Design principles of an open-source language modeling microservice package for AAC text-entry applications

Brian Roark
Google Research, US
roark@google.com

Alexander Gutkin
Google Research, UK
agutkin@google.com

Abstract

We present MozoLM, an open-source language model microservice package intended for use in AAC text-entry applications, with a particular focus on the design principles of the library. The intent of the library is to allow the ensembling of multiple diverse language models without requiring the clients (user interface designers, system users or speech-language pathologists) to attend to the formats of the models. Issues around privacy, security, dynamic versus static models, and methods of model combination are explored and specific design choices motivated. Some simulation experiments demonstrating the benefits of personalized language model ensembling via the library are presented.

1 Introduction

Designing and building text-entry systems for individuals with severe motor impairments is a key challenge in the field of augmentative and alternative communication (AAC). In successful cases this typically involves people with many diverse areas of expertise, including speech-language pathologists, those with human-computer interaction (HCI) or natural language processing (NLP) expertise, and, of course, users of the technology themselves. Given this diversity, few individuals will have the breadth of expertise to address all of the issues at play. For example, those focused on interface design or customization for a specific individual's needs may not have the NLP expertise to assemble effective predictive models to help drive the interface; and those who are building state-of-the-art language models (LMs) often lack HCI or AAC experience and are not optimizing their models with text-entry scenarios in mind. As we will illustrate later in the paper, choices of how a system employs LMs can make a big difference in the quality of the resulting predictions, thus effective LM services are critical for these systems. This paper presents the MozoLM open-source software library[1] for building services that allow user interfaces (UIs) to request probabilities from a collection of diverse LMs without having to match their requests to the formats of the models. This frees those working on interfaces and user configuration optimization from having to focus on specific LM details, and frees those working on LMs from necessarily focusing on UI or text-entry scenario specifics.

This work was initially inspired by our interest in Dasher (Ward et al., 2000, 2002), a text-entry system that in its standard implementation[2] is closely tied to certain dynamic language modeling methods (see Section 2 for details).[3] The dynamic nature of the models in Dasher have the virtue of automatically adapting to user input in an open-vocabulary manner, thus learning the idiosyncrasies of the individual over time; however the tight coupling between the modeling choice and interface has some serious drawbacks. First, Dasher could make use of large static general background LMs in addition to user-specific dynamic LMs, to provide extra predictive power for novices with little personalized text but also for more advanced users. This was demonstrated in Rough et al. (2014), who included a static word-based *n*-gram model in Dasher; we also provide evidence of the benefit of using ensembled static and dynamic LMs later in the paper. Second, Dasher may not be the only text-entry system that an individual makes use of, yet the personalized models maintained by the Dasher system are not straightforwardly accessible to these other applications. Finally, the tight coupling between modeling and the interface requires UI designers to take into account the specifics of the LM and those interested in improving the LMs must attend to the interface. Ideally, a text-entry application should be able to plug in any and all given LMs to derive whatever

[1] https://github.com/google-research/mozolm/
[2] https://www.inference.org.uk/dasher/
[3] MozoLM started as part of work on a new Dasher version.

useful information they can, even when all of the models are trained completely independently.

The design of the library was motivated by several considerations. First, large, possibly remote, general purpose LMs and small(er), local personalized models can profitably work in tandem to support open-vocabulary applications, and this is a potentially complex coordination that likely falls outside what many of those contributing to such an application design are interested in developing the expertise to perform. The library should allow for easy-to-configure support of these best practices. Section 3.1 presents the language model design criteria for the library. Second, personalized models must remain secure due to privacy concerns, so such services must include adequate security and privacy functionality. Further, multiple applications could potentially share the same microservice – either multiple text-entry applications on the same local device (hence possibly sharing dynamic models) or many clients for remotely running hubs. Finally, separation of the language modeling functionality into a completely separate component allows for independent development and testing. Such general architectural considerations are presented in Section 3.2.

2 Background

2.1 Language modeling intro and notation

Language models are used to determine the probability of a string S of discrete tokens t drawn from a vocabulary Σ. For ease of notation, let $S = t_0 t_1 t_2 \dots t_k$ where $t_i \in \Sigma$ for all i. By convention, without loss of generality, let the initial token t_0 always be a special start-of-string token <B> and the final token t_k always be a special end-of-string token <E>. For a given token t_i, let h_i be the history at that position, i.e., the tokens in S prior to t_i which are $t_0 \dots t_{i-1}$. Then, by the chain rule,

$$P(S) = \prod_{i=1}^{k} P(t_i \mid h_i) .$$

Language models can vary in what they consider a token (e.g., words or characters), what is present in their vocabulary Σ, and in the methods used to estimate $P(t_i \mid h_i)$ at each position in the string[4], but the above formulation holds in general. To provide probabilities, the model must be appropriately normalized so that, for any given history h

$$\sum_{t \in \Sigma} P(t \mid h) = 1$$

and for all $t \in \Sigma$, $0 \leq P(t \mid h) \leq 1$. Many if not most models in use today are appropriately smoothed (or regularized) so that for all $t \in \Sigma$ and $h \in \Sigma^*$, $0 < P(t \mid h) < 1$, i.e., all vocabulary items have non-zero probability in all contexts. Any token that is not found within the vocabulary Σ is called out-of-vocabulary (OOV) and receives zero probability from the model without some additional mechanism to allocate probability to OOVs.

Another language modeling concept that is key for text entry applications is whether the model is dynamic or static. **Dynamic** LMs update the model as new text is produced, so that the LM can subsequently provide higher probabilities to sequences that have already been observed, thus *personalizing* the model. Most large LMs, such as those discussed next as well as neural LMs, are **static**, i.e., they are estimated once then probabilities are served without being updated as text is produced.

2.2 Conventional Word-based n-gram LMs

Word-based n-gram models are a common class of LMs that have been widely used for many applications. They are distinguished by the nature of the vocabulary Σ, which is made up of a closed-vocabulary of words, and by methods for defining equivalence classes of histories based on a Markov assumption. The Markov assumption states that given the previous m words (for some value of m) in the history, the probability of a word is conditionally independent of words earlier in the history (Norris, 1998). Operationally, this assumption implies that, for a given token t_i and history h_i

$$P(t_i \mid h_i) = P(t_i \mid t_{i-m} \dots t_{i-1}) .$$

So, for example, if $m = 2$, then $P(t \mid h)$ can be estimated by only considering the previous 2 words in the history, e.g., if h = "they are under the bathroom", then

$$P(\text{sink} \mid h) = P(\text{sink} \mid \text{the bathroom}) .$$

The most common smoothing (regularization) method for these models relies on "backing off" to lower-order Markov models (i.e., smaller m) in certain circumstances, and using various methods to both decide when to back off and how to allocate the probabilities appropriately when doing so

[4]Some methods assign probabilities to whole sentences without relying on single-token estimates, such as Rosenfeld (1997), but for our purposes, this formulation suffices.

(Katz, 1987; Kneser and Ney, 1995). See Chen and Goodman (1996) for an overview of such methods.

Importantly in the context of text-entry applications, having a closed vocabulary means that words outside of that vocabulary are OOV, hence those words are assigned zero probability, even with backed off probabilities. Further regularizing to provide non-zero probability to words outside of Σ requires incorporation of probabilities from other kinds of language models.

Recent work in neural language modeling has generally emphasized models with tokens defined somewhere between word and character, at the level of multi-character sub-word tokens. For example, byte-pair encoding (Sennrich et al., 2016) or word-pieces (Schuster and Nakajima, 2012) are learned tokenizations that group together frequent character units, resulting in a configurable balance between the size of the vocabulary and the lengths of dependencies being effectively modeled. Models with these tokenizations provide open-vocabulary modeling like character-based models.

