Robust Dictionary Lookup in Multiple Noisy Orthographies

Lingliang Zhang, Nizar Habash and Godfried Toussaint
New York University Abu Dhabi
Abu Dhabi, UAE
`{lingliang.zhang,nizar.habash,gt42}@nyu.edu`

Abstract

We present the MultiScript Phonetic Search algorithm to address the problem of language learners looking up unfamiliar words that they heard. We apply it to Arabic dictionary lookup with noisy queries done using both the Arabic and Roman scripts. Our algorithm is based on a computational phonetic distance metric that can be optionally machine learned. To benchmark our performance, we created the ArabScribe dataset, containing 10,000 noisy transcriptions of random Arabic dictionary words. Our algorithm outperforms Google Translate's "did you mean" feature, as well as the Yamli smart Arabic keyboard.

1 Introduction

The research effort reported here was motivated by the experience of second-language learners of Arabic who, upon hearing an unfamiliar word, would repeatedly guess different spelling variations until they either give up or find a word that made sense in the context. In the data we collected, subjects were only able to spell an Arabic word correctly 35% of the time. This difficulty arises because many Arabic phones are distinguished from each other only by phonetic features that often do not exist in the learner's native language, such as pharyngealization. Furthermore, for a variety of reasons, both learners and native speakers often write Arabic words imprecisely using a variety of Roman-script conventions, such as Arabizi (Darwish, 2014), and this presents a difficult problem for disambiguating the intended word.

We address this problem with a novel algorithm MultiScript Search for phonetic searching across multiple orthographies, and apply it to performing dictionary lookup of Arabic words from noisy user inputs in both the Arabic and Roman scripts. The task of dictionary lookup here is defined as a user hearing a single word and typing their best guess of how it sounds in a system that will look the word up in a dictionary. An example of how users searched for the word تضخيم (tDxym)[1] 'magnification' is given in Table 1.

Target word: تضخيم (tDxym)			
طبخين (Tbxyn)	طبحيم (TbHym)	تبخيم (tbxym)	تبخين (tbxyn)
thabheem	Tat7iim	tohpriim	tawbheem
ta'95eem	topk'chim	takreem	tabheem

Table 1: Example of the Arabic dictionary lookup task for both Arabic and Roman script.

Our phonetic search algorithm can be generally applied to any language-script pair and does not require training data. But, we also present a machine learning method to boost the performance of our algorithm if training data is available. The algorithm first performs mapping from user input to phones, and then it searches in the dictionary for words that have a low phonetic distance from the query. We investigate a number of algorithms for both the grapheme-to-phoneme mapping, and also for calculating phonetic distance.

To benchmark our lookup accuracy, we created the ArabScribe dataset, which is comprised of almost 10,000 transcriptions from 103 participants with different degrees of knowledge of Arabic. The participants heard random Arabic dictionary words and transcribed them using either the English or Arabic keyboards (i.e., in Roman script

[1]Arabic transliteration is presented in the Habash-Soudi-Buckwalter scheme (Habash et al., 2007).

or in Arabic script). We benchmarked our system against two widely used tools to look up unfamiliar Arabic words, Google Translate and the Yamli Smart Arabic Keyboard. We show that we exceed the performance of both tools.

The paper will proceed as follows. In Section 2 we discuss the related literature in transliteration, spell correction and phonetic distance mapping. In Section 3 we describe our high level algorithm and the variations we tested. In Section 4 we describe the ArabScribe dataset. Section 5 presents our search recall results and our benchmark against the Google and Yamli systems. In Section 6, we summarize our findings and present a discussion on potential future work.

2 Related Work

The principal contribution and distinction of our work against previous work in the related areas of transliteration and spelling correction is that we are concerned with recovering from hearing errors rather than errors arising from spelling or ambiguous orthography. To our knowledge, there has not been any empirical research into the accuracy of the task of phonetic dictionary lookup of spoken words.

Spelling Correction Within a single orthography, a closely related problem is spellchecking. Commonly used algorithms employ statistical and edit-distance models over letters, phones or metaphone (Whitelaw et al., 2009; Kukich, 1992; Damerau, 1964; Oflazer, 1996; Atkinson, 2005; Philips, 1990; Toutanova and Moore, 2002). Our algorithm is distinguished from these, in that we are not only addressing the case where the user doesn't know how to spell the word, but the much more challenging case where they have not heard the sound correctly.

Whitelaw et al. (2009) describe the Google spell check system which takes a statistical approach using massive unannotated web corpora. Their dictionary of canonical terms contains the most common tokens appearing online, and they match misspelled words to their canonical form using a combination of a language context and Levenshtein edit distance. Then, they build a statistical model of how substrings change. Spelling suggestions are scored as a function of the probability of the substring substitutions combined with a language model. This statistical approach has the advantage that it requires no formal language definition, instead requiring a large amount of linguistic data and computational resources. While it is a very effective spell checker, it does not perform well on noisy user input of guessed spellings of unfamiliar words as it is trained on web corpora which are written by people who have a generally stronger command of the language and therefore errors are often due to typos rather than not knowing how to spell the word. There is unfortunately no published material about the exact method used by Google's transliteration or Yamli smart keyboard to map Roman-script input into Arabic words.

