Deep Neural Architectures for Algorithms and Sequential Data

PhD dissertation summary

Karol Kurach

June 2016

1 Introduction

My PhD thesis consists of 7 publications listed below:

1. Neural Random-Access Machines, accepted to ICLR 2016 (Conference Track) [1]
3. Adding Gradient Noise Improves Learning for Very Deep Networks, accepted to ICLR 2016 (Workshop Track) [3]
4. Learning to Discover Efficient Mathematical Identities, accepted to NIPS 2014 (as spotlight) [4]
5. Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks, accepted to IJCRS 2015 [5]
6. Detecting Dangerous Seismic Events with Recurrent Neural Networks, accepted to AAIA 2016 [6]
7. Smart Reply: Automated Response Suggestion for Email, accepted to KDD 2016 (Research Track) with presentation at the plenary session [7]

For statistics and acceptance rates please see Appendix A.
The first three publications are related to deep neural architectures with external memories. The next four publications describe novel systems based on recursive and recurrent neural networks (both with “vanilla” and Long Short-Term Memory cells) in domains of sequential data and finding efficient algorithms.

A brief introduction to Deep Learning is given in Section 2. Feedforward neural networks are described in Section 3. A generalization of feedforward networks to recurrent neural networks is presented in Section 4. Section 5 describes recently introduced attention mechanism. Finally, Section 6 contains the results of this thesis.

**Notation**

- $x$: A vector or scalar
- $x_t$: A vector or scalar at timestep $t$ (for recurrent neural networks)
- $x^i$: The $i$-th element from a set
- $X$: A matrix
- $x \cdot y$: A scalar product of two vectors
- $x \odot y$: An element-wise multiplication of two vectors
- $x \oplus y$: An element-wise sum of two vectors
- $\theta$: A vector of neural network’s trainable parameters (“weights”)
- $a(t)$: An activation function
- $f \circ g$: A composition of functions $f$ and $g$
- $\mathbb{Z}_M$: A ring of integers modulo $M$
- $\text{sigm}(\beta, t)$: The sigmoid function $\frac{1}{1+e^{-\beta t}}$
- $\text{sigm}(t)$: The sigmoid function with $\beta = 1$
- $\text{tanh}(\beta, t)$: The tanh function $\frac{e^{\beta t} - e^{-\beta t}}{e^{\beta t} + e^{-\beta t}}$
- $\text{tanh}(t)$: The tanh function with $\beta = 1$
2 Deep Learning

Machine Learning is an area of Artificial Intelligence focused on designing systems that can learn from data. It is widely applied to a variety of problems, especially those too complex for a human software engineer to define in terms of a fixed piece of software. Examples span wide range of topics, including computer vision, signals recognition and text understanding.

Deep learning is a branch of machine learning that concentrates on finding hierarchical representation of data, starting from low-level observations towards high-level abstractions. Deep architectures are usually composed of multiple layers of non-linear operations, such as neural networks with many hidden layers.

Although neural networks are not new, they recently got a lot of traction in the researchers’ community. This is mostly because of three factors: progress in understanding training, availability of big datasets and significant increase in computing power. The first one is related to the fact that in the late 90’s it was believed that training networks with more than few layers is a very complex and practically impossible task. Recent work proved that, with correct methods for initialization, careful selection of learning rates and a pre-training, even networks with many layers can be successfully trained.

The other crucial component was progress in hardware, especially the introduction of graphics processing units (GPUs). Being able to train few orders of magnitude faster and on much larger datasets allowed to create and train complex models, with millions of parameters. Deep neural networks already proved to be very successful in representing complicated functions. In some domains, like image or speech recognition, they overperformed previous state of the art methods by a wide margin.