2.3 Language Modeling in AAC

Higginbotham et al. (2012) provide a thorough overview of early work on language modeling in AAC, beyond what we have the space to provide here; we refer readers to that paper for more details. Briefly, LMs are used to optimize keyboard layout and to provide word-completion and prediction utilities, among other uses, mirroring (and often pre-dating) similar approaches for mobile text entry. Optimization of the keyboard layout for scanning methods of text entry, whereby rows and columns of text in a grid are highlighted for selection, were extensively investigated by Lesher et al. (1998a) and others, and frequency-driven placement of characters in such systems remains common. Contextual probabilities can be used for disambiguation in ambiguous keyboards, such as the well-known T9 (Grover et al., 1998), where individual keys are assigned multiple possible characters. Optimizing groupings of such symbols and use of LMs for disambiguation are long-standing practices (Lesher et al., 1998b). LMs can also be used for word prediction, whereby full words are predicted either based on prior context or on the prefix of the word that has been typed (or both), and this has been shown to provide substantial reductions in keystrokes required for text entry in an AAC setting (Higginbotham, 1992). Issues around

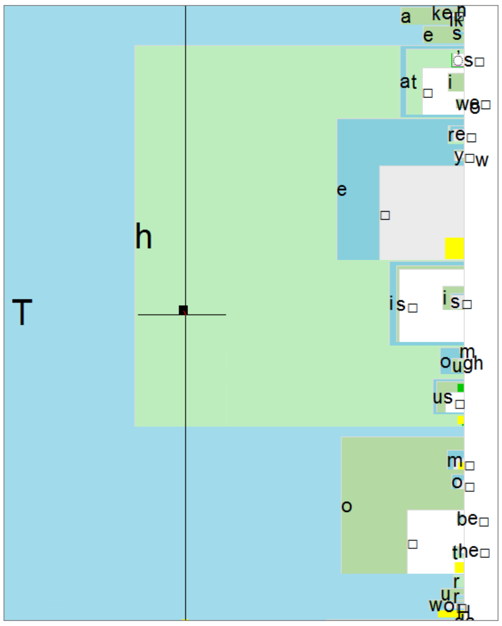

Figure 1: Screenshot of Dasher keyboard.

the cognitive load imposed by attending to word predictions in addition to keyboard manipulation cause the speedup to be less than the keystroke savings might suggest, but this remains a standard component in AAC text-entry systems (Higginbotham et al., 2012). LMs have also been used to improve the accuracy of brain-computer interface (BCI) text-entry systems (Oken et al., 2014), much as they are in mobile keyboards for autocorrection in the face of so-called fat-finger errors or for gesture-based input (Kristensson and Zhai, 2004). With the advent of larger, higher quality neural LMs such as GPT-3 (Brown et al., 2020) or T5 (Raffel et al., 2019), higher quality predictions are available to be leveraged for inclusion in such systems, in many of the same ways that they have been over the years.

2.4 Dasher and the PPM Language Model

We will provide some additional details about language modeling in the Dasher system (Ward et al., 2000, 2002; MacKay et al., 2004), both because it provided the initial motivation for the work, but also because the role of the LM in the interface is central (and unique) and the LM focus in Dasher has largely been on dynamic modeling (i.e., personalization), which is particularly important for text-entry applications. A screenshot of the system in operation is shown in Figure 1. Text entry in Dasher is achieved by navigating through an array of characters arranged in lexicographic order. Typing occurs by moving into regions to the right of the screen that are labeled with the intended letter. In the image, the central point has moved past the letters ‘T’ and ‘h’ and the most likely next letters (mostly vowels) are associated with relatively large regions, i.e., they are easy targets to navigate into. Word boundaries are marked with the square box, and some predictions extend beyond just the next character. For example, in the image, moving straight to the right of the central dot would result in the continuation “is□is□”, corresponding to the (relatively likely) string “This is . . . ” Even unlikely continuations have some probability, hence some space allocated to them.

The amount of space allocated for each character is determined by arithmetic coding (Rissanen and Langdon, 1979; Witten et al., 1987), so that high probability characters are larger targets for the navigating user than lower probability characters. Entering text is thus made easier by effective predictions of next characters, via easier-to-hit targets. The letters are arrayed in descending lexicographic order, so that one can move towards a character even if it is too small to see, until that character grows in size as one gets closer. Thus, for example, if one wants to type “Thx” – perhaps as an abbreviation for “Thanks” – then one would navigate towards the bottom of the sorted list. As one navigates in that direction, the probability that the target symbol is found on that end of the list grows, and the regions for those letters grow accordingly. Eventually the region allocated to the character will become large enough to be visible and navigation to that region becomes easier.

These examples illustrate a couple of important considerations for language modeling in Dasher. First, probabilities must be provided for the next character, not just the next word. Second, we may want to type something that does not occur in a standard lexicon, such as “Thx”, including things that we may type frequently due to our own personal conventions. Hence personalization, i.e., updating the language model as one types, can lead to higher probabilities for things that an individual frequently types. Due to these considerations, a major component of the Dasher system since the beginning was a dynamic character-based language model, most commonly Prediction by Partial Match or PPM (Cleary and Witten, 1984; Moffat, 1990).[5] See Appendix A.1 for explicit mathematical details of the specific PPM version implemented in MozoLM.

[5]In addition, Dasher supports the Context Tree Weighting (CTW) method (Willems et al., 1995; Willems, 1998) that was shown to be superior to PPM (van Veen, 2007) but has rarely been used in practical Dasher configurations.

2.5 Microservices in a Nutshell

Before the advent of sophisticated web technologies, such as cloud computing, software architectures were mostly monolithic, consisting of tightly coupled and often overlapping components hosted on the same machine and viewed as a single atomic unit. More often than not, introducing architectural changes to such a system, such as factoring out the data intensive components to run elsewhere or supporting a new platform, required a time-consuming and costly redesign. In recent years a modern alternative paradigm revolving around the notion of *microservices* has gained much popularity and wide acceptance in the industry, well attested by the plethora of books on the subject.[6]

Some of the commonly found definitions of the microservices concept are due to Dragoni et al. (2017) and Zimmermann (2017), who loosely define a microservice architecture as a collection of self-contained distributed services communicating via well-defined APIs, such as remote procedure call (RPC) message passing interfaces. The architecture follows the fine-grained separation of concerns, with each individual service designed around a particular *business capability*. One example may include a hypothetical component focused on user interaction (UI) loosely coupled with an LM component. Developing, testing and maintaining these two components in a microservices architecture can be made possible by two independent cross-functional teams each working in their own area of expertise. The component microservices are *independently deployable*, *scalable*, and *testable*. In our example, adding a new sensory interface to the UI, upgrading the LM or scaling its serving capacity, should not adversely affect the functioning of other components nor require their duplication. The architecture is often *polyglot*, which implies that the development is not restricted to any particular programming language, platform or development stack as long as individual components adhere to the same API for communication. Our implementation is based on popular gRPC high-performance communication framework. See Appendix B for the rationale behind its adoption and the review of such frameworks’ use in healthcare.

[6]See, e.g. Nadareishvili et al. (2016); Richardson (2019); Newman (2019, 2021); Vernon and Jaskula (2021); Khan et al. (2021); Ziadé and Fraser (2021).

3 Design Considerations

3.1 Language Model Issues

In this section we present general LM issues addressed by the library; further specific language modeling details are provided in Appendix A. A text-entry interface may request probabilities from the LMs given the current context (i.e., what has already been typed), then update the context (and possibly counts of observed strings in dynamic models) as further characters are typed. The interface to LMs should thus focus on two key requests: retrieving probabilities and updating counts/contexts. From the client's perspective, all models are accessed together through a single interface, so these functions must be supported by each model and coordinated by a central "hub". This raises key issues around the coordination of diverse models, including tokenization, static versus dynamic modeling, and methods of ensembling.

