University of South Bohemia

Faculty of Science

České Budějovice, Czech Republic

and

Johannes Kepler University

Faculty of Engineering and Natural Sciences

Linz, Austria

Protein solubility prediction using CNN and CNN-LSTM hybrid models

Bachelor Thesis

Author: Ekaterina Sysoykova

Supervisor: Assist.-Prof. Mag. Dr. Günter Klambauer

Philipp Seidl, MSc

Guarantor: Ing. Ph.D. Rudolf Vohnout

České Budějovice, 2021

Sysoykova E. (2022): Protein solubility prediction using CNN and CNN-LSTM hybrid models. Bc. Thesis, in English. – 77 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic and Faculty of Engineering and Natural Sciences, Johannes Kepler University, Linz, Austria.

Annotation

Protein solubility prediction based on the raw amino acid sequence using three machine/deep learning techniques: RF, CNN, and a hybrid CNN-biLSTM model. Assessment of the performance difference between the models with evaluation metrics and statistical significance tests.

Acknowledgement:

I would like to express my deep and sincere gratitude to my supervisors for their guidance, advice and moral support. The completion of this thesis couldn’t have been possible without them.

I want to extend my appreciation to my family and loved ones for their care, support and encouragement. I am very grateful they have always been there for me despite all the distances. I would like to dedicate this thesis to them.

Table of Contents

Abstract

Protein solubility is an important physicochemical property that plays an important role in scientific research and industry. Therefore, it would be useful to have a computational model to predict protein solubility from the amino acid sequence since the *in-silico* approach would be less time-consuming and costly than in vitro experiment. Over the last three decades, many sequence-based methods have been developed using different computational algorithms. In this study, we implemented three machine/deep learning models for protein solubility prediction solely based on the raw protein sequence. We implemented a random forest classifier as the baseline model, a CNN, and a hybrid model based on the combination of CNN and biLSTM. The developed models have been trained on more than 65 000 proteins. The proposed algorithms make predictions based only on the raw protein sequence without using any additional features. The CNN and hybrid biLSTM models have reached more than 65% balanced accuracy score on the independent test set. Moreover, statistical significance tests have demonstrated no significant difference between the performances of the proposed deep learning models. Furthermore, the proposed architectures could be extended to other protein prediction tasks based on the raw protein sequence.

1. Introduction

1.1. Protein solubility: definition and application

Protein solubility is a thermodynamic parameter defined as the protein concentration in a saturated solution that is in equilibrium with a solid phase under a defined set of conditions (Kramer et al., 2012). Knowledge of protein solubility can give useful information on the potential application of proteins in many biophysical, structural and pharmaceutical studies, as well as biopharmaceutical and biotechnological processes (Hou et al., 2019).

Structure determination using, for instance, nuclear magnetic resonance (NMR) spectroscopy, which is regularly applied to study proteins' structure, functions and intermolecular interactions, requires samples that are stable and soluble for an extensive amount of time at a higher concentration (Bagby et al., 2001; Maxwell et al., 2003). Moreover, protein crystals are used to determine macromolecular structures using techniques such as X-ray crystallography or diffraction. In order to form crystals, proteins are dissolved in the solution until they reach a supersaturated state (McPherson & Gavira, 2013). Besides, crystalline proteins have been used as drug formulations in pharmaceutical studies and industry due to better handling, stability, and dissolution characteristics (Jen & Merkle, 2001). Furthermore, soluble proteins are required for structural genomics (Pédelacq et al., 2002). Moreover, many biochemical experiments rely on the ability of a protein to dissolve in an aqueous solution, such as protein purification and expression, quantitative binding assays (Tjong & Zhou, 2008).

Soluble proteins play an important role in the drug discovery field. They are used, for instance, for the design of aggregation-resistant peptides (Fowler et al., 2005) and the development of methods to optimise drug efficiency and stability (Kalayan et al., 2019; Solá & Griebenow, 2010). Moreover, protein solubility is necessary for producing recombinant proteins, and it is considered the best criterion of recombinant protein quality (Mona et al., 2010). Recombinant proteins have a wide range of applications in pharmacology and medicine. They are used to produce therapeutic recombinant hormones, antibodies, interferons, interleukins, tumour necrosis factors, thrombolytic drugs, and treat diseases such as diabetes, myocardial infarction, neutropenia, and thrombocytopenia, anaemia, hepatitis, Crohn's disease, cancers therapies, and many others (Pham, 2018; Smialowski et al., 2012).

Protein solubility is an essential prerequisite for studies about protein aggregates and inclusion bodies to understand the aggregation mechanism and find methods to prevent proteins from it.

These studies are significant because protein aggregation is a significant obstacle to many processes. For instance, it has been estimated that approximately 33−50% of all expressed non-membrane proteins are insoluble, and about 25 to 57% of remaining soluble proteins tend to aggregate or precipitate at higher concentrations (Fang & Fang, 2013). Moreover, developed recombinant proteins tend to aggregate at the condition they are stored (Hou et al., 2019). Samples for structural studies, for instance, analysis with NMR, have a strong propensity to aggregate. Besides, several neurodegenerative diseases such as Alzheimer's, Parkinson's or Huntington's disease, diabetes type II and genetic disorders such as cystic fibrosis and Marfan syndrome are associated with the formation and accumulation of insoluble protein aggregates that disrupt normal cell functioning (Dobson, 2001; Ross & Poirier, 2004).

Although there are several methods to recover the biological activity of formed inclusion bodies, such as solubilisation of inclusion body followed by refolding and purification (Singh & Panda, 2005), or use of weak promoters, gene fusion, low temperatures, and growth additives such as sorbitol and ethanol (Georgiou & Valax, 1996), these approaches either have a low success rate or might be expensive (Singh & Panda, 2005). In order to focus work on potentially soluble proteins, one can use amino acid sequence to estimate whether the protein would be soluble or not using different methods.

1.2. The connection between protein's amino acid sequence and its solubility

Protein solubility can be influenced by several factors, such as ionic strength, pH, temperature, various solvent additives. Although altering these factors might enhance the protein's solubility, it is frequently not sufficient to ensure it for an extensive amount of time (Kramer et al., 2012). Under a given set of exact experimental conditions, such as the expression host, temperature, pH, the main factor determining protein solubility is its amino acid sequence (Smialowski et al., 2012).

The correlation between the protein's primary structure and its solubility has been studied for about 30 years. One of the earliest studies by Wilkinson and Harrison (1991) discovered a correlation between six protein sequences features - approximate charge average, cysteine fraction, proline fraction, hydrophilicity index, the total number of residues, and turn-forming residue fraction – and solubility. Later, the proposed model was modified by Davis et al. (1999), who discovered that only two out of six factors influenced the solubility of overexpressed proteins in E. coli. These factors are approximate average charge, determined by the relative numbers of Asp, Glu, Lys, Arg residues, and the content of turn-forming residues, Asn, Gly, Pro and Ser.

Further, Bertone et al. (2001) confirmed that high content of negative-charged residues and absence of hydrophobic patches are associated with improved solubility. Additionally, they found that a low percentage of Asp and Glu residues increases the probability of a protein being insoluble.

The study by Idicula-Thomas (2005) has shown the influence of several factors on solubility. Aliphatic index - mole fraction of Ala, Val, Leu and Ile amino acids in the protein (Ikai 1980) – has a positive correlation with the solubility of the protein, while instability index, a measure of protein half-life *in vivo*, has a negative correlation with protein solubility and longer-lived proteins tend to form aggregates. Moreover, proteins with higher frequencies of Asp, Trp, and Tyr residues tend to form inclusion bodies. Besides, the study has shown that dipeptide and tripeptide composition tend to correlate with higher protein solubility.

The study by Niwa et al. (2009) has demonstrated that proteins with higher solubility tend to have a higher content of negatively charged residues and hence, higher values of Lys/Arg- and Glu/Asp- ratios. Moreover, it has shown that proteins sequences with low aromatic residues content tend to be more soluble. Based on all of the above, we can conclude that the primary structure of a protein is directly related to its solubility, and we can use amino acid sequence in experiments in order to determine it.

1.3. Overview of *in silico* sequence-based methods for protein prediction.

Predicting protein's solubility can be done via *in vitro* experiments; however, it can be time-consuming and require costly reagents and equipment. Therefore, a computational model for predicting solubility can be beneficial to assess proteins solubility due to its rapidity and cost-effectiveness.

Existing protein solubility predictors can be grouped into two classes: sequenced-based and structure-based. As the names suggest, structure-based predictors make their decisions based on properties, such as secondary structure compositions (e.g., alpha helix, beta sheets), statistical potentials and different other properties (Hou et al., 2019). On the other hand, sequence-based predictors make predictions based on the amino acid sequence of the protein and the features that can be extracted from it, e.g., sequence length, isoelectric point, molecular weight (Fang & Fang, 2013).

Over the past three decades, several sequence-based methods have been developed based on different approaches. The first method to calculate solubility from the amino acid sequence was proposed by Wilkinson and Harrison (1991), which implemented classification with a standard Gaussian distribution, which was later improved by David et al. (1999). Later several studies such as the study by Idicula-Thomas et al. (2005), *PROSO* (Smialowski et al., 2006), *SOLpro* (Magnan et al., 2009), *CCSOL* (Agostini et al. 2012) employed support vector machine (SVM) (Cortes & Vapnik, 1995) for predicting protein's solubility. A model by Diaz et al. (2010) used logistic regression, *PROSO II* implemented a logistic function and an adapted Parzen window algorithm (Smialowski et al., 2012). A study by Huang et al. (2012) used the scoring card method (SCM). Research by Fang & Fang (2013) implemented a random forest (RF) algorithm to identify essential features for predicting protein solubility, and then the second RF is used to make the classification. Computational approach *ESPRESSO* employs two predicting methods: one used the sequence and structural features, and the other used sequence patterns' occurrence frequencies (Hirose & Noguchi, 2013). *PaRSnIP* classifier uses a gradient boosting machine (GBM) algorithm together with features extracted from the sequence and structural properties of the protein (Rawi et al., 2017). The same algorithm is used in *SoluProt* to generate the predictive model (Hon et al., 2021). *DeepSol* model implemented three convolutional neural networks (CNN); one makes classification based on features extracted from the raw protein sequences, while two others use additional sequence and structural features (Khurana et al., 2018). *ProGAN* employs a deep neural network (DNN) jointly with generative adversarial networks (GAN) algorithm that generates extra data to improve the performance of the neural network (Han et al., 2019). The study from Bhandari et al. (2020) proposed 'Solubility-Weighted Index', which derives 20 values for the standard amino acid residues and uses them to predict solubility. Research from Chen et al. (2021) proposed a structure-aware method *GraphSol* that predicts protein solubility using predicted contact maps and graph neural networks (GNN).

1.4. Our approach

In this study, we implemented three machine/deep learning models for protein solubility prediction. We employed a random forest (RF) classifier as the baseline model, convolutional neural network (CNN) and hybrid model based on CNN and bidirectional long-short term memory (biLSTM) models. In the hybrid model, CNN was used for feature extraction, while biLSTM supported sequence predictions. All proposed models make their predictions only based on the proteins sequence without using any additional features. We compared the performances of all three models using evaluation metrics and significance tests. Moreover, we determined whether adding LSTM to the CNN model would increase the model's performance.

2. Methodology

2.1. Dataset

We used a training dataset initially collected in a study by Smialowski et al. (2012), which underwent two pre-processing steps in research by Rawi et al. (2017). For pre-processing, they used a CD-HIT program that clusters proteins that meet a similarity threshold set by the user (Fu et al., 2012; Li and Godzik, 2006). In the first step, proteins with a maximum sequence identity of more than 90% were removed from the dataset to ensure heterogeneity within the dataset and decrease sequence redundancy. Afterwards, to reduce bias caused by homologous sequences, they excluded all sequences with an identity of 30% or greater to any protein in the independent test set collected by Chang et al. (2014).

Therefore, the final dataset that we used in our study contains 28 972 soluble (≈ 42%) and 40 448 insoluble (≈ 58%) proteins and their respective labels, where soluble proteins were labelled with one and insoluble with zero. We split the dataset into the training set and the validation set, where 90% of sequences were used for training and 10% for validation. An independent test set consisting of 1000 soluble and 999 insoluble proteins was used for evaluating the performance of the proposed models.

2.2. Data pre-processing

We have compiled a dictionary of amino acids, in which each amino acid was assigned a unique number from 1 to 20 inclusive. Further, we used each amino acid in every sequence in the dataset and replaced it with the respectful number from the dictionary. Afterwards, all sequences in the training set were padded with 0 values to the same length $L = 1697$, which is the length of the longest protein in the dataset.

2.3. Random forest

We chose a random forest (RF) classifier as a baseline model for our problem (Breiman, 2001). The RF algorithm is a supervised ensemble machine learning method that performs classification or regression using multiple independent decision trees *(estimators*). Each tree in the RF produces class prediction, and the class that was predicted most often becomes our model's prediction. The implemented RF classifier consists of 701 estimators. We implemented the RF classifier using *the scikit-learn* machine learning library.

2.4. Convolutional neural network

In this section, we describe the architecture of the proposed CNN model. Convolutional neural network (CNN) is a common deep-learning technique applied in a wide range of fields such as computer vision image classification, speech recognition, computer vision and bioinformatics (Fukushima, 1980; Lecun et al., 1998). The architecture of CNNs contains several distinct layers: embedding layer, convolutional layers, non-linearities, pooling layers and fully connected layers. The typical CNN architecture is illustrated in Figure 1.