3 Feedforward Neural Networks

A feedforward neural network (FNN) is a basic example of an artificial neural network. In essence, the goal of such network in supervised setting is to approximate some function \( f(x) \), given only a noisy set of evaluations in selected points \( \{(x^i, f(x^i))\} \). The FNN can be interpreted as a function \( f^*(x; \theta) \), where \( \theta \) is a set of trainable parameters (“weights”) of the network. The parameters \( \theta \) are trained from the data to minimize selected cost (e.g., MSE, logistic loss) between \( f \) and \( f^* \).
The simplest form of the feedforward network is a linear classifier, known also as perceptron. In this case:

$$f^*(x; \theta) = a(x \cdot \theta + b)$$

(1)

where $b$ is the bias term and $a(u) = \begin{cases} 1 & u > 0 \end{cases}$. In a more general scenario, $f^*(x; \theta)$ is a composition of $N$ functions $H^1, \ldots, H^N$. That is:

$$f^*(x) = H^N \circ H^{N-1} \circ \ldots \circ H^1(x)$$

(2)

where $H^N$ computes the final output. Intuitively, each function $H^i$ represents one transformation or layer of the neural network. The total number of layers $N$ is the depth of the neural network. This is why FNNs are also referred to as multi-layer perceptrons (MLP). The FNN is feedforward because connections between nodes of the network do not form cycles, i.e. it can be represented as Direct Acyclic Graph (DAG).

An example feedforward network composed of $N = 3$ layers is presented in Figure 1. The network consists of two hidden layers and an output layer. The parameters $\theta$ are the weights on the edges. Let us denote by $\theta^l_{uv}$ the weight on the edge going from the node $u$ in the layer $l$ to the node $v$ in the layer $l+1$. Then, each node’s value $h_{ij}^{l+1}$ can be computed using:

$$h_{ij}^{l+1} = a^{l+1}(\sum_i h_i^l \cdot \theta_{ij}^l + b_j^{l+1})$$

(3)

The term $a^{l+1}$ is known as activation function. It is applied to introduce a non-linearity to the network computation. This is a crucial component –
if all of the hidden layers were linear transformations, the composition of $N$ hidden layers would be a linear transformation as well. A very deep network would be then an equivalent of a simple perceptron. Popular examples of activation functions are sigmoid, tanh and ReLu [8]. The $b_{j}^{l+1}$ is the bias term for the $j$th node in the layer $l + 1$, which is also a trainable parameter of the model. Usually, it is an edge from a special node $h_{0}^{l} = 1$ in the layer $l$, so that $b_{j}^{l+1} = \theta_{0j}$.

The estimation of the parameters $\theta$ is frequently done using the backpropagation learning technique. For a given example pair $(x, y)$ first the value $y' = f^*(x; \theta)$ is computed. This is used to calculate the $C = \text{loss}(y', y)$ value, where $C$ is the “cost” we want to minimize and $\text{loss}$ is a function such as Mean Squared Error. Then, the backpropagation is used to compute derivatives of the network parameters $\theta$ w.r.t. the cost. With derivatives computed, one can use a standard optimization technique such as Stochastic Gradient Descent or Adagrad [9] to adjust weights. After a sufficient number of steps the network should usually converge to the parameters that achieve low error score on a given training set.

4 Recurrent Neural Networks

The important limitation of feedforward networks is the ability to process only inputs of a fixed size. However, many real-world problems are sequential in nature – for instance, we may want to classify e-mails which can have different number of words, translate sentences or predict stock price based on the list of historical values. Moreover, for the FNN all input examples are independent from each other. This means that, after processing one data point, the network forgets everything from the previous step and starts from scratch. This is clearly very different from the way people process data, relying on the context of elements seen so far.

A recurrent neural network (RNN) is a type of neural network that can overcome those limitations. In RNN, the dependencies between nodes can form a cycle, which allows the network to preserve the state between subsequent timesteps. In the most general form, the RNN is a function that, in timestep $t$, takes the current input ($x_t$), the state from the previous timestep ($s_{t-1}$) and produces the new state:

$$s_t = f^*(x_t, s_{t-1}; \theta)$$  (4)

RNNs process all elements from a sequence one-by-one, and the output at every timestep depends on all previous inputs. This fact has an important theoretical implication: RNNs are capable of approximating arbitrarily well
any measurable sequence-to-sequence mapping [10]. The state $s_t$ can vary depending on the specific RNN – for example, in the vanilla RNN the state consists of one hidden state vector ($h$), while in, for example, LSTM the state is represented by two vectors: $h$ and memory cell vector $c$. Below those two RNN variants are described in more details, as both are building blocks of results obtained in this thesis.

