SEQUENCE MODELING FOR SMARTPHONE APPLICATION LAUNCH PREDICTION

Master Thesis
présenté le 12 Mars 2015
École de Chimie Physique et Électronique de Lyon

pour l'obtention du grade de Master
par

Mattéo Pagliardini

Lyon, CPE, 2015

Contents

List of Figures

Introduction

Today we have access to hundred of thousands of smartphone applications. The way these applications are interfaced with the user is still nowadays very basic. Navigating through them can be burdensome, and the way these applications use the shared resources is often far from optimal. Following the progression of hardware performances, the applications are of increasing complexity, requiring more memory and power. This is accentuated by what seems to be the future trend of smartphone introducing 3D vision capabilities [2] and opening wide doors to augmented reality and computer vision applications. Future applications might take a lot of time to load, others aiming to synchronize some content might waste a lot of resources querying the network when the user is not really in need. Another strong trend seems to be connected items such as google glass [1], those items are most of the time lacking a simple user interface. Modeling the user behavior would benefit both the user experience and the smartphone capabilities, allowing for smarter ways to use and present smartphone applications. One can imagine an easy way to access the applications, presenting the most probable applications on the desktop, or recommending the right application at the right time for wearable devices. The applications could be preloaded so the user would not experience any loading time and the smartphone would appear smarter. A news recommender might fetch new data only at appropriate time hence saving resources.
In this thesis, we tried to predict smartphone applications using several algorithms. In order to test our algorithms we used a dataset which stores, for different users, their smartphone behavior. Among the methods investigated are latent dirichlet allocations, latent semantic indexing, conditional random fields and recurrent neural networks. All those methods are compared with simple algorithms that we define as our baselines. As outcome of these experiments, we demonstrated the strong temporal coherency of our data, and how it was possible to take advantage of it in order to generate accurate predictions.

1 Dataset and conventions

1.1 Dataset

The dataset used consists in the logs of 9 users. For each of them the use of their smartphone has been monitored and the smartphone applications used as well as their associated time and metadata have been recorded. Only the application launches are monitored, one application launch occurs when a user start an application on his smartphone. Among the metadata figures the GPS coordinates, the WIFIs in reach and much more. In our study we decided to omit information other than time, and focused on predicting application launches from that feature alone. figure 1.1 show the content of the dataset for several users.

We can see how different the behavior of two users can be. Some of them have obvious time features e.g. the use of an alarm in the morning. Some use many applications while others just a few. Looking more closely at the applications sequences we can see how noisy the data is, it is hard to see patterns. Users can also change of behavior with time. Many causes might influence an individual in his smartphone usage such as the time of the day, the time of the week, his location, his social groups, the global trend, and much more. All these components are tightly interacting, hence the noisy observations.

1.2 Conventions

For all the explanations that will follow, we define:

- w: a word/application/label.
- $s = w_0, w_1, w_2, ..., w_{N-1}$: a sequence of applications/words/labels. N is the number of labels in one sequence/document.
- F_w: the number of occurrences of the word w in the training set.
- L: the lexicon/dictionary of our dataset, contains all the applications used by the current user.

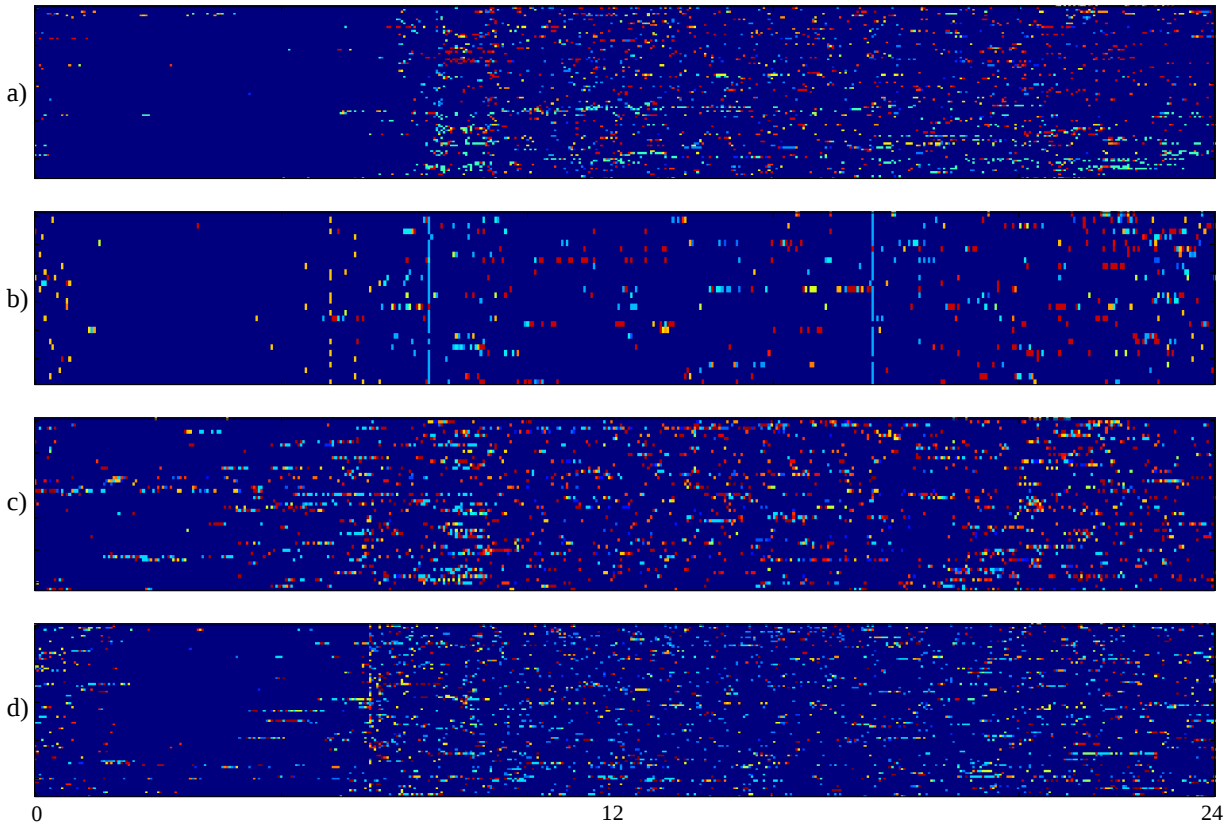

Figure 1.1 – Day sequences for different users. One color correspond to one specific application, each row is one day, each plot correspond to one user. The user *a* shows a different behavior between the beginning and the end of the monitoring. User *b* present some clear time features and uses only few applications. User *c* is very noisy and unpredictable. User *d* has a clearer morning routine but is also quite noisy.

- K: the size of the prediction.
- $s_{a:b}$ represent a slice of the sequence s from a to b included.

2 Preprocessing and evaluation method

2.1 Preprocessing

The raw user logs sequence consists in a long list of smartphone application logs. In order to work on these sequences we decided to split them into smartphone uses. One smartphone use being a sequence of applications used by a user in a short window of time. This choice is based on the intuition that applications used in a short window of time are strongly correlated. If this assumption proves itself to be true this would means sequences are strong descriptors of the local context in which the user is. We aim to use this context to predict the user behavior. Figure 2.1 shows how subsequences are created from the original sequence.

Many methods introduced hereafter require some initial information to generate a recommendation. This information generaly consists in the beginning of the sequences. We hence simplify our problem from predicting all the applications to predicting the three last applications of each sequences. That way, as long as the sequence size is larger than 4 we have at least 1 application to create our models. This requires to keep only the sequences of size larger or equal to 4, which is quite constraining. Indeed figure 2.2 shows that most of the sequences are of size inferior to 4. We will relax this constraint later. Furthermore, we are not yet interested in studying the modifications of users' behavior. To prevent our systems to be affected by them, the position of the sequences is shuffled making the time of the week and day irrelevant. For all the future experiments the default values for the time window is 5 minutes and all sub-sequences containing less than 4 applications are discarded. The value of

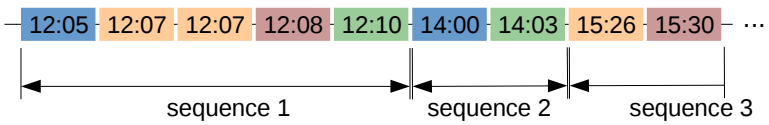

Figure 2.1 – Example of time series for one user. Each two app launches separated by more than 5 minutes belongs to different sequences.

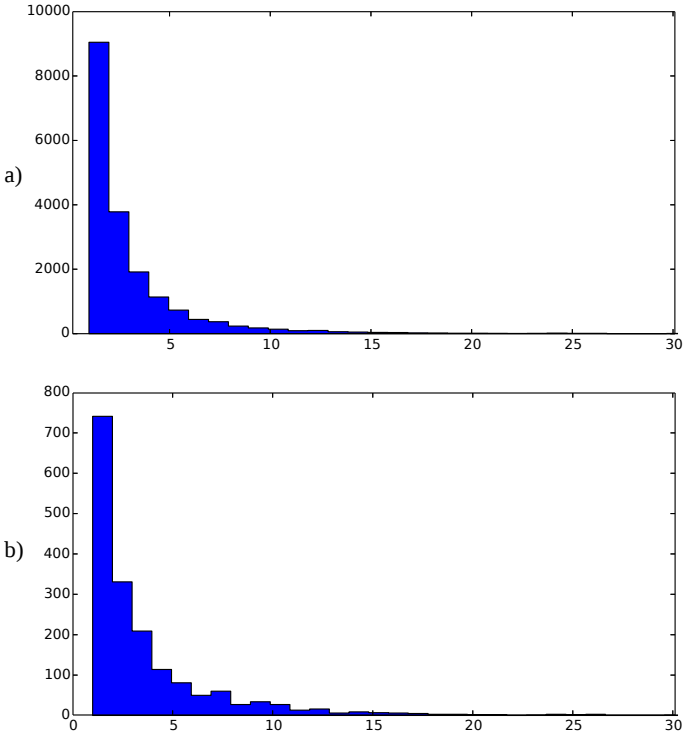

Figure 2.2 – Histograms of the sequences' size for all users (a) as well as for the default user only (b). Most of the sequences have a size inferior to 4.

5 minutes for the time window has been decided after few experiment with different window sizes. The default user studied is user 351680061098293.

2.2 Evaluation method

Predicting exactly the application that will be used is not only difficult for such a noisy dataset but also not that interesting for the applications we are targeting. We will evaluate our methods in terms of accuracy recommending K applications. The pipeline used for the evaluation step is always the same, first the initial dataset is split into a training and a test set. Typically a third of the sequences go in the test set, the remaining two thirds constitute the training set. Then some model is created based from the sequences of the training. The system is evaluated on the test set. The evaluation process consists in removing the 3 last applications of each sequence of the test set and checking if our system is able to predict them based on the created model and the beginning of the sequences. The percentage of errors is the proportion of removed applications that are missing of the predictions. The general evaluation process is described in figure 2.3.

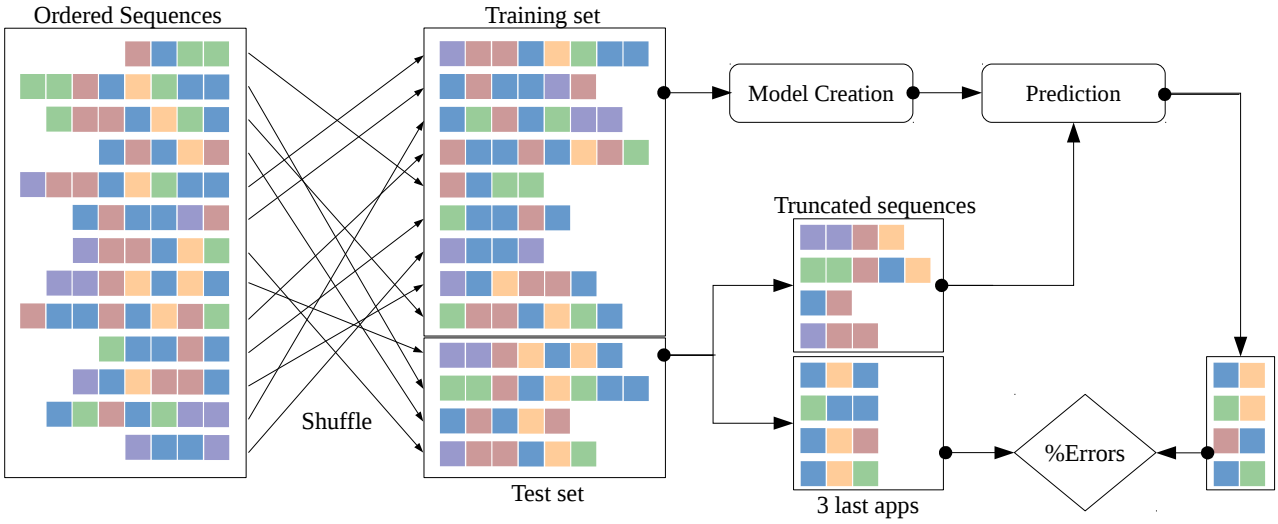

Figure 2.3 – Evaluation process, each colored square represent one application, consecutive applications form sequences. On the left we can see the initial sequences in order. The set of sequences is first shuffled to remove the effect of an eventual behavior modification. In this representation $K = 2$.

We consider two kinds of models, bag of words and sequential. Bag of words models follow the assumption that the order of the words is irrelevant in the sequences. Sequential models take into account the position of the words in the sequence.

3 Baselines

To be able to compare all the developed methods some basic techniques have been tried. They serve as baselines for our study.

3.1 Most Frequently Used Applications (MFU)

This method simply consists in recommending the K most frequently used applications in the training set to the user. Hence for $K = 1$:

$$\forall w \in L, MFU(w) = \arg\max_{w_i}(F_{w_i})$$

MFU is the method using the less prior knowledge possible after the random method. The result for MFU can be seen in figure 3.1.

3.2 Bigrams

Our second baseline is a bigram model. The bigram model used is a smoothed bigram model (smoothed by the unigram model i.e. MFU). For a sequence of applications , the model consists of the following predictive probability distribution []:

$$\forall w \in L, P(w_t|w_{t-1}) = \lambda F_{w_t} + (1-\lambda)\frac{F_{w_t|w_{t-1}}}{F_{w_{t-1}}}$$

Where λ is a coeficient in $[0,1]$. When $\lambda = 0$ the model is a classic bigram model. When $\lambda = 1$ the model is the MFU model. The result of the bigram model can be seen in figure 3.1.

3.3 Most Recently Used (MRU)

The MRU method consist in predicting, for an application w_t, the K last applications used up to w_{t-1}. Hence for $K = 1$ this method will recommend w_{t-1}. This baseline exploits the

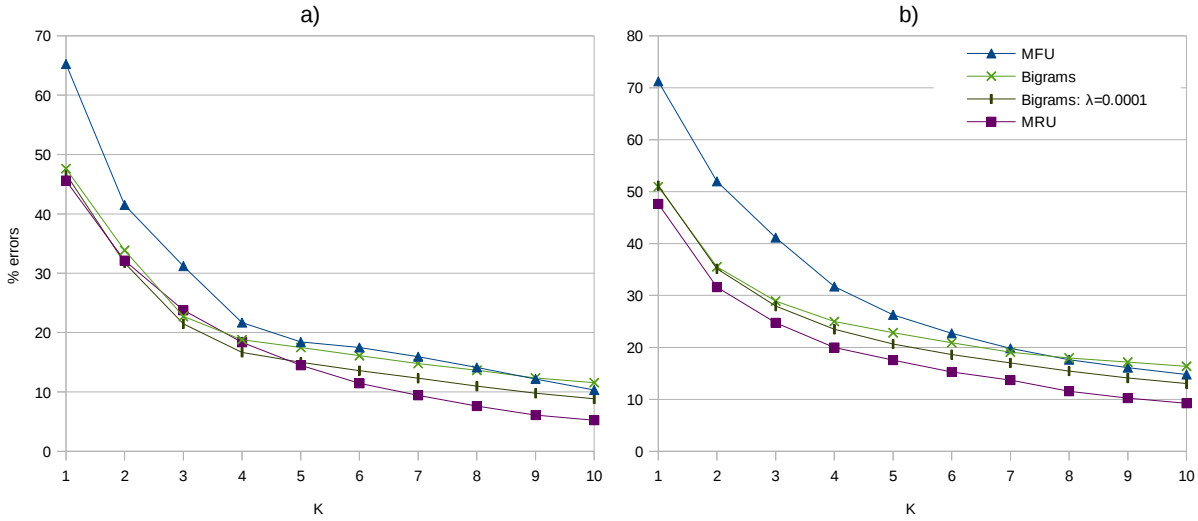

Figure 3.1 – Percentage of errors for baseline methods for the default user (a) and averaged for all users (b). *K* is the size of the prediction.

switching behavior of the users who tend to reuse applications they just used before. The evaluation is done similarly as for the previous baselines. The MRU recommendation results can be seen in figure 3.1.

First of all we can observe how different the results can be for one and all users. The default user was selected for his number of sequences and the size of his lexicon, he represents a challenging user while still giving us enough data to train our models. Other users might have less sequences and smaller lexicon lowering on one side the probability to be wrong while predicting an application and on another side raising the likelihood of overfitting our dataset when using complex models. This explains why MRU is much better than a bigram model when averaging on all the users, for users with small number of application launches, the amount of data is too small to obtain accurate bigrams statistics. The difference between K MFU and K MRU shows the temporal coherency of the data, a user is more likely to use recently used applications.

4 Neighbors based methods

Similar sequences should represent similar a user behavior. Looking at applications contained in close sequences should give an idea of the applications likely to be launched. For this basic approach inspired from collaborative filtering methods [13], we consider all the sequences in the training set as our memory bank. To get a prediction for s we scan the memory bank and compute the distance between s and the sequences in the bank. We considered two distance functions, Jaccard and cosine.