In the context of spelling correction for Arabic, there has been a large number of efforts and resources (Shaalan et al., 2010; Alkanhal et al., 2012; Eskander et al., 2013; Mohit et al., 2014; Zaghouani et al., 2014) (among others).[2] All of these efforts focus on contextual spelling correction or conventionalization. In this work we contribute a new data set, ArabScribe, that may be of use in that area of research.

Transliteration Transliteration systems invariably use either a direct mapping between substrings in the two scripts (Al-Onaizan and Knight, 2002; AbdulJaleel and Larkey, 2003; El-Kahky et al., 2011; Al-Badrashiny et al., 2014), or use an intermediate script such as the International Phonetic Alphabet (IPA) (Brawer et al., 2010) or Double Metaphones (Philips, 2000).

AbdulJaleel and Larkey (2003) proposed a statistical method for transliteration between Arabic and English. It does statistical alignment of a corpus of transliterated proper names, and produces the probability of transliterations of short n-grams between the two scripts. Then, input terms are segmented into the available n-grams, and all possible transliterations are produced and scored based on their joint probabilities. Habash (2009) used an ambiguous mapping that utilized the sounds-like indexing system Double Metaphones (Philips, 2000) combined with the direct mapping scores defined by Freeman et al. (2006) to handle out-of-vocabulary words in the context of Arabic-English machine translation. Freeman et al. (2006) extended Levenshtein Edit Distance to allow for improved matching of Arabic and English versions of the same proper names. El-Kahky et al. (2011) use graph reinforcement models to learn mapping

[2]For more information on Arabic natural language processing, see Habash (2010).

between characters in different scripts in the context of transliteration mining.

Al-Badrashiny et al. (2014) present a similar system where words are transcribed using a finite state transducer constructed from an aligned parallel Arabizi-Arabic corpus. The disadvantage of this and other learned methods (Ristad and Yianilos, 1998; Lin and Chen, 2002; Mangu and Brill, 1997; Jiampojamarn et al., 2009) is that they require aligned parallel corpora whereas our approach performs well without any data. We note that training data for attempted dictionary lookup of heard words is extremely scarce. Brawer et al. (2010) present an automatic transliteration system used to internationalize place names for Google Maps. Their approach relies on hand crafting rule sets, that map between orthographies and the IPA. Their approach does not require any training data; however, it requires expert knowledge of the writing system of different languages and is difficult for languages like English, which do not have a simple orthographic to phonetic mapping. However, it is advantageous because adding rules for a single orthographic system allows transliteration between all language pairs. As with the AbdulJaleel system, this approach does not address noisy information retrieval, where the target term may be incorrectly heard or spelled.

Phonetic Similarity Models Several phonetic similarity models have been discussed in the literature. In general, algorithms either directly compare phonetic pairs, or make use of phonetic feature vectors. Our approach in this paper uses phonetic features vectors to compute phonetic similarity.

Kuo et al. (2007) proposed a similarity model between English and Mandarin Chinese using phonetic confusion matrices. These are NxM matrices, giving the similarity two phone sets of size N and M. These similarities are generated statistically, either through aligned transliterations that have been converted to phones, or from errors produced in automatic speech recognition systems. However, the size of these matrices grow quadratically with the size of the phone set, so can require large amounts of training data.

Kondrak (2003) introduces an algorithm for calculating phonetic distance using separate phonetic feature vectors for vowels and consonants, with heuristic weights ascribed to each feature, and then calculating the best alignment using the Wagner-Fischer edit distance dynamic programming algorithm (Wagner and Fischer, 1974). There is a fixed cost for skipping a specific phone, and otherwise a cost for substitutions of phones proportionate to the difference in their weighted feature vectors.

Sriram et al. (2004) present a generic method for cross-linguistic search that shares the two stage approach in this paper, with a grapheme-to-phoneme model followed by matching in phonetic space with a distance metric. However, they only describe a rule based grapheme-to-phoneme system in abstract and do not present any sort of performance evaluation for their algorithm. Their search algorithm also requires the alignment and calculation of phonetic distance of the query against the entire database of search terms, which can be prohibitively expensive for large query sets.

3 MultiScript Search Algorithm

Our algorithm operates in two stages. First, the query term is converted from Arabic or Roman script into several possible phone sequences using several grapheme to phone conversion techniques. Then we use a phonetic distance algorithm to do search state enumeration based on a trie built from a phonetic Arabic dictionary.

We test different variations of grapheme to phone conversation techniques and phonetic distance metrics. For grapheme to phone conversion, we investigate using a simple manually created finite state transducer, and using deep bi-directional long short term recurrent neural networks. For the phonetic distance metric, we investigate using simple Levenshtein edit distance, unweighted phone feature vectors, and unweighted consonant and vowel feature vectors. We also show a method to train the system to produce weighted distance costs for performance increase.