4.1 Vanilla RNN

A basic version of the recurrent network is the vanilla RNN, which is presented in Figure 2. In this case, the Equation 4 can take the following form:

$$h_t = \tanh(U \ast x_t + W \ast h_{t-1})$$

$$o_t = \text{output}(V \ast h_t)$$

where $U$, $V$, $W$ are matrices and $\text{output}$ is a function that produces final values at timestep $t$ from the hidden state (i.e. softmax). The tanh function is frequently used as activation for vanilla RNN, but can be also replaced by any other non-linear function.

Figure 2 presents also the idea of unrolling. Since RNNs contain loops, we cannot directly apply the standard backpropagation algorithm to compute gradients. Instead, we replicate the network $N$ times and change the recurrent connections. Every edge $(u, v)$ is converted into a new edge connecting node $u$ in layer $k$ with node $v$ in layer $k + 1$. Note that the matrices $U$, $V$, $W$ are the same for each timestep. This transforms the RNN into a standard feedforward network, where we can apply backpropagation, and average gradients from all timesteps. This method is called backpropagation through time (BPTT) [11] and it is commonly used to train RNNs.
However, there are some important problems that arise when applying BPTT. Notice that the unrolled network can be extremely deep. In this case, the gradients of the activation functions will be multiplied at least $N$ times. For some activation functions, the maximal value of the derivative is small. For example, the derivative of commonly used sigmoid function is never bigger than 0.25. As a result, after $N$ timesteps the gradient is multiplied by a value less than or equal to $0.25^N$. This leads to the problem known as vanishing gradient, as the gradient signal quickly becomes too weak to learn any dependency longer than few timesteps. Similarly, the exploding gradient can occur when derivatives bigger than 1.0 are multiplied many times.

Those two problems historically made people believe that RNNs are too hard to train. To address them, various techniques were proposed, such as gradient clipping (for exploding gradient), and architectures, such as Long Short-Term Memory (for vanishing gradient).

4.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM) is a neural architecture designed for storing and accessing information better than standard RNN [12]. The LSTM block consists of a self-connected memory cell and three gates named: input, output and forget. The gates control the access to the cell and can be interpreted as “read”, “write” and “reset” operations in the standard computer’s memory. The network learns to control the gates and decides to update and/or use the value at any given timestep. Since all of the components are built from differentiable functions, the gradients can be computed for the whole system and it is possible to train it end-to-end using backpropagation. There are several variants of LSTM that slightly differ in connectivity structure and activation functions. Below are described the definitions of input, output and forget gates used in this work.

Let $h_t \in \mathbb{R}^n$ be a hidden state, $c_t \in \mathbb{R}^n$ be a vector of memory cells of the network and let $x_t \in \mathbb{R}^n$ be the input at the timestep $t$. Let $W_i, W_f, W_u, W_o$ be matrices and $b_i, b_f, b_u, b_o$ the respective bias terms. We define LSTM as a transformation that takes 3 inputs ($h_{t-1}, c_{t-1}, x_t$) and produces 2 outputs ($h_t$ and $c_t$). In all equations below $\odot$ is element-wise multiplication and $\oplus$ is element-wise addition.
The connections between all gates are presented in Figure 3. The forget gate which decides how much of the information should be removed from the cell is defined as:

\[ f_t = \text{sigm}(W_f \ast [h_{t-1} \oplus x_t] + b_f) \]  

(6)

The input modulation gate value \( i_t \) and the cell update \( u_t \) are defined as:

\[ i_t = \text{sigm}(W_i \ast [h_{t-1} \oplus x_t] + b_i) \]

\[ u_t = \tanh(W_u \ast [h_{t-1} \oplus x_t] + b_u) \]  

(7)

Intuitively, input modulation decides how much of the \( u_t \) should be added to the memory at step \( t \). For example, if \( x_t \) can be ignored, \( i_t \) will be close to 0. Knowing the values above, the new cell value \( c_t \) is computed as:

\[ c_t = f_t \odot c_{t-1} + i_t \odot u_t \]  

(8)

The last step is to compute \( h_t \), the output passed to the next LSTM’s timestep. It is controlled by the output gate \( o_t \):

\[ o_t = \text{sigm}(W_o \ast [h_{t-1} \oplus x_t] + b_o) \]

\[ h_t = o_t \odot \tanh(c_t) \]  

(9)

The LSTM networks have been successfully applied to real-world problems, including language modeling [14], handwriting [15], speech recognition [16] and machine translation [17].
Figure 4: Encoder-decoder scheme for Neural Machine Translation. The encoder is unrolled for $N$ timesteps (marked in red) and the decoder is unrolled for $M$ timesteps (marked in blue).

