What do tokens know about their characters and how do they know it?

Anonymous ACL submission

Abstract

Pre-trained language models (PLMs) that use subword tokenization schemes can succeed at a variety of language tasks that require character-level information, despite lacking explicit access to the character composition of tokens. Here, studying a range of models (e.g., GPT-J, BERT, RoBERTa, GloVe), we probe what word pieces encode about character-level information by training classifier to predict the presence or absence of a particular alphabetical character in an English-language token, based on its embedding (e.g., probing whether the model embedding for "cat" encodes that it contains the character "a"). We find that these models robustly encode character-level information and, in general, larger models perform better at the task. Through a series of experiments and analyses, we investigate the mechanisms through which PLMs acquire character information during training and argue that this knowledge is acquired through multiple phenomena, including a systematic relationship between particular characters and particular parts of speech, as well as natural variability in the tokenization of related strings.

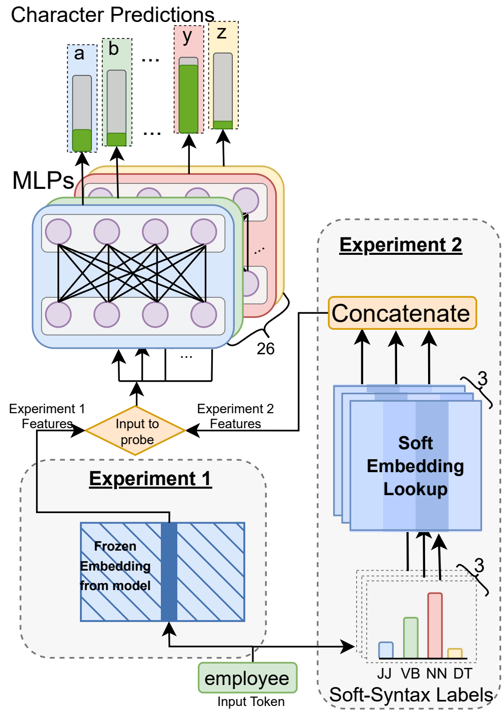

Figure 1: Overview of our probing setup. In Experiment 1, the input is a model embedding and we train MLPs to classify whether a particular character (e.g., "a") occurs in a particular token (e.g, "employee"). In Experiment 2, we use syntactic features as input, rather than model embeddings, to train our probe.

1 Introduction and Motivation

The dominant class of models in NLP (pre-trained transformer models; Brown et al., 2020; Devlin et al., 2019; Bommasani et al., 2021) use tokenization schemes, like BPE or WordPiece tokenization (Sennrich et al., 2015; Schuster and Nakajima, 2012; Kudo and Richardson, 2018), that break text into word pieces. These models face an apparent limitation in that they do not have access to information below the level of the word piece, such as information about characters. But character-level information has been claimed to be useful for a variety of tasks, including adapting text to novel domains like biomedicine, texts with misspellings, and wordplay-based tasks that require attention to character-level manipulations (Riabi et al., 2021; El Boukkouri, 2020; Clark et al., 2021).

But there are drawbacks to using character-level models: character-based sequences are long and therefore can slow down training (Mielke et al., 2021). And giving including character-level information does not necessarily improve performance on tasks where one might expect it to (Libovickỳ et al., 2021; Rosales Núñez et al., 2021; Itzhak and Levy, 2021). Therefore, the vast majority of top-performing models in languages with alphabetic scripts use models with various kinds of subword tokenization schemes (e.g., Devlin et al., 2019; Brown et al., 2020), but rarely with character-level schemes.

One possible explanation for this state of affairs is that models trained on word pieces implicitly learn something about characters, making the explicit inclusion of character-level information unnecessary. Indeed, recent work has shown that even models based on subword tokens might be able to use and manipulate character-level information. Rozner et al. (2021) and Efrat et al. (2021) both study cryptic crosswords and find that PLMs (specifically, T5) can take advantage of character-level information in order to solve wordplay tasks like unscrambling scrambled words. Itzhak and Levy (2021) show that RoBERTa can access subword information by testing it on a spelling task that requires it to map from words to characters (e.g., from *cat* to the characters *c* + *a* + *t*).

The fact that models can do tasks like this is curious: word pieces have no explicit access to character information during training, and the mechanism by which they acquire such information is not obvious. The goal of this paper is to understand the nature of this information, and how it is learned.

Thus, we make several contributions. First, we provide a thorough characterization of what character information is accessible to subword-tokenized PLMs by designing a binary probing task (§3) to probe subword tokens for the presence or absence of a particular character: e.g., does the sequence *star* contain the letter *t*? This task lets us not just assess whether this information is available, but lets us characterize, in a fine-grained way, the nature of character-level knowledge in subword tokens. We find performance far above a controlled baseline (an F1 score of 93.7 for the best-performing model, GPT-J), suggesting that subwords learn meaningful information about their characters.

To explore how this information is acquired, we introduce several possible explanations and conduct detailed analyses of the probing task (§3.3). Specifically, we consider how character knowledge varies as a function of the English character being probed for (it's easier to classify rare letters than common ones), the position in the token of the character in question (performance is somewhat better early in tokens), and the frequency of the token (frequent tokens aren't necessarily easier to probe). We then turn to the possibility than systematic correspondences between English characters and syntactic features (e.g., adverbs tend to end in "y"), play a role in how models acquire character-level information. To that end, we devise syntactic baselines, whereby we use features like part of speech as input to the classifer for detecting the presence of absence of tokens (§4). The probe performs much better than control tasks, which suggests syntactic features contribute to the tokenizer's performance. However, this correlation does not suffice to explain the totality of character information learned by PLMs.

Finally, we consider another possible mechanism, based on the variability of tokenization, by which character-level information might be learned (§5). We conduct an experiment using simple fixed embeddings, as proof of concept that increasing variability in tokenization (Cao and Rimell, 2021) affects the character information learned. Overall, given the importance of tokenization schemes for downstream performance (Bostrom et al., 2021; Mielke et al., 2021), we believe this knowledge could inform the development of tokenization schemes that improve model performance.

2 Prior work

All language models must choose what to use as the basic linguistic unit, and, as a result, there is a long history of work in NLP, evaluating the tradeoffs between models that tokenize words based on characters, words, or something in between, like bytes or word pieces (see Mielke et al., 2021; Pinter, 2021, for recent surveys).

While words are a seemingly natural kind and are often used as basic units for modeling language, there is considerable debate in the linguistics literature as to how to even define a word, due to differences across languages (Haspelmath, 2017). Moreover, word-level models have a major weakness in that they do not naturally handle out of vocabulary items (see Jurafsky, 2003, for an overview) and can have very different behaviors in languages with different morpohological systems (Mielke et al., 2019; Cotterell et al., 2018). Character-level models have their own weaknesses: they are typically slower to train at the scale required for massive language modeling. Many recent efforts have centered around trying to use meaningful sub-word units in language modeling, such as BPE (Gage, 1994; Sennrich et al., 2015), WordPiece tokenization (Schuster and Nakajima, 2012), and UnigramLM (Kudo, 2018).

While subword tokenization schemes often end up with reasonable linguistic units, they still lack access to character-level information. So there have been a number of efforts to imbue word or subword tokenization schemes with character-level information (Mielke and Eisner, 2019; Kim et al., 2016; Dos Santos and Zadrozny, 2014; Bojanowski et al., 2017; Li et al., 2018; Ma and Hovy, 2016; Aguilar et al., 2020; El Boukkouri, 2020; Clark et al., 2021). But, if models trained on subword tokens implicitly learn character-level information during training, there may be less of a need to supplement them with explicit information.

To shed new light on these questions, we use probing, which is widely used to assess what information is contained in PLM embeddings. (Belinkov, 2021; Belinkov and Glass, 2019; Hewitt and Manning, 2019; Hupkes et al., 2018). Because probing has limitations (Elazar et al., 2021; Pimentel et al., 2020; Voita et al., 2021), we use a number of control tasks (Hewitt and Liang, 2019) and baselines in order to ask what can be recovered from embeddings, relative to a control of equal expressive power.

3 Experiment 1: Probing for character information

The main goal of our first experiment is to quantify the extent to which tokens in PLMs capture character-level information and characterize that knowledge across a variety of dimensions. We train a binary classifier probe that takes as input a token's frozen embeddings from a PLMs to predict whether a particular character of the English alphabet is contained in that token. That is, if successful, the probe will predict that *cool* contains an "o" but "cat" does not. We also consider a task in which the probe must say whether one token (e.g., "coo") is a substring of another token (e.g., "cool"). We examine the probe's success as a function of the character being probed for, length of the token being probed, position of the character in the token, and frequency of the token.

