2 Materials and Methods

2.1. Data collection

The positive training dataset, i.e. Type III Secretion System effectors (T3SE), was obtained by taking the union of all T3SEs contained in BastionHub [download-dataset], the original training dataset used by EffectiveT3 2.0 and new T3SEs found in the literature (see 1, 2, 3, 4). BastionHub is a platform that contains a curated dataset of experimentally verified T3SEs with detailed annotations. This led to a total number of 1213 protein sequences.

The negative training dataset was obtained from the NCBI Reference Sequence Database RefSeq, a publicly available, non-redundant database for nucleotide and protein sequences. More specifically all protein sequences from the RefSeq database for all bacterial species that have at least one protein that was experimentally verified to be secreted by the Type III secretion system were downloaded. Given this large collection of protein sequences from multiple bacterial species a subset of 9698 protein sequences was approximately uniformly and randomly sampled over all bacterial species.

2.2 Data preprocessing

The protein sequences in the negative dataset were subject to two criteria. Firstly, the protein had to contain an N-terminal Pfam domain within the first 1-25 amino acids. This idea originates from the fact that the presence of a functional domain on the N-terminus could potentially interfere with the presence and recognition of a signal peptide, which are typically located at the N-terminus and approximately within the same length range.[1] The Pfam domains are provided by InterProScan [software] which uses HMMER, a statistical model to obtain information of possibly present Pfam domains on a protein sequence. Secondly, each protein sequence does not have a eukaryotic like domain (ELD). ELDs are protein domains found in eukaryotic genomes and display a higher frequency in genomes of host-associated bacteria compared to non host-associated bacteria, which indicates that proteins encoding these domains could be used as effectors. For this reason the non-presence of ELDs in the protein sequences increases the likelihood that our protein sequences are truly negative. A collection of ELDs was obtained by EffectiveELD, a tool developed by the Division of Computational Systems Biology (CUBE) at the University of Vienna.[2] The identification of ELDs is based on the evaluation of the taxonomic distribution of protein domains in a representative number of genomes of pathogens, non-pathogens and eukaryotes.

The final preprocessing step was the clustering of the obtained protein sequences using Cd-Hit [code]

(Cluster Database at High Identity with Tolerance). The program reduces the redundancy of the input protein sequences by clustering them based on sequence similarity. This step is crucial for decreasing the bias of the trained machine learning model. If the machine learning model was to encounter many highly homologous protein sequences it would learn a signal that may not be representative of the general signal and as such could be biased in weighing the importance of unseen protein sequences which are similar those homologous protein sequence set that occurs most often in the dataset. The clustering was performed in three steps (separately for the positive (secreted) and negative (not secreted) class).

Cd-Hit program parameters:

- *-c* | sequence identity threshold
- *-n* | word length
- *-g* | control whether sequence should be added to the first cluster for which it meets the threshold (0) or to the most similar cluster (1)
- *-aS* | alignment coverage for the shorter sequence, i.e. if set to 0.9 the alignment must cover 90% of the sequence
- *-G* | use global sequence identity (1) or use local sequence identity (0)
- *-T* | number of threads to use, if 0 then all threads will be used

1. Clustering by *cdhit* using the following parameter settings:

   Parameters: **-c** 0.9 **-n** 5 **-g** 1 **-aS** 0.4 **-p** 1 **-G** 0 **-T** 0

2. Clustering of the protein sequences obtained from step 1:

   Parameters: **-c** 0.6 **-n** 4 **-g** 1 **-aS** 0.4 **-p** 1 **-G** 0 **-T** 0

3. Clustering of the protein sequences obtained from step 2 using *psi-cd-hit_new*. Psi-cd-hit uses a similar algorithm like Cd-Hit, but uses Blast to calculate the similarities. Psi-cd-hit is required for the last step because the program Cd-Hit does not recommend to cluster protein sequences at an (PID) threshold of less than 40%. Therefore psi-cd-hit was used in combination with *blastp* from the Blast + executables for computing the protein sequence similarity, which works more reliable than Cd-Hit for lower PIDs.

   Parameters: **-c** 0.3 **-prog** blastp **-aS** 0.4 **-g** 1 **-G** 0

This results in a positive training dataset with 304 protein sequences and a negative training dataset of 3463 protein sequences, where each protein sequence displays at most a similarity of 30% to any other protein sequence within the same class, given that the alignment coverage for the shorter sequence is above 40%. Subsequently 10 different training-test dataset splits where obtained by sampling 270 random protein sequences from the positive training dataset and 2000 protein sequences from the negative training dataset. The remaining 34 positive protein sequences represent the positive test dataset while 34 additional protein sequences have been sampled from the protein sequences remaining in the negative dataset. The choice of f negative protein sequences was deliberate, as machine learning models often struggle to learn effectively when there is a significant imbalance between the positive and negative class.

