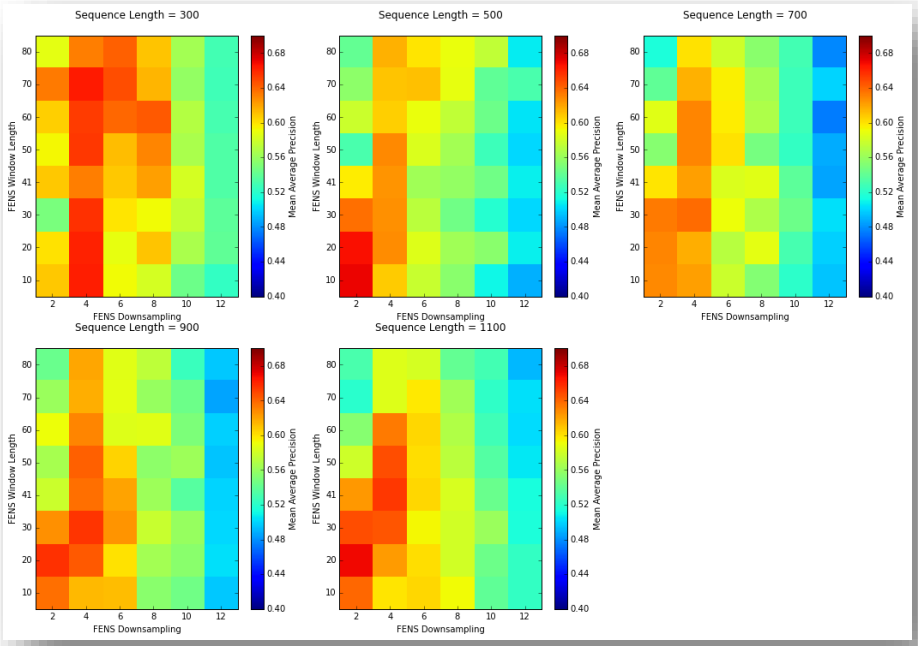


Figure 20 - Comparison of MAP results for different combination of FENS down-sampling and window length for 5 different settings of the NCD sequence length.

4.1.10 FENS quantisation steps

For CENS features, the levels (steps) at which the transformed features are thresholded are {0.05, 0.1, 0.2, 0.4}, covering a total energy range of 0.35 (0.4 – 0.05). This can be equated to a base of 0.05, raised to the power of two, successively. To investigate the best steps for FENS features, the base was varied from 0.01 to 0.09 and the power was varied from 1.8 to 2.2, which varies the total range covered by the steps from around 0.06 to 0.96. This should allow the thresholding process to find the best differentiation of energies for features which have different distributions to typical chroma, while the energy concept is preserved by maintaining the base/power step relationship.

Figure 20 shows the variation in MAP performance found for different settings of the quantisation steps. The standard CENS base of 0.05 is clearly better than other values, although a power of 1.9 performs better than the standard power of two relationship.

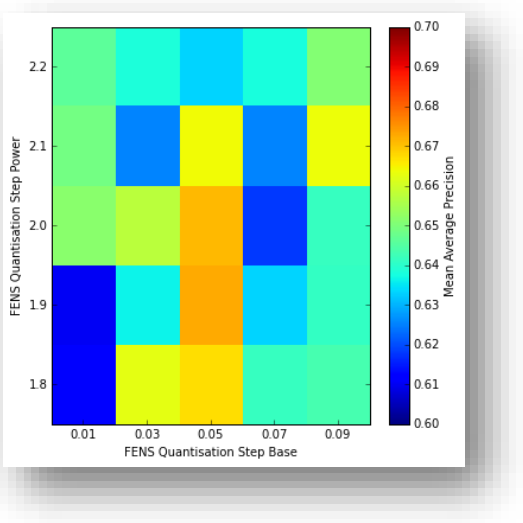


Figure 21 - MAP scores for different settings of the FENS quantisation threshold steps.

4.1.11 FENS quantisation weights

Figure 22 shows the variation in the calculated MAP score with different variations of the FENS quantisation weights. Weights 1 and 2 are constant within each subplot, and increase along the horizontal and vertical axes of the whole plot. Weights 3 and 4 are varied along the horizontal and vertical axes of each subplot.

The first observation is that MAP scores are higher when weight 2 is slightly higher than weight 1, as illustrated by the diagonal pattern of high MAP scores in the subplots immediately above the bottom-left to top-right diagonal.

Weights 1 and 2 are more important than weights 3 and 4 as evidenced by the observation that there is generally less variation in the MAP score within any one subplot than there is variation between corresponding squares of different subplots. This is likely because there are far fewer features in the quantisation buckets corresponding to high energy levels governed by weights 3 and 4. The occasional outliers within subplots can be explained by overfitting to small variations in the upper levels of the energy distribution (see the right-hand side of the chroma distributions in Figure 6(b).

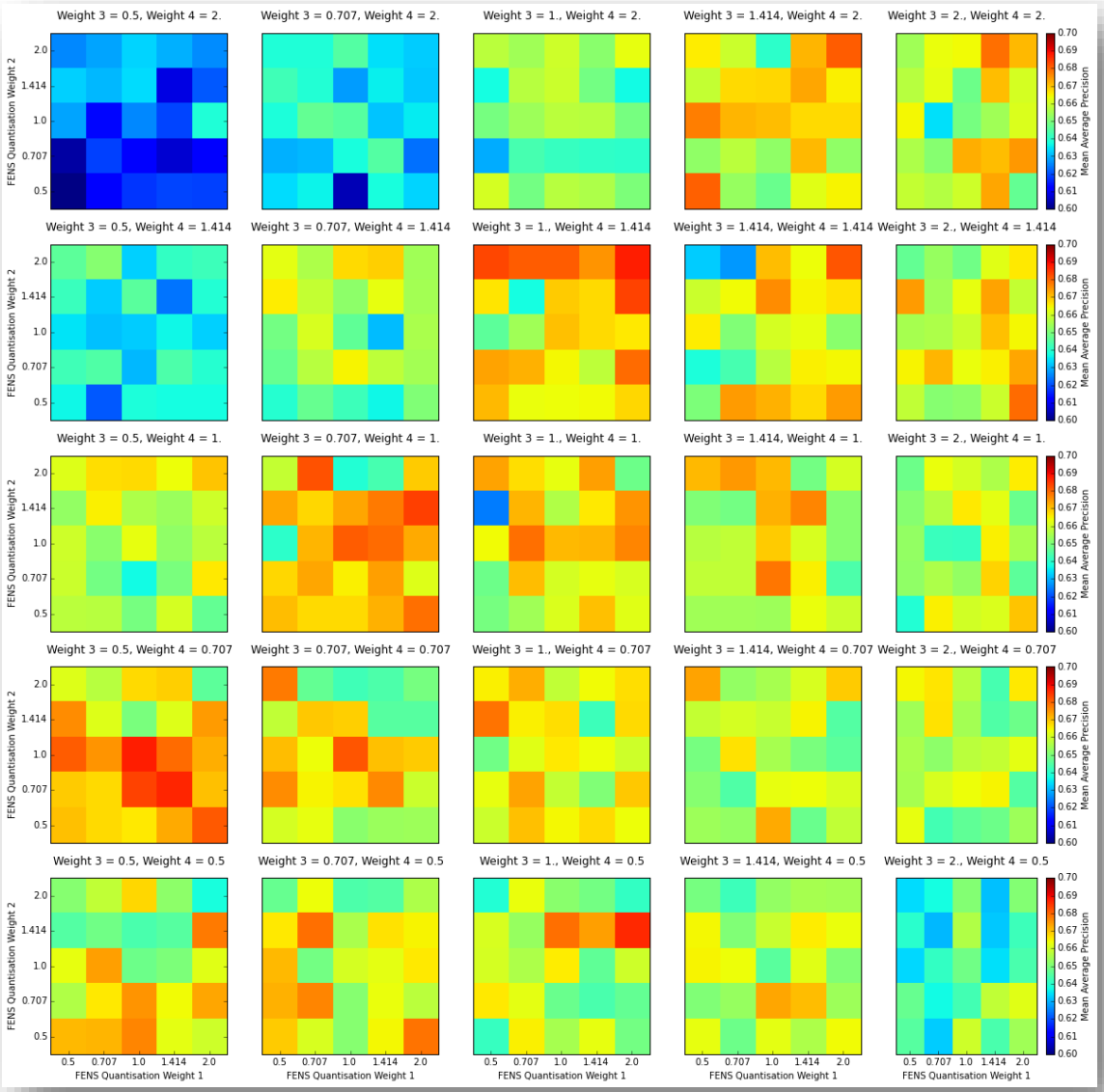


Figure 22 - Illustration of the variation of MAP with different settings for the FENS quantisation weights

4.1.12 Coercion of FENS features into a normal distribution

Initial efforts did successfully produce normal shaped distributions. However, differences in ranges of input feature frequency components illustrated in Figure 6 tended to be preserved by this process (Figure 23), leading to very poor MAP results.

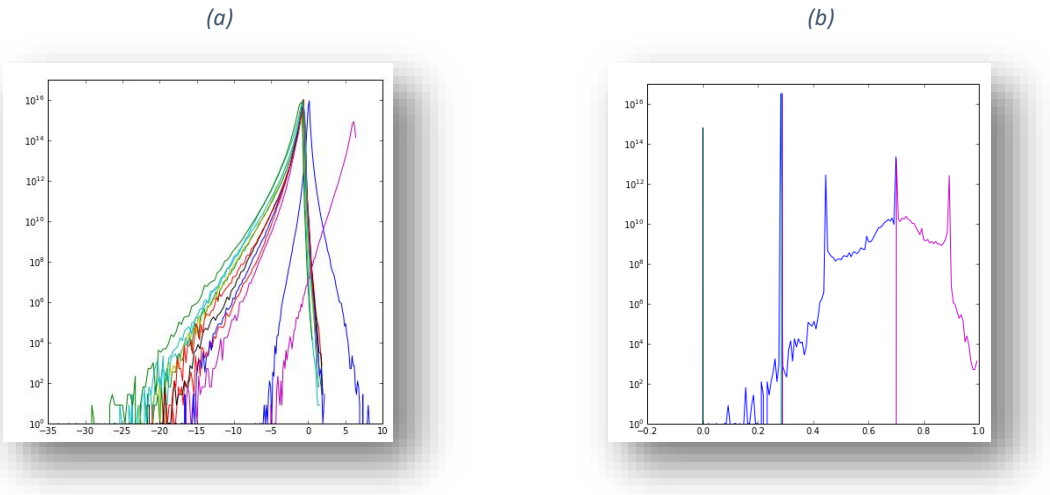


Figure 23 - Histograms of FENS features (a) after transformation of the original log-frequency spectrum features by the weights of the neural network, and (b) after down-sampling

This observation motivated the introduction of input feature component-wise standardisation into the FENS calculation procedure (this is not required for the calculation of CENS from chroma due to the assumed equal importance of each pitch class or chroma bin). However, this standardisation introduces the effect of amplifying the importance of some higher frequency components, which has been seen to not be desirable in all cases (Section 4.1.5) due to fluctuations in adjacent frequency component range amplitudes (Figure 7).

4.2 Training Results

4.2.1 Features without energy-normalised statistics

Table 3 shows the best results achieved using the greedy hill-jumping algorithm on the training set of 5 performances of each of 20 mazurkas for each of the main base feature types.

Base Feature Type	**Highest MAP value**	
	Without Feature Learning	With Feature Learning
Chroma	0.653	0.639
Constant-Q Spectrogram	N/A	0.662
Log-frequency Spectrogram	N/A	0.694

Table 3 - Highest MAP values achieved on the training set for each base feature type

4.2.2 Features using energy-normalised statistics

Table 4 shows the best results achieved after incorporating energy normalised statistics for each base feature type.

Base Feature Type (ENS Feature Type)	**Highest MAP value**
Chroma (CENS)	0.754
Constant-Q Spectrogram (CQSFENS)	0.752
Log-frequency Spectrogram (LFSFENS)	0.721

Table 4 - Highest MAP values achieved on the training set for each base feature type using energy-normalised statistics

4.3 Validation Results

The results in Table 3 and Table 4 represent the highest MAP value obtained across all iterations of each experiment. Each experiment usually contained several iterations, each of which finds its own local optimum value, depending on the randomised starting point of the search parameters. To avoid using a parameter setting on the final test set that had over-fit to the training set, the best three values found in all iterations of training for each feature was tested on the validation set. Each validation run took approximately one hour to complete. Table 5 and Table 6 show the best performing results with and without energy normalised statistics applied.

Base Feature Type	**Highest MAP value**	
	Without Feature Learning	With Feature Learning
Chroma	0.502	0.432
Constant-Q Spectrogram	N/A	0.516
Log-frequency Spectrogram	N/A	0.539

Table 5 - Highest MAP values achieved on the validation set for each base feature type

Base Feature Type (ENS Feature Type)	**Highest MAP value**
Chroma (CENS)	0.627
Constant-Q Spectrogram (CQSFENS)	0.615
Log-frequency Spectrogram (LFSFENS)	0.558

Table 6 - Highest MAP values achieved on the validation set for each base feature type using energy-normalised statistics

4.4 Test Results

Finally, the best configuration of parameters from the validation set testing was tested on the held-out test set. Each test run took approximately four hours to complete. Table 7 shows the results.

Base Feature Type (ENS Feature Type)	**MAP value**
Chroma (CENS)	0.411
Constant-Q Spectrogram (CQSFENS)	0.396

Table 7 - Highest MAP values achieved on the test set for each base feature type using energy-normalised statistics

4.5 Analysis

4.5.1 Results without Energy-Normalised Statistics

The results for chroma features in both the training and validation set indicate that there was no improvement gained from representation learning on these features. The difference between the training set results (1.4%) is much smaller than the difference in the validation set results (7.2%). This difference could be explained by the neural networks being trained only on the training data and not the validation data. This may have led to the networks finding independent components present in the training data which were not as prevalent in the validation data.

Training set results on the constant-Q spectrogram and log-frequency spectrogram features both show an improvement over the chroma features, with or without representation learning. This may indicate that the calculation of chroma vectors from the source audio is omitting some important information which is useful for structural similarity measurements and is captured by the representation learning. This would have a downstream implication on features calculated from chroma i.e. CENS and CRP.

4.5.2 Results with Energy-Normalised Statistics

Results of energy-normalised statistics features using chroma and constant-Q spectrograms as base features showed very similar training results (0.2% difference). This shows that with an appropriate transformation of the learned features prior to the energy-normalised statistics calculation, and with a sufficient proportion of representative data included in the feature learning, performance of the learned features can rival those of the engineered features.

4.5.3 Why did CENS perform better than CQSFENS?

The constant-Q spectrogram features were created using a smaller pitch range than the chroma features (see Table 1). This may be a reason that the CENS features performed consistently better than the CQSFENS features. It should be noted however, that the differences were very small ranging from 0.2% to 1.5% from the training set to the test set.

Another reason could be that the feature transformation applied at the input of the FENS calculation was not sufficient to shape the learned feature representations to fully exploit the energy normalisation process. The configuration of the FENS settings used the highest power transformation chosen for the parameter settings i.e. the tenth power. This could easily be increased with further experimentation.

4.5.4 Comparison of Results with (Bello, 2011)

The best test results achieved in (Bello, 2011) using CRP features and the FAN method was 0.767, which is significantly better than the results from the Project. While the results are not directly comparable, due to the different sized test sets and different features used for testing, it is clear that there are one or more incorrect assumptions in the details of the Project system. Unfortunately it was not possible to get clarification on various details of the system, such as the decimation and anti-aliasing process for the resizing of recurrence plots. A few variations were tried during the training phase but none were more successful than the one used in the final testing.

The one identifiable major difference between the Project system and the system in (Bello, 2011) is that the recurrence plots are resized by resampling the recurrence plot, rather than resampling the input features before the recurrence plot is created. This change was made due to unresolvable calculation errors thrown when using the scipy.signal.resample function. Other implementations were tried for resampling features during training but they either incurred too much overhead to make the system practicable, or produced worse training results. A potential candidate for future testing is the scikits.samplerate function, which is part of a dedicated signal processing library, and should be much faster than the implementations tried.

The sample rate of the original input features may also have had an effect on the results obtained. It is possible that application of the linear averaging used did not accurately substitute for using a different sample rate in the original feature generation. Unfortunately, as identified at the project outset, time and network storage limitations precluded the option of regeneration of the input features from the source audio using identical settings.

5 DISCUSSION

5.1 ORIGINAL OBJECTIVES

5.1.1 Porting of toolboxes from *Matlab* to *Python*

The first objective of porting the relevant code from the *Matlab* toolboxes to *Python* was successfully achieved. This has already shown to be of some benefit to the MIR community as it has led to a revision of the *CRP Toolbox* used to generate recurrence plots. In addition to the two methods used in the Project, three other methods were also implemented, which may prove useful either for structural similarity measurements or in other MIR applications. Moreover, the implementations may provide useful to the wider scientific community – the use of recurrence plots is not limited to music analysis but has various other applications involving time series, such as analysis of climactic patterns, time scale alignment of geological data from boreholes, and magnetostratigraphy (Marwan & Kurths, 2004).

The implementations of some of the other toolbox functionality in the *Chroma Toolbox*, such as the *chromaToCENS* function (Appendix F.1) and the associated downsampling and smoothing functions will also prove useful to members of the community who use *Python* in preference to *Matlab*. Since *Python* and its libraries are free software, it potentially reaches a wider audience than *Matlab*. Python is also particularly popular with the machine learning community, particularly those interested in using *Theano* for deep learning, so implementations of *Matlab* MIR toolkits in *Python* will be of particular interest to researchers who wish to combine the fields of deep learning and MIR.

5.1.2 Structural similarity system

The second objective of building a structural similarity system in *Python* similar to that used in (Bello, 2011) is considered mostly complete, as an end-to-end system has been developed which performs the intended task. Hopefully, some minor clarifications on the details of the *Matlab* system or some further testing with the resampling of features using a different signal processing library will improve the performance of the *Python* system to a level where the potential impact of the representation learning can be fully realised, as it is clear there are still some implementation details that need to be resolved.

Despite the current limitations in overall performance, the system developed for structural similarity measurements is considered both highly scalable and generalizable to new tasks involving structural similarity.

The *multiprocessing* module has been used to parallelise the tasks of calculation of recurrence plots and recurrence plot compression so that the run-time of each iteration is effectively divided by the number of available calculation cores. Also the system has been developed to minimise the amount of disk I/O required. Therefore it could potentially be used on much larger collections of music than the dataset used for the Project, given sufficient computational power and main memory.

Furthermore the use of recurrence plots and the normalised compression distance are not limited to the analysis of music. In principle, the calculation of latent components using representation learning for subsequent quantification of similarity can be applied to any vectorisable dataset with a complex repetitive temporal component, from environmental noise measurements, to videos of daily traffic movements on a motorway, to home energy usage.

5.1.3 Representation learning

The third objective of using representation learning to generate features which can enhance the performance of existing audio features is also considered to have been partially achieved. By training neural networks on lower level audio features, it has been shown that a compressed representation of these features can outperform chroma features on a small training set. Once issues with the similarity system performance are resolved there is the potential that this result can be extended to larger datasets.

5.2 New Findings

The main new finding is the discovery that by applying the energy-normalised statistics calculations with appropriate feature transformations, the performance of the learned features used in the Project rivals the performance of the state-of-the-art CENS features in the context of the system developed. Given the limited scope of the representation learning performed, this is considered to be a potentially significant result in the wider context of combining the fields of MIR and machine learning, but will need to be confirmed by improvements to the overall system and further study of the features learned from representation learning, particularly their distributions and the distribution of their energy.

Another new finding is confirmation that there is a general relationship between the reconstruction error of a denoising autoencoder, and the performance of a structural similarity system which uses these latent representations as features. This was assumed to be true from the start of the Project, but the result illustrated in Figure 15(b) confirms that there is a clear relationship. A quick linear regression model fit to the distribution indicated that a second order polynomial gave the best fit to the data. This implies that for each system configuration there may be an optimum reconstruction error in terms of the overall MAP performance. This could provide useful insight into preventing the representation learning from overfitting to the data it trains on. This may provide also a good foundation for future work into training other types of neural network for the task of structural similarity measurements, particularly RBMs, where it can be difficult to know when to stop training, even when the ground truth is known (Hinton, 2010).

5.3 Answer to the Research Question

The question posed for the research project was:

Can representation learning improve the performance of structural music similarity measurements in comparison with engineered features?

It is considered that this has been answered in the affirmative because significant improvements have been made over some features such as chroma. However, the limitations of the system implemented and some configuration details of the original feature files still leave the question open as to whether learned features can outperform the state-of-the-art CENS and CRP features; this will be the subject of future work using more representation learning and further investigation into distribution-shaping algorithms such as the one used for the new FENS features developed in the Project.

6 EVALUATION, REFLECTION AND CONCLUSIONS

6.1 ADEQUACY OF PROJECT PLAN

The risk register identified a medium level risk of the original code not being obtained. The system used in (Bello, 2011) was first introduced in 2009 (Bello, 2009) and then updated in 2011. This timescale indicates that there was likely a significant amount of development work. The Project Plan allowed 3 weeks for reimplementation of the structural similarity system. On reflection, this was overly optimistic, and a number of compromises needed to be made on the amount of experimentation that could be performed using representation learning in the remaining time.

It is difficult to say what would have been done differently if the actual timescales involved had been known at the project inception phase. If less time had been allocated to translation of the *CRP Toolbox*, then the potential error might not have been found, which could have been to the overall detriment of the MIR community and potentially to the wider scientific community. It would not have been possible to allocate less time to reproducing the structural similarity system as it was a necessary component to obtain any results. In the end, less representation learning was done than was desired but this sacrifice allowed focus to be placed on the application of energy-normalised statistics to the learned features, which was a new discovery.

If there was one thing that would have been done differently, then it would probably have been to put more effort into finding an implementation of resampling that would have allowed the input features to be resampled to fixed length prior to calculating the recurrence plots, as this is the only significant difference in system implementation which is currently known between the *Matlab* and *Python* versions.