3.1 Method

We consider the static non-contextualized embeddings of PLMs: GPT-J (Wang and Komatsuzaki, 2021), GPT-2 (Radford et al., 2019), RoBERTa (Liu et al., 2019), BERT (cased and uncased; Devlin et al., 2019), as well as GloVe embeddings (Pennington et al., 2014) and Language-only embeddings of the multimodal LXMERT (Tan and Bansal, 2019). See Appendix B for model details.

Each language model has its own vocabulary, consisting of tokens. We consider only the tokens consisting entirely of characters in the standard English alphabet (a-z), along with the special characters that accompany these tokens, such as preceding whitespace (denoting by Ġ in the RoBERTa and GPT-family) or symbols denoting continuations of preceeding word ('##' in BERT family).

Our probing task trains classifiers to detect the presence or absence of each of the 26 English alphabets α over each token w_i from the filtered-vocabulary V. Thus, a separate dataset for each alphabet α is constructed over V as $D'_\alpha = \{(w_1, y_1), (w_2, y_2), \dots (w_d, y_d)\}$ where the binary label y_i denotes whether α occurs at least once in $w_i \in V$. From these data-points in D'_α we create a balanced dataset D_α with equal number of positive and negative labels by undersampling the (w_i, y_i) points with y_i as the negative label (i.e., when probing for the presence of the character "z", half the tokens will contain "z" even though most tokens in general do not). We then split D_α into training and test splits in a roughly 80-20 ratio, while ensuring that tokens with the same lemma appears in the same split. This is the most challenging split, as it prevents the probe from leveraging wordform similarity across words with the same lemma in both training and test (Itzhak and Levy, 2021).

We train our probe over the static non-trainable embeddings E of these PLMs. For a data-point (w_i, y_i), the probe receives as input a token w_i. The probe predicts logits $\hat{y}_i$ by an MLP: $\hat{y}_i = \sigma(MLP_\alpha(E^T x_i))$. In the control task, we consider randomly-initialized non-trainable embeddings instead of the trained embeddings from the PLMs.

Substring Sub-experiment As an additional subexperiment for assessing the generalizability of the task, for the best-performing model (GPT-J), we consider a related substring classification task. Specifically, we probe GPT-J's embedding to detect whether a token u is a substring of the token v. That is, can it detect that the token "ome" is a substring of "some"? For this condition, we set up the experiment as before but, rather than attempt to detect the presence or absence of a character, we seek to classify whether a particular token u_i is a substring of another token v_i. To create positive examples, we consider all substrings of v_i that are in the overall vocabulary V. For each positive example, we sample a token from V of equal character length as u_i which is *not* a substring of v_i in order to create negative examples. This creates a balanced set, from

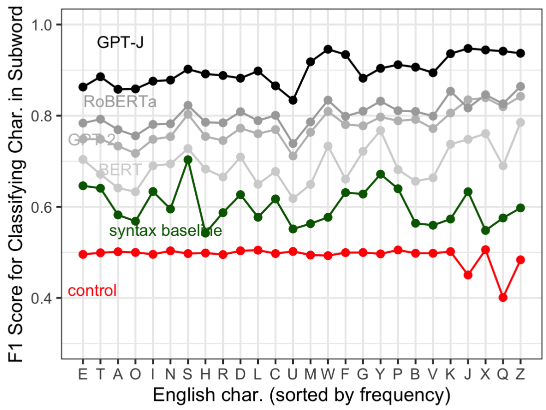

Figure 2: For selected models, the average F1-score (y-axis) for how well a character (x-axis) can be classified on our main probing task. The control (random embeddings) appears in red, the syntax baseline in green, and the 4 models shown in grayscale, with the largest and most recent model (GPT-J) in the darkest color.

Model type	PLM	Control
Main Probing Experiment		
GPT-J	93.70	48.36
GPT-2	84.25	52.31
RoBERTa	86.41	47.33
BERT-Cased	78.50	47.08
BERT-Uncased	77.48	49.37
GloVe 300D	67.57	49.57
GloVe 100D	66.04	50.33
LXMERT	62.4	53.92
Substring Sub-Experiment		
GPT-J	86.56	70.03

Table 1: Results for the main probing experiment.

which we sample an 80-20 train-test split, ensuring that the superset token v_i always occur in the same split. We train the probe as before, with the input as the concatenated embeddings of the two tokens.

3.2 Results

Main Character Probing Results Table 1 shows the results averaged across 5 train-test splits and different seeds, reporting on the Macro-F1 metric averaged across all 26 characters. We also observe very low variance for the strong performing models, as shown in the Appendix (Table 6).

For our main character probing experiment, all models perform substantially better than their matched controls (which hover around 50, which is chance level), suggesting that word piece tokens from PLMs store information about their constituent characters in their embeddings. GPT-J is the best-performing model (with F1 of 93.70 and 94.35), followed by RoBERTa and GPT-2, then the BERT models. All the transformer models outperform the GloVe fixed embedding model. Clearly, the performance of the models on this probing task correlates with performance on other language tasks, such that larger models trained on larger corpora do better.[1]

There are also other factors that may contribute to difference in performance such as the nature of the pre-training task and the tokenizer. The latter is evidence from the considerable performance gap between RoBERTa and BERT, which may be partially attributed to RoBERTa using GPT's reversible tokenizer, leading to more variability depending on preceeding whitespace. (See §5 for the potential effect of tokenizer variability on performance.)

Substring Experiment Performance on the Substring Experiment is also far above chance, with an average F1 of 86.56, compared to a control F1 (on random embeddings) of 70.03 (bottom row in Table 1). Control performance is well above 50 in this case since the data set is created to be balanced such that the superstrings have equal numbers of positive and negative examples. But there are still baseline differences in how often a token occurs as a substring, so the model can learn that certain substrings like "en" are more common than substrings like "emies". We take the performance on the Substring Experiment as evidence that the model can make use of character information to do more complicated substring tasks than just character identification.

3.3 Breakdown of results

Next, we consider a number possibilities for how character-level information gets into these embeddings and conduct analyses intended to understand the nature of the information learned and how it gets there.

Is the first letter learned best because of alphabetization? One possibility is that, because the training data likely contains many alphabetical lists and other kinds of word lists (e.g., lists of words starting with "z"), the model learns a co-occurrence relationship between words that start with the same character. We would predict that this would cause stronger performance when the probed character

[1]Since performance varies considerably based on the model used, we consider this work an additional data point for the argument that one should consider multiple models in interpretability work (Bowman, 2021).

occurs at the beginning of the word. To that end, we examine how the model's performance varies as a function of where in the token the target character is (top panel in Figure 3). While there is indeed a significant negative relationship between word position and recall as measured by a linear regression (β = -.01, p<.001), the slope is relatively small. While recall on the first letter in a token is high (95.2), it is not an outlier: performance is only somewhat higher than recall for the second character (94.5). Moreover, performance is above chance even when the target character appears 10 or more characters deep in a token. Therefore, we do not believe the effect is driven only by word beginnings, although they likely play a role.

Is it only frequent words that the probe gets right? Next, we consider whether performance varies as a function of the frequency of the token (middle panel in Figure 3). One possibility could be that character information is memorized only in high-frequency tokens like "the", which occur often enough that at least some of the time very frequent token will occur broken down into characters (e.g., "the" appearing in the context of "t h e"), and that low-frequency tokens will perform worse. This does not appear to be the case and, in fact, there is, if anything, a negative relationship ($\beta = -.013$, p=.05) between binned log frequency and performance, such that less frequent tokens are easier to attain character information from.

Is it easier to get long or short words right? The bottom panel of Figure 2 shows F1-score as a function of the length of the token. Using the GPT-J embeddings, it is easier to classify characters in short tokens, as compared to longer tokens. This may be a function of the nature of the task since there is, in some sense, less information to be represented for a short token like "be" for the purposes of the task (just that it contains a "b" and it contains an "e"), whereas a long token would have to represent information about more characters.

Which characters are learned best? Part of what makes the success of the probe is that word embeddings represent word co-occurrence information, which is typically conceived of as syntactic and semantic in nature (Erk, 2016) and so should, because of the arbitrariness of the relationship between forms and meanings (Saussure, 1916; Hockett, 1960), mean there is no relationship between individual characters and information learned by embeddings. But this arbitrariness breaks down, in that there are statistically detectable non-arbitrary form-meaning relationships in language (Blasi et al., 2016; Monaghan et al., 2014; Tamariz, 2008; Dautriche et al., 2017; Pimentel et al., 2019), such as the fact that *fl-* words in English tend to be about movement (e.g., *flap*, *fly*, *flutter*, *flicker*; Marchand, 1959; Bergen, 2004) and that different parts of speech have different phonological patterns (Dautriche et al., 2015; Kelly, 1992; Monaghan et al., 2005). An even larger source of shared information between characters and syntactic/semantic information is that morphological forms can be cues to word categories: for instance, most plural nouns end with "s" and many adverbs end in "ly". This leads to changes in character-level distributions: while roughly 12% of words in American English contain "y", 85% of adverbs do (as estimated using data from Brysbaert et al., 2012). Thus, a model with access to part of speech information could do well by guessing that all adverbs contain "y".

