Baseline Needs Even More Love: A Simple Word-Embedding-Based Model for Genetic Engineering Attribution

Abstract

(Shen et al., 2018) first demonstrated that Simple Word-Embedding-Based Models (SWEMs) outperform convolution neural networks (CNNs) and recurrent neural networks (RNNs) in many natural language processing (NLP) tasks. We apply SWEMs to the task of genetic engineering attribution. We encode genetic sequences using BPE as proposed by (Alley et al., 2020), which separates the sequence into motifs (distinct sequences of DNA). Our model uses a max-pooling SWEM to extract a feature vector from the organism's motifs, and a simple neural network to extract a feature vector from the organism's phenotypes (observed characteristics). These two feature vectors are concatenated and then used to predict the lab of origin. Our model achieves 90.24% top-10 accuracy on the private test set, outperforming RNNs (Alley et al., 2020) and CNNs (Nielsen & Voigt, 2018). The simplicity of our model makes it interpretable, and we discuss how domain experts may approach interpreting the model.

1. Encoding Genetic Sequences

We encode each DNA sequence using byte-pair encoding (BPE) (Sennrich et al., 2015) as proposed by (Alley et al., 2020). BPE is an unsupervised algorithm used to build a vocabulary from a training corpus. The algorithm is run before training the classifier. The algorithm requires the desired size of the vocabulary to be specified. We specify it as 65,500 to make the vocabulary as large as possible, whilst also allowing us to store the BPE encoded sequences using 2-byte unsigned integers, meaning the encoded training set fits into memory (less than 100MB in size), allowing us to encode the training set prior to training which speeds up training significantly. A corpus is a list of texts, and our corpus for this task is the set of genetic sequences that are in our training set. For our task the algorithm starts with a vocabulary of the letters G, A, T, C, and N. The final vocabulary contains a mixture of letters and words (where a word is a sequence of letters), henceforth when we refer to words in our vocabulary, we mean a sequence of one or more letters. The algorithm counts the frequency of each bi-gram of words (pair of words occurring next to each other in a genetic sequence) from our vocabulary constructed so far. The most frequent bi-gram is then merged into a single word and added to the end of vocabulary. This process of counting, merging, and adding to our vocabulary is repeated until we have a vocabulary of size 65,500.

Once we have a vocabulary, we can encode genetic sequences by working backwards through our vocabulary replacing every occurrence of the current word with a number corresponding to its position in the vocabulary. Say we build a vocabulary of 9 words from a corpus using BPE, and end up with the vocabulary G, A, T, C, N, GA, GAT, GAC, GATT. Given this vocabulary, the encoding of the sequence "GCGATAGATT" would be 0, 3, 6, 1, 8, and if we mapped these numbers back to their corresponding words in the vocabulary, the list would be G, C, GAT, A, GATT.

This method of encoding has some very nice properties.

There is a wide variation in the length of words in our constructed vocabulary, ranging from 1 to 16 letters in length. Our vocabulary captures a wide range of lengths of words whilst being relatively small. If we were to build a vocabulary of all possible words of lengths 1 to 16, with our alphabet of 5 letters, we would create a vocabulary of 190 million words, whereas our BPE encoded vocabulary is only 65,500 words. It achieves this feat because BPE only includes the words that occur most frequently. Assuming the corpus of DNA sequences in the training set is representative of unseen DNA sequences, then the most frequent words occurring in it are the most likely words we will see in unseen data. Only including the most frequent words also allows the model to generalize better, as the more frequent a word in the training set, the better we can learn its meaning.

We are guaranteed that we can encode every unseen genetic sequence. This is because at worst our encoder will encode an unseen sequence in terms of the letters G, A, T, C, and N, which result in an encoded sequence as long as the original sequence. But assuming the training set is representative it is likely that longer words from our vocabulary will be present in the unseen sequence, reducing the length of the encoded sequence.

The algorithm being unsupervised means that the encoding technique should age well. Hand engineered features extracted from genetic sequences may become obsolete over time as the number of samples and labs in the dataset increases. Whereas with this unsupervised technique the vocabulary can just be rebuilt from scratch by running BPE on the training set whenever new samples are added.