Tokenization is a major issue, since LMs are trained variously on different kinds of tokens, from words to sub-words to single characters. For example, the PPM model used in Dasher is a character-based LM, thus providing probabilities over single characters given the context. Large general LMs may be word-based, i.e., providing probabilities over a vocabulary of words, or based on other multi-character sub-word tokens. How does one create an ensemble over models with diverse tokenization? Our approach is to derive the estimates at the smallest unit: single characters, which in the library are defined as single Unicode code points. For a model with multi-character tokens, the probabilities must be calculated by summing the probabilities of all items in the vocabulary that have that character in that context. For example, if the already-typed context is "the dog h", then the probability of a particular letter following 'h' (say 'o') would be the sum of the probabilities of all words in the vocabulary beginning with 'ho' (house, home, hound, however, etc.) following the context 'the dog', appropriately normalized. Similar calculations must happen for sub-word models, so that all models being ensembled within the hub provide single character probabilities. If the UI requires multi-character estimates, e.g., for word prediction or completion, then some additional computation would be required to build them up from single characters. Note that whitespace is a character as well in this approach. Word-based models typically include whitespace implicitly at word boundaries, which must be accounted for.

The software library is built so that a given model type can be defined as a sub-class of the general LM class. Each sub-class must define its version of a set of core functions, such as returning probabilities given a context, returning a new context identifier given a previous context identifier and a newly typed character, and updating the model when characters are typed. Specific sub-classes will have different processing requirements to satisfy these core requests, including summing over multi-character tokens if the model has such a tokenization (as described above), normalizing the probabilities if the model stores raw counts, or actually updating counts in dynamic models. Appendix A.3 presents the model classes that have already been implemented in the MozoLM library.

Different models may provide probabilities for distinct vocabularies of characters, and the ensemble provides probabilities for the union of the model character vocabularies. For example, a local personalized LM P may have never used an accented vowel such as 'é', while a background LM Q would perhaps give that character non-zero probability, having observed it in a large corpus. Since P does not include the character in its vocabulary, its contribution to the overall probability of the character is zero, and all the probability mass for that character must come from Q. The union of all of the model vocabularies will have at least some probability mass coming from some of the models. The LM hub collects the probabilities over single characters from each model, and takes the union in the ensembling process, before returning the results to the client.

Dynamic models must be updated when text is entered, and it is the responsibility of the interface to call the update function. The hub tracks whether models are static or dynamic, and only dynamic models are updated. Dynamic models like the PPM will typically store raw counts and normalize on-the-fly to yield probabilities, while static models can pre-compute normalized probabilities.

Ensembling methods are defined at the hub level and can involve relatively simple approaches, such as interpolation with fixed weights, or more complicated ones that keep track of recent model performance on typed text to determine which model to rely upon. The hub is responsible for determining the mixing weights and ensuring that the final mixture is properly normalized – see Appendix A.2

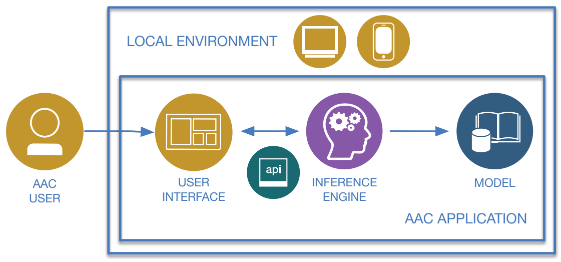

Figure 2: Monolithic AAC text-entry architecture.

for methods available in the library.

Many modeling methods require some extra processing to provide normalized character-level probabilities upon request at a given context, so it may be useful to cache the values for a context, to handle repeated visits more efficiently. This introduces a speed/memory tradeoff, and this tradeoff is generally handled at the LM sub-class level, since each modeling method may need to cache different information. For example, word-based n-gram models must sum over tokens to derive character-level estimates, and some information may be cached to make such summing more efficient. Again, see Appendix A.3 for details.

3.2 Architecture Details

AAC text-entry systems are commonly structured as a monolith compiled into a single application, shown schematically in Figure 2, where the system's tightly coupled components are crudely divided into a UI, an inference engine and an LM. The UI is often a complex system on its own, typically integrating various modes of user control, such as gaze tracking, and display. The inference engine is responsible for querying the supported LM and translating the LM estimates into a representation anticipated by the UI.

Separation of Concerns with LM Hub Splitting the business logic into interaction with the UI and display on the one hand, and LMs on the other, provides several advantages over the monolithic design. Consider the architecture in Figure 3 which shows a microservices configuration consisting of two components hosted on the same device. The main difference from the monolithic configuration in Figure 2 is that all the functionality that deals with LM inference now resides in a separated local service – the LM hub. What is left in the AAC application is a thin inter-process communication (IPC) layer for communicating with the LM hub using a gRPC UNIX socket mechanism (Stevens and Rago, 2013).

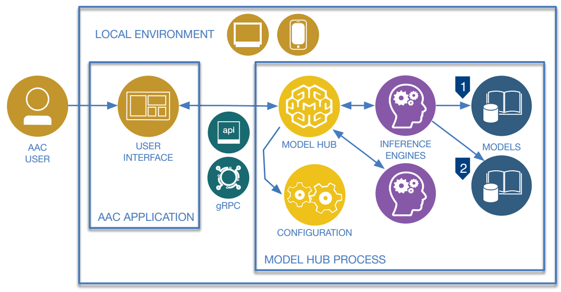

Figure 3: Monolith split into the UI and the LM hub.

In a typical scenario we envisage the LM hub as a standalone service binary that runs as a separate process from the UI application if on the same device, and as a remote gRPC service if configured to run over the network. Alternatively, the LM hub is also available as a regular library, which still allows the developer to combine the UI and LM components into a single monolith, simply a better structured one. The additional advantage of this architecture (mentioned in Section 2) is that it is "polyglot", e.g., the AAC application may be implemented in C++, while the LM hub may be implemented in Swift for accessing the native Apple iOS keyboard predictions (Ruan et al., 2018).

LM Hub Structure The primary purpose of the model hub service is to provide the LM predictions from one or several inference engines based on the service configuration (Appendix C.1 describes the configuration language). The inference engine is an abstraction that implements the model serving logic for particular types of model. The local architecture in Figure 3 has two model inference engines. The first engine serves two models. This inference engine implements light-weight dynamic LMs (models 1 and 2 in the figure) with the individual model predictions served by this engine combined by the model hub using a mixture method such as one of those presented in Appendix A. The second inference engine may serve bigger static LMs, such as pruned n-gram LMs (Heafield, 2011; Roark et al., 2012). Alternatively, this inference engine may serve a distilled neural model (Jiao et al., 2020; Sun et al., 2020; Niu et al., 2020). In either case the static model is optimized for running on an edge device.

Distributed LM Hubs Because in the proposed architecture the model hub is an independently deployable service, building a fully distributed architecture, where the model hub runs as a remote service, becomes easy. Figure 4 shows two of the

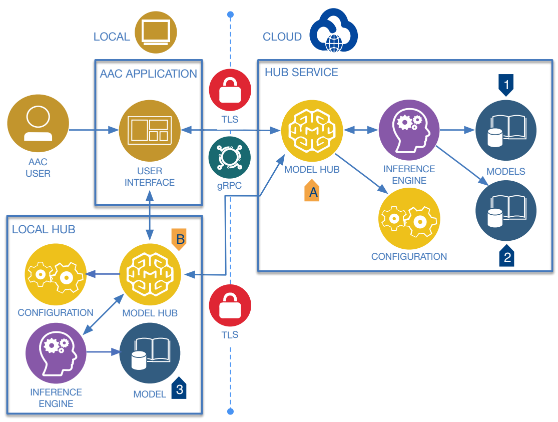

Figure 4: Architecture with a local and remote LM hubs in two configurations: a remote hub (A) only, and a local hub (B) communicating with the remote hub.

simplest distributed configurations possible.

