Classification of aligned 16S DNA Sequences

Project work for CAS Advanced Machine Learning, Univesity of Bern

Marco Kreuzer

2023-07-30

Contents

1 Introduction

Rapid advancements in genomic sequencing techniques have resulted in an exponential increase in the availability of biological sequence data such as the 16S ribosomal DNA (rRNA) sequences. The 16S rRNA sequences, universally present in bacteria and archaea, are used as a fundamental tool for microbial ecology studies, including identification, classification, and phylogenetic analysis. However, the massive influx of 16S rRNA sequences demands more efficient and accurate computational tools for analysis, interpretation, and understanding.

Traditional analysis methods, such as alignment-based techniques, may become computationally expensive and potentially less accurate as the sequence data size increases. Recent years have witnessed the rise of machine learning, and more specifically, deep learning techniques, which have been successfully applied in various fields, including image recognition, natural language processing, and genomics. The key advantage of deep learning is its ability to learn complex patterns in high-dimensional data, thus making it an attractive option for 16S rDNA sequence analysis.

The 16S ribosomal RNA (rRNA) gene, typically found in the genome of bacteria and archaea, is a highly conserved component of the 30S small subunit of prokaryotic ribosomes. Its function is pivotal to the process of protein synthesis within the cell. This gene encodes for the 16S rRNA molecule, which forms an integral part of the ribosomal machinery used for translating mRNA into proteins. The significance of the 16S rRNA gene in scientific research extends beyond its biological function. It has become a crucial tool for phylogenetic studies and taxonomic classification of bacteria and archaea due to its high conservation levels and the presence of variable regions within the gene. The 16S rRNA gene contains nine hypervariable regions (V1-V9) that show substantial diversity among different bacterial and archaeal species. These variable regions are interspersed among conserved sequences. By sequencing 16S rRNA genes, researchers can identify and classify the microbial species present in a sample, making it a standard method in microbial ecology, metagenomics, and microbiome studies.

DNA Sequencing technologies have significantly improved in the past decade. While it was common practice to only sequence one of the nine variable regions of the 16S rRNA gene, it now has become possible to sequence the whole region.

In this report, deep learning techniques are for the classification of 16S rRNA genes are explored. More generally, it provides a framework for data engineering and classifying DNA sequences of any DNA alignment.

1. Description of the data
2. Data engineering
3. Confolutional Classifier
4. Variational Autoencoder
5. Discussion and Outlook

2 Data description and engineering

DNA sequences are typically stored in a `.fasta` file format. Here is an example of a single sequence in this format:

```
>sequence_id1
ATGCCTT
```

A DNA alignment refers to a method by which multiple DNA sequences are arranged to maximize the similarity between corresponding nucleotides at each position. This alignment therefore identifies regions of similarity, providing insights into the functional, structural, or evolutionary relationships between the sequences.

An example of a DNA alignment represented in `.fasta` format could look like this:

```
>sequence_id1
ATGCCTT-GGCA-AGCTTGG
>sequence_id2
ATGC-ATTGGCATAAG-TGG
>sequence_id3
ATGCGTTGG-ATAAGCTTGG
>sequence_id4
ATGC-CTTGGCAT-AG-T-G
```

In this alignment, DNA sequences from four different organims are compared. The '-' represents gaps inserted to maximize the sequence similarity at each position. The comparison highlights the conserved nucleotides (like 'ATGC' at the start of all sequences) and the variable positions (such as the fifth and seventh nucleotides).

2.1 Data source

For this project, the SILVA database (https://www.arb-silva.de/) was used. This is a comprehensive resource that provides quality checked, and regularly curated datasets of aligned small (16S/18S, SSU) and large subunit (23S/28S, LSU) ribosomal RNA (rRNA) sequences for all three domains of life (Bacteria, Archaea, and Eukarya). In this study, an aligned version of reference sequences of the SSU (https://ftp.arb-silva.de/current/Exports/SILVA_138.1_SSURef_tax_silva_full_align_trunc.fasta.gz) was used. A crucial aspect of the SILVA database is that it includes hierarchical taxonomic information for each sequence in the sequence header. An example of a sequence header is given below:

```
>HG530070.1.1349 Bacteria;Actinobacteriota;Actinobacteria;Actinomycetales;
Actinomycetaceae;Trueperella;Trueperella pyogenes
```