4.1 Recommendation using the Jaccard distance

The Jaccard distance function is a distance function working on sets. For two sets A and B it is expressed as the number of item in common normalized by the total number of items in the two sets:

$$d_{Jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard distance in [0,1], its value is 1 when the two sets are identical.

To generate recommendations for a sequence we transform this sequence into a set $set(s)$. This operation drop the application count information i.e. it is like each application could occur only once in a sequence. We compute a score for each application w summing the Jaccard distances:

$$score_w = \sum_{s' \in bank} d_{Jaccard}(set(s), set(s')).\delta_{w \in s'}$$

Where $\delta_{w \in s'}$ is the impulse function of value one when the condition $w \in s'$ is true else zero. This equation can be understood as each sequence in the bank voting for the applications they contain with a value $d_{Jaccard}(set(s), set(s'))$. Applications belonging to sequences very close to s will get a high score. We recommend the K applications associated to the largest score. The results for this method can be seen on figure 4.1.

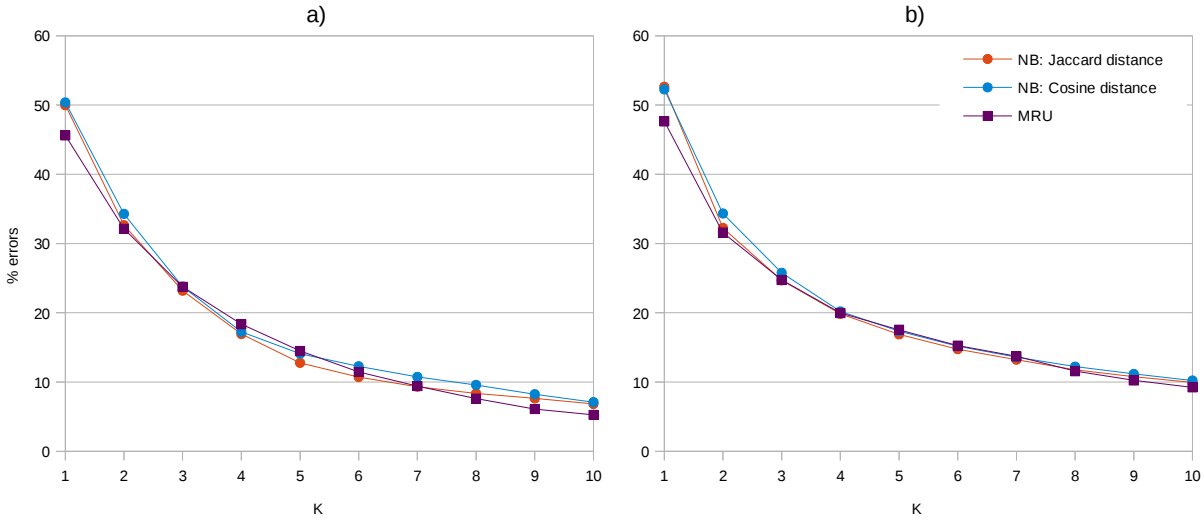

Figure 4.1 – Neighbor based methods against the MRU and bigrams baselines, for the default user (a) and averaged for all users (b). K is the size of the prediction.

4.2 Recommendation using the Cosine distance

As said before the Jaccard distance function remove the application count information transforming the bag of word representations into sets. The cosine distance function allows us to easily keep all the information contained in our bag of words sequences. It measures the cosine of the angle between two vectors:

$$d_{cosine}(v1, v2) = \frac{< v1, v2 >}{||v1||.||v2||} = cos(v1, v2)$$

The cosine distance is in [-1,1], its value is 1 when the two vectors are identical.

To generate a recommendation we perform a similar calculation as for the Jaccard distance:

$$score_w = \sum_{s' \in bank} d_{cosine}(vec(s), vec(s')).count(w, s').\delta_{w \in s'}$$

Where $vec(s)$ is the vector representation of the sequence s. To get the vector representation each application has to be associated with an index in the vector, the value of the vector at index i is the number of occurrences of the application of index i in s. For example, if we consider the following sequence of applications: $s = A, A, B, C, C, A$ then we can attribute each application to an index $A : 0, B : 1, C : 2$ and $vec(s) = [3, 1, 2]$. $count(w, s)$ is the number of occurrences of w in the sequence s'. The results for this method can be seen on figure 4.1.

From figure 4.1 we can observe that the NB methods beat most of the baselines, only MRU is partially beaten. Since the cosine distance takes into account the counts in the computations we could have expected it to perform better than the Jaccard method. However this is not

the case, Jaccard distance is less precise than the cosine distance and knowing how noisy our dataset is this is probably one reason. Another plausible explanation is that the cosine distance makes the length of each vector irrelevant by normalizing them. So this imply that two vectors have the same influence as long as they are collinear. When working with applications sequences two vectors corresponding two one kind of behavior (so two vectors that should be close to each other) might not be collinear. For example the sequences $s_1 = A, B, A, C, A, A, A$ and $s_2 = A, B, A, C, A, A, A, A, A, A, A$ might represent the same mindset yet once normalized they appear as different, the Jaccard distance does not make this mistake and hence is better for our task.

5 Latent Dirichlet Allocation (LDA) based methods

We can easily understand that a user might have a different behavior depending on his location and time or simply have different routines such as checking his social networks using mainly facebook, twitter and whatsapp or checking his emails before opening his calendar. It is those topics that we wish to recover using topic modeling methods such as LDA. This clustering would allow us to generate predictions customized for each sequence depending on their topics distribution.

5.1 Basic LDA

5.1.1 Introduction to LDA

Latent Dirichlet Allocation is a generative model for topic modeling first introduced by Blei in 2003 [6]. Let's change of context and imagine we have textual data instead of smartphone applications sequences. Our dataset D consists in a set of documents, each document being a set of words under the bag of words assumption. LDA assumes each document is a mixture of topics. For example, a document such as the one introduced in figure 5.1 can be considered as a blend between mainly the NLP, machine learning and Bayesian inference topics. The LDA model assumes that each word contains topical information and can be interpreted as being generated by one topic. In this setup a document can be generated by sampling repetitively a topic z from the document's topic distribution $\theta_{z,doc} = P(z|doc)$ before sampling a word w from the distribution over the vocabulary for that topic $\phi_{w,z} = P(w|z)$.

The generative process is simple, however we initially only have our unlabeled corpus D, with no knowledge of what the topics ϕ and topics mixtures θ might be. We hence need to reverse the generative model and infer those latent parameters θ and ϕ from D. All this is done without any prior semantic knowledge, the topics may or may not make sense for us, but they make sense from the point of view of the data.

For what comes next we consider Z as the number of topics, w_i is a word occurring in position

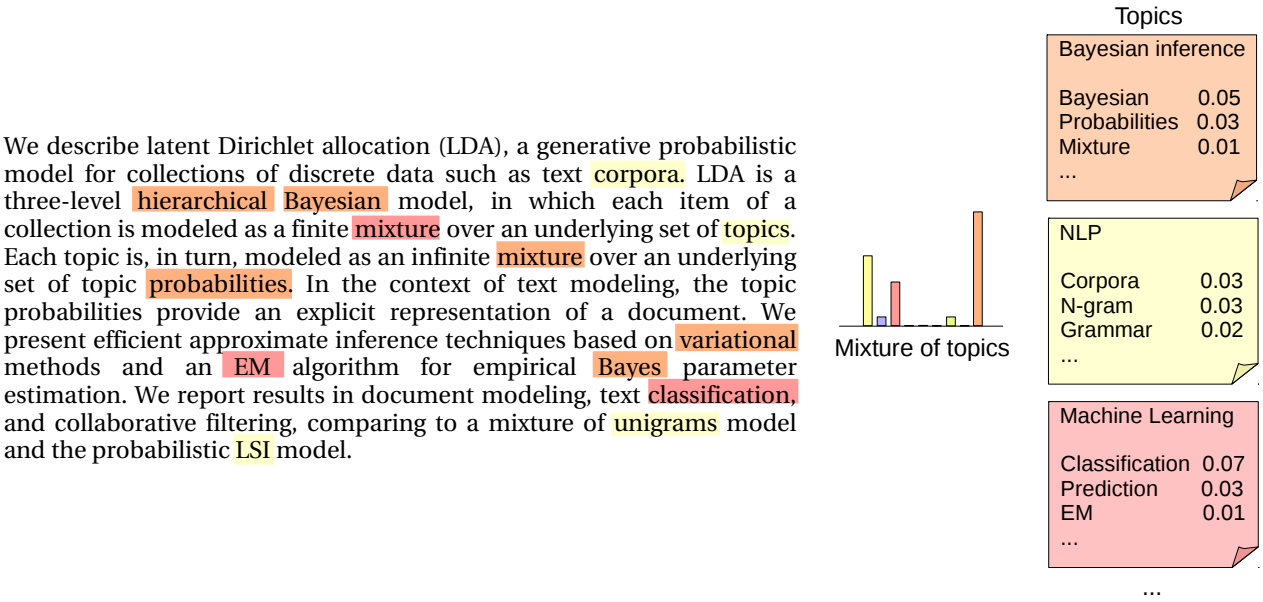

Figure 5.1 – This text is the abstract of [6]. A document d is associated with a mixture of topic θ_d, each topic is a probability distribution over the vocabulary. Some of the words have been underlined to show which topic generated them. Documents similar in content could be generated by repetitive sampling of the document topics mixture i.e. by first sampling a topic from θ_d and then sampling a word from ϕ_z

i, not to confuse with w, the word (i.e. label) itself. Similarly z_i is the topic associated to w_i, not to confuse with z, the topic index.

The inference problem is the problem of finding the topical information contained in each word. Several topic models already existed before LDA such as PLSI. The main contribution of LDA comes from the priors added on θ and ϕ: $\theta \sim Dirichlet(\theta|\alpha)$ and $\phi \sim Dirichlet(\phi|\beta)$. They add prior knowledge on the shapes of the topic mixtures and topic distributions thus reducing overfitting. They also create a link between new observations and the model allowing, when a new document d comes in, to estimate the topic distribution θ_d. The choice of dirichlet is not random, the topic attribution being multinomial the posterior will also be a dirichlet distribution.

$\alpha = \{\alpha_1, ..., \alpha_Z\}$ and $\beta = \{\beta_1, ..., \beta_{|L|}\}$ are hyper-parameters of those priors. Each α_i, resp. β_i, relate to one topic, resp. one word. A large α_i among others implies a relatively high probability to sample the associated topic. If all the α_i are small, we are likely to sample sparse distributions. Then θ will tend to contain only few important topics and ϕ only few important words for each topic. A representation of a dirichlet distribution for several parameters is in figure 5.2. In all our experiment we consider the topics and words to be equiprobable and all the α_i and all the β_i are the same, we will simply call them α and β.

As we are trying to infer the latent parameters of our model the distribution we are interested

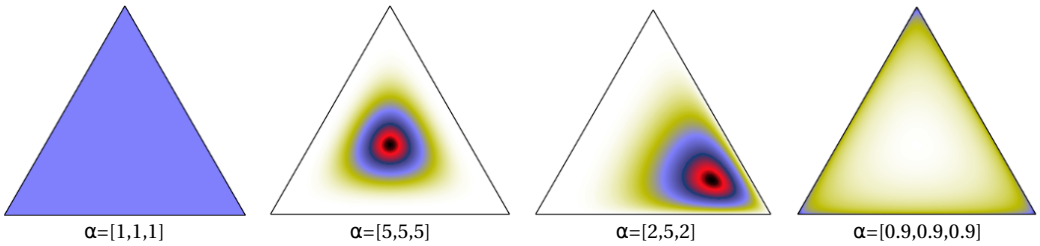

Figure 5.2 – Dirichlet distribution of dimension 3 represented in the 2-simplex. Each point expressed in barycentric coordinates inside the triangles is associated to a color representing the probability of this distribution to be sampled. The darker, the more probable. For α_i less than 1 the probability mass is concentrated on the side and sampled distribution will likely be sparse.

in is $P(z,\theta,\phi|w,\alpha,\beta)$. Using the product rule we can write:

$$P(z,\theta,\phi|w,\alpha,\beta) = \frac{P(w,z,\theta,\phi|\alpha,\beta)}{P(w|\alpha,\beta)}$$

This denominator of this equation is not traceable due to the exponential number of configurations:

$$P(w|\alpha,\beta) = \int\int \sum_z P(w,z,\theta,\phi|\alpha,\beta) d\phi d\theta$$

A common method to answer this problem is to try to estimate the distribution $P(z|w,\alpha,\beta)$ where we marginalized over θ and ϕ. θ and ϕ can then be obtained via an empirical evaluation of this distribution.

$$P(z|w,\alpha,\beta) = \frac{P(z,w|\alpha,\beta)}{P(w|\alpha,\beta)} = \frac{\prod_i P(z_i,w_i|\alpha,\beta)}{\prod_i \sum_z P(z_i=z,w_i|\alpha,\beta)}$$

Here again the denominator is an issue since it consists of $|D|^Z$ terms. One solution is to use Markov Chain Monte-Carlo (MCMC) methods such as Gibbs sampling [3, 12, 14] to estimate this distribution. Gibbs sampling consists in sequentially estimating the probability of z_i to be affiliated with w_i, conditioned on the topic affiliation of all the other terms z^{-i}. We then change the current z_i for a new one sampled from this $P(z_i|z^{-i},w,\alpha,\beta)$. We perform this procedure iteratively on all the terms of the corpus to converge to a high probability state.

$$P(w,z|\alpha,\beta) = P(w,z_i,z^{-i}|\alpha,\beta) = P(w,z^{-i}|\alpha,\beta).P(z_i|w,z^{-i},\alpha,\beta)$$

Hence:

$$P(z_i|w,z^{-i},\alpha,\beta) = \frac{P(w,z|\alpha,\beta)}{P(w,z^{-i}|\alpha,\beta)}$$

The denominator is constant with respect to z_i, the numerator can be written as:

$$\begin{aligned} P(w,z|\alpha,\beta) &= P(w_i,z_i,w^{-i},z^{-i}|\alpha,\beta) \\ &= P(w_i|z,w^{-i},\beta).P(z_i,w^{-i},z^{-i}|\alpha,\beta) \\ &= P(w_i|z,w^{-i},\beta).P(z_i|w^{-i},z^{-i},\alpha).P(w^{-i},z^{-i}|\alpha,\beta) \\ &\propto P(w_i|z,w^{-i},\beta).P(z_i|z^{-i},\alpha) \end{aligned}$$

Which yields $P(z_i|w,z^{-i},\alpha,\beta) \propto P(w_i|z,w^{-i},\beta).P(z_i|z^{-i},\alpha)$

At each step a new z_i is sampled from $P(z_i|w,z^{-i},\alpha,\beta)$. The distribution hence need to be re-estimated at each step. It can be expressed in terms of word/topic associations counts:

$$P(z_i = z|w,z^{-i},\alpha,\beta) \propto \frac{n^{-i}_{w_i,z}+\beta}{\sum_{w\in V}(n^{-i}_{w,z}+\beta)}.\frac{m^{-i}_{d_i,z}+\alpha}{\sum_{z'\in Z}(m^{-i}_{d_i,z'}+\alpha)} \tag{5.1}$$

Where d_i is the document associated to w_i, $n^{-i}_{w_i,z}$ is the number of times the label of w_i is associated to the topic z, without taking into account the current topic attribution of w_i. Similarly $m^{-i}_{d_i,z}$ is the number of times the topic z occurs in document d, without considering the current topic attribution of w_i. The first part of the formula can be understood as the probability of generating the word w_i from topic z, the second part as probability to pick the topic z knowing we are in document d_i. The more the topic z is present in d_i the more likely this topic is to be affected again to d_i. The more one words is attributed to z on the whole corpus the more this topic is likely to be associated with this same word in the corpus.

By randomly initializing the topic attribution and iteratively sampling z_i as we just described we can converge to high probability states. Now if we need ϕ and θ we can estimate them from the word-topic attribution:

$$\phi_{z,w} = \frac{n_{w_i,z}+\beta}{\sum_{w'\in V}(n_{w',z}+\beta)} \qquad \theta_{d,z} = \frac{m_{d_i,z}+\alpha}{\sum_{z'\in Z}(m_{d_i,z'}+\alpha)} \tag{5.2}$$

To summarize, a pseudocode of the gibbs sampling procedure:

```
input: CORPUS //a set of document

//Initialization:
for docIdx in CORPUS:
    for wIdx in CORPUS[docIdx]:
        currentLabel = CORPUS[docIdx,wIdx]
        topicPaire[docIdx,wIdx] = random(0,Z)
        n[currentLabel,z] += 1
        m[docIdx,z]       += 1

//Gibbs sampling
for each iteration:
    for docIdx in CORPUS:
        for wIdx in CORPUS[docIdx]:
            currentLabel = CORPUS[docIdx,wIdx]
```

```
16              currentTopic = topicPaire[docIdx,wIdx]
17
18              //removing influence of current element:
19              n[currentLabel,currentTopic] -= 1
20              m[docIdx,currentTopic]       -= 1
21
22              //resampling the current topic according to (4.1)
23              newTopic = sample(docIdx,currentLabel)
24
25              //restituting the influence of the updated topic paire
26              n[currentLabel,newTopic] += 1
27              m[docIdx,newTopic]       += 1
28
29  //Estimate phi and theta from counts:
30  estimate_phi()
31  estimate_theta()
```

Now we know how to infer LDA parameters on a set of documents. We still need to define the prediction procedure to be able to estimate θ for new documents. When a new document d comes in we already have, from the gibbs procedure, the topic attribution z as well as the parameters ϕ and θ for our corpus D. To get the topic attribution of d we can reiterate the gibbs procedure for d only. We start by randomly initializing the topic attribution for the words of d. We then iterate over d each time sampling a new topic for each word. The sampling is a bit different from (4.1). Indeed, since we know from our training on D the probability of picking a word knowing the topic we sample according to:

$$P(z_i = z | w, z^{-i}, \alpha, \beta) \propto \phi^D_{z,w_i} \cdot \frac{m^{-i}_{d_i,z} + \alpha}{\sum_{z' \in Z} (m^{-i}_{d_i,z'} + \alpha)} \tag{5.3}$$

We could sample according to (4.1) letting the distribution of words in the new document be taken into account but this would change the model. Since we don't know how relevant the new document is and if it is coherent to add it to the corpus we prefer not modifying the model using (4.3). After several iteration of the sampling part θ_d can be computed using (4.2).

