Machine Learning for Molecular Engineering
3/7/10/20.01 (U) 3/7/10/20.C51 (G)
Spring 2025

Problem Set #1

Date: April 4, 2025
Due: Monday, April 7 @ 3pm ET

Instructions

- This problem set has two modeling tasks with several sub-questions. Some are marked grad version, which are required for graduate students (X.C51) but optional for others. Points for all students are in blue, while grad-only points are in orange. The total points are 75 for undergraduates and 100 for graduates.
- To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset1.ipynb (direct Colab link). You will need to use the data files here. Make your own copy of this template to save changes, by selecting "Save a copy in Drive". If you have not used Google Colab, you might find this example notebook helpful.
- **Important:** This problem set requires a GPU. Before you start in Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select a GPU as your hardware accelerator. Changing the runtime resets the notebook, so make this change before starting! Read Part 2.1 for additional help.
- Collaboration is encouraged and AI tools are permitted, but submitting work that is not your own is plagiarism. Any collaboration or assistance from an LLM (including utilties present within Google Colab) should be described at the end of your submission.
- Upon completion, submit your IPython Notebook `pset1.ipynb` to Gradescope. Ensure your submission includes *all* necessary code to be run without error, with all plots and outputs. Comments are encouraged; put answers to conceptual questions in Markdown/Text cells.

Part 1: Preliminary modeling

Background

Imagine if all diseases could be diagnosed with from a tiny drop of blood. While that day may still be far off, much progress has been made in searching for what are called *biomarkers*, molecules in the blood that are associated with particular diseases. Many studies use mass spectrometry to search for such diagnostic molecules. Biomarkers can be proteins, nucleic acids, lipids, or any of thousands of other chemical compounds that are found in the blood (see Figure 1).

Because many of these chemical compounds are products of metabolism, they are often called *metabolites*, and the detection of metabolites is called metabolomics. Recent studies have shown that metabolites can be predictive of human health conditions [2, 3]. In this problem, we will use two different classification methods to detect breast cancer from patients' metabolite data collected from human plasma/serum, following data processing steps found here [4].

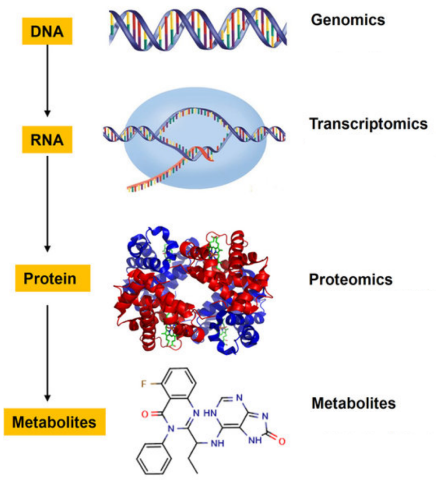

Figure 1: Figure adapted from Ref. [1]

Part 1.1 (5 points) Load and inspect the raw data

To perform supervised machine learning on vector-valued data, you need labeled examples $\{(\mathbf{x}, y)\}$, where $\mathbf{x}$ is a vector of input features and y the known label. Your goal is to train a model $\hat{f}$ that maps features to labels: $\hat{f}(\mathbf{x}) \approx y$. For diagnosing breast cancer from metabolite data, the metabolite signal is $\mathbf{x}$, and the binary label (positive or negative) is y, both provided as `.csv` files.

Task: We provide code utilizing pandas and numpy to load the data. Make sure you understand what each line of code is doing. **Briefly explain each line; then, use `X.shape` to report the number of samples and features per sample.**

Part 1.2 (5 points) Generate train/test splits

To fairly evaluate the performance, split your data into a training data set and testing data set. Only training data should be used to train the model; testing data are for unbiased evaluations of model performance. During training, the model should not have access to *any* information about the testing dataset, so you should not train (or preprocess!) on the testing data. Use `sklearn.model_selection.train_test_split` to *randomly* split your dataset into training and testing data with an **80%:20% ratio** (you should get two feature arrays, and two corresponding label arrays).

Task: Show your `train_test_split` code, print the shapes of your four variables, `X_train`, `X_test`, `y_train`, and `y_test` and ensure that the dimensions match your expectations.