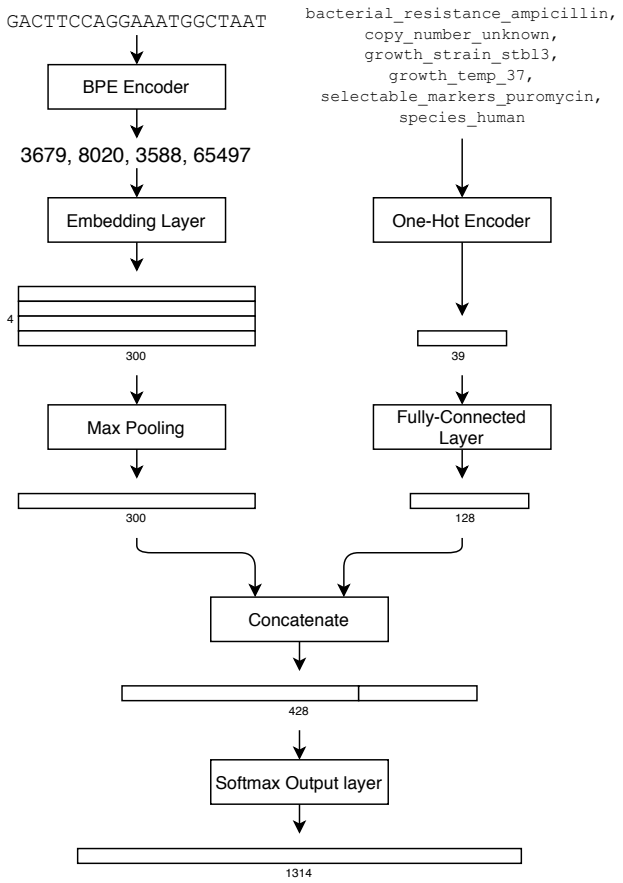

Figure 1. Model pipeline for sample *UZNFO*.

2. Model

In figure 1 we show the architecture of our model. In the figure we also show the output at each step for sample *UZNFO* from the dataset, to help better understand what the model is doing. We chose this sample for the example as it has a short genetic sequence and BPE encoding, which allows us to fit the full output at each step in the diagram. The first step in the model is the BPE encoder, as we described in section 1. The output of the BPE encoder is 3679, 8020, 3588, 65497, which corresponds to GAC, TTCCAGGAAA, TGGCTAA, T in the vocabulary.

The words in the encoded sequence are then converted into vectors using an embedding layer. The embedding layer assigns each word in the vocabulary its own vector. The output of the layer is produced by mapping each word number in the input to its corresponding vector. All vectors are the same size, and the size of the vector v is a hyperparameter. In our model we found empirically $v = 300$ was a good size. These vectors are initialized uniformly at the start of training the model, and the values of the vectors are learnt during training. Henceforth we refer to these vectors as word embeddings.

The embeddings layer produce a matrix of size $t \times 300$, where t is the length of the encoded sequence (in the case of sample *UZNFO*, $t = 4$). The size of the matrix varies based on the sequence length, so we use max pooling to reduce the matrix to a single vector of v components (so 300 components). Max pooling reduces the matrix to a single vector by taking the maximum value for each component observed across the word embeddings of the encoded sequence. The effect of this operation is it extracts the most salient features (Shen et al., 2018) of the sequence. As the word embeddings are learnt in the context of the task, these salient features will be the ones that are most significant in identifying the likelihood of the labs given the sequence. The vector output by max pooling can be thought of as a sequence feature vector that summarises the most important information from the genetic sequence in the context of the task.

The six phenotypes are one-hot encoded (the data is provided in this format but we include this step in the diagram for clarity), which produces a vector of 39 components. We feed these into a fully-connected layer with 128 units and a ReLU activation function. The 128 component vector output by the layer is concatenated with our sequence feature vector to produce a vector of size 428. This vector is passed to the output layer, which is a fully-connected layer with 1314 units (one for each lab). We apply a softmax activation function to the outputs of the units, which converts the outputs into probabilities that sum to 1. The output vector is the likelihood of each lab given the input genetic sequence and phenotypes.

This model has a major advantage over the LSTM RNN proposed by (Alley et al., 2020) in that it has no recurrent connections. The problem with recurrent connections is that they are susceptible to dying gradients. LSTMs suffer this problem much less than plain RNNs, but empirical research has found that they are only sensitive to hundreds of previous words. This is a problem when applying them to genetic sequences, as sequences are extremely long. Our BPE encoded sequences in the test set are on average 574 words long, and at maximum 6162 words long.

In contrast SWEMs replace recurrent connections with a single max pooling operation, meaning they do not suffer the same limitation. This means they are able to learn the relationship between words any distance apart. This might explain why SWEMs outperform RNNs, as if there are extremely long range dependencies SWEMs have a significant advantage. This advantage of SWEMs over RNNs and CNNs was observed by (Shen et al., 2018) for natural language, with them finding that SWEMs outperform RNNs and CNNs when classifying long (hundreds of words) documents.