5.1.2 LDA for smartphone applications sequences

To adapt LDA to our application we consider one sequence to be one document (see figure 2.1). With this representation one topic ϕ_z is a distribution over the smartphone applications. As shown previously in figure 2.2, our documents are quite small compared to standard NLP applications. This might be an issue at the prediction phase when we have only a small subpart of a sequence to predict the associated topic mixture θ. We will quantify the associated loss after introducing the application prediction mechanism.

Using LDA on the training set of sequences of applications we can get ϕ^{train} and θ^{train}. The later, θ^{train} represent the topic mixture for all the sequences of the training set and is therefor not very interesting to generate a prediction for a new sequence. ϕ^{train}, however, is representative of the topics extracted on the training set, it contains the topic information

stored in each word. For a new sequence coming into our system $s = w_1, w_2, ..., w_n$, we would like to predict the application w_{n+1}. We can predict the topic mixture of s, θ^s. Now we have the global model ϕ^{train} and the local model θ^s we can rank all the applications by their probability of being generated by the LDA generative procedure i.e. integrating their probability of being generated by each topic knowing θ^s and ϕ^{train}:

$$\begin{aligned} P(w|\phi^{train}, \theta^s, \alpha, \beta) &= \sum_z P(w, z|\phi^{train}, \theta^s, \alpha, \beta) \\ &= \sum_z \phi^{train}_{z,w} . \theta^s_z \end{aligned}$$

With our evaluation procedure, the prediction is done for each of the 3 last applications of each sequence. For all the sequences of the test set $s_i = w_{i,1}, w_{i,2}, ..., w_{i,N}$ we predict $w_{i,N}$, $w_{i,N-1}$, and $w_{i,N-2}$ building θ^i with respectively $s_{1:N-1}$, $s_{1:N-2}$ and $s_{1:N-3}$. Once the ranking has been generated we keep only the K most probable applications and see if the target application is contained in it. Result of this procedure can be seen figure 5.3.

When working with machine learning algorithms, overfitting should be monitored. If we create a model on a training set how well will we generalize to another dataset? No overfitting means no significant differences of results between the training set and the test set. Another important effect to monitor is the impact of inferring θ from short sequences. To measure those two components we each time plot two curves in addition to the standard one:

- **The oracle:** Those curves are obtained working only on the training set. First ϕ^{train} is computed on the training set, then we run the prediction process for all the sequences of the **training** set and each time we infer θ^i from $s_{1:N}$. With this procedure the algorithm knows everything at the moment of the prediction, even the label it is supposed to predict, hence the name oracle. The oracle is representative of how well the model fit the training data regardless of how well it generalizes to other dataset, and without taking into account the problem of inferring θ^i from truncated sequences. Hence the oracle show results we would have if the two components describe before were non-existent.
- **The lower bound:** Using the training set to get ϕ^{train}, we generate prediction for the sequences of the test set. However for each prediction θ^i is inferred from the entire sequence $s_{1:N}$. This curve represent the result we would have if the topic mixture estimated were perfectly estimated.

From these two curves, we can see the difference between the standard pipeline and the lower bound shows the cost of doing the inference on truncated sequences. Therefor quantify the problem of working with small sequences. The difference between the lower bound and the oracle show the generalization of the model i.e. the overfitting.

We can see a small partial improvement over the MRU baseline for the default user, yet MRU is still far better when averaging over all users. LDA generalize well, there is no real overfitting.

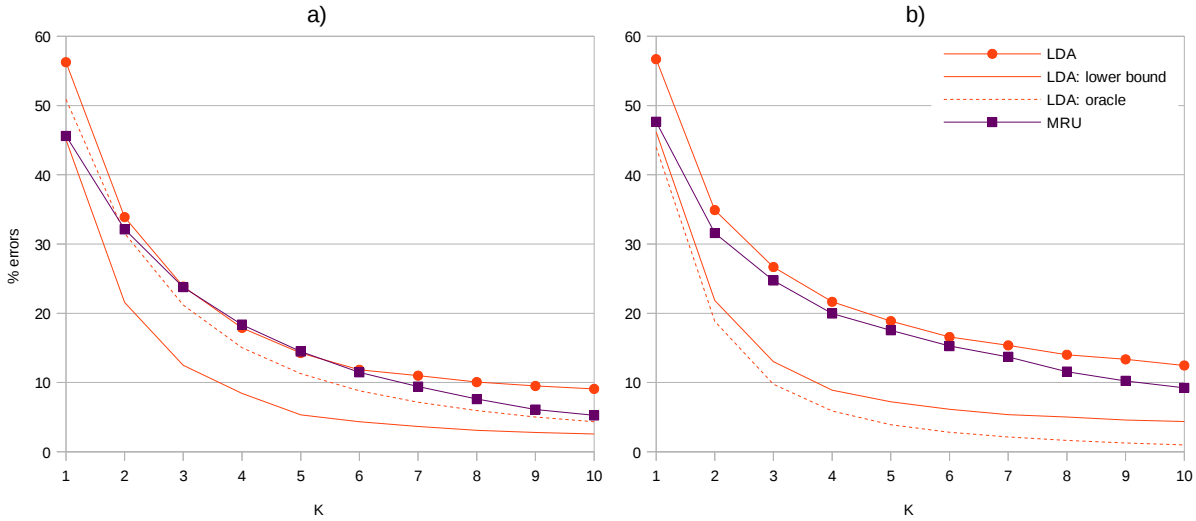

Figure 5.3 – LDA method against the MRU and bigrams baselines, for the default user (a) and averaged for all users (b). K is the size of the prediction.

However with this approach we are under the bag of words (bag of apps) assumption. We are not considering any order in our sequences. The MRU baseline shows how important local information is and reveal there might be correlations between one element and its immediate past. The next methods try to take into consideration the sequence order.

From figure 5.3 we can see we partially beat the bigrams baseline. However we are doing way worst than MRU. Comparing the oracle and the lower bound we can see LDA generalizes well, the overfitting stay relatively small. LDA use the bag of words assumption. The success of MRU reveals how local information matter to generate good prediction. In the next methods we are going to break the bag of word assumption and try to use the order of the applications in the sequences to refine our prediction.

5.2 Distant N-gram Topic Model (DNTM)

The DNTM model was first introduced to mine long sequences from large scale location data [7, 8, 9]. It adds a topic soft clustering step on top of the distant N-grams tool. This allows to associate a topic z to sequences of different sizes.

5.2.1 Distant N-grams

Considering a sequence $d = w_1, w_2, ..., w_n$, a distant N-gram model is the predictive distribution: $P(w_i|w_j,\delta)$, i.e. the probability for the label w_i to be at a distance δ of label w_j. Conditioning on the distance avoid conditioning on all the intermediate labels between w_i and w_j like standard N-grams model do. This yields that the parameters of our model will

grow linearly with the maximum distance δ considered. If the maximum distance is N and the vocabulary size is V we will have $V \times V \times N$ parameters. Standard N-grams however have their parameters growing exponentially with N: V^N.

5.2.2 Topic modeling and distant N-grams

The DNTM model define a corpus as a set of documents, each document being a set of sequences of size inferior or equal to N. N is the maximum length to which two labels can be correlated. The DNTM model assign a topic to each sequence just as LDA assign a topic for each word. The dependencies of the model are represented in the probabilistic graphical model in figure 5.4. Here are the main parameters of the DNTM model:

- $\theta^d = P(z|d) \sim Dirichlet(\alpha)$: same as in the LDA model.
- $\phi_1 = P(w_1|z) \sim Dirichlet(\beta)$: probability distribution over the first label of the sequences.
- $\phi_j = P(w_i|z, j, w_1) \sim Dirichlet(\beta)$: probability to sample one word knowing it is at distance j from w_1, for the topic z.

We can create new documents using the following generative process:

- Draw a topic $z \sim \theta^d$. Then, for each sequence of our document:
- Draw the first term of the sequence $w_1 \sim \phi_{1_z}$
- Draw all the other terms $w_j \sim \phi_{j_z}$ for $j \in [2, N]$

The inference rules can be obtained by the same procedure as for LDA. The equations are a bit more complex since instead of words we have sequences. For one document consisting in a set of sequences $d = s_1, ..., s_i, ..., s_M$. One can prove the following equation for the gibbs sampling procedure[], for each sequence $s = w_1, w_2, ..., w_j, ..., w_N$ we resample the topic z_i of s_i following:

$$P(z_i = z|w, z^{-i}, \alpha, \beta) \propto (m^{-i}_{d_i,z} + \alpha).\frac{n^{-i}_{w_1,z} + \beta}{\sum_{w \in V}(n^{-i}_{w,z} + \beta)}.\prod_{j=2}^{n}\frac{n^{-i}_{(w_1,w_j)j,z} + \beta}{\sum_{w_a \in V}\sum_{w_b \in V} n^{-i}_{(w_a,w_b)j,z} + \beta} \quad (5.4)$$

Where $m^{-i}_{d_i,z}$ is the number of times the topic z is present in the document d_i for the current topic association without considering z_i. $n^{-i}_{w_1,z}$ is the number of times the label associated to w_1 appear first in all the sequences of our corpus, with topic z, and without considering the current topic attribution z_i. Finally $n^{-i}_{(w_a,w_b)j,z}$ is the number of time the label associated to w_b occurred at a distance j of w_a, with topic z and once again without considering the influence of the current sequence s_i.

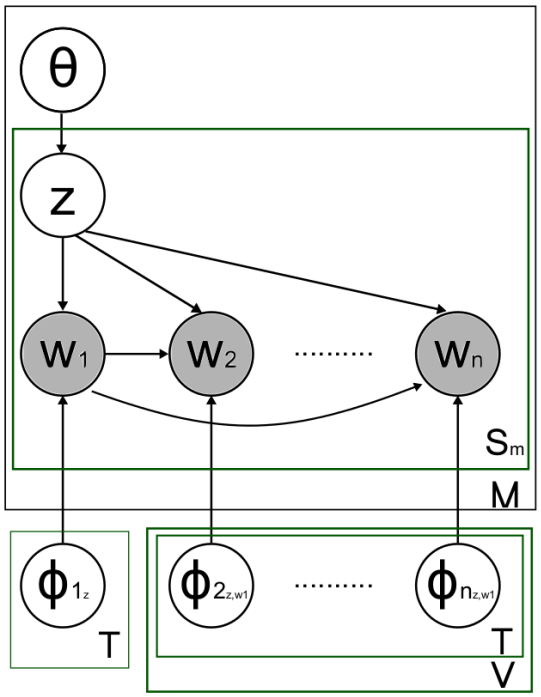

Figure 5.4 – Graphical representation of the DNTM model taken from [8]. We can see how one topic is attributed to one sequence and how the elements of the sequence are generated by the $\phi_{[1:N]}$.

From the sequence/topic pairs, we can extract the parameters ϕ_1, ϕ_j and θ of our model:

$$\phi_{1_{z,w}} = \frac{n_{w_1,z} + \beta}{\sum_{w' \in V} (n_{w'_1,z} + \beta)} \tag{5.5}$$

$$\phi_{j_{z,(w_a,w_b)}} = \frac{n_{(w_a,w_b)j,z} + \beta}{\sum_{w' \in V} \sum_{w'' \in V} n_{(w',w'')j,z} + \beta} \tag{5.6}$$

$$\theta_{d,z} = \frac{m_{d_i,z} + \alpha}{\sum_{z' \in Z} (m_{d_i,z'} + \alpha)} \tag{5.7}$$

When a new document δ comes in, we can perform few gibbs sampling iterations to get the topic mixture of the sequence. During these step we use the previously computed model consisting in ϕ_1^D and ϕ_j^D.

$$P(z_i = z | w, z^{-i}, \alpha, \beta) \propto \phi^D_{1_{z,w_1}} \cdot \frac{m^{-i}_{\delta,z} + \alpha}{\sum_{z' \in Z} (m^{-i}_{\delta,z'} + \alpha)} \cdot \prod_{j=2}^{n} \phi^D_{j_{(w_1,w_j),z}}$$

We then use (4.7) to get θ_δ.

Now let's consider how to interface the DNTM model with our problem. There are two possibilities. The first one is to consider that each of our applications sequence is one sequence

in the DNTM model. This implies that the size of the sequences in the DNTM, N, needs to be big enough. Indeed we wont be able to predict applications outside of our model e.g. for applications sequences larger than N. This is not the only drawback of this first approach. Since the DNTM conditioned each application by the distance it is to the first one, the model will be very sensible to which application comes first. This might be an issue for noisy datasets. The second way to proceed is to consider all the possible subsequences included in our applications sequences. Each application is considered as the beginning of a sequence. For example, if $N = 4$ and we have the following sequence of apps: $s = w_1, w_2, w_3, w_4, w_5$, we will give to the DNTM model 5 sequences: $s[1:4]$, $s[2:5]$, $s[3:5]$, $s[4:5]$ and $s[5]$. With this approach we can deal with sequences longer than N, we are also less influenced by noises.

From the beginning of an applications sequence s, we can predict the most probable next label according to our model. If we follow the "one apps sequence/one DNTM sequence" approach then we just need to integrate over all the possible topics of that sequence:

$$P(w|\phi_1^{train}, \phi_j^{train}, \theta^s, \alpha, \beta) = \sum_z \phi_{l_{z,(w_1,w)}} . \phi_{1_{z,w_1}} . \theta_z^s \tag{5.8}$$

Where l is the distance from the first label to the label we are predicting. Now if we take the second approach we need also to integrate over all the subsequences:

$$P(w|\phi_1^{train}, \phi_j^{train}, \theta^s, \alpha, \beta) = \sum_{w_1 \in Sub(s)} \sum_z \phi_{l_{z,(w_1,w)}} . \phi_{1_{z,w_1}} . \theta_z^s \tag{5.9}$$

Where $Sub(s)$ is the set of all subsequences of s.

We use the same evaluation procedure as for LDA. On the training set we train ϕ_1^{train} and ϕ_j^{train}. For each sequence of the test set we try to predict the 3 last applications based on our model. To find the most probable next label from a truncated sequence we use equation (4.8) or (4.9). Results of the DNTM method for the second approach are shown in figure 5.5.

From figure 5.5 we can observe that the overfitting is of the same order of magnitude as with LDA. The DNTM method performs better than the basic LDA method but we are still less good than the MRU baseline. Figure 5.6 compares the results obtained with the two approaches considered previously. This is no surprises that the first approach performs worse than the second one. As mention earlier, the noisiness of the dataset is too big to have a reliable prediction by conditioning on the first element of the sequence.

5.3 Topical N-gram Model (TNG)

The Topical N-gram model [16, 17] is yet another model trying to break the bag of words assumption that is limiting the LDA model. A recurrent problem occurring when doing topic modeling on documents under the bag of words assumption is dealing with name entities. The sentence *"The white house stands against the threat"* can be interpreted differently depending on the meaning we associate to *white house*. If white house is a name entity then we may

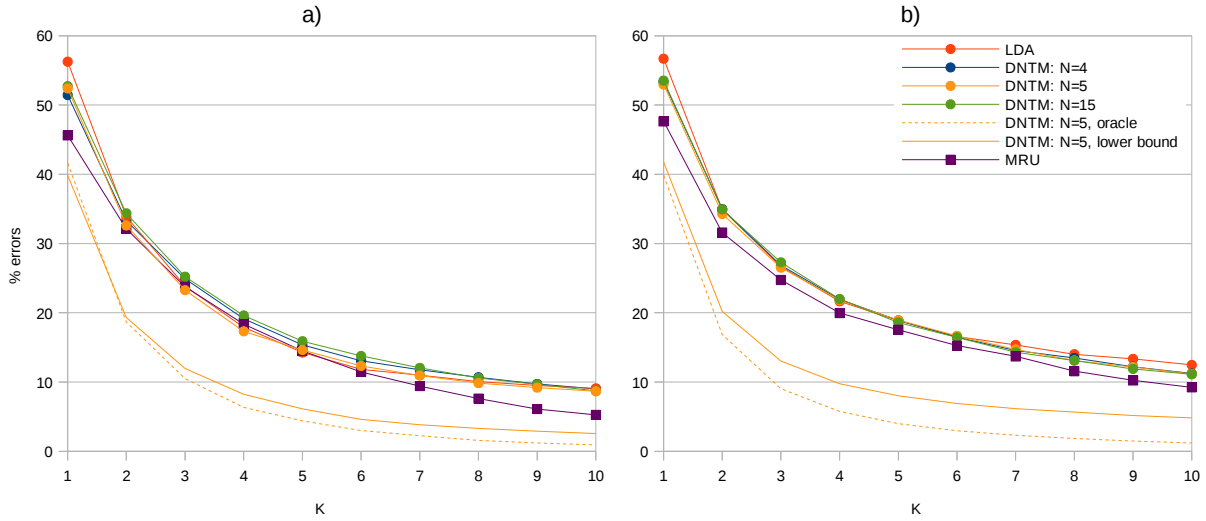

Figure 5.5 – DNTM method against the MRU and LDA, for the default user (a) and averaged for all users (b). K is the size of the prediction.

understand the sentence as a statement on the behavior of the US government against some threat. In the other case where white refers just to the color of the house, the sentence has another interpretation. We can see how tightly linked the context and whether several words form a name entity are. The TNG model is a probabilistic model aiming to perform the topic modeling task, as well as the name entity recognition, at the same time. Depending on the topic, two words have different probabilities to form a collocation together i.e. to co-occur together inside this topic.