2.3. Feature engineering

Feature engineering refers to the process of transforming raw data (e.g. strings of amino acids) into numerical features that represent the underlying patterns of the data that can be exposed to the learning algorithms of machine learning models. Effective T3 is a model that solely incorporates features directly extracted from the amino acid sequence. The reasons are two-fold. Firstly, the model should learn a general signal for secretion. Studies have shown that the signal for secretion of T3SS effectors is contained in the N-terminal region of the protein sequence. Removing the N-terminal region of a protein leads to an abolishment of secretion of these proteins [4]. Secondly, eschewing features that require additional information as for instance position specific scoring matrix (PSSM) composition encoding or protein disorder decreases the total computational cost.

The following protein sequence encodings have been considered for use in Effective T3:

Amino acid composition (AAC): This encoding refers to the frequency of each amino acid within a protein sequence. This results in a 20 dimensional feature-vector for each protein sequence.

$$f(i) = \frac{N_i}{N} \qquad i = 1, 2, \dots, 20.$$

N_i represents the number of amino acids of type i while N corresponds the length of the protein.

Dipeptide Composition (DPC): 20 different amino acids lead to a total of 400 dipeptide combinations. This type of encoding provides a more detailed information since it also takes the local order of the amino acids into account.

$$f(i,j) = \frac{N_{ij}}{N-1} \qquad i,j = 1, 2, \dots, 20.$$

N_{ij} is the number of dipeptides with amino acids of type i, j in given order of amino acids and N represents the length of the protein sequence.

Composition, Transition, and Distribution (CTD): A type of encoding that incorporates three descriptors. The following amino acid properties and classes have been used to obtain the composition descriptor (*CTDC*) and the transition descriptor (*CTDT*) for each protein sequence:

Amino Acid Property	**Class 1**	**Class 2**	**Class 3**
Hydrophobicity PRAM900101	Polar	Neutral	Hydrophobic
	RKEDQN	GASTPHY	CLVIMFW
Normalized van der Waals volume	0-2.78	2.95-4.0	4.03-8.08
	GASTPDC	NVEQIL	MHKFRYW
Polarity	4.9-6.2	8.0-9.2	10.4-13.0
	LIFWCMVY	PATGS	HQRKNED
Polarizability	0-1.08	0.13-0.19	0.22-0.41
	GASDT	CPNVEQIL	KMHFRYW
Charge	Positive	Neutral	Negative
	KR	ANCQGHILMFPSTWYV	DE
Secondary structure	Helix	Strand	Coil
	EALMQKRH	VIYCWFT	GNPSD
Solvent accessibility	Buried	Exposed	Intermediate
	ALFCGIVW	RKQEND	MSPTHY

Composition encodes the fraction of each class in the protein sequence. It is defined as

$$C_x = \frac{n_x}{N} \quad x = 1, 2, 3$$

where n_x corresponds to the number of amino acid type x in the protein sequence and N is the length of the protein sequence.

Transition computes the frequency that one class is followed by another class in the sequence and vice versa:

$$T_{xy} = \frac{n_{xy} + n_{yx}}{N - 1} \quad xy = 12, 13, 23$$

where n_{xy}, n_{yx} reflect the number of dipeptides xy and yx in the sequence, while N is the length of the protein sequence.

Amino Acid Property patterns (AAPP): The different amino acids can also be distinguished by patterns of the following physicochemical properties: polar, hydrophilic, hydrophobic, acidic, alkaline, aromatic and ionisable. Using this set of properties (as described in the original Effective T3 paper[5]), one can obtain a feature vector encoding different combinations of physicochemical properties. Here, we considered amino acid properties with a maximal length of 3. This results in a 71-dimensional feature vector per protein sequence. The list of property patterns can be found in the appendix (add link to section). The 20 amino acids have been assigned to their corresponding amino acid properties the following way:

Physicochemical property	Amino acids
Polar	N, Q, S, T
Aromatic	V, W, Y
Acidic	D, E
Alkaline	K, L, R
Ionizable	C, Y
Hydrophobic	V, M, L, A, I, G
Hydrophilic	S, F, T, N, K, Y, E, Q, C, W, P, H, D, R, U

Given the 71 different physicochemical property patterns of length 1-3 as provided in the appendix, one can obtain a 71-dimensional feature vector for each protein sequence by applying the following formula for each listed pattern and concatenating the results:

$$A_p = \frac{n_p}{N-L} \quad p = 1, 2, \dots 71$$

where A_p represents the frequency of the pattern p, n_p is the number of occurrences of pattern p in the sequence and L is the length of the pattern p.