6.2 CHOICE OF OBJECTIVES

The choice of objectives was ambitious, and while progress was made in all of them, there were limitations imposed by factors which were not entirely knowable at the time of proposing the Project. On reflection however, it is considered that sufficient progress was made in each objective to make a valuable contribution to the MIR field. It is also considered that the Project provides a good basis for further work, in terms of its findings and the code base produced, for performance improvements in the task of structural similarity measurement using representation learning.

6.3 RECOMMENDATIONS FOR FURTHER WORK

6.3.1 Structural similarity system

The main recommendation for further work on the structural similarity system is to resolve the differences in the *Matlab* and *Python* implementations which lead to different final results. If the original *Matlab* code can be obtained, then a direct A/B comparison can be performed and the reasons for the differences should be fairly straightforward to resolve.

The second recommendation would be to confirm the details of the third, normalised Euclidean distance method (NEUC), which was not in the CRP toolbox and therefore not used in the Project, and test this method against the two methods used in the Project.

The third recommendation would be to parallelise the final computation of the MAP score, if possible. Currently it is calculated on a single core using the *pandas* library and completes in a few seconds on the training set. However, for the validation and training set, the calculation time

increases approximately with the square of the number of feature files, which is approximately a 250-fold increase from the training set to the test set. This calculation is an obvious candidate for parallelisation.

6.3.2 Representation learning

A limitation of the representation learning was that it used only one type of neural network, the denoising autoencoder. While this is considered an important proof of concept of the application of representation learning to structural similarity measurements, there are many more types of network which could have been used instead and could be used in the future. These include RBMs, recurrent neural networks and convolutional neural networks. One example that could be tried is to use of a convolutional neural network to tie the weights of harmonically related frequency components of a constant-Q spectrogram together to better direct the learning process.

Another interesting area to investigate would be networks with many hidden layers trained in a pairwise manner as autoencoders or RBMs; this is commonly known as *deep learning*. While the universal approximation theorem says that *a single hidden layer is sufficient for a multilayer perceptron to compute a uniform approximation to a given training set* (Haykin, 2009), a key limitation is that there need to be sufficient units in this hidden layer to adequately represent the function. This conflicts with the preference to use a feature vector with a low number of components in order to minimise the size of the embedded vectors required in the calculation of the recurrence plots. A deep network, such as a stacked denoising autoencoder, might be able to calculate a higher level representation of the input vector which is more useful for the structural similarity task. When properly trained, deep networks have been shown to be useful for several applications related to MIR (Dieleman, 2014), (Lee, et al., 2009), (van den Oord, et al., 2013) and there is no obvious reason to suspect that they could not help with structural similarity, given what has been found in this Project using single hidden layer networks.

6.3.3 Interface of representation learning with the structural similarity system

It was assumed for the purposes of the Project that a network with a lower reconstruction error would lead to a better MAP score in the structural similarity system. While this was found to be true for the training set, it is clear from the results that the performance of the learned features on the validation and test set decreases (although this was also true for the engineered features). While this effect could be due to not using enough training data, it may also be due to overfitting to the training set, i.e. continuing to learn past the point where the learned representations would generalise well to new data.

A better, but more complex, approach could involve using the early stopping method to continue training the neural network on the training data up until the point where the performance on a known set of “good” settings for the structural similarity system stopped improving on a validation set. This would be a slower approach, and the size of the validation set would have to be chosen carefully, but it could potentially enhance the overall system performance in a more robust predictable manner.

6.3.4 Learning from more data

The issues encountered with limitations of the sizes of the *Theano* training files could be overcome with some additional work, by reading the data directly from the features and storing them in memory for the training, rather than writing them to separate files. This could help the networks to learn from more data, which would be an important enhancement for learning features which generalise better to new data.

7 GLOSSARY

The following terms are defined for unfamiliar readers. Where definitions have been used from other literature, the reference is cited at the end of the definition.

Word / Phrase	Definition
down-sampling	The process of reducing the sample rate of a signal. Interpolation may be applied if the down-sampling rate is not an integer. Smoothing may and filtering may also be applied.
forward-backward filtering	Forward-backward filtering is used to implement zero-phase filtering of an offline signal. This squares the amplitude of the signal and zeros the phase response. (Smith, 2007)
epoch	In the context of neural networks, an epoch elapses after a complete presentation of all the training examples to the network.
reconstruction error	The error in reconstructing a feature vector from a compressed representation of the vector.
Hann window	The Hann window, also called the raised cosine window, is given by: $$w[i] = 0.5 - 0.5\cos\left(\frac{2\pi i}{m}\right)$$ It has an approximately 18dB per octave roll-off on the side lobes, and a stop-band attenuation of -44 dB. (Smith, 1999)

8 REFERENCES

Bartsch, M. A. & Wakefield, G. H., 2001. *To catch a chorus: using chroma-based representations for audio thumbnailing.* New York, s.n.

Bartsch, M. A. & Wakefield, G. H., 2005. Audio Thumbnailing of Popular Music Using Chroma-Based Representations. *IEEE Transactions on Multimedia,* 7(1), pp. 96 - 104.

Bello, J. P., 2009. *Grouping Recorded Music by Structural Similarity.* Kobe, s.n.

Bello, J. P., 2011. Measuring Structural Similarity in Music. *IEEE Transactions on Audio, Speech, and Language Processing,* 19(7), pp. 2013 - 2025.

Bergstra, J. et al., 2010. *Theano: A CPU and GPU Math Expression Compiler.* Austin, Proceedings of the Python for Scientific Computing Conference (SciPy).

Bogert, B. P., Healy, M. J. R. & Tukey, J. W., 1963. *The Quefrency Alanysis of Time Series for Echoes: Cepstrum, Pseudo-autocovariance, Cross-Cepstrum, and Saphe Cracking.* New York, s.n.

Bustinza, O. F., Vendrell-Herrero, F., Parry, G. & Myrthianos, V., 2013. Music business models and piracy. *Industrial Management & Data Systems,* 113(1), pp. 4 - 22.

Butler, S., 2007. Piracy Losses. *Billboard*, 8 September, p. 18.

bzip2, 2010. *bzip2.org.* [Online]
Available at: http://www.bzip.org/

Cannam, C. et al., 2010. Linked Data And You: Bringing music research software into the Semantic Web. *Journal of New Music Research,* 39(4), pp. 313 - 325.

CHARM, 2009. *CHARM Mazurka Project.* [Online]
Available at: http://www.mazurka.org.uk/

Cillibrasi, R., Cruz, A. L., de Rooij, S. & Keijzer, M., 2008. *complearn.org.* [Online]
Available at: http://complearn.org/
[Accessed 2015].

Dannenberg, R. B. & Hu, N., 2003. Pattern Discovery Techniques for Music Audio. *Journal of New Music Research,* 32(2), pp. 153 - 163.

Dieleman, S., 2014. *http://benanne.github.io/2014/08/05/spotify-cnns.html.* [Online]
Available at: http://benanne.github.io/2014/08/05/spotify-cnns.html
[Accessed 21 April 2015].

Ellis, D. P. W. & Poliner, G. E., 2007. *Identifying 'Cover Songs' with Chroma Features and Dynamic Programming Beat Tracking.* Honolulu, s.n.

Ewert, S. E., Muller, M. & Grosche, P., 2009. *High Resolution Audio Synchronization using Chroma Onset Features.* Taipei, s.n.

Gutierrez, E. G., 2006. *Ph.D. Dissertation: Tonal Description of Music Audio Signals,* Barcelona: Universtat Pompeu Fabra.

Hamel, P. & Eck, D., 2010. *Learning features from music audio with deep belief networks.* Utrecht, s.n.

Haykin, S., 2009. Chapter 4.12 - Universal Approximation Theorem. In: *Neural Networks and Learning Machines - A Comprehensive Foundation, Third Edition.* Upper Saddle River: Pearson, pp. 167 - 168.

Herrada, O. C., 2008. *Ph.D. Thesis: Music Discovery and Recommendation in the Long Tail.* Barcelona: Herrada, Oscar Celma;.

Hinton, G., 2010. *A Practical Guide to Training Restricted Boltzmann Machines, Version 1.* [Online]
Available at: https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf
[Accessed 15 6 2015].

Inkpen, D., 2014. *Information Retrieval on the Internet.* [Online]
Available at: http://www.site.uottawa.ca/~diana/csi4107/IR_draft.pdf
[Accessed 15 06 2015].

Kinsler, L. E., Frey, A. R., Coppens, A. B. & Sanders, J. V., 1998. Chapter 11: Noise, Signal Detection, Hearing, and Speech. In: *Fundamentals of Acoustics, Third Edition.* New York: John Wiley and Sons, p. 273.

Koren, Y., 2015. *http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.* [Online]
Available at: http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf
[Accessed 1 April 2015].

Lee, H., Pham, P., Largman, Y. & Ng, A. N., 2009. *Unsupervised feature learning for audio classification using convolutional deep belief networks.* Vancouver, s.n.

Marwan, N., 2015. *Cross-Recurrence Plot Toolbox.* [Online]
Available at: http://tocsy.pik-potsdam.de/CRPtoolbox/index.html

Marwan, N. & Kurths, J., 2004. Cross Recurrence Plots and their Applications. In: C. V. Benton, ed. *Mathematical Physics Research at the Cutting Edge.* s.l.:Nova Science Publishers, pp. 101 - 139.

Moffat, A., 1990. Implementing the PPM data compression scheme. *IEEE Transactions on Communications,* 38(11), pp. 1917 - 1921.

Muller, M., 2007. Pitch- and Chroma-Based Audio Features. In: *Information Retrieval for Music and Motion.* Berlin: Springer Berlin Heidelberg, pp. 51 - 67.

Muller, M., 2015. Music Structure Analysis. In: *Fundamentals of Music Processing - Audio, Analysis, Algorithms, Applications.* New York: Springer, pp. 154-155.

Muller, M. & Ewert, S., 2010. Towards Timbre-Invariant Audio Features for Harmony-Based Music. *IEEE Transactions on Audio, Speech, and Language Processing,* 18(3), pp. 649 - 662.

Muller, M. & Ewert, S., 2011. *Chroma Toolbox: MATLAB Implementations for Extracting Variants of Chroma-Based Audio Features.* Miami, Proceedings of the International Conference on Music Information Retrieval (ISMIR).

Muller, M., Kurth, F. & Clausen, M., 2005. *Chroma-based Statistical Audio Features for Audio Matching.* New Paltz, NY, s.n.

National Institute of Standards and Technology, 1998. *MNIST database,* Maryland, VA: National Institute of Standards and Technology.

Popham, J., 2011. Factors influencing music piracy. *Criminal Justice Studies,* 24(2), pp. 199 - 209.

Raczynski, S. A., Ono, N. & Sagayama, S., 2007. *Multipitch analysis with harmonic nonnegative matrix approximation.* Vienna, s.n.

Schorkhuber, C. & Klapuri, A., 2010. *Constant-Q transform toolbox for music processing.* Barcelona, s.n.

Shepard, R. N., 1964. Circularity in Judgments of Relative Pitch. *Journal of the Acoustical Society of America,* 36(12), pp. 2346 - 2353.

Smith, J. O., 2007. *Introduction to Digital Filters with Audio Applications.* [Online]
Available at: https://ccrma.stanford.edu/~jos/fp/Forward_Backward_Filtering.html

Smith, S. W., 1999. Windowed-Sinc Filters. In: *The Scientist and Engineer's Guide to Digital Signal Processing, Seond Editiom.* San Diego: California Technical Publishing, p. 288.

Sutton, R. S. & Barto, A. G., 2012. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

van den Oord, A., Dieleman, S. & Schrauwen, B., 2013. *Deep content-based music recommendation.* Lake Tahoe, s.n.

Vincent, P., Larochelle, H., Bengio, Y. & Manzagol, P.-A., 2008. *Extracting and composing robust features with denoising autoencoders.* New York, s.n.

Appendix A DESCRIPTIONS OF COMMON AUDIO FEATURE TYPES

A.1 SPECTROGRAM

One of the most basic audio features is the spectrogram. This feature is obtained by performing a Fourier Transform $\mathcal{F}$ on a short window x_t of the full audio time history x_T so that it is represented in the frequency domain (a Short-Time Fourier Transform or STFT), and squaring the magnitude of the result. The frequency resolution of the spectrogram is determined by the duration of the time window used as input to the Fourier Transform. Longer time windows give higher resolutions in the frequency domain, but the time resolution is lower. This is the well-known time vs. frequency trade-off. Figure 24 illustrates the calculation of a spectrogram $s(f)_t$ from a window of a signal x_T. In each case, the $\diamond$ symbol represents the results of the previous stage in the calculation.

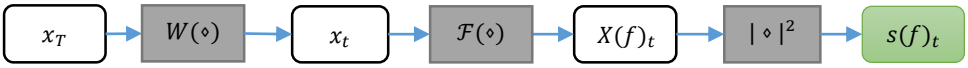


Figure 24 - Calculation of a spectrogram from an audio signal

A.2 LOG -FREQUENCY SPECTROGRAM

The log-frequency spectrogram is also calculated from the STFT but uses a logarithmic frequency axis to mimic both the logarithmic perception of sound, and the logarithmic distribution of equally separated pitches in the equal temperament scale, ubiquitous in western music. Each spectral coefficient is assigned to the pitch with a centre frequency that is closest to the centre frequency of the coefficient.

A.3 CONSTANT-Q SPECTROGRAM

The Constant-Q spectrogram transforms the time domain signal into a frequency domain representation where the frequency components are logarithmically spaced, and the Q factors of the frequency components are all equal. This gives a better frequency resolution at lower frequencies and a better time resolution at higher frequencies (Schorkhuber & Klapuri, 2010). This is most efficiently achieved by using a bank of filters with each octave being sampled at a different rate (Muller, 2015).

A.4 MEL-FREQUENCY CEPSTRAL COEFFICIENTS (MFCCS)

A cepstrum (Bogert, et al., 1963) can be thought of as the spectrum of a spectrum. After calculation of the STFT, the spectrum is mapped onto the mel scale (Kinsler, et al., 1998), which is logarithmic to the frequencies of the spectral coefficients, and is subjectively equally spaced, according to curves produced by Stevens and Volkmann. Then after taking logarithms of the mel values, the Discrete Cosine Transform $\mathbb{C}$ of the log values is taken, returning the MFCCs. Figure 25 illustrates the calculation procedure for MFCCs from an STFT of a windowed signal.

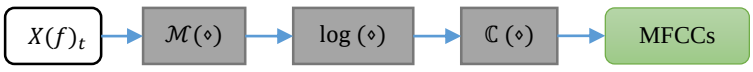


Figure 25 - Calculation of MFCCs from a Short-Time Fourier Transform of a windowed signal

MFCCs are widely used in speech processing and telecommunications. They have also been used in music applications in the identification of musical instruments, as the lower MFCCs are useful for detecting the timbre of musical instruments (Muller & Ewert, 2010).

A.5 CHROMA

Chroma features, also known as Pitch Class Profiles, are a family of features which are designed primarily for western music. They derive from Shepard's work in (Shepard, 1964), where he quoted earlier work by Drobisch in noting that a helical representation of the continuum of pitch has "the advantage of brining tones an octave apart into closer spatial proximity", and separated the concepts of tonal height (overall pitch) from tonal chroma (pitch class). Each chroma can be represented as a 12-dimensional vector, with each member representing the energy in all octaves of each of the semitones of the equal-tempered scale {C, C#, D … A, A#, B}. Each chroma vector represents an equal duration of time which is much shorter than the durations of individual musical notes (Dannenberg & Hu, 2003).

Each member of the chroma family of features is calculated in a slightly different way, but they all share some basic characteristics. Figure 26 shows the basic calculation method, where the STFT is passed through a logarithmic-frequency filter bank $\mathcal{B}$, and then folded into bins of the chroma vector by summing the log-frequency magnitude spectrum over Z octaves for each pitch class b, where there are β classes per octave.

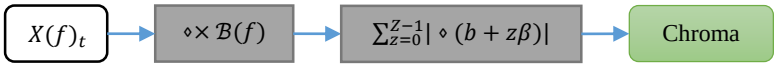


Figure 26 - Calculation of a chroma vector from an STFT

More sophisticated members of the Chroma family include the Harmonic Pitch Class Profile (HPCP) (Gutierrez, 2006), Chroma Energy Normalised Statistics (CENS) (Muller, 2007), Chroma DCT-Reduced log Pitch (CRP) (Muller & Ewert, 2010) and Beat Synchronous Chroma features (Bartsch & Wakefield, 2001). Each of these is designed to overcome various shortcomings of the basic feature for certain applications, such as polyphonic music transcription or structural similarity, for example reducing the effects of transients or the use of different instrumentation (Bello, 2011).

A.6 CHROMA ENERGY NORMALISED STATISTICS

Chroma Energy-Normalised Statistics (CENS), developed in 2005, were introduced in (Muller, et al., 2005) as a feature which *shows a high degree of robustness to variations in parameters such as dynamics, timbre, articulation, and local tempo deviations*.

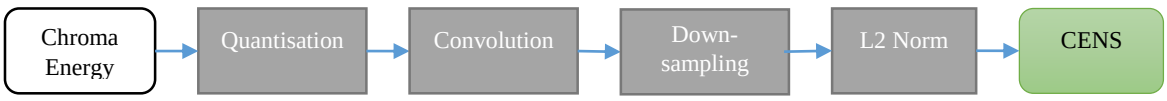


Figure 27 - Calculation of a CENS vector from a chroma vector (Muller, et al., 2005)

CENS features are created in two stages; the first is similar to the creation of basic chroma features, whereby the audio signal is decomposed into 88 frequency bands representing the 88 musical pitches of a piano, converted into a short-time mean square power and then binned into the twelve chroma bins.

The second stage involves a quantisation of the components of each frame of the vector depending on the proportion of the total frame's energy contained by each component. The four quantisation steps given are as follows:

$$Q(a) = \begin{cases} 4 \text{ for } & 0.4 \leq a \leq 1 \\ 3 \text{ for } & 0.2 \leq a \leq 0.4 \\ 2 \text{ for } & 0.1 \leq a \leq 0.2 \\ 1 \text{ for } & 0.05 \leq a \leq 0.1 \\ 0 \text{ for } & 0 \leq a \leq 0.05 \end{cases}$$

where: a is the proportion of the frame's energy contained by each chroma component, and
$Q(a)$ is the value assigned to the chroma component.

After quantisation of each component and frame, the feature vector is convolved with a Hann window of length 41, and then down-sampled to a factor of 10 and normalised to the Euclidean norm. For a chroma feature rate of 10 frames per second, this results in each CENS frame representing 4.1s of audio at a rate of one frame per second (the time windows covered by adjacent features are overlapping).

A.7 CHROMA DCT-REDUCED LOG PITCH

Chroma DCT-reduced log pitch (CRP) features were introduced in 2010 (Muller & Ewert, 2010) as a timbre-invariant feature which, to paraphrase the original text, improve the timbre invariance of chroma features without affecting their discriminative ability. This is accomplished by borrowing ideas from MFCC features and noting that the lower cepstral coefficients are closely related to the timbre of musical instruments. The new features are created using a pitch scale in place of the mel scale, and removing the lower timbre-related cepstral coefficients. The steps for the creation of the features are illustrated in Figure 28.

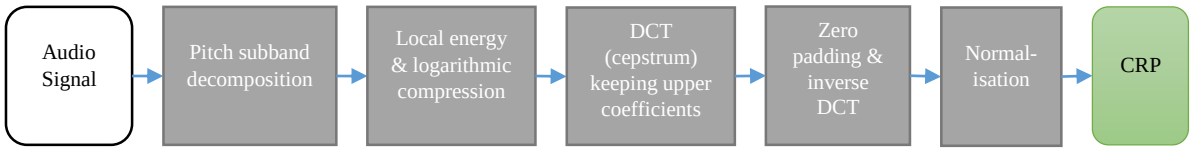


Figure 28 - Steps in creation of CRP features (Muller & Ewert, 2010)

Appendix B ANALYSIS OF THE GREEDY, HILL-JUMPING ALGORITHM

Since the algorithm searches across a single dimension of settings once, it does not assume that the objective function space is smooth and therefore should be able to find and choose the best value within the dimensions of the parameter. Also, the number of settings which needs to be tested increases linearly with the number of additional settings of each parameter added, so the increase between the system in (Bello, 2011) increases from 40 to 86.

However, as with any search strategy, there are limitations. The choice of a greedy methodology, as with other analogous algorithms such as CART, does not guarantee that the choice made which gives the highest immediate reward is optimal. A more sophisticated algorithm could borrow from an approach such as Q-Learning (Sutton & Barto, 2012), to look ahead one step for the future reward before choosing the best direction (i.e. choosing the optimal policy). This would increase the run-time of each step but may be more likely to proceed towards the optimal solution than the existing algorithm.

Another disadvantage to the algorithm is that it has an inherent order dependency, since it tests each feature one at a time. This order dependency means that the algorithm may choose to take one path along a dimension of one parameter, which does not give as good a result as if it had taken the best value of another parameter instead. One way of addressing this issue was to use multiple random restarts; by starting the algorithm at a different location each time, then the order of parameters may be more or less important depending on this starting point. However, a randomised order of parameters could also be used without any increase in run-time to alleviate the order dependency.

Finally, the algorithm makes an assumption of a certain amount of independence between parameters; however it has been shown in section 4.1 that there is a significant amount of interaction between some parameters. A more careful algorithm could choose to vary together those parameters which are known a priori to have a high level of interaction (such as FENS quantisation weights). The drawback of such an approach would also be increased training times but this would help to resolve some of the order dependency.