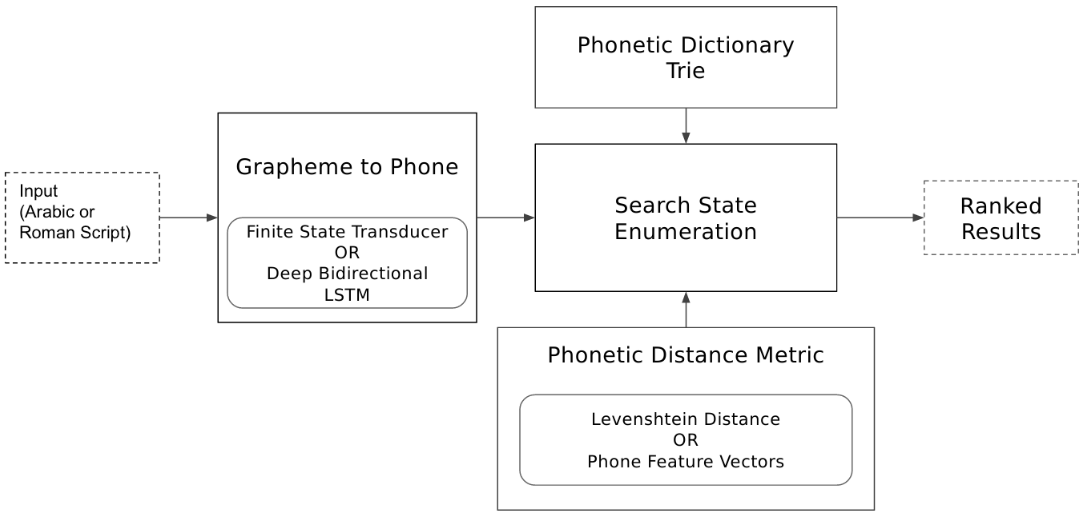
3.1 Grapheme to Phone Model

3.1.1 Finite State Transducer

We can define a very simple finite state transducer, which maps an orthographic input into a set of one or more IPA string outputs. This FST is compiled from a simple hand written list of rules of many-to-many mapping between orthographic substrings and single IPA phones. Creating such a list only requires basic knowledge of the orthographic system of the script.

The Arabic script FST is relatively simple, with

Figure 1: Overview of Phonetic Similarity Search algorithm


Phonetic Dictionary Trie
Input (Arabic or Roman Script)
Grapheme to Phone
Finite State Transducer OR Deep Bidirectional LSTM
Search State Enumeration
Ranked Results
Phonetic Distance Metric
Levenshtein Distance OR Phone Feature Vectors

most consonants just mapping to a single IPA output, but with one-to-many mappings for some characters such as the long vowels. Our Arabic FST contains all the rules described in Biadsy et al. (2009).

The Roman script FST contains Arabizi-specific heuristic mappings from one- or two-character Roman-script strings into IPA found in the Arabic language, e.g. *gh* mapping to /ʁ/. It also includes digits, such as 5 mapping to /x/ خ and 7 mapping to /ħ/ ح. We also introduce the constraint that IPA phones that are repeated in the output string are collapsed into a single phone, to deal with doubled characters (i.e., كتّاب *ktAb* /*kutta:b*/ 'authors' is mapped to /kuta:b/).

3.1.2 BiLSTM Recurrent Neural Network

A common strategy for grapheme to phone mapping is to use recurrent neural networks trained on parallel orthography to phonetic corpora. We train a deep bidirectional long-short term memory cells (LSTM), loosely based on the grapheme-to-phoneme model presented by Rao et al. (2015). Our RNN has two hidden layers, each containing 512 LSTM cells in both the forwards and backwards directions. We use the connectionist temporal classification cost (CTC) function to train the network (Graves et al., 2006). The use of the CTC cost function helps us avoid doing phone alignments between our parallel corpora. We decode the neural net outputs using a beam search with width 100, taking only the top 1 result.

We train on two different parallel corpora. The first is the CMU pronouncing dictionary, a phonetic dictionary for English.[3] We also train on a hybrid corpus, which contains data from both CMU dictionary and ArabScribe. We do not train LSTM for Arabic orthography because there is not enough variation in the mapping between the script and the phonetic output to justify their use.

3.2 Search State Enumeration

Our goal is to find the words in the phonetic dictionary with the lowest phonetic distance from our query input encoded as phones. A naive approach would be to use the Wagner-Fischer dynamic programming algorithm to calculate the optimal alignment of insertions, deletions and substitutions, of our query against all terms in the dictionary. However, the cost of this would be prohibitive. Instead, we dynamically search for the most phonetically similar words by maintaining a set of search states against a phonetic trie containing our dictionary. An overview of the approach is in Figure 1.