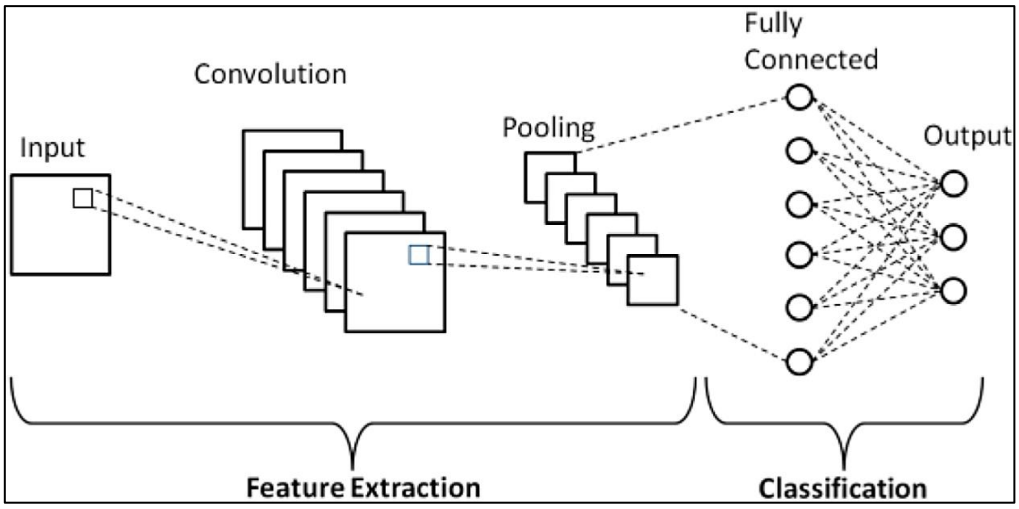


Figure 1. Convolutional Neural Network Diagram (Phung & Rhee, 2019)

In the proposed model, the pre-processed amino acid sequence is passed through the embedding layer and converted into continuous vector space. The embedding matrix is in parallel convolved with one-dimensional (1D) convolution blocks. Each convolution block has the same number of input and output channels and only differ in the size of the kernels. In our model, we used seven convolution blocks with the window sizes range $\{2,3,\dots,8\}$. Furthermore, since the convolution is a linear operation, we introduce non-linearity by applying the activation function on every feature map. We used rectified linear activation unit (ReLU) as activation function (Nair & Hinton, 2010). As we can observe from Figure 2. ReLU returns the input directly if it is positive; otherwise, it returns zero.

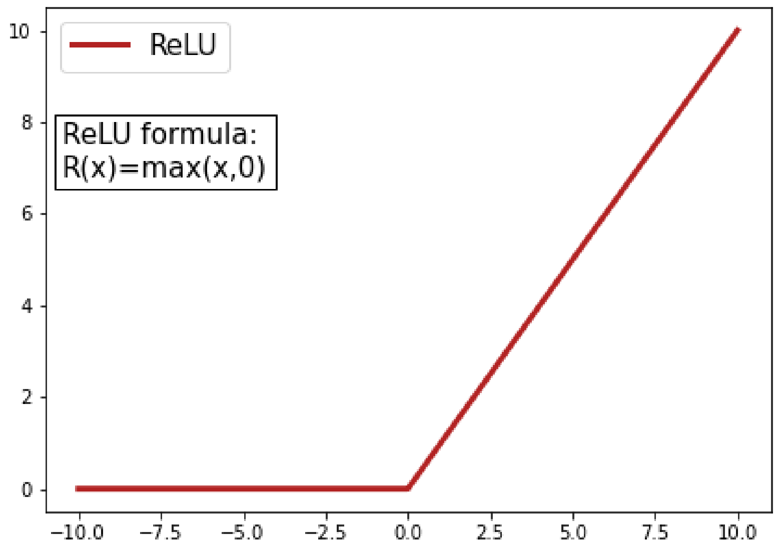


Figure 2. Rectified linear activation unit (ReLU) graph and formula

After we obtain seven convolved rectified feature maps, we perform a pooling operation to decrease the dimensionality of obtained mapping in order to reduce computational cost. In order to keep the strongest features, we used the max pooling operation, which takes the largest element from the rectified feature maps and discards all the other values. Each pooling operation is applied separately to each obtained feature map.

Further, the outputs of each pooling operation are concatenated into one vector and flattened into a one-dimensional array. The array is loaded into a fully connected layer, in which every input is connected to every output of the neural network by learnable weights. The proposed CNN model has two fully connected layers that apply linear transformations to the flattened array through a weights' matrix. Moreover, we perform a non-linear transformation to the output of the first fully connected layer through ReLU. The output of fully connected layers is the probabilities for each class in classification tasks. The output is represented in a one-hot encoding format. Typically, for a n-class problem, the output is a vector is of the dimension n; therefore, for our problem $n = 2$.

2.5. Long-short term memory

The LSTM is a specialised form of recurrent neural network (RNN) that is able to learn long-term dependencies in sequential data (Hochreiter & Schmidhuber, 1997). The LSTM architecture commonly includes three gates: input, forget, and output gates and the memory cell. The architecture of the LSTM model is illustrated in Figure 3.

Forget gate takes two inputs: input at a given time step t X_t and a previous memory state h_{t-1}. Two given inputs are multiplied with weight matrices, followed by the addition of bias. Further, the sigmoid activation function is applied to the result. Sigmoid generates values between 0 and 1 corresponding to each value in cell state. Based on the output value, the forget gate establishes whether the given piece of information should be removed. If for a particular value in cell state the output is closer to 0, the forget gate erases that piece of information; otherwise, if the output is closer to 1, the information is kept.

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_t - 1 + b_{hf})$$

Input gate takes two inputs: input at a given time step t X_t (current hidden state) and a previous memory state h_{t-1}. Simultaneously two inputs are passed through the sigmoid function that generates values between 0 and 1 and through hyperbolic tangent activation function (tanh) that outputs the number between -1 and 1 for each value in cell state C_{t-1}. Outputs of the tanh and the sigmoid are multiplied in order to determine which information is essential to keep.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_t - 1 + b_{hi})$$

$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_t - 1 + b_{hg})$$

Cell state updates old cell state C_{t-1} into new cell state C_t. Previous cell state C_{t-1} values are pointwise multiplied by forget vector f_t. If, for a particular value, the output is 0, the value gets dropped from the memory. Further, point-by-point addition is performed with the output vector of the input gate, and the cell is updated into a new cell state C_t.

$$c_t = f_t * c_{t-1} + i_t * g_t)$$

In the *output gate*, sigmoid is applied on values of the current state and previous hidden state and the tanh function if applied on new cell state C_t. The output of sigmoid and output of than are multiplied point-by-point, and the network determines the value of the next hidden state. Then the new cell state C_t and new hidden state h_t are passed to the next time step.

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_t - 1 + b_{ho})$$

$$h_t = o_t * tanh(c_{t-1} + i_t * g_t)$$

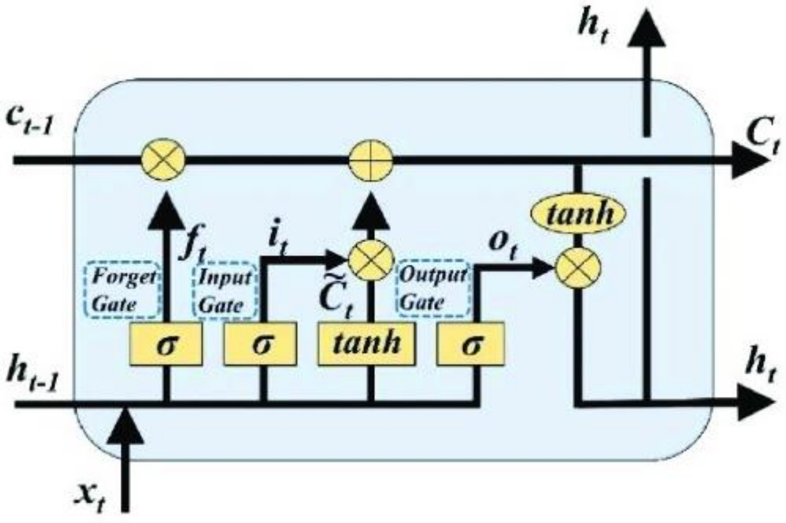


Figure 3. Long Short-Term Memory Diagram (Jiang et al., 2020)

2.6. Bidirectional LSTM

Bidirectional long-short term memory (biLSTM) consists of two LSTM models: the first model learns the input sequence in the forward direction, while the second model learns the sequence in the backward direction. The BiLSTM preserves information from the past and the future, increasing the amount of available information and providing additional context to the neural network (Schuster & Paliwal, 1997).

2.7. Hybrid BiLSTM

CNN-LSTM models, also known as the long-term recurrent convolutional network (LRCN) models, were explicitly designed for activity recognition, image captioning and video description involving sequential inputs and outputs (Donahue et al., 2017). The CNN-LSTM model combines CNN layers to extract the features from input data and LSTM layers to classify the input data.

In the standard CNN architecture, we described before, the outputs of the pooling layer would be concatenated, flattened and sent to a fully connected layer. In the implemented BiLSTM, we pass the concatenated feature maps into the biLSTM. Hybrid biLSTM outputs three tensors: tensor of the output features from the last layer of the biLSTM, tensor containing the final hidden state for each element in the batch and tensor containing the final cell state for each element in the batch. Further, the tensor with the final hidden state is passed through fully connected layers, producing the output.

2.8. CNN and hybrid biLSTM training

Training of CNN and BiLSTM models was done using the PyTorch framework. As a loss function, we applied binary cross-entropy with logits loss that combines a sigmoid layer and the binary cross-entropy loss in one single class. Moreover, we used Adam optimiser (Kingma & Ba, 2014) for optimising model training by changing weights and learning rate to reduce the loss. Both models were trained with a maximum of 100 epochs with an early stopping technique: if the validation loss does not decrease for five consultative epochs, the training is stopped. Such a technique is used in order to prevent overfitting.

2.9. Hyperparameter tuning

For hyperparameter tuning, we used Optuna, a hyperparameter optimisation framework to automate hyperparameter search (Akiba et al., 2019). We chose this framework because it is intuitive, simple to integrate into python code, and has a variety of sampling and pruning algorithms. We used the Tree-structured Parzen Estimator (TPE) algorithm as an optimisation algorithm (Bergstra et al., 2011). On each trial, for each parameter, TPE fits one Gaussian Mixture Model (GMM) $l(x)$ to the set of parameter values associated with the best objective values, and another GMM $g(x)$ to the remaining parameter values. It chooses the parameter value x that maximises the ratio $l(x)/g(x)$. We ran a hyperparameter tuning algorithm for 25 trials. The tested hyperparameters, ranges and the optimal hyperparameter values are presented in Appendix A.

3. Results & Discussion:

3.1. Performance evaluation:

The prediction performance of RF, CNN and hybrid biLSTM were obtained using the independent test set. For both CNN and BiLSTM models, the threshold value was set to 0.4, while for the baseline model, we used the default threshold value.

We evaluated the model performance with given metrics:

- Accuracy – the number of correctly predicted data points out of all the data points.
- Balanced accuracy – the average of recall obtained on each class.
- Matthew's correlation coefficient (MCC) is a measure of the classification quality, which considers true and false positives and negatives. MCC formula:

  $$MCC = \frac{tp*tn+fp*fn}{(tp+fp)*(tp+fn)*(tn+fp)*(tn+fn)},$$

  where tp – number of true positives, tn - number of true negatives,
  fp - number of false positives, fn - number of false negatives
- Average precision (AP) – an average precision at all possible thresholds. Average precision formula:

  $$AP = \sum_n (R_n - R_{n-1}) * P_n,$$

  where P_n and R_n are the precision and recall at the n-th threshold.
- Recall – number of total relevant results correctly classified by your algorithm. Recall's formula:

  $$\frac{tp}{tp+fn}$$
- Precision – number of results that are relevant. Precision's formula :

  $$\frac{tp}{tp+fp}$$
- F1 score - harmonic mean of the precision and recall. F1 formula:

  $$F1 = \frac{2*precision*recall}{precision+recall}$$
- ROC AUC score – area under the ROC curve

Moreover, we plotted a precision-recall curve, ROC curve and confusion matrix to illustrate the performance of implemented algorithms.

The performance is presented in Table 1. and on the built graphs (see appendix B, C, D). As we can observe, both CNN and BiLSTM models have better performance than our baseline model and have higher values for each metric, except for precision. The precision of RF has the value of 0.66, which is higher than the BiLSTM precision value of 0.66 but lower than CNN's that has a precision value of 0.7. Moreover, CNN and BiLSTM have an equal AP value of 0.62 compared to RF's AP value of 0.58.

The BiLSTM model has the highest value of 0.68 for accuracy and balanced accuracy. We can observe that accuracy and balanced accuracy scores are equal in CNN and RF models and have values of 0.67 and 0.61, respectfully.

Table 1 demonstrates that the BiLSTM model has the highest recall, F1 and MCC scores of 0.72, 0.69 and 0.36, respectfully. CNN has a value of 0.6 for recall, 0.65 for F1 score and 0.35 for MCC. The RF has the lowest values of 0.42, 0.52 and 0.25 for these metrics, respectfully.

The CNN model has the highest value of MCC that is 0.39, while BiLSTM and RF have respectively 0.37 and 0.2 values for this metric. The CNN model has the biggest ROC AUC score of 0.76, while CNN and RF have 0.75 and 0.7 scores, respectively.

Metric / Model	ROC AUC score	accuracy	balanced accuracy	MCC	recall	F1 score	precision	AP
RF	0.7	0.61	0.61	0.25	0.42	0.52	0.68	0.58
CNN	0.76	0.67	0.67	0.35	0.6	0.65	0.7	0.62
BiLSTM	0.75	0.68	0.68	0.36	0.72	0.69	0.66	0.62
DeepSol1	NaN	0.73	NaN	0.46	NaN	NaN	NaN	NaN

Table 1. Evaluation metrics' score for RF, CNN and biLSTM. Metrics' score of DeepSol1 (Khurama et al., 2018), if NaN - metric was not specified in the paper.

We compared our approaches to the DeepSol1 approach (Khurama et al., 2018) because it was the only approach evaluated on the same test set that, like our models, used raw amino acid sequences without any additional features. As Table 1. illustrates, the proposed CNN and CNN-biLSTM model has slightly lower performance than the DeepSol1 model. We implemented different CNN architecture by using two times fewer kernels of smaller sizes than DeepSol1 to observe the effect of such change on the model's performance. Moreover, even though we used the same dataset, the longest protein in our study consisted of 1697 amino acids, while in DeepSol1, it had only 1200 amino acids. Such difference might have been caused by different data pre-processing and might affect the model's performance.

3.2. Statistical tests

3.2.1. 10-fold cross-validation

We used the 10-fold cross-validation method to estimate the skill of the constructed models. The training dataset was split into ten sections, where each section is used as a test set at a particular iteration. We performed 10-fold cross-validation for each of the three models and sampled the balanced accuracy score from each of ten iterations.