Which corresponds to the following taxonomic levels:

```
>ncbi_identifier Domain;Kingdom;Phylum;Order;Family;Genus;Species
```

2.2 Data description

For this project, the dataset was subset to only include Bacteria and consists of ~1.9M sequences. The 16S sequence is typically 1500 base pairs (1.5 kb) long. Since a very diverse set of organisms are included in the data set, the alignment contains large amounts of gaps. Therefore, the total length of the alignment is 50000 base pairs long. The frequencies of bases are A: 0.73%, T: 0.6%, G: 0.91%, C: 0.66%. The remainder of poistions consists of gaps.

2.2.1 Sequence Taxonomy

Each sequence contains hierarchical taxonomic information as described above. However, many sequences do not contain all eight levels and would have to be curated manually. Therefore, the sequences were filtered to only include cases where the full taxonomy is known. This resulted in a dataset of 1788512 sequences (~1.7 M). Since machine learning classification tasks require to have multiple samples per class, the classes were filtered to include a minimum amount of samples per class. The number of unique classes per taxonomic level are given in Table 1.

Table 1: Number of classes within the domain Bacteria given a classification level. Min 1, Min 10 and Min 20 describe the number of classes that remain when each class has a minimum of 1,10 or 20 sequences.

Classification Level	**Min 1**	**Min 10**	**Min 20**
Kingdom	46	43	41
Phylum	114	110	103
Order	277	267	255
Family	582	558	525
Genus	3259	2520	2075
Species	151880	3947	1812

The taxonomic classes are highly unbalanced at every level in terms of members per class (see Figure 1).

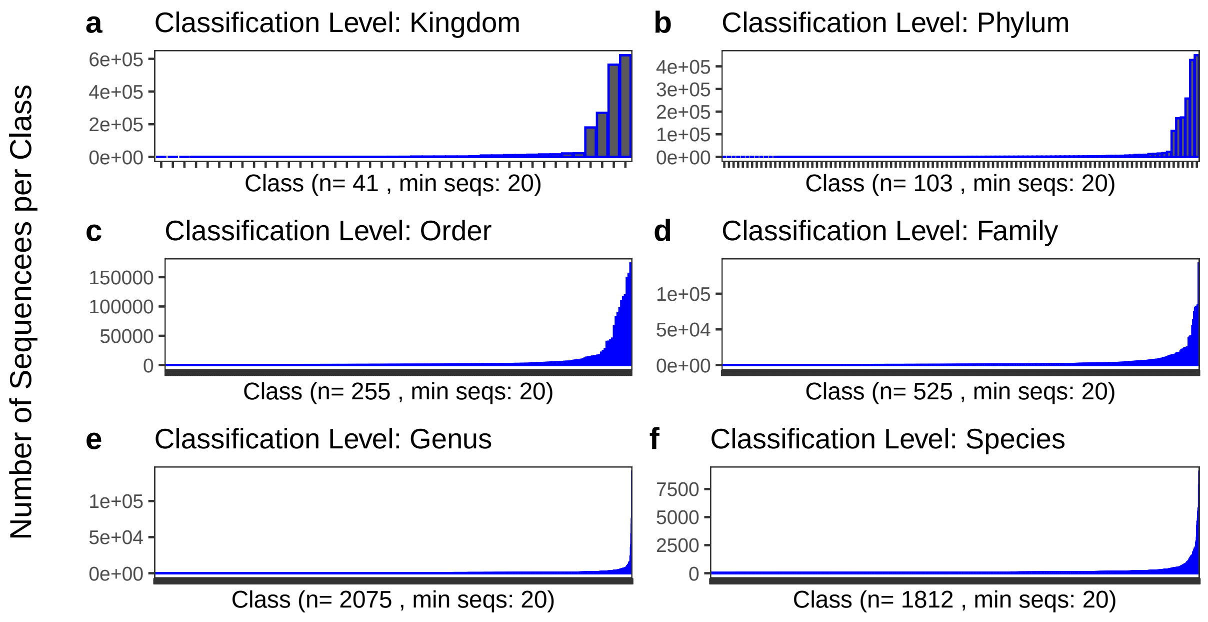

Figure 1: The datasets are highly imbalanced. **a-f**: The number of sequences in each of the classes at a specific taxonomic level. In all cases, only classes with more than 20 sequences were retained.