We first build a trie containing the phonetic representation of all words in the Arabic dictionary. We use the IPA as our phone set. For our dictionary, we use fully diacritized lemmas from the Buckwalter Morphological Analyzer (Buckwalter, 2004). We convert the fully diacritized Arabic into phones using the simple rule based method proposed in Biadsy et al. (2009). We store all these terms in a trie data structure, where each node optionally contains a set of words for which it is the terminating node.

[3]`http://www.speech.cs.cmu.edu/cgi-bin/cmudict`

Given a user input, we convert it into one or more phone strings, using one of our grapheme to phone algorithms. For each of these phone strings, we initialize a 4-tuple search state $(queryString, queryIndex, trieNode, cost)$ which represent the IPA encoded query, the index of the current phone in that string that is being consumed, the current node in the trie and the accumulated cost. For the initial search on "qamuus" (Arabic for 'dictionary'), one such four-tuple might be: $(kamus, 0, root, 0)$.

We then do search state enumeration until we discover N final search states which are our results for a $TopN$ search. The search state enumeration process is illustrated in Figure 2. We define a function $Transition$, that takes our search state s, and returns a set S' of zero or more search states. $Transition$ allows the search state to accept a single *insertion*, *deletion*, or *substitution* edit as in the Wagner-Fischer algorithm (Wagner and Fischer, 1974). The exact cost increase of each new search state s depends on the phonetic distance algorithm that we select.

$$
\begin{aligned}
Transition(s) &= S' \\
S' &= Ins(s) \cup Del(s) \cup Sub(s) \\
s'.cost &\geq s.cost \quad \forall s' \in S'
\end{aligned}
$$

$$
\begin{aligned}
Ins(s) &= \{\, s' \mid s'.tNode \in s.tNode.children \,\} \\
Del(s) &= \{\, s' \mid s'.qIndex = s.qIndex + 1 \quad \} \\
Sub(s) &= \{\, s' \mid s'.qIndex = s.qIndex + 1 \quad \wedge \\
&\qquad s'.tNode \in s.tNode.children \,\}
\end{aligned}
$$

We are looking for the words with the lowest phonetic distance overall, or equivalently, the lowest cost paths to final search states. A search state is considered final if we have consumed all characters in our query, and our resulting trie node is the end of a dictionary word. These conditions can be expressed as:

$$
\begin{aligned}
isFinal(s) =&(s.qIndex == len(s.qString) \\
&\wedge s.tNode.word \neq NULL)
\end{aligned}
$$

An optimization we use to prevent enumerating the entire tree is using a min-heap, containing all the search states we have seen. We always pop the minimum search state, and therefore are guaranteed to traverse the global set of search states in cost order. For searching a query that fully matches a dictionary word, the number of transitions to discover the first word is equal exactly to the number of characters in the phonetic representation of that word.

If we use a Fibonacci heap as our min-heap, the worst case complexity of this algorithm is O(ldqlog(dl)), where d is the length of the longest phonetic string in our dictionary, and l is the number of phones in our alphabet, and q is the length of the query, considering a complete Trie. However, as this algorithm discovers search states in cost order, in practice it is quite fast and does not need to enumerate many states before termination. It is also practical to implement a cost ceiling, after which transitions no longer need occur as you are too phonetically distant to have meaningful results. In this case, the phonetic search can be guaranteed to always terminate in a reasonable time.

3.3 Phonetic Distance Metric

We present below three strategies for defining the cost of insertion, deletion and substitution edits.

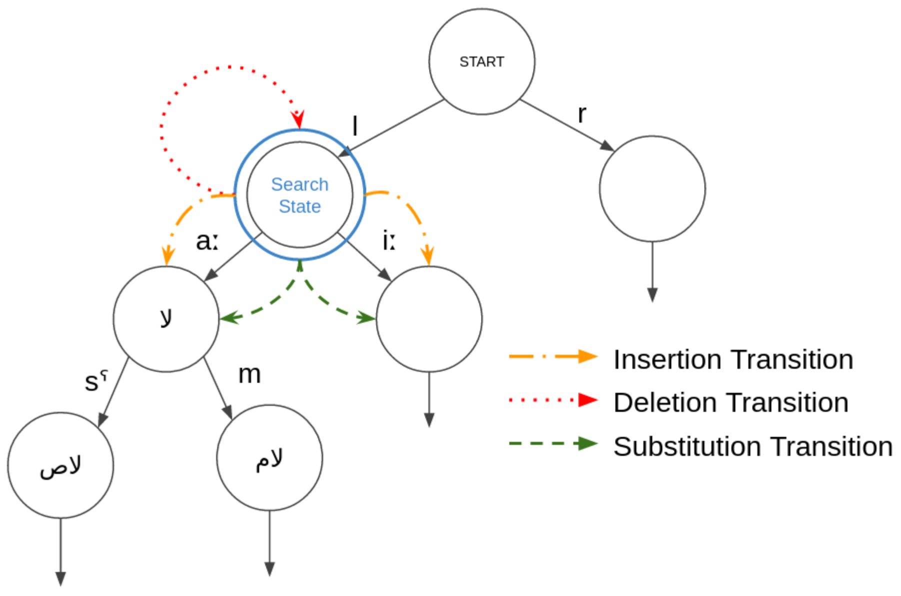
3.3.1 Levenshtein Distance

We investigate a naive approach where the cost of all insertion, deletion and substitution operations has a cost of 1. Thus, we find results that have low edit distance to our query. This approach is widely used in applications such as conventional spell correction, transliteration similarity and music information retrieval (Freeman et al., 2006; Toussaint and Oh, 2016).