3.2.2. Model variance

Variance refers to the variability of model prediction depending on the given data. We calculate the variance for each model to determine whether the results are reliable or were produced by chance.

Using the obtained balanced accuracy scores distributions, we calculated the variance, the mean and the standard deviation for each model. As we can observe from Table 2. and Figure 4, the RF model's variance is the lowest and equals 0.209, the CNN has a variance of 0.377, and BiLSTM has the highest variance of 0.696.

Model / Metric	RF	CNN	BiLSTM
mean	63.072	66.937	66.624
variance	0.209	0.377	0.696
standard deviation	0.457	0.614	0.834

Table 2. Mean, variance and standard deviation of RF, CNN and biLSTM.

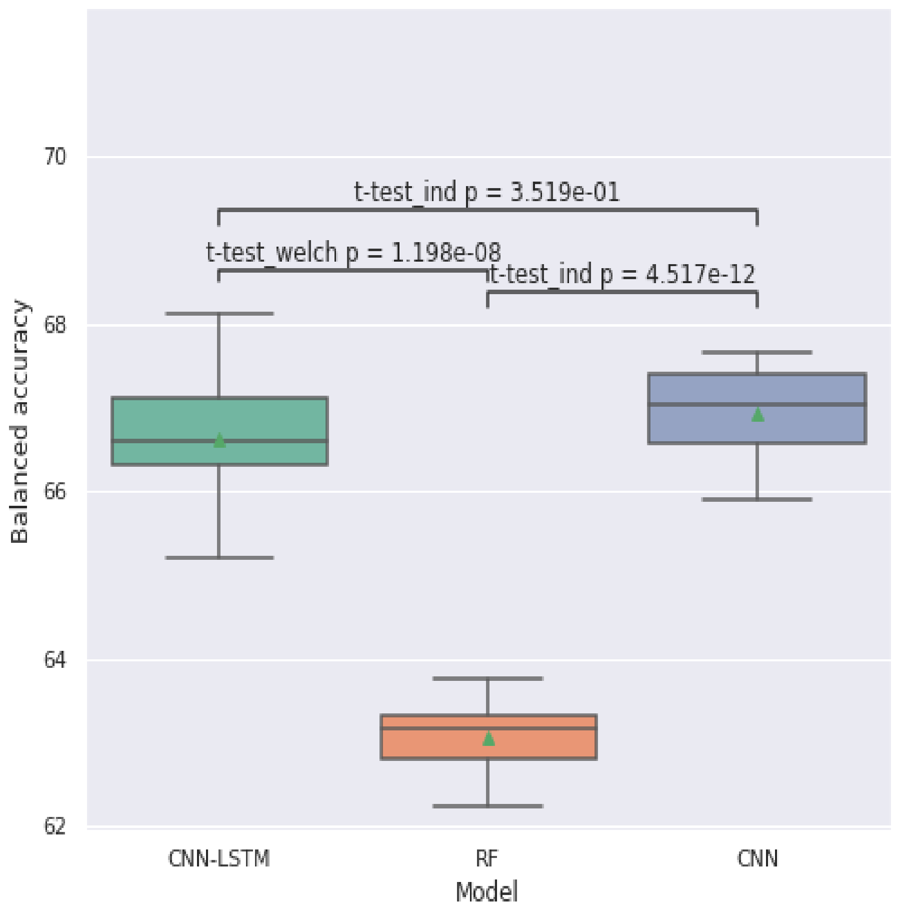


Figure 4. The variance of RF, CNN, biLSTM. Statistical significance tests' p-values for each pair of models.

3.2.3. Statistical significance test

We performed the Shapiro–Wilk normality test on each accuracy score distribution. The tests have demonstrated that each sample is normally distributed. Then we performed F-tests and determined whether the variances between each pair of models were equal. Afterwards, we applied independent t-tests. If the variances between the two models were equal, we performed an independent two-sample t-test; otherwise, we performed Welch's test.

	F-test			T-test		
H_0	Variances of the two groups are equal			The means of the two groups are equal		
H_A	Variances of two groups are not equal			The means of two groups are not equal		
	f-value	p-value	conclusion	t-score	p-value	conclusion
CNN/RF	1.802	0.197	p-value>0.05, H_0 is not rejected, variances are equal. We apply a two-sample t-test.	15.966	$5 * 10^{-12}$	p-value<0.05 H_0 is rejected, means of the two groups are not equal, the result is significant.
BiLSTM /RF	3.332	0.043	p-value<0.05 H_0 is rejected, variances are not equal. We apply Welch's test.	11.802	$1 * 10^{-8}$	p-value<0.05 H_0 is rejected, means of the two groups are not equal, the result is significant.
BiLSTM /CNN	1.848	0.187	p-value>0.05, H_0 is not rejected, variances are equal. We apply a two-sample t-test.	-0.956	0.352	p-value>0.05 H_0 is not rejected, means of the two groups are equal; the result is not significant.

Table 3. Results of F-tests and t-test for each pair of models.

As we can observe from Table 3. variances of CNN and RF are equal as well as variances of CNN and BiLSTM; therefore, for these pairs, an independent two-sample t-test was applied. However, the variances of BiLSTM and RF are not equal; therefore, we have to apply Welch's test.

Table 3. demonstrate that the result of the t-test performed on CNN and BiLSTM is not significant. However, the results of the t-test performed on RF and CNN and of the t-test performed on RF and BiLSTM are significant.

The results indicate that the hybrid model does not perform significantly better than the pure CNN model on the given problem. Therefore, adding biLSTM to the CNN does not improve the proposed model's performance for a protein solubility prediction task.

3.3. Limitations & future work

This research is subject to computational limitations. First of all, due to the computational limitations, we were not able to employ more complex model's architectures. Moreover, it restricted the number of hyperparameters and hyperparameter values we could test.

We have used half of the kernels that have been used in the DeepSol1 approach. However, to our best knowledge, Khurama et al. (2018) did not evaluate how the tested kernels configurations were chosen and only tested three kernels' combinations for proposed models. Therefore, we cannot confirm whether the combination of the kernel they used is optimal, and further research is needed to determine the best configuration for a given problem. Moreover, different amino acid encodings can be used instead of label encoding. The study from ElAbd et al. (2020) has demonstrated that, for instance, BLOSUM62 or VHSE8 encoding can improve a model's performance.

As stated earlier, we determined that a hybrid model does not have better performance than the pure CNN model. However, these conclusions can only be made for a given models' configurations. Many variations of the proposed model architecture can be explored to determine whether the hybrid model is more optimal than CNN models for protein's solubility prediction task.

4. Conclusion

Protein solubility prediction is an important physicochemical property used in a wide range of studies, such as structural, biochemical, and biotechnological. Moreover, many studies have established the correlation between protein's solubility and its amino acid sequence; therefore, the primary structure of the protein can be used to determine its solubility. Since this physicochemical property plays a significant role in a wide range of research fields, it would be useful to have a sequence-based computational tool that would *in silico* determine solubility based on protein’s primary structure.

In this research, we introduced three machine/deep learning models for protein solubility prediction based on the raw amino acid sequences: RF, CNN and hybrid biLSTM. The assessment of the performances using evaluation metrics demonstrated that both models have an accuracy of over 65%. Moreover, the statistical analysis determined that proposed deep learning models have no significant difference in performance. In the future, the proposed models could be extended to other protein prediction tasks based on the raw protein sequence.

References:

Agostini, F., Vendruscolo, M., & Tartaglia, G. G. (2012). Sequence-Based Prediction of Protein Solubility. *Journal of Molecular Biology*, *421*(2–3), 237–241. https://doi.org/10.1016/j.jmb.2011.12.005

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Published. https://doi.org/10.1145/3292500.3330701

Bagby, S., Tong, K. I., & Ikura, M. (2001). Optimisation of Protein Solubility and Stability for Protein Nuclear Magnetic Resonance. *Methods in Enzymology*, 20–41. https://doi.org/10.1016/s0076-6879(01)39307-2

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimisation. Advances in neural information processing systems, 24.

Bertone, P., Kluger, Y., Lan, N., Zheng, D., Christendat, D., Yee, A., et al. (2001). SPINE: an integrated tracking database and data mining approach for identifying feasible targets in high-throughput structural proteomics. Nucleic Acids Res. 29, 2884–2898. doi: 10.1093/nar/29.13.2884

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*(1), 5–32. https://doi.org/10.1023/a:1010933404324

Chang,C.C.H. et al. (2014) Bioinformatics approaches for improved recombinant protein production in Escherichia coli: protein solubility prediction. Brief. Bioinform., 15, 953–962.

Chen, J., Zheng, S., Zhao, H., & Yang, Y. (2021). Structure-aware protein solubility prediction from sequence through graph convolutional network and predicted contact map. *Journal of Cheminformatics*, *13*(1). https://doi.org/10.1186/s13321-021-00488-1

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297. https://doi.org/10.1007/bf00994018

Davis GD, Elisee C, Newham DM, Harrison RG. New fusion protein systems designed to give soluble expression in Escherichia coli. Biotechnol Bioeng. 1999 Nov 20;65(4):382-8. PMID: 10506413.

Diaz, A. A., Tomba, E., Lennarson, R., Richard, R., Bagajewicz, M. J., & Harrison, R. G. (2010). Prediction of protein solubility in Escherichia coliusing logistic regression. Biotechnology and Bioengineering, 105(2), 374–383. https://doi.org/10.1002/bit.22537

Dobson, C. M. (2001). The structural basis of protein folding and its links with human disease. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, *356*(1406), 133–145. https://doi.org/10.1098/rstb.2000.0758

ElAbd, H., Bromberg, Y., Hoarfrost, A., Lenz, T., Franke, A., & Wendorff, M. (2020). Amino acid encoding for deep learning applications. *BMC Bioinformatics*, *21*(1). https://doi.org/10.1186/s12859-020-03546-x

Georgiou, G., & Valax, P. (1996). Expression of correctly folded proteins in Escherichia coli. *Current Opinion in Biotechnology*, *7*(2), 190–197. https://doi.org/10.1016/s0958-1669(96)80012-7

Fang, Y., & Fang, J. (2013). Discrimination of soluble and aggregation-prone proteins based on sequence information. *Molecular BioSystems*, *9*(4), 806. https://doi.org/10.1039/c3mb70033j

Fowler, S. B., Poon, S., Muff, R., Chiti, F., Dobson, C. M., & Zurdo, J. (2005). Rational design of aggregation-resistant bioactive peptides: Reengineering human calcitonin. *Proceedings of the National Academy of Sciences*, *102*(29), 10105–10110. https://doi.org/10.1073/pnas.0501215102

Fu,L. et al. (2012) CD-HIT: accelerated for clustering the next-generation sequencing data. Bioinformatics, 28, 3150–3152

Fukushima, K. (1980). Neocognitron: A self-organising neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4), 193–202. https://doi.org/10.1007/bf00344251

Han, X., Zhang, L., Zhou, K., & Wang, X. (2019). ProGAN: Protein solubility generative adversarial nets for data augmentation in DNN framework. *Computers & Chemical Engineering*, *131*, 106533. https://doi.org/10.1016/j.compchemeng.2019.106533

Hirose, S., & Noguchi, T. (2013). ESPRESSO: A system for estimating protein expression and solubility in protein expression systems. *PROTEOMICS*, *13*(9), 1444–1456. https://doi.org/10.1002/pmic.201200175

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hon, J., Marusiak, M., Martinek, T., Kunka, A., Zendulka, J., Bednar, D., & Damborsky, J. (2021). SoluProt: prediction of soluble protein expression in Escherichia coli. *Bioinformatics*, *37*(1), 23–28. https://doi.org/10.1093/bioinformatics/btaa1102

Hou, Q., Kwasigroch, J. M., Rooman, M., & Pucci, F. (2019). SOLart: a structure-based method to predict protein solubility and aggregation. *Bioinformatics*. https://doi.org/10.1093/bioinformatics/btz773

Huang, H. L., Charoenkwan, P., Kao, T. F., Lee, H. C., Chang, F. L., Huang, W. L., Ho, S. J., Shu, L. S., Chen, W. L., & Ho, S. Y. (2012). Prediction and analysis of protein solubility using a novel scoring card method with dipeptide composition. *BMC Bioinformatics*, *13*(S17). https://doi.org/10.1186/1471-2105-13-s17-s3

Idicula-Thomas, S., Kulkarni, A. J., Kulkarni, B. D., Jayaraman, V. K., & Balaji, P. V. (2005). A support vector machine-based method for predicting the propensity of a protein to be soluble or to form inclusion body on overexpression in Escherichia coli. *Bioinformatics*, *22*(3), 278–284. https://doi.org/10.1093/bioinformatics/bti810

Jen, A., & Merkle, H. P. (2001). Diamonds in the Rough: Protein Crystals from a Formulation Perspective. *Pharmaceutical Research*, *18*(11), 1483–1488. https://doi.org/10.1023/a:1013057825942

Jiang, H., Li, Y., Zhou, C., Hong, H., Glade, T., & Yin, K. (2020). Landslide Displacement Prediction Combining LSTM and SVR Algorithms: A Case Study of Shengjibao Landslide from the Three Gorges Reservoir Area. *Applied Sciences*, *10*(21), 7830. https://doi.org/10.3390/app10217830

Kalayan, J., Henchman, R. H., & Warwicker, J. (2019). Model for Counterion Binding and Charge Reversal on Protein Surfaces. *Molecular Pharmaceutics*, *17*(2), 595–603. https://doi.org/10.1021/acs.molpharmaceut.9b01047

Khurana, S., Rawi, R., Kunji, K., Chuang, G. Y., Bensmail, H., & Mall, R. (2018). DeepSol: a deep learning framework for sequence-based protein solubility prediction. *Bioinformatics*, *34*(15), 2605–2613. https://doi.org/10.1093/bioinformatics/bty166

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimisation. arXiv preprint arXiv:1412.6980

Kramer, R., Shende, V., Motl, N., Pace, C., & Scholtz, J. (2012). Toward a Molecular Understanding of Protein Solubility: Increased Negative Surface Charge Correlates with Increased Solubility. Biophysical Journal, 102(8), 1907–1915. https://doi.org/10.1016/j.bpj.2012.01.060

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324. https://doi.org/10.1109/5.726791

Li,W. and Godzik,A. (2006) CD-HIT: a fast program for clustering and comparing large sets of protein or nucleotide sequences. Bioinformatics, 22, 1658–1659.