5 Recurrent Networks with Attention

An important limitation of RNNs described in Section 4 is the size of their internal state. It was observed that, the bigger the size of the LSTM memory, the better the language modelling results [13]. It was also noticed that some heuristics, like feeding the input twice or providing the network with both a sequence and a reversed sequence, improve the performance [17]. To understand why those techniques help, let us take a look at Figure 4 which presents the standard encoder-decoder RNN setup used for Machine Translation.

The goal of the presented model is to translate sentences. For example, the input could be a sentence in Polish (with $x_1, \ldots, x_N$ being words in Polish) and we want to produce a translation into English (with $o_1, \ldots, o_M$ being words in English). There is a special symbol for missing word, so this model can translate any sentence consisting of up to $N$ words into any sentence consisting of up to $M$ words.

The first, lower part of the model is the encoder, which reads an input sequence word by word and update its internal state $h^E$. After processing the whole sequence, it passes its final state $h^E_N$ to the decoder. The decoder’s tasks is to predict the translation conditioned only on the input from the encoder.

Notice, that in this setup the encoded state $h^E_N$ has to preserve all information about the initial sentence. This “compression” into a fixed-length vector...
results in a dropping some information, and explains why giving the network both input and the reversed input can sometimes improve the performance.

One obvious solution to this problem would be to just use a bigger memory. However, this can significantly increase the number of model’s parameters. For example, in LSTM the number of network’s parameters grows quadratically with the memory size, making it harder to train and more prone to overfitting.

Recently, there has been a significant interest in creating neural network architectures that avoid the problem of memorization by employing the so-called attention mechanism. Such networks can attend to parts of the (potentially preprocessed) input sequence [18] while generating the output sequence. It is implemented by giving the network as an auxiliary input a linear combination of input symbols, where the weights of this linear combination can be controlled by the network. This approach has already proven to be very successful in areas of machine translation [18], speech recognition [19] and syntactic parsing [20].

5.1 Learning algorithms

One of the important applications of attention mechanism was the highly influential paper Neural Turing Machines (NTMs) [21], where the authors presented a versatile neural network architecture capable of learning simple algorithms from pure input-output examples. The main idea behind this type of neural networks is to let the network operate on an “external” memory, which size is independent of the number of the model parameters.

The NTM paper caused an outbreak of other neural network architectures with a goal of learning algorithms and operating on external memory. They usually fall into one of the categories:

- **Memory architectures based on attention**, such as Memory Networks [22] or Pointer Networks [23]

- **Memory architectures based on data structures**, such as Stack-Augmented RNN [24] or LSTM extended with a stack, a FIFO queue or double-ended queue [25]

- **Parallel memory architectures**, such as Grid-LSTM [26] or Neural GPU [27]
6 Main results

6.1 Neural Random-Access Machines

In the paper Neural Random-Access Machines [1] we propose a neural architecture that has, as primitive operations, the ability to manipulate, store in memory, and dereference pointers into its working memory. The Neural Random-Access Machine (NRAM) is an computationally-universal model employing an external memory, which size does not depend on the number of model’s parameters.

By providing our model with dereferencing as a primitive, it becomes possible to train it on problems whose solutions require pointer manipulation and chasing. We were able to train the proposed model from pure input/output pairs using the standard backpropagation algorithm. It has learned to solve algorithmic tasks and is capable of learning the concept of data structures that require pointers, like linked-lists and binary trees. For a subset of tasks we show that the found solution can generalize to sequences of arbitrary length. Moreover, memory access during inference can be done in a constant time under some assumptions. The most important contributions of the NRAM architecture include:

- A mechanism for location-based addressing using fuzzy pointers (probability distributions over $\mathbb{Z}_M$).
- A way of combining neural network with pre-defined modules.
- A differentiable mechanism for deciding when to terminate the computation.
- Novel training techniques, such as entropy term or way of enforcing distribution constrains.