2.3 Data Engineering and Processing

2.3.1 One-hot encoding

DNA sequences are represented as strings of nucleotides (A,T,C,G). In the context of deep learning, this representation has to be one-hot encoded. Furthermore, gaps were treated as follows: If the gap is larger than four consecutive positions, the positions were encoded as `N` (missing data), otherwise the data was encoded as `-`. Therefore, a nucleotide can be encoded into six values (e.g. `[0,1,0,0,0,0]`).

This task has been achieved with the custom python class `hot_dna`. To instantiate the class, a DNA sequence and the taxonomic description as described above have to be supplied. The methods of `hot_dna` can be used to encode and decode a one-hot encoded DNA sequence.

```
class hot_dna:
    ### Class for One Hot Encoding DNA sequences
    def __init__(self, sequence, taxonomy):
        sequence = sequence.upper()
        self.sequence = self._preprocess_sequence(sequence)
        self.category_mapping = {'A': 0, 'C': 1, 'G': 2, 'T': 3, 'U': 3,
                                 '-': 4, 'N': 5}
        if sequence:
            self.onehot = self._onehot_encode(self.sequence)
        # splitting by ';' to get each taxonomy level
        self.taxonomy = taxonomy.split(';')

    def _preprocess_sequence(self, sequence):
        ambiguous_bases = {'R', 'Y', 'S', 'W', 'K', 'M', 'B',
                           'D', 'H', 'V', '.',}
        new_sequence = ""
        for base in sequence:
            if base in ambiguous_bases:
                new_sequence += 'N'
            else:
                new_sequence += base
        # replace sequences of four or more '-' characters with 'N' characters
        new_sequence = re.sub('(-{4,})', lambda m: 'N' * len(m.group(1)),
                              new_sequence)
        return new_sequence

    def _onehot_encode(self, sequence):
        integer_encoded = np.array([self.category_mapping[char] for char in sequence]).reshape(-1, 1)
        onehot_encoder = OneHotEncoder(sparse=False, categories='auto',
                                       handle_unknown='ignore')
        onehot_encoded = onehot_encoder.fit_transform(integer_encoded)

        # Fill missing channels with zeros
        full_onehot_encoded = np.zeros((len(sequence), 6))
        full_onehot_encoded[:, :onehot_encoded.shape[1]] = onehot_encoded

        return full_onehot_encoded

    def _onehot_decode(self, onehot_encoded):
        # Reverse the mapping dictionary
        reverse_category_mapping = {v: k for k, v in self.category_mapping.items()}
        # Convert one-hot encoding back to integer encoding
        integer_encoded = np.argmax(onehot_encoded, axis=1)
        # Convert integer encoding back to original sequence
        original_sequence = "".join(reverse_category_mapping[i.item()] for i in integer_encoded)
        return original_sequence
```

2.3.2 Processing large DNA alignments

A RAM-saving strategy was necessary for managing large DNA alignments. The fundamental approach involved processing a single sequence at a time, assigning it an index, and transforming the DNA string into a one-hot encoded tensor. These results were cataloged in a dictionary and preserved as a '.pt' file, named using the assigned index (for example, '0.pt'). An example of the contents of a `.pt` file is given below:

```
{'sequence_id': '3', 'sequence_tensor': tensor([[0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 1.],
        ...,
        [0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 1.]])}
```

The taxonomy information of the sequences were stored in a list and saved a a `pickle` file. The list consists of a dictionary with the correspoding index and the taxonomy labels:

```
{'sequence_id': '3', 'label': ['FW369114.1.1462', 'Bacteria',
'Proteobacteria', 'Alphaproteobacteria', 'Rhizobiales',
'Xanthobacteraceae', 'Bradyrhizobium', 'unidentified']}
```