Magnan, C. N., Randall, A., & Baldi, P. (2009). SOLpro: accurate sequence-based prediction of protein solubility. *Bioinformatics*, *25*(17), 2200–2207. https://doi.org/10.1093/bioinformatics/btp386

Maxwell, K. L., Bona, D., Liu, C., Arrowsmith, C. H., & Edwards, A. M. (2003). Refolding out of guanidine hydrochloride is an effective approach for high-throughput structural studies of small proteins. *Protein Science*, *12*(9), 2073–2080. https://doi.org/10.1110/ps.0393503

McPherson, A., & Gavira, J. A. (2013). Introduction to protein crystallisation. *Acta Crystallographica Section F Structural Biology Communications*, *70*(1), 2–20. https://doi.org/10.1107/s2053230x13033141

Mona, A., I, Hasan, M., Farzaneh, M. N., Golnaz, T., Hossein, A., & Soroush, S. (2010). Improving recombinant protein solubility in Escherichia coli: Identification of best chaperone combination which assists folding of human basic fibroblast growth factor. *African Journal of Biotechnology*, *9*(47), 8100–8109. https://doi.org/10.5897/ajb10.867

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In Icml.

Niwa, T., Ying, B. W., Saito, K., Jin, W., Takada, S., Ueda, T., & Taguchi, H. (2009). Bimodal protein solubility distribution revealed by an aggregation analysis of the entire ensemble of Escherichia coli proteins. Proceedings of the National Academy of Sciences, 106(11), 4201–4206. https://doi.org/10.1073/pnas.0811922106

Pédelacq, J. D., Piltch, E., Liong, E. C., Berendzen, J., Kim, C. Y., Rho, B. S., Park, M. S., Terwilliger, T. C., & Waldo, G. S. (2002). Engineering soluble proteins for structural genomics. *Nature Biotechnology*, *20*(9), 927–932. https://doi.org/10.1038/nbt732

Pham, P. V. (2018). Medical Biotechnology. *Omics Technologies and Bio-Engineering*, 449–469. https://doi.org/10.1016/b978-0-12-804659-3.00019-1

Phung, V., & Rhee, E. (2019). A High-Accuracy Model Average Ensemble of Convolutional Neural Networks for Classification of Cloud Image Patches on Small Datasets. *Applied Sciences*, *9*(21), 4500. https://doi.org/10.3390/app9214500

Rawi, R., Mall, R., Kunji, K., Shen, C. H., Kwong, P. D., & Chuang, G. Y. (2017). PaRSnIP: sequence-based protein solubility prediction using gradient boosting machine. *Bioinformatics*, *34*(7), 1092–1098. https://doi.org/10.1093/bioinformatics/btx662

Ross, C. A., & Poirier, M. A. (2004). Protein aggregation and neurodegenerative disease. Nature Medicine, 10(S7), S10–S17. https://doi.org/10.1038/nm1066

Schuster, M., & Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681. https://doi.org/10.1109/78.650093

Singh, S. M., & Panda, A. K. (2005). Solubilisation and refolding of bacterial inclusion body proteins. *Journal of Bioscience and Bioengineering*, *99*(4), 303–310. https://doi.org/10.1263/jbb.99.303

Smialowski, P., Martin-Galiano, A. J., Mikolajka, A., Girschick, T., Holak, T. A., & Frishman, D. (2006). Protein solubility: sequence-based prediction and experimental verification. *Bioinformatics*, *23*(19), 2536–2542. https://doi.org/10.1093/bioinformatics/btl623

Smialowski, P., Doose, G., Torkler, P., Kaufmann, S., & Frishman, D. (2012). PROSO II - a new method for protein solubility prediction. *FEBS Journal*, *279*(12), 2192–2200. https://doi.org/10.1111/j.1742-4658.2012.08603.x

Solá, R. J., & Griebenow, K. (2010). Glycosylation of Therapeutic Proteins. BioDrugs, 24(1), 9–21. https://doi.org/10.2165/11530550-000000000-00000

Tjong, H., & Zhou, H. X. (2008). Prediction of Protein Solubility from Calculation of Transfer Free Energy. *Biophysical Journal*, *95*(6), 2601–2609. https://doi.org/10.1529/biophysj.107.127746

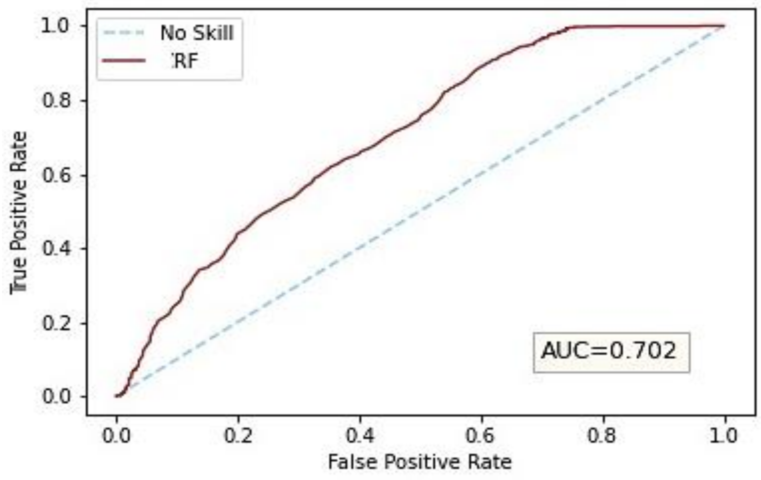
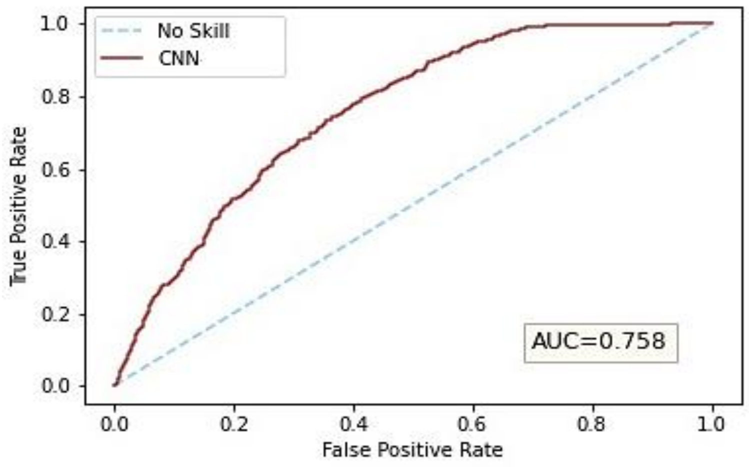
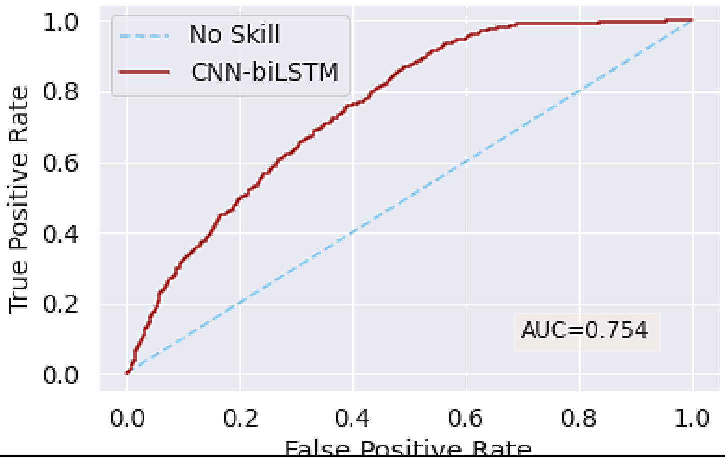
Wilkinson, D. L., & Harrison, R. G. (1991). Predicting the Solubility of Recombinant Proteins in Escherichia coli. *Nature Biotechnology*, *9*(5), 443–448. https://doi.org/10.1038/nbt0591-443

List of appendices

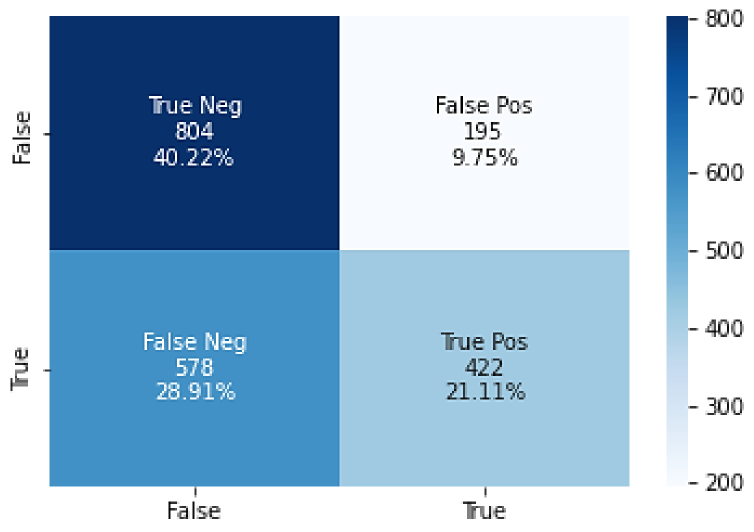
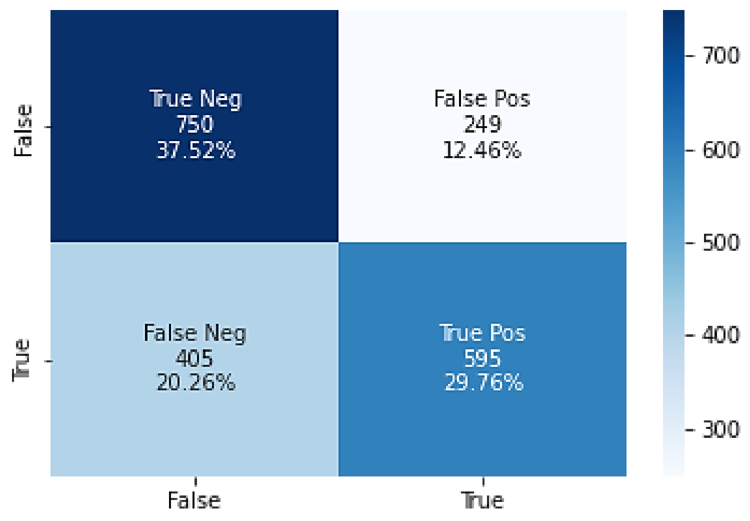
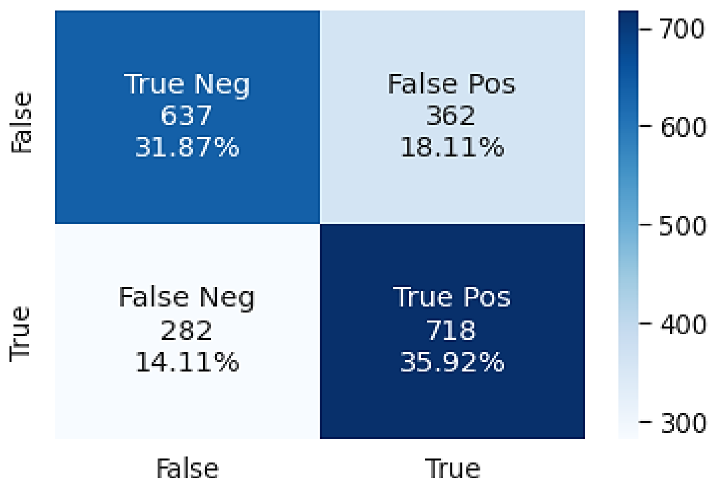
Appendix A. Hyperparameters for CNN and biLSTM.

Hyperparameter	Tested range	Optimal value	
		CNN	Hybrid biLSTM
Leaning rate	$\{10^{-k} \mid k \in \{3,4,\ldots,8\}\}$	2.1397772235811666e-05	1.890793210966646e-05
Batch size	$\{16 + 16 * k \mid k \in \{0,1,\ldots,15\}\}$	128	256}
Embedding dimension	$\{50 + 14 * k \mid k \in \{0,1\}\}$	50	50
Embedding dropout	$\{0.1 + 0.1 * k \mid k \in \{0,1,2\}\}$	0.3	0.2
Fully connected layer dropout	$\{0.1 + 0.1 * k \mid k \in \{0,1,\ldots,5\}\}$	0.5	0.30000000000000004
LSTM dropout	$\{0.1 + 0.1 * k \mid k \in \{0,1,\ldots,5\}\}$	-	0.4
Fully connected layer output channels	$\{8 + 8 * k \mid k \in \{0,1,\ldots,31\}\}$	224	152
Number of output channels in 1D convolutions	$\{16 + 16 * k \mid k \in \{0,1,\ldots,7\}\}$	112	
Convolution stride	$\{k \mid k \in \{1,2\}\}$	2	2
Max pooling stride	$\{k \mid k \in \{1,2\}\}$	2	2
Number of LSTM layers	$\{k \mid k \in \{2,3\}\}$	-	2
Number of features in a hidden state	$\{8 + 8 * k \mid k \in \{0,1,\ldots,31\}\}$	-	224