6.1.1 Noise addition

The NRAM model can be very deep (up to hundreds of layers after unrolling) which makes the training very challenging. One of the important training techniques that we employ is the addition of gaussian noise to gradients during backpropagation. This technique was proposed in our paper Adding Gradient Noise Improves Learning for Very Deep Networks [3], in which we evaluate it on a number of modern neural architectures, including Neural Programmer [28], End-To-End Memory Networks [29] and Neural GPU [27]. We show that the noise addition can significantly improve the training results as well as stability of the re-training.
6.2 Hierarchical Attentive Memory

One of the limitations of the standard attention mechanism (see Section 5) is the fact, that the memory access complexity is $O(n)$. This can make it impractical for real-world problems, when the size of the input can be large (e.g., attention over books or long DNA sequences). To address this problem, in the paper Learning Efficient Algorithms with Hierarchical Attentive Memory [2] we propose a novel memory module for neural networks, called Hierarchical Attentive Memory (HAM).

The HAM module is generic and can be used as a building block of larger neural architectures. Its crucial property is that it scales well with the memory size — the memory access requires only $\Theta(\log n)$ operations, where $n$ is the size of the memory. This complexity is achieved by using a new attention mechanism based on a binary tree with leaves corresponding to memory cells. The novel attention mechanism is not only faster than the standard one commonly used in Deep Learning [18], but it also facilitates learning algorithms due to a built-in bias towards operating on intervals.

We show that an LSTM augmented with HAM is able to learn algorithms for tasks like merging, sorting or binary searching. In particular, it is the first neural network, that is able to learn sorting algorithm from pure input-output examples and generalizes well to input sequences much longer than the ones seen during the training. Moreover, the learned sorting algorithm runs in time $\Theta(n \log n)$. We also show that the HAM module itself can act as a drop-in replacement for classic data structures, like a stack, a FIFO queue or a priority queue.

6.3 RNN for efficient algorithms

The running time of a computer program can critically depend on the algorithms used. For instance, one could solve a problem of sorting by using a very natural algorithm known as selection sort, with a complexity of $O(n^2)$. However, people discovered an alternative algorithm that can produce exactly the same output with a complexity of $O(n \log n)$. It would be desirable to have a system that, for given algorithm, can find the fastest drop-in replacement. That is, an algorithm providing exactly the same final result, but with a low computational complexity.

In our work we restrict our domain of algorithms to mathematical expressions over matrices. The motivational example is presented in Figure 5. Assume we are given matrices $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times p}$. We wish to compute the sum of all elements of the matrix $A \ast B$, i.e. $\sum_{n,p} AB = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{p} A_{i,j} B_{j,k}$ which naively takes $O(nmp)$ time. This formula
could be expressed in matlab notation as $\text{sum}(\text{sum}(A*B))$. However, there exists an efficient version of the formula, that computes the same result in $O(n(m+p))$ time by avoiding matrix-matrix multiplication. It can be written in matlab as $\text{sum}((\text{sum}(A, 1) * B)^T, 1)$.

In the paper Learning to Discover Efficient Mathematical Identities [4] we introduce a framework based on attribute grammars for finding computationally efficient versions of symbolic math expressions. We show how machine learning techniques can be integrated into this framework, and demonstrate how training models on simpler expressions can help with the discovery of more complex ones. In particular, we present a novel application of a recursive neural network[30] to learn a continuous representation of mathematical structures.

We show how the exploration of the search space can be learned from previously successful solutions to simpler expressions. This allows us to discover complex expressions that random or brute-force strategies cannot find. We present examples of (i) $O(n^3)$ target expressions which can be computed in $O(n^2)$ time and (ii) cases where naive evaluation of the target would require exponential time, but can be computed in $O(n^2)$ or $O(n^3)$ time. The majority of these examples are too complex to be found manually or by exhaustive search and, as far as we are aware, are previously undiscovered.
6.4 RNN for monitoring systems

Monitoring systems that can predict dangerous events play a key role in ensuring people’s safety. Recent data mining competitions: IJCRS’15 and AAIA’16 presented problems related to ensuring the safety of the underground coal mining workers. The IJCRS’15 competition was concerned with predicting dangerous levels of methane concentration, that can lead to explosion. The task of the AAIA’16 competition was related to predicting high-energy seismic events. Both problems are similar in nature: they were an instance of a classification problem with unbalanced data provided in a form of multivariate, non-stationary time series. The data was collected from Polish coal mines using various sensors.

