**Supplementary Material: Detailed Methodology of FAALPred**

This supplementary material contains a comprehensive description of the methodology and steps implemented in the Python code of the FAALPred program for the classification of Fatty Acyl-AMP Ligase (FAAL) substrates. The code integrates various processes — including data preprocessing, sequence alignment, feature embedding extraction, Word2Vec and Random Forest model training, and performance evaluation — to classify new FAAL protein sequences according to the associated variable categories (fatty acid specificities) using a Random Forest classifier.

**1. Data validation and pre-processing**

**Loading and checking the data:**

The first step is to load the sequence alignment file and the corresponding data table. The paths to the alignment file and to the table data are specified. The table data is loaded using the Pandas library. The script checks the existence of the alignment file to ensure that exactly one alignment file is found. This is crucial to ensure that the correct data set is used for further analysis.

**2. Generation of the protein embedding**

The ProteinEmbeddingGenerator class orchestrates the generation of numerical vector embeddings for protein sequences by combining k-mer extraction with Word2Vec training and an optional aggregation step. Upon initialization, it receives the path to a FASTA file of protein sequences, a DataFrame that can associate each sequence ID with additional experimental variables if desired, and an aggregation method that determines whether the k-mer embeddings should be concatenated (“none”) or averaged (“mean”). Before the alignment is loaded, the class checks whether the sequences are already aligned by calling are\_sequences\_aligned. If this is not the case, realign\_sequences\_with\_mafft executes MAFFT to align the sequences and saves the aligned FASTA under a new name. Next, the constructor loads the final alignment via BioPython’s AlignIO and saves it in self.alignment. Once generate\_embeddings is called, it loops through each dataset in that alignment. For each protein, k-mers are extracted by moving a window of size k with a step size along the sequence. All k-mers consisting only of hyphens are discarded, and k-mers containing less than k letters (due to gaps) can also be skipped. All these collected k-mers form the “sentences” that train (or are derived from) a Word2Vec model. If no previously trained model is found at the specified location, the class automatically trains a new model with parameters such as vector\_size=390, window=window, epochs=epochs and hierarchical softmax enabled (hs=1). This creates a learned embedding vector for each unique k-mer found. After training the Word2Vec model, the class determines the minimum number of k-mers (min\_kmers) among all protein sequences, which is critical for creating consistent length embeddings. For each sequence, only the first min\_kmers k-mers are selected. If a particular protein has fewer k-mers than min\_kmers, the remainder is filled with null vectors. Then, depending on self.aggregation\_method, either the Word2Vec embeddings of these k-mers are concatenated (if “none”) or their average is calculated (if “mean”) to create a single numeric field. Finally, each embedding field is collected in self.embeddings and a default scaler is either loaded (if it already exists in the model dictionary) or retrained on these embeddings for consistent scaling.

Imagine a slightly larger alignment of three protein sequences, each with eight columns, and suppose we want to extract 3-mers. We can represent the alignment in a simple table:

**Protein Alignment:**

Column: 0 1 2 3 4 5 6 7

Protein1: M A A K X Y Z -

Protein2: M A A - W Y Z K

Protein3: M A A G W Y Z K

**Step 1: Extracting k-mers (with k = 3)**

For each protein, a sliding window of length 3 is moved along the sequence to gather overlapping 3-mer segments:

**Protein1:**

* Columns [0..2]: "MAA"
* Columns [1..3]: "AAK"
* Columns [2..4]: "AKX"
* Columns [3..5]: "KXY"
* Columns [4..6]: "XYZ"
* Columns [5..7]: "YZ-"

*(Note: “YZ-” includes a gap. If the rule allows up to two gaps, this k-mer remains valid.)*

**Protein2:**

* Columns [0..2]: "MAA"
* Columns [1..3]: "AA-"
* Columns [2..4]: "A-W"
* Columns [3..5]: "-WY"
* Columns [4..6]: "WYZ"
* Columns [5..7]: "YZK"