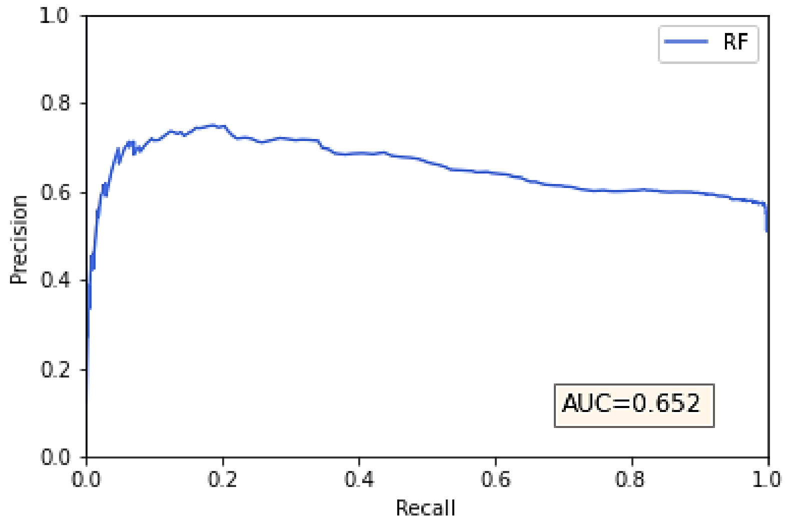
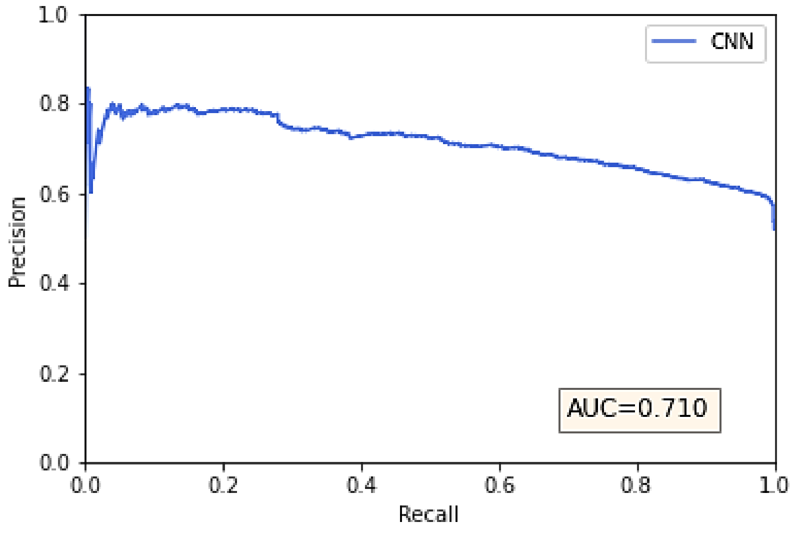
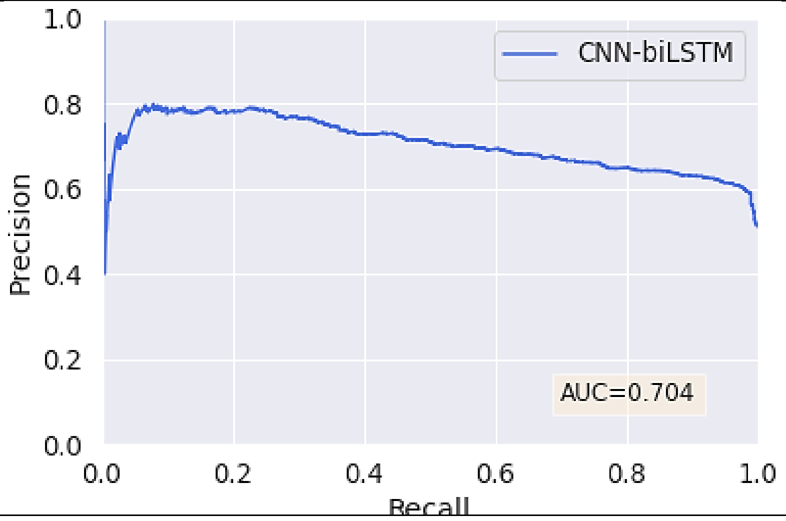
Appendix B. ROC Curve for RF, CNN and biLSTM.

Random Forest	
CNN	
BiLSTM	

Appendix C. Confusion matrices for RF, CNN and biLSTM.

Random Forest	
CNN	
BiLSTM	

Appendix D. Precision-Recall Curve for RF, CNN and biLSTM

Random Forest	
CNN	
BiLSTM	

Appendix E. ReLU plot (relu-graph.py).

```
import numpy as np
from matplotlib import pyplot

x = np.linspace(-10, 10, 1000)
y = np.maximum(x, 0)
pyplot.figure(figsize=(7, 5))
pyplot.text(-10.5, 8, "ReLU formula: \nR(x)=max(x,0)",
fontsize=15, verticalalignment='top',
            bbox=dict(facecolor='white', alpha=1))

pyplot.plot(x, y, color="firebrick", linewidth=3)
pyplot.legend(['ReLU'], markerscale=3, facecolor="white",
fontsize=15)
pyplot.savefig("graphs/ReLU.png")
pyplot.show()
```

Appendix F. Data preprocessing (data-pre-processing.py).

```
import torch


def make_data(src_file, tgt_file, train=False):
    src, tgt = [], []

    print('Processing %s & %s ...' % (src_file, tgt_file))
    srcF = open(src_file, 'r')
    tgtF = open(tgt_file, 'r')

    all_lines = []
    all_tgts = []

    while True:
        sline = srcF.readline()
        tline = tgtF.readline()

        # end of file
        if sline == "" and tline == "":
            break

        # source or target does not have same number of lines
        if sline == "" or tline == "":
            print('Error: source and target do not have the same
number of sentences')
            sys.exit(-1)
            break

        sline = sline.strip()
        tline = tline.strip()
        if sline == "" or tline == "":
            print('WARNING: ignoring an empty line (' + str(count
+ 1) + ')')
            continue

        all_lines.append(sline)
        all_tgts.append(int(tline))

    srcF.close()
    tgtF.close()

    src = all_lines
    tgt = all_tgts

    return src, tgt


# pre-process sequences and labels from the dataset
seq, label = make_data("data/train_src", "data/train_tgt")
val_seq, val_label = make_data("data/val_src", "data/val_tgt")
test_seq, test_label = make_data("data/test_src",
"data/test_tgt")

all_seq = seq + val_seq + test_seq
all_label = label + val_label + test_label

fordict = list(set(seq[0]))
n = list(range(1, 21))
vocab = dict(zip(fordict, n))

sequences = []
for i in all_seq:
    fortensor = []
    for j in range(len(i)):
        fortensor.append(vocab[i[j]])
    sequences.append(torch.tensor(fortensor))

# in order to make proteins of equal length we pad sequences with
0.
seq = torch.nn.utils.rnn.pad_sequence(sequences,
batch_first=True, padding_value=0.0)
# labels one-hot encoding
labels = torch.nn.functional.one_hot(torch.tensor(all_label),
num_classes=2).float()

torch.save(seq, 'padded_sequences.pt')
torch.save(labels, 'onehot_labels.pt')

print("data pre-processing is finished")
```

Appendix G. Random Forest training and evaluation (random-forest.py)

```
import torch
from torch.utils.data import Dataset, DataLoader

import pickle
import numpy as np
from numpy import sqrt, argmax
import seaborn as sns
from matplotlib import pyplot

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import recall_score, precision_score,
f1_score, confusion_matrix, roc_curve
from sklearn.metrics import matthews_corrcoef,
balanced_accuracy_score, auc
from sklearn.metrics import accuracy_score, roc_auc_score,
average_precision_score, precision_recall_curve
from sklearn.model_selection import train_test_split

# import data
sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

# split data set into training and test set
x_train, x_test, y_train, y_test =
train_test_split(sequences[0:69420], labels[0:69420],
test_size=0.1, random_state=42)

test_labels = labels[69420:71419]
test_labels = test_labels[:, 1]

print("start traning")
clf = RandomForestClassifier(701)
clf.fit(x_train, y_train.argmax(1).numpy())

print("model evaluation using independent test set")
pred_test = clf.predict_proba(sequences[69420:71419])
pred_label = np.where(pred_test[:, 1] > 0.5, 1, 0)

print("roc_auc_score:", round(roc_auc_score(test_labels,
pred_test[:, 1]), 2),
      "\naccuracy:", round(accuracy_score(test_labels,
pred_label), 2),
      "\nbalanced accuracy:",
round(balanced_accuracy_score(test_labels, pred_label), 2),
      "\nMCC:", round(matthews_corrcoef(test_labels, pred_label),
2),
      "\nrecall:", round(recall_score(test_labels, pred_label,
pos_label=1, average='binary'), 2),
      "\nf1:", round(f1_score(test_labels, pred_label,
average='binary'), 2),
      "\nprecision:", round(precision_score(test_labels,
pred_label, average='binary', zero_division=1), 2),
      "\nAP:", round(average_precision_score(test_labels,
pred_label), 2))

# plot confusion matrix

conf_matrix = confusion_matrix(test_labels, pred_label)

categories = ['True Neg', 'False Pos', 'False Neg', 'True Pos']
categories_counts = ["{0:0.0f}".format(value) for value in
                     conf_matrix.flatten()]
categories_percentages = ["{0:.2%}".format(value) for value in
                          conf_matrix.flatten() /
np.sum(conf_matrix)]

label = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
         zip(categories, categories_counts,
categories_percentages)]

label = np.asarray(label).reshape(2, 2)

ax = sns.heatmap(conf_matrix, annot=label, fmt='', cmap='Blues')

ax.set_title('Confusion Matrix RF \n\n')
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('True Values ')

# Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])
pyplot.savefig('graphs/cf_matrix-rf.png')
pyplot.show()

# plot precision-recall curve

precision, recall, thresholds =
precision_recall_curve(test_labels, pred_test[:, 1])
f1, auc_score = f1_score(test_labels, pred_label), auc(recall,
precision)
pyplot.plot(recall, precision, label='RF', color="royalblue")
pyplot.text(0.7, 0.1, "AUC=%.3f" % auc(recall, precision),
fontsize=12, bbox=dict(facecolor='papayawhip', alpha=0.5))

pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.xlim(0, 1)
pyplot.ylim(0, 1)
pyplot.legend()
# show the plot
pyplot.savefig('graphs/prc_rf.png')
pyplot.show()
```

```
false_positive_rate, true_positive_rate, thresholds =
roc_curve(test_labels, pred_test[:, 1])

print("Area Under ROC Curve=%.3f" % roc_auc_score(test_labels,
pred_test[:, 1]))
pyplot.plot([0, 1], [0, 1], linestyle='--', label='No Skill',
color="lightskyblue")
pyplot.plot(false_positive_rate, true_positive_rate, label='CRF',
color="firebrick")
pyplot.text(0.7, 0.1, "AUC=%.3f" % roc_auc_score(test_labels,
pred_test[:, 1]), fontsize=12,
            bbox=dict(facecolor='papayawhip', alpha=0.3))
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.legend()
# show the plot
pyplot.savefig('graphs/roc-rf.jpg')
pyplot.show()
```

Appendix H. Random Forest 10-fold cross-validation (random-forest-10-fold-CV.py)

```
# validation

print("10-fold cross-validation of the model")

from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

X = sequences[0:69420]
Y = labels[0:69420]
Y = Y[:, 1]
# prepare the cross-validation procedure
cv = KFold(n_splits=10, random_state=42, shuffle=True)
# create model
clf = RandomForestClassifier(701)
# evaluate model
scores = cross_val_score(clf, X, Y, scoring='balanced_accuracy',
cv=cv, n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))

rf = {"RF": list(np.around(scores * 100, 3))}

# update the file that contains accuracy distributions for each
model
```

```
with open('boxplot.pkl', 'rb') as f:
    d = pickle.load(f)
d.update(rf)
with open('boxplot.pkl', 'wb') as f:
    pickle.dump(d, f)
```

Appendix I. CNN training and evaluation (cnn-model.py).

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

import pickle
import numpy as np
from numpy import sqrt, argmax

from sklearn.metrics import recall_score, precision_score,
f1_score, precision_recall_curve, confusion_matrix
from sklearn.metrics import roc_curve, roc_auc_score, auc,
fbeta_score
from sklearn.metrics import accuracy_score,
balanced_accuracy_score, average_precision_score,
matthews_corrcoef
from matplotlib import pyplot
import seaborn as sns

# download the preprocess data
sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

# randomly split the given dataset randomly into the training set
and validation set of given lengths
train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,
valid_size])

# obtain the amount of soluble and insoluble proteins training,
validation and test sets
l1, l2, l3 = [], [], []
for i1, j1 in train_set:
    l1.append(torch.argmax(j1, dim=0))
```

```
for i2, j2 in valid_set:
    l2.append(torch.argmax(j2, dim=0))

for i3, j3 in dataset_test:
    l3.append(torch.argmax(j3, dim=0))

print("overall:", round((l1 + l2 + l3).count(0) * 100 / len(l1 +
l2 + l3)), "% insoluble ",
      round((l1 + l2 + l3).count(1) * 100 / len(l1 + l2 + l3)),
"% soluble",
      "\ntraining set insoluble:", round(l1.count(0) * 100 /
len(l1)), "%",
      "\ntraining set soluble:", round(l1.count(1) * 100 /
len(l1)), "%",
      "\nvalidation set insoluble:", round(l2.count(0) * 100 /
len(l2)), "%",
      "\nvalidation set soluble:", round(l2.count(1) * 100 /
len(l2)), "%",
      "\ntest set insoluble:", round(l3.count(0) * 100 /
len(l3)), "%",
      "\ntest set soluble:", round(l3.count(1) * 100 / len(l3)),
"%")

# load the optimal hyperparamters obtained during tuning in the
model
with open('parameters/best_parameters_cnn.pkl', 'rb') as f:
    model_param = pickle.load(f)

max_len = len(max(sequences, key=len))
model_param.update({"max_len": max_len})  # add the maximal
length of proteins to the parameters dictionary


# model architecture
class Sol(nn.ModuleList):

    def __init__(self, model_param):
        super(Sol, self).__init__()

        self.maxlen = model_param["max_len"]
        self.drop_embed = model_param['drop_em']
        self.drop1 = model_param['drop1']
        self.stride1 = model_param['stride1']
        self.stride2 = model_param['stride2']
        self.embed_dim = model_param['embed_size']
        self.out_size = model_param['out_size']
        self.fc_layer = model_param['fclayer']

        self.dropout_em = nn.Dropout(self.drop_embed)
        self.dropout = nn.Dropout(self.drop1)

        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8

        # Calculate the output of each convolution and add them.
       # We use this number further to calculate the amount of
input channels in LSTM.
        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1

        fc_layer_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])

        # Convolution layers definition
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.embedding_layer = nn.Embedding(21, self.embed_dim)
        # Max pooling layers definition
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        # Fully connected layer definition
        self.fc1 = nn.Linear(self.out_size * fc_layer_input_size,
self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_em(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)

        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)

        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)

        fc_output = self.fc1(torch.flatten(union, start_dim=1))
        fc_output = self.dropout(fc_output)