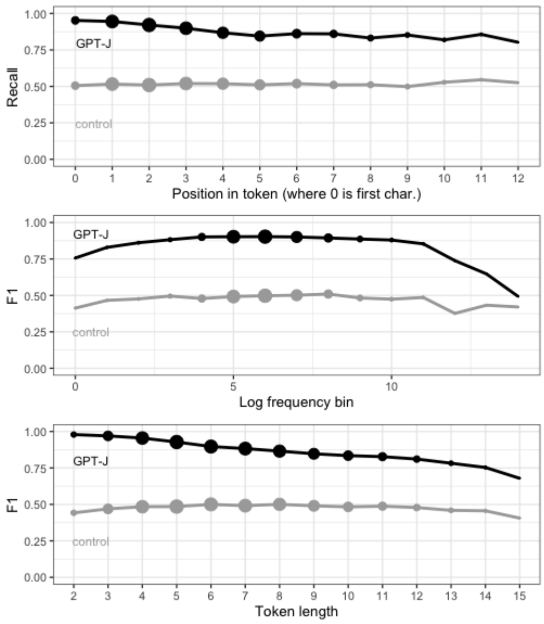

Figure 3: Performance on the GPT-J probe, relative to a control probe, as a function of the character's position in the token (top), the log frequency of the token (middle), and the length of the token (bottom). The size of the point reflects the amount of data.

So one possibility is that the probe's performance is largely driven by characters that correlate with syntactic and semantic features. If this were the case, we might expect some characters to show much better performance than others. Figure

Measure	SpaCy	GPT-J	Control
Aggregate Performance			
F1	52.34	**61.24**	49.68
Best performing characters			
s	64.60	**66.82**	40.32
y	61.96	**64.89**	48.68
e	62.05	**62.32**	47.27
Worst performing characters			
b	48.92	**55.13**	48.25
m	48.13	**55.61**	46.11
q	43.79	**53.54**	49.28

Table 2: The best and worst performing characters from Experiment 2 on the SpaCy syntactic baseline, the GPT-J syntactic baseline, and the Control.

2 shows the F1-Macro as a function of character. For GPT-J, the best-performing model, there are some clear trends. For instance, it is easiest to classify rare letters: J, W, X, Q, Z all have F1-scores over 93. And it is hardest for the probe to classify vowels: U, A, O, and E are the lowest performing characters between 83 and 86. But even those lower-performing characters do far better than the chance baseline (at about 50 F1 score)

To further explore this, we conducted a qualitative analysis of the probe's successes and failures. Consider the probe for classifying the presence/absence of "y": the model assigns highest confidence to the following 4 tokens: "lly", " selectively", " subtly", " mechanically". These all have "ly" endings, which in English is typically associated with adverbs. Similarly, the top performing tokens for the "s" classifier all end with a morphologically meaningful "-s" suffix: " socialists", " stocks"," suggestions".

This analysis suggests that the strong classifier performance could be explained by the model learning systematic relationships between certain characters and syntactically or semantically meaningful morphology. Is syntactic information the window through which character-level information enters PLMs? To address that question, our next experiment focuses on a syntactic baseline, to see how well character-level information can be predicted based on syntactic features.

4 Experiment 2: The effect of syntactic information

In this experiment, we focus on building probes for the same task as in Experiment 1 (identifying whether a particular character occurs in a particular token). But, rather than using the token embeddings from a large language model as input, we attempt to classify the presence/absence of characters in a token based on syntactic information.

Our first model (the SpaCy model) uses SpaCy (Honnibal and Montani, 2017) to obtain distributions over features for each token in the vocabulary: Fine-Grained Part of Speech tag (PoS; e.g., for "Jane", NNP for a proper noun), Coarse-Grained Part of Speech tag (Coarse-grained PoS; e.g., for "Jane", PROPN for proper noun), and a Named Entity Recognition tag (NER; e.g., for "Jane", PERSON for a personal name). We use these features to construct a syntactic vector for each token.

Because SpaCy is built to operate over words, not tokens, we also construct custom syntactic baselines that can tag subwords, as opposed to tokens.

The performance of these probes will serve as a baseline for ascertaining how much character-level information can be learned by these features alone, without a full language model. If they can perform just as well as the full GPT-J embeddings, that would suggest that morphosyntactic information (of the sort that we already know is learned by PLMs during pretraining) is sufficient for the performance on the probing task.

The method is the same as in Experiment 1, where the goal is to predict the presence or absence of a character α in a token, except that instead of using the token's model embeddings as input, we instead use syntactic feature vectors (obtained either from SpaCy or a custom tagger) as input. We describe these syntactic vectors below.

Syntactic baselines The SpaCy model has 3 features for each token: NER, PoS, and Coarse-Grained PoS tags. The resultant features are discrete one-hot feature vectors over labels.

The custom syntactic tagger, which solves the problem that SpaCy tags words, not subword tokens, takes a (subword) token's model embedding as input and outputs a vector of probabilities over part of speech and named entity categories. Here, we describe results for our custom GPT-J Tagger, trained using GPT-J model embeddings, since GPT-J is the best-performing of our models for our main task. See Appendix C for descriptions and the results for 2 additional BERT-based custom taggers that we built.

To build our custom GPT-J-Tagger, we train an MLP model to predict PoS and NER label based on GPT-J's static embedding layer for each token. The tagger is trained on the CoNLL 2003 dataset's train and valid splits (Sang and De Meulder, 2003), which contains part of speech and named entity information. Unlike the SpaCy tagger, our custom GPT-J-Tagger outputs a probability distribution over categories. We use this distribution over labels as input, rather than a one-hot vector. In the Appendix, Table 10 shows the performance of the tagger's performance *qua* tagger.

Probing for characters using syntactic baselines We run the character probing experiment as before. But, rather than using the model embeddings, we use the syntactic feature vectors as the target of our probe. Table 2 shows the results of these experiments. Using the syntactic baselines leads to substantially improved performance over control tasks, and the GPT-J-Tagger does better than the SpaCy tagger. We hypothesize that this is because the custom GPT-J-Tagger is better suited to handling subwords, and because it enables us to use label distribution rather than one-hot vectors. Zooming in on the performance over individual characters, we observe that some English alphabets consistently perform much better when using the syntactic features, than the control task. As predicted, these are precisely the characters that are highly correlated with particular parts of speech. The best performing characters are: "s" (associated with plural nouns and third-person singular verbs) and "y" (associated with adjective and adverb endings). Thus, the syntactic baselines seem to be capturing the information that they were intended to capture. But their performance still fell far below the best performing PLMs, suggesting that the large models are capturing more than just the information captured by the syntactic models. Moreover, as can be seen in Figure 2, the syntax baseline shows a sharp peak for morphologically informative characters like "s", but this pattern is much weaker in GPT-J (which shows only a slight performance increase for "s"). Therefore, we do not think syntactic information can explain all the character information learned by PLMs. In the next section, we consider another possibility: variability of tokenization, the focus of the next section.

5 Experiment 3: Tokenization variability

Here, we posit that the variability of tokenization is another avenue by which character-level information could be learned by models. We first quantify this variability and then run an experiment using CBOW Word Embeddings (Mikolov et al., 2013) showing how increasing the variability in tokenization can lead to more character information being learned.

Word	Tokenizations
"dictionary"	"d + ictionary"
" dictionary"	" dictionary"
"dictionaries"	"d + iction + aries"
" dictionaries"	" diction + aries"
"dicionary"	"d + icion + ary"

Table 3: Some GPT tokenizations for "dictionary".

Subword tokenization like the one used by GPT models can cause the same lemma to have very different tokenizations, depending on its form and/or its spelling. See Table 3 for possible tokenizations of "dictionary" and related forms, including a misspelling (bottom row). This is a subset of the possible misspellings, variants, and morphological forms of the word. But the listed forms alone generate 8 unique tokens.

It would be useful for the model to learn a relationship between all these tokens, since they represent the same lemma. We posit that the desirability of learning this mapping is a mechanism by which character information could be learned, by inducing an objective to map between atomic tokens like ‘ dictionary’ and the various substring tokens that can arise. While each of these mappings could be learned individually, learning character-level spelling information offers a more general solution to the problem, such that even a completely tokenization could be interpreted by composing characters.