Appendix C TABLES

C.1 TOOLKITS USED IN THE ORIGINAL STUDY (BELLO, 2011)

Technique	Software used in (Bello, 2011)	Platform
Chroma and CENS	Chroma Toolbox	*Matlab*
Beat Tracking	http://labrosa.ee.columbia.edu/projects/coversongs/	*Matlab*
Recurrence Plots	CRP Toolbox (requires registration)	*Matlab*
NCD	CompLearn Toolkit	Windows / Linux

C.2 RISK REGISTER

Description	Likelihood (1 – 3)	Consequence (1 – 5)	Impact (L x C)	Mitigation
Original source audio cannot be obtained	3	4	12	Use British Library audio
Original code cannot be obtained	2	4	8	Make early enquiries to obtain code Review toolkits
Extracting spectrograms takes a long time	1	3	3	Start this early Ask Daniel Wolff to help
Other students are using the computing cluster when I want to run experiments	3	4	12	Meet with supervisor at kick-off meeting to review other students' project timelines
Neural network models are hard to write	3	5	15	Use Theano example scripts and modify Test small models on home computer
Neural network models take too long to run	2	5	10	Reduce number of models Consider renting cluster space
City University cluster crashes for a long time	1	5	5	Consider renting cluster space
Existing results cannot be replicated exactly	3	3	9	Resort to relative comparison
Code is lost or accidentally written over	2	5	10	Use version control software e.g. Git

C.3 FACTORS INFLUENCING THE CHOICE OF PROGRAMMING LANGUAGE

Development Item	Preferred Language	Discussion / Provisos
Reuse of toolkits used in existing system	*Matlab*	Only an advantage if existing system code can be obtained. Potentially faster.
Reimplementation of existing code from Juan Pablo Bello	*Matlab*	No response received therefore no advantage.
Development of Neural Networks	*Python*	Theano deep learning library could easily be run on the university server and GPU. Example scripts available in online tutorial.
General Code Development	*Python*	More experience with *Python* and more online support available.
Sharing of Project Work	*Python*	*Matlab* is expensive to license and therefore of limited use amongst the wider MIR community. An open source *Python* implementation considered more widely usable as it ports some existing *Matlab* tools into *Python* and is not licensed.

C.4 NUMBERS OF COMBINATIONS OF RUN SETTINGS

Stage	Setting	Typical Range of Settings	Number of Settings	New System Permutations
Feature Learning	Number of Visible Units	128, 256	2	112
	Number of Hidden Units	3, 6, 9, 12, 15, 18, 21, 24	8	
	Corruption Level	10%, 20%, 30%, 40%, 50% 60%, 70%	7	
Feature Pre-processing	Feature Down-sampling Rate (--> feature sample rate)	1, 2, 4, 8, 12	5	5
Feature Energy Normalised Statistics Calculation	Input Feature Transformation	log, exponent, square, square root, cube, cube root	6	5
	FENS Normalisation Threshold	0.0001, 0.0003,0.001, 0.003, 0.01	5	546875
	FENS Quantisation Steps	$[a, a{\cdot}p, a{\cdot}p^2, a{\cdot}p^3]$ | a in [0.01, 0.03, 0.05, 0.07, 0.09]	5	
	FENS Quantisation Weights	[0.8 -> 1.2 step 0.1] * 4	5^4	
	FENS Down-sampling Window Length	10, 20, 30, 41, 50, 60, 70	7	
	FENS Down-sampling Rate	1, 2, 4, 8, 12	5	
CRP Calculation	CRP Method	fan, rr, norm	3	399
	CRP Neighbourhood Size	0.05 -> 0.95 step 0.05	19	
	CRP Time Delay	1, 2, 3, 4, 5, 6, 7	7	
NCD Calculation	NCD Sequence Length	300, 500, 700, 900, 1100, var	6	6
MAP Calculation	No Parameters	N/A	1	1

C.5 NEURAL NETWORK RUNS

Base Feature	Visible Units	Hidden Units	Frequency Standard-isation	Batch Size	Learning Rate	Learning Rate Boost	Time Stacking	Corruption Levels
CENS[1]	12	3, 6, 9, 12	No	10, 20, 30, 40	0.01, 0.03, 0.1	n/a	1	0 - 90, step 10
CENS[1]	12	3, 6, 9, 12, 15, 18, 21, 24	No	10	0.05	1.5	1	0 - 70, step 10
CENS[1]	12	3, 6, 9, 12	No	20	0.05	1.5	1	0 - 70, step 10
CENS[1]	12	3, 6, 9, 12, 15, 18, 21, 24	No	40	0.05	1.5	1	0 - 70, step 10
chroma	12	3, 6, 9, 12	No	10, 20, 30, 40	0.01, 0.03, 0.1	n/a	1	0 - 90, step 10
chroma	12	3, 6, 9, 12	No	40	0.05	1.5	1	0 - 70, step 10
constant-Q spectrogram	48	12, 24, 36, 48	No	40	0.05	1.5	1	0 - 70, step 10
constant-Q spectrogram	480	12, 24, 36, 48, 60, 72, 84, 96	No	40	0.05	1.5	10	0 - 70, step 10
constant-Q spectrogram	48	12, 24, 36, 48	No	40	0.1	n/a	1	0 - 90, step 10
constant-Q spectrogram	480	12, 24, 36, 48	No	40	0.1	n/a	10	0 - 90, step 10
constant-Q spectrogram	48	12, 24, 36, 48	No	10, 20, 30, 40	0.01, 0.03, 0.1	n/a	1	0 - 90, step 10
log-frequency spectrogram	256	3, 6, 9, 12, 15, 18, 21, 24	No	10	0.05	1.5	1	0 - 70, step 10
log-frequency spectrogram	256	12, 24, 36, 48, 60, 72, 84, 96	No	40	0.05	1.5	1	0 - 70, step 10
log-frequency spectrogram	128	3, 6, 9, 12, 15, 18, 21, 24	No	40	0.05	1.5	1	0 - 70, step 10
log-frequency spectrogram	256	3, 6, 9, 12, 15, 18, 21, 24	Yes	40	0.05	1.5	1	0 - 70, step 10

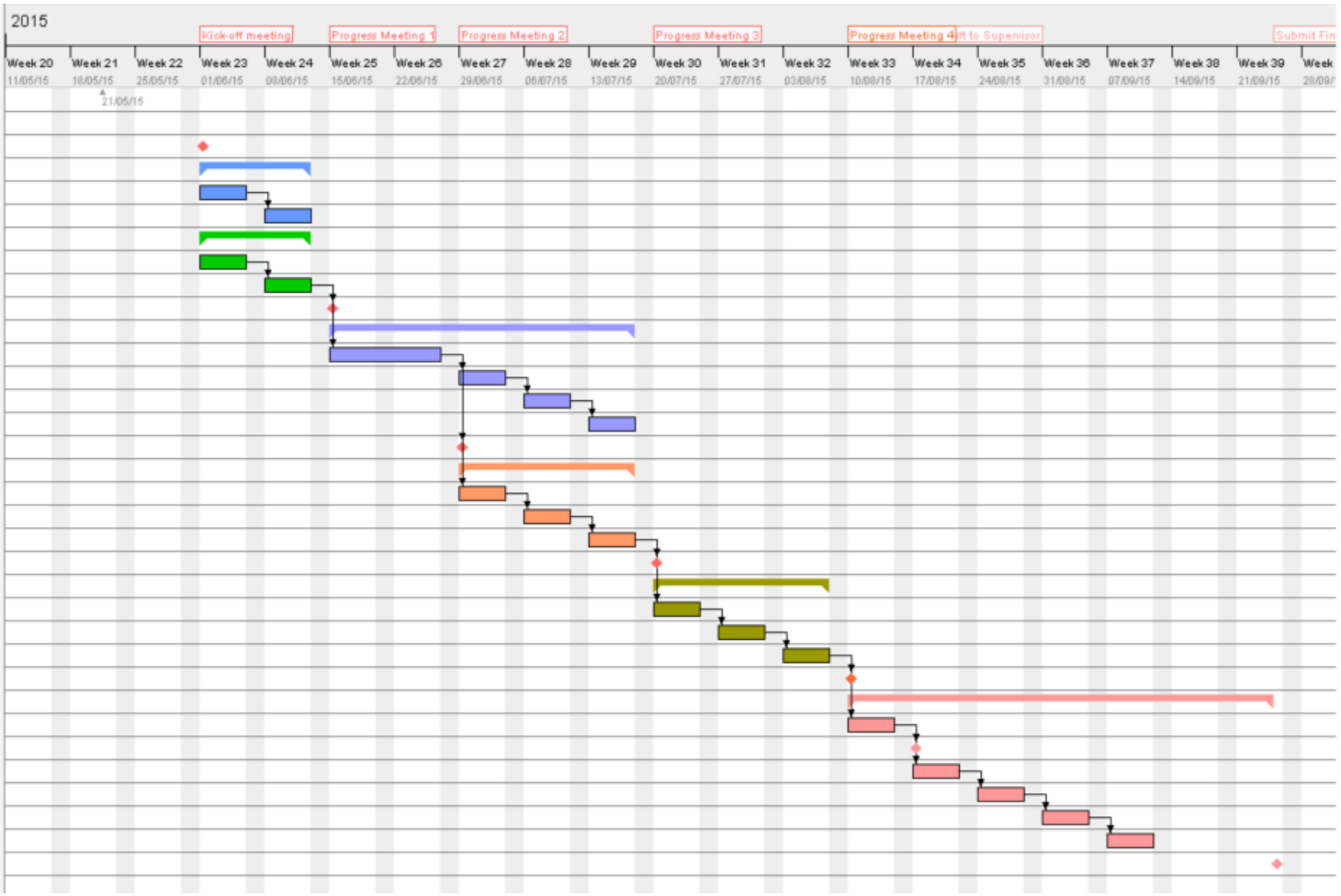
C.6 Record of Training Runs

Run Name	Feature Name	Mean Average Precision	DownSample Factor	CRP Dimension	CRP Method	CRP Neighbourhood Size	CRP Time Delay	NCD Sequence Length	NN Time Stacking	NN Frequency Standardisation	NN Num Folders	NN Num Files per Folder	dA Batch Size	dA Corruption Level	dA Learning Rate	dA # Hidden Units	dA # Visible Units	dA Learning Rate Boost Factor	Folders	Files per Folder	FENS Downsampling	FENS Normalisation Threshold	FENS Quantisation Step Base	FENS Quantisation Step Power	FENS Quantisation Weight 1	FENS Quantisation Weight 2	FENS Quantisation Weight 3	FENS Quantisation Weight 4	FENS Transformation Function	FENS Window Length	Notes
1-2	chroma	0.336	16	1	euclidean	0.85	4												5	5											method comparison
1-2	chroma	0.397	16	7	fan	0.05	4												5	5											method comparison
1-2	chroma	0.336	16	1	maxnorm	0.85	3												5	5											method comparison
1-2	chroma	0.336	16	1	minnorm	0.85	1												5	5											method comparison
1-2	chroma	0.332	8	1	nrmnorm	0.85	5												5	5											method comparison
1-2	chroma	0.362	16	5	rr	0.05	4												5	5											method comparison
3	chroma			7	fan	0.05	1												5	5											method comparison
9	CENS_dsf2_wl41	~~0.769~~	2	3	fan	0.1	5	1100											20	5											method comparison
9	CENS_dsf2_wl41	~~0.757~~	1	2	rr	0.1	3	1100											20	5											method comparison
10	CENS_dsf2_wl41	~~0.575~~	1	2	nrmnorm	0.8	3	1100											20	5											method comparison
11	CENS_dsf2_wl41	~~0.489~~	2	2	euclidean	0.95	4	1100											20	5											method comparison
12	chroma	0.650	4	3	fan	0.05	3	900											20	5											method comparison
12	chroma	0.525	4	3	rr	0.05	5	1100											20	5											method comparison
run14	CENS_dsf2_wl41	~~0.831~~	2	5	fan	0.15	7	900											20	5											method comparison
run15	CENS_wl41	~~0.811~~	1	2	rr	0.05	2	700											20	5											method comparison
run16	chroma	0.576	16	3	rr	0.05	2	300											20	5											method comparison
run17	chroma	0.653	4	2	fan	0.05	4	500											20	5											method comparison
run18	CENS_dsf2_wl90	~~0.741~~	4	4	fan	0.15	7	1100											20	5											method comparison
run19	CENS_dsf2_wl90	~~0.728~~	12	2	rr	0.05	3	300											20	5											method comparison
run20	chroma	0.557	12	2	rr	0.05	4	500					20	0	0.1	3	12		20	5											method comparison
run21	chroma	0.549	16	1	fan	0.05	4	300					10	0	0.01	12	12		20	5											method comparison
run22	CENS_dsf2_wl41	~~0.832~~	2	4	fan	0.15	6	900					10	80	0.1	12	12		20	5											method comparison
run23	CENS_dsf2_wl41	~~0.801~~	1	2	rr	0.05	1	700					10	70	0.1	12	12		20	5											method comparison
run24	constantQspec	0.658	4	2	fan	0.05	3	700					10	0	0.1	12	48		20	5											method comparison
run25	constantQspec	0.513	12	4	rr	0.35	4	300					30	60	0.03	12	48		20	5											method comparison
run26	constantQspec	0.645	4	2	fan	0.05	7	700					40	10	0.1	12	48		20	5											
run27	constantQspec	0.639	8	2	fan	0.05	2	500					40	0	0.1	24	48		20	5											
run28	constantQspec	0.662	4	2		0.05	7	700					40	0	0.05	12	48	1.5	20	5											
run29	CENS_dsf2_wl41	~~0.831~~	2	5	fan	0.15	7	900											20	5											
run30	CENS_dsf2_wl41	~~0.831~~	2	5	fan	0.15	7	900											20	5											
run31	constantQspec	0.658	8	3	fan	0.05	4	500		False			40	20	0.05	84	48		20	5											
run32	chroma	0.639	4	5	fan	0.05	4	700					40	0	0.05	6	12		20	5											method comparison
run33	chroma	0.566	16	3	rr	0.05	4	300					40	10	0.05	6	12		20	5											method comparison
run34	CENS_dsf2_wl41	~~0.832~~	2	5	fan	0.15	7	900					40	40	0.05	12	12		20	5											
run35	logfreqspec	0.694	4	1	fan	0.05	4	500					40	10	0.05	12	256	1.5	20	5											
run36	CENS_dsf2_wl41	~~0.837~~	2	5	fan	0.15	7	900					40	70	0.05	9	12		20	5											

Run Name	Feature Name	Mean Average Precision	DownSample Factor	CRP Dimension	CRP Method	CRP Neighbourhood Size	CRP Time Delay	NCD Sequence Length	NN Time Stacking	NN Frequency Standardisation	NN Num Folders	NN Num Files per Folder	dA Batch Size	dA Corruption Level	dA Learning Rate	dA # Hidden Units	dA # Visible Units	dA Learning Rate Boost Factor	Folders	Files per Folder	FENS Downsampling	FENS Normalisation Threshold	FENS Quantisation Step Base	FENS Quantisation Step Power	FENS Quantisation Weight 1	FENS Quantisation Weight 2	FENS Quantisation Weight 3	FENS Quantisation Weight 4	FENS Transformation Function	FENS Window Length	Notes
run37	CENS_dsf2_wl41	~~0.839~~	2	5	fan	0.15	7	900					40	0	0.05	18	12		20	5											
run38	logfreqspec	0.698	4	1	fan	0.05	7	500					10	20	0.05	15	256		20	5											
run39	logfreqspec	0.656	1	3	fan	0.05	2	500					40	0	0.05	15	128		20	5											
run40	logfreqspec-dA-256v-12h-FENS_dsf2_dswl41	~~0.426~~	8	2	fan	0.05	4	700					40						20	5											
run41	CENS_dsf2_wl41	~~0.777~~	2	4	fan	0.2	7	1100					40	0	0.05	12	12		20	5											
run42	logfreqspec	0.581	8	4	fan	0.05	2	500					40	60	0.05	15	256		20	5											
run43	logfreqspec-dA-256v-12h-FENS_dsf8_dswl70	~~0.441~~	4	5	fan	0.05	7	1100					40						20	5											
run44	logfreqspec-dA-256v-12h-FENS_dsf8_dswl41	~~0.437~~	4	5	fan	0.05	7	1100					40						20	5											
run45	logfreqspec	0.604	4	1	fan	0.05	7	500	1	True	20	5	40	20	0.05	15		1.5	20	5											weights fine-tuning
run46	*CENS_dsf2_wl41*	*0.707 -> 0.727*	*2*	*4*	*fan*	*0.05*	*7*	*900*	*1*	*True*			*40*	*70*	*0.05*	*12*		*1.5*	*20*	*5*											*weights fine-tuning*
run47	*CENS_dsf2_wl41*	*0.754 -> 0.764*	*2*	*4*	*fan*	*0.2*	*7*	*1100*	*1*	*False*			*40*	*70*	*0.05*	*12*		*1.5*	*20*	*5*											*weights fine-tuning*
run48	logfreqspec-dA-256v-12h-FENS_dsf1_dswl20	0.624	4	5	fan	0.05	7	1100	1	False	20	5	40						20	5											
run49	logfreqspec-dA-256v-12h-FENS_dsf8_dswl30	0.635	1	5	fan	0.1	5	900	1	True	20	5	40						20	5											
run50	logfreqspec-dA-256v-12h-FENS_dsf8_dswl30	0.645	1	5	fan	0.15	5	700	1	True	20	5	40						20	5											
run51	logfreqspec-dA-256v-12h-FENS_dsf10_dswl41	0.634	1	5	fan	0.1	7	500	1	True	20	5	40						20	5	10										41	
run52	logfreqspec	0.645	1	4	fan	0.1	7	900	1	True	20	5	40	10	0.05	12	256	1.5	20	5	12										50	
run53	logfreqspec	0.653	1	4	fan	0.1	5	500					40	30	0.05	12	128	1.5	20	5	8										60	
run54	constantQspec	0.645	1	1	fan	0.1	2	300	1	False	20	5	40	50	0.05	12	48	1.5	20	5	1										20	
run55	constantQspec	0.704	8	4	fan	0.25	5	300	1	False	20	5	40	10	0.05	12	48	1.5	20	5	2	0	0.09	1.9	0.8	1.2	1	1.1		70	
run56	constantQspec	0.675	1	5	fan	0.15	2	1100	1	False	20	5	40	40	0.05	12	48	1.5	20	5	4	0	0.03	1.9	0.9	0.8	1	1.1	cube	30	
run57	logfreqspec	0.678	4	4	fan	0.2	4	700	1	False	20	5	40	40	0.05	12	128	1.5	20	5	2	0.01	0.09	2	0.8	1	0.8	0.9		50	
run58	logfreqspec	0.698	2	5	fan	0.2	3	300	1	False	20	5	40	30	0.05	12	256	1.5	20	5	8	0.001	0.03	2	1.1	0.8	1.2	1.2	cube	60	
run59	logfreqspec-dA-128v-12h-FENS_dsf10_dswl41	0.481	2	5	fan	0.45	2	300																							
run60	logfreqspec	0.613	2	5	fan	0.1	3	900	1	True			40	10	0.05	12	128	1.5	20	5	12	0.001	0.05	1.8	1.2	0.9	0.8	0.8		70	
run61	constantQspec	0.691	8	4	fan	0.1	7	1100	1	True	20	5	40	40	0.05	12	48	1.5	20	5	1	0.0001	0.09	1.8	1.1	1.2	1	1.2		50	
run62	logfreqspec	0.721	2	5	fan	0.15	5	300	1	True	20	5	40	0	0.05	12	128	1.5	20	5	8	0.003	0.05	1.9	1	1.414	0.5	0.707	tenth	70	
run63	constantQspec	0.752	4	4	fan	0.25	4	300	1	True	20	5	40	10	0.05	12	48	1.5	20	5	4	0.0001	0.03	2	2	2	2	0.707	tenth	60	
run64	chroma	0.596	2	2	fan	0.05	7	1100					40	10	0.05	12	12	1.5	20	5											
run65	chroma	0.606	4	2	fan	0.05	7	700					40	0	0.05	12	12	1.5	20	5											
run66	CENS_dsf10_wl41	0.722	2	4	fan	0.1	2	300																							

Appendix D ORIGINAL PLANNED PROJECT TIMELINE

Name	Begin date	End date
Write Project Plan	14/04/15	24/04/15
Submit Project Plan	27/04/15	27/04/15
Kick-off meeting	01/06/15	01/06/15
Source Material	01/06/15	12/06/15
Choose Source Audio	01/06/15	05/06/15
Extract Primary Features	08/06/15	12/06/15
Software	01/06/15	12/06/15
Testing and Selection	01/06/15	05/06/15
Network Installation	08/06/15	12/06/15
Progress Meeting 1	15/06/15	15/06/15
Feature Learning	15/06/15	17/07/15
Build and Test Models	15/06/15	26/06/15
Run Feature Extraction	29/06/15	03/07/15
Evaluate Results	06/07/15	10/07/15
Additional Tests	13/07/15	17/07/15
Progress Meeting 2	29/06/15	29/06/15
Replicate Existing System	29/06/15	17/07/15
Delay-Coordinate Embedding	29/06/15	03/07/15
Recurrence Plots	06/07/15	10/07/15
Calculation of NCD	13/07/15	17/07/15
Progress Meeting 3	20/07/15	20/07/15
Evaluation	20/07/15	07/08/15
Run Feature Extraction	20/07/15	24/07/15
Run Models	27/07/15	31/07/15
Evaluate Results	03/08/15	07/08/15
Progress Meeting 4	10/08/15	10/08/15
Report	10/08/15	24/09/15
Finish First Draft	10/08/15	14/08/15
First Draft to Supervisor	17/08/15	17/08/15
1st Review	17/08/15	21/08/15
Address Comments	24/08/15	28/08/15
2nd Review	31/08/15	04/09/15
Address Comments	07/09/15	11/09/15
Submit Final Report	25/09/15	25/09/15



Appendix E COMMUNICATION WITH DR. NORBERT MARWAN

Dear Dr. Marwan,

Thanks for granting me access to the toolbox - it will be very useful for my project.

I was going through the code in the crp.m file to understand how it works, and I noticed a possible bug that might occur if the matrix passed in is transposed and contains null values. The check for null values occurs before the transposition of the input matrix, which I believe could lead to deletion of a whole column of values in some situations (lines 279 - 289). I think that the check should occur after the matrix transposition.

```
if ~isempty(find(isnan(x)))

disp('Warning: NaN detected (in first variable) - will be cleared.')

for k=1:size(x,2), x(find(isnan(x(:,k))),:)=[]; end

end

if ~isempty(find(isnan(y)))

disp('Warning: NaN detected (in second variable) - will be cleared.')

for k=1:size(y,2), y(find(isnan(y(:,k))),:)=[]; end

end

if size(x,1)<size(x,2), x=x'; end

if size(y,1)<size(y,2), y=y'; end
```

Regards,

Vahndi

Dear Vahndi

thanks for this hint! Yes you are right. I have changed it and it will be published in the next update.