        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)
        return fc_output


# model training
model = Sol(model_param)

criterion = nn.BCEWithLogitsLoss()   #
optimizer = torch.optim.Adam(model.parameters(),
lr=model_param['lr'])   #
device = 'cuda' if torch.cuda.is_available() else 'cpu'

model.to(device)

train = DataLoader(train_set,
                   batch_size=model_param["batch_size"],
                   shuffle=True,
                   num_workers=8)

val = DataLoader(valid_set,
                 batch_size=model_param['batch_size'],
                 shuffle=True,
                 num_workers=8)

min_val_loss = np.inf
n_stop = 5
not_improved = 0
early_stop = False
# iter=0

print("start training")
for epoch in range(100):

    training_loss = 0.0
    step = 0
    model.train()
    # training the model
    for inputs, labels in train:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()

        loss = criterion(model(inputs), labels)

        loss.backward()
        optimizer.step()

        training_loss += loss.item()

    valid_loss = 0.0
    model.eval()
    # validation of the model
    with torch.no_grad():

        for inputs, labels in val:
            inputs, labels = inputs.to(device), labels.to(device)

            output = model(inputs)
            loss = criterion(output, labels)

            valid_loss += loss.item()

        print(
            f'Epoch {epoch + 1} \t\t Training Loss:
{round(training_loss / len(train), 4)} \t\t Validation Loss:
{valid_loss / len(val)}')
        # early stopping technique:
        # if there is no decrease in validation loss for 5 epochs
is a row,
        # then training is stopped
        if valid_loss < min_val_loss:
            epochs_no_improve = 0
            min_val_loss = valid_loss
            print("validation loss decreased")

        else:
            not_improved += 1
        # iter += 1

        if epoch > 5 and not_improved == n_stop:
            print('Early stopping!')
            early_stop = True
            break
        else:
            continue
        break

if early_stop:
    print("Stopped")

print("Finished Training")

# model evaluation

print("Evaluating model using test set")

device = "cpu"
model = model.to(device)

test = DataLoader(dataset_test,
                  batch_size=32,
                  shuffle=False,
                  num_workers=8)

prediction = torch.tensor([])
true_values = torch.tensor([])

with torch.no_grad():
    for seq, lbl in test:
        pred_prob = model(seq)
        outputs = torch.sigmoid(pred_prob)
        p = outputs[:, 1]
        l = lbl[:, 1]
        prediction = torch.cat((prediction, p.detach().cpu()),
dim=0)
        true_values = torch.cat((true_values, l.detach().cpu()),
dim=0)

prediction = prediction.numpy()
true_values = true_values.numpy()

predicted = np.where(prediction > 0.4, 1, 0)

print("roc_auc_score:", round(roc_auc_score(true_values,
prediction), 2),
      "\naccuracy:", round(accuracy_score(true_values,
predicted), 2),
      "\nbalanced accuracy:",
round(balanced_accuracy_score(true_values, predicted), 2),
      "\nMCC:", round(matthews_corrcoef(true_values, predicted),
2),
      "\nrecall:", round(recall_score(true_values, predicted,
pos_label=1, average='binary'), 2),
      "\nf1:", round(f1_score(true_values, predicted,
average='binary'), 2),
      "\nprecision:", round(precision_score(true_values,
predicted, average='binary', zero_division=1), 2),
      "\nAP:", round(average_precision_score(true_values,
predicted), 2))

# plot confusion matrix

conf_matrix = confusion_matrix(true_values, predicted)

categories = ['True Neg', 'False Pos', 'False Neg', 'True Pos']
categories_counts = ["{0:0.0f}".format(value) for value in
                     conf_matrix.flatten()]
categories_percentages = ["{0:.2%}".format(value) for value in
                          conf_matrix.flatten() /
np.sum(conf_matrix)]

labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
          zip(categories, categories_counts,
categories_percentages)]

labels = np.asarray(labels).reshape(2, 2)

ax = sns.heatmap(conf_matrix, annot=labels, fmt='', cmap='Blues')

ax.set_title('Confusion Matrix CNN \n\n')
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('True Values ');

ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])
pyplot.savefig('graphs/cf_matrix-cnn.png')
pyplot.show()

# plot precision-recall curve
precision, recall, thresholds =
precision_recall_curve(true_values, prediction)
f1, auc_score = f1_score(true_values, predicted), auc(recall,
precision)
pyplot.plot(recall, precision, label='CNN', color="royalblue")
print('CNN: f1=%.3f auc=%.3f' % (f1, auc_score))
pyplot.text(0.7, 0.1, "AUC=%.3f" % auc(recall, precision),
fontsize=12, bbox=dict(facecolor='papayawhip', alpha=0.5))
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.xlim(0, 1)
pyplot.ylim(0, 1)
pyplot.legend()
# show the plot
pyplot.savefig('graphs/prc_cnn.png')
pyplot.show()

# plot ROC curve
false_positive_rate, true_positive_rate, thresholds =
roc_curve(true_values, prediction)
print("Area Under ROC Curve=%.3f" % roc_auc_score(true_values,
prediction))
pyplot.plot([0, 1], [0, 1], linestyle='--', label='No Skill',
color="lightskyblue")
pyplot.plot(false_positive_rate, true_positive_rate, label='CNN',
color="firebrick")
pyplot.text(0.7, 0.1, "AUC=%.3f" % roc_auc_score(true_values,
prediction), fontsize=12,
            bbox=dict(facecolor='papayawhip', alpha=0.3))
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.legend()
# show the plot
pyplot.savefig('graphs/roc-cnn.jpg')
pyplot.show()
```

Appendix J. CNN 10-fold cross-validation (cnn-10-fold-CV.py).

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import KFold

import pickle
import numpy as np

from sklearn.metrics import roc_curve, roc_auc_score, auc
from sklearn.metrics import accuracy_score,
balanced_accuracy_score
from matplotlib import pyplot

# download the preprocess data
sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,
valid_size])

# load the optimal hyperparamters obtained during tuning in the
model
with open('parameters/best_parameters_cnn.pkl', 'rb') as f:
    model_param = pickle.load(f)

max_len = len(max(sequences, key=len))
model_param.update({"max_len": max_len})  # add the maximal
length of proteins to the parameters dictionary


# model architecture
class Sol(nn.ModuleList):

    def __init__(self, model_param):
        super(Sol, self).__init__()

        self.maxlen = model_param["max_len"]
        self.drop_embed = model_param['drop_em']
        self.drop1 = model_param['drop1']
        self.stride1 = model_param['stride1']
        self.stride2 = model_param['stride2']
        self.embed_dim = model_param['embed_size']
        self.out_size = model_param['out_size']
        self.fc_layer = model_param['fclayer']

        self.dropout_em = nn.Dropout(self.drop_embed)
        self.dropout = nn.Dropout(self.drop1)

        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8
        # Calculate the output of each convolution and add them.
        # We use this number further to calculate the amount of
input channels in fully connected layer.
        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1

        fc_layer_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])

        # Convolution layers definition
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
```

```
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.embedding_layer = nn.Embedding(21, self.embed_dim)

        # Max pooling layers definition
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        # Fully connected layer definition
        self.fc1 = nn.Linear(self.out_size * fc_layer_input_size,
self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_em(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)

        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)
        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)

        fc_output = self.fc1(torch.flatten(union, start_dim=1))
        fc_output = self.dropout(fc_output)

        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)

        return fc_output


# reset model weights after each fold
def reset_weights(m):
    for layer in m.children():
        if hasattr(layer, 'reset_parameters'):
            layer.reset_parameters()


k_folds = 10
num_epochs = 100
results = {}

# Set fixed random number seed
torch.manual_seed(42)

# Define the K-fold Cross Validator
kfold = KFold(n_splits=k_folds, shuffle=True)

# K-fold Cross Validation model evaluation
for fold, (train_ids, test_ids) in
enumerate(kfold.split(dataset_train)):

    min_val_loss = np.inf
    n_epochs_stop = 5
    epochs_no_improve = 0
    early_stop = False

    print(f'FOLD {fold}')

    train_subsampler =
torch.utils.data.SubsetRandomSampler(train_ids)
    test_subsampler =
torch.utils.data.SubsetRandomSampler(test_ids)

    train = DataLoader(dataset_train,
                       batch_size=model_param["batch_size"],
                       num_workers=8,
                       sampler=train_subsampler)

    val = DataLoader(dataset_train,
                     batch_size=model_param['batch_size'],
                     num_workers=8,
                          sampler=test_subsampler)

    model = Sol(model_param)
    model.apply(reset_weights)

    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(),
lr=model_param['lr'])
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model.to(device)

    for epoch in range(0, num_epochs):

        train_loss = 0.0
        epoch_steps = 0
        model.train()

        for i, data in enumerate(train, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            probs = model(inputs.to(device))
            loss = criterion(probs, labels.to(device))

            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        model.eval()
        accuracy = 0
        predict = torch.tensor([])
        lab = torch.tensor([])
        valid_loss = 0.0

        with torch.no_grad():

            for data in val:
                inputs, label = data
                inputs, label = inputs.to(device),
label.to(device)

                output = model(inputs)
                loss = criterion(output, label)

                valid_loss += loss.item()

                pred = torch.sigmoid(output)
                l = label[:, 1]
                prediction = pred[:, 1]
                predict = torch.cat((predict,
prediction.detach().cpu()), dim=0)
```

```
                lab = torch.cat((lab, l.detach().cpu()), dim=0)

            p = np.where(predict > 0.4, 1, 0)

            accuracy = round(balanced_accuracy_score(lab.numpy(),
p), 5)  # calculate balanced accuracy for each fold

            print(
                f'Epoch {epoch + 1} \t\t Training Loss:
{round(train_loss / len(train), 5)} \t\t Validation Loss:
{round(valid_loss / len(val), 5)}')

            # early stopping technique
            if valid_loss < min_val_loss:
                epochs_no_improve = 0
                min_val_loss = valid_loss
                print(valid_loss)

            else:
                epochs_no_improve += 1

            if epoch > 5 and epochs_no_improve == n_epochs_stop:
                print('Early stopping!')
                early_stop = True
                break
            else:
                continue
            break

    if early_stop:
        print("Stopped")

    # Print accuracy
    print('Accuracy for fold %d: %d %%' % (fold, 100.0 *
accuracy))
    results[fold] = 100.0 * (accuracy)

# Print fold results
print(f'K-FOLD CROSS VALIDATION RESULTS FOR {k_folds} FOLDS')
sum = 0.0
for key, value in results.items():
    print(f'Fold {key}: {value} %')
    sum += value
print(f'Average: {sum / len(results.items())} %')

fold_accuracy = []
for key, value in results.items():
    fold_accuracy.append(value)

kfoldaccuracy = dict({"CNN": fold_accuracy})

# update file with accuracy distributions for each model
```

```
with open('boxplot.pkl', 'rb') as f:
    d = pickle.load(f)
d.update(kfoldacc)
with open('boxplot.pkl', 'wb') as f:
    pickle.dump(d, f)
```

Appendix K. CNN hyperparameter tuning (cnn-hyperparam-tune.py).

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import functional as F
import optuna
import os

import pickle
import json
import numpy as np

from sklearn.metrics import balanced_accuracy_score

# load data

sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

# randomly split the given dataset randomly into the training set
and validation set of given lengths
train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,

valid_size])


def load_data(train=train_set, val=valid_set):
    return train, val


max_len = len(max(sequences, key=len))


# model
```

```
class Sol(nn.ModuleList):

    def __init__(self, maxlen, trial):
        super(Sol, self).__init__()

        self.maxlen = maxlen

        self.dropout_em = trial.suggest_float("drop_em", 0.1,
0.3, step=0.1)
        self.drop1 = trial.suggest_float("drop1", 0.1, 0.6,
step=0.1)
        self.fc_layer = trial.suggest_int("fclayer", 8, 256,
step=8)
        self.stride1 = trial.suggest_int("stride1", 1, 2, step=1)
        self.stride2 = trial.suggest_int("stride2", 1, 2, step=1)
        self.out_size = trial.suggest_int("out_size", 16, 128,
step=16)
        self.embed_dim = trial.suggest_int("embed_size", 50, 64,
14)

        self.dropout_embed = nn.Dropout(self.dropout_em)
        self.dropout = nn.Dropout(self.drop1)

        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8

        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
```

```
        fc_layer_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])

        # Convolution layers definition
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.embedding_layer = nn.Embedding(21, self.embed_dim)

        # Max pooling layers definition
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        # Fully connected layer definition
        self.fc1 = nn.Linear(self.out_size * fc_layer_input_size,
self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_embed(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)

        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)

        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)

        fc_output = self.fc1(torch.flatten(union, start_dim=1))
        fc_output = self.dropout(fc_output)

        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)

        return fc_output


def objective(trial):
    model = Sol(max_len, trial)
    device = "cpu"

    # if torch.cuda.is_available():
    #   device = "cuda"

    model.to(device)

    lr = trial.suggest_float("lr", 1e-8, 1e-3, log=True)
    batch_size = trial.suggest_int("batch_size", 16, 256,
step=16)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    trainlist, vallist = load_data()

    trainloader = torch.utils.data.DataLoader(
        trainlist,
        batch_size=batch_size,
        shuffle=True,
        num_workers=16)

    valloader = torch.utils.data.DataLoader(
        vallist,
        batch_size=batch_size,
        shuffle=True,
        num_workers=16)

    min_val_loss = np.inf
    n_epochs_stop = 5
    epochs_no_improve = 0
    early_stop = False

    for epoch in range(100):

        model.train()

        for inp in trainloader:
            sequence, labels = inp
            sequence, labels = sequence.to(device),
labels.to(device)

            optimizer.zero_grad()

            output = model(sequence)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()

        # Validation of the model.
        model.eval()
        accuracy = 0
        predict = torch.tensor([])
        lab = torch.tensor([])
        valid_loss = 0.0

        with torch.no_grad():

            for data in valloader:
                inputs, label = data
                inputs, label = inputs.to(device),
label.to(device)

                output = model(inputs)
                loss = criterion(output, label)

                valid_loss += loss.item() * inputs.size(0)

                pred = torch.sigmoid(output)

                l = label[:, 1]
                prediction = pred[:, 1]

                # prediction = torch.argmax
                predict = torch.cat((predict,
prediction.detach().cpu()), dim=0)
                lab = torch.cat((lab, l.detach().cpu()), dim=0)
        p = np.where(predict > 0.4, 1, 0)

        accuracy = round(balanced_accuracy_score(lab.numpy(), p),
5)

        trial.report(accuracy, epoch)

        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

        if valid_loss < min_val_loss:
            epochs_no_improve = 0
            min_val_loss = valid_loss

        else:
            epochs_no_improve += 1

        if epoch > 2 and epochs_no_improve == n_epochs_stop:
            print('Early stopping!')
            early_stop = True
            break
        else:
            continue
        break

    if early_stop:
        print("Stopped")

    return accuracy


study = optuna.create_study(direction='maximize',
study_name="cnn_tune")
study.optimize(objective, n_trials=25)

trial = study.best_trial

print('Accuracy: {}'.format(trial.value))
print("Best hyperparameters: {}".format(trial.params))

best_config = trial.params

with open('parameters/best_parameters_cnn.pkl', 'wb') as f:
    pickle.dump(best_config, f)
```

Appendix L. BiLSTM model training and evaluation (cnn-bilstm-mode.py).