One limitation of SWEMs is they have no concept of where words occurred in the sequence, with the max pooling operation removing this information. They treat sequences more like a bag of words than a time series, which means they can not learn the relationship between words in the context of their proximity. This is somewhat alleviated by using a large vocabulary like we do, as it means that longer words are included in the vocabulary. Long words captures the relationship between letters and their neighbours within

the word.

The max pooling operation also means we lose information about the frequency of the words in the sequence. Repeated words will be represented as the same word embedding vector. The max pooling operation only takes the largest value for each component, and as such is not influenced by the number of times that maximum value (and consequently the corresponding word) occurred.

3. Interpretability

As there are only 300 components, this also means that at most 300 words can influence the max pooled vector. In this regard max pooling has a side effect of shortlisting the most significant words in the sequence. To find the shortlist we can just take the argmax of the word embeddings instead of applying max pooling. This then tells us which positions in the BPE encoded sequence influenced the components of the vector, and then the corresponding word numbers can be mapped back to their corresponding words in the vocabulary to create the shortlist. This can be helpful for domain experts to understand which words the model perceives as most important in each sample. This does have the limitation that 300 is still a lot of words for a domain expert to consider. This could be solved by reducing the size of the embedding, as this would make the shortlist smaller. But there is a trade-off that reducing the size of the embedding layer decreases performance. Another limitation is the model gives no insight into the importance of each shortlisted word for the sample. Given how simplistic the model is, we feel that it may be possible to infer perceived word importance, but this would require further research.

(Shen et al., 2018) observed when applying max-pooled SWEMs to NLP that the embeddings learnt were very sparse with most weights concentrated around 0. This suggested that the model may have only been relying on a few key words in the vocabulary to predict the classes, as when these few words with components with huge magnitudes were present they dominated the output of the max-pooling, and thus the output of the network. They looked at the top-5 largest values in the vocabulary for each component, and found that the top-5 words for each component were semantically similar. This allowed them to infer the meaning of the components from what the words had in common.

We observe that the embeddings learnt for our model are also very sparse, with most weights concentrated around 0. Hence we hypothesise that our model may also only be relying on a few key words, and perhaps by looking at the corresponding words to the top-5 values for each component, a domain expert may be able to deduce what the words have in common, and use this to infer the meaning of the component. We are not domain experts so we have no way to confirm this. But this may be an avenue of future research to improve the model's interpretability.

Word embeddings can be visualized using projector.tensorflow.org, which reduces the 300 dimension word embeddings to two or three dimensions using PCA. This tool may be helpful for domain experts to understand which words the model perceives to be related. As the model will learn similar embeddings for words with similar meaning in the context of the task. Words can be searched for by name, and their closest neighbours in the original space are also listed. For example in figure 2 we show the projection for the word "GAC" and the twenty-five nearest words to it. The distance of their embeddings from GAC in the original space is described in the panel on the right of the figure.

4. Deployment

It takes under 15 minutes to run the BPE algorithm to generate the vocabulary from scratch, and training the model from scratch only takes a further 75 minutes on a Tesla T4. This means retraining is fast and consequently relatively inexpensive. This makes it feasible to retrain the model frequently, which would be useful if new data and labs are continuously being added to the dataset.

Almost all of the model's pipeline is implemented in TensorFlow. TensorFlow offers many easy deployment scenarios which would make taking this model to production relatively straightforward. Trained TensorFlow models can be deployed on a web server using TensorFlow Serving, exposing the model as an API. Models can also be deployed in a website using TensorFlowJS (TFJS). TFJS offers the advantages of inference being run on the end user's machine, making deployment less expensive as the server requires less computation resources. TFJS also offers greater end user privacy as inference is run on end-users machine, meaning no data is sent back to a central server, maintaining users' privacy of their data.

The only part of the model's pipeline that is not implemented in TensorFlow is BPE which we implemented using the SentencePiece library (Kudo & Richardson, 2018). As this library is only implemented in C++ and Python, this means that a different library would have to be used for BPE if the model was deployed on a website. But implementations of BPE in JavaScript do exist, so this shouldn't be too much extra engineering work.

References

Alley, Ethan C., Turpin, Miles, Liu, Andrew Bo, Kulp-McDowall, Taylor, Swett, Jacob, Edison, Rey, Von Stetina, Stephen E., Church, George M., and Esvelt, Kevin M. Attribution of genetic engineering: A practical and accurate machine-learning toolkit for biosecurity. *bioRxiv*, 2020. doi: 10.1101/2020.08.22.262576. URL https://www.biorxiv.org/content/early/2020/08/22/2020.08.22.262576.

Kudo, Taku and Richardson, John. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226, 2018. URL http://arxiv.org/abs/1808.06226.