Part 1.3 (5 points) Preprocess the data through scaling

Features in your dataset may have different units and magnitudes: for example, an atom's molecular weight ($1 - 294$ grams/mol) and covalent radius ($30 - 250 \times 10^{-12}$ meters) have different ranges. To avoid unfairly weighting features, we *standardize* all features to a consistent scale.

Standardization: Let N be the total number of features, and M is the total sample size. Let $X_{0,j}...X_{i,j}...X_{N-1,j}$ be the feature vector for the j^{th} sample (patient), where $X_{i,j}$ refers to the unnormalized abundance of the i^{th} metabolite in the j^{th} patient. The mean and standard deviation[1]

[1] scikit-learn and numpy use a normalization of M for the *population* standard deviation, rather than $M - 1$ for the sample standard deviation, which is numpy's default as well. pandas defaults to the *sample* standard deviation.

of each feature are calculated as:

$$\mu_i = \frac{1}{M}\sum_{j=1}^{M} X_{i,j} \qquad \sigma_i^2 = \frac{1}{M}\sum_{j=1}^{M}(X_{i,j} - \mu_i)^2 \tag{1}$$

Each feature is transformed under an affine mapping for the feature vectors $X_{i,j}$. We call the transformed feature vectors $X'_{i,j}$:

$$X'_{i,j} = \frac{1}{\sigma_i}(X_{i,j} - \mu_i) \tag{2}$$

Note that this procedure is invertible (meaning you can get the original feature vector back if you know σ_i and μ_i), meaning no information loss for your data.

Task: Use scikit-learn's `preprocessing.StandardScaler` to process your input features, `X`, into `X_train_scaled`, following the feature standardization above. Show the μ_i and σ_i of each feature are 0 and 1 respectively after scaling (use `np.mean` and `np.var`). Apply the same ScalerTransform to `X_test` to produce `X_test_scaled`. Note: the ScalerTransform should *only* be fit upon `X_train` and applied to `X_test`. Take one sentence to discuss why, and what information leak might occur if one were to fit the scaler upon both `X_train` **and** `X_test`.

Part 1.4 (10 points) Train and evaluate a Logistic Regression model

Now, you should be ready to train a logistic regression model and evaluate its performance on the test data. We will use simple model modules from scikit-learn, a machine learning library. We will use three handy evaluations for this task:

- **Confusion matrices** (reference): A visualization to stratify model performance by looking at positive and negative samples separately, and how many are correctly classified (true positives and negatives) vs. misclassified (false positives and negatives).
- **ROC Curve** (reference): The receiver operating characteristic (ROC) curve plots the true positive rate (TPR) against the false positive rate (FPR) at different decision thresholds. We usually report the area under the curve (AUC) of the ROC (AUC-ROC) for a scalar metric.
- **Precision-recall (PR) curve** (reference): This plots the precision (proportion of true positives to all predicted positives) against the recall (or true positive rate). We report the area under the PR curve (AUPRC) as a single metric from the PR curve. This evaluation is handy when the dataset is highly imbalanced, i.e. many more positives than negatives or vice versa.

Task: Train a logistic regression model on the scaled training data and evaluate it on testing data with scikit-learn's `LogisticRegression` class. For both the train and test datasets, generate plots for confusion matrices and the ROC curve with help of `plot_clf`, and report the AUC-ROC score. Finally, plot a histogram with `matplotlib.pyplot.hist` of the model coefficients (you can retrieve model coefficients from `model.coef_`, where `model` is your model) to get an understanding of the parameters learned for the dataset's features.

Part 1.5 (5 points) Introduce L1 regularization

In the previous part, you visualized the distribution of model coefficients. Now, we consider how these change after regularizing the model with L1 loss (we'll explore L2 regularization in Part 2).

Task 1: For this part, modify your logistic regression model to include L1 regularization using the keyword arguments `penalty='l1'`, `solver='saga'`. Use the same pre-processed training and

testing data from Part 1.1. Print the AUC-ROC score, plot the ROC curve, and plot the confusion matrix for both training and testing data.

Task 2: Probe the effect of regularization by plotting the histogram of the new model coefficients to find any qualitative changes. Comment on any differences between the two models' distribution of model coefficients, thinking specifically about the *geometric* interpretation of L1 regularization.

Part 1.6 (optional +2.5 points) Connect model coefficients back to metabolites

Task: The column of your feature set `X` which you loaded in Part 1.1 contains the chemical name for each metabolite. Based on the trained model from Part 1.4, identify the top 5 metabolites that are most correlated the most with a positive diagnosis.