5.3.1 Theory behind the TNG model

The TNG model is complex, its parameters are:

- $d = w_1, w_2, ..., w_i, ...$ is a document.
- z_i is the topic attributed to w_i. x_i is a binary parameter of the TNG which decide whether w_{i-1} and w_i form a collocation.
- $\phi = P(w|z) \sim Dirichlet(\beta)$ as before.
- $\theta = P(z|doc) \sim Dirichlet(\alpha)$ as before.
- $\psi = P(x_i|z_{i-1}, w_{i-1}) \sim Dirichlet(\gamma)$ defines the probability for two consecutive words to form a collocation based on the previous word and attributed topic. For example $P(x_2 = 1|z_{i-1} = "politics", w_{i-1} = "white")$ will be high since we expect the name entity *"white house"* for the topic *"politics"*.

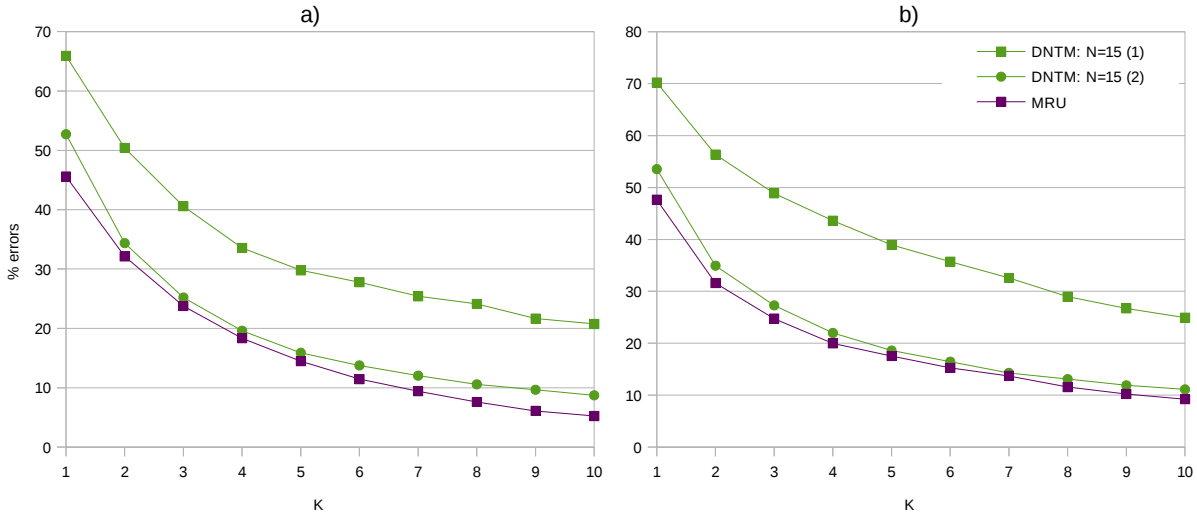

Figure 5.6 – Approach 1 apps sequence/1 DNTM sequence (1) against approach multiple DNTM sequences for 1 apps sequences (2). $N = 15$ hence we cannot predict labels for sequences larger than 15, those sequences are not taken into account in our evaluation.

- $\sigma = P(w_i|z_i, w_{i-1}) \sim Dirichlet(\delta)$ defines the probability to sample a word knowing his topic z_i and the previous word w_{i-1}.

The graphical model summarizing the probabilistic dependencies is shown in figure 5.7. The generative procedure to create one new document d is the following:

- Draw a topic distribution θ^d from $Dirichlet(\beta)$
- For each word w_i of the document we are going to generate:
- Start by sampling a topic z_i from θ^d.
- Draw x_i from $\psi_{w_{i-1},z_{i-1}}$.
- If $x_i = 0$ i.e if the two consecutive words form a collocation, draw w_i from ϕ_{z_i}.
- Else, the two words form a collocation and we sample w_i from $\sigma_{z_{i-1},w_{i-1}}$.

From here we can see that in the case $\forall i, x_i = 1$ then the model is a standard LDA model. If $\forall i, x_i = 0$ then the model is a LDA bigram model. In order to do the inference of the parameters on a corpus D, we can develop the same logic as for LDA and find [,]:

$$P(z_i = z, x_i = x|w, z^{-i}, x^{-i}, \alpha, \beta, \delta, \gamma) \propto (\gamma + p^{-i}_{x,z_{i-1},w_{i-1}}).(\alpha + m^{-i}_{d_i,z}) \times \begin{cases} \frac{\beta + n^{-i}_{z,w_i}}{\sum_{w' \in V}(\beta + n^{-i}_{z,w'})} & \text{if } x = 0 \\ \frac{\delta + q^{-i}_{z,w_{i-1},w_i}}{\sum_{w' \in V} \delta + q^{-i}_{z,w_{i-1},w'}} & \text{if } x = 1 \end{cases}$$

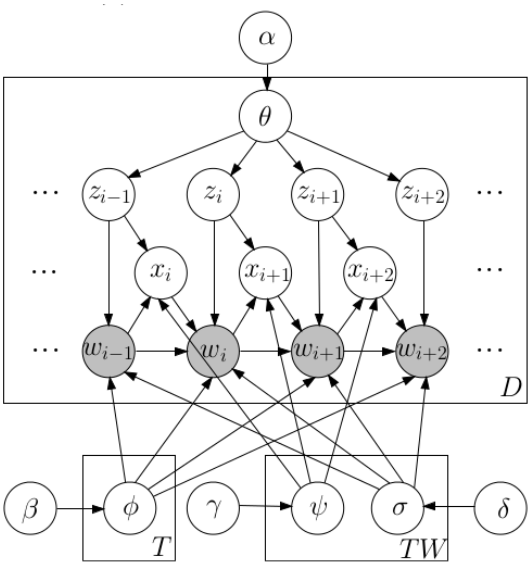

Figure 5.7 – Graphical model for the TNG model. Figure taken from [17].

where $p^{-i}_{x,z_{i-1},w_{i-1}}$ is the number of times $x_i = x$ knowing w_{i-1} and z_{i-1}. $m^{-i}_{d_i,z}$ is the number of times the topic z is attributed in the document associated with w_i, d_i. n^{-i}_{z,w_i} is the number of times the topic z has been attributed to the label of w_i. Finally q^{-i}_{z,w_{i-1},w_i} is the number of times the topic z has been attributed to the label of w_i knowing w_{i-1}. Once again the superscript $-i$ means the current values of x_i and z_i should not be added into the counts.

Once the sampling is done we can extract all our parameters using the counts:

$$\phi_{z,w} = \frac{\beta + n_{z,w}}{\sum_{w' \in V} (\beta + n_{z,w'})} \tag{5.10}$$

$$\theta_{d,z} = \frac{\alpha + m_{d,z}}{\sum_{z'} (\alpha + m_{d,z'})} \tag{5.11}$$

$$\psi_{z,w,x} = \frac{\gamma + p_{x,z,w}}{\sum_{x' \in \{0,1\}} (\gamma + p_{x',z,w})} \tag{5.12}$$

$$\sigma_{z,w,w_{i-1}} = \frac{\delta + q_{z,w_{i-1},w}}{\sum_{w' \in V} \delta + q_{z,w_{i-1},w'}} \tag{5.13}$$

When a new document δ comes in, we can perform few gibbs sampling iterations to get the topic mixture of the sequence. During these step we use the previously computed model consisting in ϕ^D, ψ^D and σ^D.

$$P(z_i = z, x_i = x | w, z^{-i}, \alpha, \beta, \delta, \gamma) \propto \psi^D_{z,w_i,x}.(\alpha + m^{-i}_{d_i,z}) \times \begin{cases} \phi^D_{z,w} \\ \sigma^D_{z,w_i,w_{i-1}} \end{cases}$$

We then use (4.11) to get θ_δ.

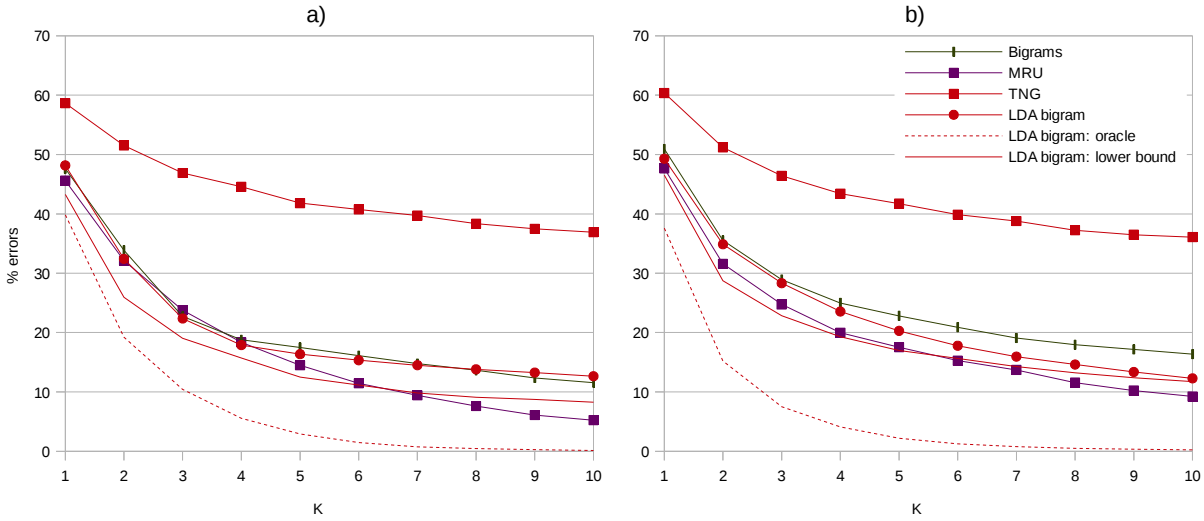

Figure 5.8 – TNG model versus DNTM and MRU. When $x_i = 1$ the model is equivalent to a LDA bigram model.

5.3.2 TNG for smartphone applications sequences

Our hope is to train the system to detect co-occurring applications in different context. As before we compute our model $\phi^{train}, \psi^{train}, \sigma^{train}$ on the training set. When a new sequence $s = w_1, ..., w_n$ comes in and we want to predict the most probable w_{n+1}, we need to integrate over all the topics z_{n+1} and all the x_{n+1} possible:

$$P(w_n = w|\phi^{train}, \psi^{train}, \sigma^{train}, \theta^s, \alpha, \beta, \delta, \gamma) = \underbrace{\sum_z \phi^{train}_{z,w}.\theta^s_z.\psi_{z,w,x=0}}_{w \text{ and } w_{i-1} \text{ do not form a collocation}} + \underbrace{\sum_z \sigma^{train}_{z,w,w_{i-1}}.\theta^s_z.\psi_{z,w,x=1}}_{w \text{ and } w_{i-1} \text{ form a collocation}}$$

In the case we force all the x to 1, we end up with a LDA bigram model and the prediction equation becomes:

$$P(w_n = w|\sigma^{train}, \theta^s, \alpha, \beta, \delta, \gamma) = \sum_z \sigma^{train}_{z,w,w_{i-1}}.\theta^s_z$$

Results for the TNG method are shown in figure 5.8. The LDA bigram model performs way better than the TNG model. This might be due to the high the number of parameters of the TNG. Being a quite complex model it requires a lot of data to train correctly. The oracle and lower bound are shown for the LDA bigram model, they reveal a lot of overfitting happening. The LDA bigram model does no better than the DNTM. Comparing the simple bigram model with the LDA bigram model we can see the improvement of using topics in the prediction. Several alphas and betas have been tried to get those curves yet it might be possible to find a better set of parameters.

We observe much more overfitting than with the other methods. The results are less good than with the DNTM. Looking at the lower bound we can see that even if we were able to infer a perfect topic mixture for every sequence we would only partially beat MRU. The TNG model performs very bad, because of its complexity and number of parameters it requires a large dataset to give correct results.

5.4 LDA trigram model

The LDA trigrams model is extremely similar to the LDA or LDA bigram model. For LDA, tokens are unigrams, for LDA bigrams tokens are bigrams, for LDA trigrams tokens are trigrams. We directly give the gibbs sampling inference equation:

$$P(z_i = z|w, z^{-i}, \alpha, \beta) \propto \frac{n^{-i}_{w_i,w_{i-1},w_{i-2},z} + \beta}{\sum_{w' \in V}(n^{-i}_{w',w_{i-1},w_{i-2},z} + \beta)} \cdot \frac{m^{-i}_{d_i,z} + \alpha}{\sum_{z' \in Z}(m^{-i}_{d_i,z'} + \alpha)}$$

Where $n^{-i}_{w_i,w_{i-1},w_{i-2},z}$ is the number of time the trigram (w_i, w_{i-1}, w_{i-2}) is associated to the topic z in the corpus. $m^{-i}_{d_i,z}$ is the number of times the topic z is sampled in the document d_i. The superscript $-i$ indicates the statistics of the current word w_i are not taken in the counts.

Once we have the counts from the gibbs sampling step we can get our parameters ϕ and θ:

$$\phi_{w,w_{i-1},w_{i-2},z} = \frac{n_{w_i,w_{i-1},w_{i-2},z} + \beta}{\sum_{w' \in V}(n_{w',w_{i-1},w_{i-2},z} + \beta)}$$
$$\theta_{z,d} = \frac{m_{d,z} + \alpha}{\sum_{z' \in Z}(m_{d,z'} + \alpha)}$$

To generate our prediction for an application w_i of a sequence s, knowing the two previous labels w_{i-1} and w_{i-2}, we integrate over all the possible topics that could have generated this trigram:

$$p(w_i|\phi^{train}, \alpha, \beta) = \sum_z \theta^s_z . \phi^{train}_{w_i,w_{i-1},w_{i-2},z}$$

Results for the LDA trigrams method can be seen in figure 5.9. The curves reveal a huge amount of overfitting. We already had a strong overfitting problem with bigrams so this is not a big surprise. For a small dataset like ours, most of the trigrams are unique, we don't have enough data to infer a relevant statistic over trigrams. Since many of the trigrams of the test set cannot be found in the training set the lower bound and normal pipeline perform equally bad.

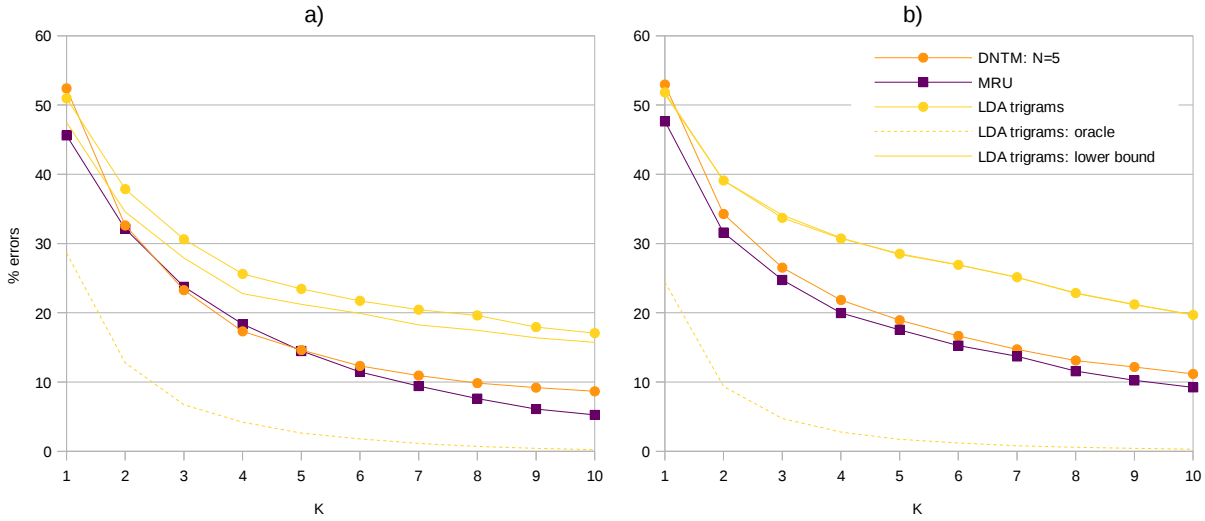

Figure 5.9 – LDA trigrams vs. DNTM and MRU.

5.5 The generalization problem

In nearly all the results presented so far we significantly overfitted our data. All those models have been first introduced to handle texts. They all showed themselves successful when applied on large corpus of hundred of thousand documents. Our dataset is much smaller, and our sequences contains only few labels. The smaller the dataset is the more likely we are to overfit. When studying natural languages strong assumptions are being made. Assumptions that do not necessarily fit our problem. A language is built upon a strict set of rules and two trigrams like *"reads the book"* and *"the book reads"* have fundamentally different meanings. The importance of order in a sentence allows the use of ngrams statistics. In our case nothing enforces a grammar on the applications sequences, or let's say that the grammar is very loose, at least locally. In an applications sequence, two nearby applications are correlated but their order might not matter that much. Still in the same sequence, two applications dispatched on the two sides of the sequence are correlated, and their order matters. At least this is the hypothesis we are formulating here: a sequence is locally under the bag of words assumption and globally under strict ordering assumption. If the dataset was large enough the ngram statistic could naturally reflect this.