The first scenario involves a single remote model hub service (denoted A in the figure). In our example this hub is virtually identical to the local in-process hub from Figure 3 in that it also serves two models. The main difference, of course, is that the models served by the remote hub may be chosen to be static rather than dynamic for a number of reasons, e.g., related to privacy (remote machine may not be fully trusted with user data) or computation (the machine may be powerful enough to serve large unoptimized models). Another important difference is that the communication between the AAC application and the model hub uses network gRPC channel secured using standard authentication mechanism, such as transport layer security (TLS) protocol (IETF, 2018). More implementation details are provided in Appendix C.2.

The second scenario in Figure 4 involves a distributed dual hub architecture: the AAC application communicates with a local model hub service (denoted B in the figure), which in turn communicates with the remote LM hub service (A) described above. The design feature that allows each LM hub to act as a client for other LM hubs enables more sophisticated architectures such as the one we are describing. In this example, the local hub is serving a single dynamic model, while the remote hub may be providing predictions from two third-party static models, with model ensembling being performed by each hub. Even more complex architectures are supported (see Appendix D).

4 Personalization Experiment

We have motivated this work in part with the idea that LMs with different characteristics can be profitably ensembled to provide better estimates, and in this section we present a small experiment to demonstrate this. This experiment was run using the MozoLM library with differently configured LM hubs and implementations of several common LM sub-classes.[7] We evaluated LM performance using data from the Enron Personalization Validation Set[8] (Fowler et al., 2015). That data collects emails written by 89 individuals, each in their own separate file, 45 of which are available for dev and 44 for test. Here we use the text from the 45 dev individuals, found in files `dev??.message.text.tsv`, up to a maximum of 140,000 characters per individual, in aggregate over 720k words and 3.9M characters.

Language models can be used for any number of applications – including text entry, the focus of this software library – but evaluation of language model quality is often performed intrinsically, by examining the probabilities assigned by models to attested text from the domain. Operationally, one measures the log probability of the validation corpus; and for ease of comparison at the character level, normalizes by the number of characters. If the log is base 2, this provides the number of bits per character (BPC), and lower values correspond to higher probabilities, i.e., better models.

Since each file in the dev set consists of text written by a single individual, dynamic models that update counts as the text file is processed will personalize the model to predict frequent patterns of that particular user. We score the cumulative BPC of the model, which, at each character, shows us how well the model has predicted the text that was typed up to that point. Lower BPC corresponds to higher probability assigned to the actual characters that were typed, i.e., better predictions of what the user will type. In Dasher, for example, this would correspond to larger regions being allocated to actual target characters within its arithmetic coding approach. Since we measure cumulative BPC at each character position, we can track the learning of any dynamic models that are included in the ensemble.

Figure 5 presents cumulative BPC aggregated over all 45 individuals in the dev partition, reporting the aggregate bits divided by aggregate characters as we synchronously step through each text

[7]The data used to train models is available at https://github.com/agutkin/slpat2022.

[8]https://github.com/google-research-datasets/EnronPersonalizationValidation

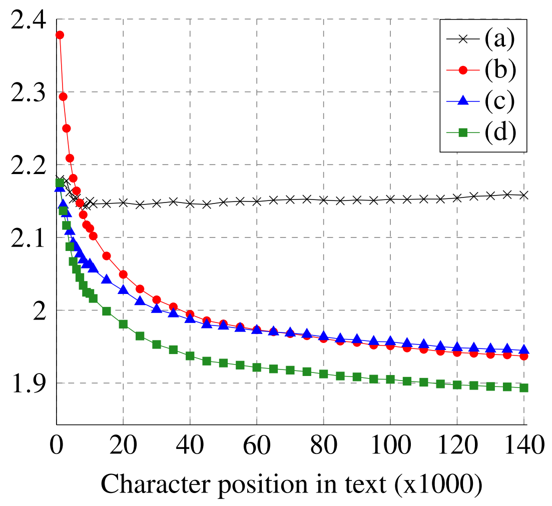

Figure 5: Cumulative BPC at text positions in collections of emails by the same individual, in four conditions: (a) uniformly-mixed ensemble of two static LMs; (b) uniformly-mixed ensemble of the two static models and a dynamic PPM 6-gram for each individual; and (c) and (d) Bayesian-mixed ensembles of the two static models and a dynamic PPM 6-gram for each individual, using history lengths 6 and 1, respectively, to determine mixing coefficients.

file. Thus, at position 5000, the cumulative BPC shows that measure for all users up to and including character 5000 in each of the files. Note that the dynamic LMs are being updated only for the specific individual, not shared among individuals.

Figure 5 presents cumulative BPC for four conditions. First, we combined two large static LMs trained on 20M sentences of English Wikipedia text (426M words). The first is a trigram word model with a closed vocabulary of 414,715 words; all other words are mapped to an out-of-vocabulary (OOV) token. This model was built using the OpenGrm NGram library (Roark et al., 2012) and pruned to contain a total of 100M n-grams. The second static model is an unpruned PPM model with a maximum n-gram length of 6, trained on the same data. Since this is a character-level model, the ensembling of these two models provides a fully open-vocabulary model over the characters found in the Wikipedia training data, and there are no OOV characters in the evaluation text. Condition (a) in Figure 5 shows the performance of an ensemble of the two static models, mixed uniformly. Note that this yields lower BPC (i.e., better performance) than using either model independently, conditions which are not shown for clarity.

The other three conditions mix the above static models with a dynamic PPM model (also maximum 6-gram) that is only trained on previously typed strings in the dev set itself. This dynamic model on its own performs far worse than any of the ensembles shown (results are omitted for clarity). The difference between these three dynamic model conditions is in the ensembling method. In condition (b), all three models are mixed uniformly, i.e., each contributes 1/3 of the probability mass. One can see from Figure 5 that condition (b) ends up improving substantially over condition (a), but at early character positions (b) has significantly higher BPC, since the dynamic model needs many observations before it can begin to produce useful probabilities. The other two conditions use a generalization of Bayesian interpolation (see Appendix A.2) to establish the ensemble mixing coefficients, which, among other things, reduces reliance on the dynamic model at earlier positions. Choosing the mixing coefficients based on just the previously typed character (condition d) outperforms using the previous 6 typed characters to calculate the coefficients (condition c).

This experiment is simply intended to motivate the ensembling of multiple diverse models, as we advocate for in the design of this library, as well as demonstrating the software in action. Of course, actual optimal model configuration will depend on the user and on the specific text-entry system being used. We can at least say that different models can have complementary characteristics, so that combining them, even in simple ways, can yield better models.

5 Conclusion and Future Work

We have presented the rationale for many choices made in designing an open-source microservice package for language modeling in AAC text-entry applications. The code presented is available open-source, and the experiments run in Section 4 were performed using the library. Future work will include adding more LM sub-classes, including commonly used neural LMs.

Acknowledgements

Thanks to Keith Vertanen, Jim Hawkins, Jeremy Cope, Will Wade, Jay Beavers, Stefan Zecevic, Lisie Lillianfeld and Jiban Adhikary for sharing their insights and expertise with us during the work on this project. The authors also thank Richard Sproat and the three anonymous reviewers for their useful feedback on the earlier version of this paper.

References

Randy Abernethy. 2019. *Programmer's Guide to Apache Thrift*. Manning Publications Co., Shelter Island, NY, USA.

Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. 2019. Character-level language modeling with deeper self-attention. In *Proceedings of the 33rd AAAI conference on Artificial Intelligence*, pages 3159–3166, Honolulu, Hawaii, USA. ACM.

Cyril Allauzen and Michael Riley. 2011. Bayesian language model interpolation for mobile speech input. In *Proceedings of Interspeech*, pages 1429–1432, Florence, Italy. International Speech Communication Association (ISCA).

Adam Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mauro Caporuscio, Danny Weyns, Jesper Andersson, Clara Axelsson, and Göran Petersson. 2017. IoT-enabled physical telerehabilitation platform. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 112–119, Gothenburg, Sweden. IEEE.

Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 310–318, Santa Cruz, California, USA. Association for Computational Linguistics.

John Cleary and Ian Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402.

Ciprian Dobre, Lidia Băjenaru, Ion Alexandru Marinescu, Mihaela Tomescu, Gabriel Ioan Prada, and Susanna Spinsante. 2021. New opportunities for older adults care transition from traditional to personalised assistive care: vINCI platform. In *Proceedings of 2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pages 515–520, Bucharest, Romania. IEEE.

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pages 195–216.

Hicham El Boukkouri, Olivier Ferret, Thomas Lavergne, Hiroshi Noji, Pierre Zweigenbaum, and Jun'ichi Tsujii. 2020. CharacterBERT: Reconciling ELMo and BERT for word-level open-vocabulary representations from characters. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6903–6915, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Marius Eriksen. 2014. Your server as a function. *ACM SIGOPS Operating Systems Review*, 48(1):51–57.

Andrew Fowler, Kurt Partridge, Ciprian Chelba, Xiaojun Bi, Tom Ouyang, and Shumin Zhai. 2015. Effects of language modeling and its personalization on touchscreen typing performance. In *Proceedings of the 33rd ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 649–658, Seoul, Korea. ACM.

Dale L. Grover, Martin T. King, and Clifford A. Kushler. 1998. Reduced keyboard disambiguating computer. U.S. Patent US5818437A, October. Tegic Communications Inc.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland. Association for Computational Linguistics.

D. Jeffery Higginbotham. 1992. Evaluation of keystroke savings across five assistive communication technologies. *Augmentative and Alternative Communication*, 8(4):258–272.

D. Jeffery Higginbotham, Gregory W. Lesher, Bryan J. Moulton, and Brian Roark. 2012. The application of natural language processing to augmentative and alternative communication. *Assistive Technology*, 24(1):14–24.

Paul Glor Howard. 1993. *The design and analysis of efficient lossless data compression systems*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, USA. Tech. Report No. CS-93-28.

Bo-June Hsu. 2007. Generalized linear interpolation of language models. In *Proceedings of IEEE Workshop on Automatic Speech Recognition & Understanding (ASRU)*, pages 136–140, Kyoto, Japan. IEEE.

Joshua Humphries, David Konsumer, David Muto, Robert Ross, and Carles Sistare. 2018. *Practical gRPC*. Bleeding Edge Press, Santa Rosa, CA, USA.

IETF. 2018. The transport layer security (TLS) protocol version 1.3. Internet Engineering Task Force, RFC 8446. Version 1.3, August.

Kasun Indrasiri and Danesh Kuruppu. 2020. *gRPC Up & Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, Inc., USA.

Frederick Jelinek and R. L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, The Netherlands. North-Holland Publishing.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online. Association for Computational Linguistics.

Simon Josefsson and Sean Leonard. 2015. Textual encodings of PKIX, PKCS, and CMS structures. Internet Engineering Task Force (IETF), RFC 7468. April.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401.

Ovais Mehboob Ahmed Khan, Arvind Chandaka, and Robert Vettor. 2021. *Developing Microservices Architecture on Microsoft Azure with Open Source Technologies*. Microsoft Press.

Dietrich Klakow. 1998. Log-linear interpolation of language models. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP)*, page paper 0522, Sydney, Australia. International Speech Communication Association (ISCA).

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 181–184, Detroit, Michigan, USA. IEEE.

Per-Ola Kristensson and Shumin Zhai. 2004. SHARK^2: a large vocabulary shorthand writing system for pen-based computers. In *Proceedings of the 17th annual ACM symposium on User Interface Software and Technology (UIST)*, pages 43–52, Santa Fe, NM, USA. Association for Computing Machinery (ACM).

Gregory Lesher, Bryan Moulton, and D. Jeffery Higginbotham. 1998a. Techniques for augmenting scanning communication. *Augmentative and Alternative Communication*, 14(2):81–101.

Gregory W. Lesher, Bryan J. Moulton, and D. Jeffery Higginbotham. 1998b. Optimal character arrangements for ambiguous keyboards. *IEEE Transactions on Rehabilitation Engineering*, 6(4):415–423.

Xunying Liu, Mark John Francis Gales, and Philip C. Woodland. 2013. Use of contexts in language model interpolation and adaptation. *Computer Speech & Language*, 27(1):301–321.

David J. C. MacKay, Chris J. Ball, and Mick Donegan. 2004. Efficient communication with one or two buttons. *AIP Conference Proceedings*, 735(1):207–218.

Argyro Mavrogiorgou, Spyridon Kleftakis, Konstantinos Mavrogiorgos, Nikolaos Zafeiropoulos, Andreas Menychtas, Athanasios Kiourtis, Ilias Maglogiannis, and Dimosthenis Kyriazis. 2021. beHEALTHIER: A microservices platform for analyzing and exploiting healthcare data. In *Proceedings of IEEE 34th International Symposium on Computer-Based Medical Systems (CBMS)*, pages 283–288, Aveiro, Portugal. IEEE.

Andrea Melis, Silvia Mirri, Catia Prandi, Marco Prandini, Paola Salomoni, and Franco Callegati. 2016. A microservice architecture use case for persons with disabilities. In *Proceedings of 2nd International Conference on Smart Objects and Technologies for Social Good (GOODTECHS)*, pages 41–50, Venice, Italy. Springer.

Russ Miles and Kim Hamilton. 2006. *Learning UML 2.0: A Pragmatic Introduction to UML*. O'Reilly Media, Inc., USA.

Alistair Moffat. 1990. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921.

Mukhriddin Mukhiddinov and Jinsoo Cho. 2021. Smart glass system using deep learning for the blind and visually impaired. *Electronics*, 10(22):2756.

Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., USA.

Sam Newman. 2019. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc., USA.

Sam Newman. 2021. *Building Microservices: Designing Fine-Grained Systems*, 2nd edition. O'Reilly Media, Inc., USA.

Wei Niu, Zhenglun Kong, Geng Yuan, Weiwen Jiang, Jiexiong Guan, Caiwen Ding, Pu Zhao, Sijia Liu, Bin Ren, and Yanzhi Wang. 2020. Real-time execution of large-scale language models on mobile. *arXiv preprint arXiv:2009.06823*.

James Robert Norris. 1998. *Markov Chains*. Number 2 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge, UK.

Barry S. Oken, Umut Orhan, Brian Roark, Deniz Erdogmus, Andrew Fowler, Aimee Mooney, Betts Peters, Meghan Miller, and Melanie B. Fried-Oken.

2014. Brain–computer interface with language model–electroencephalography fusion for locked-in syndrome. *Neurorehabilitation and Neural Repair*, 28(4):387–394.

Adina M. Panchea, Dominic Létourneau, Simon Brière, Mathieu Hamel, Marc-Antoine Maheux, Cédric Godin, Michel Tousignant, Mathieu Labbé, François Ferland, François Grondin, and François Michaud. 2021. OpenTera: A microservice architecture solution for rapid prototyping of robotic solutions to COVID-19 challenges in care facilities. *arXiv preprint arXiv:2103.06171*.

Anelis Pereira-Vale, Eduardo B Fernandez, Raúl Monge, Hernán Astudillo, and Gastón Márquez. 2021. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200.