Part 1.7 (5 points) Hyperparameter tuning the regularization parameter

To optimize your model further, one may tune the hyperparameters, which are parameters whose values are used to control some aspect of the model or the learning process, but is not optimized during the training. These might include the layer widths, the number of layers, the learning rate, optimizer, and more. This is done for a number of reasons: to prevent overfitting, make training most efficient, improve generalization, etc.

Task: For this problem, tune the regularization parameter by testing 4 provided values, and selecting the best model based on the *development set* (the internal test set) to select the parameters. Report the hyperparameter with the best validation performance, and that model's performance on the test set.

Part 1.8 (5 points) Train a Random Forest classifier

Besides Logistic Regression classifiers, another popular classification model architecture is Random Forests, which are ensembles of decision tree classifiers. Ensemble models can be often useful for mitigating overfitting and reducing the variance of your classification method.

To ensure our model's performance isn't just due to luck, we evaluate it on different splits of the data, known as *k-fold cross validation*. Following Figure 2, we effectively run the same experiment K times, each time using a distinct train/validation split. This yields K different performance values, which can be averaged to yield a more robust reflection of performance.

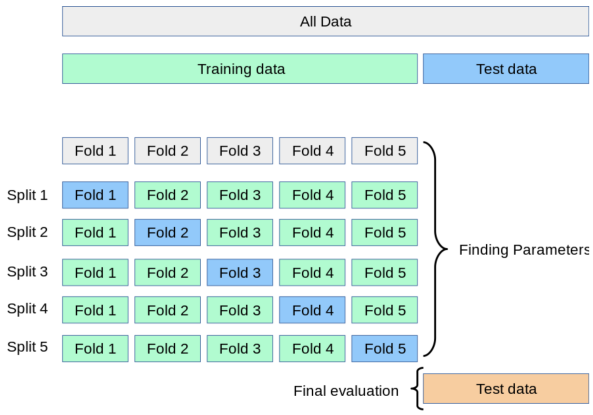

Figure 2: Overview of k-fold cross-validation. Source: scikit-learn.

Task: Initialize a Random Forest classifier using `RandomForestClassifier` from the scikit-learn library. Use the following parameter set: `{max_depth=2, n_estimators=20}`. Use the function `cross_val_score` to perform a 5-fold cross validation, using `scoring='roc_auc'` as the metric to report; report AUC-ROC scores via their mean and standard deviation.

To relearn the scaler transform in each fold, `sklearn` has the `Pipeline` API to chain together multiple steps in a machine learning model. Feed `cross_val_score` a `Pipeline` so that it correctly fits the scaler to only the training set within each cross-validation fold.

Part 2: Binding Affinity Regression

Background

This problem covers applying neural networks to predict binding of major histocompatibility (MHC) molecules to antigenic peptides; these interactions modulate immune response to diseases.

MHC Molecules and T Cell Response

As part of the adaptive immune system, T cells rely on MHC molecules to recognize particular antigens; below, we describe how MHC molecules work [5]. MHC molecules are cell surface proteins important for triggering adaptive immune response. There are two important classes of MHC molecules: class I molecules, found on almost every cell, and class II molecules, which are only present on specific antigen-presenting cells, like B cells. MHC Class I (or MHC-I) molecules present specific cytosolic peptides digested by intracellular proteolysis [6].

MHC-I molecules come in three classes: HLA-A, HLA-B, and HLA-C, each from a different gene. They consist of a 44-kD α-chain and a 12-kD β_2 microglobulin protein. Peptides will bind in a "peptide-binding" groove between the α_1 and α_2 domains. T cells recognize peptides only when bound to MHC-I as seen in Fig. 3, preventing the recognition of unbound peptides in the cytosol.

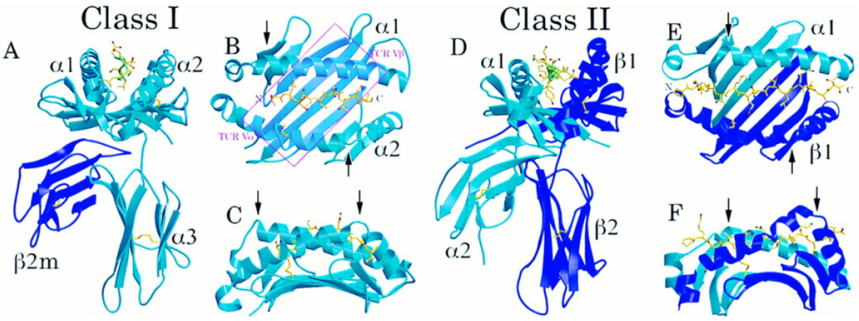

Figure 3: Structure of MHC molecule in complex with an antigenic peptide [5].