**Protein3:**

* Columns [0..2]: "MAA"
* Columns [1..3]: "AAG"
* Columns [2..4]: "AGW"
* Columns [3..5]: "GWY"
* Columns [4..6]: "WYZ"
* Columns [5..7]: "YZK"

All these 3-mer strings from all proteins are then combined into a single “corpus” If a 3-mer contains three hyphens (k=3, all gaps), it is removed. Otherwise, it becomes a valid training token for the Word2Vec model. Next, Word2Vec learns a vector representation (e.g. of size 390) for each unique 3-mer token. Thus, tokens such as “MAA",” “AAK",” “XYZ",” “YZ-",” “AA-",” “WYZ” or “YZK” each receive their own learned embedding vector.

The script then specifies a minimum number of k-mers (min\_kmers) among all sequences. In the example above, protein1 and protein3 each have six valid 3-mers. If protein2 had only five valid 3-mers after discarding the sequences with too many hyphens, the global minimum would be five. In this case, each protein is restricted to the first five k-mers, or the sequence that has fewer is padded with null vectors.

Finally, the aggregation method determines how these k-mer embeddings are combined into a final vector for each sequence. If the aggregation method is “none",” the individual k-mer embeddings are linked together. With five valid k-mers and a Word2Vec vector\_size of 390, the final embedding of each protein would consist of 5 × 390 elements. If the method is “average",” we take the average over these k-mer embeddings and generate a single 390-dimensional vector per protein.

This entire procedure is repeated for each protein in the alignment, resulting in a numerical embedding array of uniform size for each sequence. These embeddings are then scaled (e.g. with StandardScaler) and stored in self.embeddings so that they are suitable for classification, clustering or other machine learning tasks.

**3. Standardization:**

Once the embeddings are generated, a standard scaler is used to standardize the embeddings and ensure that they have a mean of zero and a standard deviation of one. This step is important to optimize the performance of machine learning algorithms. The scaler is saved for later use when converting new data.

**4. Support class**

The Support class is designed to provide a comprehensive framework for training, tuning and evaluating Random Forest models on imbalanced datasets using advanced oversampling techniques. Its design ensures that each class in the dataset is adequately represented during model training and cross‐validation, improving the robustness of the prediction model. During initialization, the constructor (init) sets up several important parameters and attributes. It accepts the number of cross-validation folds (cv), a random value (seed) for reproducibility and the number of parallel jobs (n\_jobs) to speed up the calculation. In addition, lists are initialized for tracking training results, test results, F1 results, AUC (precision recall) results and ROC results across folds. A default scaler instance is also created for later normalization of the feature space. Two sets of hyperparameters are defined: one, called init\_params, provides a conservative baseline (with settings such as n\_estimators set to 100, max\_depth of 2, and class\_weight set to "balanced\_subsample") to minimize overfitting; the other, stored in parameters, specifies a grid of values for subsequent hyperparameter optimization using grid search.

**4.1. Key Features:**

**4.2. Oversampling:**

The internal method \_oversample\_single\_sample\_classes implements a customized oversampling strategy. The oversampling process implemented in the \_oversample\_single\_sample\_classes method is designed to ensure that each class in the dataset contains at least (cv + 1) samples. This is critical in biological datasets, as some classes may have very few samples, which can lead to biased model training. The method proceeds in several steps: First, the original class distribution is calculated by counting the occurrences of each label in the input field y using Python’s Counter. Based on this count, a sampling strategy is created in which the target number of samples for each class is set to the maximum between the current count and (cv + 1). This ensures that even classes with a very low initial count are increased to meet the minimum requirement. Next, the method applies the RandomOverSampler from the imbalanced-learn library. This tool randomly replicates samples from the minority classes until the number of samples for each class reaches the target defined in the previous step. If additional metadata (such as protein IDs and associated variables) is provided during this process, the method captures synthetic sample identifiers for the new data points generated by RandomOverSampler. After the initial oversampling, the method further refines the class balance using SMOTE (Synthetic Minority Over-sampling Technique). SMOTE creates new synthetic samples by interpolating between the existing minority samples. This step is applied to the data that has already been augmented by the RandomOverSampler. Again, synthetic identifiers are added for the samples generated by SMOTE if the corresponding metadata is available. After both oversampling steps, the method writes the final class distribution (after SMOTE) to a file for recording and debugging purposes. Finally, it returns a tuple containing the oversampled features, labels and any synthetic metadata captured during the process. This two-step approach (first replicating existing samples, then generating new synthetic samples by interpolation) ensures robust class balancing, which is essential for reliable downstream model training and evaluation. An optional UMAP visualization of the latent space is created with a separate function (visualize\_latent\_space\_with\_similarity) to illustrate the relationship between original and synthetic samples after oversampling.