```
# Import

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

import pickle
import numpy as np
from numpy import sqrt, argmax

from sklearn.metrics import recall_score, precision_score,
f1_score, precision_recall_curve, confusion_matrix
from sklearn.metrics import roc_curve, roc_auc_score, auc,
fbeta_score
from sklearn.metrics import accuracy_score,
balanced_accuracy_score, average_precision_score,
matthews_corrcoef
from sklearn.metrics import classification_report
from matplotlib import pyplot
import seaborn as sns

# load data

sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

# randomly split the given dataset randomly into the training set
and validation set of given lengths
train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,

valid_size])

# obtain the amount of soluble and insoluble proteins training,
validation and test sets
l1, l2, l3 = [], [], []
for i1, j1 in train_set:
    l1.append(torch.argmax(j1, dim=0))

for i2, j2 in valid_set:
    l2.append(torch.argmax(j2, dim=0))

for i3, j3 in dataset_test:
    l3.append(torch.argmax(j3, dim=0))

print("overall:", round((l1 + l2 + l3).count(0) * 100 / len(l1 +
l2 + l3)), "% insoluble ",
      round((l1 + l2 + l3).count(1) * 100 / len(l1 + l2 + l3)),
"% soluble",
      "\ntraining set insoluble:", round(l1.count(0) * 100 /
len(l1)), "%",
      "\ntraining set soluble:", round(l1.count(1) * 100 /
len(l1)), "%",
      "\nvalidation set insoluble:", round(l2.count(0) * 100 /
len(l2)), "%",
      "\nvalidation set soluble:", round(l2.count(1) * 100 /
len(l2)), "%",
      "\ntest set insoluble:", round(l3.count(0) * 100 /
len(l3)), "%",
      "\ntest set soluble:", round(l3.count(1) * 100 / len(l3)),
"%")

# load the optimal hyperparamters obtained during tuning in the
model
with open('parameters/best_parameters.pkl', 'rb') as f:
    model_param = pickle.load(f)

max_len = len(max(sequences, key=len))
model_param.update({"max_len": max_len})  # add the maximal
length of proteins to the parameters dictionary


# model architecture

class Sol(nn.ModuleList):

    def __init__(self, model_param):
        super(Sol, self).__init__()

        self.maxlen = model_param["max_len"]
        self.drop_embed = model_param['drop_em']
        self.drop1 = model_param['drop1']
        self.drop2 = model_param['drop2']
        self.stride1 = model_param['stride1']
        self.stride2 = model_param['stride2']
        self.embed_dim = model_param['embed_size']
        self.out_size = model_param['out_size']
        self.fc_layer = model_param['fclayer']
        self.hidden = model_param['hid']
        self.lstm_layers = model_param['layers']

        self.dropout_embed = nn.Dropout(self.drop_embed)
        self.dropout = nn.Dropout(self.drop1)
        # kernel sizes
        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8

        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1

        self.lstm_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])

        # convolutional layers configuration
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.lstm = nn.LSTM(self.lstm_input_size, self.hidden,
self.lstm_layers, batch_first=True, dropout=self.drop2,
                            bidirectional=True)
        self.embedding_layer = nn.Embedding(21, self.embed_dim)

        # pooling layers configuration
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        self.fc1 = nn.Linear(self.hidden, self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_embed(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)

        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)

        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)
        # out - tensor of output features,
        # hs - tensor containing final hidden state for each
element,
        # cs- tensor containing final cell state for each element
        out, (hs, cs) = self.lstm(
            union)
        fc_output = self.fc1(torch.flatten(hs[-1], start_dim=1))
        fc_output = self.dropout(fc_output)
        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)

        return fc_output


# model training

model = Sol(model_param)

criterion = nn.BCEWithLogitsLoss()  #
optimizer = torch.optim.Adam(model.parameters(),
lr=model_param['lr'])  #
device = 'cuda' if torch.cuda.is_available() else 'cpu'

model.to(device)

train = DataLoader(train_set,
                   batch_size=model_param["batch_size"],
                   shuffle=True,
                   num_workers=8)

val = DataLoader(valid_set,
                 batch_size=model_param['batch_size'],
                 shuffle=True,
                 num_workers=8)

min_val_loss = np.inf
n_stop = 5
not_improved = 0
early_stop = False

for epoch in range(100):

    training_loss = 0.0
    step = 0
    model.train()

    for inputs, labels in train:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        # print(model(inputs).shape)
        # print(labels.shape)
        loss = criterion(model(inputs), labels)

        loss.backward()
        optimizer.step()

        training_loss += loss.item()
    valid_loss = 0.0
    model.eval()

    with torch.no_grad():

        for inputs, labels in val:
            inputs, labels = inputs.to(device), labels.to(device)

            output = model(inputs)
            loss = criterion(output, labels)

            valid_loss += loss.item()

        print(
            f'Epoch {epoch + 1} \t\t Training Loss:
{round(training_loss / len(train), 4)} \t\t Validation Loss:
{valid_loss / len(val)}')

        if valid_loss < min_val_loss:
            epochs_no_improve = 0
            min_val_loss = valid_loss
            print(valid_loss)

        else:
            not_improved += 1

        if epoch > 5 and not_improved == n_stop:
            print('Early stopping!')
            early_stop = True
            break
        else:
            continue
        break

if early_stop:
    print("Stopped")

print("Finished Training")

# Evaluation

print("Evaluating model using test set")

device = "cpu"
model = model.to(device)

test = DataLoader(dataset_test,
                  batch_size=32,
                  shuffle=False,
                  num_workers=8)

prediction = torch.tensor([])
true_values = torch.tensor([])
```

```
with torch.no_grad():
    for seq, lbl in test:
        pred_prob = model(seq)
        outputs = torch.sigmoid(pred_prob)
        p = outputs[:, 1]
        l = lbl[:, 1]
        prediction = torch.cat((prediction, p.detach().cpu()),
dim=0)
        true_values = torch.cat((true_values, l.detach().cpu()),
dim=0)

prediction = prediction.numpy()
true_values = true_values.numpy()

predicted = np.where(prediction > 0.4, 1, 0)

print("roc_auc_score:", round(roc_auc_score(true_values,
prediction), 2),
      "\naccuracy:", round(accuracy_score(true_values,
predicted), 2),
      "\nbalanced accuracy:",
round(balanced_accuracy_score(true_values, predicted), 2),
      "\nMCC:", round(matthews_corrcoef(true_values, predicted),
2),
      "\nrecall:", round(recall_score(true_values, predicted,
pos_label=1, average='binary'), 2),
      "\nf1:", round(f1_score(true_values, predicted,
average='binary'), 2),
      "\nprecision:", round(precision_score(true_values,
predicted, average='binary', zero_division=1), 2),
      "\nAP:", round(average_precision_score(true_values,
predicted), 2))

# plot confusion matrix

conf_matrix = confusion_matrix(true_values, predicted)

categories = ['True Neg', 'False Pos', 'False Neg', 'True Pos']
categories_counts = ["{0:0.0f}".format(value) for value in
                     conf_matrix.flatten()]
categories_percentages = ["{0:.2%}".format(value) for value in
                          conf_matrix.flatten() /
np.sum(conf_matrix)]

labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
          zip(categories, categories_counts,
categories_percentages)]

labels = np.asarray(labels).reshape(2, 2)

sns.set(font_scale=1.2)
ax = sns.heatmap(conf_matrix, annot=labels, fmt='', cmap='Blues')
```

```
ax.set_title('Confusion Matrix CNN-biLSTM \n\n')
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('True Values ')

ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])
pyplot.savefig('graphs/cf_matrix-cnn-bilstm.png')
pyplot.show()

# plot precision-recall curve
precision, recall, thresholds =
precision_recall_curve(true_values, prediction)
f1, auc_score = f1_score(true_values, predicted), auc(recall,
precision)
pyplot.plot(recall, precision, label='CNN-biLSTM',
color="royalblue")
pyplot.text(0.7, 0.1, "AUC=%.3f" % auc(recall, precision),
fontsize=12, bbox=dict(facecolor='papayawhip', alpha=0.5))
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.xlim(0, 1)
pyplot.ylim(0, 1)
pyplot.legend()
# show the plot

pyplot.savefig('graphs/prc_cnn-bilstm.png')
pyplot.show()

# plot ROC curve
false_positive_rate, true_positive_rate, thresholds =
roc_curve(true_values, prediction)

print("Area Under ROC Curve=%.3f" % roc_auc_score(true_values,
prediction))
pyplot.plot([0, 1], [0, 1], linestyle='--', label='No Skill',
color="lightskyblue")
pyplot.plot(false_positive_rate, true_positive_rate, label='CNN-
biLSTM', color="firebrick")
pyplot.text(0.7, 0.1, "AUC=%.3f" % roc_auc_score(true_values,
prediction), fontsize=12,
            bbox=dict(facecolor='papayawhip', alpha=0.3))

# axis labels

pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.legend()

# show the plot
pyplot.savefig('graphs/roc-cnn-bilstm.png')
pyplot.show()
```

Appendix M. BiLSTM 10-fold cross-validation (cnn-bilstm-10-fold-CV.py)

```
# Import

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import KFold

import pickle
import numpy as np

from sklearn.metrics import roc_curve, roc_auc_score, auc
from sklearn.metrics import accuracy_score,
balanced_accuracy_score
from matplotlib import pyplot

# load data

sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')

dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

# randomly split the given dataset randomly into the training set
and validation set of given lengths
train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,

valid_size])

# load the optimal hyperparamters obtained during tuning in the
model
with open('parameters/best_parameters.pkl', 'rb') as f:
    model_param = pickle.load(f)

max_len = len(max(sequences, key=len))
model_param.update({"max_len": max_len})  # add the maximal
length of proteins to the parameters dictionary


# model architecture

class Sol(nn.ModuleList):
    def __init__(self, model_param):
        super(Sol, self).__init__()

        self.maxlen = model_param["max_len"]
        self.drop_embed = model_param['drop_em']
        self.drop1 = model_param['drop1']
        self.drop2 = model_param['drop2']
        self.stride1 = model_param['stride1']
        self.stride2 = model_param['stride2']
        self.embed_dim = model_param['embed_size']
        self.out_size = model_param['out_size']
        self.fc_layer = model_param['fclayer']
        self.hidden = model_param['hid']
        self.lstm_layers = model_param['layers']

        self.dropout_embed = nn.Dropout(self.drop_embed)
        self.dropout = nn.Dropout(self.drop1)

        # kernel sizes
        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8

        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1

        self.lstm_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])
```

```
        # convolutional layers configuration
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.lstm = nn.LSTM(self.lstm_input_size, self.hidden,
self.lstm_layers, batch_first=True, dropout=self.drop2,
                            bidirectional=True)
        self.embedding_layer = nn.Embedding(21, self.embed_dim)

        # pooling layers configuration
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        self.fc1 = nn.Linear(self.hidden, self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_embed(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)

        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)

        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)
        # out - tensor of output features,
        # hs - tensor containing final hidden state for each
element,
        # cs- tensor containing final cell state for each element
        out, (hs, cs) = self.lstm(
            union)

        fc_output = self.fc1(torch.flatten(hs[-1], start_dim=1))
        fc_output = self.dropout(fc_output)
        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)

        return fc_output


def reset_weights(m):
    for layer in m.children():
        if hasattr(layer, 'reset_parameters'):
            layer.reset_parameters()


k_folds = 10
num_epochs = 100
results = {}

# Set fixed random number seed
torch.manual_seed(42)

# Define the K-fold Cross Validator
kfold = KFold(n_splits=k_folds, shuffle=True)

# K-fold Cross Validation model evaluation
for fold, (train_ids, test_ids) in
enumerate(kfold.split(dataset_train)):

    min_val_loss = np.inf
    n_epochs_stop = 5
    epochs_no_improve = 0
    early_stop = False

    print(f'FOLD {fold}')
    train_subsampler =
torch.utils.data.SubsetRandomSampler(train_ids)
    test_subsampler =
torch.utils.data.SubsetRandomSampler(test_ids)

    train = DataLoader(dataset_train,
                       batch_size=model_param["batch_size"],
                       num_workers=8,
                       sampler=train_subsampler)

    val = DataLoader(dataset_train,
                     batch_size=model_param['batch_size'],
                     num_workers=8,
                     sampler=test_subsampler)

    model = Sol(model_param)
    model.apply(reset_weights)

    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(),
lr=model_param['lr'])
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model.to(device)

    for epoch in range(0, num_epochs):

        train_loss = 0.0
        epoch_steps = 0
        model.train()

        for i, data in enumerate(train, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            probs = model(inputs.to(device))
            loss = criterion(probs, labels.to(device))

            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        model.eval()
        accuracy = 0
        predict = torch.tensor([])
        lab = torch.tensor([])
        valid_loss = 0.0

        with torch.no_grad():

            for data in val:
                inputs, label = data
                inputs, label = inputs.to(device),
label.to(device)

                output = model(inputs)
                loss = criterion(output, label)

                valid_loss += loss.item()

                pred = torch.sigmoid(output)
                l = label[:, 1]
                prediction = pred[:, 1]
                predict = torch.cat((predict,
prediction.detach().cpu()), dim=0)
                lab = torch.cat((lab, l.detach().cpu()), dim=0)

            p = np.where(predict > 0.4, 1, 0)

            accuracy = round(balanced_accuracy_score(lab.numpy(),
p), 5)  # calculate balanced accuracy for each fold

            print(
                f'Epoch {epoch + 1} \t\t Training Loss:
{round(train_loss / len(train), 5)} \t\t Validation Loss:
{round(valid_loss / len(val), 5)}')

            # early stopping technique
            if valid_loss < min_val_loss:
                epochs_no_improve = 0
                min_val_loss = valid_loss
                print(valid_loss)

            else:
                epochs_no_improve += 1

            if epoch > 5 and epochs_no_improve == n_epochs_stop:
                print('Early stopping!')
                early_stop = True
                break
            else:
                continue
            break

    if early_stop:
        print("Stopped")

    # Print accuracy
    print('Accuracy for fold %d: %d %%' % (fold, 100.0 *
accuracy))
    results[fold] = 100.0 * (accuracy)

# Print fold results
print(f'K-FOLD CROSS VALIDATION RESULTS FOR {k_folds} FOLDS')
```

```
sum = 0.0
for key, value in results.items():
    print(f'Fold {key}: {value} %')
    sum += value
print(f'Average: {sum / len(results.items())} %')

fold_accuracy = []
for key, value in results.items():
    fold_accuracy.append(value)

kfoldaccuracy = dict({"CNN": fold_accuracy})

# update file with accuracy distributions for each model

with open('boxplot.pkl', 'rb') as f:
    d = pickle.load(f)
d.update(kfoldacc)
with open('boxplot.pkl', 'wb') as f:
    pickle.dump(d, f)

fold_lstm = []
for key, value in results.items():
    fold_lstm.append(value)

kfold_lstm = dict({"CNN-LSTM": fold_lstm})

with open('boxplot.pkl', 'rb') as f:
    d = pickle.load(f)
d.update(kfold_lstm)
with open('boxplot.pkl', 'wb') as f:
    pickle.dump(d, f)
```

Appendix N. BiLSTM hyperparameter tuning (cnn-bilstm-hyperparam-tune.py).