Each HLA class is highly structurally polymorphous; most of this variation is in the peptide binding groove, so each HLA allele corresponds to a unique MHC-I molecule, and thus unique peptide binding specificity. Ideally, a model for predicting binding specificity between MHC-I molecules and peptides must account for the structures (or sequences) of both the MHC-I molecule and the target peptide. Ultimately, predicting this interaction is particularly valuable for understanding how the immune system responds to a given disease or to a given protein therapeutic.

Data Generation and Processing

The data for this problem set is a subset of the IEDB, as collected from several quantitative assays used to measure the affinity of MHC-I molecules for specific test peptides and processed by a previous machine learning paper [6]. Graduate students interested in immunology-related final projects may find this database and related research helpful.

In practice, modeling binding affinity would require a model that understands both the structures (or sequences, as these should be sufficient to describe the task) of the MHC-I molecule and the peptide. To simplify the challenge, we have subsetted the dataset to look at only affinities involving the MHC-I molecule HLA-B15:17 and only provided peptides of length 9. This means that your model will only need to featurize peptide sequences of constant length.

Let's start with loading the dataset! You will be asked to use Pandas DataFrames. If you have not used Pandas before, please work through this tutorial.

Part 2.1 (5 points) Encoding amino acids into feature vectors

After loading the pandas dataframe, take a moment to inspect your data by looking at the rows (samples) and columns (features). The sequence of the peptide is stored in the column `Peptide` and the binding affinity is stored in the column `Binding Affinity`.

First, you will train a model that uses the amino acid sequence to predict binding affinity, which is a scalar property. How can you map the amino acid sequence like KSNRIPFLY to numeric features for training a model? You need to define a dictionary to encode amino acids into numeric features so that we can apply programs like linear regressions on these input features. We provide and load the list of amino acids in `amino_acids.npy` for you; these amino acids are the "vocabulary" you need to use to construct your bit vectors. One way to do is to use one-hot encoding to transform amino acids into bit vectors. For example, if we have amino acids K, S, and N, we can assign bit vectors of size 3 to fully encode this label information, as shown in Table 1.

Peptide Sequence	bit vector representation
'KS'	[1, 0, 0, 0, 1, 0]
'KN'	[1, 0, 0, 0, 0, 1]
'SN'	[0, 1, 0, 0, 0, 1]

Table 1: Example bit vectors of peptide sequences with simplified amino acid alphabet of "KSN"

Task: After reading the documentation for `preprocessing.LabelBinarizer()`, use it to transform the peptide sequences into one-hot encoded numpy arrays with the alphabet from `amino_acids.npy`. Next, convert the binding affinities to a `np.array` for use in regression. Your code should output `X_train` and `y_train`, and `X_test` and `y_test`. You will randomly split the first two arrays, `X_train` and `y_train`, into train/validation with an 80%/20% ratio in a later cell when DataLoaders are made. You'll have a separate test set that you should should process separately, and not combine or split further for the first parts of this problem.

Hint: Multiple approaches are possible. One way is to loop over rows (the peptide sequences) and then loop over characters within each row, converting each character using `preprocessing.LabelBinarizer()`.

Accelerating neural networks with GPUs

Scikit-learn provides common machine learning models but is limited in customization and speed of training. In contrast, PyTorch (along with TensorFlow, JAX, and CNTK) is designed for deep learning, allowing manual expressivity and efficient training through usage of GPU speedup. PyTorch

uses reverse-mode automatic differentiation (AD) to compute gradients automatically, making it easier to optimize model weights. It also supports training on a Graphical Processing Unit (GPU), which enables significantly accelerated training.

In this part, you'll call certain functions from PyTorch to construct a Multi-Layer Perceptron, which is composed of alternating linear affine and non-linear transformations (layers), and train it on a GPU in Google Colab (where PyTorch is pre-installed). After this exercise, we hope you will be comfortable building your own machine learning workflow using PyTorch by adapting this and other example code.