For this to be plausible, though, variable tokenizations like this must be frequent enough for it to matter. In Appendix D, we use heuristics to identify different forms in which a word appears and conduct a series of back-of-the-envelope calculations to determine how many different unique tokenizations are expected for a long word (8+ characters) like *dictionary*, in all its variant forms and misspellings in a sample of the Pile corpus (we used 1/6 of the corpus as a sample; Gao et al., 2020). We found that, on average, we should expect over 200 different tokenizations for a word like "dictionary", many which have no tokens in common.

This result leads to a prediction: increasing the variability of tokenization should increase the amount of character-level information learned. To test this, we train models using tokenization schemes with different levels of variability and then test how much character-level information they learn, using our probing task.

Tokenization	ρ	Embedding	Control
Word	-	60.55	47.12
GPT-J	-	63.23	47.51
GPT-J	0.05	**66.00**	47.23
GPT-J	0.1	65.64	46.72
GPT-J	0.2	64.23	47.01
GPT-J	0.5	62.33	46.47

Table 4: Average F1 scores for probing results, as a function of change in tokenization variability

Because the overall goal of our paper is to characterize and explain the nature of character-level information learned, and not to use it to build a better model, we conduct a proof-of-concept experiment using CBOW Word Embeddings (Mikolov et al., 2013) on a portion of the Pile corpus with 1.1B characters, as opposed to training a large transformer model from scratch varying tokenization schemes. We train 6 CBOW models from scratch, each with a different tokenization scheme. As baselines, we consider vanilla rule-based word-tokenization (the CBOW default, labeled "Word" in Table 4) and GPT-J's default word piece tokenization scheme. Comparing these two baselines against each other lets us compare the effect of word tokenization vs. subword tokenization on character information. But our key manipulation is to consider variations of GPT-J's tokenizer in which we systematically increase tokenization variability. In pre-processing the word-tokenized corpus for input, for each word token w_i, with probability $(1-\rho)$, we tokenize it using the standard GPT-J tokenizer. Under the standard tokenizer, " schematics" becomes " sche + mat + "ics". With probability ρ, however, we tokenize w_i using a random tokenization that consists of alternative valid tokens from GPT-J. So, " schematics" could become " schema + tics" or " schematic + s" (but not " schemati + cs" since " schemati" is not a valid GPT token). We vary ρ from 0.05 to 0.5. See Appendix D for more details on this procedure. The result is a series of tokenized corpora, which have more variable tokenization than the vanilla GPT-J-tokenized corpus.

We train CBOW models, separately for each of these corpora. Table 4 shows the results of these experiments on our probing task (using the same method as in Experiment 1). As expected, probes on the subword tokenization schemes reveal they learn more information about characters than the default word-level tokenizer. Most importantly, upon increasing the variability on GPT-J's tokenization scheme, the performance of the probe increases, peaking at $\rho = 0.05$ and $\rho = 0.1$. Thereafter, the performance decreases with variability, suggesting that increasing variability leads to increased character knowledge but only up to a point, like because there is a tradeoff: since the corpus size for the toy experiment is small, having very high variability leads to the model seeing fewer instances of each token.

While the magnitude of these differences are relatively small, they are consistent across random seeds and train-test splits. Thus, we believe that these results offer proof of concept that (a) the variability of tokenization affects how much character information is learned by PLMs and (b) that increasing tokenization variability could be a means by which PLMs could be built to learn more character-level information.

6 Discussion and Conclusion

Overall, our results suggest a possible explanation for why efforts to infuse subword models with character-level information may not be necessary: the information already gets learned during training through a variety of methods. Insofar as these methods (e.g., tokenizer variability) can be manipulated in model construction, this knowledge could be used to build models that perform better at tasks dependent on such knowledge. In future work, we believe it will be important to test the generalizability of these results in languages other than English. Given the particular importance of tokenization in multilingual models (Rust et al., 2021; Singh et al., 2019), it would be fruitful to consider these results in multilingual settings.

More generally, while the linguistic capabilities of PLMs are much studied (Rogers et al., 2020; Bommasani et al., 2021), the question whether PLMs learn the constituent characters of tokens is of a different nature in that it depends on learning a property of language (spelling) that is not explicitly tied to meaning. There is no *a priori* reason "dog" is spelled "D-O-G", and, in a sense, the spelling of the word does not matter. But, in another sense, it *does* matter: humans routinely use language in creative and character-dependent ways: e.g., alphabetizing text, scrambling letters to create codes, and solving crossword puzzles. Understanding whether the building blocks of this knowledge can emerge during self-supervised training on a word prediction task could be of interest not just in NLP, but in the cognitive sciences.

7 Ethics and Broader Impacts

This work consists of probing experiments and interpretability analyses of PLMs, and the risks and ethical considerations are largely those that affect any work with large PLMs (e.g., energy costs; see Bommasani et al., 2021, for an overview of risks and tradeoffs). The intended use of our code is for academic research. We consider probing publicly available PLMs, which are made publicly available in part for research purposes, to be within the intended use of PLMs.

References

Gustavo Aguilar, Bryan McCann, Tong Niu, Nazneen Rajani, Nitish Keskar, and Thamar Solorio. 2020. Char2subword: Extending the subword embedding space from pre-trained models using robust character compositionality. *arXiv e-prints*, pages arXiv–2010.

Yonatan Belinkov. 2021. Probing classifiers: Promises, shortcomings, and alternatives. *arXiv preprint arXiv:2102.12452*.

Yonatan Belinkov and James Glass. 2019. Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72.

Benjamin K Bergen. 2004. The psychological reality of phonaesthemes. *Language*, 80(2):290–311.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Damián E Blasi, Søren Wichmann, Harald Hammarström, Peter F Stadler, and Morten H Christiansen. 2016. Sound–meaning association biases evidenced across thousands of languages. *Proceedings of the National Academy of Sciences*, 113(39):10818–10823.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Kaj Bostrom, Xinyu Zhao, Swarat Chaudhuri, and Greg Durrett. 2021. Flexible generation of natural language deductions. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6266–6278, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Samuel R Bowman. 2021. When combating hype, proceed with caution. *arXiv preprint arXiv:2110.08300*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Marc Brysbaert, Boris New, and Emmanuel Keuleers. 2012. Adding part-of-speech information to the subtlex-us word frequencies. *Behavior research methods*, 44(4):991–997.

Kris Cao and Laura Rimell. 2021. You should evaluate your language model on marginal likelihood over tokenisations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2104–2114, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jonathan H Clark, Dan Garrette, Iulia Turc, and John Wieting. 2021. Canine: Pre-training an efficient tokenization-free encoder for language representation. *arXiv preprint arXiv:2103.06874*.

Ryan Cotterell, Christo Kirov, Mans Hulden, and Jason Eisner. 2018. On the complexity and typology of inflectional morphological systems. *Transactions of the Association for Computational Linguistics*.

Isabelle Dautriche, Kyle Mahowald, Edward Gibson, Anne Christophe, and Steven T Piantadosi. 2017. Words cluster phonetically beyond phonotactic regularities. *Cognition*, 163:128–145.

Isabelle Dautriche, Daniel Swingley, and Anne Christophe. 2015. Learning novel phonological neighbors: Syntactic category matters. *Cognition*, 143:77–86.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Cicero Dos Santos and Bianca Zadrozny. 2014. Learning character-level representations for part-of-speech tagging. In *International Conference on Machine Learning*, pages 1818–1826. PMLR.

Avia Efrat, Uri Shaham, Dan Kilman, and Omer Levy. 2021. Cryptonite: A cryptic crossword benchmark for extreme ambiguity in language. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4186–4192, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hicham El Boukkouri. 2020. Ré-entraîner ou entraîner soi-même ? stratégies de pré-entraînement de BERT en domaine médical (re-train or train from scratch ? pre-training strategies for BERT in the medical domain). In *Actes de la 6e conférence conjointe Journées d'Études sur la Parole (JEP, 33e édition), Traitement Automatique des Langues Naturelles (TALN, 27e édition), Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RÉCITAL, 22e édition). Volume 3 : Rencontre des Étudiants Chercheurs en Informatique pour le TAL*, pages 29–42, Nancy, France. ATALA et AFCP.

Yanai Elazar, Shauli Ravfogel, Alon Jacovi, and Yoav Goldberg. 2021. Amnesic probing: Behavioral explanation with amnesic counterfactuals. *Transactions of the Association for Computational Linguistics*, 9:160–175.

Katrin Erk. 2016. What do you know about an alligator when you know the company it keeps? *Semantics and Pragmatics*, 9:17–1.

Philip Gage. 1994. A new algorithm for data compression. *C Users Journal*, 12(2):23–38.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The pile: An 800gb dataset of diverse text for language modeling.

Martin Haspelmath. 2017. The indeterminacy of word segmentation and the nature of morphology and syntax. *Folia Linguistica*, 51(s1000):31–80.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743, Hong Kong, China. Association for Computational Linguistics.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, Minneapolis, Minnesota. Association for Computational Linguistics.