The full dataset was created with the script `process_data.py`, where the main function is given below. The runtime of this operation was approximately six hours.

```
def process_sequences(msa_file_path, alignment_length, sequence_path,
                      full_taxonomy_labels, original_indices):
    full_labels = []

    with open(msa_file_path) as handle:
        for i, record in enumerate(SeqIO.parse(handle, 'fasta')):
            if i not in original_indices:
                continue

            if len(str(record.seq)[:alignment_length]) == alignment_length:
                encoded_dna = hot_dna(str(record.seq)[:alignment_length],
                                      record.description)
                sequence_tensor = torch.tensor(encoded_dna.onehot).float()

                original_index = original_indices.index(i)

                sequence_id = f"{original_index}"
                torch.save({"sequence_id": sequence_id,
                            "sequence_tensor": sequence_tensor},
                            f'{sequence_path}/{sequence_id}.pt')
                full_labels.append({"sequence_id": sequence_id,
                                    "label": \
                                    full_taxonomy_labels[original_index]})

    pickle.dump(full_labels, open(f'{sequence_path}/full_labels.pkl', 'wb'))
```

2.3.3 Data selection based on taxonomy

In this repository, a special focus has been set to be able to dynamically set classification tasks at different taxonomic levels. A part of the analyses in this report were based on 20% of the sequences from Phylum Actinobacteria in order to classify them to genus level (see Figure 2). Only classes that have more than 20 sequences per class were inlcuded. This resulted in a dataset of 92 classes with genus *Streptomyces* being the most prominent representative (n=4702 sequences) and genus *Timonella* having the least sequences (n=20).

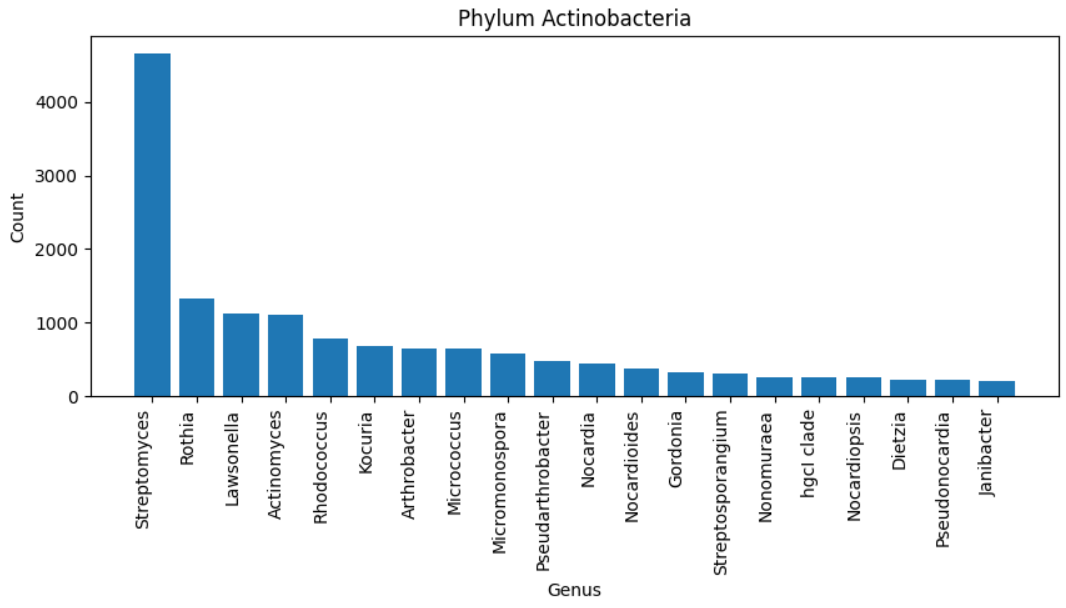

Figure 2: The size and the classes of the dataset if only 20 % of the sequences from the Phylum Actinobacteria are included. Only classes with more than 200 sequences are shown.

The second dataset used in this study contains 80% of the Domain Bacteria, which corresponds to 80% of the whole alignment. The minimum number of samples per group was set to 8 or 10 and the classificaton level was species. This dataset resulted in 4079 classes.