3.3.2 Phone Binary Feature Vector

Again we define the cost of insertion and deletion edits to be 1. We represent phones using a 21-valued binary feature vector based on the Panphon package (Mortensen, 2016). Each feature is assigned either a value of + positive, - negative, or 0 as not relevant. The cost of substitution is then simply the distance between the two phones, which is calculated by sum the weights of the relevant non-zero features divided by the total weight of all features that are relevant in at least one of the two phones. For the untrained version, the weight of all features is equal to 1.

Figure 2: Search State Enumeration



$$D(p,q) = \frac{\sum_i W_i I(p_i,q_i) R(p_i,q_i)}{\sum_i W_i R(p_i,q_i)}$$

$$I(x,y) = \begin{cases} 1, & if x == y \\ 0, & otherwise \end{cases}$$

$$R(x,y) = \begin{cases} 1, & if\ (x! = 0 \lor y! = 0) \\ 0, & otherwise \end{cases}$$

Because we divide by the sum of all relevant weights, the output of our distance function is always in the range of $[0, 1]$, with $D(x, x) = 0$. To improve performance, we introduce two language specific rules. For Roman-script inputs, substitutions between consonants and vowels are disallowed; and for Arabic-script inputs, vowel insertions are free, as Arabic is usually written without short vowel diacritics.

3.3.3 Machine Learned Weights

A machine learning method is introduced to improve the accuracy of phonetic search by fine tuning the weights associated with inserting, deleting and substituting phones on a corpus of transcriptions. Each discrete phone p is given insertion and deletion costs in the range $[0, 10]$, while substitutions are given the range $[0, 1]$. The higher range for insertions and deletions is to allow insertion and deletion errors to take a higher cost than substitutions; this prevents our algorithm from enumerating over many unlikely insertion and deletions. A weight W_i is also assigned to each phone feature, but our distance function $D(p, q)$ still produces costs in the range $[0, 1]$. The weights are trained such that the phonetic distance between a given transcription and the actual word is minimized relative to the phonetic distances between the transcription and all other words in the training set.

We define an edit vector $\vec{E}_{t,w}$ between a transcription t and a word w as the sum of all edits required in the shortest path between the phonetic representations of t and w as calculated by the Wagner-Fischer algorithm (Wagner and Fischer, 1974). We define an arbitrary but fixed ordering on our set of phones P and on our phone features F. Thus the first $2|P|$ indices of $\vec{E}$ represent the number of inserts and deletions on specific phones, and the last $|F|$ indices represent the number of times a specific feature was different between two phones.

We construct an edit matrix for each transcription $\mathbf{M_t}$, where the first row M_t^0 is the edit vector between the transcription and the target word, followed by the edit vectors between the transcription and all other words in our training set. For ArabScribe, we use 41 phones and 21 phone features so the dimension of $\vec{E}$ is $2(41) + 21 = 103$. Our training set contains 400 words, so our edit matrix has dimensions $(400, 103)$.

To train, we initialize three weight variable vectors for insertion $\vec{W}_i$, deletion $\vec{W}_d$ and substitution $\vec{W}_s$ of length $|P|$, $|P|$ and $|F|$, respectively. We make a concatenated projected weight vector as follows:

$$\begin{aligned}\vec{W}_{\text{proj}} = concatenate(&10 \cdot sigmoid(W_i),\\ &10 \cdot sigmoid(W_d),\\ &\frac{W_s^2}{\sum_i W_s^{i2}})\end{aligned}$$

This makes $\vec{W}_{\text{proj}}$ align with $\vec{E}$ element-wise. We do the $sigmoid$ and division by squared sum operations in order to force the cost of edits into the ranges we specified above. Finally we define the cost function for a given transcription t as:

$$C_t = exp(D \cdot \frac{\vec{E}_{t,w} \cdot \vec{W}_{\text{proj}}}{\mathbf{M}_t \vec{W}_{\text{proj}}}) - 1$$

Where D is a scaling factor hyper-parameter that punishes higher costs much more than lower costs, therefore forcing the training to improve the more difficult cases rather than over-optimizing the easy cases. The optimal value is dependent on the data size but we found $D = 500$ worked well for the ArabScribe training set. Note $C_t = 0$ when t is a perfect transcription.

We do batch gradient descent with early stop at the point test performance drops. In our case, the entire ArabScribe training set could fit in memory, but for larger training sets stochastic gradient descent is recommended.