Nielsen, Alec AK and Voigt, Christopher A. Deep learning

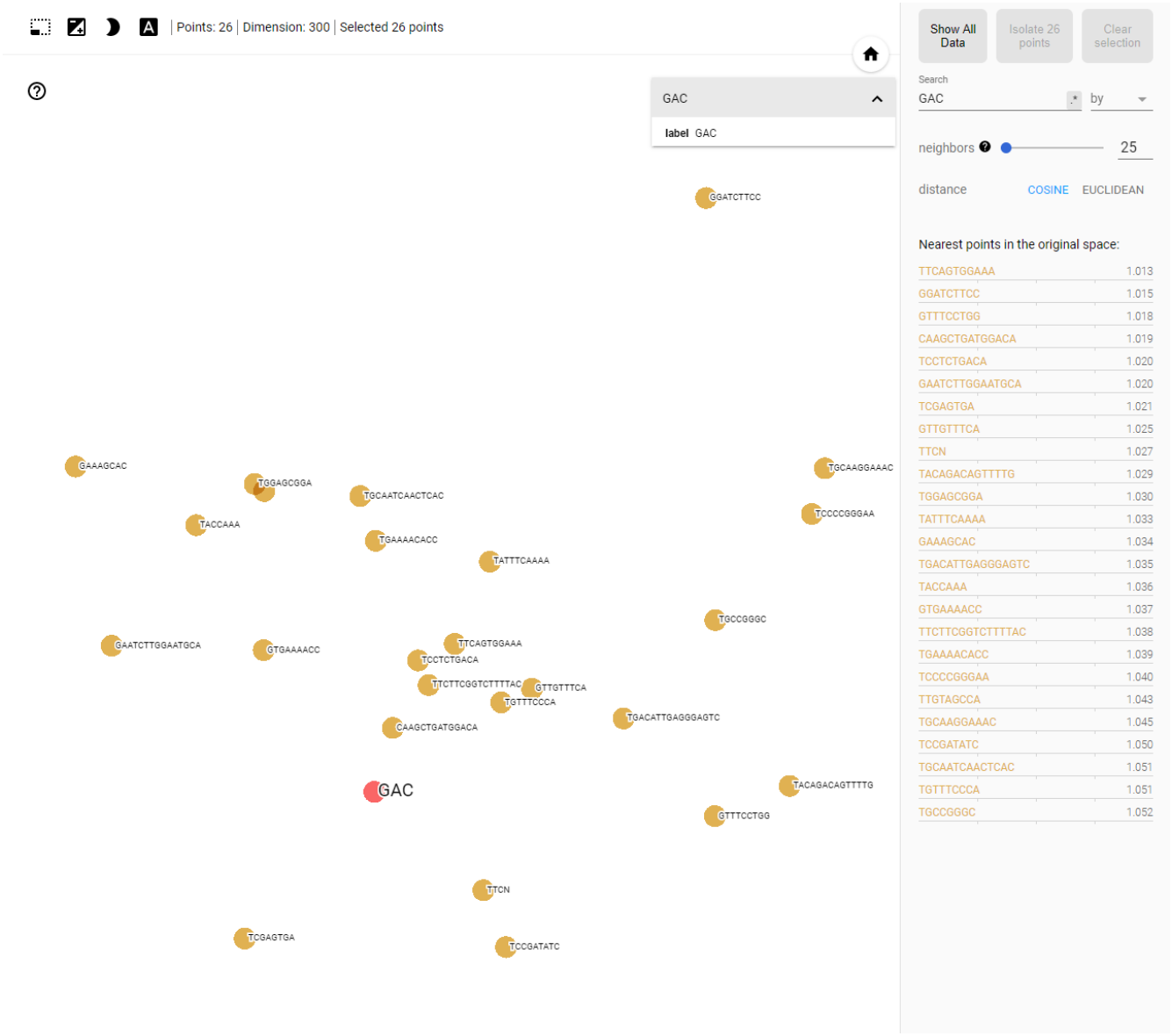

Figure 2. Twenty-five nearest neighbours of word "GAC", projected into two-dimensional space using PCA.

to predict the lab-of-origin of engineered dna. *Nature communications*, 9(1):1–10, 2018.

Sennrich, Rico, Haddow, Barry, and Birch, Alexandra. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015. URL http://arxiv.org/abs/1508.07909.

Shen, Dinghan, Wang, Guoyin, Wang, Wenlin, Min, Martin Renqiang, Su, Qinliang, Zhang, Yizhe, Li, Chunyuan, Henao, Ricardo, and Carin, Lawrence. Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms. *CoRR*, abs/1805.09843, 2018. URL http://arxiv.org/abs/1805.09843.