3 Convolutional Classifier

3.1 Models

The first deep learning method applied consists of a convolutional classifier, where four different models were tested. The models have similar structures, consisting of a combination of convolutional layers, max-pooling layers, and fully connected networks. They differ mainly in the number of layers, number of neurons in the FCN, and use of techniques such as dropout to avoid overfitting. The complexity and expressiveness of the models increase from SmallModel to LargerModel. The Kaiming He initialization was employed to initialize the layers in the models.

1. SmallModel (trainable parameteres: ~1.7 M): This model utilizes two convolutional layers and two max-pooling layers, followed by a three-layer fully connected network (FCN). The network starts with 6 input channels and gradually reduces down to 8 channels after the convolutional layers. Then, a FCN consisting of 256, 128, and num_classes neurons follows. The convolutional layers have relatively high strides (3 and 5) which might result in loss of spatial information.

2. ConvClassifier2 (trainable parameters: ~8.4 M): This model introduces more depth with six convolutional layers and four max-pooling layers. The number of channels increases from 6 to 32, and then gradually decreases to 4. The classifier uses an FCN with a higher number of neurons than

the SmallModel. Unlike SmallModel, it uses smaller strides in its convolutional layers, which could potentially capture more fine-grained patterns in the input data.

3. ModelWithDropout (trainable parameters: ~8.4 M): This model is identical to ConvClassifier2 in terms of architecture but introduces dropout in the FCN. Dropout helps to prevent overfitting by randomly setting a fraction of input units to 0 during training, which helps to improve the model's generalization capability.

4. LargerModel (trainable parameters: ~27.5 M): This model expands on the ConvClassifier2 and ModelWithDropout with an increased number of channels in the convolutional layers, going up to 64 channels. The FCN is also larger, potentially making this model more expressive at the cost of increased computational complexity and risk of overfitting.

3.2 Training

The training was performed using a grid search over specified hyperparameters. The hyperparameters were stored in the configuration file `config.yaml`:

```yaml
# Data specifications
data_folder: "/scratch/mk_cas/full_silva_dataset/sequences/"
alignment_length: 50000

# Taxonomy parameters
taxonomic_level: "Phylum"
taxonomic_group: "Actinobacteria"
classification_level: "Genus"
minimum_samples_per_group: 20
fraction_of_sequences_to_use: 0.2

# Hyperparameters
lr: [0.001, 0.0001, 0.00001]
n_epoch: [50]
batch_size: [128, 64, 32]
model: [ConvClassifier2, SmallModel, ModelWithDropout, LargerModel]
```

The data was prepared by loading and filtering labels based on a taxonomic filter specified in the configuration. The labels were then encoded, and the data was split into training, validation, and testing sets using the respective indices. Importantly, the data was split using the `StratifiedShuffleSplit` function in order to keep the classes proportionally represented.

For each parameter combination various auxiliary data such as the classification counts, label map, and the indices of the training, validation, and testing sets were saved. The Adam optimizer was used, with the learning rate specified by the current parameter set. Finally, the trained model's weights were saved.

The model was then switched to evaluation mode, and performance was assessed on the validation set, with both the true and predicted labels stored for later evaluation. Various evaluation metrics (like confusion matrix, training-validation loss curves, and F1 score) were computed and saved.

3.3 Results

Each model has been trained for 30 epochs and the F1 score calculated. The results are displayed in Figure 3. Overall, a lower batch size (32) outperformed batch sizes 64 and 128. The best results were achieved with the model `ConvClassifier2` and a learning rate of 0.0001 with an F1 score of 0.98. score was 0.98. Therefore, this model and settings were used for assessing the final performance with the test data. The results are dipslayed as a confusion matrix in Figure 5. The F1 score was 0.99.