4 The ArabScribe Dataset

In order to benchmark the performance of our algorithms, and to provide training data for the bidirectional LSTM and trained weight vectors, we created the ArabScribe dataset of Arabic transcriptions. This dataset is freely available for research purposes.[4]

We randomly selected 500 MSA lemmas (the words) from the list of lemmas in our dictionary, the Buckwalter Arabic Morphological Analyzer (BAMA) (Buckwalter, 2004). BAMA contains 36,918 lemmas total. Five native speakers were recorded slowly and clearly speaking 100 words each, covering all 500 entries. Then, 103 participants listened to up to 100 words each. Each participant was assigned either the "English keyboard" (with digits and common punctuation) or sometimes the "Arabic keyboard" (without diacritics) if they reported that they could type Arabic. They were required to transcribe each word they heard in a way that was most natural to them. We collected 3,234 transcriptions with the Arabic keyboard (Arabic script) and 6,699 transcriptions with the English keyboard (Roman Script).

[4]ArabScribe can be downloaded from `http://camel.abudhabi.nyu.edu/resources/`.

No Experience	26
0-1 years of Arabic education	45
1-2 years of Arabic education	14
2+ years of Arabic education	6
Heritage speaker	3
Native speaker	9
Total	**103**

Table 2: ArabScribe participant skill levels

	Arabic Script	**Roman Script**
Train	654	1342
Test	2,580	5,357
Total	3,234	6,699

Table 3: ArabScribe transcription types

We collected native language and Arabic skill level of each participant were also collected. The skill levels are:

1. No Arabic experience
2. 0-1 years of Arabic education
3. 1-2 years of Arabic education
4. 2+ years of Arabic education
5. Heritage speaker (some Arabic exposure from family background but not fluent)
6. Native speaker

We tried to balance the dataset for having all words transcribed by an equal number of participants at all skill levels with both the Arabic and English keyboards. The transcriptions were also divided into a 20% testing set comprised of 100 words, and a 80% training set comprised of the other 400 words. Table 2 gives the the breakdown of the range of Arabic skill levels of the participants in our experiment; and Table 3 gives the breakdown of the number of samples in the train and test sets.

Script	Grapheme-to-Phone	Distance Metric	Top 1	Top 10
Roman	EngLSTM	Levenshtein	19.6%	21.3%
Roman	HybridLSTM	Levenshtein	20.5%	22.3%
Roman	EngLSTM	MultiScript Untrained	19.9%	37.1%
Roman	HybridLSTM	MultiScript Untrained	21.4%	36.8%
Roman	FST	Levenshtein	43.6%	49.1%
Roman	FST	MultiScript Untrained	36.8%	53.3%
Roman	**FST**	**MultiScript Trained**	**46.0%**	**60.7%**
Arabic	-	Exact Match	34.9%	-
Arabic	FST	Levenshtein	54.7%	57.3%
Arabic	FST	MultiScript Untrained	54.4%	65.4%
Arabic	**FST**	**MultiScript Trained**	**55.0%**	**69.0%**

Table 4: Recall rates for different methods in TopN dictionary lookup

5 Experimental Results

To benchmark our phonetic search algorithm, we search against each transcription from our test set against the Arabic dictionary of all terms from BAMA. We then look for the correct word in the top 1 and top 10 results returned. We varied the grapheme-to-phone technique and distance metric, and performed the test on both the Roman and Arabic script data. For grapheme-to-phone techniques, we used the finite state transducers (FST) for Arabic and Roman scripts. We also trained a BiLSTM for Roman script only on either the English dictionary (EngLSTM) or a mix of the English dictionary and the ArabScribe training set (HybridLSTM). For the distance metrics, we used the Levenshtein distance metric, and the untrained and trained varieties of the MultiScript algorithm. The full results are listed in Table 4. Our Baseline techniques are exact matching for Arabic Script and searching using the FST with a Levenshtein edit distance (Damerau, 1964) for Roman Script. In general the LSTM based techniques performed poorly compared to the FST techniques. MultiScript without training was similar in performance to the Levenshtein distance metric. However, using a FST with trained MultiScript Search was the best performer for both Roman and Arabic Script. The top performing algorithms identified the target word within the top 10 results for 69.0% of cases with Arabic script, and 60.7% of cases with Roman script.

FSTs likely outperformed BiLSTMs because the English LSTM is not capable of producing any phones in the Arabic phone set, and it also cannot recognize Arabizi special symbols such as punctuation and digits. Training with some samples from ArabScribe slightly boosted the LSTM performance, but we hypothesize that the very small size of this training set made it ineffective. However, a disadvantage of the FST approach, is that it was tailored for the English-Arabic mapping, and would not generalize well to searching against other language pairs.