C.F. Hockett. 1960. The origin of language. *Scientific American*, 203(3):88–96.

Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear.

Dieuwke Hupkes, Sara Veldhoen, and Willem Zuidema. 2018. Visualisation and 'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research*, 61:907–926.

Itay Itzhak and Omer Levy. 2021. Models in a spelling bee: Language models implicitly learn the character composition of tokens. *arXiv preprint arXiv:2108.11193*.

Daniel Jurafsky. 2003. Probabilistic modeling in psycholinguistics: Linguistic comprehension and production. In R. Bod, J. Hay, and S. Jannedy, editors, *Probabilistic Linguistics*. MIT Press.

Michael H. Kelly. 1992. Using sound to solve syntactic problems: The role of phonology in grammatical category assignments. *Psychological Review*, 99(2):349–364.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Thirtieth AAAI conference on artificial intelligence*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Bofang Li, Aleksandr Drozd, Tao Liu, and Xiaoyong Du. 2018. Subword-level composition functions for learning word embeddings. In *Proceedings of the second workshop on subword/character level models*, pages 38–48.

Jindřich Libovickỳ, Helmut Schmid, and Alexander Fraser. 2021. Why don't people use character-level machine translation? *arXiv preprint arXiv:2110.08191*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354*.

Hans Marchand. 1959. Phonetic symbolism in english wordformation. *Indogermanische Forschungen*, 64:146.

Sabrina J Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y Lee, Benoît Sagot, et al. 2021. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp. *arXiv preprint arXiv:2112.10508*.

Sabrina J. Mielke, Ryan Cotterell, Kyle Gorman, Brian Roark, and Jason Eisner. 2019. What kind of language is hard to language-model? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4975–4989, Florence, Italy. Association for Computational Linguistics.

Sabrina J Mielke and Jason Eisner. 2019. Spell once, summon anywhere: A two-level open-vocabulary language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6843–6850.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space.

Padraic Monaghan, Nick Chater, and Morten H. Christiansen. 2005. The differential role of phonological and distributional cues in grammatical categorisation. *Cognition*, 96(2):143–182.

Padraic Monaghan, Richard C. Shillcock, Morten H. Christiansen, and Simon Kirby. 2014. How arbitrary is language. *Philosophical Transactions of the Royal Society B*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Tiago Pimentel, Arya D McCarthy, Damián E Blasi, Brian Roark, and Ryan Cotterell. 2019. Meaning to form: Measuring systematicity as information. *arXiv preprint arXiv:1906.05906*.

Tiago Pimentel, Josef Valvoda, Rowan Hall Maudslay, Ran Zmigrod, Adina Williams, and Ryan Cotterell. 2020. Information-theoretic probing for linguistic structure. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4609–4622, Online. Association for Computational Linguistics.

Yuval Pinter. 2021. Integrating approaches to word representation. *arXiv preprint arXiv:2109.04876*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Arij Riabi, Benoît Sagot, and Djamé Seddah. 2021. Can character-based language models improve downstream task performances in low-resource and noisy language scenarios? In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, pages 423–436, Online. Association for Computational Linguistics.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

José Carlos Rosales Núñez, Guillaume Wisniewski, and Djamé Seddah. 2021. Noisy UGC translation at the character level: Revisiting open-vocabulary capabilities and robustness of char-based models. In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, pages 199–211, Online. Association for Computational Linguistics.

Joshua Rozner, Christopher Potts, and Kyle Mahowald. 2021. Decrypting cryptic crosswords: Semantically complex wordplay puzzles as a target for nlp. In *NeurIPS 2021*, Proceedings of Machine Learning Research.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. 2021. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3118–3135, Online. Association for Computational Linguistics.

Erik F Sang and Fien De Meulder. 2003. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*.

F. de Saussure. 1916. *Course in general linguistics*. Open Court Publishing Company.

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

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Jasdeep Singh, Bryan McCann, Richard Socher, and Caiming Xiong. 2019. Bert is not an interlingua and the bias of tokenization. In *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*, pages 47–55.

Monica Tamariz. 2008. Exploring systematicity between phonological and context-cooccurrence representations of the mental lexicon. *The Mental Lexicon*, 3(2):259–278.

Hao Tan and Mohit Bansal. 2019. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490*.

Elena Voita, Rico Sennrich, and Ivan Titov. 2021. Analyzing the source and target contributions to predictions in neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1126–1140, Online. Association for Computational Linguistics.

Ben Wang and Aran Komatsuzaki. 2021. Gpt-j-6b: A 6 billion parameter autoregressive language model.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Appendix A Code details

We release our code anonymously at `https://github.com/Anonymous-ARR/code` under MIT License.

The models weights, data and other dependencies required for experiment are at `https://github.com/Anonymous-ARR/Releases/releases`.

The intended use of our code is for academic research. We consider probing publicly available PLMs, which are made available for research as well as end use cases, to be within the intended use of PLMs.

Appendix B Probing for Character Information

We use off-the-shelf APIs for lemmatization and WordNet from NLTK (Bird et al., 2009) (Apache License 2.0). Our implementation uses PyTorch (Paszke et al., 2019) (BSD License), HuggingFace (Wolf et al., 2019) (Apache License 2.0) and custom APIs for GPT-J's embedding.

The probes for each MLP are trained separately starting with random initialization weights. We train the probe via a binary classification task via backpropagation, using the Adam optimizer (Kingma and Ba, 2014) with betas of 0.9 & 0.999 and epsilon of 1e-08 without weight decay, over the standard Binary Cross Entropy loss across the predicted logits $\hat{y}_i$ and ground truth logits y_i.

B.1 PLMs considered

Details of the PLMs used along with their model-card on Huggingface:

Model type	Case-Sensitive	
	PLM	Control
GPT-J	94.35	52.76
GPT-2	84.69	51.05
RoBERTa	83.87	49.00
BERT-Cased	78.47	45.35
BERT-Uncased	77.48	49.37
GloVe 300D	69.40	49.40
GloVe 100D	61.56	49.55
LXMERT	60.30	49.61

Table 5: Results for the main probing experiment, across models.

- **GPT-J:** We used the standard GPT-J with 6 Billion parameters and its reversible Byte-Pair encoding based subword tokenizer. We extracted the embeddings and have released it separately. Model Card: 'EleutherAI/gpt-j-6B' under Apache 2.0 License.

- **GPT-2:** We consider the base model for GPT-2 with 124 Million parameters. The tokenizer used in this model is the exact same as the one used in GPT-3 and is also a subword tokenizer based on reversible Byte-Pair encoding. Model Card: 'gpt2' under Modified MIT License.

- **RoBERTa:** We again use the Base model for fairer comparison to GPT-2 model with 125 Million parameters. This model has partially reversible Byte-Pair Encoding based on GPT-2's byte-pair tokenizer but with additional tokens for a BERT-like MLM discriminative pre-training. Model Card: 'roberta-base' under MIT License

- **BERT:** The BERT-base models have roughly 110 Million parameters. Both the Uncased and Cased versions of this model are considered with their Word-Piece tokenizers. For this tokenizer, we also consider the character '##' while filtering out vocabulary, as it denotes the token continues on the preceeding word. Model Card: 'bert-base-uncased', 'bert-base-cased' under Apache 2.0 License

- **GloVe:** We experiment with 100 and 300 dim version of 400K-Vocab GloVe trained on 6B tokens. We consider the 40k most frequent tokens in GloVe, comparable to the vocabulary sizes of the other models. GloVe version used: 'Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d,

Model	Case-insensitive		Case-Insensitive	
	PLM	Control	PLM	Control
GPT-J	0.83	3.12	1.39	2.27
GPT-2	2.01	3.09	2.21	2.75
RoBERTa	2.27	3.13	2.79	2.46
BERT-Cased	2.93	7.46	2.77	5.67
BERT-Uncased	3.32	4.33	3.32	4.33

Table 6: Standard Deviation in our probing experiment 1 for the key models considered.

Property	Statistics
Dataset	Tokenizer's Vocab for each model
Data-filtered	Tokens having only letters (a-z,A-Z) *GPTs, RoBERTa*: Allow preceding Ġ *BERT*: Allow preceding '##'
Train-Test split	80-20
Preprocessing	None
Output labels	26 tasks (each with binary label)
Link	Model Card & links in §B.1

Table 7: Dataset Checklist for experiment 1.