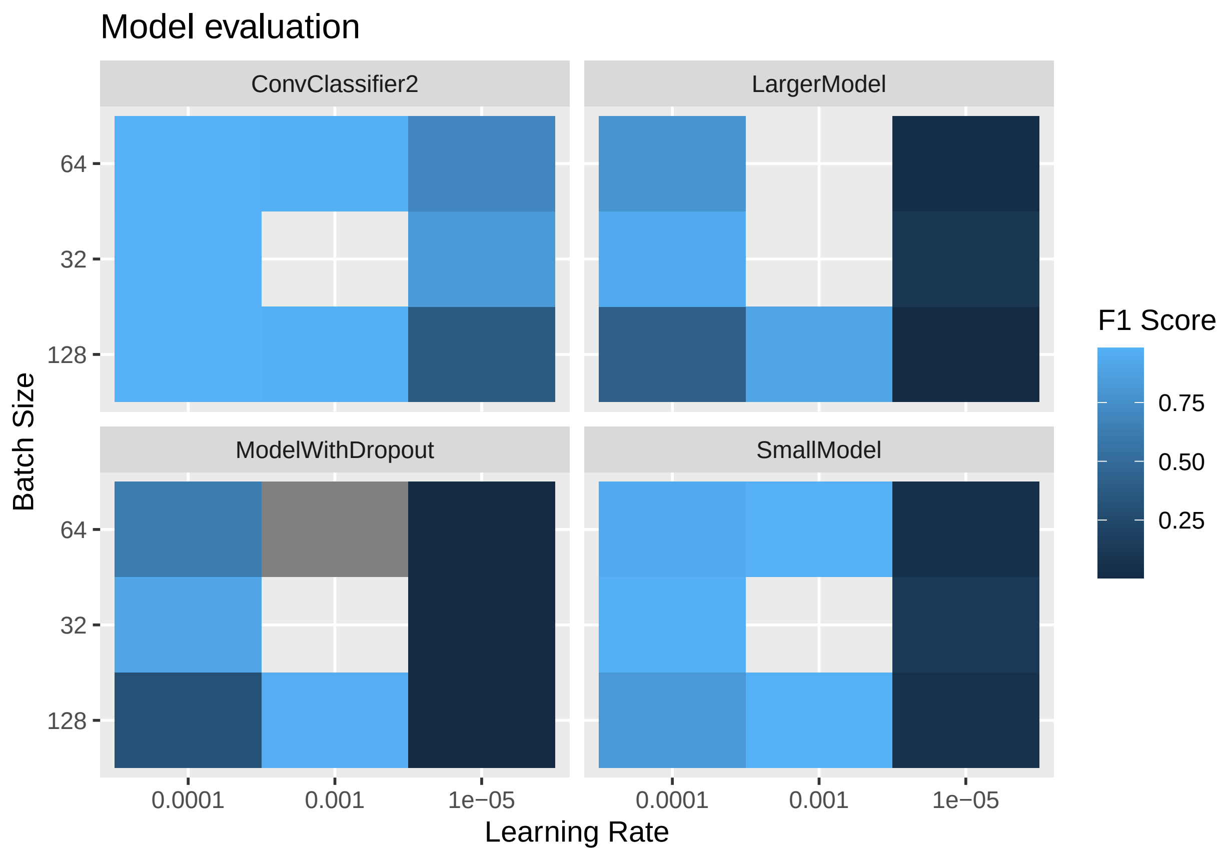

Figure 3: Comparison of different models with different hyperparameters.