We found that for both Arabic and Roman script inputs, the accuracy of the algorithm is much higher for participants with more experience. This is likely because they can hear the sounds more accurately and also because they recognize more of the words in the test set. Our recall rate for advanced learners (2+ years of Arabic education) was 87% for Arabic script input and 78.5% for Roman script input. This search accuracy is in fact slightly higher than what native speakers scored which was 85.4% and 77.8% for Arabic and Roman script, respectively. Users with Arabic experience found their desired word in the top 10 significantly more often than beginner learners (0-1 years experience), whose recall rates were only 56.1% for Arabic script and 55.7% for Roman script. The full recall rates of FST+MultiScript search can be found in Table 5.

The instances in which MultiScript failed to match were often because the user thoroughly misheard the sound, or that there were too many real Arabic words similar to the target word which all were matched with low phonetic distance scores. Some errors also arose due to our FST not being complex enough to account for all the different ways people write the sounds they hear.

Arabic Experience	**Roman**	**Arabic**
Native Speaker	68.1%	84.4%
Heritage Speaker	63.3%	No Data
Advanced (2+ years)	70.8%	87.2%
Intermediate (1-2 years)	53.3%	69.0%
Beginner (0-1 years)	55.7%	56.1%
No Experience	44.9%	N/A

Table 5: Top 10 recall rates at different skill levels for FST + MultiScript trained search.

Method	**Recall**
Google Translate Roman Top 1	21.5%
MultiScript Roman Top 1	**46.0%**
Yamli Roman Top 10+	44.5%
MultiScript Roman Top 10	**60.7%**
Google Translate Arabic Top 1	9.4%
MultiScript Arabic Top 1	**31.0%**

Table 6: Benchmarking against Google Translate and Yamli on **non-exact** dictionary matches.

Benchmarking against Google and Yamli Two popular tools that are commonly used by students of Arabic to look up unfamiliar words are Google Translate[5], through the “did you mean feature", and the Yamli Smart Arabic Keyboard[6]. We recognize that this comparison is not entirely fair as these tools target a much larger problem domain. However, we do this comparison as these are the tools most commonly used by students to address this unfamiliar word lookup problem.

Google Translate sometimes offers a single correction for non-dictionary word inputs, so we compare it to MultiScript top 1. For Arabic script inputs, 35% of the inputs were spelled exactly correctly, so we only measure rates of recovery from spelling errors rather than the rate of finding the correct word. The Yamli smart keyboard is a transliteration system where users can type Arabic using Roman letters, and it can return more than 10 suggestions per typed word, so we benchmark it against MultiScript top 10.

The results of this comparative benchmark are presented in Table 6. We significantly outperformed both Google Translate and Yamli in this specific dictionary lookup task for both Arabic and Roman scripts.

[5]https://translate.google.com/

[6]http://www.yamli.com/arabic-keyboard/

6 Conclusion and Future Work

We have demonstrated a novel algorithm for performing an efficient phonetic search against a large Arabic dictionary using both the Roman and Arabic scripts. We have also introduced the ArabScribe dataset containing around 10,000 transcription attempts of Arabic words in Roman and Arabic scripts. A comparative benchmark shows that our best setup significantly improves on the widely used Google Translate "did you mean" feature, as well as the Yamli Arabic smart keyboard.

Though our work was done for Arabic only, the only language specific methods involved were the finite state transducer rules and the system could easily be extended to other languages and orthographies. Other extensions of this work would be to apply these phonetic search algorithms in targeted information retrieval for one or more orthographies. For example, it could be applied in searching for names of people on Wikipedia, or names of places in a mapping application. It would also be interesting to see how the performance of the system improves with much larger scales of training data, and if using multivalued phone features would improve the distance metric.

Acknowledgments

The creation of ArabScribe data set was support by a New York University Abu Dhabi Capstone Project Fund.

References

Nasreen AbdulJaleel and Leah S. Larkey. 2003. Statistical transliteration for English-Arabic cross language information retrieval. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 139–146. ACM.

Mohamed Al-Badrashiny, Ramy Eskander, Nizar Habash, and Owen Rambow. 2014. Automatic Transliteration of Romanized Dialectal Arabic. *CoNLL-2014*, page 30.

Yaser Al-Onaizan and Kevin Knight. 2002. Machine transliteration of names in Arabic text. In *Proceedings of the ACL-02 workshop on Computational approaches to semitic languages*, pages 1–13. Association for Computational Linguistics.

Mohamed I. Alkanhal, Mohammed A. Al-Badrashiny, Mansour M. Alghamdi, and Abdulaziz O. Al-Qabbany. 2012. Automatic Stochastic Arabic Spelling Correction With Emphasis on Space Insertions and Deletions. *IEEE Transactions on Audio, Speech & Language Processing*, 20:2111–2122.

Kevin Atkinson. 2005. GNU Aspell. *Dostupno na: http://aspell. net/[01.10. 2013].*

Fadi Biadsy, Nizar Habash, and Julia Hirschberg. 2009. Improving the Arabic pronunciation dictionary for phone and word recognition with linguistically-based pronunciation rules. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 397–405. Association for Computational Linguistics.

Sascha Brawer, Martin Jansche, Hiroshi Takenaka, and Yui Terashima. 2010. Proper name transcription/transliteration with ICU transforms. In *34th Internationalization and Unicode Conference*.