Best

Norbert

Dear Norbert,

Thanks for the reply! The code is genius – it took me a long time to work out all the vectorisations. I also found a potential issue with the calculation of the minimum norm but I am not sure. At line 650 of *crp.m* the maximum norm is calculated as:

```
s = max(abs(s1),[],3);
```

And then at line 668 the minimum norm is calculated as:

```
s = sum(abs(s1), 3);
```

Should this instead be:

```
s = min(abs(s1),[],3);
```

Regards,

Vahndi

no, it is correct. it is the L1 norm:

http://mathworld.wolfram.com/L1-Norm.html

but I agree that the name is misleading.

Appendix F EXAMPLE CODE

The full code written for the project totals well in excess of 100 files, each of which may have several hundred lines of code. Therefore the full project code is submitted as an accompanying zip file, and will be made publically available online after submittal of the Project at https://bitbucket.org/vahndi/msc_project. The subsections of this appendix list the code for some of the key files, for interested readers.

F.1 CENS AND FENS FEATURE GENERATION FUNCTIONS

```
1.  from __future__ import division
2.
3.
4.  import pandas as pd
5.  import numpy as np
6.  import copy
7.  import cPickle, gzip
8.  import pickle
9.  from scipy import hanning
10. from math import ceil, floor
11. from multirate import upfirdn
12. from itertools import islice
13. from numpy.linalg import norm
14.
15. from paths import getFileNames
16. from other import rcut
17.
18.
19.
20. def smoothDownSampleFeature(dataframe, windowLength, downSampleFactor):
21.     '''''
22.     Temporal smoothing and downsampling of a feature sequence.
23.     Adapted from the smoothDownsampleFeature.m file of the Matlab Chroma Toolbox
24.     at http://resources.mpi-inf.mpg.de/MIR/chromatoolbox/
25.     '''
26.     def downsample_to_proportion(rows, proportion = 1):
27.         return list(islice(rows, 0, len(rows), int(1 / proportion)))
28.
29.     if windowLength == 1 and downSampleFactor == 1:
30.         return dataframe
31.
32.     statWindow = hanning(windowLength)
33.     statWindow = statWindow / statWindow.sum()
34.     statWindow = np.tile(statWindow, [1, 1])
35.
36.     f_feature = dataframe.as_matrix()
37.     seg_num = f_feature.shape[0]
38.     stat_num = int(ceil(seg_num / downSampleFactor))
39.     f_feature_stat = upfirdn(f_feature, statWindow.transpose(), 1, downSampleFactor
    )
40.     cut = floor((windowLength - 1) / (2 * downSampleFactor))
41.     f_feature_stat = f_feature_stat[cut: stat_num + cut, :]
42.
43.     timeIndex = downsample_to_proportion(dataframe.index, 1 / downSampleFactor)
44.     dfSmoothed = pd.DataFrame(f_feature_stat, index = timeIndex)
45.
46.     return dfSmoothed
47.
48.
49. def normaliseFeature(dataframe, normalisationPower, threshold):
```

```
    '''''
    Normalizes a feature sequence according to the l^p norm
    - If the norm falls below threshold for a feature vector, then the
      normalized feature vector is set to be the unit vector.
    Code adapted from the normalizeFeature.m file of the Matlab Chroma Toolbox
    at http://resources.mpi-inf.mpg.de/MIR/chromatoolbox/
    '''
    unit_vec = np.ones([1, dataframe.shape[1]]) # changed from the Matlab version of using 12 in case this is used for other feature types
    unit_vec = unit_vec / norm(unit_vec, normalisationPower)
    dfValues = dataframe.as_matrix()
    norms = norm(dfValues, ord = normalisationPower, axis = 1)
    smallIndices = np.where(norms < threshold)[0]
    bigIndices = np.where(norms >= threshold)[0]

    dfValues[smallIndices, :] = unit_vec
    dfValues[bigIndices, :] = dfValues[bigIndices, :] / np.expand_dims(norms[bigIndices], 1)

    df = pd.DataFrame(dfValues, index = dataframe.index)

    return df


def chromaToCENS(dfChroma,
                 normalisationThreshold = 0.001,
                 quantisationSteps = [0.4, 0.2, 0.1, 0.05], quantisationWeights = [1, 1, 1, 1],
                 downSampleWindowLength = 41, downSampleFactor = 10.0):
    '''''
    Converts a chroma feature .csv file created by Sonic Annotator's Queen Mary Chroma
    plugin into a Chroma Energy Normalised Statistics File
    Code converted from the pitch_to_CENS.m file of the Matlab Chroma Toolbox
    at http://resources.mpi-inf.mpg.de/MIR/chromatoolbox/
    '''

    # Normalise the chroma vectors (Matlab line 123)
    # ----------------------------------------------
    dfChroma['Chroma Sum'] = dfChroma.apply(lambda row: row.sum(), axis = 1)
    dfChroma['Exceeds Threshold'] = dfChroma.apply(lambda row: int(row['Chroma Sum'] > normalisationThreshold), axis = 1)
    dfChromaStandardised = dfChroma[[i for i in range(1, 13)]].multiply(dfChroma['Exceeds Threshold'], axis = 'index').divide(dfChroma['Chroma Sum'], axis = 'index')

    # Calculate CENS Feature
    # ----------------------

    # Component-
wise quantisation of the normalised chroma vectors (Matlab line 134)
    ChromaArray = dfChromaStandardised.as_matrix()
    CENSarray = np.zeros(dfChromaStandardised.shape)
    for i in range(len(quantisationSteps)):
        CENSarray += quantisationWeights[i] * (ChromaArray > quantisationSteps[i])

    dfCENSraw = pd.DataFrame(CENSarray, index = dfChromaStandardised.index)

    # Temporal smoothing and downsampling (Matlab line 140)
            dfCENSsmoothed = smoothDownSampleFeature(dfCENSraw, downSampleWindowLength, downSampleFactor)

            # Normalise each vector with its l^2 norm (Matlab line 143)
            dfCENS = normaliseFeature(dfCENSsmoothed, 2, normalisationThreshold)

            # Return CENS dataframe
```

```
106.            # ---------------------
107.            return dfCENS
108.
109.
110.        def featuresToFENS(dfFeatures, numFeatures,
111.                           transformationFunction = None,
112.                           normalisationThreshold = 0.001,
113.                           quantisationSteps = [0.4, 0.2, 0.1, 0.05],
114.                           quantisationWeights = [1, 1, 1, 1],
115.                           downSampleWindowLength = 41, downSampleFactor = 10.0):
116.            '''''
117.            Mimics the creation of CENS features but using any dataframe of features
     of any dimension
118.            Inputs:
119.                :dfFeatures: a pandas dataframe of the features
120.                :other settings: see documentation for chromaToCENS
121.            '''
122.            # Sort quantisation steps descending in case they have been passed in a
    different order
123.            quantisationSteps = sorted(quantisationSteps, reverse = True)
124.
125.            # Transform features if the function name is given
126.            if transformationFunction is not None:
127.                dfFeatures = applyFENSTransformationFunction(dfFeatures, transformat
    ionFunction)
128.
129.            # Rescale the features to a range from 0 to 0.2
130.            # N.B. this is an additional step to the chromaToCENS script to reduce t
    he
131.            # scale of the feature to be the same range as typical chroma features
132.            dfFeatures = 0.2 * (dfFeatures - dfFeatures.min()) / (dfFeatures.max() -
     dfFeatures.min())
133.
134.
135.            # Standardise the learned feature vectors (Matlab line 123)
136.            # ---------------------------------------------------------
137.            dfFeatures['Features Sum'] = dfFeatures.apply(lambda row: row.sum(), axi
    s = 1)
138.            dfFeatures['Exceeds Threshold'] = dfFeatures.apply(lambda row: int(row['
    Features Sum'] >
139.                                                                                norma
    lisationThreshold),
140.                                                                    axis = 1)
141.            dfFeaturesStandardised = dfFeatures[[i for i in range(1, numFeatures + 1
    )]].multiply(dfFeatures['Exceeds Threshold'],
142.
                  axis = 'index').divide(dfFeatures['Features Sum'],
143.
                                         axis = 'index')
144.
145.            # Calculate FENS Feature
146.            # ----------------------
147.            # Component-wise quantisation of the normalised chroma vectors
148.            FeaturesArray = dfFeaturesStandardised.as_matrix()
149.            FENSarray = np.zeros(dfFeaturesStandardised.shape)
150.            for i in range(len(quantisationSteps)):
151.                FENSarray += quantisationWeights[i] * (FeaturesArray > quantisationS
    teps[i])
152.            dfFENSraw = pd.DataFrame(FENSarray, index = dfFeaturesStandardised.index
    )
153.            # Temporal smoothing and downsampling
154.            dfFENSsmoothed = smoothDownSampleFeature(dfFENSraw, downSampleWindowLeng
    th, downSampleFactor)
155.            # Normalise each vector with its l^2 norm
156.            dfFENS = normaliseFeature(dfFENSsmoothed, 2, normalisationThreshold)
157.
```

```
158.            # Return FENS dataframe
159.            # ---------------------
160.            return dfFENS
```

F.2 Calculation of Recurrence Plots

```
1.  import numpy as np
2.
3.
4.  def normalise(X, doNormalisation):
5.
6.      scale = X[:, 0]
7.      assert (np.diff(scale) >= 0).all(), 'First column of the first vector must be m
    onotonically non-decreasing.'
8.      idx = np.where(X[:, 1:] != np.inf)
9.      if doNormalisation == True:
10.         Xnorm = (X[:, 1:] - X[idx, 1:].mean()) / X[idx, 1:].std()
11.     else:
12.         Xnorm = X[:, 1:]
13.
14.     return scale, Xnorm
15.
16.
17. def getDistance(X, NX, Y, NY, dimension, timeDelay):
18.
19.     delays = np.arange(0, dimension * timeDelay, timeDelay)
20.     s = np.zeros([NX, NY, delays.size])
21.     for iRow in np.arange(NX):
22.         IX = np.tile(iRow + delays, [NY, 1]).transpose()
23.         arrXs = X[IX]
24.         iCols = np.tile(np.arange(NY), [delays.size, 1])
25.         rowDelays = np.tile(delays, [NY, 1]).transpose()
26.         IY = rowDelays + iCols
27.         arrYs = Y[IY]
28.         s[iRow, :, :] = np.sqrt(np.sum(np.power(arrXs - arrYs, 2), axis = 2).transp
    ose())
29.
30.     return s
31.
32.
33. def getDistanceNrmNorm(X, NX, Y, NY, dimension, timeDelay):
34.
35.     delays = np.arange(0, dimension * timeDelay, timeDelay)
36.     s = np.zeros([NX, NY, delays.size])
37.     for iRow in np.arange(NX):
38.         IX = np.tile(iRow + delays, [NY, 1]).transpose()
39.         arrXs = X[IX] / np.sqrt(np.power(X[IX], 2).sum(axis = 0))
40.         iCols = np.tile(np.arange(NY), [delays.size, 1])
41.         rowDelays = np.tile(delays, [NY, 1]).transpose()
42.         IY = rowDelays + iCols
43.         arrYs = Y[IY] / np.sqrt(np.power(Y[IY], 2).sum(axis = 0))
44.         s[iRow, :, :] = np.sqrt(np.sum(np.power(arrXs - arrYs, 2), axis = 2).transp
    ose())
45.
46.     return s
47.
48.
49. def crp_v4(x, y = None,
50.             dimension = 1, timeDelay = 1, neighbourhoodSize = 0.1,
51.             method = 'euclidean', normalisation = True):
52.     '''''
53.     Does a recurrence plot of x (and y) and returns a recurrence matrix
54.
55.     If y is not None then does a cross-recurrence plot\n
56.     If y is None the does an auto-recurrence plot\n
57.
58.     Args:
```

```
59.         :x (2D numpy.ndarray): first time-series (first column is time)
60.         :y (2D numpy.ndarray): (optional) second time-
    series (first column is time)
61.         :dimension (int): dimension of input (not including time column)
62.         :timeDelay (int): time delay to use for embedding
63.         :neighbourhoodSize: neighbourhood size
64.         :method (str): Methods for finding the neighbours of the plot
65.             (maxnorm: maximum norm,
66.             euclidean: Euclidean norm,
67.             minnorm: minimum norm,
68.             nrmnorm: Euclidean norm of normalized distance,
69.             rr: maximum norm fixed RR,
70.             fan: fixed amount of nearest neighbours,
71.             inter: interdependent neighbours [not implemented],
72.             omatrix: order matrix [not implemented],
73.             opattern: order pattern [not implemented],
74.             distance: distance plot [not implemented])
75.         :normalisation (boolean): whether to standardise values to mean = 0, stdev
    = 1
76.
77.     Returns:
78.         :2D numpy array: Recurrence Matrix
79.
80.     Changes:
81.         :v2:    Dropped 'inter' method.
82.         :v2:    Unvectorised the implementation to consume less memory in preparati
    on for use with
83.                 multi-dimensional feature vectors. This was accomplished by
84.
85.                 - replacing the previous embedVector() and get_s1() functions for m
    ethods ['maxnorm',
86.                   'euclidean', 'minnorm', 'rr', 'fan'] with a combined getDistance(
    ) function
87.                 - replacing the previous implementation of the 'nrmnorm' method wit
    h getDistanceNrmNorm()
88.
89.         :v3:    Modified the normalise(), getDistance() and getDistanceNrmNorm() fu
    nctions to handle
90.                 multi-dimensional feature vectors.
91.
92.         :v4:    Partially revectorised the getDistance() and getDistanceNrmNorm() f
    unctions to increase
93.                 execution speed - iterations are done over entire rows instead of r
    ows and columns
94.     '''
95.
96.     # Methods for finding the neighbours of plot
97.     # ------------------------------------------
98.     methods = ['maxnorm', 'euclidean', 'minnorm',
99.                'nrmnorm', 'rr', 'fan']
100.
101.            # Error Checks
102.            # ------------
103.            assert dimension >= 1, 'dimension must be at least 1'
104.            assert timeDelay >= 1, 'timeDelay must be at least 1'
105.            assert neighbourhoodSize >= 0, 'neighbourhoodSize must not be negative'

106.            assert method in methods, 'method is invalid'
107.
108.
109.            # Cross-recurrence or auto-recurrence plot (Matlab line 253)
110.            # ----------------------------------------------------------
111.            if y == None:
112.                y = x
113.
114.            # Check that matrices are right way up (Matlab line 288)
```

```
115.            # ------------------------------------------------------
116.            if x.shape[1] > x.shape[0]:
117.                x = x.transpose()
118.            if y.shape[1] > y.shape[0]:
119.                y = y.transpose()
120.
121.            # Check for and delete rows with nans (Matlab line 279)
122.            # ------------------------------------------------------
123.            if True in np.isnan(x).any(axis = 1):
124.                print('Warning: NaN detected in x - time slice will be removed.')
125.                x = x[~np.isnan(x).any(axis = 1)]
126.            if True in np.isnan(y).any(axis = 1):
127.                print('Warning: NaN detected in x - time slice will be removed.')
128.                y = y[~np.isnan(y).any(axis = 1)]
129.
130.            # Calculate embedding vectors lengths (Matlab line 291)
131.            # ------------------------------------------------------
132.            NX = x.shape[0] - timeDelay * (dimension - 1)
133.            assert NX >= 1, 'The embedding vectors cannot be created: dimension and
    / or timeDelay are too big. Please use smaller values.'
134.            NY = y.shape[0] - timeDelay * (dimension - 1)
135.            assert NY >= 1, 'The embedding vectors cannot be created: dimension and
    / or timeDelay are too big. Please use smaller values.'
136.
137.            # Normalise the data (Matlab line 297)
138.            # ------------------------------------
139.            xScale, x = normalise(x, normalisation)
140.            yScale, y = normalise(y, normalisation)
141.
142.
143.            # Computation
144.            # -----------
145.
146.            X = np.zeros([NY, NX], np.uint8)
147.
148.            # local CRP, fixed distance (Matlab line 634)
149.            if method in ['maxnorm', 'euclidean', 'minnorm', 'rr']:
150.
151.                s1 = getDistance(x, NX, y, NY, dimension, timeDelay)
152.
153.                if method == 'maxnorm':
154.                    # maximum norm (Matlab line 649)
155.                    s = np.abs(s1).max(axis = 2) # take the maximum of the absolute
    distances across the dimension axis
156.
157.                elif method == 'rr':
158.                    # maximum norm, fixed RR (Matlab line 653)
159.                    s = np.abs(s1).max(axis = 2) # take the maximum of the distances
     across the dimension axis
160.                    ss = np.sort(s.reshape([s.shape[0] * s.shape[1], 1]), axis = 0)
    # ss = convert s to vector and sort
161.                    idx = np.ceil(neighbourhoodSize * ss.size)
162.                    neighbourhoodSize = ss[idx - 1] # set neighbourhood size to be t
    he value at neighbourhoodSize * length of ss along ss
163.
164.                elif method == 'euclidean':
165.                    # euclidean norm (Matlab line 662)
166.                    s = np.sqrt(np.power(s1, 2).sum(axis = 2)) # take the square roo
    t of the sum of the squared distances across the dimension axis
167.
168.                elif method == 'minnorm':
169.                    # minimum norm (Matlab line 666)
170.                    s = np.abs(s1).sum(axis = 2) # take the sum of the absolute dist
    ances across the dimension axis (should this be min not sum?!?!)
171.
172.                X2 = s < neighbourhoodSize
```

```
173.                X = np.uint8(X2)
174.
175.            # local CRP, normalized distance euclidean norm (Matlab line 679)
176.            elif method == 'nrmnorm':
177.
178.                s1 = getDistanceNrmNorm(x, NX, y, NY, dimension, timeDelay)
179.                s = np.sqrt(np.power(s1, 2).sum(axis = 2))
180.                X = np.uint8(s / s.max() < neighbourhoodSize / s.max())
181.                del s, s1
182.
183.            # local CRP, fixed amount of neigbours (Matlab line 710)
184.            elif method == 'fan':
185.
186.                assert neighbourhoodSize < 1, 'The value for fixed neigbours amount
    has to be smaller than one.'
187.
188.                s1 = getDistance(x, NX, y, NY, dimension, timeDelay)
189.                s = np.power(s1, 2).sum(axis = 2)
190.                minNS = round(NY * neighbourhoodSize)
191.                JJ = np.argsort(s, axis = 1)
192.                X1 = np.zeros(NX * NY, dtype = np.uint8)
193.                X1[JJ[:, 0: minNS] + np.tile(np.matrix(np.arange(0, NX * NY, NY)).tr
    anspose(), [1, minNS])] = np.uint8(1)
194.                X = X1.reshape(NY, NX).transpose()
195.                del s, s1, JJ, X1
196.
197.
198.            return X
```

F.3 Calculation of the Normalised Compression Distance

```
1.  import bz2
2.
3.
4.  def memoryNCD(data1, data2, compressionLevel = 9):
5.
6.      C_o1 = len(bz2.compress(data1, compresslevel = compressionLevel))
7.      C_o2 = len(bz2.compress(data2, compresslevel = compressionLevel))
8.      C_o1o2 = len(bz2.compress(data1 + data2, compresslevel = compressionLevel))
9.
10.     f_ncd = float(C_o1o2 - min(C_o1, C_o2)) / max(C_o1, C_o2)
11.
12.     dictFileNCD = {}
13.     dictFileNCD['Compression Level'] = compressionLevel
14.     dictFileNCD['C_o1'] = C_o1
15.     dictFileNCD['C_o2'] = C_o2
16.     dictFileNCD['C_o1o2'] = C_o1o2
17.     dictFileNCD['NCD'] = f_ncd
18.
19.     return dictFileNCD
20.
21.
22. def fileNCD(inputFn1, inputFn2, compressionLevel = 9):
23.
24.     f1 = open(inputFn1, 'rb')
25.     data1 = f1.read()
26.     f1.close()
27.
28.     f2 = open(inputFn2, 'rb')
29.     data2 = f2.read()
30.     f2.close()
31.
32.     return memoryNCD(data1, data2, compressionLevel)
```

F.4 Calculation of Mean Average Precision from NCD pairs

```
import pandas as pd
import numpy as np


def _getAveragePrecision(dataFrame, queryPiece):
    '''''
    Returns the average precision for a particular query (i.e. performance of a
    piece) from the dataframe
    '''
    def getOmega(row, pieceId):

        return int((row['Piece 1 Id'] == pieceId) &
                   (row['Piece 2 Id'] == pieceId))

    # Filter dataframe to only include NCDs featuring the query piece
    dfPiece = dataFrame[((dataFrame['Piece 1 Id'] == queryPiece[0]) &
                         (dataFrame['Piece 1 Performance Id'] == queryPiece[1])) |

                        ((dataFrame['Piece 2 Id'] == queryPiece[0]) &
                         (dataFrame['Piece 2 Performance Id'] == queryPiece[1]))]

    # Add Rank and Omega
    dfPieceRanked = dfPiece.sort('NCD')
    dfPieceRanked['Rank'] = 1 + np.argsort(dfPieceRanked['NCD'])
    dfPieceRanked['Omega'] = dfPieceRanked.apply(lambda row: getOmega(row, queryPie
ce[0]), axis = 1)
    dfPieceRanked = dfPieceRanked[dfPieceRanked['Omega'] == 1]

    # Calculate Precision at each row
    sumOmega = 0
    precision = []
    for idx in dfPieceRanked.index:
        sumOmega += dfPieceRanked.ix[idx]['Omega']
        precision.append(float(sumOmega) / dfPieceRanked.ix[idx]['Rank'])
    dfPieceRanked.loc[:, 'Precision'] = pd.Series(precision, index = dfPieceRanked.
index)

    # Calculate and return Average Precision for the Query
    averagePrecision = dfPieceRanked['Precision'].mean()
    return averagePrecision


def getMeanAveragePrecision(dataFrame):
    '''''
    Returns the mean average precision for all queries of results in a dataframe
    The dataframe should therefore only contain results from one configuration of
    CRP settings
    '''
    # Get unique pieces and performance tuples
    piece1Ids = list(dataFrame['Piece 1 Id'])
    piece1PerformanceIds = list(dataFrame['Piece 1 Performance Id'])
    piecesPerformances1 = zip(piece1Ids, piece1PerformanceIds)
    piece2Ids = list(dataFrame['Piece 2 Id'])
    piece2PerformanceIds = list(dataFrame['Piece 2 Performance Id'])
    piecesPerformances2 = zip(piece2Ids, piece2PerformanceIds)
    pieces = sorted(list(set(piecesPerformances1 + piecesPerformances2)))

    # Get Precision for each query
    averagePrecisions = []
    for queryPiece in pieces:
        averagePrecision = _getAveragePrecision(dataFrame, queryPiece)
```

```
59.         averagePrecisions.append(averagePrecision)
60.
61.     # Return Mean Average Precision
62.     averagePrecisions = np.array(averagePrecisions)
63.     return averagePrecisions.mean()
```