To enforce this assumption for ngrams models we add a preprocessing step. For a given window size W representing the scale at which we are under the bag of words assumption we generate all the ngrams (example for a window of size 4 and a trigram model):

$$w_0, w_1, w_2, w_3 \rightarrow (w_0, w_1, w_2), (w_0, w_2, w_3), (w_1, w_2, w_3), (w_0, w_1, w_3)$$

We can see we do not generate every N-grams possible since it would yields an exponential

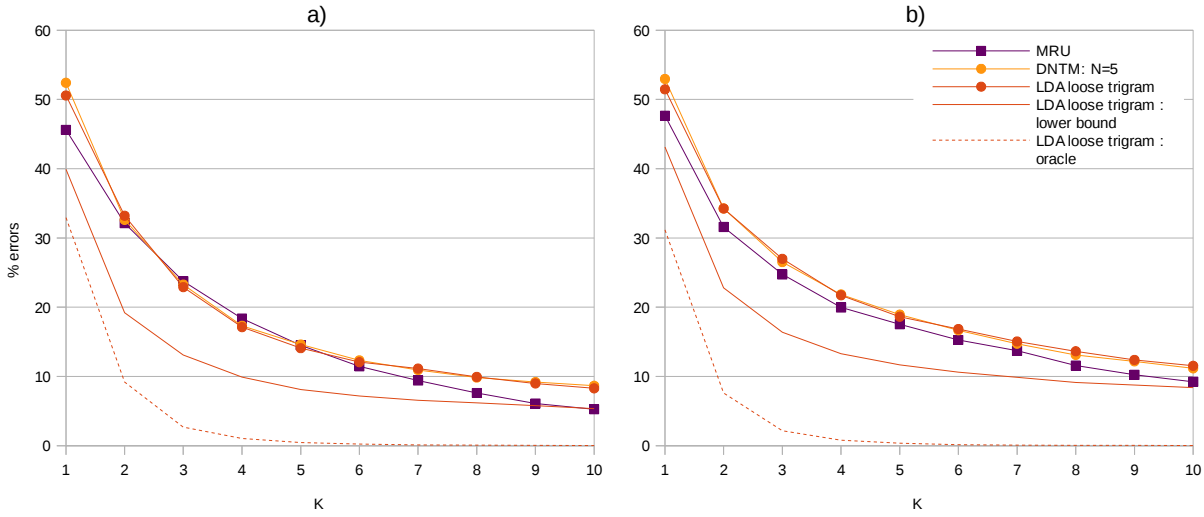

Figure 5.10 – Loose trigram model, DNTM and MRU. The size of the window here is 5.

complexity. Instead we only generate N-grams that respect the time order. Since we are not really generating all the possible combinations of apps inside the window, the model is similar to a skip n-gram model. The size of the dataset is greatly increased by this process. This method smooths our dataset more or less depending on the window size.

5.6 Loose LDA trigram model

To force the previously defined assumption, we preprocess the sequences to generate all the possible trigrams according to the procedure defined previously. The inference part stay unchanged, but we need to adapt our prediction formula. Before we were integrating over all possible topics for the observation w_{t-1}, now we can have several w_{t-1} since the order is not relevant inside our window of size W. To predict the label w_n of a sequence $s = w_1, ..., w_n$.

$$P(w_n = w|\theta^s, \alpha, \beta) = \sum_{\{w_i, w_j \in s_{[n-W:n]} | i<j\}} \sum_z \phi^{train}_{z,w,w_j,w_i} . \theta^s_z$$

Where here again $phi_{z,w,w_j,w_i} = P(w|w_i, w_j, z)$. The results obtained with this method can be seen on figure 5.10. The lower bound and oracle are closer than for the non-loose LDA trigram model, which means we have less overfitting.

If we compare figure 5.10 and figure 5.9 we can see a huge improvement over the simple LDA trigram model. The lower bound and oracle are much closer which means we overfit less. However the results did not really improve the current best LDA based method that we had. I don't think those results are sufficient to confirm our hypothesis of being locally under bag of words. Generating all the possible trigrams as we did just smoothed the dataset which solved the overfitting problem, it just confirms that the dataset is too noisy to use trigrams. This also

implies that we do not really have much sequence information inside the sequences. This observation is confirmed by the comparison between LDA and DNTM, considering sequences only slightly improved the results.

5.7 Conclusion on LDA methods

After investigating all those LDA based methods, it seems we got closer and closer to the MRU baseline without ever being able to beat it. Looking at the top 5 words for each topics in table 5.1, we can see how similar to each others all those topics are. They are all very close to the most frequently used ranking. Similar results have been obtained when modifying the parameters and number of topics. The similarity between each topic reveal the redundancy of information present in all our sequences. Sequences seem to be all working around the same core applications (the one at the top of the MFU ranking) and may differ more for less frequently used applications. The same observation can be done for sequences analysis methods derived from LDA and if we get better results out of these methods it is merely because of the sequence modeling tool (bigrams, trigrams, distant n-grams ...) used more than the topic modeling aspect. It is however important to note that using topic modeling improve the results, see for example standard bigrams and LDA bigrams. Identical observations have been done with basic loose trigrams and LDA loose trigrams as well as distant n-grams and distant n-gram topic model.

Table 5.1 – 6 LDA generated topics picked at random. The parameters used for LDA are the same as for the LDA experiments, which are the parameters giving us the best results.

topic#1	topic#2	topic#3
mozilla.firefox	nianticproject.ingress	nianticproject.ingress
nianticproject.ingress	android.calendar	mozilla.firefox
google.android.wearable.app	mozilla.firefox	google.android.gm
google.android.gm	google.android.gm	android.calendar
android.calendar	sonyericsson.album	sonyericsson.album

topic#4	topic#5	topic#6
nianticproject.ingress	nianticproject.ingress	nianticproject.ingress
mozilla.firefox	google.android.gm	mozilla.firefox
google.android.gm	mozilla.firefox	google.android.gm
android.calendar	android.calendar	android.calendar
sonyericsson.album	sony.sensing.has.activitypug	google.android.wearable.app

Another weak point of LDA based method is the inference of the topic mixture. The different lower bounds shown previously reveal that we could get much better results, even beat MRU, if we were able to infer θ^s properly. The problem lies in the short size of the sequences. When we infer θ^s from the beginning of a sequence, each word in this sequence will have a huge impact

on the topic mixture since we have 2 or 3 words on average. If the missing word that we want to predict is among the important words of one topic, the fact that it is missing will greatly reduce the influence of that topic in θ^s. Hence we will likely not recommend the missing word. On another hand, if the missing word is not important for all topic, then the fact that it is missing will not affect θ^s, but the word wouldn't have been recommended in the first place since it is associated with a low probability in every topic. This inability to model incomplete sequences is compensated by how close the topics are from each other.

6 Conditional Random Fields

In previous chapter we saw many methods using different parts of the information contained in the data. We also tried merging these methods with other ones, like when we added topic modeling on top of n-grams using LDA. Conditional Random Field (CRF) is a tool to perform prediction over sequences of labels, considering the labels as a set of features. This ability to do prediction by merging a lot of independent features is the great advantage of CRF. We hope that blending all our previously defined features using CRF will yield better results than the features used alone.

6.1 Introduction to CRF

6.1.1 Log-linear model

In this section we are going to introduce logistic regression, a special case of log-linear model. The model is the following:

$$p(y|X,\alpha,\beta) = \frac{1}{1+e^{-(\alpha+\sum_j \beta_j.x_j)}}$$

Where $X = x_1, x_2, ..., x_d$ is a vector of features, associated to the observation y. α and $\beta = \beta_1, \beta_2, ..., \beta_d$ are two parameters of our model. If we are trying to predict the outcome of a coin flip, then the features might be the weight of the coin, its material, etc.. The observation y would be the outcome of the coin flip, *head* or *tail*. The model could then be interpreted as a linear combination of features squashed through a logistic function. This is very similar to a simple one neurone neural network, as introduced in chapter 9. The aim of logistic regression is to learn from a dataset D of (y, X) pairs the best α and β, i.e. the parameters that gives the best prediction y over this dataset. The linear combination of features $-(\alpha + \sum_j \beta_j.x_j)$ is the simplest model possible. The logistic function is there to ensure that the final value is between 0 and 1 as it represents a probability. $p(y|X,\alpha,\beta)$ is the conditional likelihood.

To train the two parameters we use gradient ascent to maximize the conditional likelihood i.e. maximize the probability of observing the y's given the X's and α and β:

$$argmax_{\alpha,\theta} = \prod_{i \in D} p(y_i|X_i, \alpha, \beta)$$

Since the log function is strictly monotonous we can write:

$$argmax_{\alpha,\theta} = \sum_{i \in D} log(p(y_i|X_i, \alpha, \beta))$$

To simplify the notation we can add 1 at the end of every X, and include α in β, so $\beta = \beta_1, \beta_2, ..., \beta_d, \beta_{d+1}$ with $\beta_{d+1} = \alpha$. We then can write:

$$p(y|X, \alpha, \beta) = \frac{1}{1 + e^{-\sum_j \beta_j . x_j}}$$

Gradient ascent consist in iteratively moving in the direction of the gradient to increase the value of the objective function. We hence need to compute the gradient of the log conditional likelihood with respect to the weights $\beta_1, ..., \beta_{d+1}$, one can show that we have, for one training sample (y, X):

$$\frac{\partial log(p(y|X, \alpha, \beta))}{\partial \beta_i} = \begin{cases} -p(y|X, \alpha, \beta).x_i & \text{if } y = 0 \\ (1 - p(y|X, \alpha, \beta)).x_i & \text{if } y = 1 \end{cases}$$

To get the gradient over the entire dataset we can average the gradient for every sample point. Updating the β_i moving in the direction of the gradient allows us to find an optimal parameters. Logistic regression is great to combine features and perform a binary classification task. If we have more than 2 output values possibles we can use multinomial logistic regression which is very similar. The only inconvenient with logistic regression is that it does not handle sequences. Thankfully its generalization, CRF, is doing precisely that.

6.1.2 Conditional Random Field

In this section we introduce the part of speech tagging problem. Given a sentence $X = w_1, ..., w_n$, we would like to associate to each word w_i, a part of speech tag y_i i.e. a label for the category of a word such as noun, verb, article, etc.. We can see logistic regression is not the proper tool to do this since we need to match an output sequence to an input sequence, and the size of this output sequence is of variable size.

The model defines the conditional likelihood as follows:

$$p(Y|X,\beta) = \frac{exp(\sum_j \beta_j.F_j(X,Y))}{Z(X,\beta)}$$
$$Z(X,\beta) = \sum_{Y'} exp(\sum_j \beta_j.F_j(X,Y'))$$

Where each $F_j(X,Y)$ is a feature function computed from the input X and the target Y. Features might take real or binary values. Examples of binary features for the part of speech tagging problem might be: *"if w_i start with a capital letter and $i \neq 0$ then w_i is a proper noun"*. The feature value multiplied by the weight β can be interpreted as a measure of how well the outcome Y goes with X. The larger $\beta_j.F_j(X,Y)$, the higher $p(Y|X,\beta)$. The denominator sum over all the possible Y' to ensure the final value is a probability.

The most commonly used type of CRF are linear chain CRF, where the input and the output are sequences of same length, e.g. the part of speech tagging problem. For these types of CRF, there are some restrictions over the features functions, they should have the following form:

$$F_j(X,Y) = \sum_i f_i(y_{i-1}, y_i, X, i)$$

The feature function are expressed as a sum of sub-features evaluated over the current and previous elements y_i and y_{i-1}, for each position in the entire sequence. This being said features can use the entire input sequence X, or none of it. An example of local feature that depends only on Y might be *"if y_{i-1} is the verb part of speech then y_i is also the verb part of speech"*. This feature should have a very negative weight associated to it after training since to verbs one after another is very unlikely in English. The restriction obviously is that f_i can only consider two consecutive tags. This is a necessary restriction for the algorithm to be tractable. It would hence be impossible to formulate a feature of the type *"current word w_i is ending in "ing" and w_{i-2} is a pronoun"*. To make a prediction we need to find the best tag sequences $\tilde{Y}$:

$$\tilde{Y} = argmax_y(exp(\sum_j \beta_j.F_j(X,Y))) = argmax_y(\sum_j \beta_j.F_j(X,Y)))$$

Furthermore, to evaluate the probability of $\tilde{Y}$ we need to compute the denominator $Z(X,\beta)$ which implies summing over every Y possible. Details are not given here yet the $argmax$ can be found using dynamic programming and the denominator can be computed using a special form of forward-backward algorithm.

The last thing we need to do is to train the weights β. We simply use gradient ascent on the log conditional likelihood just as for logistic regression. A nice particularity of CRF is that we will converge to a global extrema unlike usual application of gradient ascent which tend to converge to local extrema.

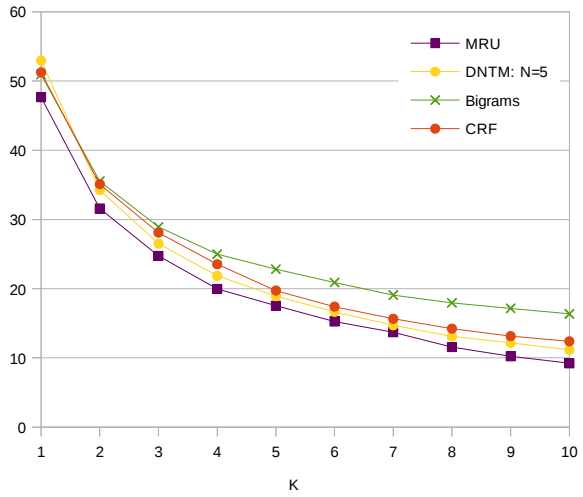

Figure 6.1 – Conditional Random Field vs MRU, bigrams and DNTM. The results are obtained averaged over all users.

CRF hence allows us to express many local features function f_i and provide a global optimization mechanism to extract the best sequence of labels possible given our training data.

6.2 CRF for smartphone application prediction

In our application the labels of each application of a sequence is the next application. Our features are a mix of high level representations obtained using LDA, as well as baseline features:

- The index of the two top LDA topics.
- The previous application w_{t-1} i.e. a bigram model.
- The two previous applications w_{t-1}, w_{t-2} i.e. a trigram model.
- The index of the current application in the sequence.
- All the applications that are present from the begining of the sequence up to the current application.

The results for this method are shown in figure 6.1. We can see CRF performs worse than MRU and DNTM. The main reason is here again overfitting. We do not have enough data to really train a complex model like CRF.

7 Matrix Factorization methods

LDA is a generative approach, once the model is trained we can generate new documents from it. Seen from above LDA is just a smooth clustering method, a way to decompose a document into a features vector θ. This idea of generating features vector is central to machine learning. There exist many method to do so. One successful method especially successful to create recommendation engine [5, 4, 15] is Matrix Factorization (MF).

7.1 Truncated Singular Value Decomposition

From the training sequences it is possible to create a matrix D containing all their bag of words representations as shown in figure 7.1.

If we factorize D using Singular Value Decomposition (SVD), we would get three matrices:

$$D = U.S.V^t$$

Where $U \in M^{|L|\times|L|}$, $S \in M^{|L|\times T}$ and $V^t \in M^{T\times T}$. S is a diagonal matrix containing the sin-

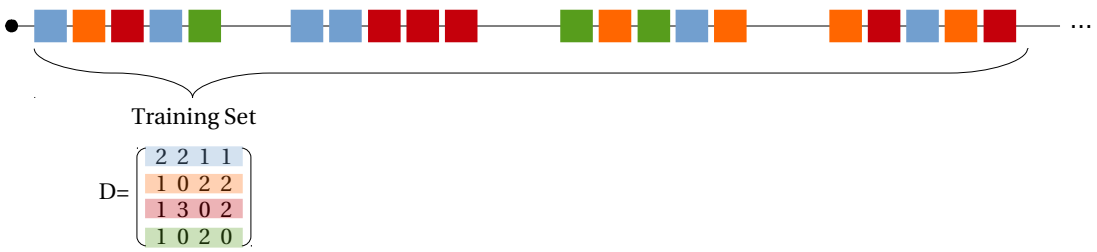

Figure 7.1 – Each color represent one application, the sequences are encoded as bag of words vectors and stored in D. The size of the matrix is $|L| \times T$ where T is the number of sequences in the training dataset.

gular values sorted in descending order. From the SVD decomposition we can get the best approximation $\widetilde{D}$ for D where $\widetilde{D}$ is of rank k. To do so we truncate the U, S and V matrices to respectively the k first columns, singular values and rows yielding $U \in M^{|L| \times k}$, $S \in M^{k \times k}$ and $V^t \in M^{k \times T}$. This approximation is known as the k-truncated SVD.

If we look at the truncated matrices U and V^t, the rows of U can be interpreted as a decomposition of an application into k features (so the columns of U could be seen as the equivalent of ϕ for LDA). The columns of V^t can be interpreted as the feature decomposition of one sequence into k features (similar to θ for LDA). SVD can be used to get the best representation possible of a dataset using k dimensions; best in terms of reconstruction error (frobenius norm). We can understand that a matrix approximating optimally another matrix have to use patterns present in the data. In that sense by reducing the dimensionality with an objective function in mind, we are learning representations of our data. U is the application (or item) space and V^t is the sequence space. Intuitively the application space contains for each application the value of some latent features for that application, the sequence space contains for each sequence how that sequence "like" this feature.

For example, if we consider a movie recommendation system. Instead of sequences of applications let's imagine we have set of movie ratings for several users. Our matrix D would contain in its columns the movies rating for each user. Each movie can be though of a combination of latent genres, for example Star Wars is a lot of fantasy, a lot of action, a little bit of romance, etc. . Of course we don't know these genre up-front since they are latent in our data. If we compute the k-SVD on D we would get automatically the genre decomposition of each movies and the genre appreciation for each user. Those genre would naturally emerge as a solution to a compression problem: how to best encode each movie and user so that I am as close as possible of the original dataset? Compressing data is the same as learning. It is good to emphasize that just like with LDA the feature decomposition is entirely dependent on the data and may or may not make sense to us.