200d, & 300d vectors, 822 MB download): glove.6B.zip' [2]

- **LXMERT:** We use the uncased version of LXMERT-base model and similar to the BERT model, filtering out also for '##' preceding symbols. Model Card: 'unc-nlp/lxmert-base-uncased' under

B.2 Hyperparameter and other Details

Each probe is trained for 5 epochs, with 128 batch-size. The Learning rate is tuned over averaged Macro-F1 in the grid $\{1e-5, 3e-5, 5e-5, 1e-4, 3e-4, 1e-3, 3e-3, 1e-2, 3e-2\}$. We trained the probe on the best hyperparameter settings across 5 different train-test splits and seeds. Table 8 shows these best learning rate and the number of parameter (and frozen-parameters) in the probe. For all the control embedding, we assume the same dimension as the largest model (4096) and considered a maximum vocab of 100k, even though only the first few thousands might be used. These experiments take less than 20 minutes for each run requiring less than 12 GB of GPU memory and were run on a mix of NVidia Tesla K80, GTX 1080 Ti, P100, V100 GPUs with Dell R740 and Intel Xeon CPUs.

Table 5 shows the result of the probe in a case-sensitive setting. The case-insensitive probe treats both "Cat" and "cat" as a hit for both "c". The case-sensitive probe treats only "cat" (not "Cat") as a hit for "c". Note that performance is the same for BERT-Uncased since it does not distinguish between these conditions.

[2]Accessible at nlp.stanford.edu/projects/glove/, Apache v2.0 License

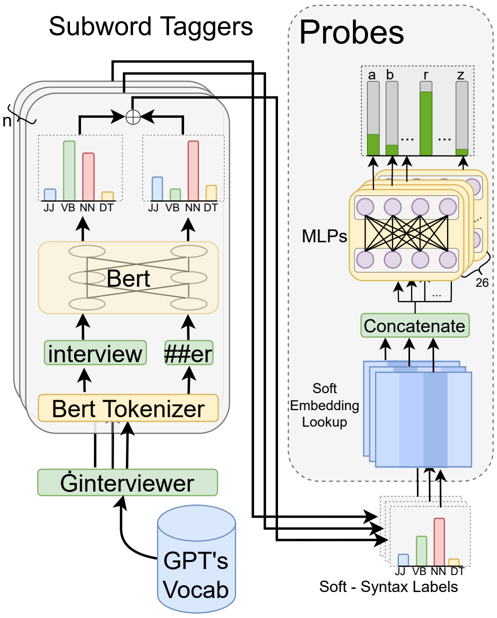

Figure 4: Experiment 2: syntax baselines with BERT-sentence and BERT-token custom taggers.

Appendix C Syntax Baseline for Character information

C.1 Custom syntax taggers

First we consider an off-the-shelf SpaCy model with its 3 features for each token: NER, PoS, and Coarse-Grained PoS tags. Before running this model, we remove the preceding whitespace characters in the token, if present. The resultant features are discrete one-hot feature vector over labels. The SpaCy tagger is not perfectly suited to our task since it operates at the word level, whereas we are concerned with obtaining a subword token's embeddings. To solve that problem, we also built 3 custom taggers for obtaining PoS and NER labels on subword tokens. These tagger takes (a subword) token's model embedding as input and outputs a vector of probabilities over part of speech and named entity categories.

To build our custom GPT-J-Tagger, we train an MLP to predict PoS and NER label based on GPT-J's static embedding layer for each token. The tagger is trained on the CoNLL 2003 dataset's train and valid splits (Sang and De Meulder, 2003), which contains part of speech and named entity information. Unlike the SpaCy tagger, our custom GPT-J-Tagger outputs a probability distribu-

Model Probe	Case-insensitive				Case-Sensitive			
	Lemma		Control		Lemma		Control	
	LR	# Params	LR	# Params	LR	# Params	LR	# Params
GPT-J	1e-4	240M (206M)	1e-4	443M (410M)	1e-4	240M (206M)	3e-4	443M (410M)
GPT2	3e-4	40M (39M)	1e-4	443M (410M)	3e-4	40M (39M)	3e-4	443M (410M)
RoBERTa	3e-4	40M (39M)	1e-4	443M (410M)	1e-3	40M (39M)	1e-2	443M (410M)
BERT-cased	1e-3	23M (22M)	3e-3	443M (410M)	1e-3	23M (22M)	5e-5	443M (410M)
BERT-uncased	3e-3	25M (23M)	3e-4	443M (410M)	3e-4	25M (23M)	1e-4	443M (410M)
LXMERT	1e-4	24M (23M)	3e-4	443M (410M)	3e-4	24M (23M)	1e-4	443M (410M)
GloVe 100D	1e-4	4.02M (4.00M)	3e-4	12.2M (12.0M)	3e-4	4.02M (4.00M)	3e-4	12.2M (12.0M)
GloVe 300D	3e-4	12.2M (12.0M)	1e-4	12.2M (12.0M)	3e-4	12.2M (12.0M)	3e-5	12.2M (12.0M)

Table 8: Experiment 1 Hyperparameters.

Property	Statistics
Train Sentences	14986
Train Tokens	219553
Valid Sentences	3465
Valid Tokens	55043
Test Sentences	3683
Test Tokens	50349
NER Tags	5
PoS Tags	45
Preprocessing	None
Link	github: davidsbatista/NER-datasets

Table 9: Dataset Checklist for training POS/NER CoNLL set.

tion over categories so we can use this distribution over labels as the vector of interest, rather than a one-hot vector.

Table 10 show the performance of the tagger's performance *qua* tagger. Table 9 shows the Dataset Checklist for this experiment. To build the BERT sequence-labeling tagger, we fine-tuned a BERT sequence labeling model for the PoS and NER tasks, in order to output a label for each (sub-word) token in a sentence. When extracting syntactic features for this model, we first do the same pre-processing of removing the special preceeding whitespace of GPT's tokens as SpaCy before input into the BERT model. Since BERT's tokenizer could have more than one token for a single GPT-J's token, we consider average of the logits as the pre-softmaxed feature vector.

In addition to the BERT sentence-level tagger, we consider a BERT token classifier model fine-tuned for NER and PoS at token level rather than at sequence (sentence level) in the preceding one with same pre-processing. This token-level model does not leverage context to deduce the label, and is closer to how we use this model later to get features for predicting NER/PoS features.

C.2 Results and Hyperparameters

We use off-the-shelf APIs for lemmatization and WordNet from NLTK. Our implementation uses PyTorch (Paszke et al., 2019), HuggingFace (Wolf et al., 2019) and custom APIs (now released) for GPT-J's embedding. The hyperparameter tuning was done on the dev set for only the learning rate in the grid $\{1e-5, 3e-5, 1e-4\}$ for BERT and $\{1e-5, 3e-5, 5e-5, 1e-4, 3e-4, 1e-3, 3e-3, 1e-2, 3e-2\}$ for GPT-J. Our MLP model is 3-layered with SELU and Tanh activation and 0.1 Dropout before the last layer. Our BERT-Model is initialized with 'bert-base-cased' from Huggingface with default values of hyperparameters. Our implementation was done using PyTorch and optimized via Adam with betas of 0.9 & 0.999 and epsilon of 1e-08 without weight decay over the standard Cross Entropy loss. We set the batch size to 32 sentences for BERT and 64 for GPT-J. All the experiments can be done within 16GB of GPU memory and no run individually takes more than 2 hours. We release these models along with our codebase with instructions to run them.

Table 10 shows the performance of these NER and PoS models. As expected, the BERT-sentence model performs the best on both the tasks as it leverages the context while tagging. Whereas, GPT-J slighlty outperfoms BERT-token on both the tasks. Note that these performance are not comparable as their tokenization differ and only one of the model leverages context to predict NER and PoS tag.

C.3 Method

Assume we have m syntactic features. Consider the tokenizer Vocabulary V (with only alphabetic tokens) and the D_α datapoint pairs for each letter α of the lowercased English alphabets. For each token-label pair (w_i, y_i), we obtain the m syntactic features of the word $\{x_i^{(1)}, x_i^{(2)} \dots x_i^{(m)}\}$ using the trained models to tag the features.