2.4. Machine learning model and optimization process

The Light Gradient Boosting Machine (LightGBM) is a gradient boosting framework that incorporates tree-based learning algorithms. Two of its main features are: Gradient-Based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). The GOSS-method filters out data instances based on the magnitude of their gradient - in short it keeps instances with large gradients and randomly samples instances with smaller gradients. EFB bundles mutually exclusive features (two features can be considered mutually exclusive, when they never take non-zero values at the same time for all instances) into a single feature, designed to reduce the dimensionality of the dataset without loss of information.

The LightGBM model in combination with the gradient-based decision tree (GBDT) learning algorithm has been used for Effective T3. Since this model offers a plethora of hyperparameters and we are dealing with a rather small and imbalanced training dataset, care must be taken to avoid training a model that does not generalize well on unseen data.

For this reason, to ensure the model is properly regularized and to avoid overfitting, bagging of hyperparameters is done. Bagging is an ensemble learning method that creates multiple versions of the model by training on different subsets of the data and then combines their outputs. It helps to stabilize the learning and reduces the variance of the model.

The following shows a listing of all hyperparameters included in the optimization procedure:

Hyperparameter	**Description**
boosting_type	Specifies the type of algorithm to use for training the model weights.
learning_rate	The step size to perform while moving towards the minimum of the optimization function.
num_leaves	Controls the complexity of the tree model by controlling the maximum number of leaves in one tree. Should be less than 2^{max_depth}. A crucial parameter to control overfitting.
max_depth	Sets the maximum allowed depth of one tree. Crucial to control overfitting.
min_child_samples	Minimal number of training instances that need to be present in a node for the node to become a leaf. If there are more instances than the nodes in the tree need to be split until the number of instances is below the threshold. Crucial parameter to control overfitting.
max_bin	Minimum number of bins that feature values will be bucketed into. A smaller bin count reduces the possibility of overfitting.
colsample_bytree	Fraction of columns/features to be randomly sample for each tree in every iteration of the training process.
min_child_weight	Acts as a regularization term in the objective function, where higher values

	cause the model to create fewer leaves, which is a way to control the complexity of the model.
reg_alpha / reg_lambda	Also known as L1 / L2 regularization. L1 encourages sparsity in the model, while L2 encourages the model weights to be small, effectively reducing the model complexity.
drop_rate	Specifies the fraction of trees to drop at each training iteration. In general it makes a model more robust, since dropping out trees means that the trained model cannot rely on individual trees. (only used in the *dart* gradient-boosting method)
max_drop	Maximum number of trees to drop in one iteration (only used in *dart*)
subsample / bagging	Determines the size of the randomly selected fraction of training data to use for any tree building. Since each tree is trained on a slightly different dataset it decreases the chance of an overfitted model.
subsample_freq	The frequency of boosting iterations at which to sample *subsample*-sized fractions of training data to use for each tree.
path_smooth	Strength of smoothing applied to tree nodes. This ensures overfitting on leaves with few samples to be less likely.
min_split_gain	Sets the minimum reduction in the loss required to further partition the leaf node of a tree.

We optimized over the hyperparameter space[appendix] via a two-step approach.

1. One-by-one hyperparameter optimization: each hyperparameter was optimized individually using the grid search technique with 4-fold cross-validation as implemented by the scikit-learn machine learning library. Once a hyperparameter was optimized, the value was included to initialize the next model that was used to grid search the next hyperparameter value in the listing above until all hyperparameters were obtained.
2. Heuristic optimization: a new model was initialized with the hyperparameter values obtained from step 1. Due to the large hyperparameter space it was deemed necessary to use a heuristic approach. This was accomplished by applying an evolutionary algorithm, more specifically, a genetic algorithm, in combination with 4-fold cross-validation, as implemented in the sklearn-genetic module. The range r_h of each continuous hyperparameter value used for heuristic optimization can be obtained using

$$r_h = [\hat{h} * 0.2\,, \hat{h} * 1.8]$$

where $\hat{h}$ is the optimal hyperparameter value retained in step 1. The range of integer-type hyperparameters were obtained by rounding the outer bounds either up or down.

Genetic optimization was done over 4 generations with a population size of 30 per generation. This means that the gradient boosting model consisted of 30 different decision trees.