**4.3. Cross-validation and tuning of hyperparameters:**

For model evaluation, the fit method uses StratifiedKFold cross-validation. To solve the potential problem of classes with extremely few samples, the number of splits is dynamically adjusted so that each fold receives a sufficient representation of each class. In each fold, the training set is resampled to maintain balance, and a RandomForestClassifier is instantiated with the conservative initial parameters. The model is trained with the resampled training data and its performance is evaluated using metrics such as training score, test score, weighted F1 score and precision-recall AUC on the test set. The ROC AUC is calculated either directly for binary classification or for multi-class classification by binarizing the labels. The results for each fold are logged and saved. After cross-validation, the fit method calls the \_perform\_grid\_search method to optimize the hyperparameters. The \_perform\_grid\_search method is responsible for automatically tuning the hyperparameters of the random forest model. In this function, the oversampled training features and the corresponding labels — X\_train\_resampled and y\_train\_resampled— - are used as input. First, a StratifiedKFold cross-validation object is instantiated with the number of splits specified by the cv parameter to ensure that each split maintains the same class distribution as the entire dataset. This is particularly important for unbalanced datasets, as it prevents a single class from dominating a fold. Next, the method sets up a GridSearchCV instance in which a new RandomForestClassifier (with the random seed for reproducibility) is defined as the base estimator. The grid search is configured to iterate over a predefined set of hyperparameters. These include options for the number of estimators, the maximum depth, the minimum samples required to split an internal node, the minimum samples per leaf, and several other parameters defined in self.parameters. Grid search uses the StratifiedKFold object for cross-validation, utilizing multiple CPU cores in parallel (n\_jobs) and evaluating each parameter combination against a scoring metric based on the one-versus-one ROC AUC (roc\_auc\_ovo). During cross-validation, performance metrics such as training score, test score, weighted F1 score, precision-recall AUC and ROC AUC are calculated and logged. The verbose parameter is set to provide detailed output during the grid search, which can be useful for debugging or understanding the tuning process. Once the grid search has been performed by fitting to the oversampled training data, the method logs the best hyperparameters discovered during the search. It then returns a tuple consisting of the best estimator — i.e. the RandomForestClassifier configured with the optimal hyperparameters — and a dictionary of these best parameters. This refined model is intended for further calibration and evaluation to ensure that the final classifier is tuned to the specific characteristics of the training data. The best estimator and its corresponding parameters are then selected and stored.

**4.4 Model calibration:**

After the grid search, the model is further refined by integrating the likelihood calibration by CalibratedClassifierCV using an isotonic regression method. This calibrated model is then saved to disk and returned by the fit method. The get\_best\_param method is a simple accessor that retrieves specific hyperparameter values from the best\_params dictionary determined by the grid search.

**4.5. Evaluation metrics:**

The plot\_learning\_curve method visualizes the evolution of the performance metrics (training score, cross-validation score, F1 score, and precision-recall AUC) across the cross-validation folds; the resulting plot is saved in a custom file.

The get\_class\_rankings method uses the trained and calibrated model to obtain the predicted class probabilities for new data and then returns a sorted, formatted list of class rankings for each sample.