7.2 Prediction using SVD

In the following s is not the sequence of application but the vector representation of that sequence as introduced in chapter 5. From the application space, we have for each application a decomposition into k features. We can use this feature decomposition to generate recommendations. We use the training sequences to get the item space U_{train}. We call v the feature decomposition of s. In our application we know s and we know our model U_{train}, we would like to query this model to get which applications should be in s. The k-SVD gives us for a sequence s an approximation:

$$s \approx s' = U_{train}.S.v$$

The matrix S scales the features in terms of their global importance in the data (if the feature is "action" and we have many action movies in the dataset it should scale the action feature

accordingly), we incorporate it in v:

$$s \approx s' = U_{train}.v$$

s' is the sequence seen by our model. If the value associated to an application increase in s' this means the model is expecting more of that application and it should be recommended. That way, looking at the variation of values $s' - s$ we can see generate a score for each application. The applications with the largest increase should be recommended. However, before getting s', we need to get v, it has to minimize the reconstruction error $s' - s$. From s the best v possible is obtained by projecting s onto the item space. In our movie example this would be equivalent to ask: knowing this user like Star-Wars and James-Bond movies, and knowing the feature decomposition of these movies, what does this tells me for that user? Eventually that would tells you this user like action movies. So we understand we can get the best v available by projecting the apps sequence vector onto our knowledge of what those apps means (learn on the training set):

$$v = U_{train}^{t}.s$$

Which yields:

$$\begin{aligned} s' &= U_{train}.v \\ &= U_{train}.U_{train}^{t}.s \end{aligned}$$

We now have all the information we need to compute s' and generate our recommendation $s' - s$.

7.3 Refining the predictive model

We saw in the previous sections that $s' - s$ was a way to get a recommendation score for each application. In reality we are using a different model:

$$score = s' - s + \lambda.s$$

Where λ is a scalar. We can understand this model as mix between the svd score $s' - s$ and a local Most Frequently Used (MFU) score $\lambda.s$. We tried different values for λ, the results can be observed in figure 7.2.

From figure 7.2 we decided to use $\lambda = 2$ in all our future experiments. We also tried several values for the number of features k. From figure 7.3, it is clear that the best number of feature is 1. This is a bit odd since it means that one value optimally encode each sequence and each application, we will get back to this point later. Figure 7.4 compares the svd recommendation to MRU.

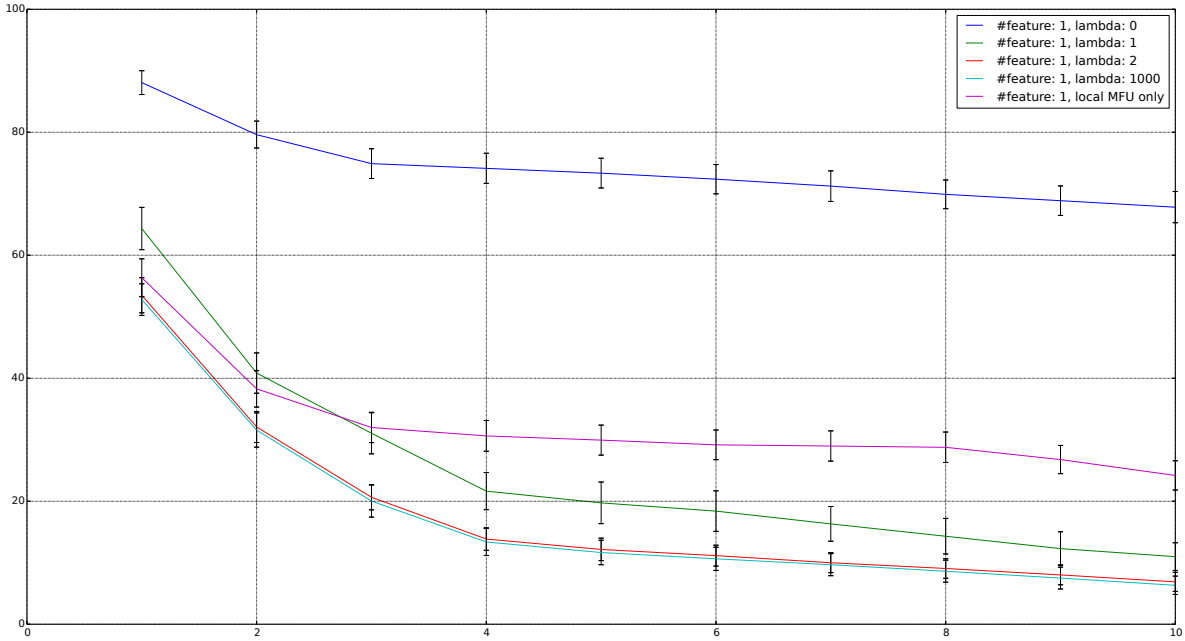

Figure 7.2 – Error ratio for different values of λ. The x-axis is the size of the prediction K. All the curves are shown for $k = 1$ i.e. one feature only. We also show the score for the local MFU method only, we can observe that for $\lambda > 1$ the combination of svd and local MFU gives better results than the two methods alone.

Seeing figure 7.4 we finally sensibly defeated the MRU baseline. In the next section we will explain what is really happening and what lessons we can get from those results.

7.4 MFU-global, MFU-local, and k-SVD

We showed that the best results where obtained for $k = 1$ feature and $\lambda = 2$. With these parameters our model is:

$$score = s' + s$$

As before s is the MFU local score, it pushes the model to recommend applications that are frequent in that sequence. If we look at s', we know:

$$s' = U_{train}.U_{train}^{t}.s$$

Since $k = 1$, U_{train} is a vector of length the number of applications $|L|$. Furthermore $U_{train}^{t}.s$ is a scalar corresponding to the projection of the applications of s onto the item space U. In the end U can be seen as a score for all applications, if we look at the values of U we find a ranking very close to the global MFU score. In the end our predictive model for $k = 1$ can be

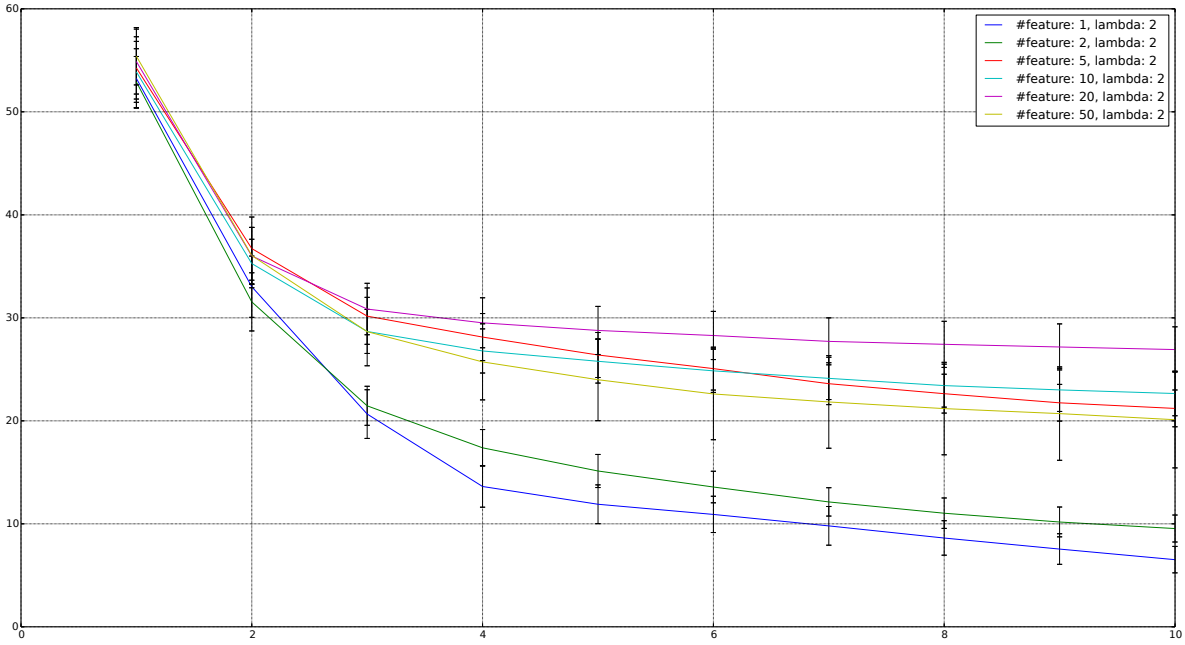

Figure 7.3 – Score of our predictive model with $\lambda = 2$ for different number of features.

approximated by:

$$score = MFU_{global} + MFU_{local}$$

Figure 7.5 shows how close the two models are from one another. The SVD/MFU-local method is slightly better than the MFU-global/MFU-local one.

Let's review what we did until now, we tried neighbor based methods and they are quite efficient. Neighbor based methods will tend to recommend applications present in close sequences and therefore generate a score certainly very close to MFU-local. We tried to use LDA to cluster our applications sequences in terms of topics, as conclusion we saw how similar each topic are and how they all are also very close to MFU-global. The conclusion of the LDA methods was that all the sequences are based on the same small set of applications very often used and differ only on less used applications. The MF methods developed here tends to confirm this observation since the best results are obtained with $k = 1$ which is nearly equivalent to mix MFU-global and MFU-local.

The success of MFU-global/MFU-local shows the applications already used in the sequence are very likely to be used again. This method also confirms the sequence approach that has been decided for all our algorithms, applications used in a short window of time are indeed correlated. There are not much correlation between consecutive sequences since MFU-global/MFU-local seems better than MRU for large prediction sizes K.

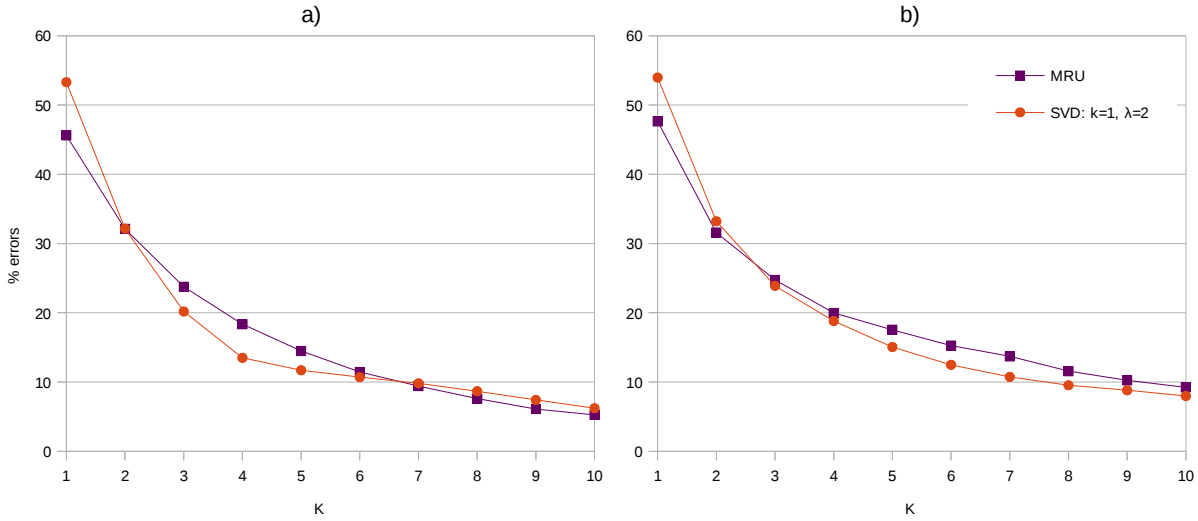

Figure 7.4 – SVD recommendation and MRU for the default user (a) and averaged over all users (b).

Finally we can see how to make an even better method combining all that we just learn. The two first elements recommended should be w_{t-1} and w_{t-2} since we have never beaten MRU for $K = 1$ and $K = 2$. We can then switch to SVD/MFU-local for $K > 2$.

7.5 Exploiting the time coherency

So we learn to recommend applications that already occurred in the sequence in priority. We tried to use MFU, however MFU-local is the method using the least amount of knowledge. We propose a hierarchical recommendation pipeline as follow:

- For the two first applications to recommend, use MRU i.e. recommend w_{t-1} then w_{t-2} in priority.
- If in the beginning of the sequence, $s_{[0:t-1]}$, we have local bigram data for w_{t-1}, we use this information in priority.
- If we still need applications to predict we can use MFU-local.
- Finally, we can use global-bigrams to complete the prediction.

The result for this MRU>local-bigrams>MFU-local>global-bigrams pipeline can be seen in figure 7.6.

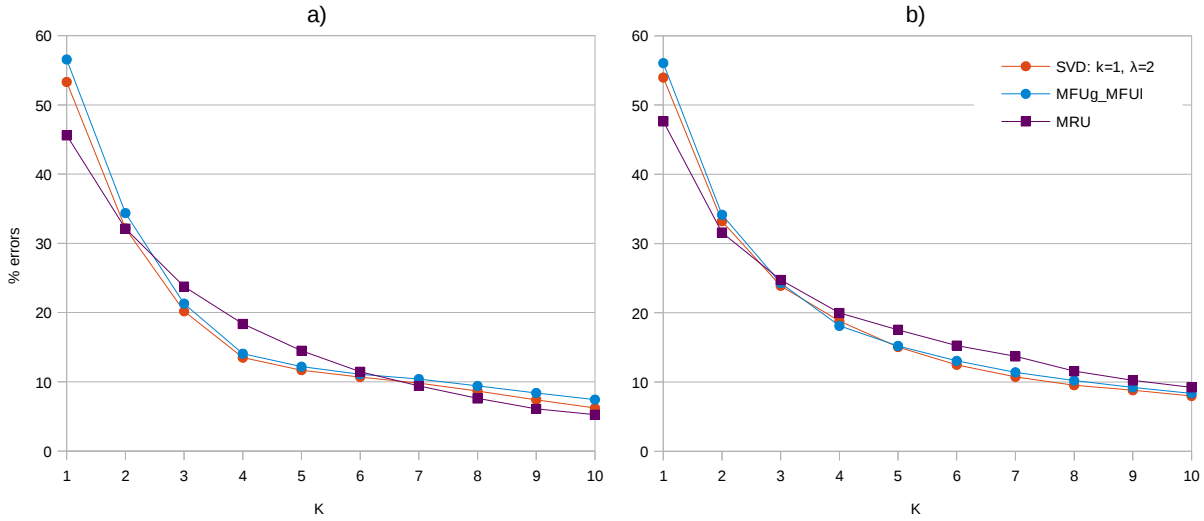

Figure 7.5 – SVD recommendation, MFU-local/MFU-global and MRU for the default user (a) and averaged over all users (b).

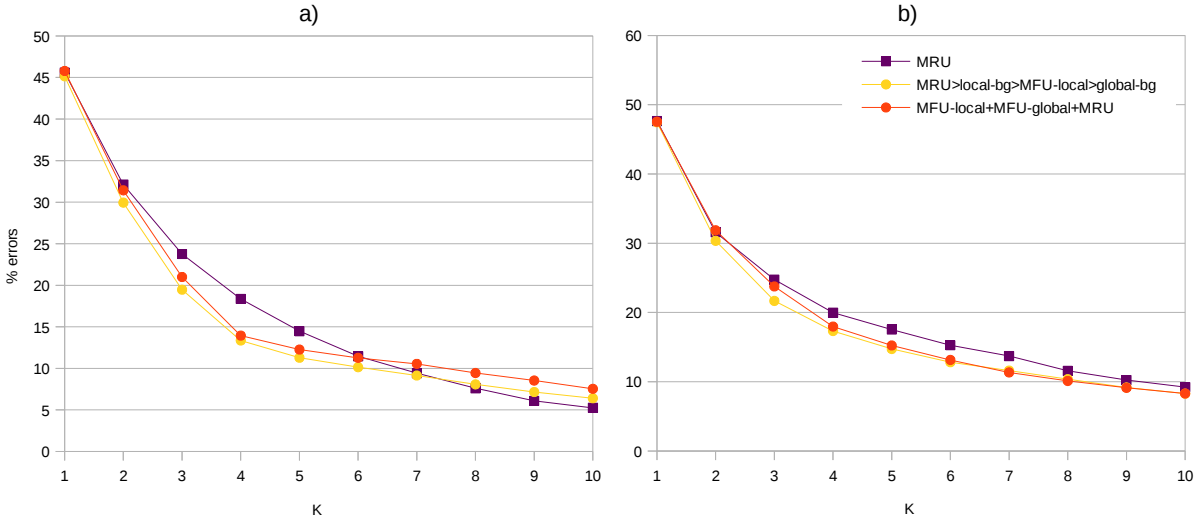

Figure 7.6 – Hierarchical pipeline and MRU for the default user (a) and averaged over all users (b).

8 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are powerful algorithms to work on sequences. They can learn correlation in a dataset very efficiently. We tried a specific type of RNN, Long-Short-Term Memory (LSTM) RNN. We will start by briefly introduce NN, and then describe our setup.

8.1 Introduction to Neural Networks

A very common problem of machine learning is classification. We are given a set of input/target pairs:

Input value x	Input value y	Output label t
1.5	1	+1
1	1	+1
1	2	+1
1	3	-1
2	1	-1
2	2	-1

In this example we have two possible output classes -1 and $+1$ so this is a binary classification problem. Each input has two features, x and y. The goal is to learn a function that takes a 2-dimensional vector and predicts the label. In the next section we will see how to parametrize this function using a set of parameters. The training algorithm will tune the parameters to make the function consistent with the dataset we have. Finally the trained model can be use to classify previously unseen vectors.