Gabriela Postolache, Pedro Silva Girão, Octavian Adrian Postolache, José Miguel Dias Pereira, and Vitor Viegas. 2019. IoT based model of healthcare for physiotherapy. In *Proceedings of 2019 13th International Conference on Sensing Technology (ICST)*, pages 1–6, Sydney, Australia. IEEE.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Krzysztof Rakowski. 2015. *Learning Apache Thrift*. Packt Publishing, Birmingham, UK.

Jorge Rendulich, Jorge R. Beingolea, Milagros Zegarra, Isaac G. G. Vizcarra, and Sergio T. Kofuji. 2019. An IoT environment for the development of assistive applications in smart cities. In *Proceedings of 2019 IEEE 1st Sustainable Cities Latin America Conference (SCLA)*, pages 1–4, Arequipa, Peru. IEEE.

Chris Richardson. 2019. *Microservices Patterns: With examples in Java*. Manning Publications Co., Shelter Island, NY, USA.

Jorma Rissanen and Glen G. Langdon. 1979. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66, Jeju Island, Korea. Association for Computational Linguistics.

Ronald Rosenfeld. 1997. A whole sentence maximum entropy language model. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 230–237, Santa Barbara, CA, USA. IEEE.

Daniel Rough, Keith Vertanen, and Per Ola Kristensson. 2014. An evaluation of Dasher with a high-performance language model as a gaze communication method. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, pages 169–176, Como, Italy. Association for Computing Machinery (ACM).

Sherry Ruan, Jacob O. Wobbrock, Kenny Liou, Andrew Ng, and James A. Landay. 2018. Comparing speech and keyboard text entry for short messages in two languages on touchscreen phones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(4):1–23.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and Korean voice search. In *Proceedings of 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, Kyoto, Japan. IEEE.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Christian Steinruecken, Zoubin Ghahramani, and David MacKay. 2015. Improving PPM with dynamic parameter updates. In *Proceedings of 2015 Data Compression Conference (DCC)*, pages 193–202, Snowbird, Utah, USA. IEEE.

W. Richard Stevens and Stephen A. Rago. 2013. *Advanced Programming in the UNIX® Environment*, 3rd edition. Addison-Wesley Professional Computing Series. Addison-Wesley.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, Online. Association for Computational Linguistics.

Martijn van Veen. 2007. Using context-tree weighting as a language modeler in Dasher. Master's thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, February.

Vaughn Vernon and Tomasz Jaskula. 2021. *Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture*. Pearson Addison-Wesley Signature Series. Addison-Wesley Professional.

David J. Ward. 2001. *Adaptive Computer Interfaces*. Ph.D. thesis, Inference Group, Cavendish Laboratory, University of Cambridge, Cambridge, UK.

David J. Ward, Alan F. Blackwell, and David J. C. MacKay. 2000. Dasher — a data entry interface using continuous gestures and language models. In *Proceedings of the 13th annual ACM symposium on User Interface Software and Technology (UIST)*, pages 129–137, San Diego, California, USA. Association for Computing Machinery (ACM).

David J. Ward, Alan F. Blackwell, and David J. C. MacKay. 2002. Dasher: A gesture-driven data entry interface for mobile computing. *Human–Computer Interaction*, 17(2-3):199–228.

Frans M. J. Willems. 1998. The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, 44(2):792–798.

Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. 1995. The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664.

Edwin B. Wilson. 1927. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212.

Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32. Curran Associates, Inc.

Tetiana Yarygina. 2018. *Exploring Microservice Security*. Ph.D. thesis, Department of Informatics, University of Bergen, Bergen, Norway.

Yeliz Yesilada and Simon Harper. 2019. Futurama. In Yeliz Yesilada and Simon Harper, editors, *Web Accessibility: A Foundation for Research*, 2nd edition, Human–Computer Interaction Series, pages 791–803. Springer, London, UK.

Tarek Ziadé and Simon Fraser. 2021. *Python Microservices Development: Build efficient and lightweight microservices using the Python tooling ecosystem*, 2nd edition. Packt Publishing, Birmingham, UK.

Olaf Zimmermann. 2017. Microservices tenets: Agile approach to service development and deployment. *Computer Science — Research and Development*, 32(3):301–310.

A Language Modeling Specifics

A.1 PPM Language Model

There are many PPM variants – see Ward (2001) for a review. Here we will present the PPMD variant (Howard, 1993) that has been implemented in this library. We follow the "blending" and "update exclusion" (known as *single counting* from Moffat, 1990) approach taken in Steinruecken et al. (2015), and assign probabilities using a variant of equation 4 in that paper. In such an approach, there are three hyperparameters: α, β and m. Both α and β fall between 0 and 1, and $m \geq 0$ specifies that the longest strings included in the model are of length $m \quad 1$.

Let Σ be a vocabulary of characters, including a special end-of-string symbol. Let $h \in \Sigma^*$ be the contextual history and $t \in \Sigma$ a token following h, e.g., h might be "this is the contextual histor" and t might be "y". Let h' be the back-off contextual history for h, which is the longest proper suffix of h if one exists, and the empty string otherwise. Thus, for our example above, h' is "his is the contextual histor". For any $x \in \Sigma^*$ let $c(x)$ denote the count of x, and $C(x) = \max(c(x) - \beta, 0)$. We will specify how counts are derived later. Finally, let $U(h) = \{t : c(ht) > 0\}$ and $S(h) = \sum_x c(hx)$.

Probabilities are defined based on "blending" multiple orders, a calculation which recurses to lower orders, terminating at the unigram probability, which is when h is the empty string. For the unigram probability, we smooth via add-one Laplace smoothing (Wilson, 1927), i.e., for all $t \in \Sigma$

$$P(t) = \frac{c(t) \quad 1}{\sum_x c(x) \quad 1}.$$

If h is non-empty, then its probability is defined using the metaparameters α, β mentioned earlier:

$$P(t \mid h) = \frac{C(ht) \quad (U(h)\beta \quad \alpha)\, P(t \mid h')}{S(h) \quad \alpha}.$$

Counting occurs via "update exclusion". With each new observation t in the context of h, we update our count $c(ht)$. Let $k = \min(\text{length}(ht), m \quad 1)$, and let $X = h't$ be the suffix of ht of length k. Let X' be the longest suffix of X that was previously observed, i.e., where $c(X') > 0$.[9] Then we increment the counts by one for all substrings Y of ht such that length(X) >= length(Y) >= length(X'). See Steinruecken et al. (2015) for further details about this method.

A.2 Ensembling Methods

Two language models, such as M_1 and M_2 shown in Figure 3, can be combined into an ensemble model in many ways. Perhaps the simplest requires just a single parameter λ between 0 and 1 that determines

[9] We assume that t has been observed, since we use Laplace add-one smoothing for the unigram.

how much of the probability to derive from M_1, with the rest $(1-\lambda)$ coming from M_2:

$$P(t \mid h) = \lambda P_1(t \mid h) \quad (1-\lambda)P_2(t \mid h)\,,$$

where P_k is the probability as given by model M_k. This approach can generalize beyond two models by mixing a third model, such as M_3 in the local hub of Figure 4, with the above ensemble using a second mixing parameter γ (also between 0 and 1):

$$\begin{aligned} P(t \mid h) \;&=\; \gamma(\lambda P_1(t \mid h) \quad (1-\lambda)P_2(t \mid h)) \\ &\qquad (1-\gamma)P_3(t \mid h) \end{aligned}$$

The method of estimating the mixing parameters λ, γ can vary, and includes long-known methods such as using expectation maximization (EM) on a held-aside corpus (Jelinek and Mercer, 1980), which is also used for backoff smoothing parameter estimation in some approaches.