**4.6. Test Best Random forest (RF) model**

The test\_best\_RF method provides an external validation procedure. It starts by loading a pre-trained standard scaler (from a file such as scaler\_associated.pkl) and transforming the feature matrix. The method applies the oversampling procedure to the entire data set before splitting it into training and test sets. A RandomForestClassifier is then instantiated using the best hyperparameters found in the grid search and the model is retrained using the training data. The model is calibrated with CalibratedClassifierCV and then used to predict probabilities on the test set. These predictions are adjusted using the adjust\_predictions\_global function and the final performance metrics — including ROC AUC (calculated using the \_calculate\_score utility method), weighted F1 score and precision-recall AUC — are calculated. The method returns a tuple containing these metrics, the best parameters, the final calibrated model and the test data split.

The \_calculate\_score utility method is designed to calculate the ROC AUC in a way that adapts to both binary and multi-class settings. For binary classifications or one-dimensional predictions, it directly computes the ROC AUC score; for multiclass problems, it first binarizes the true labels and then computes the ROC AUC using a one-versus-one approach with macro-averaging. Finally, the plot\_roc\_curve method wraps a global plot function (plot\_roc\_curve\_global) to generate ROC curves for both binary and multiclass classification tasks and save the resulting plots when a file path is specified. Overall, the Support class integrates oversampling, cross-validation, hyperparameter tuning, model calibration and detailed performance evaluation into a single modular framework, making it suitable for complex classification tasks in bioinformatics.

**5. Auxiliary method functions:**

**5.1 Visualization of results:**

The plot\_predictions\_scatterplot\_custom function generates a scatterplot that visualizes the main specificity category for each protein based on the prediction results. It is given a dictionary of predictions, an output path to save the plot, and an optional parameter that specifies how many top categories to plot (one by default). The function starts by traversing each protein in the input dictionary and extracting its ranking data. For each protein, it calls the helper function format\_and\_sum\_probabilities to determine the top category of specificity and calculate a normalized confidence value. The auxiliary function format\_and\_sum\_probabilities processes a list of ranking strings associated with a protein. Each ranking string contains a category label and a percentage value that represents a probability. The function extracts these percentages, converts them into decimal probabilities and sums them across several predefined specificity categories such as “C4-C6-C8”, “C6-C8-C10” and so on. Then, these summed probabilities are normalized by dividing the sum of each category by the total probability to ensure that the relative contribution of each category is preserved. The category with the highest normalized probability is identified as the major category for the protein in question, and the function also calculates a confidence value based on the sum of the three highest normalized values. Once the main specificity category for each protein has been determined, plot\_predictions\_scatterplot\_custom orders the proteins by sorting their identifiers and assigning them positions along the Y-axis. The X-axis is set to a scale from C4 to C18. For each protein, the function analyzes the main category string to extract individual specificities (e.g. from “C4-C6-C8” it extracts 4, 6 and 8) and plots each specificity as a dot on the graph at the corresponding X coordinate. If there are multiple specificities for a protein, these dots are connected with lines to link them visually. The diagram is designed with a uniform blue color for the dots, complemented by black borders, and has clearly labeled axes and grid lines to improve readability. After all elements are drawn and the layout has been optimized, the figure is saved in high resolution under the specified output path. The code begins by iterating over the prediction results stored in a dictionary, where each key corresponds to a protein identifier. For each protein, the associated ranking data is retrieved and processed using the format\_and\_sum\_probabilities function, which returns the main specificity category and a normalized confidence value. These values are then formatted into a list containing the identifier of the protein, the identified specificity category and the prediction probability formatted to two decimal places. This list of formatted results is used to create a Pandas DataFrame with three columns: "Query Name", "SS Prediction Specificity" and "Prediction Confidence (Range: 0 - 1)". After creating the DataFrame, the code defines a helper function called highlight\_table to apply custom CSS styling to the table. This styling includes a dark blue background and white text for the header cells, with borders and centered text to improve readability. The table cells are styled with a navy blue background, white text and a uniform font, while even-numbered rows are given a slightly lighter blue background to visually differentiate them. A hover effect has also been added so that the background of a row turns a darker blue when the mouse pointer hovers over it. Once the DataFrame has been styled using the highlight\_table function, it is converted to an HTML table using the to\_html method, resulting in a visually appealing and well-structured display of the prediction data in table format. The plot\_prediction\_confidence\_bar function is used to create an interactive horizontal bar chart that visualizes the prediction confidence for a series of queries. It accepts as input a Pandas DataFrame (df\_results) with at least two columns: "Query name" and "Prediction confidence ( Range: 0 - 1 )", where the latter represents the prediction confidence as a value between 0 and 1. First, the function converts the "Prediction confidence" column into floating point numbers to ensure that numerical operations are exact. Then it defines an internal helper function called get\_confidence\_level that categorizes each confidence value into one of three levels:

“Low Confidence (0 - 0.3)” for low confidence (values ​​below 0.3), “Medium Confidence (0.3 - 0.5)” for medium confidence (values ​​between 0.3 and 0.5), and “High Confidence (0.5 - 1)”. This helper function is applied to the DataFrame and creates a new column called 'confidence\_level' that maps each query to one of these intervals. It then defines a discrete color map that corresponds to the blue background of the application by assigning low confidence a golden yellow, medium confidence an orange, and high confidence a light blue. With Plotly Express, the function creates a horizontal bar chart where the x-axis shows the prediction confidence (from 0 to 1) and the y-axis shows the query names. The bars are colored based on the corresponding confidence level, and the actual numerical value of each prediction confidence is displayed as text on the bars. The layout of the chart has been updated so that the title, axis labels, tick labels and legend text are all white, providing a strong contrast to the dark blue background. Both the background of the chart and the background of the paper are set to this dark blue color for visual consistency. Finally, the chart is rendered in a Streamlit application with st.plotly\_chart with container width enabled so that it can adapt to the available display area. In addition, the plotly chart is converted to a PNG image with the to\_image method and a download button is provided with st.download\_button. You can use this button to download the image of the bar chart directly. In summary, this function not only creates an informative and visually appealing visualization of the forecast reliability, but can also be seamlessly integrated into an interactive web interface.

**6. Other complementary functions**

The plot\_dual\_tsne function creates two interactive 3D t-SNE plots — one for training data and one for prediction data — and saves them as separate HTML files. As input, it accepts the embedding matrices and the corresponding labels for the training and prediction datasets, as well as the protein IDs for additional context and an output directory where the HTML files are saved. Internally, the function starts by importing the TSNE module from the scikit-learn manifold package. It then applies the t-SNE dimensionality reduction separately to the training embeddings and the prediction embeddings, reducing each to three components. The parameters for t-SNE (n\_components=3, random\_state=42, perplexity=30 and n\_iter=1000) are set to obtain consistent and reproducible results. After you have obtained the 3D projections, the function creates color maps for both data sets. For the training dataset, it extracts the unique labels, sorts them and maps each label to a color from the qualitative Dark24 palette of Plotly Express. A similar process is performed for the prediction set using the Light24 palette. These mappings are then used to convert the label lists into corresponding color lists, ensuring that each data point in the scatter plots is visually identifiable according to its label. Next, the function constructs two 3D scatter plots using Plotly’s graph\_objects module. For the training data, a Scatter3d trace is added where the x, y and z coordinates are taken from the t-SNE result, the marker properties (such as size, color and opacity) are defined using the training color mapping, and the hover text is configured to show both the protein ID and the associated label. The layout of the graph is updated to include a descriptive title and axis labels for each t-SNE component. The graph for the prediction data is created in a similar way, using the t-SNE prediction results and the corresponding color mappings and labels. Finally, the function saves each figure as an HTML file in the specified output directory using Plotly IO's pio.write\_html to ensure that the plots can be easily viewed in a web browser. The function logs the file paths where the training and prediction plots were saved and then returns the two figures as a tuple. This dual t-SNE visualization facilitates an intuitive comparison of latent space structures between the training and prediction datasets and highlights potential differences in class separation or clustering. The plot\_dual\_umap function was developed to create two interactive three‐dimensional UMAP plots — one for the training data and one for the prediction data — and to save these visualizations as HTML files. The function accepts as input the high-dimensional embedding matrices (for both training and prediction), the corresponding class labels and protein labels, and an output directory where the resulting HTML files are stored. Internally, the function first applies UMAP for dimensionality reduction. Separate UMAP reducers are instantiated for both the training and prediction embeddings, which are configured to reduce the data to three dimensions (n\_components=3), with additional parameters such as n\_neighbors set to 15, min\_dist set to 0.1 and a fixed random state (42) to ensure reproducibility. The reducers then transform the high-dimensional embeddings into a three-dimensional space that preserves the global and local structure of the data and enables effective visualization of complex relationships. Once the data is projected into the three-dimensional space, the function creates color mappings for the two data sets. It identifies the unique class labels in the training set, sorts them and assigns each label a specific color from Plotly Express’ qualitative Dark24 palette. A similar process is performed for the prediction set using the Light24 palette. These assignments convert the list of labels into corresponding color values that are used to visually distinguish the data points in the scatter plots. Next, the function creates two interactive 3D scatter plots using Plotly’s graph\_objects module. To visualize the training data, a figure is created and a 3D point cloud is added, whose x, y and z coordinates are extracted from the UMAP result of the training embeddings. The marker settings are defined to use a size of 5, the colors determined from the training label mapping and an opacity of 0.8. In addition, the hover text for each label is configured to display the corresponding protein ID and label, providing contextual information directly in the visualization. The layout of the training diagram is updated to include an informative title and axis labels for each of the three UMAP components. The graph for the prediction data is created in a similar way with the UMAP result from the prediction embeddings, its own color mapping and the corresponding protein IDs and labels. After both graphs are created, the function saves them as HTML files in the specified output directory using Plotly IO’s write\_html function. Specifically, one file is saved as "umap\_train\_3d.html" for the training data and another as "umap\_predict\_3d.html" for the prediction data. Logging instructions record the paths under which the HTML files were saved so that the output can be easily traced. Finally, the function returns a tuple containing the two plotly representations, allowing further editing or integration into a larger visualization system if required. This dual UMAP visualization enables a side-by-side comparison of the latent space representations for the training and prediction datasets, providing insights into class separability and data structure that are critical for downstream analyzes.