In our paper Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks [5] we propose a method based on an RNN and an FNN for predicting the probability of dangerous methane concentration. The method is further improved in our next paper, Predicting Dangerous Seismic Activity with Recurrent Neural Networks [6], where we rely solely on an RNN.

The important aspect of our solution is the fact, that it learns to predict from raw sensor values, with a very minimal preprocessing. Most of the other top solutions relied heavily on feature engineering, either manual or automatic, such as: automatic variable construction [31], window-based feature engineering [32], hand-crafted features [33] or thousands of automatically generated features [34]. Such approach, while effective in the competition, is less likely to generalize to a different setting, as well as more complex to reproduce and deploy in a real production system.

Our method achieved a competitive score and placed 6th (out of 90) in the IJCRS’15 and 5th (out of 203 teams) in the AAIA’16 competition. Top performance in both competitions suggests that our approach is versatile and can be successfully applied to different multivariate time series problems.

6.5 RNN for email responses

Email is the primary medium of communication for billions of users across the world to connect and share information [35]. In our paper Smart Reply: Automated Response Suggestion for Email [7] we propose and implement a novel end-to-end system for generating short email responses. The system was implemented in Google Inbox and is currently responsible for generating 10% of responses on mobile. An example usage of Smart Reply is presented in Figure 6. A user is presented with 3 short response options available to use with just one tap.
The Smart Reply system is extremely high-throughput oriented and can handle the processing of hundreds of millions of messages daily. The system is based on sequence-to-sequence LSTM trained on a larger scale than before. We also present our solution to challenges that we faced while developing such system, including:

- **Quality** How to ensure that our system produces good quality and diverse set of responses.

- **Scalability** How to process hundreds of millions of messages and stay within latency requirements for an email system.

- **Privacy** How to develop such system without ever inspecting the data.
Acknowledgements

First of all, I would like to thank my supervisor, Hung Son Nguyen for introducing me to the area of Machine Learning and convincing me to focus on it during my PhD studies.

I would also like to express my gratitude to Krzysztof Diks and Joanna Śmigielska for being inspirational algorithms teachers at the beginning of my programming career. The Machine Learning and algorithmic topics had a huge impact on the direction of this thesis.

Special thanks go to my co-authors: Greg Corrado, Rob Fergus, Marina Ganea, Andrzej Janusz, Łukasz Kaiser, Anjuli Kannan, Tobias Kaufmann, Quoc V. Le, László Lukács, James Martens, Tomasz Michalak, Balint Miklos, Arvind Neelakantan, Talal Rahwan, Vivek Ramavajjala, Sujith Ravi, Łukasz Romaszko, Ilya Sutskever, Marcin Tatjewski, Andrew Tomkins, Luke Vilnis, Peter Young and in particular to Marcin Andrychowicz, Krzysztof Pawłowski and Wojciech Zaremba with whom I have worked the most.

Last but not least, I would like to thank my amazing family for the support and constant encouragement. In particular, I would like to thank my wife Anna, brother Kamil, parents Agnieszka & Krzysztof as well as all relatives and friends who have been constantly asking me when I finish this thesis.
Appendix A  Acceptance rates

Below are detailed statistics (if available) for the relevant conferences.

1. **ICLR 2016 Conference Track**: acceptance rate is 24.5% (65 out of 265)

2. **NIPS 2014 spotlight**: acceptance rate for spotlight is 3.7% (62 out of 1678)

3. **IJCRS 2015**: for the “Competition Track” the papers from the top 6 (out of 90) teams were accepted (6.7%)

4. **AAIA 2016**: for the “Competition Track” the papers from the top 8 (out of 203) teams were accepted (3.9%)

5. **KDD 2016 Research Track**: acceptance rate for presentation is 9.2% (72 out of 784)

Appendix B  Other publications

During my PhD studies I also published the following two papers, which I have decided not to include in this dissertation, because they are not related to neural architectures.

1. **Coalition structure generation with the graphics processing unit**, accepted to AAMAS 2014 [36]

2. **An Ensemble Approach to Multi-label Classification of Textual Data**, accepted to ADMA 2012 [37]
References