Optional reading The PyTorch Tutorial and Quickstart Guide are great companion resources to the functional PyTorch primer we present in this question. Likewise for debugging, we suggest leafing through this guide.

PyTorch setup: GPU basics, Datasets and DataLoaders

There are no points associated with this part, but spend some time understanding the cells in this section.

Setting up GPU usage: In Google Colab, go to `Edit` > `Notebook Settings`, and select T4 GPU under `Hardware accelerator`. You can also do this via `Runtime` > `Change runtime type` or by clicking the three dots in the top right corner. Verify GPU access using the provided code. In PyTorch, the main object is `torch.Tensor`, similar to `np.array`. In the second cell, confirm that you can move the sample tensor made to and from the GPU, with `.to(device)`.

Building `Datasets` and `DataLoaders` in PyTorch: To organize our data, PyTorch wisely modularizes the data into smaller chunks, or minibatches, that allow the model to process one at a time to avoid memory overhead. In PyTorch, data is stored in `torch.utils.data.Dataset` and batched with `torch.utils.data.DataLoader`.

From Part 2.1, you should already have the data featurized for the train and test datasets (with the latter being loaded from the holdout file), so we split the `X_train` and `y_train` data into a train and validation set for you. Take a moment to parse the `SequenceDataset` class construction that implements `Dataset` for this problem. Also parse the `DataLoader` instances that wrap each `SequenceDataset` object made, paying attention to the `shuffle=True` parameter, to avoid overfitting to minibatches; and the size of the minibatch, set with `batch_size=128`. You will need to know how to construct your own in PSET 2.

Part 2.2 (10 points) Define the MLP in PyTorch

We will now proceed to build a MLP using PyTorch's `nn.Module` class. The `nn.Module` class requires two methods: an `__init__` method to define components and layers, and a `forward` class where you define the network's computation, i.e. how to compute a prediction with the layers given input data. Many standard layers, such as `nn.Linear` affine layers and either the `nn.ReLU` or `nn.Tanh` layer for your activation function, are already made in PyTorch. You can use a `nn.Sequential` module to stack layers to automatically process your data in order. We ensure all layers are class attributes so PyTorch can handle auto-differentiation properly.

Task: We have provided the skeleton code for implementing your MLP in PyTorch as a `nn.Module`. Fill in the remainder of the code. Use three hidden layers, of widths 512, 256, and 128; and ReLu activation layers after each `nn.Linear` layer.

Part 2.3 (10 points) Implement functions for training and testing

Now with a defined model architecture and DataLoaders, we will set up the training. We initialize a model instance, `model`, and then an `optimizer` to perform backpropagation with a variant of stochastic gradient descent. Two options are stochastic gradient descent (`torch.optim.SGD`) or the Adam optimizer (`torch.optim.Adam`).

Task 1: Construct the optimizer by using the Adam optimizer, and pass the following as arguments: `model.parameters()`; a learning rate `lr` of `1e-3`; and L2 regularization with `weight_decay=0.01`. Then, define your loss function by using `MSE_loss()` to use mean squared error loss (important hint: make sure your input vectors to this call are the same shape; consider using `.view_as()`, `.flatten()`, or `.reshape()`).

Task 2: We have provided the skeleton code for training and validation loops, which is quite standardized for any training task. We use the `DataLoader` to loop over the training data's minibatches and use the model to predict the binding probabilities for each minibatch. Use the loss initialized above to compute a loss on that minibatch; this requires both your model output and the ground-truth values. PyTorch can then compute your gradients with `loss.backward()`. Model weights are then updated by calling `optimizer.step()`. Clear the gradients from the previous calculation (minibatch) by calling `optimizer.zero_grad()` at the start of each minibatch processing. Complete the `train()` function following these guidelines. Similarly, complete the `validate()` function, which uses the validation `DataLoader` and computes the loss on the validation set, though note: *no gradients need be computed or model weights changed.*

The `train()` and `validate()` functions implemented will operate on only one epoch, and should ultimately return a train loss and validation loss averaged over all minibatches in that one epoch.