**7. Main function**

The main function serves as the central coordinator for the entire FAALPred workflow. It manages every stage from data pre-processing and model training to prediction and export— of results - all via an interactive web interface powered by Streamlit. It starts with retrieving the output directory (model) from the input arguments and setting up a progress bar and progress text elements using Streamlit so that the user is constantly aware of the current state of processing. The workflow is divided into several consecutive steps. In the first step, the function loads the training data: it reads the training alignment file (FASTA) and the associated table (TSV). It checks whether the training sequences are already aligned. If not, they are realigned with the MAFFT tool by calling a special function. Once the alignment is confirmed and the table data is loaded with Pandas, the progress is updated. Next, the function initializes the ProteinEmbeddingGenerator class with the training alignment and table data and specifies the selected aggregation method (either “none” for concatenation or “mean” for averaging). The generate\_embeddings method is then called with parameters such as K-Mer size, step size, Word2Vec model path and other hyperparameters. This method extracts k-mers from the sequences, trains (or loads) a Word2Vec model to create vector representations for each k-mer, and generates a standardized embedding for each protein based on a consistent minimum number of k-mers (min\_kmers). The function logs the number of embeddings generated and extracts protein IDs and associated variables from these embeddings. It also fits a default scaler to the training embeddings to normalize the data and saves the scaler for later use. The scaled embeddings are then prepared for training the Random Forest model. The function then defines file paths for saving the trained Random Forest model and its calibrated version. It checks whether a calibrated model already exists. If this is the case, the model is loaded, otherwise the Support class is instantiated to train a new model. This training process involves oversampling the minority classes, performing cross-validation with dynamic adjustment of foldings based on the class distribution, tuning the hyperparameters via a grid search and finally calibrating the model using isotonic regression. Once training is complete, performance metrics such as ROC AUC, F1-Score and Precision-Recall AUC are recorded and the ROC curve is plotted and saved. The calibrated model is then saved to disk and progress is updated. Once the training phase is complete, the function moves on to classifying new sequences (step 2). It first loads the previously saved min\_kmers value to ensure the consistency of the embedding process and then reads the FASTA file containing the predictions. Like the training data, the prediction sequences are also checked for alignment and realigned if necessary. The ProteinEmbeddingGenerator is reinitialized — this time without table data — to generate embeddings for the prediction sequences with the same parameters as during training. These prediction embeddings are scaled using the previously saved default scaler and the calibrated model is applied to predict the associated variable for each new sequence. In addition, class rankings (i.e. the sorted probabilities for each class) are calculated for each prediction using a global utility function. The predictions are stored and logged in a file together with the corresponding rankings. A scatter plot to visualize the prediction results is created and saved; this plot is then displayed in the Streamlit app with st.image. The feature also formats the forecast results in a styled DataFrame— - with custom CSS for a dark blue background and white text — which is displayed as HTML in the main area. To facilitate further analysis, several download buttons are available: one for exporting the results in CSV format, another for Excel and another for downloading all output files (including models, charts and logs) as a ZIP archive. An interactive horizontal bar chart showing the reliability of the forecast is also created with Plotly Express, rendered directly in the app and made available for download as a PNG image. Optionally (step 3), the function can generate dual UMAP or t-SNE visualizations to compare the latent spaces of the training and prediction datasets, although these steps are currently commented out in the code. After processing is complete, the function updates the progress to 100% and displays a success message. Custom CSS styling throughout the app provides a consistent look and feel, and a footer with support logos and copyright information is displayed at the bottom of the page. In summary, the main feature carefully controls data loading, alignment checking, embedding generation, model training and calibration, prediction of new sequences, and comprehensive visualization and export— of results - providing a complete, interactive bioinformatics pipeline via Streamlit.