2.5. Feature Selection

The initial feature dimension of 513 for each protein sequence was reduced to 84, among which 83 make up dipeptide compositions and the remaining one is the *polar* amino acid property pattern of length one (see amino acid property patterns on page 10). The feature selection process involved the following steps:

- Sample 10 train-test splits from the entire preprocessed training dataset. To be more precise, given 304 secreted and 3463 non-secreted proteins, each train-test split included the entire dataset but with a different distribution among training and test set: 270 positive training samples, 34 positive test samples, 2000 negative training samples and 34 negative test samples.
- Train a classifier on each of the 10 training datasets and compute the mean of the absolute SHAP-values (Shapley Additive exPlanations) over the corresponding 10 test datasets.
- Normalize the SHAP-values over all 513 features and select the 84 features with the highest absolute SHAP-values
- Retrain the model with the previously optimized hyperparameters, but on the selected subset of 84 features.

Reasons for choosing SHAP-values for the feature selection process:

1. They provide instance-level feature importance, so that we know how much each feature contributes to every individual prediction. This is of particular interest when dealing with complex datasets where the impact of each feature can vary significantly from one instance to another.
2. By aggregating the feature importance across all instances we can obtain a measure of global feature importance, which allows us to figure out which features are generally most influential in the model's prediction process.
3. SHAP-values account for interaction effects between features. In other words, they provide the marginal contribution of each feature, factoring in all possible coalitions of features, capturing complex and non-linear relationships.
4. If a machine learning model were to change, such that it relies more on a particular feature then the attributed importance for that feature will not decrease. Hence, SHAP-values can be regarded as reliable and justifiable when used to rank the significance of features.

3 Prediction Tool

In the following we describe the handling of the prediction tool Effective T3 3.0 (https://github.com/nicolasrnemeth/EffectiveT3) and the Bastion 3 clone (https://github.com/nicolasrnemeth/bastion3clone).

Make sure pip is installed. Install Effective T3 using “pip install EffectiveT3”. If you have git installed you can also clone the git repository using “git clone https://github.com/nicolasrnemeth/EffectiveT3.git” and install the tool from within the folder using “pip install .”. The same steps should be followed when installing the Bastion 3 clone.

The Effective T3 installation comes with a packaged and trained model. Predictions can be executed using the command “effectivet3”. Entering “effectivet3 --help” into the console shows an overview of the possible parameters and options controlled by this command:

```
effectivet3 [-h] -f FILE [-o OFILE] [-c {1,2,3,4,5,6,7,8}] [-l TRUELABELS] [-v]
```

Arguments:

-h, --help show help message and exit

-f FILE, --file FILE (required): path to the input fasta-file, e.g. 'your_folder/your_file.fasta'

-o OFILE, --ofile OFILE (required): provide the file path for the output file

--ofile path/{file_name}.json to save it in json-format or path/{file_name}.txt to save it in txt-format

-c {1,2,3,4,5,6,7,8}, --cores {1,2,3,4,5,6,7,8} (optional): the number of CPU-cores that you want to use for prediction. By default all available CPU cores are used. If your selected number of cores is above the available number of cores you will be provided with the number of accessible CPU-cores on your operating system, to provide a valid choice the this parameter.

-l TRUELABELS, --truelabels TRUELABELS (optional): path to the file containing the comma-separated true labels encoded as integers (0 for False (not-secreted) and 1 for True (secreted)). The labels must be in the same order as the protein sequences contained in the input fasta-file. If this command line parameter is set, i.e. not None. Then an additional file ("{output_file_name_as_set_in_the_command_line_argument}_metrics.json") containing all kinds of evaluation metrics for the prediction will be saved. The json-file containing the evaluation metrics will be saved inside the same path as the output-file name containing only the predictions but with "_metrics.json" concatenated to the filename. Example: if you have 10 protein sequences, then your labels file should contain the following: “1,1,0,1,0,0,0,1,0,0” assuming you have 4 secreted and 6 non-secreted proteins in the exact same order. There must be neither spaces in between the commas nor trailing commas.

-v, --version (optional) Show program's version number and exit

The command “effectiveTrain” can be used to train a model, save its optimized hyperparameters and evaluation metrics. Entering “effectiveTrain --help” into the console provides an overview of all possible parameters and options:

usage: effectiveTrain [-h] -p POS -n NEG [-i]

Arguments:

-h, --help show this help message and exit

-p POS, --pos POS (required) Path to the fasta-file containing positive protein sequences.

-n NEG, --neg NEG (required) Path to the fasta-file containing negative protein sequences.

-i, --featureimportance (optional) Set this flag to save feature importances and their labels to a file with the same filepath as the metrics file, but named 'feature_importances.json'