Model Type	# Epochs	Batch Size	LR	Dev $F1_{Wtd}$	Dev $F1_{Macro}$	Test $F1_{Wtd}$	Test $F1_{Macro}$
BERT-sentence (PoS)	10	32	1e-5	98.17	94.80	93.42	87.40
BERT-token (PoS)	10	32	1e-5	76.42	56.75	77.24	56.74
GPT-J MLP (PoS)	20	64	1e-4	62.90	68.72	60.15	69.14
BERT-sentence (NER)	10	32	1e-5	97.88	93.18	96.02	86.92
BERT-token (NER)	10	32	1e-5	83.50	56.97	81.57	54.88
GPT-J MLP (NER)	20	64	5e-5	85.59	63.56	82.71	57.34

Table 10: Labels from POS/NER labels. LR denotes learning rate

Split Type	SpaCy	BERT-sentence	BERT-token	GPT-J	Control
Aggregate across 26 characters					
F1	52.338	55.008	59.7525	**61.2395**	49.6772
Best performing ones					
s	64.5967	60.7179	**70.3299**	66.8159	40.3154
y	61.9632	60.3871	**67.1591**	64.8863	48.6838
e	62.0518	57.7531	**64.6152**	62.3213	47.2712
t	60.6848	54.3826	**64.0681**	60.7345	48.4873
p	50.235	55.2361	**63.9658**	60.5067	46.5612
i	60.8024	56.4055	**63.3518**	61.6032	42.8155
Worst performing ones					
w	45.748	52.7235	57.6919	**58.2666**	48.6947
q	43.7924	56.5274	**57.5407**	53.5437	49.2841
k	47.7873	49.3832	**57.3084**	55.9559	46.2371
o	52.9403	53.6138	**56.8312**	55.6293	43.5871
b	48.9159	**56.739**	56.3873	55.1265	48.252
m	48.1349	53.4036	**56.2846**	55.6094	46.1084

Table 11: Syntax baseline: Probing over syntax label distribution.

Split Type	SpaCy	BERT-sentence	BERT-token	GPT-J	Control
Aggregate across 26 letters					
F1	4.4354	2.9588	3.7989	2.724	4.3973
Best performing ones					
s	0.6947	1.2941	0.4853	0.6514	5.5055
y	1.8665	1.6406	0.5697	1.4251	3.2417
e	0.6645	0.8544	0.3245	0.3233	1.8349
t	0.2643	3.4695	0.9129	0.5924	1.7645
p	6.1928	1.1628	0.5669	0.2985	3.7013
i	0.512	1.4392	0.5998	0.4867	5.5685
Worst performing ones					
w	4.9794	2.2996	1.9614	1.9536	1.7453
q	2.7071	3.4438	4.5954	4.7932	5.5068
k	2.9332	6.885	2.0885	1.6864	1.6311
o	6.24	1.6009	1.0449	0.463	3.5961
b	4.0455	1.5597	1.4074	2.0701	2.7857
m	7.2995	2.4854	2.1762	1.0948	6.152

Table 12: Standard Deviation of POS/NER labels

We train a classifier to predict whether a character α is present in the token w_i using only its syntactic features. Assume randomly initialized 'trainable' embeddings $\{E_1, E_2 \ldots E_m\}$ for each of the m syntactic features. We predict the logits $\hat{y}_i$ for token w_i over each letter α using an MLP classifier over the embeddings:

$$\hat{y}_i = \sigma(MLP_\alpha([E_1^T x_i^{(1)} \,;\, \ldots \,;\, E_m^T x_i^{(m)}]))$$

where T, σ, ; denotes transpose, sigmoid function and vector concatenation respectively. Each syntactic feature $x_i^{(j)}$ is a vector denoting probability distribution of a token over the corresponding feature labels (including being a one-hot vector), this is crucial because a token (especially subword-token) might have different labels depending on the context.

We train different MLPs and Embeddings from scratch for each alphabet α with no shared parameters across the (case-insensitive) 26 English alphabets. We train our model for binary classification via backpropagation over the standard Binary Cross Entropy loss across the predicted logits $\hat{y}_i$ and ground truth logits y_i.

As before, for each character we create a balanced dataset consisting of equal number of positive and negative examples, where each example is made up entirely of either English alphabets or whitespace. These are randomly divided into training and test split which keep words with same tokens with same lemmas in the same split. For control task we randomly assign the syntactic features for each token. We set the batch size for runs with one-hot vectors as features to 128 and to 64 for others, the learning rate is tuned in $\{1e{-}5, 3e{-}5, 1e{-}4, 3e{-}4, 1e{-}3, 3e{-}3, 1e{-}2\}$ for all the features over the metric of Averaged F1-Scores across the 26 English letters. The best learning rates for SpaCy, BERT-sentence, BERT-token, GPT-J and Control were found to be 1e-3, 1e-3, 3e-3, 1e-4, 1e-2. Using Adam Optimizer we train each of the 26 models for 5 epochs with betas of 0.9 & 0.999 and epsilon of 1e-8. Our implementation is done using PyTorch and Huggingface. Finally for the best hyperparameter, we perform 5 runs with different train/test splits and seeds. Our MLP model is 3-layered with SELU and Tanh activation and 0.1 Dropout before the last layer.

Tables 11 and 12 show the mean and variance of the results over the 4 taggers and control task. We also show the performance over the best and worst performing letters.

Appendix D Variability of Tokenization

D.1 Quantifying variability in the Pile Corpus

To quantify the variability in the tokenization of frequent words in the kind of corpora used to train these models, we consider 1/6th of the publicly available Pile Corpus used to train GPT-J (250 GB of text). For our analysis we consider 500 frequent words of 8+ characters (as measured using Google Ngrams) since long words are more likely to be the source of variability.

For each target word, we first case-insensitively detect each of its occurrence in the sub-corpus. In order to also account for spelling errors, we used case-insensitive fuzzy search allowing matches of substring up to 1 Levenshtein distance away. Over these occurrences, we discard those where the substring is part of a bigger word, such as some 'differentiation' for the target word 'different' or if the fuzzy match has whitespaces.

Once we have such occurrences, we want to obtain the tokenization of the target word in the context. For this reason, we obtain the adjust the indices of the possibly-misspelt matched-substring for our target word till the nearest non-word, this allows for matches of [somethin', somethin", somethin] all to be considered as the string 'somethin'. We also account the factors that leads to differing tokenization, such as preceeding whitespaces.

Now, for each of the target words, we have a list of probable tokenization at most 1 Levensthein distance away. Since two target words such as 'projection' and 'protection' could themselves be at 1 Levenshtein distance, these may act as pseudo matches for each other. So we consider only one of these two from our target list, leading to 466 word down from 500 words. Now, for each of these target words, we count the number of possible unique tokenizations.

For each of these 466 target words, we also obtain a list of words from the WordNet which are 1 Levenshtein distance away. We call this word list as the pseudo-match list. We also consider the number of tokenization for each target words, excluding their pseudo-match list as well as by excluding all those which are equally (or closer) to any word in pseudo-match list than the target word. We also compute the statistics of those with exact matches.

Table 13 shows these statistics for the target words. On average, a target word is expected to have 213 different tokenization depending on the context. We observe that while one may expect the number of tokenization to go up with the number of characters in the target word, it doesn't perfectly increase monotonically. This is because the number of occurrences of the target word, dictates the number of tokenization it will have, and there we see a consistent trend that the number of tokenization greatly increases with increasing occurrences. It is due to this reason, we expect these number to be even higher, when considering the entire Pile corpus, instead of the subset in our case.

We observe three factors contributing to surprisingly large number of tokenization. First, it is because of Case-Sensitive tokenization, which leads to upto 6 different tokenization for each of the target words. Second, its the context dependent tokenization, which increases the expected number of different tokenization to 12.91. Lastly, the remainder is contributed by the previous two combined with misspellings.

Our implementation is sped up using multiprocessing and fuzzy regex. For this we split the subcorpus across multiple pieces. These runs takes about 3 days across 40 CPU Cores, 60 GB of RAM and less than 600GB hard disk space. Our experiment was only conducted on a portion of the Pile corpus, and the possible tokenizations for each target word is expected to increase with larger corpus size. We report the mean and standard deviation in the number of tokenizations a word has across the portion of the Pile corpus considered. These are also reported as a function of word length and its frequency of occurrence in the corpus.

Tables 13 and 14 shows these score. The 'All matches' field considers the unique tokenizations of all matched substrings including those at 1 (case and whitespace insensitive) Levensthein distance away. These word at 1 Levensthein distance could be either misspellings or a different English word (for example an occurrence of the word 'projection' for target word 'protection'). The latter of these are identified using the wordnet dictionary and the statistics recalculated and shown in the column 'Matches except pseudo'. Some of the misspelling contribute to this score could be misspelling of either the target word or of one of the other English words at 1 Levensthein distance away ('prohection' could be a misspelling of either 'projection' or 'protection' being at distance 1 from both). Such occurrences are removed and statistics recomputed for the column 'Matches closer pseudo'. The column 'Exact contain' considers only those occurrences, which contain the exact target word (case-insensitively) in the string ignoring whitespaces. Whereas the 'Exact match' does not consider the occurrences involving a preceeding whitespace.

Table 15 shows some examples for the variation in tokenization.