**8. Streamlit Application Integration and Results Export**

The Streamlit application provides a fully interactive web interface that guides the user through the entire FAALPred workflow, from data entry to visualization and export of results. When theapplication launches, it is configured with a wide layout, a custom page title ("FAAL\_Pred") and a DNA-inspired icon, while a custom CSS style sets a dark blue background for both the main application and the sidebar, ensuring that all text appears in white for a consistent and professional look. In the sidebar, the user is offered several input options: They can choose to use the default workout data or upload their own FASTA and TSV files for the workout; they must also upload a FASTA file for the prediction. You can also set parameters such as k-mer size and stride size using numerical inputs. There is also a dropdown menu where you can select the aggregation method for generating embeddings (“none” for concatenation or “mean” for averaging). There is even an optional Word2Vec parameter customization section that gives the user control over the window size, number of workers, and number of epochs if desired. Once the parameters are set, the output directory is dynamically created based on the chosen aggregation method. The user then clicks on the “Run analysis” button. The application then decides whether to use the internal default data or the files uploaded by the user and saves the uploaded files to the specified output folder, if applicable. When the main analysis process begins, the application displays a progress bar and text updates in the sidebar to inform the user of the current progress of the pipeline execution. The main function is called with all required parameters encapsulated in an argparse.namespace object. During its execution, the application performs several important operations: It validates the input sequences and realigns them if necessary; it generates protein embeddings using the ProteinEmbeddingGenerator class, which uses k-mer extraction and trains (or loads) a Word2Vec model; it scales the embeddings and uses them to train a Random Forest classifier with oversampling and hyperparameter tuning via cross-validation; and it calibrates the classifier and performs predictions on the new sequences. After the analysis is completed, a success message is displayed on the screen. The results generated during the analysis are comprehensive. The application automatically saves several files in the output directory: Alignment files (if realignment was required), the trained Word2Vec model, the StandardScaler, the trained and calibrated Random Forest models, and various graphs. For example, learning curves, ROC curves and dual visualizations with UMAP or t-SNE are saved as HTML or image files. The application then displays the scatter plot of the prediction results within the user interface with st.image if the file is found in the output folder. In addition, the results are formatted in a clear table showing the identifier of each protein, its predicted specificity category and the associated prediction probability. This table is styled with custom CSS (so that headers and cells have a dark blue background and white text) and rendered as HTML on the main page. To facilitate further analysis, the application offers several download options: There are download buttons for exporting the formatted results as CSV and Excel files, and a ZIP file containing all generated result files is also offered for download via a dedicated button. An interactive horizontal bar chart visualizing the prediction probability for each query is rendered with Plotly Express and is embedded in the app with st.plotly\_chart. This chart is also designed to match the dark blue theme and includes a download button that allows you to save it as a PNG image. Finally, the application concludes with a custom footer displayed at the bottom of the page. This footer is implemented with HTML and CSS embedded via st.markdown and displays a series of logos (encoded in base64 and resized accordingly) as well as a supporting text and copyright notice. Throughout the process, each generated file— - whether models, diagrams or results tables — is saved in the specified output directory. This ensures that the entire workflow is not only interactive, but also generates a comprehensive set of files that the user can later download and review.