F.5 Calculation of Mean Average Precision for a Single Parameter Setting

```
from __future__ import division

import numpy as np

from FeatureFileProps import FeatureFileProps
from paths import getFolderNames
from FeatureFileProps import FeatureFileProps as FFP
from MAPhelpers import calculateRequiredNCDs, calculateRequiredCRPs
from ncd import memoryNCD


# Revision History
# ----------------


### As create_ncds{i}.py ###

# v1: Initial version

# v2: This version compares all files in 2 folders

# v3: This version compares a specified number of files from each folder

# v4: This version allows for changing the length of the CRPs before finding the NCD

# v5: This version allows for an additional preprocessing step on input features
#    to apply the weights of a neural network before calculating CRPs

# v6: This version allows for stacking of horizontal features to a certain number
#    of time steps before multiplication by the weight matrix to incorporate
#    temporal effects from the feature learning

# v7: Added support for using an integer subset of features from the feature files

# v8: Moved a lot of the functions out to different files


### As calculate_NCDs.py ###

# v1 Doing everything in memory now - only use for training runs


def calculateCRP(crpProps):

    crpProps.calculateCRP()
    return crpProps


def multi_calculateCRPs(args):
    '''''
    Run wrapper for calculateCRP()
    '''
    return calculateCRP(*args)


def calculateNCD(NCDfn, CRP1, CRP2):

    ncd = memoryNCD(CRP1, CRP2)
    return (NCDfn, ncd)
```

```
60.
61.
62. def multi_calculateNCDs(args):
63.     '''''
64.     Run wrapper for calculateNCD()
65.     '''
66.     return calculateNCD(*args)
67.
68.
69. def calculateNCDs(processPool,
70.                     featureName, numFeatures,
71.                     downSampleFactor, timeDelay, dimension, method, neighbourhoodSize
    ,
72.                     numFolders, numFilesPerFolder, sequenceLength,
73.                     weightMatrix = None, biases = None, featureOffset = 0.0, featureS
    caling = 1.0, timeStacking = None,
74.                     featureFileDict = None, pieceIds = None):
75.     '''''
76.     Inputs:
77.         :processPool: a pool of multiprocessing processes to use for running the sc
    ript
78.         :featureName: the name of the feature e.g 'chroma', 'mfcc'
79.         :downSampleFactor: the factor to use in downsampling the original signals b
    efore creating CRPs
80.         :timeDelay: the time delay to use in creating the CRPs
81.         :method: the method to use in creating the CRPs
82.         :neighbourhoodSize: the neighbourhood size to use in creating the CRPs
83.         :numFilesPerFolder: the number of performances of each piece to use - set t
    o None to use all performances
84.         :sequenceLength: fixed sequence length to normalise CRPs to (use 'var' for
    variable length)
85.
86.     Feature Transformation Inputs (optional - specify all or none):
87.         :weightMatrix: a matrix of weights (inputFeatureLength rows x outputFeature
    Length columns)
88.                         to transform the input feature files with before calculating
     the CRPs
89.         :biases: a matrix of biases to add to the transformed features
90.         :featureOffset: the offset to add to the features before scaling
91.         :featureScaling: the scaling to apply to the features
92.         :timeStacking: how much to stack the features by horizontally
93.
94.     Precalculated Inputs:
95.         :featureFileDict: a pre-loaded or -calculated featureFileDict
96.         :pieceIds: The names of the pieces
97.     '''
98.
99.     # Get performances from folders or use ones in memory
100.          piecesPath = FeatureFileProps.getRootPath(featureName)
101.          if pieceIds is None:
102.              pieceIds = getFolderNames(piecesPath, contains = 'mazurka', orderAlp
    habetically = True)[:numFolders]
103.          if featureFileDict is None:
104.              print 'Loading feature file dict'
105.              featureFileDict = FFP.loadFeatureFileDictAllFolders(piecesPath, piec
    eIds, featureName, numFilesPerFolder)
106.
107.          # Create list of required NCDs
108.          requiredNCDs = calculateRequiredNCDs(featureFileDict, method, dimension,

109.                                               timeDelay, neighbourhoodSize, downS
    ampleFactor,
110.                                               sequenceLength, featureName)
111.
112.          # Create Required CRPs for NCD files
113.          if len(requiredNCDs) > 0:
```

```
114.
115.                 # Calculate required CRPs from featureFileDict
116.                 requiredCRPs = calculateRequiredCRPs(requiredNCDs)
117.                 numRequiredCRPs = len(requiredCRPs)
118.
119.                 print 'Creating %i required CRPs' % numRequiredCRPs
120.                 if numRequiredCRPs > 0:
121.                     CRPargList = []
122.                     for crp in requiredCRPs:
123.                         # assign CRP properties
124.                         crp.weightMatrix = weightMatrix
125.                         crp.biases = biases
126.                         crp.featureOffset = featureOffset
127.                         crp.featureScaling = featureScaling
128.                         crp.timeStacking = timeStacking
129.                         crp.numFeatures = numFeatures
130.                         # find feature file data for CRP from featureFileDict
131.                         # currently matches on path for a file that does not exist -
     not pretty
132.                         for featureFileId in featureFileDict.keys():
133.                             featureFileProps = featureFileDict[featureFileId]
134.                             if featureFileProps.pieceId == crp.pieceId and \
135.                                featureFileProps.performanceId == crp.performanceId:

136.                                 crp.featureFileData = featureFileProps.featureFileDa
    ta
137.                         CRPargList.append((crp, ))
138.                     CRPs = processPool.map(multi_calculateCRPs, CRPargList)
139.
140.                     # Calculate NCDs
141.                     numNCDs = len(requiredNCDs)
142.                     print 'Creating %i NCDs' % numNCDs
143.                     NCDindex = 0
144.                     NCDs = []
145.                     while NCDindex < numNCDs:
146.                         NCDargList = []
147.                         for iNCD in np.arange(NCDindex, min(NCDindex + 100, numNCDs)
    ):
148.                             requiredNCD = requiredNCDs[iNCD]
149.                             NCDfn = requiredNCD.getFileName()
150.                             crp1 = requiredNCD.getCRP1()
151.                             crp2 = requiredNCD.getCRP2()
152.                             CRPdata1 = None
153.                             CRPdata2 = None
154.                             for crp in CRPs:
155.                                 if crp.pieceId == crp1.pieceId and crp.performanceId
     == crp1.performanceId:
156.                                     CRPdata1 = crp.CRPdata
157.                                 if crp.pieceId == crp2.pieceId and crp.performanceId
     == crp2.performanceId:
158.                                     CRPdata2 = crp.CRPdata
159.                             if CRPdata1 is None or CRPdata2 is None:
160.                                 return None
161.                             NCDargList.append((NCDfn, CRPdata1.tostring(), CRPdata2.
    tostring()))
162.                         if NCDargList:
163.                             NCDs.extend(processPool.map(multi_calculateNCDs, NCDargL
    ist))
164.                         NCDindex += 100
165.                         print '\r%i...' % NCDindex,
166.
167.                     return NCDs
168.
169.             return None
```

F.6 Optimiser Class used in Greedy Hill-Jumping Algorithm

```
import pandas as pd
from numpy import float64, random, inf


class Optimiser(object):


    def __init__(self, categoricalSettings,
                 oldResultsDataFrame = None, resultsColumn = 'Results',
                 noImprovementStoppingRounds = None, floatRounding = 3):
        '''''
        Arguments:
            :categoricalSettings: a {dict of settingName, [setting1, setting2, ...]
}
            :oldResultsDataFrame: a pandas DataFrame of existing results to use in
the selection of new settings
            :resultsColumn:  the name of the results column in the results DataFram
e
            :floatRounding:  the number of decimal places to round float values to
        '''
        self.categoricalSettings = categoricalSettings
        self.resultsColumn = resultsColumn
        self.resultsDataFrame = oldResultsDataFrame
        self.floatRounding = floatRounding
        self.categories = sorted(list(self.categoricalSettings.keys()))
        self.numCategories = len(self.categories)
        self.currentCategoryIndex = 0
        self.noImprovementStoppingRounds = noImprovementStoppingRounds

        # Initialise current settings to random values
        self.initialiseRandomSettings()


    def initialiseRandomSettings(self):
        '''''
        Randomly set the settings to different values
        '''

        self.roundsNoImprovement = 0
        self.currentSettings = {}
        for category in self.categories:
            self.currentSettings[category] = Optimiser._getRandomValue(self.categor
icalSettings[category])


    @classmethod
    def _getRandomValue(cls, fromList):

        return fromList[random.randint(len(fromList))]


    def isFinished(self):

        return False


    def hasResultFor(self, settings):

        if self.resultsDataFrame is None:
            return False
```

```
        else:
            dfSub = self.resultsDataFrame
            for category in settings.keys():
                categoryValue = settings[category]
                dfSub = dfSub[dfSub[category] == categoryValue]
            return dfSub.shape[0] > 0


    def getNextSettings(self):
        '''''
        Returns a list of settings to try next
        '''
        if self.noImprovementStoppingRounds is not None:
            if self.roundsNoImprovement == self.noImprovementStoppingRounds:
                return None

        # Get a list of settings across the dimension of the current category
        nextSettings = []
        numCategoriesTried = 0
        # Loop until some new settings have been acquired or all categories have be
    en tried
        while not nextSettings and numCategoriesTried < self.numCategories:

            loopCategory = self.categories[self.currentCategoryIndex]
            for val in self.categoricalSettings[loopCategory]:
                setting = {}
                for category in self.categories:
                    if category == loopCategory:
                        setting[category] = val
                    else:
                        setting[category] = self.currentSettings[category]
                nextSettings.append(setting)

            # Remove any settings which already have results for
            nonDuplicates = []
            for setting in nextSettings:
                if not self.hasResultFor(setting):
                    nonDuplicates.append(setting)
            nextSettings = nonDuplicates

            # Update loop and category parameters
            numCategoriesTried += 1
            self.currentCategoryIndex += 1
                    self.currentCategoryIndex = self.currentCategoryIndex % self.num
    Categories

            # Return the list of settings or None if the run is finished
            if not nextSettings:
                return None
            else:
                self._currentSettingsParameter = self.categories[(self.currentCa
    tegoryIndex - 1) % self.numCategories]
                return nextSettings


        def getCurrentSettingsParameter(self):

            return self._currentSettingsParameter


        def currentBestResult(self):

            if self.resultsDataFrame is None:
                return -inf
            else:
                return self.resultsDataFrame[self.resultsColumn].max()
```

```
121.
122.
123.           def addResults(self, resultsDataFrame):
124.               '''''
125.               Adds a list of results to the existing results and changes the curre
    nt
126.               settings to the best result so far
127.               '''
128.               # Add results to any existing results
129.               if self.resultsDataFrame is None:
130.                   self.resultsDataFrame = resultsDataFrame
131.               else:
132.                   # Check for improvement
133.                   if self.currentBestResult() >= resultsDataFrame[self.resultsColu
    mn].max():
134.                       self.roundsNoImprovement += 1
135.                   else:
136.                       self.roundsNoImprovement = 0
137.
138.                   # Merge results with existing results
139.                   self.resultsDataFrame = pd.concat([self.resultsDataFrame, result
    sDataFrame], ignore_index=True)
140.
141.               df = self.resultsDataFrame
142.
143.               # Get the best result so far and change the current settings to the
    settings that produced it
144.               bestResult = df[self.resultsColumn].max()
145.               bestSettingsRow = df[df[self.resultsColumn] == bestResult].iloc[0]
146.               for category in self.categories:
147.                   categoryValue = bestSettingsRow[category]
148.                   if type(categoryValue) in (float64, float):
149.                       categoryValue = round(categoryValue, self.floatRounding)
150.                       if categoryValue == round(categoryValue, 0):
151.                           categoryValue = int(round(categoryValue, 0))
152.
153.                   self.currentSettings[category] = categoryValue
154.
155.
156.           def saveResults(self, filePath):
157.
158.               self.resultsDataFrame.to_csv(filePath)
```

F.7 Stepper and Adjuster classes used in Weights fine-tuning and Coercion of FENS Distributions

```
import numpy as np


class ListStepper(object):
    '''''
    A class to randomly step along the elements of a list
    '''
    def __init__(self, listValues, initialValue = None):

        self._listValues = listValues
        self._numValues = len(listValues)
        if initialValue is not None and initialValue in listValues:
            self._initialValue = initialValue
            self._currentValue = initialValue
            self._adjustedValue = initialValue
        else:
            initialIndex = np.rand.randint(self._numValues)
            self._initialValue = self._listValues[initialIndex]
            self._currentValue = self._initialValue
            self._adjustedValue = self._initialValue

        self._numUnsuccessfulAdjustments = 0
        self._lastAdjustmentSuccess = False
        self._currentAdjustment = None


    def getCurrentValue(self):

        return self._currentValue


    def getAdjustedValue(self):

        currentIndex = self._listValues.index(self._currentValue)
        if currentIndex == 0:
            self._currentAdjustment = np.random.randint(2)
        elif currentIndex == self._numValues - 1:
            self._currentAdjustment = - np.random.randint(2)
        else:
            self._currentAdjustment = np.random.randint(3) - 1

        self._adjustedValue = self._listValues[currentIndex + self._currentAdjustme
nt]
        return self._adjustedValue


    def keepCurrentValue(self):

        self._numUnsuccessfulAdjustments += 1
        self._lastAdjustmentSuccess = False


    def keepAdjustedValue(self):

        self._numUnsuccessfulAdjustments = 0
        self._currentValue = self._adjustedValue
        self._lastAdjustmentSuccess = True


```

```
60.     def getNumUnsuccessulAdjustments(self):
61.
62.         return self._numUnsuccessfulAdjustments
63.
64.
65.
66. class ValueAdjuster(object):
67.
68.
69.     def __init__(self, initialValue, minValue = None, maxValue = None, maxAdjustmen
    t = 0.001):
70.
71.         self._initialValue = initialValue
72.         self._currentValue = initialValue
73.         self._adjustedValue = initialValue
74.         self._numUnsuccessfulAdjustments = 0
75.         self._lastAdjustmentSuccess = False
76.         self._currentAdjustment = None
77.         self._minValue = minValue
78.         self._maxValue = maxValue
79.         self._maxAdjustment = maxAdjustment
80.
81.
82.     def getCurrentValue(self):
83.
84.         return self._currentValue
85.
86.
87.     def getAdjustedValue(self):
88.
89.         if not self._lastAdjustmentSuccess or self._currentAdjustment is None:
90.             self._currentAdjustment = (np.random.rand() - 0.5) * (2 * self._maxAdju
    stment)
91.             if self._minValue is not None:
92.                 while self._currentValue + self._currentAdjustment < self._minValue
    :
93.                     self._currentAdjustment = (np.random.rand() - 0.5) * (2 * self.
    _maxAdjustment)
94.             if self._maxValue is not None:
95.                 while self._currentValue + self._currentAdjustment > self._maxValue
    :
96.                     self._currentAdjustment = (np.random.rand() - 0.5) * (2 * self.
    _maxAdjustment)
97.
98.         self._adjustedValue = self._currentValue + self._currentAdjustment
99.         if self._minValue is not None:
100.                 if self._adjustedValue < self._minValue:
101.                     self._adjustedValue = self._minValue
102.             if self._maxValue is not None:
103.                 if self._adjustedValue > self._maxValue:
104.                     self._adjustedValue = self._maxValue
105.
106.             return self._adjustedValue
107.
108.
109.         def keepCurrentValue(self):
110.
111.             self._numUnsuccessfulAdjustments += 1
112.             self._lastAdjustmentSuccess = False
113.
114.
115.         def keepAdjustedValue(self):
116.
117.             self._numUnsuccessfulAdjustments = 0
118.             self._currentValue = self._adjustedValue
119.             self._lastAdjustmentSuccess = True
```

```
120.
121.
122.            def getNumUnsuccessulAdjustments(self):
123.
124.                return self._numUnsuccessfulAdjustments
125.
126.
127.
128.        class ArrayAdjuster(object):
129.
130.
131.            def __init__(self, initialArray, maxAdjustment = 0.001):
132.                '''''
133.                Arguments:
134.                    :initialArray: a numpy array or matrix
135.                '''
136.                self._initialArray = initialArray
137.                self._arrayShape = initialArray.shape
138.                self._currentArray = initialArray
139.                self._adjustedArray = initialArray # needed for initial call to keep
    AdjustedArray in finetune_weights.py
140.                self._numUnsuccessfulAdjustments = 0
141.                self._lastAdjustmentSuccess = False
142.                self._currentAdjustments = None
143.                self._maxAdjustment = maxAdjustment
144.
145.
146.            def getCurrentArray(self):
147.
148.                return self._currentArray
149.
150.
151.            def getAdjustedArray(self):
152.
153.                if not self._lastAdjustmentSuccess or self._currentAdjustments is No
    ne:
154.                    self._currentAdjustments = (np.random.random_sample(self._arrayS
    hape) - 0.5) * (2 * self._maxAdjustment)
155.
156.                self._adjustedArray = self._currentArray + self._currentAdjustments

157.                return self._adjustedArray
158.
159.
160.            def getAdjustmentMean(self):
161.
162.                if self._currentAdjustments is not None:
163.                    return self._currentAdjustments.mean()
164.
165.                return 0.0
166.
167.
168.            def getAdjustmentStandardDeviation(self):
169.
170.                if self._currentAdjustments is not None:
171.                    return self._currentAdjustments.std()
172.                return 0.0
173.
174.
175.            def keepCurrentArray(self):
176.
177.                self._numUnsuccessfulAdjustments += 1
178.                self._lastAdjustmentSuccess = False
179.
180.
181.            def keepAdjustedArray(self):
```

```
182.
183.                 self._numUnsuccessfulAdjustments = 0
184.                 self._currentArray = self._adjustedArray
185.                 self._lastAdjustmentSuccess = True
186.
187.
188.             def getNumUnsuccessulAdjustments(self):
189.
190.                 return self._numUnsuccessfulAdjustments
```

F.8 Learned Weights Fine-tuning Algorithm

```
import pickle
import pandas as pd
from datetime import datetime
from multiprocessing import Pool
from copy import deepcopy

from paths import runHistoryPath, getWeightsPath
from create_ncds7 import createNCDfiles
from ncd_processing import convertNCDfiles
from getResults2 import getDataFrameMAPresult
from MAPhelpers import moveResultsFiles, get_NN_NCD_params, cleanCRPfolder, cleanNCDfolder
from adjusters import ArrayAdjuster

################################### Settings ###################################

subFolder = 'run47'
numProcesses = 8
numFolders = 20
numFilesPerFolder = 5 # Set to None to compare all files in each folder


# Feature Settings
featureName = 'CENS_dsf2_wl41'
numFeatures = 12
numHidden = 12
downSampleFactor = 2

NNtype = 'dA'
batchSize = 40
corruptionLevel = 70
learningRate = 0.05
learningRateBoostFactor = 1.5
timeStacking = None
freqStd = False


# CRP settings
CRPmethod = 'fan'
CRPdimension = 4
CRPneighbourhoodSize = 0.2
CRPtimeDelay = 7

# NCD settings
NCDsequenceLength = 1100

# Fine Tuners
weightsAdjustment = 0.01
biasesAdjustment = 0.01

# Stop Criteria
stopRunningAt = datetime(2015, 9, 4, 18)
maxUnsuccessfulFinetunes = 50


################################################################################


# Load initial weights
weightsPath = getWeightsPath(NNtype, featureName, batchSize = batchSize,
```

```
                                learningRate = learningRate, learningRateBoostFactor =
    learningRateBoostFactor,
                                timeStacking = timeStacking, numFeatures = numFeatures
   , frequencyStandardisation = freqStd)
weightMatrix, biases, featureOffset, featureScaling = get_NN_NCD_params(
    weightsPath, featureName, learningRate, corruptionLevel, numFeatures,
    numHidden, batchSize,
    freqStd, numFolders, numFilesPerFolder, timeStacking)

origWeightMatrix = weightMatrix
origBiases = biases


# Initialise
cleanCRPfolder()
cleanNCDfolder()
existingNCDs = None
processPool = Pool(numProcesses)

numRuns = 0
iteration = 0


currentDateTime = datetime.now()

settingsList = []

while currentDateTime < stopRunningAt:

    # Initialise Iteration
    iteration += 1
    bestMAPresult = 0
    weightsAdjuster = ArrayAdjuster(origWeightMatrix)
    biasesAdjuster = ArrayAdjuster(origBiases)
    weightMatrix = weightsAdjuster.getCurrentArray()
    biases = biasesAdjuster.getCurrentArray()

    # Loop while maxUnsuccessfulFinetunes is not exceeded
    while (weightsAdjuster.getNumUnsuccessulAdjustments() < maxUnsuccessfulFinetune
   s
           and currentDateTime < stopRunningAt):

        createNCDfiles(existingNCDs, processPool,
                               featureName, numFeatures,
                               downSampleFactor, CRPtimeDelay,
                               CRPdimension, CRPmethod, CRPneighbourhoodSize,
                               numFolders, numFilesPerFolder, NCDsequenceLength,
                               weightMatrix, biases, featureOffset, featureScaling,
   timeStacking)

            # create and save a record of the run settings
            runDict = {'CRP Dimension': CRPdimension,
                       'CRP Method': CRPmethod,
                       'CRP Neighbourhood Size': CRPneighbourhoodSize,
                       'CRP Time Delay': CRPtimeDelay,
                       'Feature DownSample Factor': downSampleFactor,
                       'Feature Name': featureName,
                       'NCD Sequence Length': NCDsequenceLength,
                       'NN Time Stacking': timeStacking,
                       'NN Type': NNtype,
                       'Run Name': subFolder,
                       'dA Batch Size': batchSize,
                       'dA Corruption Level': corruptionLevel,
                       'dA Learning Rate': learningRate,
                       'dA Num Hidden Units': numHidden,
```

```
121.                         'dA Learning Rate Boost Factor': learningRateBoostFactor
    }
122.
123.                 # Save run settings
124.                 runTime = str(datetime.now()).replace(':', '-')
125.                 pickle.dump(runDict, open(runHistoryPath + runTime + '.pkl', 'wb'))

126.
127.                 # Convert NCD files into a dataframe
128.                 dfNCDs = convertNCDfiles(dataFrameFileName = runTime)
129.
130.                 # Create subfolders and move results files into them
131.                 NCDdest = moveResultsFiles(subFolder)
132.
133.                 # Get the overall MAP of the run and add to the setting
134.                 MAPresult = getDataFrameMAPresult(dfNCDs)
135.                 if MAPresult is not None:
136.                     print 'Mean Average Precision: %0.3f\n' % MAPresult
137.                     if MAPresult > bestMAPresult:
138.                         print 'MAP increased by %0.3f' % (MAPresult - bestMAPresult)

139.                         weightsAdjuster.keepAdjustedArray()
140.                         biasesAdjuster.keepAdjustedArray()
141.                         bestWeights = weightMatrix
142.                         bestBiases = biases
143.                         pickle.dump(bestWeights, open(NCDdest + subFolder + '_best_w
    eights_%i.pkl' % iteration, 'wb'))
144.                         pickle.dump(bestBiases, open(NCDdest + subFolder + '_best_bi
    ases_%i.pkl' % iteration, 'wb'))
145.                         bestMAPresult = MAPresult
146.                     else:
147.                         print 'MAP did not increase' % (MAPresult - bestMAPresult)
148.                         weightsAdjuster.keepCurrentArray()
149.                         biasesAdjuster.keepCurrentArray()
150.                 else:
151.                     print 'No MAP result found!'
152.                     weightsAdjuster.keepCurrentArray()
153.                     biasesAdjuster.keepCurrentArray()
154.
155.                 # Save the settings and results
156.                 settings = deepcopy(runDict)
157.                 settings['Run Index'] = numRuns
158.                 settings['Iteration'] = iteration
159.                 settings['Mean Average Precision'] = MAPresult
160.                 settings['Weight Adjustment Mean'] = weightsAdjuster.getAdjustmentMe
    an()
161.                 settings['Biases Adjustment Mean'] = biasesAdjuster.getAdjustmentMea
    n()
162.                 settings['Weight Adjustment StDev'] = weightsAdjuster.getAdjustmentS
    tandardDeviation()
163.                 settings['Biases Adjustment StDev'] = biasesAdjuster.getAdjustmentSt
    andardDeviation()
164.                 settingsList.append(settings)
165.                 df = pd.DataFrame(settingsList)
166.                 df.to_csv(NCDdest + subFolder + '_' + str(iteration) + '.csv')
167.
168.                 # Update the weights and biases matrices
169.                 weightMatrix = weightsAdjuster.getAdjustedArray(maxAdjustment = weig
    htsAdjustment)
170.                 biases = biasesAdjuster.getAdjustedArray(maxAdjustment = biasesAdjus
    tment)
171.
172.                 # Increment the number of runs
173.                 numRuns += 1
174.
```

```
175.                print '\nNumber of runs: %i\nIteration: %i\nBest Result: %0.3f\n\n'
    \
176.                                                        %(numRuns, iteration, bestMAPresul
    t)
177.
178.                # Prepare for next iteration
179.                cleanCRPfolder()
180.                cleanNCDfolder()
181.                currentDateTime = datetime.now()
```