```
mport torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import functional as F
import optuna
import os

import pickle
import json
import numpy as np

from sklearn.metrics import balanced_accuracy_score

# load data

sequences = torch.load('pre-processed data/padded_sequences.pt')
labels = torch.load('pre-processed data/onehot_labels.pt')
```

```
dataset_train = list(zip(sequences[0:69420], labels[0:69420]))
dataset_test = list(zip(sequences[69420:71419],
labels[69420:71419]))

train_size = round(len(dataset_train) * 0.9)  # size of training
set
valid_size = round(len(dataset_train) * 0.1)  # size of
validation set

# randomly split the given dataset randomly into the training set
and validation set of given lengths
train_set, valid_set =
torch.utils.data.random_split(dataset_train, [train_size,

valid_size])


def load_data(train=train_set, val=valid_set):
    return train, val


# load the optimal hyperparameters obtained during tuning in the
model
with open('parameters/best_parameters.pkl', 'rb') as f:
    model_param = pickle.load(f)

max_len = len(max(sequences, key=len))
model_param.update({"max_len": max_len})  # add the maximal
length of proteins to the parameters dictionary


# model


class Sol(nn.ModuleList):

    def __init__(self, maxlen, trial):
        super(Sol, self).__init__()

        self.maxlen = maxlen

        self.dropout_em = trial.suggest_float("drop_em", 0.1,
0.3, step=0.1)
        self.drop1 = trial.suggest_float("drop1", 0.1, 0.6,
step=0.1)
        self.drop2 = trial.suggest_float("drop2", 0.1, 0.6,
step=0.1)
        self.fc_layer = trial.suggest_int("fclayer", 8, 256,
step=8)
        self.stride1 = trial.suggest_int("stride1", 1, 2, step=1)
        self.stride2 = trial.suggest_int("stride2", 1, 2, step=1)
        self.out_size = trial.suggest_int("out_size", 16, 128,
step=16)
```

```
        self.lstm_layers = trial.suggest_int("layers", 2, 5,
step=1)
        self.hidden = trial.suggest_int("hid", 8, 256, 8)
        self.embed_dim = trial.suggest_int("embed_size", 50, 64,
14)

        self.dropout_embed = nn.Dropout(self.dropout_em)
        self.dropout = nn.Dropout(self.drop1)

        # kernel sizes
        self.kernel_1 = 2
        self.kernel_2 = 3
        self.kernel_3 = 4
        self.kernel_4 = 5
        self.kernel_5 = 6
        self.kernel_6 = 7
        self.kernel_7 = 8

        # Calculate the output of each convolution and add them.
        # We use this number further to calculate the amount of
input channels in LSTM.
        out_size1 = int(
            (self.embed_dim - self.kernel_1 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size2 = int(
            (self.embed_dim - self.kernel_2 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size3 = int(
            (self.embed_dim - self.kernel_3 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size4 = int(
            (self.embed_dim - self.kernel_4 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size5 = int(
            (self.embed_dim - self.kernel_5 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size6 = int(
            (self.embed_dim - self.kernel_6 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1
        out_size7 = int(
            (self.embed_dim - self.kernel_7 * (1 + self.stride1)
+ self.stride1) / (self.stride1 * self.stride2)) + 1

        self.lstm_input_size = sum([out_size1, out_size2,
out_size3, out_size4, out_size5, out_size6, out_size7])

        # convolutional layers configuration
        self.conv_1 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_1, self.stride1)
        self.conv_2 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_2, self.stride1)
        self.conv_3 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_3, self.stride1)
```

```
        self.conv_4 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_4, self.stride1)
        self.conv_5 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_5, self.stride1)
        self.conv_6 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_6, self.stride1)
        self.conv_7 = nn.Conv1d(self.maxlen, self.out_size,
self.kernel_7, self.stride1)

        self.lstm = nn.LSTM(self.lstm_input_size, self.hidden,
self.lstm_layers, batch_first=True, dropout=self.drop2,
                            bidirectional=True)

        self.embedding_layer = nn.Embedding(21, self.embed_dim)

        # pooling layers configuration
        self.pool_1 = nn.MaxPool1d(self.kernel_1, self.stride2)
        self.pool_2 = nn.MaxPool1d(self.kernel_2, self.stride2)
        self.pool_3 = nn.MaxPool1d(self.kernel_3, self.stride2)
        self.pool_4 = nn.MaxPool1d(self.kernel_4, self.stride2)
        self.pool_5 = nn.MaxPool1d(self.kernel_5, self.stride2)
        self.pool_6 = nn.MaxPool1d(self.kernel_6, self.stride2)
        self.pool_7 = nn.MaxPool1d(self.kernel_7, self.stride2)

        self.fc1 = nn.Linear(self.hidden, self.fc_layer)
        self.fc2 = nn.Linear(self.fc_layer, 2)

    def forward(self, x):
        x = self.dropout_embed(self.embedding_layer(x))

        x1 = self.conv_1(x)
        x1 = torch.relu(x1)
        x1 = self.pool_1(x1)

        x2 = self.conv_2(x)
        x2 = torch.relu(x2)
        x2 = self.pool_2(x2)

        x3 = self.conv_3(x)
        x3 = torch.relu(x3)
        x3 = self.pool_3(x3)

        x4 = self.conv_4(x)
        x4 = torch.relu(x4)
        x4 = self.pool_4(x4)

        x5 = self.conv_5(x)
        x5 = torch.relu(x5)
        x5 = self.pool_5(x5)

        x6 = self.conv_6(x)
        x6 = torch.relu(x6)
        x6 = self.pool_6(x6)
        x7 = self.conv_7(x)
        x7 = torch.relu(x7)
        x7 = self.pool_7(x7)

        union = torch.cat((x1, x2, x3, x4, x5, x6, x7), 2)
        # out - tensor of output features,
        # hs - tensor containing final hidden state for each
element,
        # cs- tensor containing final cell state for each element
        out, (hs, cs) = self.lstm(
            union)

        fc_output = self.fc1(torch.flatten(hs[-1], start_dim=1))
        fc_output = self.dropout(fc_output)
        fc_output = F.relu(fc_output)
        fc_output = self.fc2(fc_output)

        return fc_output


def objective(trial):
    model = Sol(max_len, trial)
    device = "cpu"

    if torch.cuda.is_available():
        device = "cuda"

    model.to(device)

    lr = trial.suggest_float("lr", 1e-8, 1e-3, log=True)
    batch_size = trial.suggest_int("batch_size", 16, 256,
step=16)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    trainlist, vallist = load_data()

    trainloader = torch.utils.data.DataLoader(
        trainlist,
        batch_size=batch_size,
        shuffle=True,
        num_workers=16)

    valloader = torch.utils.data.DataLoader(
        vallist,
        batch_size=batch_size,
        shuffle=True,
        num_workers=16)

    min_val_loss = np.inf
    n_epochs_stop = 5
    epochs_no_improve = 0
    early_stop = False

    for epoch in range(50):

        model.train()

        for inp in trainloader:
            sequence, labels = inp
            sequence, labels = sequence.to(device),
labels.to(device)

            optimizer.zero_grad()

            outputs = model(sequence)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        # Validation of the model.
        model.eval()
        accuracy = 0
        predict = torch.tensor([])
        lab = torch.tensor([])
        valid_loss = 0.0

        with torch.no_grad():

            for data in valloader:
                inputs, label = data
                inputs, label = inputs.to(device),
label.to(device)

                output = model(inputs)
                loss = criterion(output, label)

                valid_loss += loss.item() * inputs.size(0)

                pred = torch.sigmoid(output)

                l = label[:, 1]
                prediction = pred[:, 1]

                # prediction = torch.argmax
                predict = torch.cat((predict,
prediction.detach().cpu()), dim=0)
                lab = torch.cat((lab, l.detach().cpu()), dim=0)

        p = np.where(predict > 0.4, 1, 0)

        accuracy = round(balanced_accuracy_score(lab.numpy(), p),
5)
```

```
        trial.report(accuracy, epoch)

        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

        if valid_loss < min_val_loss:
            epochs_no_improve = 0
            min_val_loss = valid_loss

        else:
            epochs_no_improve += 1

        if epoch > 2 and epochs_no_improve == n_epochs_stop:
            print('Early stopping!')
            early_stop = True
            break
        else:
            continue
        break

    if early_stop:
        print("Stopped")

    return accuracy


study = optuna.create_study(direction='maximize',
study_name="accuracy")
study.optimize(objective, n_trials=25)

trial = study.best_trial

print('Accuracy: {}'.format(trial.value))
print("Best hyperparameters: {}".format(trial.params))

best_config = trial.params

with open('parameters/best_parameters.pkl', 'wb') as f:
    pickle.dump(best_config, f)
```

Appendix O. Statistical analysis (statistical-analysis.py)

```
import pickle
import scipy
import statistics
from statannot import add_stat_annotation
import pandas as pd
import numpy as np
from scipy import stats

import seaborn as sns
```

```python
from matplotlib import pyplot

with open('boxplot.pkl', 'rb') as f:
    data = pickle.load(f)

rf = data['RF']
cnn = data['CNN']
cnn_lstm = data['CNN-LSTM']


# obtain mean, variance and sd of the model

def stat(x):
    mean = statistics.mean(x)
    var = statistics.variance(x)
    std = statistics.stdev(x)
    print(f'mean=%.3f, variance=%.3f, standart devation=%.3f' %
(mean, var, std))
    return


stat(rf)

stat(cnn)

stat(cnn_lstm)


# check whether the sample is normally distributed

def normality_test(x):
    alpha = 0.05
    stat, p_value = scipy.stats.shapiro(x)
    print('statistics=%.3f, p_value=%.3f' % (stat, p_value))
    if p_value > alpha:
        print('Sample is normally distributed')
    else:
        print('Sample is not normally distributed')
    return stat, p_value


normality_test(rf)
normality_test(cnn)
normality_test(cnn_lstm)


# F-test to assest whether the variances between two models are
equal

def f_test(x, y):
    x = np.array(x)
    y = np.array(y)
    alpha = 0.05
    f = statistics.variance(x) / statistics.variance(y)
    dfn = x.size - 1
    dfd = y.size - 1
    p_value = 1 - scipy.stats.f.cdf(f, dfn, dfd)
    if p_value < alpha:
        print("variances are not equal")
    else:
        print("variances are equal")
    return f, p_value


f_test(cnn, rf)

f_test(cnn_lstm, rf)

f_test(cnn_lstm, cnn)


# statistical significance

def ttest(x, y):
    f, p = f_test(x, y)
    # we check whther the variances are equal, so we specify this
parameter in scipy.stats.t-test.
    # if variances are equal then two-saple independent t-test is
apllied,
    # otherwise we apply Welch's t-test
    if p < 0.05:
        equal_var = False
    else:
        equal_var = True
    stat, p_value = scipy.stats.ttest_ind(x, y,
equal_var=equal_var)
    print("p-value=", p_value)
    if p_value < 0.05:
        print("result is significant")
    else:
        print("result is not significant")
    return stat, p_value


ttest(cnn, rf)

ttest(cnn_lstm, rf)

ttest(cnn_lstm, cnn)

# build the boxlot to illustrate the variances of models

names = list(data.keys())
metrics = list(data.values())

for_df = []
```

```
models = list([names[0]] * 10 + [names[1]] * 10 + [names[2]] *
10)
distributions = list(metrics[0] + metrics[1] + metrics[2])
for i in range(len(models)):
    for_df.append(list([models[i], distributions[i]]))

df = pd.DataFrame(for_df, columns=["Model", "Balanced accuracy"])

sns.set(rc={'figure.figsize': (8, 7)})
ax = sns.boxplot(x="Model", y="Balanced accuracy", data=df,
palette="Set2", width=0.8,
                 showmeans=True)
add_stat_annotation(ax, data=df, x="Model", y="Balanced
accuracy",
                    box_pairs=[("CNN-LSTM", "RF")],
                    test='t-test_welch',
comparisons_correction=None, text_format='full', loc='inside',
verbose=5)

add_stat_annotation(ax, data=df, x="Model", y="Balanced
accuracy",
                    box_pairs=[("CNN", "RF"), ("CNN-LSTM",
"CNN")],
                    test='t-test_ind',
comparisons_correction=None, text_format='full', loc='inside',
verbose=5)

pyplot.savefig("graphs/boxplot.png")
pyplot.show()
```