Part 2.4 (5 points) Train the multi-layer perceptron

For regression tasks, one metric we can use to evaluate model performance is the coefficient of determination, or the R^2 score. It is defined as:

$$R^2 = 1 - \frac{\sum_i^{N_{\text{data}}} (y_i - \hat{f}(\mathbf{x}_i))^2}{\sum_i^{N_{\text{data}}} (y_i - \text{mean}(y_i))} \tag{3}$$

where i is the index for each sample, y_i is the target value, $\hat{f}$ is the model, and $\mathbf{x}_i$ is the feature vector. The larger the R^2 score, the more accurate the prediction is. If your prediction is perfect $R^2 = 1$. Note that metrics like mean absolute error (MAE) or mean squared error (MSE) can provide more meaningful and interpretable measures of performance.

Task: After ensuring you've initialized your model and optimizer with the hyperparameters listed in Table 2, train and validate your model for 250 epochs. Record the average train and validation loss for each epoch and plot these on a single graph (the plotting code is provided). Then, report the *test* R^2 of your trained model on the test data[1], and visualize your prediction for train and test data with a scatter plot. Finally, briefly comment on the `hidden_layers_sizes` values as they pertain to the MLP's intercepts and coefficients.

[1]: You'll need to convert your prediction from `torch.Tensor` to `np.array` with `.cpu().detach().numpy()`, then compute the R^2 score with `sklearn.metrics.r2_score`.

Part 2.5 Graduate (5 points) Compute model size

Task: Calculate the number of parameters used in your MLP given the provided '`hidden_layer_sizes`'. You can do this manually or iterate over `model.parameters()`.

`'hidden_layer_sizes'`	(512, 256, 128)
`activation`	nn.ReLU
`'alpha'`	0.01
`'solver'`	Adam
`'early_stopping'`	False
`'epochs'`	250

Table 2: Hyperparameters to be used

Part 2.6 (5 points) Chemical transferability of one-hot representations

We created a holdout/test set, which includes amino acids in new positions not seen in the training set, to test your model's performance.

Task: Load the holdout set and featurize the data with the label encoder from Part 2.1. Validate your MLP by computing the R^2 score, and visualize your predictions with a scatter plot. Briefly describe your observations: does your model generalize well to this new data?

Part 2.7 Graduate (10 points) Featurize amino acids with physical descriptors

Let's try using a more informative featurization, with physical descriptors of the amino acids. These physical descriptors were generated by the `peptide` package in Python and should provide physiochemical descriptions of various properties of amino acids.

Task: Load the data for physical descriptors stored in `amino_acid.csv` with the code we provided. Use all the numerical features to construct a new feature matrix; as you did for the one-hot features, construct your feature set by concatenating the individual amino acid features into a feature vector for the peptide. Split the data into a training (80%) set and a validation (20%) set; since we have continuous features again, rescale with `preprocessing.StandardScalar` as seen in Problem 1. Train on the training set with a MLP with the same hyperparameters we provided to you in Part 2.4 (you will need to tweak the number of input features). Plot a scatterplot of predictions on the train and validation datasets.

Part 2.8 Graduate (10 points) Chemical transferability of physical descriptors

Task: Report the R^2 score on your holdout/test set using the model trained in Part 2.7 and visualize your prediction with a scatter plot like you did before. Has your prediction on the holdout set improved? Briefly explain why.

References

[1] Gonzalez-Riano, C., Garcia, A. & Barbas, C. Metabolomics studies in brain tissue: a review. *Journal of Pharmaceutical and Biomedical Analysis* **130**, 141–168 (2016).

[2] Bar, N. *et al.* A reference map of potential determinants for the human serum metabolome. *Nature* **588**, 135–140 (2020).

[3] Evans, E. D. *et al.* Predicting human health from biofluid-based metabolomics using machine learning. *Scientific Reports* **10**, 1–13 (2020).

[4] Huang, S. *et al.* Novel personalized pathway-based metabolomics models reveal key metabolic pathways for breast cancer diagnosis. *Genome Medicine* **8** (2016).

[5] Chaplin, D. D. Overview of the Immune Response. *J Allergy Clin Immunol* **125**, S3–23 (2010).

[6] Reynisson, B., Alvarez, B., Paul, S., Peters, B. & Nielsen, M. NetMHCpan-4.1 and NetMHCIIpan-4.0: improved predictions of MHC antigen presentation by concurrent motif deconvolution and integration of MS MHC eluted ligand data. *Nucleic Acid Res* **48**, W449–W454 (2020).