**9- References:**

**1. Python Packages and Libraries**

1. Harris CR, Millman KJ, van der Walt SJ, et al. Array programming with NumPy. Nature. 2020;585(7825):357–362. doi:10.1038/s41586-020-2649-2.
2. McKinney W. Data structures for statistical computing in Python. In: Proceedings of the 9th Python in Science Conference; 2010:51–56. doi:10.25080/Majora-92bf1922-00a.
3. Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine Learning in Python. J Mach Learn Res. 2011;12:2825–2830.
4. Lemaître G, Nogueira F, Aridas CK. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. J Mach Learn Res. 2017;18(1):559–563. Available from: <http://imbalanced-learn.org>.
5. Joblib. Joblib: Tools to provide lightweight pipelining in Python. Available from: <https://joblib.readthedocs.io/en/latest/>.
6. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781. 2013. Available from: <https://arxiv.org/abs/1301.3781>.
7. Cock PJ, Antao T, Chang JT, et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. Bioinformatics. 2009;25(11):1422–1423. doi:10.1093/bioinformatics/btp163.
8. Plotly Technologies Inc. Collaborative data science. 2015. Available from: <https://plot.ly>.
9. Hunter JD. Matplotlib: A 2D Graphics Environment. Comput Sci Eng. 2007;9(3):90–95. doi:10.1109/MCSE.2007.55.
10. McInnes L, Healy J, Melville J. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. arXiv preprint arXiv:1802.03426. 2018. Available from: <https://arxiv.org/abs/1802.03426>.
11. Tabulate. Tabulate: Pretty-print tabular data in Python. Available from: <https://pypi.org/project/tabulate/>.
12. Streamlit Inc. Streamlit—Turn Python scripts into shareable web apps in minutes. 2020. Available from: <https://streamlit.io>.
13. Clark A, Pillow Team. Pillow (PIL Fork). Available from: <https://python-pillow.org/> (accessed 2025).
14. Python Software Foundation. Python Language Reference, version 3.9. Available from: <https://www.python.org> (accessed 2025).  
      (Standard libraries including base64, os, sys, subprocess, time, argparse, logging, zipfile, and shutil are part of the Python Standard Library.)
15. Breiman L. Random forests. Mach Learn. 2001;45:5–32. doi:10.1023/A:1010933404324.
16. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. SMOTE: Synthetic Minority Over-sampling Technique. J Artif Intell Res. 2002;16:321–357. doi:10.1613/jair.953.

**2. External Programs and Tools**

1. Katoh K, Standley DM. MAFFT multiple sequence alignment software version 7: improvements in performance and usability. Mol Biol Evol. 2013;30(4):772–780. doi:10.1093/molbev/mst010.

**3. Related Scientific Articles and Methodology References**

1. Yang KK, Wu Z, Bedbrook CN, Arnold FH. Learned protein embeddings for machine learning. Bioinformatics. 2018 Mar 23;34(15):2642–2648. doi:10.1093/bioinformatics/bty178. Correction in: Bioinformatics. 2018 Dec 1;34(23):4138. doi:10.1093/bioinformatics/bty455.