F.9 FENS Distribution Coercion Algorithm

```
from datetime import datetime
import pandas as pd
from scipy.stats import norm
from numpy import linspace
import numpy as np
import pickle
import multiprocessing as mp

from adjusters import ValueAdjuster
from paths import getFolderNames, getFileNames, FENSparamsPath
from FeatureFileProps import FeatureFileProps as FFP
from NNs import get_NN_NCD_params
from featureAnalysis import loadFeatureFile, loadAndTransformFeatureFile
from featureConverter import featuresToFENS


################################### Settings ###################################

numProcesses = 8

featureName = 'logfreqspec'
numFeatures = 12

# network and weights
NNtype = 'dA'
numOriginalFeatures = 128
numNewFeatures = 12
batchSize = 40
corruptionLevel = 0
learningRate = 0.05
learningRateBoostFactor = 1.5
timeStacking = 1
frequencyStandardisation = False

numFolders = 20 # set to None to use all folders
numFilesPerFolder = 5 # set to None to use all files

# loops
#windowLengths = [10, 20, 30, 41, 50, 60, 70]
#FENSdownsamplings = [1, 2, 4, 8, 12]

targetDistribution = 'normal'

# Quantisation Parameters
maxQadjustment = 0.01
q1Args = (0.2, 0.01, 0.95, maxQadjustment)
q2Args  = (0.4, 0.01, 0.95, maxQadjustment)
q3Args  = (0.6, 0.01, 0.95, maxQadjustment)
q4Args  = (0.8, 0.01, 0.95, maxQadjustment)

# Quantisation Weights
maxWadjustment = 0.01
w1Args = (1.0, 0.5, 2, maxWadjustment)
w2Args = (1.0, 0.5, 2, maxWadjustment)
w3Args = (1.0, 0.5, 2, maxWadjustment)
w4Args = (1.0, 0.5, 2, maxWadjustment)

# Normalisation Threshold
maxNadjustment = 0.001
nArgs = (0.001, 0.0, 0.01, maxNadjustment)
```

```
62.
63.
64. # Stop Criteria
65. stopRunningAt = datetime(2015, 9, 16, 23)
66. maxUnsuccessfulChanges = 50
67.
68. ##############################################################################
69.
70.
71. def getTargetHistogram(targetDistribution):
72.
73.     assert targetDistribution in ['normal', 'uniform']
74.
75.     if targetDistribution == 'normal':
76.         x = linspace(0, 1,101)
77.         pdf_fitted = norm.pdf(x, loc = 0.5, scale = 0.2)
78.         normalHistogram = 0.1 + 0.9 * (pdf_fitted - pdf_fitted.min()) / (pdf_fitted
    .max() - pdf_fitted.min())
79.         return normalHistogram
80.
81.     elif targetDistribution == 'uniform':
82.
83.         x = np.ones(101)
84.         return x
85.
86.
87. def multiFeaturesToFENS(args):
88.
89.     features = args[0]
90.     numFeatures = args[1]
91.     quantisationSteps = args[2]
92.     quantisationWeights = args[3]
93.     normalisationThreshold = args[4]
94.
95.     return featuresToFENS(features, numFeatures,
96.                           quantisationSteps = quantisationSteps,
97.                           quantisationWeights = quantisationWeights,
98.                           normalisationThreshold = normalisationThreshold)
99.
100.
101.        pool = mp.Pool(numProcesses)
102.
103.        # TODO: Create function adjuster for common transformations of the feature b
    efore
104.        # converting to FENS
105.
106.        # Get the folders (performances)
107.        piecesPath = FFP.getRootPath(featureName)
108.        piecesFolders = getFolderNames(piecesPath,
109.                                       contains = 'mazurka',
110.                                       orderAlphabetically = True) # added the conta
    ins parameter to avoid the new powerspectrum folder
111.
112.        if numFolders is not None:
113.            piecesFolders = piecesFolders[: numFolders]
114.
115.        # Load weights and biases
116.        if NNtype is not None:
117.            weightMatrix, biases, featureOffset, featureScaling = get_NN_NCD_params(

118.                                                                    NNtype, featureN
    ame, learningRate, learningRateBoostFactor,
119.                                                                    corruptionLevel,
     numOriginalFeatures, numNewFeatures, batchSize,
120.                                                                    freqStd = freque
    ncyStandardisation, NNnumFolders = numFolders,
```

```
                                                                    NNnumFilesPerFolder = numFilesPerFolder,
                                                                    NNtimeStacking = timeStacking)
        # Load (and optionally transform) the feature files
        p = 0
        featuresDataFrames = []
        for piecesFolder in piecesFolders:
            performancesPath = FFP.getFeatureFolderPath(piecesPath + piecesFolder + '/', featureName)
            performances = getFileNames(performancesPath,
                                        orderAlphabetically = True,
                                        endsWith = '.csv')
            if numFilesPerFolder is not None:
                performances = performances[: numFilesPerFolder]
            for performance in performances:
                p+= 1
                print '\rloading feature file %i...' % p,

                performanceFilePath = performancesPath + performance
                if NNtype is None:
                    featuresDataFrames.append(loadFeatureFile(performanceFilePath))

                else:
                    featuresDataFrames.append(loadAndTransformFeatureFile(performanceFilePath,
                                                                          featureOffset, featureScaling,
                                                                          timeStacking, weightMatrix, biases))
            print

        # Create Target histogram
        targetHistogram = getTargetHistogram(targetDistribution)

        # Initialise
        numRuns = 0
        iteration = 0
        currentDateTime = datetime.now()

        settingsList = []

        while currentDateTime < stopRunningAt:

            # Initialise Iteration
            iteration += 1
            bestMatchResult = 1e9
            # initialise adjuster values
            q1Adjuster = ValueAdjuster(*q1Args)
            q2Adjuster = ValueAdjuster(*q2Args)
            q3Adjuster = ValueAdjuster(*q3Args)
            q4Adjuster = ValueAdjuster(*q4Args)
            w1Adjuster = ValueAdjuster(*w1Args)
            w2Adjuster = ValueAdjuster(*w2Args)
            w3Adjuster = ValueAdjuster(*w3Args)
            w4Adjuster = ValueAdjuster(*w4Args)
            nAdjuster = ValueAdjuster(*nArgs)
            # get current values from adjusters
            q1Value = q1Adjuster.getCurrentValue()
            q2Value = q2Adjuster.getCurrentValue()
            q3Value = q3Adjuster.getCurrentValue()
            q4Value = q4Adjuster.getCurrentValue()
            w1Value = w1Adjuster.getCurrentValue()
            w2Value = w2Adjuster.getCurrentValue()
            w3Value = w3Adjuster.getCurrentValue()
            w4Value = w4Adjuster.getCurrentValue()
```

```
180.            nValue = nAdjuster.getCurrentValue()
181.
182.            # Loop while maxUnsuccessfulChanges is not exceeded
183.            while (q1Adjuster.getNumUnsuccessulAdjustments() < maxUnsuccessfulChange
    s
184.                   and currentDateTime < stopRunningAt):
185.
186.                # Create FENS features (in memory - about 38MB for 100 files)
187.                FENSdataFrames = []
188.                f = 0
189.                quantisationSteps = sorted([q1Value, q2Value, q3Value, q4Value], rev
    erse = True)
190.                print 'Quantisation Steps: [%0.3f, %0.3f, %0.3f, %0.3f]' % (quantisa
    tionSteps[0],
191.                                                                            quantisa
    tionSteps[1],
192.                                                                            quantisa
    tionSteps[2],
193.                                                                            quantisa
    tionSteps[3])
194.                quantisationWeights = [w1Value, w2Value, w3Value, w4Value]
195.                print 'Quantisation Weights: [%0.3f, %0.3f, %0.3f, %0.3f]' % (w1Valu
    e, w2Value, w3Value, w4Value)
196.                print 'Normalisation Threshold: %0.4f' % nValue
197.                FENSdataFrames = pool.map(multiFeaturesToFENS, [(feature, numFeature
    s, quantisationSteps, quantisationWeights, nValue)
198.
                     for feature in featuresDataFrames])
199.
200.                # Join dataframes
201.                print 'Joining features...'
202.                dfAllFENSfeatures = pd.concat(FENSdataFrames, ignore_index = True)
203.                # Rescale from 0 to 1
204.                dfAllFENSfeatures = (dfAllFENSfeatures - dfAllFENSfeatures.min()) /
    (dfAllFENSfeatures.max() - dfAllFENSfeatures.min())
205.
206.                matchResult = 0
207.                for col in dfAllFENSfeatures.columns:
208.                    # Take histogram of FENS features
209.                    colHistogram = np.histogram(dfAllFENSfeatures[col], bins = 101)[
    0]
210.                    # Mutltiply FENS histogram by Target histogram
211.                    # Get result as a fraction of the squared Target distribution
212.                    matchResult += np.square(colHistogram - targetHistogram).mean()
    / (101 * numFeatures)
213.                print 'result = %0.3f' % matchResult
214.
215.                # If result is best so far, keep the current adjuster settings
216.                if matchResult < bestMatchResult:
217.                    print 'Match improved by %0.3f' % (matchResult - bestMatchResult
    )
218.                    q1Adjuster.keepAdjustedValue()
219.                    q2Adjuster.keepAdjustedValue()
220.                    q3Adjuster.keepAdjustedValue()
221.                    q4Adjuster.keepAdjustedValue()
222.                    w1Adjuster.keepAdjustedValue()
223.                    w2Adjuster.keepAdjustedValue()
224.                    w3Adjuster.keepAdjustedValue()
225.                    w4Adjuster.keepAdjustedValue()
226.                    nAdjuster.keepAdjustedValue()
227.                    bestMatchResult = matchResult
228.                    bestQuantisationSteps = quantisationSteps
229.                    bestQuantisationWeights = quantisationWeights
230.                    bestNormalisationThreshold = nValue
231.                    FENSparamFileName = FENSparamsPath + featureName + '_nf_%i' % nu
    mFeatures
```

```
232.                     if NNtype is not None:
233.                         FENSparamFileName += '_NNtype_%s_nvis_%i_nhid_%i_basz_%i_crpt_%i_lr_%0.2f_lrbf_%0.2f_ts_%i_fstd_%s' \
234.                     % (NNtype, numOriginalFeatures, numNewFeatures, batchSize, corruptionLevel,
235.                        learningRate, learningRateBoostFactor, timeStacking, frequencyStandardisation)
236.                     FENSparamFileName += '_dist_%s_iter_%i' % (targetDistribution, iteration)
237.                     resultsDict = {}
238.                     resultsDict['Best Quantisation Steps'] = bestQuantisationSteps
239.                     resultsDict['Best Quantisation Weights'] = bestQuantisationWeights
240.                     resultsDict['Best Normalisation Threshold'] = bestNormalisationThreshold
241.                     resultsDict['Histogram'] = np.histogram(dfAllFENSfeatures.as_matrix(), bins = 101)
242.                     resultsDict['Mean Squared Error'] = bestMatchResult
243.                     pickle.dump(resultsDict, open(FENSparamFileName, 'wb')) # TODO: fill in filename, including iteration
244.                 else:
245.                     print 'Match did not improve'
246.                     q1Adjuster.keepCurrentValue()
247.                     q2Adjuster.keepCurrentValue()
248.                     q3Adjuster.keepCurrentValue()
249.                     q4Adjuster.keepCurrentValue()
250.                     w1Adjuster.keepCurrentValue()
251.                     w2Adjuster.keepCurrentValue()
252.                     w3Adjuster.keepCurrentValue()
253.                     w4Adjuster.keepCurrentValue()
254.                     nAdjuster.keepCurrentValue()
255.
256.                 # Update the weights and biases matrices
257.                 q1Value = q1Adjuster.getAdjustedValue()
258.                 q2Value = q2Adjuster.getAdjustedValue()
259.                 q3Value = q3Adjuster.getAdjustedValue()
260.                 q4Value = q4Adjuster.getAdjustedValue()
261.                 w1Value = w1Adjuster.getAdjustedValue()
262.                 w2Value = w2Adjuster.getAdjustedValue()
263.                 w3Value = w3Adjuster.getAdjustedValue()
264.                 w4Value = w4Adjuster.getAdjustedValue()
265.                 nValue = nAdjuster.getAdjustedValue()
266.
267.                 # Increment the number of runs
268.                 numRuns += 1
269.                 print '\nNumber of runs: %i\nIteration: %i\nBest Result: %0.3f\n\n' \
270.                                                         %(numRuns, iteration, bestMatchResult)
271.
272.                 # Prepare for next iteration
273.                 currentDateTime = datetime.now()
```

F.10 MAP Training Algorithm (with or without neural network weights)

```
import pickle
import pandas as pd
from datetime import datetime
from multiprocessing import Pool

from paths import runHistoryPath, NCDpath, createPath
from optimiser import Optimiser
from calculate_NCDs import calculateNCDs
from ncd_processing import convertNCDs
from getResults2 import getDataFrameMAPresult
from NNs import get_NN_NCD_params

# This script was formally know as run_ncds6.py but has been renamed to reflect
# the type of run that it does i.e. an 'optimisation' over parameters to produce
# the best MAP

################################### Settings ###################################

subFolder = 'run66'

numProcesses = 10
numFolders = 20
numFilesPerFolder = 5 # Set to None to compare all files in each folder

# Feature Settings
featureName = 'CENS_dsf10_wl41'
numFeatures = 12

# CRP settings
CRPmethod = 'fan'
CRPdimensions = [1, 2, 3, 4, 5]
CRPneighbourhoodSizes = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45]
CRPtimeDelays = [1, 2, 3, 4, 5, 6, 7]
downSampleFactors = [2, 4, 8, 12, 24]

# NCD settings
sequenceLengths = [300, 500, 700, 900, 1100]

# Neural Network settings
#for weights
NNtype = None
learningRate = 0.05
learningRateBoostFactor = 1.5
batchSize = 40
numVisibles = [12]
numHiddens = [12]
corruptionLevels = [0]
timeStacking = 1
freqStd = False

stopRunningAt = datetime(2015, 9, 23, 23)

################################################################################

# Create folder for results
createPath(NCDpath + subFolder)

# Create a dictionary of settings to optimise over
settingsDict = {'Dimension': CRPdimensions,
                'DownSample Factor': downSampleFactors,
                'Neighbourhood Size': CRPneighbourhoodSizes,
```

```
                'Sequence Length': sequenceLengths,
                'Time Delay': CRPtimeDelays}

if NNtype is not None:
    settingsDict['dA Num Hidden Units'] = numHiddens
    settingsDict['dA Num Visible Units'] = numVisibles
    settingsDict['dA Corruption Level'] = corruptionLevels

# Initialise
numRuns = 0
processPool = Pool(numProcesses)
iteration = 0
opt = Optimiser(settingsDict,
                oldResultsDataFrame = None,
                resultsColumn = 'Mean Average Precision',
                noImprovementStoppingRounds = None)
currentDateTime = datetime.now()


while currentDateTime < stopRunningAt:

    nextSettings = True
    iteration += 1

    while nextSettings is not None and currentDateTime < stopRunningAt:
        nextSettings = opt.getNextSettings()
        if nextSettings is not None:
            for setting in nextSettings:
                # load weights etc. if this is for a neural net run
                if NNtype is not None:
                    weightMatrix, biases, featureOffset, featureScaling = get_NN_NCD_params(NNtype,
                     featureName, learningRate, learningRateBoostFactor, setting['dA Corruption Level'],
                     setting['dA Num Visible Units'], setting['dA Num Hidden Units'], batchSize,
                     freqStd, numFolders, numFilesPerFolder, timeStacking)
                else:
                    weightMatrix = biases = featureOffset = featureScaling = None

                # Calculate NCDs
                       for key in setting.keys():
                           print key, ':', setting[key]
                       NCDlist = calculateNCDs(processPool,
                                               featureName, numFeatures,
                                               setting['DownSample Factor'], setting['Time Delay'],
                                               setting['Dimension'], CRPmethod, setting['Neighbourhood Size'],
                                               numFolders, numFilesPerFolder, setting['Sequence Length'],
                                               weightMatrix, biases, featureOffset, featureScaling, timeStacking)

                       # Convert NCD files into a dataframe
                       runTime = str(datetime.now()).replace(':', '-')
                       MAPresult = None
                       if NCDlist is None: # there were errors e.g. in CRP calculation after downsampling
                            MAPresult = 0 # need to use something that is not None for the optimiser to find its best result
                       else:
                           dfNCDs = convertNCDs(NCDlist, dataFrameFileName = runTime)
                            # Get the overall MAP of the run and add to the setting
```

```
                            MAPresult = getDataFrameMAPresult(dfNCDs)
                        if MAPresult is not None and MAPresult != 0:
                            print 'Mean Average Precision: %0.3f\n' % MAPresult
                        else:
                            print 'No MAP result found!'
                        setting['Mean Average Precision'] = MAPresult

                        # Create and save a runDict of the settings and result
                        # assign single (non-optimised) settings to the runDict
                        runDict = {'featureName': featureName,
                                   'method': CRPmethod,
                                   'numFilesPerFolder': numFilesPerFolder}
                        if NNtype is not None:
                            runDict['NN Type'] = NNtype
                            runDict['dA Learning Rate'] = learningRate
                            runDict['dA Learning Rate Boost Factor'] = learningRateB
    oostFactor
                            runDict['dA Batch Size'] = batchSize
                            runDict['dA # Features'] = numFeatures
                            runDict['NN timeStacking'] = timeStacking
                            runDict['NN frequency standardisation'] = freqStd
                        # assign settings from optimiser
                        for key in setting.keys():
                            runDict[key] = setting[key]
                        runDict['Mean Average Precision'] = MAPresult

                        pickle.dump(runDict, open(runHistoryPath + runTime + '.pkl',
     'wb'))

                    # Increment the number of runs
                    numRuns += len(nextSettings)

                    # Add the results to the optimiser
                    opt.addResults(pd.DataFrame(nextSettings))
                    print '\nNumber of runs: %i\nIteration: %i\nBest Result: %0.3f\n
    \n' \
                          %(numRuns, iteration, opt.currentBestResult())

                    # Save a copy of the results dataframe for the current iteration

                    # after every setting that is run in case of interruption
                    df = opt.resultsDataFrame
                    df.to_csv(NCDpath + subFolder + '/' + subFolder + '_' + str(iter
    ation) + '.csv')

            # Clear optimiser settings for next iteration
            opt.resultsDataFrame = None
            opt.initialiseRandomSettings()
            currentDateTime = datetime.now()
```