Tim Buckwalter. 2004. Buckwalter Arabic morphological analyzer version 2.0. Linguistic Data Consortium, University of Pennsylvania, 2002. LDC catalog no.: LDC2004l02. Technical report, ISBN 1-58563-324-0.

Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176.

Kareem Darwish. 2014. Arabizi detection and conversion to Arabic. *ANLP 2014*, page 217.

Ali El-Kahky, Kareem Darwish, Ahmed Saad Aldein, Mohamed Abd El-Wahab, Ahmed Hefny, and Waleed Ammar. 2011. Improved transliteration mining using graph reinforcement. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1384–1393. Association for Computational Linguistics.

Ramy Eskander, Nizar Habash, Owen Rambow, and Nadi Tomeh. 2013. Processing Spontaneous Orthography. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Atlanta, GA.

Andrew Freeman, Sherri Condon, and Christopher Ackerman. 2006. Cross linguistic name matching in English and Arabic. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 471–478, New York City, USA.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM.

Nizar Habash, Abdelhadi Soudi, and Tim Buckwalter. 2007. On Arabic Transliteration. In A. van den Bosch and A. Soudi, editors, *Arabic Computational Morphology: Knowledge-based and Empirical Methods*. Springer.

Nizar Habash. 2009. Remoov: A tool for online handling of out-of-vocabulary words in machine translation. In *Proceedings of the 2nd International Conference on Arabic Language Resources and Tools (MEDAR), Cairo, Egypt*.

Nizar Habash. 2010. *Introduction to Arabic Natural Language Processing*. Morgan & Claypool Publishers.

Sittichai Jiampojamarn, Aditya Bhargava, Qing Dou, Kenneth Dwyer, and Grzegorz Kondrak. 2009. Directl: a language-independent approach to transliteration. In *Proceedings of the 2009 Named Entities Workshop: Shared task on transliteration*, pages 28–31. Association for Computational Linguistics.

Grzegorz Kondrak. 2003. Phonetic alignment and similarity. *Computers and the Humanities*, 37(3):273–291.

Karen Kukich. 1992. Techniques for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439.

Jin-Shea Kuo, Haizhou Li, and Ying-Kuei Yang. 2007. A phonetic similarity model for automatic extraction of transliteration pairs. *ACM Transactions on Asian Language Information Processing (TALIP)*, 6(2):6.

Wei-Hao Lin and Hsin-Hsi Chen. 2002. Backward machine transliteration by learning phonetic similarity. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics.

Lidia Mangu and Eric Brill. 1997. Automatic rule acquisition for spelling correction. In *ICML*, volume 97, pages 187–194.

Behrang Mohit, Alla Rozovskaya, Nizar Habash, Wajdi Zaghouani, and Ossama Obeid. 2014. The first qalb shared task on automatic text correction for arabic. In *Proceedings of the EMNLP 2014 Workshop on Arabic Natural Language Processing (ANLP)*, pages 39–47.

David Mortensen. 2016. Panphon. `https://github.com/dmort27/panphon`.

Kemal Oflazer. 1996. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89.

Lawrence Philips. 1990. Hanging on the metaphone. *Computer Language*, 7(12 (December)).

Lawrence Philips. 2000. The double metaphone search algorithm. *C/C++ Users Journal*, June.

Kanishka Rao, Fuchun Peng, Haşim Sak, and Françoise Beaufays. 2015. Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4225–4229. IEEE.

Eric Sven Ristad and Peter N. Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532.

K. Shaalan, R. Aref, and A. Fahmy. 2010. An approach for analyzing and correcting spelling errors for non-native Arabic learners. *Proceedings of Informatics and Systems (INFOS)*.

S. Sriram, P. P. Talukdar, S. Badaskar, K. Bali, and A. G. Ramakrishnan. 2004. Phonetic distance based crosslingual search. In *Proc. of the Intl. Conf. on Natural Language Processing*.

Godfried T. Toussaint and Seung Man Oh. 2016. Measuring musical rhythm similarity: Edit distance versus minimum-weight many-to-many matchings. In *Proceedings of the 16th International Conference on Artificial Intelligence*, pages 186–189, Las Vegas, USA, July 25–28.

Kristina Toutanova and Robert C. Moore. 2002. Pronunciation modeling for improved spelling correction. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 144–151. Association for Computational Linguistics.

Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173.

Casey Whitelaw, Ben Hutchinson, Grace Y. Chung, and Gerard Ellis. 2009. Using the web for language independent spellchecking and autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2-Volume 2*, pages 890–899. Association for Computational Linguistics.

Wajdi Zaghouani, Behrang Mohit, Nizar Habash, Ossama Obeid, Nadi Tomeh, Alla Rozovskaya, Noura Farra, Sarah Alkuhlani, and Kemal Oflazer. 2014. Large scale Arabic error annotation: Guidelines and framework. In *LREC*, pages 2362–2369.