8.1.1 An optimization problem

Let's consider the following simple function $f(x,y) = w_0.x + w_1.y + w_2$, this formula is the parametrization of a line in the 2D plane. We want to use this function to classify our data,

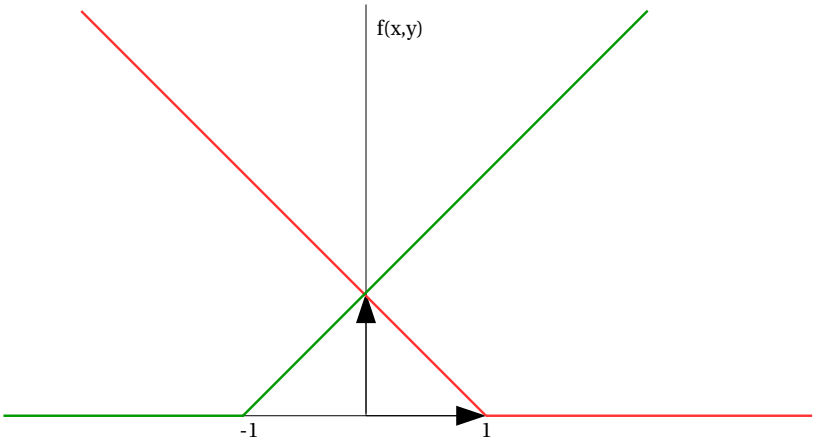

Figure 8.1 – Hinge error function for $t = -1$ in green and for $t = +1$ in red. When the data is correctly classified $E_{Hinge} = 0$.

for each input pair (x, y) if $f(x, y) > 0$ then the label associated to the input is $+1$ else it is -1. We aim to have $f(x, y).t > 0$ for all input data of our dataset. This is equivalent to ask for a line parametrized by the weights (w_0, w_1, w_2) that would split the dataset optimally. Indeed, for (x, y) belonging to the line then $f(x, y) = 0$, on one side $f(x, y) < 0$ and on the other side $f(x, y) > 0$.

To help us in this task we define the Hinge error metric:

$$E_{Hinge}(x, y, t) = max(0, 1 - t.f(x, y))$$

The Hinge error function can be seen in figure 8.1. The error value is high when the input is strongly misclassified.

The problem is now to reduce the error as much as possible for the curve to fit our data. This is a regression problem. One simple way to reduce the error is to use gradient descent. The gradient tells us the direction of largest increase, moving in the opposite way of the gradient can bring us "downhill" and reduce the value of the function. In our case the partial derivatives are very simple:

$$\frac{\partial E_{Hinge}(x, y, t)}{\partial w_0} = -t.x$$
$$\frac{\partial E_{Hinge}(x, y, t)}{\partial w_1} = -t.y$$
$$\frac{\partial E_{Hinge}(x, y, t)}{\partial w_2} = -t$$

We can now iteratively move in the opposite direction of the gradient to minimize the error. However the gradient is a local property and locality is a very relative notion. We introduce a

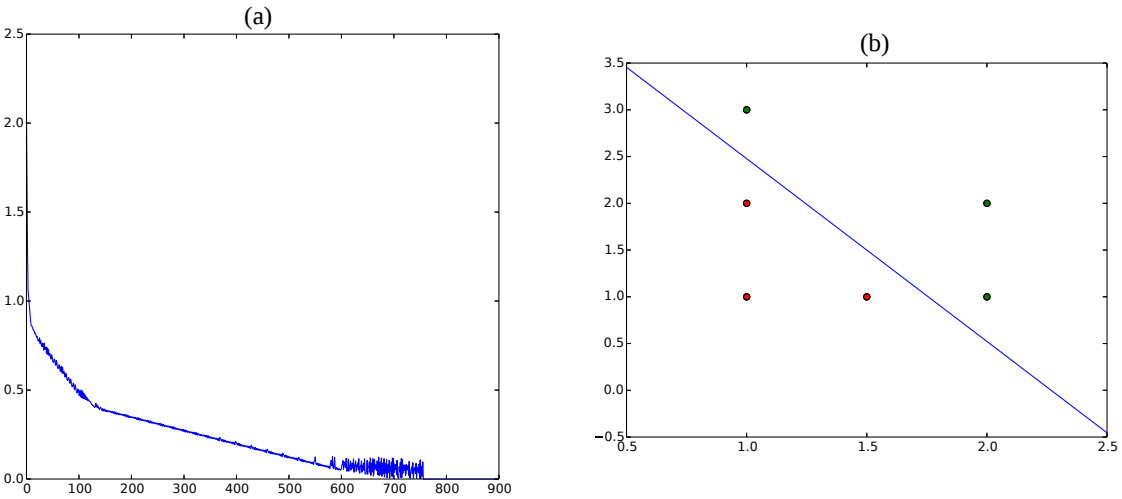

Figure 8.2 – Variation of the error E_{Hinge} over 900 epochs (a), the weights w_0, w_1, w_2 are initialized at random. Dataset and splitting line obtained after gradient descent drawn in the 2D plane (b).

learning rate lr which represent the size of the step we are going to make in one direction. If the learning rate is too large then we might increase the value of the error when updating the weights, if it is too small the convergence will be very slow. We can now design an update rule to iteratively minimize the error for one input sample:

$$w_0 = w_0 + lr.t.x$$
$$w_1 = w_1 + lr.t.y$$
$$w_2 = w_2 + lr.t$$

We can define a better update rule by averaging the gradient over all the dataset. Figure 8.2 present the result of this procedure over 900 epoch. In the literature the action of computing the output for one input sample is called forward pass, the action of propagating the derivative of the error from the output to the input is called backward pass. The simple application of the chain rule to do gradient descent is called the back-propagation algorithm. This terminology will become clear when we will introduce the network representation of the NN (figure 8.3). Gradient descent is a very simple procedure, in most cases it does not guaranty us to converge to a global minimum. For that reason different starting points might give very different results, the initial weights are chosen at random. The curve obtained after convergence can be seen in figure 8.2.

Expressing a classification problem as a regression problem with a certain error metric, and using gradient descent to minimize this error, this is the main idea behind NN. Many error metrics exists, among the most used ones are the squared error, logistic error and Hinge error.

If the dataset is representative enough, we can use the trained classifier efficiently classify new input points. The system we just design can be represented as shown in figure 8.3.

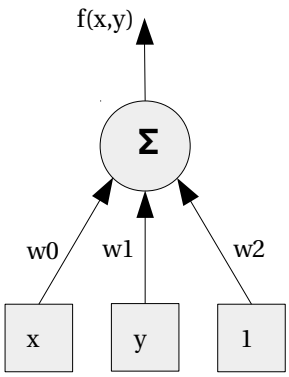

Figure 8.3 – Representation of the classifier as a network. The forward pass compute the output starting from the input, the back-propagation algorithm is the action of going from bottom to top and then from top to bottom propagating the error derivative downwards.

8.1.2 Multi-Layer Perceptron

The method used above is great but everything is linear. In practice it is very often that the boundary of our classifier will not be just a line but an arbitrary curve. To introduce a non-linearity we introduce a squashing function on top of our model. The aim of the squashing function is to bring the output in a certain interval like $[0,1]$ (hence the squashing) as well as to introduce a non linearity that is easily differentiable. Widely used squashing function are $tanh$ and the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-t}}$$

The addition of a squashing function on top of our previous system constitute a standard neurone as shown in figure 8.4. If the previous section was clear then this one should be no problem, a standard neural network is just a bunch of neurones. The neurones are presented layer by layer, figure 8.5 shows a 2-layers NN. Each layer of neurone beside the last one is called a hidden layer since it is hidden inside the network and the values leaving the neurones are not observed. Several layers NN are also called multi-layer perceptrons. Computing the partial derivatives for all the weights might seems complicated, however we can easily use the chain rule of derivation and combine the simple gradient expression of each of the neurones of the network. Each weight is updated by moving in the opposite direction of the gradient to minimize a certain error metric.

NN find application not only for classification tasks but anywhere we need to learn a function

from any input space to any output space. In the example of figure 8.5 the output is of dimension 2. Each neurone linearly combine the input before squashing it, and creates intermediary representations of the data in its hidden layers. Those intermediary representation can be though of as features, in that sense a NN is a tool to discover features in data.

To predict applications we could use a multi-layer perceptron. The input could be the vector representation of an application and the associated target would be the vector representation of the next application. For example if in our dataset we got the following sequence of applications A, A, B, C we can translate it to the following input/label $([1,0,0],[1,0,0])$, $([1,0,0],[0,1,0])$ and $([0,1,0],[0,0,1])$. Using the procedure described before we can try to reduce the error as good as possible for our dataset. For any input vector we will then have a tri-dimensional output vector, the value of its components can be interpreted as a score for each application, the larger the score is the better. If each time we normalize the output vector we can interpret the output as a probability distribution of the output applications given the input application. This way we could train a NN to learn a bigram model and predict for one application which one is the most likely to be used next.

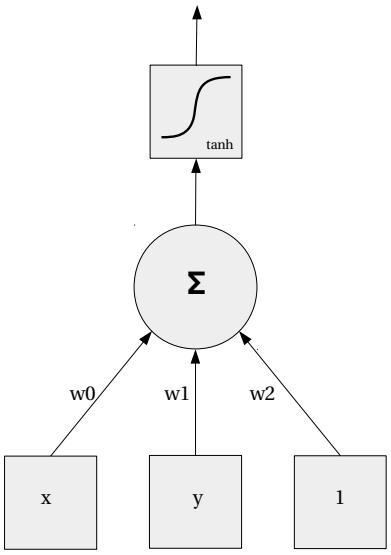

Figure 8.4 – Architecture of a standard neurone.

8.1.3 Recurrent Neural Networks

If we want a bigram model using NN we can use a simple multi-layer perceptron, if we want a trigram model we can concatenate the vector representation of the two previous applications. What if we want a N-gram model? We could concatenate the vector representation of the $N-1$ last applications but the complexity would increase linearly with respect to N. Even worst, we would have to pre-process our sequences to generate all the possibles N-grams increasing the

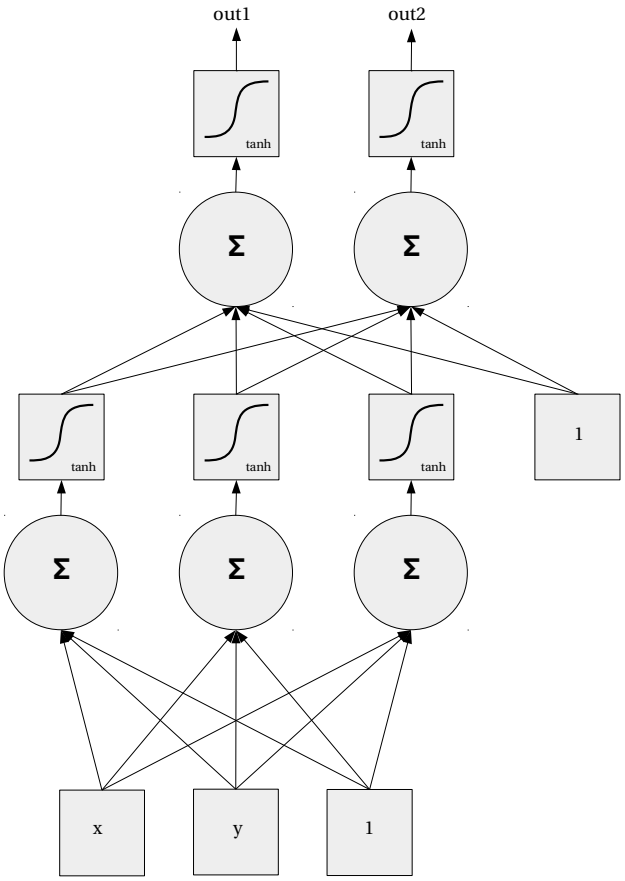

Figure 8.5 – Architecture of a 2 layers NN. The first layer is a hidden layer of size 3. The second layer is an output layer of dimension 2. The "1" blocks are introduced to represent the bias (the constant in the equation of the hyperplanes).

size of the dataset. To answer this problem we introduce Recurrent Neural Networks (RNN).

RNN are like standard NN with a time dimension. Instead of presenting each input to the network independently, we present sequences. Concretely if we take the sequence A, A, B, C we will present to the network first the representation of A, then successively the one of A, B and C. That way the size of the network does not increase and we can consider sequences of any length.

As described now the network is still just a simple NN, what makes it recurrent and able to process sequences is the presence of recurrent connexions on the neurones. The output of a recurrent neurone is fed back as input of that neurone. The recurrent connexions are described in figure 8.6. That way the order we follow to present each input of the sequence to the network matters. In order to train such a system we need as before to compute the partial derivative with respect to the weights. This step is a bit more complicated than before since we need to take into account the recurrent connexions. We can however unwrap the network

through time as shown in figure 8.6 and then handle the derivation as a standard NN. The back-propagation algorithm for RNN is called back-propagation through time.

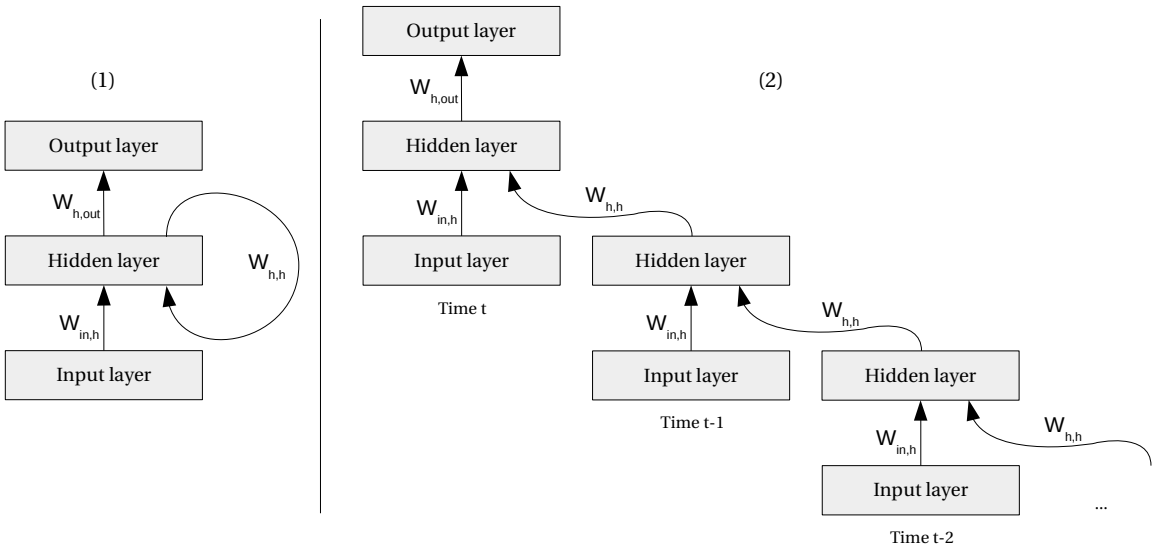

Figure 8.6 – (1): block representation of a RNN, the recurrent connexion loops the output of the hidden layer to its input, making the signal loop in the network allows the creations of temporal states. (2): The network can be unwrapped through time, once the network is unwrapped it can be handle as a simple neural network.

During the process of presenting the applications of a sequence to the RNN, it will create internal states that are going to evolve with time. In the case of smartphone application prediction, those time relative features could hopefully allow us to accurately represent which application is likely to be used next.

RNN are powerful, yet they suffer one major drawback. Looking at figure 8.6 we can see how the past inputs are multiplied over and over by the same weights $W_{h,h}$. depending on the value of the weights this will exponentially increase or decrease the influence of the past inputs. This problem is known as the vanishing gradient problem, the gradient during the back-propagation through time vanishes exponentially. For this reason simple RNN cannot handle long sequences or deep architecture (more than two hidden layers).

8.1.4 Long-Short-Term Memory Neural Networks

Long-Short-Term Memory (LSTM) RNN answer the vanishing gradient problem [10] by replacing all neurones by LSTM cells. LSTM cells are like latches, they keep information intact inside, they can free this information or reset its value. A standard LSTM cells is depicted in figure 8.7. The blue input gate receive the signal from the rest of the network, this signal will be modulated by the green input gate. If this input gate gives an output strongly negative then

the squashing function will yields a value close to 0 and the input signal will not enter the cell i.e. the cell is closed. The same mechanism is implemented with the output gate which allow the signal to leave the cell or not. The constant error flow is assured by the Constant Error Carousel (CEC) which consists in a recurrent connexion with a weight of 1. The CEC allows the cell to store the information for an indefinite amount of time solving the vanishing gradient problem. The CEC is modulated by the output of a forget gate. Without the forget gate, for large sequences, since the input gate always allows some information, even small amounts, to enter the cell, the value stored in the cell would increase continuously. The forget gate compensate this effect allowing the CEC value to be modulated.