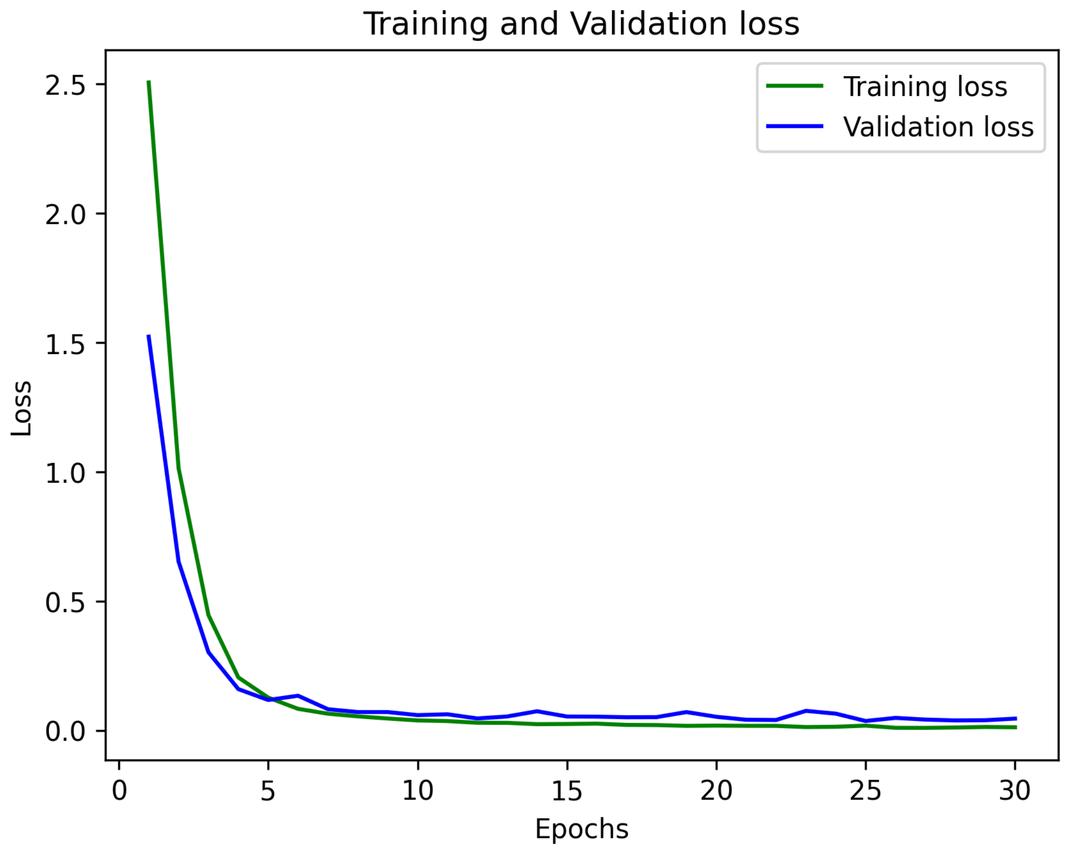

Figure 4: Training curve of the best model ConvClassifier2

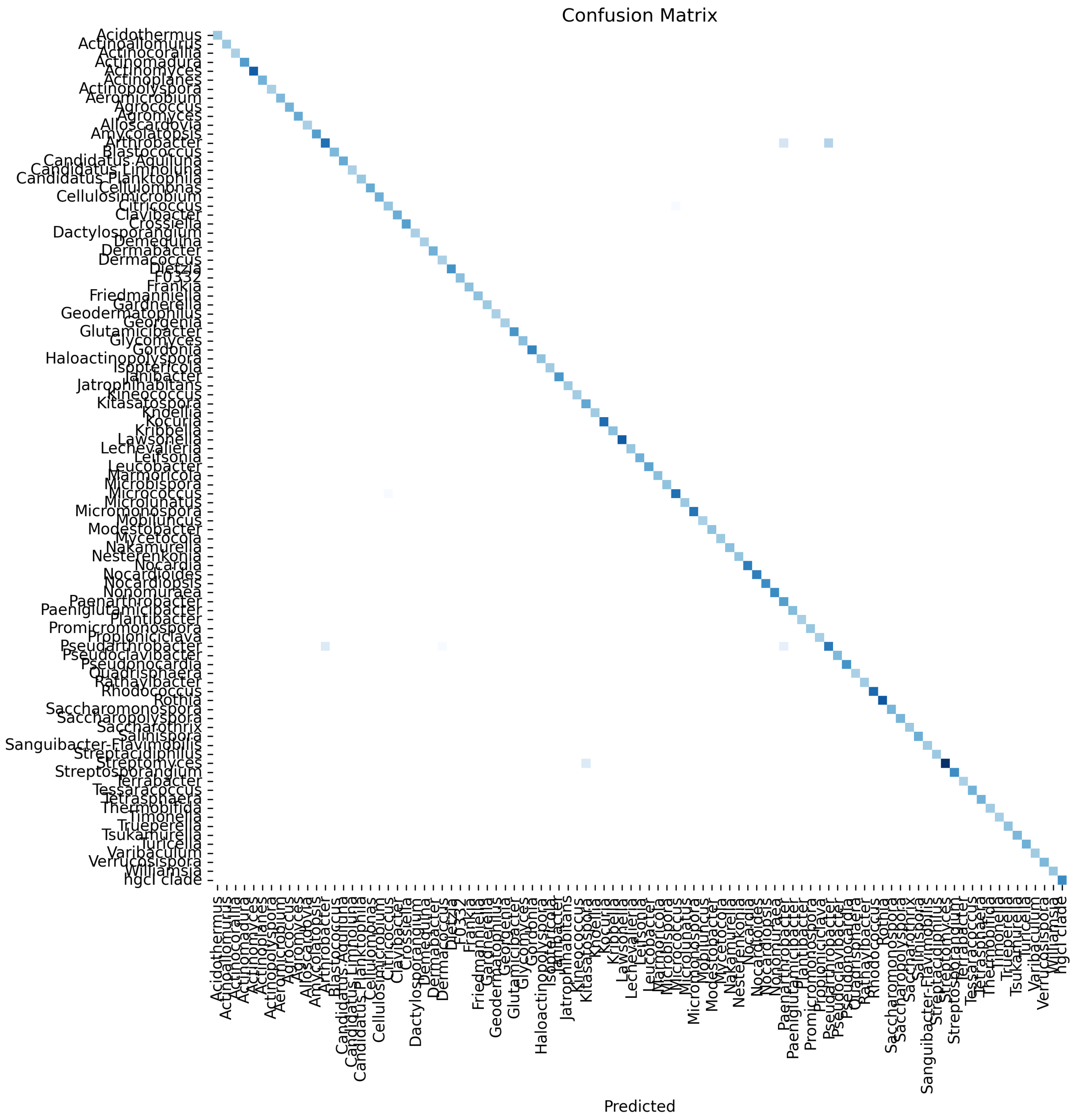

Figure 5: Confusion matrix of the best model.

4 Variational Autoenconder

Variational Autoencoders (VAEs) are a powerful class of generative models with diverse applications. Their utility extends to the analysis of DNA sequences. Primarily, the focus within this scope is to delve into the exploration of the latent space that these VAEs provide. The main goal is to discern whether DNA sequences show a proclivity for clustering at specific taxonomic levels.

4.1 Model

The model consists of three main components: an encoder, a decoder, and a reparameterization trick in the middle. It is primarily used for unsupervised learning of hidden representations and is able to generate new data that is similar to the training data.

1. **Encoder**: The encoder is a deep Convolutional Neural Network (CNN) that reduces the dimensionality of the input data, here DNA sequences, and encodes it into a latent space. The encoder includes several Conv1d layers with ReLU activations, interspersed with MaxPool1d layers for down-sampling, and ends with linear layers. The layers are designed to progressively decrease the spatial dimensions while increasing the depth, thereby learning more complex representations.
2. **Latent Space**: The output of the encoder is passed through two separate fully connected (linear) layers. One computes the mean (mu) and the other computes the log-variance (logvar) of the latent variables. These parameters are used to define a probability distribution of the latent variables.