D.2 Algorithm for increasing tokenization variability

Algorithm 1 A simplified version of subword Tokenization with controllable variability

```
Require: 0 <= ρ <= 1
  procedure YOURFUNCTION(sentence)
      tokens ← List()
      words ← wordTokenize(sentences)
      for each w in words do
          u ~ Uniform[0, 1]
          if u < ρ then
              V ← GPTJ.Vocab
              filter(V, λx.isAlphabetic(x))
              Choices ← List()
              for i in 1, 2 ... (w.length() − 1) do
                  if w[:i] ∈ V & w[i:] ∈ V then
                      push(Choices, w[:i], w[i:])
                  end if
              end for
              if ¬isEmpty(Choices) then
                  s ~ Choices
                  tokens ← Merge(tokens, s)
                  continue
              end if
          end if
          s ← GPTJ.Tokenize(w)
          tokens ← Merge(tokens, s)
      end for
  end procedure
```

Measure	All Matches	Matches except pseudo	Matches closer pseudo	Exact contain	Exact match	Num Words
Aggregate	232.90	229.70	213.74	17.91	5.97	466
7 Length words	297.50	271.00	223.50	22.00	6.5	2
8 Length words	332.29	325.68	288.07	25.00	7.89	28
9 Length words	231.48	227.78	206.95	16.94	5.93	190
10 Length words	225.51	222.58	209.53	17.97	5.87	127
11 Length words	213.28	211.02	202.97	17.88	5.85	61
12 Length words	224.14	223.54	218.64	18.25	5.79	28
13 Length words	218.14	217.00	214.76	16.57	5.19	21
14 Length words	238.33	238.33	238.33	16.67	5.00	9
exp(12) occurrence	88.70	86.67	82.11	10.33	5.90	27
exp(13) occurrence	155.78	153.87	146.55	13.61	5.15	74
exp(14) occurrence	210.36	207.51	195.74	16.70	5.75	174
exp(15) occurrence	278.88	275.00	251.69	19.91	5.96	139
exp(16) occurrence	370.02	365.04	336.48	26.62	8.56	52

Table 13: Tokenization Variance statistics - mean score.

Measure	All Matches	Matches except pseudo	Matches closer pseudo	Exact contain	Exact match
Aggregate	95.12	94.29	91.26	17.91	2.67
7 Length words	155.50	129.00	81.50	13.00	2.50
8 Length words	100.90	99.17	91.19	8.46	2.47
9 Length words	90.97	90.00	86.03	7.34	2.50
10 Length words	88.56	89.04	90.71	7.86	2.75
11 Length words	107.55	107.65	108.46	8.77	2.84
12 Length words	63.25	63.53	62.53	8.26	2.82
13 Length words	81.22	81.30	82.20	7.82	2.59
14 Length words	62.48	62.48	62.48	4.52	1.05
exp(12) occurrence	38.59	37.65	34.60	3.15	1.26
exp(13) occurrence	39.75	39.13	39.36	4.92	2.10
exp(14) occurrence	51.84	52.17	53.73	6.19	2.51
exp(15) occurrence	70.46	70.59	77.22	7.86	2.38
exp(16) occurrence	101.86	100.38	103.83	9.99	3.44

Table 14: Variability across target words in Tokenization Variance statistics.

String	Tokenization	String	Tokenization	String	Tokenization
signature		playstation		personal	
Exact match case insensitive		Exact match case insensitive		Exact match case insensitive	
"SIGNATURE"	["SIGN", "ATURE"]	"playstation"	["play", "station"]	"pERSONAL"	["p", "ERSON", "AL"]
"sIGNATURE"	["s", "IGN", "ATURE"]	"PLaySTATION"	["PL", "ay", "ST", "ATION"]	"PeRSonAl"	["Pe", "RS", "on", "Al"]
"SigNature"	["S", "ig", "Nature"]	"playStation"	["play", "Station"]	"personal"	["personal"]
"Signature"	["Sign", "ature"]	"PLAYSTATION"	["PLAY", "ST", "ATION"]	"Personal"	["Personal"]
"SIgnature"	["SI", "gn", "ature"]	"Playstation"	["Play", "station"]	"PERSONAL"	["P", "ERSON", "AL"]
"signature"	["sign", "ature"]	"PlayStation"	["Play", "Station"]	"PErsonal"	["P", "Er", "son", "al"]
Exact match and whitespaces		Exact match and whitespaces		Exact match and whitespaces	
" signature"	["Ġsignature"]	" Playstation"	["ĠPlaystation"]	" PERSONal"	["ĠPERSON", "al"]
" Signature"	["ĠSignature"]	" PLayStation"	["ĠPL", "ay", "Station"]	" PerSoNAl"	["ĠPer", "So", "N", "Al"]
" SigNature"	["ĠSig", "Nature"]	" PLAYstation"	["ĠPLAY", "station"]	" pErsonal"	["Ġp", "Er", "son", "al"]
" signaTure"	["Ġsign", "a", "T", "ure"]	" PLAYSTATION"	["ĠPLAY", "ST", "ATION"]	" perSonal"	["Ġper", "S", "onal"]
" SIGNATure"	["ĠSIGN", "AT", "ure"]	" PlayStation"	["ĠPlayStation"]	" perSONal"	["Ġper", "SON", "al"]
" SiGNATURE"	["ĠSi", "GN", "ATURE"]	" plAYsTaTion"	["Ġpl", "AY", "s", "Ta", "T", "ion"]	" pERSonal"	["Ġp", "ERS", "onal"]
" SIGNATURE"	["ĠSIGN", "ATURE"]	" playStation"	["Ġplay", "Station"]	" PERSONAL"	["ĠPERSON", "AL"]
" signAture"	["Ġsign", "At", "ure"]	" playstation"	["Ġplay", "station"]	" PERSONAl"	["ĠPERSON", "Al"]
" SIGNature"	["ĠSIGN", "ature"]	" PLaystation"	["ĠPL", "ay", "station"]	" personal"	["Ġpersonal"]
" sIgnature"	["Ġs", "Ign", "ature"]	" PlaySTation"	["ĠPlay", "ST", "ation"]	" Personal"	["ĠPersonal"]
Fuzzy match and misspellings		Fuzzy match and misspellings		Fuzzy match and misspellings	
"S1GNATURE"	["S", "1", "GN", "ATURE"]	"Play-station"	["Play", "-", "station"]	"p-ersonal"	["p", "-", "erson", "al"]
" SIGNATUTRE"	["ĠSIGN", "AT", "UT", "RE"]	" PLAY-STATION"	["ĠPLAY", "-", "ST", "ATION"]	"per-sonal"	["per", "-", "son", "al"]
" signatyure"	["Ġsign", "at", "y", "ure"]	"play-station"	["play", "-", "station"]	pers-onal"	["Ġpers", "-", "onal"]
" signatre"	["Ġsign", "atre"]	" Play-station"	["ĠPlay", "-", "station"]	Per-sonal"	["ĠPer", "-", "son", "al"]
"Signiature"	["Sign", "i", "ature"]	" play-station"	["Ġplay", "-", "station"]	"PER$oNAL"	["PER", "$", "o", "N", "AL"]
" signnature"	["Ġsign", "nature"]	"Play-Station"	["Play", "-", "Station"]	" PER.SONAL"	["ĠPER", ".", "SON", "AL"]
" signatrre"	["Ġsign", "at", "r", "re"]	"Play]station"	["Play", "]", "station"]	" PERSONA.L"	["ĠPERSON", "A", ".", "L"]
" sigature"	["Ġsig", "ature"]	" Playst4tion"	["ĠPlay", "st", "4", "tion"]	"persona[l"	["person", "a", "[", "l"]
" Sign(ature"	["ĠSign", "(", "ature"]	" PlayStati0n"	["ĠPlay", "St", "ati", "0", "n"]	"person,L"	["person", ",", "L"]
"signnature"	["sign", "nature"]	" Play-Station"	["ĠPlay", "-", "Station"]	"Person(al"	["Person", "(", "al"]
"SIG(NATURE"	["S", "IG", "(", "NAT", "URE"]	"Playstaton"	["Play", "st", "aton"]	" p[ersonal"	["Ġp", "[", "erson", "al"]
" Si2nature"	["ĠSi", "2", "nature"]	" play.Station"	["Ġplay", ".", "Station"]	"p]ersonal"	["p", "]", "erson", "al"]
"Singnature"	["Sing", "nature"]	" playstaton"	["Ġplay", "st", "aton"]	" p)ersonal"	["Ġp", ")", "erson", "al"]
" signatuure"	["Ġsign", "atu", "ure"]	" PLAYTSTATION"	["ĠPLAY", "T", "ST", "ATION"]	"P_ersonal"	["P", "_", "erson", "al"]
" Signaturs"	["ĠSign", "at", "urs"]	"playstatiom"	["play", "st", "ati", "om"]	"Persnal"	["Pers", "n", "al"]
" sigNUTure"	["Ġsig", "N", "UT", "ure"]	"playsstation"	["plays", "station"]	" peRSSonal"	["Ġpe", "R", "SS", "onal"]

Table 15: Some examples of variations in Tokenization for 3 frequent long words.