Ensembling can have several benefits in the context of text-entry applications. First, it can allow for the use of closed-vocabulary word-based models without assigning zero probability to OOV words, when mixed with an open-vocabulary, e.g., character-based, model. Second, models trained on different training sets can be complementary in their distributions, so that a mixture of the two provides better overall probabilities than either model on its own. Finally, mixtures of static and dynamic models can provide better personalization than is provided by just dynamic models on their own, as evidenced in the results of Section 4.

For those experiments, we used a somewhat more complicated method (also available in the library) for calculating the mixture than single parameters λ and γ, based on a generalization of Bayesian interpolation (Allauzen and Riley, 2011). Given K models, each $k \in K$ having a normalized prior weight w_k such that $\sum_{k \in K} w_k = 1.0$, then

$$P(t \mid h) = \sum_{k \in K} m_k(h) P_k(t \mid h)\,,$$

where $P_k(t \mid h)$ is the probability of t given h in model k, and $m_k(h)$ is the mixture weight for model k given history h, calculated as

$$m_k(h) \;=\; \frac{w_k P_k(h)}{\sum_{k' \in K} w_{k'} P_{k'}(h)}\,.$$

In this version, the length of the history considered when calculating $P_k(h)$ is parameterized, so that we consider only the previous j symbols regardless of the order of the model, where j is a given parameter. For $j > 0$:

$$m_k(h_i) \;=\; \frac{1}{Z} w_k P_k(t_{i-1} \mid h_{i-1}) \ldots P_k(t_{i-j} \mid h_{i-j})\,,$$

where Z is the appropriate normalization across all models so that the mixture weights sum to one.

The plot in Figure 5 shows that using this method can effectively balance the use of static and dynamic models so that, before sufficient observations have been accrued in the dynamic model, the static models are relied upon. Using just one previous character to assign the mixtures yielded a better ensemble model than using the prior 6 characters.

In addition to the linear formulation that we've been presenting in this section, more sophisticated types of fixed-weight interpolation exist, including log-linear interpolation (Klakow, 1998) inspired by maximum entropy models (Berger et al., 1996); generalized linear interpolation using context (history)-dependent weights (Hsu, 2007); and the combination of both linear and log-linear methods (Liu et al., 2013).

A.3 Model Classes Implemented in Library

At the current time, four language model subclasses have been defined in the library: a simple bigram character model class using a dense matrix to encode the model; a character n-gram class that uses the OpenGrm NGram model finite-state transducer (FST) format (Roark et al., 2012) to encode the model; a word-based n-gram class also using the OpenGrm FST format; and a PPM model class, also represented internally as a finite-state transducer. The point of the library is to allow the addition of new model classes, and these existing classes provide examples of how to do this for, say, neural language models. In this section, we will briefly identify some of the features of the model classes that were required to make them function within the ensembling framework.

Two of the classes require extra processing to serve the probabilities from the stored model format. First, for the word-based n-gram model, character-level probabilities must be derived by summing over all words that match the history. To do this, we sort the model lexicon in lexicographic order and collect all word probabilities at the word-initial position. Then, as each letter of the current word is typed, all the words that match that prefix fall within an interval in the lexicon. Pre-summing the probabilities over the whole list allows us to calculate the total probability in the interval via a single difference in probabilities. Second, counts are stored in the PPM model, rather than normalized probabilities, since the model is typically dynamic, i.e., it is being updated with new counts as the system operates. For this reason, calculation of probabilities from counts is required before serving probabilities in this model class.

Because both of these models require extra processing, a small bounded caching approach is included in both model classes, to permit states in the model to store calculated results in case the states are revisited during revision or as part of probability calculation.

B gRPC and Microservices in Healthcare

Several open-source high-performance RPC communication frameworks for microservice architectures have emerged over the years, Google gRPC (Humphries et al., 2018; Indrasiri and Kuruppu, 2020),[10] Apache Thrift (Rakowski, 2015; Abernethy, 2019),[11] and Finagle from Twitter (Eriksen, 2014),[12] among several others. Our work adopts gRPC not least because of its feature maturity, stability, popularity in the industry and academia, as well as the availability of security mechanisms, crucial in microservice environments (Yarygina, 2018; Pereira-Vale et al., 2021), which it provides out of the box.

There is a growing body of literature either solely devoted to or mentioning the use of microservices architectures in healthcare, in particular in health information systems (HIS) (Mavrogiorgou et al., 2021), mobility (Melis et al., 2016; Rendulich et al., 2019; Mukhiddinov and Cho, 2021), physiotherapy (Caporuscio et al., 2017; Postolache et al., 2019), and elderly patients care (Dobre et al., 2021; Panchea et al., 2021). Furthermore, there is a growing awareness of the importance of flexible software architectures in assistive technologies as the Web becomes even more ubiquitous (Yesilada and Harper, 2019). Our work investigates one such architecture in the area of text entry for AAC.

[10] https://grpc.io/
[11] https://thrift.apache.org/
[12] https://twitter.github.io/finagle/

```
// Model hub section.
model_hub_config {
  mixture_type: LINEAR_INTERPOLATION
  model_config {  // First model.
    type: PPM
    weight: 0.301  // -std::log10(0.5)
    storage {
      model_file: "${PRIVATE_TEXT_FILE}"
      ppm_options {
        max_order: 5  // 5-gram.
        static_model: false  // Dynamic.
      }
    }
  }
  model_config {  // Second model.
    type: CHAR_NGRAM_FST
    weight: 0.301  // -std::log10(0.5)
    storage {
      model_file: "${FST_FILE}"
      vocabulary_file: "${VOCAB_FILE}"
    }
  }
}
// Networking and authentication.
address_uri: "x.x.x.x:${PORT}"
auth {  // Authentication.
  tls {  // Transport layer security.
    // Strings below are PEM-encoded.
    private_key: "..."
    // Public certificate.
    server_cert: "..."
    // Custom certificate authority.
    custom_ca_cert: "..."
    // Require valid client certificate.
    client_verify: true
  }
}
```

Table 1: Example microservice configuration consisting of two linearly interpolated dynamic and static models.

C Practical Example

C.1 Configuration Language

We use the flexible text format of Google protocol buffers[13] as a configuration language for customizing the LM hub, where a number of different LM algorithms, their particular run-time parameters, types of tokens (e.g., character or word-based models), alphabets, and various prediction blending techniques can be defined for a particular LM hub configuration. Here we present a concrete example of this configuration language.

Table 1 shows an example of a two-model configuration using this syntax that may correspond to the model hub running locally (Figure 3) or remotely (model hub H_A in Figure 4).

The configuration consists of two main sections: the LM hub, and the microservice settings for networking and authentication mechanisms. In this

[13] https://developers.google.com/protocol-buffers

particular configuration, the LM hub is configured to serve the linearly interpolated predictions from two models: the dynamic PPM 5-gram model (the first model in hub's configuration) and the character n-gram model encoded as a finite state transducer (FST) with unspecified model order (which is assumed to be stored in the model file) and explicitly specified external vocabulary file. Both models are contributing equally to the final prediction with interpolation weight $\lambda = 0.5$.[14] Also note, that in this example the dynamic model relies on the external text file (provided by the `PRIVATE_TEXT_FILE` environment variable) for initialization: this file is used to bootstrap the dynamic PPM model from user's previous typing history during the initialization, similar to implementation in Dasher. If the PPM file is empty or not provided, the model starts with a uniform distribution.

The second configuration section in our example contains the networking and authentication setup for the LM hub microservice: the IP address and the port of the network interface, as well as the configuration for the TLS authentication mechanism with the necessary cryptographic keys and certificates encoded as strings in Privacy Enhanced Mail (PEM) format (Josefsson and Leonard, 2015). Note that in this example the microservice authenticates all the client connections for added security by requiring the clients to present the valid client certificates, achieved by enabling the `client_verify` configuration flag.