F.11 MAP Training Algorithm (with FENS parameters)

```
import pickle
import pandas as pd
from datetime import datetime
from multiprocessing import Pool

from paths import runHistoryPath, NCDpath, createPath, getFolderNames
from optimiser import Optimiser
from calculate_NCDs import calculateNCDs
from ncd_processing import convertNCDs
from getResults2 import getDataFrameMAPresult
from NNs import get_NN_NCD_params
from FeatureFileProps import FeatureFileProps as FFP


# Train new FENS file features from an existing base feature and some neural
# network weights and FENS settings

# v2: added support for FENS parameters of quantisation steps and weights
# v3: added a transformation function to the FENS settings

################################### Settings ###################################

subFolder = 'run63'

numProcesses = 10
numFolders = 20
numFilesPerFolder = 5 # Set to None to compare all files in each folder

# Feature Settings
baseFeatureName = 'constantQspec'
numFeatures = 12

# FENS settings
FENStransformationFunctions = ['square', 'cube', 'fourth', 'fifth', 'sixth',
                               'seventh', 'eighth', 'ninth', 'tenth']
FENSwindowLengths = [10, 20, 30, 41, 50, 60, 70]
FENSdownsamplings = [1, 2, 4, 8, 10, 12]
FENSnormalisationThresholds = [0.0001, 0.0003, 0.001, 0.003, 0.01]
FENSquantisationStepBases = [0.01, 0.03, 0.05, 0.07, 0.09]
FENSquantisationStepPowers = [1.8, 1.9, 2.0, 2.1, 2.2] # these have been derived ba
sed on the top step not exceeding 1
FENSquantisationWeights1 = [0.5, 0.707, 1.0, 1.414, 2.0]
FENSquantisationWeights2 = [0.5, 0.707, 1.0, 1.414, 2.0]
FENSquantisationWeights3 = [0.5, 0.707, 1.0, 1.414, 2.0]
FENSquantisationWeights4 = [0.5, 0.707, 1.0, 1.414, 2.0]
FENScalcParameters = ['FENS Window Length', 'FENS Downsampling',
                      'FENS Transformation Function',
                      'FENS Normalisation Threshold',
                      'FENS Quantisation Step Base' , 'FENS Quantisation Step Power
',
                      'FENS Quantisation Weight 1', 'FENS Quantisation Weight 2',
                      'FENS Quantisation Weight 3', 'FENS Quantisation Weight 4',
                      'dA Num Hidden Units',  'dA Num Visible Units',
                      'dA Corruption Level'] # these categories trigger recalculati
on of FENS

# CRP settings
CRPmethod = 'fan'
CRPdimensions = [1, 2, 3, 4, 5]
CRPneighbourhoodSizes = [0.05, 0.1, 0.15, 0.2, 0.25]
CRPtimeDelays = [1, 2, 3, 4, 5, 6, 7]
```

```
downSampleFactors = [1, 2, 4, 8]

# NCD settings
sequenceLengths = [300, 500, 700, 900, 1100]

# Neural Network settings
#for weights
NNtype = 'dA'
learningRate = 0.05
learningRateBoostFactor = 1.5
batchSize = 40
numVisibles = [48]
numHiddens = [12]
corruptionLevels = [0, 10, 20, 30, 40]
timeStacking = 1
freqStd = False
NNnumFolders = 20
NNnumFilesPerFolder = 5

stopRunningAt = datetime(2015, 9, 23, 8)

################################################################################

# Create folder for results
createPath(NCDpath + subFolder)
createPath(runHistoryPath + subFolder)

# Create a dictionary of settings to optimise over
settingsDict = {'Dimension': CRPdimensions,
                'DownSample Factor': downSampleFactors,
                'Neighbourhood Size': CRPneighbourhoodSizes,
                'Sequence Length': sequenceLengths,
                'Time Delay': CRPtimeDelays,
                'FENS Window Length': FENSwindowLengths,
                'FENS Downsampling': FENSdownsamplings,
                'FENS Transformation Function':FENStransformationFunctions,
                'FENS Normalisation Threshold': FENSnormalisationThresholds,
                'FENS Quantisation Step Base' : FENSquantisationStepBases,
                'FENS Quantisation Step Power': FENSquantisationStepPowers,
                'FENS Quantisation Weight 1': FENSquantisationWeights1,
                'FENS Quantisation Weight 2': FENSquantisationWeights2,
                      'FENS Quantisation Weight 3': FENSquantisationWeights3,
                      'FENS Quantisation Weight 4': FENSquantisationWeights4}

      if NNtype is not None:
          settingsDict['dA Num Hidden Units'] = numHiddens
          settingsDict['dA Num Visible Units'] = numVisibles
          settingsDict['dA Corruption Level'] = corruptionLevels

      # Initialise
      numRuns = 0
      iteration = 0
      processPool = Pool(numProcesses)
      opt = Optimiser(settingsDict,
                      oldResultsDataFrame = None,
                      resultsColumn = 'Mean Average Precision',
                      noImprovementStoppingRounds = None,
                      floatRounding = 4)
      featureFileDict = None
      FENSfeatureFileDict = None
      currentDateTime = datetime.now()

      # Load base features
      piecesPath = FFP.getRootPath(baseFeatureName)
      pieceIds = getFolderNames(piecesPath, contains = 'mazurka', orderAlphabetica
  lly = True)[:numFolders]
```

```
124.        print 'Loading feature file dict...'
125.        featureFileDict = FFP.loadFeatureFileDictAllFolders(piecesPath, pieceIds, ba
    seFeatureName, numFilesPerFolder)
126.        print '...done.'
127.
128.
129.        # Iterate
130.        while currentDateTime < stopRunningAt:
131.
132.            nextSettings = True
133.            iteration += 1
134.
135.            while nextSettings is not None and currentDateTime < stopRunningAt:
136.                nextSettings = opt.getNextSettings()
137.                if nextSettings is not None:
138.                    print 'Trying %s' % opt.getCurrentSettingsParameter()
139.                    for setting in nextSettings:
140.                        # load weights etc. if this is for a neural net run
141.                        if NNtype is not None:
142.                            weightMatrix, biases, featureOffset, featureScaling = ge
    t_NN_NCD_params(NNtype,
143.                                baseFeatureName, learningRate, learningRateBoostFact
    or, setting['dA Corruption Level'],
144.                                setting['dA Num Visible Units'], setting['dA Num Hid
    den Units'], batchSize,
145.                                freqStd, NNnumFolders, NNnumFilesPerFolder, timeStac
    king)
146.                        else:
147.                            weightMatrix = biases = featureOffset = featureScaling =
     None
148.
149.
150.                        # Calculate featureFileDict of CENS features
151.                        if opt.getCurrentSettingsParameter() in FENScalcParameters o
    r FENSfeatureFileDict is None:
152.                            print 'Generating FENS features...'
153.                            FENSfeatureFileDict = FFP.generateFENSfeatureFileDict(
154.                                                    featureFileDict, numFeatures
    , baseFeatureName,
155.                                                    NNtype, setting['dA Num Visi
    ble Units'], setting['dA Num Hidden Units'],
156.                                                    weightMatrix, biases, featur
    eOffset, featureScaling, timeStacking,
157.                                                    setting['FENS Downsampling']
    , setting['FENS Window Length'],
158.                                                    FENStransformationFunction =
     setting['FENS Transformation Function'],
159.                                                    FENSnormalisationThreshold =
     setting['FENS Normalisation Threshold'],
160.                                                    FENSquantisationSteps = FFP.
    getFENSQuantisationSteps(setting['FENS Quantisation Step Base'],
161.
                              setting['FENS Quantisation Step Power']),
162.                                                    FENSquantisationWeights = [s
    etting['FENS Quantisation Weight 1'],
163.                                                                               s
    etting['FENS Quantisation Weight 2'],
164.                                                                               s
    etting['FENS Quantisation Weight 3'],
165.                                                                               s
    etting['FENS Quantisation Weight 4']],
166.                                                    processPool = processPool)
167.
168.                        # Calculate NCDs
169.                        for key in setting.keys():
170.                            print key, ':', setting[key]
```

```
171.                         featureName = '%s-%s-%iv-%i-
    FENS_dsf%i_dswl%i' % (baseFeatureName, NNtype,
172.                                                                    setting['d
    A Num Visible Units'],
173.                                                                    setting['d
    A Num Hidden Units'],
174.                                                                    setting['F
    ENS Downsampling'],
175.                                                                    setting['F
    ENS Window Length'])
176.                         NCDlist = calculateNCDs(processPool,
177.                                                 featureName, numFeatures,
178.                                                 setting['DownSample Factor'], settin
    g['Time Delay'],
179.                                                 setting['Dimension'], CRPmethod, set
    ting['Neighbourhood Size'],
180.                                                 numFolders, numFilesPerFolder, setti
    ng['Sequence Length'],
181.                                                 featureFileDict = FENSfeatureFileDic
    t, pieceIds = pieceIds)
182.
183.                         # Convert NCD files into a dataframe
184.                         runTime = str(datetime.now()).replace(':', '-')
185.                         MAPresult = None
186.                         if NCDlist is None: # there were errors e.g. in CRP calculat
    ion after downsampling
187.                             MAPresult = 0 # need to use something that is not None
    for the optimiser to find its best result
188.                         else:
189.                             dfNCDs = convertNCDs(NCDlist, dataFrameFileName = runTim
    e)
190.                             # Get the overall MAP of the run and add to the setting

191.                             MAPresult = getDataFrameMAPresult(dfNCDs)
192.                         if MAPresult is not None and MAPresult != 0:
193.                             print 'Mean Average Precision: %0.3f\n' % MAPresult
194.                         else:
195.                             print 'No MAP result found!'
196.                         setting['Mean Average Precision'] = MAPresult
197.
198.                         # Create and save a runDict of the settings and result
199.                         # assign single (non-optimised) settings to the runDict
200.                         runDict = {'Feature Name': baseFeatureName,
201.                                    'CRP Method': CRPmethod,
202.                                    'numFilesPerFolder': numFilesPerFolder}
203.                         if NNtype is not None:
204.                             runDict['NN Type'] = NNtype
205.                             runDict['dA Learning Rate'] = learningRate
206.                             runDict['dA Learning Rate Boost Factor'] = learningRateB
    oostFactor
207.                             runDict['dA Batch Size'] = batchSize
208.                             runDict['dA # Features'] = numFeatures
209.                             runDict['NN timeStacking'] = timeStacking
210.                             runDict['NN frequency standardisation'] = freqStd
211.                         # assign settings from optimiser
212.                         for key in setting.keys():
213.                             runDict[key] = setting[key]
214.                         runDict['Mean Average Precision'] = MAPresult
215.                         # save settings to run history path
216.                         pickle.dump(runDict, open(runHistoryPath + subFolder + '/' +
     runTime + '.pkl', 'wb'))
217.
218.                     # Increment the number of runs
219.                     numRuns += len(nextSettings)
220.
221.                     # Add the results to the optimiser
```

```
222.                     opt.addResults(pd.DataFrame(nextSettings))
223.                     print '\nNumber of runs: %i\nIteration: %i\nBest Result: %0.3f\n
    \n' \
224.                           % (numRuns, iteration, opt.currentBestResult())
225.
226.                     # Save a copy of the results dataframe for the current iteration
227.                     # after every setting that is run in case of interruption
228.                     df = opt.resultsDataFrame
229.                     df.to_csv(NCDpath + subFolder + '/' + subFolder + '_' + str(iter
    ation) + '.csv')
230.
231.             # Clear optimiser settings for next iteration
232.             opt.resultsDataFrame = None
233.             opt.initialiseRandomSettings()
234.             currentDateTime = datetime.now()
```

F.12 TRAINING OF DENOISING AUTOENCODERS (ADAPTED FROM EXAMPLE THEANO CODE)

```
"""
 This tutorial introduces denoising auto-encoders (dA) using Theano.

 Denoising autoencoders are the building blocks for SdA.
 They are based on auto-encoders as the ones used in Bengio et al. 2007.
 An autoencoder takes an input x and first maps it to a hidden representation
 y = f_{\theta}(x) = s(Wx+b), parameterized by \theta={W,b}. The resulting
 latent representation y is then mapped back to a "reconstructed" vector
 z \in [0,1]^d in input space z = g_{\theta'}(y) = s(W'y + b').  The weight
 matrix W' can optionally be constrained such that W' = W^T, in which case
 the autoencoder is said to have tied weights. The network is trained such
 that to minimize the reconstruction error (the error between x and z).

 For the denosing autoencoder, during training, first x is corrupted into
 \tilde{x}, where \tilde{x} is a partially destroyed version of x by means
 of a stochastic mapping. Afterwards y is computed as before (using
 \tilde{x}), y = s(W\tilde{x} + b) and z as s(W'y + b'). The reconstruction
 error is now measured between z and the uncorrupted input x, which is
 computed as the cross-entropy :
      - \sum_{k=1}^d[ x_k \log z_k + (1-x_k) \log( 1-z_k)]


 References :
   - P. Vincent, H. Larochelle, Y. Bengio, P.A. Manzagol: Extracting and
   Composing Robust Features with Denoising Autoencoders, ICML'08, 1096-1103,
   2008
   - Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle: Greedy Layer-Wise
   Training of Deep Networks, Advances in Neural Information Processing
   Systems 19, 2007

"""

'''''
Version 2 - instead of training to 15 epochs, trains until the change in cost funct
ion < e.g. 0.001
          - also removed saving of weights and images after each epoch and added sa
ving of final weights
'''

import os
import sys
import time

import numpy
import cPickle
import theano
import theano.tensor as T
from theano.tensor.shared_randomstreams import RandomStreams

from shared import load_data



# start-snippet-1
class dA(object):
    '''''
    Denoising Auto-Encoder class (dA)

    A denoising autoencoder tries to reconstruct the input from a corrupted
    version of it by projecting it first in a latent space and reprojecting
    it afterwards back in the input space. Please refer to Vincent et al.,2008
```

```
60.     for more details. If x is the input then equation (1) computes a partially
61.     destroyed version of x by means of a stochastic mapping q_D. Equation (2)
62.     computes the projection of the input into the latent space. Equation (3)
63.     computes the reconstruction of the input, while equation (4) computes the
64.     reconstruction error.
65.
66.     .. math::
67.
68.         \tilde{x} ~ q_D(\tilde{x}|x)                                     (1)
69.
70.         y = s(W \tilde{x} + b)                                           (2)
71.
72.         x = s(W' y  + b')                                                (3)
73.
74.         L(x,z) = -sum_{k=1}^d [x_k \log z_k + (1-x_k) \log( 1-z_k)]   (4)
75.
76.     '''
77.
78.     def __init__(self, numpy_rng, theano_rng = None,
79.                  input = None, n_visible = 12, n_hidden = 12,
80.                  W = None, bhid = None, bvis = None):
81.         '''''
82.         Initialize the dA class by specifying the number of visible units (the
83.         dimension d of the input ), the number of hidden units ( the dimension
84.         d' of the latent or hidden space ) and the corruption level. The
85.         constructor also receives symbolic variables for the input, weights and
86.         bias. Such a symbolic variables are useful when, for example the input
87.         is the result of some computations, or when weights are shared between
88.         the dA and an MLP layer. When dealing with SdAs this always happens,
89.         the dA on layer 2 gets as input the output of the dA on layer 1,
90.         and the weights of the dA are used in the second stage of training
91.         to construct an MLP.
92.
93.         :type numpy_rng: numpy.random.RandomState
94.         :param numpy_rng: number random generator used to generate weights
95.
96.         :type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
97.         :param theano_rng: Theano random generator; if None is given one is
98.                            generated based on a seed drawn from `rng`
99.
100.                :type input: theano.tensor.TensorType
101.                :param input: a symbolic description of the input or None for
102.                                 standalone dA
103.
104.                :type n_visible: int
105.                :param n_visible: number of visible units
106.
107.                :type n_hidden: int
108.                :param n_hidden:  number of hidden units
109.
110.                :type W: theano.tensor.TensorType
111.                :param W: Theano variable pointing to a set of weights that should b
    e
112.                          shared belong the dA and another architecture; if dA shoul
    d
113.                          be standalone set this to None
114.
115.                :type bhid: theano.tensor.TensorType
116.                :param bhid: Theano variable pointing to a set of biases values (for

117.                             hidden units) that should be shared belong dA and anoth
    er
118.                             architecture; if dA should be standalone set this to No
    ne
119.
120.                :type bvis: theano.tensor.TensorType
```

```
121.                :param bvis: Theano variable pointing to a set of biases values (for
122.                             visible units) that should be shared belong dA and anot
    her
123.                             architecture; if dA should be standalone set this to No
    ne
124.
125.
126.                '''
127.                self.n_visible = n_visible
128.                self.n_hidden = n_hidden
129.
130.                # create a Theano random generator that gives symbolic random values
131.                if not theano_rng:
132.                    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
133.
134.                # note : W' was written as `W_prime` and b' as `b_prime`
135.                if not W:
136.                    # W is initialized with `initial_W` which is uniformely sampled
137.                    # from -4*sqrt(6./(n_visible+n_hidden)) and
138.                    # 4*sqrt(6./(n_hidden+n_visible))the output of uniform if
139.                    # converted using asarray to dtype
140.                    # theano.config.floatX so that the code is runable on GPU
141.                    initial_W = numpy.asarray(
142.                        numpy_rng.uniform(low = -
    4 * numpy.sqrt(6. / (n_hidden + n_visible)),
143.                                          high = 4 * numpy.sqrt(6. / (n_hidden + n_v
    isible)),
144.                                          size = (n_visible, n_hidden)),
145.                        dtype = theano.config.floatX)
146.
147.                    W = theano.shared(value = initial_W, name = 'W', borrow = True)
148.
149.                if not bvis:
150.                    bvis = theano.shared(value = numpy.zeros(n_visible, dtype = thea
    no.config.floatX),
151.                                         borrow = True)
152.
153.                if not bhid:
154.                    bhid = theano.shared(value = numpy.zeros(n_hidden, dtype = thean
    o.config.floatX),
155.                                         name = 'b',
156.                                         borrow = True)
157.
158.                self.W = W
159.                # b corresponds to the bias of the hidden
160.                self.b = bhid
161.                # b_prime corresponds to the bias of the visible
162.                self.b_prime = bvis
163.                # tied weights, therefore W_prime is W transpose
164.                self.W_prime = self.W.T
165.                self.theano_rng = theano_rng
166.                # if no input is given, generate a variable representing the input
167.                if input is None:
168.                    # we use a matrix because we expect a minibatch of several
169.                    # examples, each example being a row
170.                    self.x = T.dmatrix(name = 'input')
171.                else:
172.                    self.x = input
173.
174.                self.params = [self.W, self.b, self.b_prime]
175.            # end-snippet-1
176.
```

```
177.          def get_corrupted_input(self, input, corruption_level):
178.              '''''
179.              This function keeps ``1-
    corruption_level`` entries of the inputs the
180.              same and zero-
    out randomly selected subset of size ``coruption_level``
181.              Note : first argument of theano.rng.binomial is the shape(size) of
182.                     random numbers that it should produce
183.                     second argument is the number of trials
184.                     third argument is the probability of success of any trial
185.
186.                      this will produce an array of 0s and 1s where 1 has a
187.                      probability of 1 - ``corruption_level`` and 0 with
188.                      ``corruption_level``
189.
190.                      The binomial function return int64 data type by
191.                      default.  int64 multiplicated by the input
192.                      type(floatX) always return float64.  To keep all data
193.                      in floatX when floatX is float32, we set the dtype of
194.                      the binomial to floatX. As in our case the value of
195.                      the binomial is always 0 or 1, this don't change the
196.                      result. This is needed to allow the gpu to work
197.                      correctly as it only support float32 for now.
198.
199.              '''
200.              return self.theano_rng.binomial(size = input.shape, n = 1,
201.                                              p = 1 - corruption_level,
202.                                              dtype = theano.config.floatX) * inpu
    t
203.
204.          def get_hidden_values(self, input):
205.              ''''' Computes the values of the hidden layer '''
206.              return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
207.
208.          def get_reconstructed_input(self, hidden):
209.              '''''
210.              Computes the reconstructed input given the values of the hidden laye
    r
211.              '''
212.              return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)
213.
214.          def get_cost_updates(self, corruption_level, learning_rate):
215.              """ This function computes the cost and the updates for one trainng
216.              step of the dA """
217.
218.              tilde_x = self.get_corrupted_input(self.x, corruption_level)
219.              y = self.get_hidden_values(tilde_x)
220.              z = self.get_reconstructed_input(y)
221.              # note : we sum over the size of a datapoint; if we are using
222.              #       minibatches, L will be a vector, with one entry per
223.              #       example in minibatch
224.              L = - T.sum(self.x * T.log(z) + (1 - self.x) * T.log(1 - z), axis=1)

225.              # note : L is now a vector, where each element is the
226.              #       cross-entropy cost of the reconstruction of the
227.              #       corresponding example of the minibatch. We need to
228.              #       compute the average of all these to get the cost of
229.              #       the minibatch
230.              cost = T.mean(L)
231.
232.              # compute the gradients of the cost of the `dA` with respect
233.              # to its parameters
234.              gparams = T.grad(cost, self.params)
235.              # generate the list of updates
236.              updates = [(param, param - learning_rate * gparam)
237.                         for param, gparam in zip(self.params, gparams)]
```

```
238.
239.                 return (cost, updates)
240.
241.
242.         def test_dA(dataset, numFeatures, numHidden,
243.                     minCorruptionLevel = 0.0, maxCorruptionLevel = 1.0, corruptionLe
    velStep = 0.1,
244.                     learning_rate = 0.1,
245.                     batch_size = 20, output_folder = 'dA_plots',
246.                     deltaCostStopThreshold = 0.001, learningRateBoostFactor = 1.5):

247.             '''''
248.             Inputs:
249.                 :dataset (str): path to the dataset in .pkl.gz format
250.                 :numFeatures: number of input features
251.                 :numHidden: number of hidden features
252.                 :learning_rate (float): learning rate used for training the DeNosing

253.                                         AutoEncoder
254.                 :batch_size (int): number of examples to use in each batch
255.                 :output_folder (str): where to put the weights and biases
256.                 :deltaCostStopThreshold (float): string
257.                 :learningRateBoostFactor: a factor to multiply the learning rate by
    when
258.                                           deltaCostStopThreshold is not met, use Non
    e to not use boosting
259.
260.             '''
261.
262.             datasets = load_data(dataset)
263.             train_set_x, train_set_y = datasets[0]
264.
265.             # compute number of minibatches for training, validation and testing
266.             n_train_batches = train_set_x.get_value(borrow = True).shape[0] / batch_
    size
267.
268.             # allocate symbolic variables for the data
269.             index = T.lscalar() # index to a [mini]batch
270.             x = T.matrix('x')  # the data is presented as rasterized images
271.
272.             # Create output folder if it doesn't exist
273.             if not os.path.isdir(output_folder):
274.                 os.makedirs(output_folder)
275.             os.chdir(output_folder)
276.
277.             origLearningRate = learning_rate
278.
279.             for corruptionLevel in numpy.arange(minCorruptionLevel,
280.                                                 maxCorruptionLevel + corruptionLevel
    Step,
281.                                                 corruptionLevelStep):
282.
283.                 strCorruptionLevel = str(int(100 * corruptionLevel))
284.                 learning_rate = origLearningRate
285.
286.                 ######################
287.                 # BUILDING THE MODEL #
288.                 ######################
289.
290.                 rng = numpy.random.RandomState(123)
291.                 theano_rng = RandomStreams(rng.randint(2 ** 30))
292.
293.                 da = dA(numpy_rng = rng, theano_rng = theano_rng,
294.                         input = x, n_visible = numFeatures, n_hidden = numHidden)
295.
```

```
296.                 cost, updates = da.get_cost_updates(corruption_level = corruptionLev
    el,
297.                                                     learning_rate = learning_rate)
298.
299.                 train_da = theano.function([index], cost, updates = updates,
300.                                            givens = {x: train_set_x[index * batch_si
    ze: (index + 1) * batch_size]})
301.
302.                 start_time = time.clock()
303.
304.                 ############
305.                 # TRAINING #
306.                 ############
307.
308.                 # Go through training epochs until stopping criterion is met
309.                 deltaCost = -1
310.                 epoch = 0
311.                 lastEpochCost = 1e6
312.                 epochCost = 1e6
313.
314.                 while deltaCost < -deltaCostStopThreshold or deltaCost < 0:
315.                     # Go through training set
316.                     c = []
317.
318.                     weights = da.W.get_value(borrow = True).T
319.                     biases = da.b.get_value(borrow = True)
320.                     lastEpochCost = epochCost
321.
322.                     for batch_index in xrange(n_train_batches):
323.                         c.append(train_da(batch_index))
324.
325.                     epochCost = numpy.mean(c)
326.                     print 'Training epoch %d, cost ' % epoch, epochCost
327.                     deltaCost = epochCost - lastEpochCost
328.
329.                     epoch += 1
330.
331.                     # Check stopping criteria and update learning rate
332.                     if abs(deltaCost) < deltaCostStopThreshold:
333.                         learning_rate *= learningRateBoostFactor
334.                         print 'increasing learning rate to %0.2f...' % learning_rate
    
335.                         cost, updates = da.get_cost_updates(corruption_level = corru
    ptionLevel,
336.                                                             learning_rate = learning
    _rate)
337.                         train_da = theano.function([index], cost, updates = updates,
    
338.                                                    givens = {x: train_set_x[index *
    batch_size: (index + 1) * batch_size]})
339.
340.                 # Save final weights and biases
341.                 outputFnRoot = 'corruption_%s_nin_%s_nhdn_%s_basz_%s_lnrt_%s' \
342.                                % (strCorruptionLevel, str(numFeatures), str(numHidden),
     str(batch_size), str(origLearningRate))
343.                 numpy.savetxt('weights_' + outputFnRoot + '.csv', weights, delimiter
     = ',')
344.                 numpy.savetxt('biases_' + outputFnRoot + '.csv', biases, delimiter =
     ',')
345.
346.                 # calculate time taken for training
347.                 end_time = time.clock()
348.                 training_time = (end_time - start_time)
349.
350.                 # save a list of the training parameters
351.                 if not os.path.exists('training_records.pkl'):
```

```
352.                     trainingRecords = []
353.                 else:
354.                     trainingRecords = cPickle.load(open('training_records.pkl', 'rb'
    ))
355.                 record = {'Corruption': corruptionLevel,
356.                           '# Feature': numFeatures,
357.                           '# Hidden Units': numHidden,
358.                           '# Epochs': epoch,
359.                           'Batch Size': batch_size,
360.                           'Initial Learning Rate': origLearningRate,
361.                           'Final Learning Rate': learning_rate,
362.                           'Delta Cost Stopping Threshold': deltaCostStopThreshold,
363.                           'Learning Rate Boost Factor' : learningRateBoostFactor,
364.                           'Final Cost': lastEpochCost}
365.
366.                 trainingRecords.append(record)
367.                 cPickle.dump(trainingRecords, open('training_records.pkl', 'wb'))
368.
369.                 # Print time taken for training
370.                 print >> sys.stderr, ('The ' + strCorruptionLevel + '% corruption co
    de for file ' +
371.                                       os.path.split(__file__)[1] +
372.                                       ' ran for %.2fm' % (training_time / 60.))
373.
374.             os.chdir('../')
```