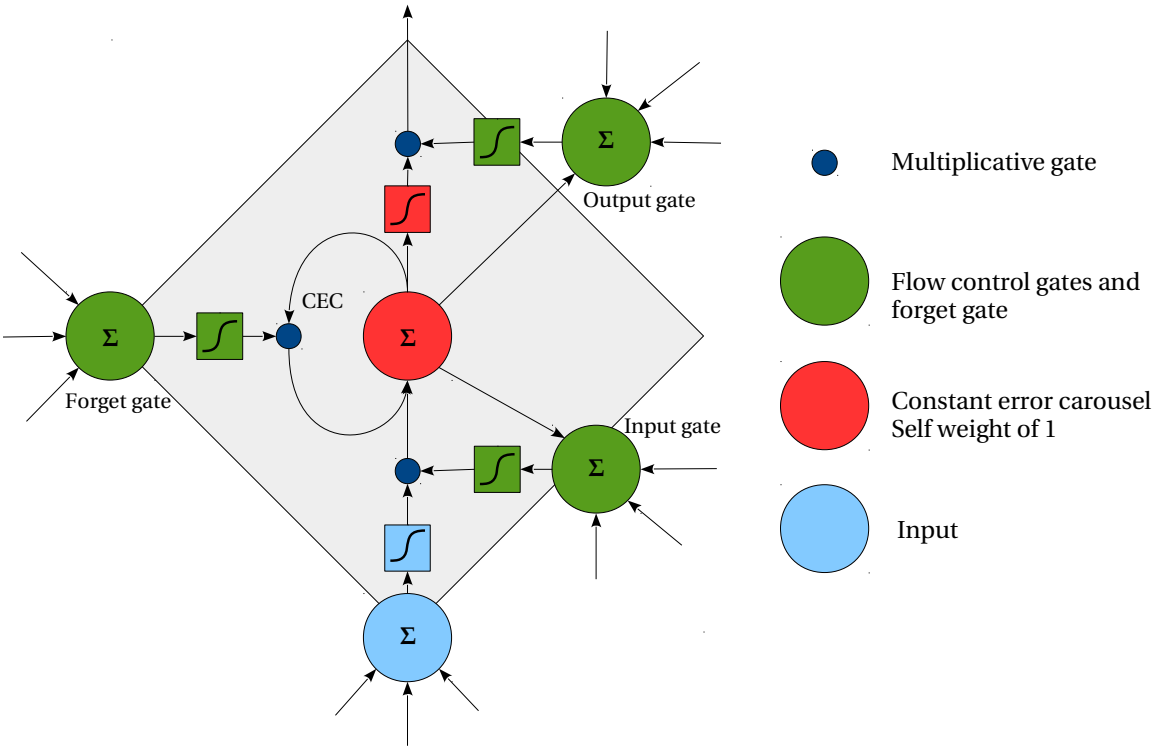

Figure 8.7 – Architecture of a LSTM cell. The input and output gates decide if the information is entering or leaving the cell. The Forget gate allows the cell to compensate for the information coming at all times from the input.

Just as before, we train a network made of LSTM cells using gradient descent. The partial derivatives of the error with respect to the weights are computed on the unwrapped network.

8.2 LSTM networks for smartphone application prediction

The library we used that implements LSTM RNN is RNNlib [11]. As stated in previous chapters one application can be considered as a vector of size $|L|$ (the total number of applications) containing a 1 only at the index associated with this application. This is the 1-hop representa-

tion, if we have 5 applications in total and the application A is associated to the index 3 then its 1-hop representation is $[0,0,1,0,0]$. Using this representation we can express a sequence s of application as a sequence of 1-hop representations. Those sequences can be fed to a LSTM RNN with a softmax output layer of size $|L|$ to try to learn the probability distribution over all the applications given a sequence.

The success of a NN application depends on the ratio of the difficulty of the learning task with the size of the data we have. The more data we have the more we can learn, the data we need in order to learn also increase with the task complexity. The problem with our current representation is the noisiness of the input. We more than once mentioned the noisiness of the dataset, asking the network to learn from that noisy representation, for the amount of data we have, will very likely result in the back-propagation through time algorithm falling in a bad local extrema. To prevent this we can do some pre-learning on our data to make the learning task easier.

We mentioned in chapter 8 that compressing data was exactly the same as learning, so this is what we are going to use, we use k-SVD to reduce the dimensionality of the 1 hop representations from $|L|$ to k. Doing this smooths the input and the learning becomes easier. After several tests $k = 10$ features gives us the best results. The best results were obtained with a 2 layers LSTM RNN of sizes 50 and 5. Figure 8.8 shows the structure of the network.

The network seems to learn MRU for $K = 1$ and $K = 2$ prediction. The results are better for the default user than averaged on all users. One reason might be the number of training sequences we have for the default user is larger than for the average user. This let us hope that LSTM would gives good results if we had larger training sets. This also seems to confirm that we cannot do really better than MRU for $K = 1$ and $K = 2$.

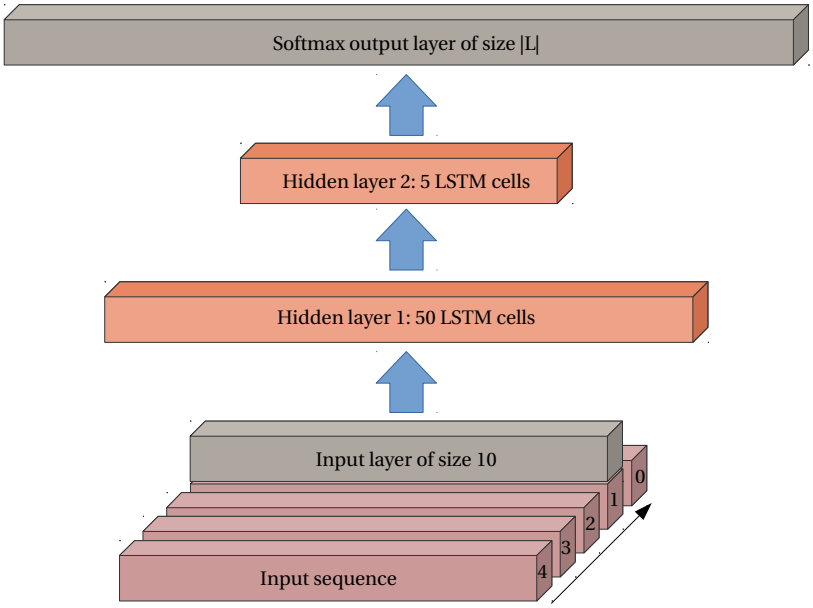

Figure 8.8 – Topology of the network. The input sequence (here of size 5) is presented application after application to the network. Each sub-vector of the input sequence is the feature representation of one application reduced to 10 dimensions. The output layer consist in a softmax layer, the values of the $|L|$ outputs form a probability distribution of each app to be the next app of the sequence knowing the past apps.

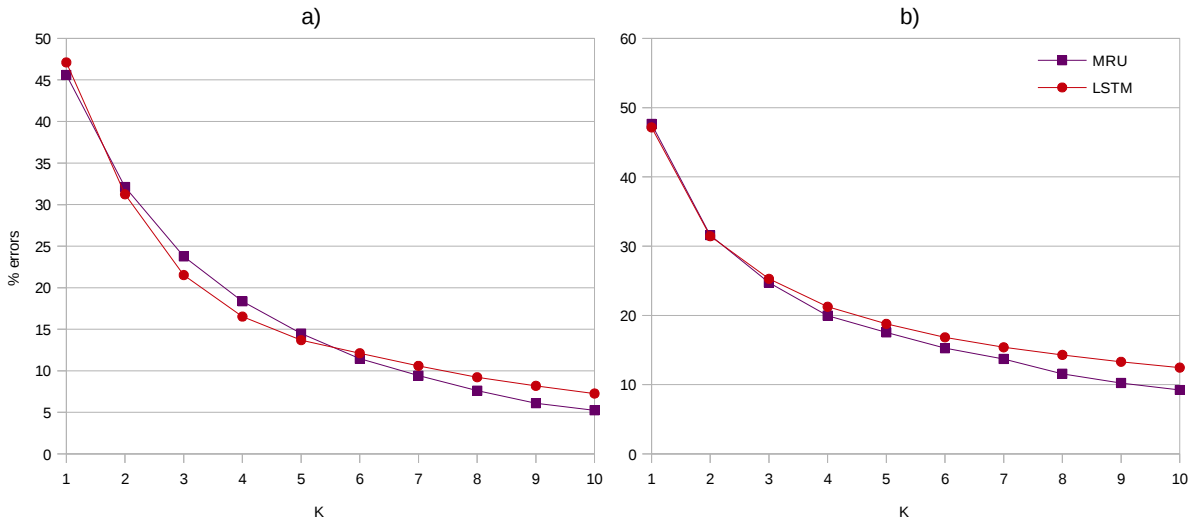

Figure 8.9 – LSTM method using pre-learning vs MRU for the default user (a) and averaged over all users (b).

Conclusion

The main contribution of this thesis has been to explore if splitting the large user sequences into subsequences was relevant to predict smartphone applications. It seems that this hypothesis is verified, applications close in time are strongly correlated. We saw how the information we get from the beginning of a sequence prevail over global statistics. This pushes us to create hierarchical pipelines like the MRU>local-bigrams>MFU-local>global-bigrams, aiming to exploit all the local information available before using global statistics. We also showed that the sequences do not contain much more information than the bag of words of that sequence, at least our small dataset combined with several sequence modeling methods did not performed exceptionally better and often worse than bag of words methods. It is possible that we are lacking of discriminative features to correctly define the user behavior. For that reason, adding new features like the GPS coordinates, the reachable wifi networks, or using contact information might improve the results.
For future exploration I recommend to try combining more features. One could imagine a pipeline where a first ranking of probable application is created conditioned over all possible features, then, as soon as we have information for the current sequence, that information should be used to refine the score since local information (local means in the same time window) is much more relevant than global statistics.

Bibliography

[1] Google glass: http://www.google.de/glass/start/.

[2] Project tango: https://www.google.com/atap/projecttango/project.

[3] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and MichaelI. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[4] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, pages 43–52, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The bellkor solution to the netflix prize.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[7] Katayoun Farrahi and Daniel Gatica-Perez. Discovering routines from large-scale human locations using probabilistic topic models. *ACM Trans. Intell. Syst. Technol.*, 2(1):3:1–3:27, January 2011.

[8] Katayoun Farrahi and Daniel Gatica-Perez. Extracting mobile behavioral patterns with the distant n-gram topic model. In *Proceedings of the IEEE International Symposium on Wearable Computers*, June 2012.

[9] Katayoun Farrahi and Daniel Gatica-Perez. A probabilistic approach to mining mobile phone data sequences. *Personal and Ubiquitous Computing*, December 2012.

[10] Felix Gers. Long short-term memory in recurrent neural networks, 2001.

[11] Alex Graves. Rnnlib: A recurrent neural network library for sequence learning problems. http://sourceforge.net/projects/rnnl/.

[12] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[13] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[14] Philip Resnik and Eric Hardisty. Gibbs sampling for the uninitiated. *Technical report, University of Maryland*, 2010.

[15] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, June 2009.

[16] Xuerui Wang and Andrew Mccallum. A note on topical n-grams. Technical report, University of Massachusetts, 2005.

[17] Xuerui Wang, Andrew McCallum, and Xing Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, pages 697–702, Washington, DC, USA, 2007. IEEE Computer Society.

Paths to experiments

Figure 3.1:
bigrams: `baselines/bigrams/`
MRU: `baselines/MRU/`
MFU: `baselines/bigrams/`

Figure 4.1:
default user, Jaccard: `Neighbor_Based_methods(MB)/experiments/jaccard_defaultUser`
default user, Cosine: `Neighbor_Based_methods(MB)/experiments/cosine_defaultUser`

all users, Jaccard: `Neighbor_Based_methods(MB)/experiments/jaccard_allUsers`
all users, Cosine: `Neighbor_Based_methods(MB)/experiments/cosine_allUsers`

Figure 5.3:
default user, LDA: `LDA_based_methods/LDA/experiments/14_11_07_BasicPipeline`
default user, LDA lower bound: `LDA_based_methods/LDA/experiments/14_11_07_lower_Bound`
default user, LDA oracle: `LDA_based_methods/LDA/experiments/14_11_07_oracle`

all users, LDA: `LDA_based_methods/LDA/experiments/14_11_07_BasicPipeline_allUsers`
all users, LDA lower bound: `LDA_based_methods/LDA/experiments/14_11_07_lower_Bound _allUsers`
all users, LDA oracle: `LDA_based_methods/LDA/experiments/14_11_07_oracle_allUsers`

Figure 5.5:
default user, DNTM seq. size 4: `LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/severalSequences_Approach /14_10_21_basicPipeline_defaultUser_seqSize4_smallParam`
default user, DNTM seq. size 5: `LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/severalSequences_Approach /14_10_21_basicPipeline_defaultUser_seqSize5_smallParam`
default user, DNTM seq. size 5 lower bound: `LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/severalSequences_Approach /`

14_10_21_lowerBound_defaultUser_seqSize5_smallParam
default user, DNTM seq. size 5 oracle: LDA_based_methods/Distant_Ngram_TM(DNTM)/
experiments/severalSequences_Approach /14_10_21_oracle_defaultUser_seqSize5_smallParam
default user, DNTM seq. size 15: LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/
severalSequences_Approach /14_10_21_basicPipeline_defaultUser_seqSize15_smallParam

all users, DNTM seq. size 4: LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/
severalSequences_Approach /14_10_21_basicPipeline_allUsers_seqSize4_smallParams
all users, DNTM seq. size 5: LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/
severalSequences_Approach /14_10_21_basicPipeline_allUsers_seqSize5_smallParam
all users, DNTM seq. size 5 lower bound: LDA_based_methods/Distant_Ngram_TM(DNTM)/
experiments/severalSequences_Approach /14_10_21_lowerBound_allUsers_seqSize5_smallParam
all users, DNTM seq. size 5 oracle: LDA_based_methods/Distant_Ngram_TM(DNTM)/
experiments/severalSequences_Approach /14_10_21_oracle_allUsers_seqSize5_smallParam
all users, DNTM seq. size 15: LDA_based_methods/Distant_Ngram_TM(DNTM)/experiments/
severalSequences_Approach /14_10_21_basicPipeline_allUsers_seqSize15_smallParam

Figure 5.6:
default user, DNTM seq. size 15 approach 1: LDA_based_methods/Distant_Ngram_TM(DNTM)/
experiments/oneSequence_approach/
17_11_21_basicPipeline_defaultUser_seqSize15_oneSeqApproach_smallParam

all users, DNTM seq. size 15 approach 1: LDA_based_methods/Distant_Ngram_TM(DNTM)/
experiments/oneSequence_approach/
17_11_21_basicPipeline_allUsers_seqSize15_oneSeqAppr_smallParam

Figure 5.8:
default user, TNG: LDA_based_methods/Topical_Ngram(TNG)/true_TNG/
14_10_20_StandardPipeline_trueIntegration
default user, LDA bigram: LDA_based_methods/Topical_Ngram(TNG)/LDA-bigrams/
14_10_20_StandardPipeline_xSetTo1_zprewSetToz
default user, LDA bigram lower bound: LDA_based_methods/Topical_Ngram(TNG)/
LDA-bigrams/14_10_20_LowerBound_xSetTo1_zprewSetToz
default user, LDA bigram oracle: LDA_based_methods/Topical_Ngram(TNG)/
LDA-bigrams/14_10_20_Oracle_xSetTo1_zprewSetToz

all users, TNG: LDA_based_methods/Topical_Ngram(TNG)/true_TNG/
14_10_20_StandardPipeline_allusers_trueInteg
all users, LDA bigram: LDA_based_methods/Topical_Ngram(TNG)/LDA-bigrams/
14_10_20_StandardPipeline_xSetTo1_zprewSetToz_allusers
all users, LDA bigram lower bound: LDA_based_methods/Topical_Ngram(TNG)/

LDA-bigrams/14_10_20_LowerBound_xSetTo1_zprewSetToz_allusers
all users, LDA bigram oracle: LDA_based_methods/Topical_Ngram(TNG)/
LDA-bigrams/14_10_20_Oracle_xSetTo1_zprewSetToz_allusers

Figure 5.9:
default user, LDA trigrams: LDA_based_methods/LDA_trigrams/experiments/
14_10_20_basicPipeline_BigParams
default user, LDA trigrams lower bound: LDA_based_methods/LDA_trigrams/
experiments/14_10_20_lowerBound_bigParams
default user, LDA trigrams oracle: LDA_based_methods/LDA_trigrams/
experiments/14_10_20_oracle_BigParams

all users, LDA trigrams: LDA_based_methods/LDA_trigrams/experiments/
14_10_20_basicPipeline_BigParams_allusers
all users, LDA trigrams lower bound: LDA_based_methods/LDA_trigrams/
experiments/14_10_20_lowerBound_bigParams_allUsers
all users, LDA trigrams oracle: LDA_based_methods/LDA_trigrams/
experiments/14_10_20_oracle_BigParams_allusers

Figure 5.10:
default user, LDA loose trigrams: LDA_based_methods/Loose_LDA_Trigram/
experiments/14_10_07_standardPipeline_w5
default user, LDA loose trigrams lower bound: LDA_based_methods/
Loose_LDA_Trigram/experiments/14_10_07_lowerBound_w5
default user, LDA loose trigrams oracle: LDA_based_methods/Loose_LDA_Trigram/
experiments/14_10_07_oracle_w5

all users, LDA loose trigrams: LDA_based_methods/Loose_LDA_Trigram/experiments/
14_10_07_standardPipeline_w5_allusers
all users, LDA loose trigrams lower bound: LDA_based_methods/Loose_LDA_Trigram/
experiments/14_10_07_lowerBound_w5_allusers
all users, LDA loose trigrams oracle: LDA_based_methods/Loose_LDA_Trigram/
experiments/14_10_07_oracle_w5_allusers

Figure 6.1:
CRF: Conditional_Random_Field_methods(CRF)/experiments/verySmallParam

Figure 7.2 and figure 7.3:
script: Matrix_Factorization_methods(MF)/experiments/parameters_lmbda_k_curves

Figure 7.4:
default user, MF: `Matrix_Factorization_methods(MF)/experiments/defaultUser_normalized`
all users, MF: `Matrix_Factorization_methods(MF)/experiments/allUsers_normalized`

Figure 7.5:
MFU-local/MFU-global: `baselines/MFUl_MFUg`

Figure 7.6:
MRU>local-bigrams>MFU-local>global-bigrams: `developed after Sony, send email to: matteo.pagliardini@gmail.com`

Figure 8.9:
LSTM: `Long_Short_Term_Memory_RNN_methods`