C.2 Life of an Estimate

The simplified class diagram providing the details of the core LM hub components (excluding the gRPC-based microservice details) in Unified Modeling Language (UML) notation (Miles and Hamilton, 2006) is shown in Figure 6. The bare bones LM interface is provided by the `LanguageModel` abstract class from which the concrete implementations for the dynamic (`PpmModel`) and static (`CharNGramFstModel`) models discussed in the previous Section C.1 are derived. Each of the models implements its own input-output (I/O) mechanism and provides its own character inference engine. Each concrete model implements several prediction interfaces, such as obtaining probability distribution over the entire alphabet (`GetScores`) given the context (represented by

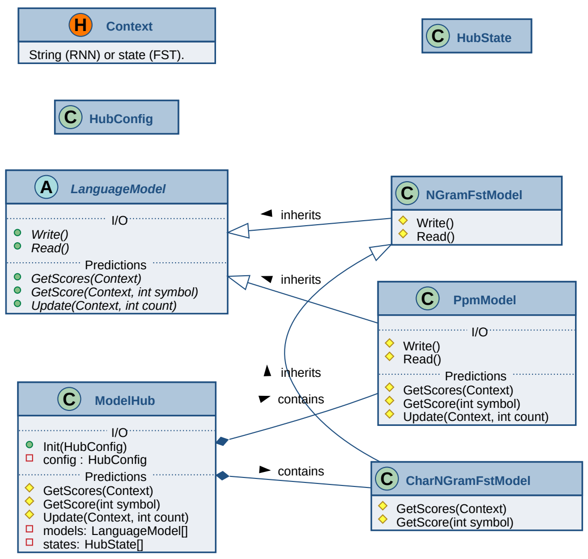

Figure 6: Simplified UML class diagram corresponding to the LM stack of configuration shown in Table 1.

the handle `Context` in the figure),[15] as well as querying for probabilities of individual symbols (`GetScore`). In addition, the dynamic model also provides a concrete implementation of dynamic updates of symbol counts (`Update`). Note that the character n-gram FST model derives from an intermediate abstract class shared by all the n-gram FST implementations (`NGramFstModel`). This class provides common functionality for representing n-grams within the FST formalism and is extended by other classes that implement character inference from more complex models, such as word-based n-grams (not shown in the figure).

The LM API that is integrated with the gRPC microservice layer is provided by the `ModelHub` class. It provides a facade over all the model and interpolation types provided by the configuration (`HubConfig`). The responsibility of this class is to manufacture the required LMs and provide a unified prediction and model update mechanism at run-time. Internally, the class maintains model and input-specific state (denoted `HubState` in the figure) for efficiency at inference time.

A simplified UML sequence diagram describing the sequence of events involved in establishing a gRPC connection with the remote LM hub (`Connect`) and performing a single inference query (`RPC GetScores`) is shown in Figure 7. In our

[14]Internally we represent the probabilities as negative log-likelihoods, hence the weights for both models in the configuration are set to be approximately equal to -0.301.

[15]The handle `Context` representing the current context in which the predictions or updates are to be made is implementation specific: for finite-state models, such as n-grams, this is simply an integer FST state ID. For neural models, this handle can point to a particular input string in a cache of histories.

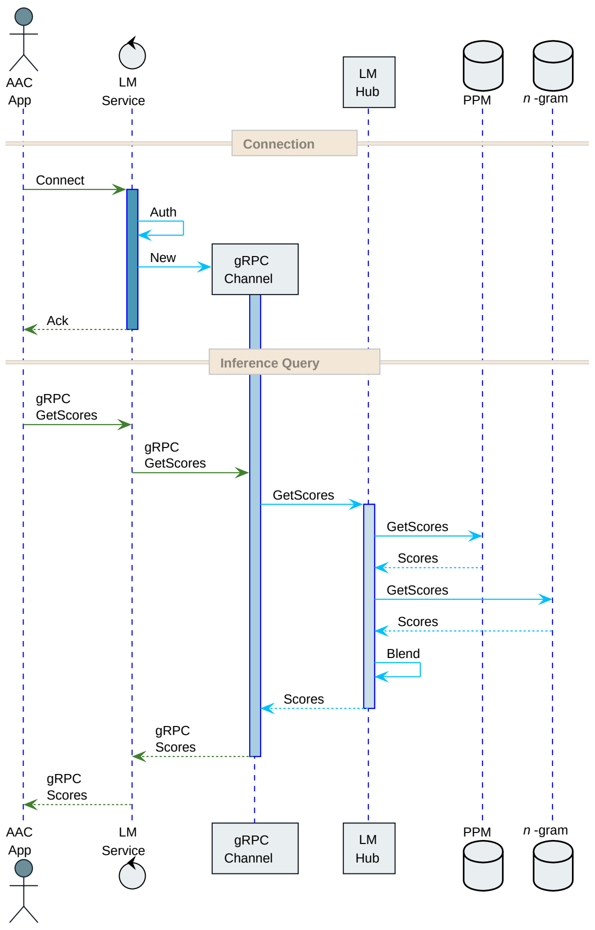

Figure 7: Simplified UML sequence diagram detailing the processing of a single inference query against the remote LM hub configured according to Table 1.

example the LM hub runs as a standalone gRPC microservice on a remote node. The network API (denoted `LM Service` in the figure) implements the RPC protocol that exposes the underlying LM hub API via portable gRPC protocol buffer messages for requests (such as `RPC GetScores`) and responses (`RPC Scores`). After authenticating the client and establishing a communication channel (denoted `gRPC Channel`) an asynchronous event loop within the service processes the incoming requests, dispatching them to LM hub engine for processing. In our example, the inference call `GetScores` to LM hub yields the predictions `Scores` blended from predictions of two models (PPM and n-gram). A gRPC response (`RPC Scores`) is then formed by the microservice and returned to the client application.

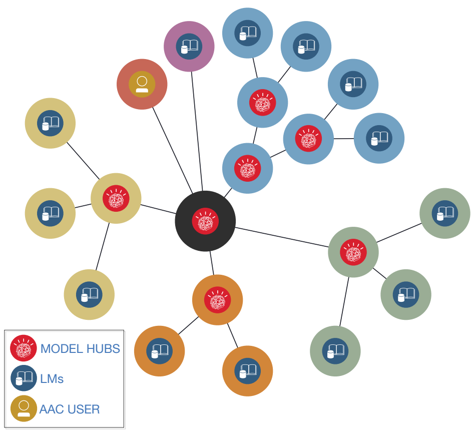

Figure 8: Hypothetical (and impractical) LM inference network over many models. The inference engines are not shown for brevity — the LM hubs directly use the models.

D Complex LM Hub Network Example

Allowing each LM hub microservice to both serve the models as well as securely communicate to other LM hub microservices allows one to construct significantly more complex inference architectures than the ones described above. One such hypothetical configuration is shown in Figure 8. In this example, the user interacts with a local hub (shown in a black node) that serves one local dynamic model (shown in a purple node). The local hub mixes the predictions from this dynamic model with the multiple predictions arriving via a complex network of other model hubs. The groups of nodes (hubs) and leafs (models) with the same color may denote the external organization providing the model service and hosting, specific types of models[16] or the provenance of the data the models were trained on (e.g., the medical domain). Needless to say, the LM microservices architectures of such complexity are unlikely to be required in practice, yet constructing configurations along the lines of the one shown in Figure 8 is definitely a feasible task in our framework.

[16]Such as XLNet (Yang et al., 2019), character-based BERT (El Boukkouri et al., 2020) and simpler transformer architectures (Al-Rfou et al., 2019).