F.13 TRAINING OF RBMS (ADAPTED FROM EXAMPLE THEANO CODE)

```
import time

try:
    import PIL.Image as Image
except ImportError:
    import Image

import numpy
import cPickle
import theano
import theano.tensor as T
import os

from theano.tensor.shared_randomstreams import RandomStreams

from utils import tile_raster_images
from logistic_sgd import load_data


# start-snippet-1
class RBM(object):
    """Restricted Boltzmann Machine (RBM)  """
    def __init__(self, input = None, n_visible=784, n_hidden=500,
                 W = None, hbias = None, vbias = None,
                 numpy_rng = None, theano_rng=None):
        """
        RBM constructor. Defines the parameters of the model along with
        basic operations for inferring hidden from visible (and vice-versa),
        as well as for performing CD updates.

        :param input: None for standalone RBMs or symbolic variable if RBM is
        part of a larger graph.

        :param n_visible: number of visible units

        :param n_hidden: number of hidden units

        :param W: None for standalone RBMs or symbolic variable pointing to a
        shared weight matrix in case RBM is part of a DBN network; in a DBN,
        the weights are shared between RBMs and layers of a MLP

        :param hbias: None for standalone RBMs or symbolic variable pointing
        to a shared hidden units bias vector in case RBM is part of a
        different network

        :param vbias: None for standalone RBMs or a symbolic variable
        pointing to a shared visible units bias
        """

        self.n_visible = n_visible
        self.n_hidden = n_hidden

        if numpy_rng is None:
            # create a number generator
            numpy_rng = numpy.random.RandomState(1234)

        if theano_rng is None:
            theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

        if W is None:
            # W is initialized with `initial_W` which is uniformely
```

```
            # sampled from -4*sqrt(6./(n_visible+n_hidden)) and
            # 4*sqrt(6./(n_hidden+n_visible)) the output of uniform if
            # converted using asarray to dtype theano.config.floatX so
            # that the code is runable on GPU
            initial_W = numpy.asarray(numpy_rng.uniform(low = -
4 * numpy.sqrt(6. / (n_hidden + n_visible)),
                                                        high = 4 * numpy.sqrt(6. /
(n_hidden + n_visible)),
                                                        size = (n_visible, n_hidden
)),
                                      dtype = theano.config.floatX)
            # theano shared variables for weights and biases
            W = theano.shared(value = initial_W, name = 'W', borrow = True)

        if hbias is None:
            # create shared variable for hidden units bias
            hbias = theano.shared(value = numpy.zeros(n_hidden, dtype=theano.config
.floatX),
                                  name = 'hbias', borrow = True)

        if vbias is None:
            # create shared variable for visible units bias
            vbias = theano.shared(value = numpy.zeros(n_visible, dtype = theano.con
fig.floatX),
                                  name ='vbias', borrow = True)

        # initialize input layer for standalone RBM or layer0 of DBN
        self.input = input
        if not input:
            self.input = T.matrix('input')

        self.W = W
        self.hbias = hbias
        self.vbias = vbias
        self.theano_rng = theano_rng
        # **** WARNING: It is not a good idea to put things in this list
        # other than shared variables created in this function.
        self.params = [self.W, self.hbias, self.vbias]
        # end-snippet-1

    def free_energy(self, v_sample):
        ''''' Function to compute the free energy '''
        wx_b = T.dot(v_sample, self.W) + self.hbias
               vbias_term = T.dot(v_sample, self.vbias)
               hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
               return -hidden_term - vbias_term

           def propup(self, vis):
               '''''This function propagates the visible units activation upwards t
o
               the hidden units

               Note that we return also the pre-sigmoid activation of the
               layer. As it will turn out later, due to how Theano deals with
               optimizations, this symbolic variable will be needed to write
               down a more stable computational graph (see details in the
               reconstruction cost function)

               '''
               pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
               return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activatio
n)]

           def sample_h_given_v(self, v0_sample):
               ''''' This function infers state of hidden units given visible units
    '''
```

```
120.                # compute the activation of the hidden units given a sample of
121.                # the visibles
122.                pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
123.                # get a sample of the hiddens given their activation
124.                # Note that theano_rng.binomial returns a symbolic sample of dtype
125.                # int64 by default. If we want to keep our computations in floatX
126.                # for the GPU we need to specify to return the dtype floatX
127.                h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
128.                                                     n=1, p=h1_mean,
129.                                                     dtype=theano.config.floatX)
130.                return [pre_sigmoid_h1, h1_mean, h1_sample]
131.
132.            def propdown(self, hid):
133.                '''This function propagates the hidden units activation downwards
    to
134.                the visible units
135.
136.                Note that we return also the pre_sigmoid_activation of the
137.                layer. As it will turn out later, due to how Theano deals with
138.                optimizations, this symbolic variable will be needed to write
139.                down a more stable computational graph (see details in the
140.                reconstruction cost function)
141.
142.                '''
143.                pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
144.                return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activatio
    n)]
145.
146.            def sample_v_given_h(self, h0_sample):
147.                ''''' This function infers state of visible units given hidden units
     '''
148.                # compute the activation of the visible given the hidden sample
149.                pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
150.                # get a sample of the visible given their activation
151.                # Note that theano_rng.binomial returns a symbolic sample of dtype
152.                # int64 by default. If we want to keep our computations in floatX
153.                # for the GPU we need to specify to return the dtype floatX
154.                v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
155.                                                     n=1, p=v1_mean,
156.                                                     dtype=theano.config.floatX)
157.                return [pre_sigmoid_v1, v1_mean, v1_sample]
158.
159.            def gibbs_hvh(self, h0_sample):
160.                ''''' This function implements one step of Gibbs sampling,
161.                    starting from the hidden state'''
162.                pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample
    )
163.                pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample
    )
164.                return [pre_sigmoid_v1, v1_mean, v1_sample,
165.                        pre_sigmoid_h1, h1_mean, h1_sample]
166.
167.            def gibbs_vhv(self, v0_sample):
168.                ''''' This function implements one step of Gibbs sampling,
169.                    starting from the visible state'''
170.                pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample
    )
171.                pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample
    )
172.                return [pre_sigmoid_h1, h1_mean, h1_sample,
173.                        pre_sigmoid_v1, v1_mean, v1_sample]
174.
175.            # start-snippet-2
176.            def get_cost_updates(self, lr=0.1, persistent=None, k=1):
177.                """This functions implements one step of CD-k or PCD-k
178.
```

```
179.                :param lr: learning rate used to train the RBM
180.
181.                :param persistent: None for CD. For PCD, shared variable
182.                    containing old state of Gibbs chain. This must be a shared
183.                    variable of size (batch size, number of hidden units).
184.
185.                :param k: number of Gibbs steps to do in CD-k/PCD-k
186.
187.                Returns a proxy for the cost and the updates dictionary. The
188.                dictionary contains the update rules for weights and biases but
189.                also an update of the shared variable used to store the persistent
190.                chain, if one is used.
191.
192.                """
193.
194.                # compute positive phase
195.                pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.inpu
    t)
196.
197.                # decide how to initialize persistent chain:
198.                # for CD, we use the newly generate hidden sample
199.                # for PCD, we initialize from the old state of the chain
200.                if persistent is None:
201.                    chain_start = ph_sample
202.                else:
203.                    chain_start = persistent
204.                # end-snippet-2
205.                # perform actual negative phase
206.                # in order to implement CD-k/PCD-k we need to scan over the
207.                # function that implements one gibbs step k times.
208.                # Read Theano tutorial on scan for more information :
209.                # http://deeplearning.net/software/theano/library/scan.html
210.                # the scan will return the entire Gibbs chain
211.                (
212.                    [
213.                        pre_sigmoid_nvs,
214.                        nv_means,
215.                        nv_samples,
216.                        pre_sigmoid_nhs,
217.                        nh_means,
218.                        nh_samples
219.                    ],
220.                    updates
221.                ) = theano.scan(
222.                    self.gibbs_hvh,
223.                    # the None are place holders, saying that
224.                    # chain_start is the initial state corresponding to the
225.                    # 6th output
226.                    outputs_info=[None, None, None, None, None, chain_start],
227.                    n_steps=k
228.                )
229.                # start-snippet-3
230.                # determine gradients on RBM parameters
231.                # note that we only need the sample at the end of the chain
232.                chain_end = nv_samples[-1]
233.
234.                cost = T.mean(self.free_energy(self.input)) - T.mean(
235.                    self.free_energy(chain_end))
236.                # We must not compute the gradient through the gibbs sampling
237.                gparams = T.grad(cost, self.params, consider_constant=[chain_end])
238.                # end-snippet-3 start-snippet-4
239.                # constructs the update dictionary
240.                for gparam, param in zip(gparams, self.params):
241.                    # make sure that the learning rate is of the right dtype
242.                    updates[param] = param - gparam * T.cast(
243.                        lr,
```

```
                        dtype=theano.config.floatX
                    )
                if persistent:
                    # Note that this works only if persistent is a shared variable
                    updates[persistent] = nh_samples[-1]
                    # pseudo-likelihood is a better proxy for PCD
                    monitoring_cost = self.get_pseudo_likelihood_cost(updates)
                else:
                    # reconstruction cross-entropy is a better proxy for CD
                    monitoring_cost = self.get_reconstruction_cost(updates,
                                                                   pre_sigmoid_nvs[-
    1])

                return monitoring_cost, updates
                # end-snippet-4

            def get_pseudo_likelihood_cost(self, updates):
                """Stochastic approximation to the pseudo-likelihood"""

                # index of bit i in expression p(x_i | x_{\i})
                bit_i_idx = theano.shared(value=0, name='bit_i_idx')

                # binarize the input image by rounding to nearest integer
                xi = T.round(self.input)

                # calculate free energy for the given bit configuration
                fe_xi = self.free_energy(xi)

                # flip bit x_i of matrix xi and preserve all other bits x_{\i}
                # Equivalent to xi[:,bit_i_idx] = 1-xi[:, bit_i_idx], but assigns
                # the result to xi_flip, instead of working in place on xi.
                xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])

                # calculate free energy with bit flipped
                fe_xi_flip = self.free_energy(xi_flip)

                # equivalent to e^(-FE(x_i)) / (e^(-FE(x_i)) + e^(-FE(x_{\i})))
                cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip -
                                                                    fe_xi)))

                # increment bit_i_idx % number as part of updates
                updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

                return cost

            def get_reconstruction_cost(self, updates, pre_sigmoid_nv):
                """Approximation to the reconstruction error

                Note that this function requires the pre-sigmoid activation as
                input.  To understand why this is so you need to understand a
                bit about how Theano works. Whenever you compile a Theano
                function, the computational graph that you pass as input gets
                optimized for speed and stability.  This is done by changing
                several parts of the subgraphs with others.  One such
                optimization expresses terms of the form log(sigmoid(x)) in
                terms of softplus.  We need this optimization for the
                cross-entropy since sigmoid of numbers larger than 30. (or
                even less then that) turn to 1. and numbers smaller than
                -30. turn to 0 which in terms will force theano to compute
                log(0) and therefore we will get either -inf or NaN as
                cost. If the value is expressed in terms of softplus we do not
                get this undesirable behaviour. This optimization usually
                works fine, but here we have a special case. The sigmoid is
                applied inside the scan op, while the log is
                outside. Therefore Theano will only see log(scan(..)) instead
                of log(sigmoid(..)) and will not apply the wanted
```

```
309.                optimization. We can not go and replace the sigmoid in scan
310.                with something else also, because this only needs to be done
311.                on the last step. Therefore the easiest and more efficient way
312.                is to get also the pre-sigmoid activation as an output of
313.                scan, and apply both the log and sigmoid outside scan such
314.                that Theano can catch and optimize the expression.
315.
316.                """
317.
318.                cross_entropy = T.mean(T.sum(self.input * T.log(T.nnet.sigmoid(pre_s
    igmoid_nv)) +
319.                                             (1 - self.input) * T.log(1 - T.nnet.sig
    moid(pre_sigmoid_nv)),
320.                                             axis = 1))
321.
322.                return cross_entropy
323.
324.
325.        def test_rbm(dataset = 'mnist.pkl.gz', learning_rate = 0.1, batch_size = 20,

326.                     output_folder='rbm_plots', n_hidden = 500, n_visible = 784,
327.                     learningRateBoostFactor = 1.5, deltaCostStopThreshold = 0.001,
    k = 15):
328.            """
329.            Demonstrate how to train and afterwards sample from it using Theano.
330.            This is demonstrated on MNIST.
331.            :param learning_rate: learning rate used for training the RBM
332.            :param training_epochs: number of epochs used for training
333.            :param dataset: path the the pickled dataset
334.            :param batch_size: size of a batch used to train the RBM
335.            """
336.
337.            if type(dataset) == str:
338.                datasets = load_data(dataset)
339.            else:
340.                datasets = dataset
341.
342.            train_set_x, train_set_y = datasets[0]
343.            test_set_x, test_set_y = datasets[2]
344.
345.            origLearningRate = learning_rate
346.
347.
348.            # compute number of minibatches for training, validation and testing
349.            n_train_batches = train_set_x.get_value(borrow = True).shape[0] / batch_
    size
350.
351.            # allocate symbolic variables for the data
352.            index = T.lscalar() # index to a [mini]batch
353.            x = T.matrix('x')  # the data is presented as rasterized images
354.
355.            rng = numpy.random.RandomState(123)
356.            theano_rng = RandomStreams(rng.randint(2 ** 30))
357.
358.            # initialize storage for the persistent chain (state = hidden
359.            # layer of chain)
360.            persistent_chain = theano.shared(numpy.zeros((batch_size, n_hidden),
361.                                             dtype = theano.config.floatX),
362.                                             borrow = True)
363.
364.            # construct the RBM class
365.            rbm = RBM(input = x, n_visible = n_visible, n_hidden = n_hidden,
366.                      numpy_rng = rng, theano_rng = theano_rng)
367.
368.            # get the cost and the gradient corresponding to one step of CD-15
369.            cost, updates = rbm.get_cost_updates(lr = learning_rate,
```

```
370.                                                 persistent = persistent_chain, k =
    k)
371.
372.             ###################################
373.             #    Training the RBM         #
374.             ###################################
375.             if not os.path.isdir(output_folder):
376.                 os.makedirs(output_folder)
377.             os.chdir(output_folder)
378.
379.             # it is ok for a theano function to have no output
380.             # the purpose of train_rbm is solely to update the RBM parameters
381.             train_rbm = theano.function([index], cost, updates = updates,
382.                                         givens = {x: train_set_x[index * batch_size:
     (index + 1) * batch_size]},
383.                                         name = 'train_rbm')
384.
385.
386.             start_time = time.clock()
387.
388.             # Go through training epochs until stopping criterion is met
389.             deltaCost = -1
390.             epoch = 0
391.             lastEpochCost = 1e6
392.             epochCost = 1e6
393.
394.             while deltaCost < -deltaCostStopThreshold or deltaCost < 0:
395.
396.                 weights = rbm.W.get_value(borrow = True).T
397.                 biases = rbm.hbias.get_value(borrow = True)
398.
399.                 # go through the training set
400.                 mean_cost = []
401.                 for batch_index in xrange(n_train_batches):
402.                     mean_cost += [train_rbm(batch_index)]
403.
404.                 epochCost = numpy.mean(mean_cost)
405.                 print 'Training epoch %d, cost is ' % epoch, epochCost
406.                 deltaCost = epochCost - lastEpochCost
407.
408.                 epoch += 1
409.
410.                 # Check stopping criteria and update learning rate
411.                 if abs(deltaCost) < deltaCostStopThreshold:
412.                     learning_rate *= learningRateBoostFactor
413.                     print 'increasing learning rate to %0.2f...' % learning_rate
414.                     cost, updates = rbm.get_cost_updates(lr = learning_rate,
415.                                                          persistent = persistent_cha
    in, k = k)
416.
417.             # Save final weights and biases
418.             outputFnRoot = 'k_%s_nin_%s_nhdn_%s_basz_%s_lnrt_%s' \
419.                            % (str(k), str(n_visible), str(n_hidden), str(batch_size), s
    tr(origLearningRate))
420.             numpy.savetxt('weights_' + outputFnRoot + '.csv', weights, delimiter = '
    ,')
421.             numpy.savetxt('biases_' + outputFnRoot + '.csv', biases, delimiter = ','
    )
422.
423.             # calculate time taken for training
424.             end_time = time.clock()
425.             training_time = (end_time - start_time)
426.
427.             # save a list of the training parameters
428.             if not os.path.exists('training_records.pkl'):
429.                 trainingRecords = []
```

```
430.            else:
431.                trainingRecords = cPickle.load(open('training_records.pkl', 'rb'))
432.            record = {'# Features': n_visible,
433.                      '# Hidden Units': n_hidden,
434.                      '# Epochs': epoch,
435.                      'Batch Size': batch_size,
436.                      'Initial Learning Rate': origLearningRate,
437.                      'Final Learning Rate': learning_rate,
438.                      'Delta Cost Stopping Threshold': deltaCostStopThreshold,
439.                      'Learning Rate Boost Factor' : learningRateBoostFactor,
440.                      'Final Cost': lastEpochCost,
441.                      'k': k}
442.
443.            trainingRecords.append(record)
444.            cPickle.dump(trainingRecords, open('training_records.pkl', 'wb'))
445.
446.            # Print time taken for training
447.            print ('Training took %f minutes' % (training_time / 60.))
448.
449.            os.chdir('../')
```