3. **Reparameterization**: This is a technique used to allow backpropagation through random nodes. It introduces a random variable so that during training, the model generates a new value in the latent space that maintains the distribution's randomness. This is performed in the reparameterize function, which calculates the standard deviation (std) from the logvar, generates a random tensor (eps), and returns a new sample from the latent distribution (mu + eps*std).
4. **Decoder**: The decoder is another deep CNN, but in reverse order compared to the encoder. It maps the latent space back to the original input dimension. The decoder contains several ConvTranspose1d layers with ReLU activations for up-sampling, interspersed with Upsample layers to increase the spatial dimensions, and ends with a sigmoid activation function. The sigmoid function ensures the output is a probability distribution over the possible output values, which is suitable for a binary classification task such as predicting presence or absence of a particular feature in a DNA sequence.

In the forward function, the model first applies the encoder to the input, then the reparameterization trick, and finally the decoder. The model returns the reconstructed input, as well as the mean and log-variance of the latent variables. These outputs were then used in a loss function to train the model: the reconstruction loss measures how well the model can reconstruct the input, and the KL divergence measures how much the latent variable distribution deviates from a prior (usually a standard normal distribution).

4.2 Trainig

For the variational autoencoder, sequences which were member of a species that had at least 8 members were used for filtering and splitting the data into train, validation and test set.. This resulted in a data set of 227,835 sequences from 4079 species.

The