Machine Learning for Molecular Engineering
3/7/10/20.C01 (U) 3/7/10/20.C51 (G)
Spring 2024

Problem Set #2

Date: February 22, 2024
Due: March 7, 2024 @ 11:59 pm

Instructions

- **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 don't wait until part 3 to do this.

- This problem set contains three questions, each containing sub-questions. Some questions have sub-questions marked as grad version. These questions are mandatory for those who have registered for the graduate version of this class (X.C51). They are optional for others. We've listed the points associated with each question in blue for the undergraduate and graduate version and orange for the graduate-only version. The undergraduate version is out of 75 and the graduate version is out of 100 points.

- Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. We highly suggest using Google Colab for this course and using this template for the second problem set. If you have not used Google Colab, you might find this example notebook helpful. After opening this template link, you should select "Save a copy in Drive" from the File menu. When you've completed the assignment, you can download the `*.ipynb` file from the File menu to upload to Canvas. Your submission should contain *all* of the code that is needed to fully answer the questions in this problem set when executed from top to bottom and should run without errors (unless intentional). Please ensure that your plots are visible in the output file and that you are printing any additional comments or variables as needed. Commenting your code in-line and adding any additional comments in markdown (as "Text cells") will help the grader understand your approach and award partial credit.

- Note that collaboration between students is encouraged, however every student must be responsible for all the work in their individual submission. I.e., discussing solutions and debugging together should be considered fruitful teamwork, however physically taking the keyboard of another student and directly entering code for their submission would be intellectually dishonest and as such is prohibited. You should first read through and attempt every problem before discussing with others. Please list the names of all collaborators at the bottom of your .ipynb submission file.

- To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset2.ipynb (direct Colab link). You will need to use the data files here.

Background

In this problem set, you will learn how to use neural networks to predict binding of major histocompatibility (MHC) molecules to antigenic peptides; these interactions help determine immune response to diseases. The following background section is optional, but you are encouraged to read through it to gain a better understanding of the importance of the data you will be working with on this problem set.

An Overview of the Immune Response

The immune system's response to an antigen (any foreign object that enters the body) can be broadly divided into two parts: the innate immune system and the adaptive immune system. The innate immune system encompasses anything that is not specific to the antigen in question, including physical barriers like the skin, mucus, and saliva, inflammatory molecules like chemokines and cytokines, and lymphocytes like natural killer cells. The adaptive immune system includes anything that can recognize a specific antigen and remember the antigen to trigger future, more rapid responses to particular threats. This includes B cells, which make antibodies in response to particular antigens, helper T cells, which recognize antigens identified by other antigen-presenting cells and coordinate the adaptive immune system, and cytotoxic T cells, which trigger cell lysis of cells infected by a given antigen. The adaptive immune system is reliant on MHC molecules to recognize particular antigens, so we will focus on understanding how MHC molecules work [1].

MHC Molecules and T Cell Response

MHC molecules are cell surface proteins important for triggering T cell responses to antigens. There are two important classes of MHC molecules: class I molecules, which are present on almost every cell in the body, and class II molecules, which are only present on specific antigen-presenting cells, like macrophages, B cells, and dendritic cells. MHC Class I (or MHC-I) molecules present specific cytosolic peptides digested by intracellular proteolysis; if a non-self antigenic peptide is presented, cytotoxic T cells will recognize it and trigger cell lysis. MHC Class I (MHC-II) molecules present antigens processed by antigen-processing cells, which can bind to CD4 receptors to helper T cells and trigger downstream responses from the adaptive immune system [2].

The interaction between MHC molecules and peptide binders is essential for the triggering and function of the adaptive immune system, so predicting this interaction is particularly valuable for understanding how the immune system responds to a given disease or to a given protein therapeutic. In this problem set, we will develop machine learning methods to perform these predictions given data gathered from assays describing the interaction between MHC molecules and small test peptides.

MHC-I Molecule Structure

MHC-I molecules can be divided into three classes: HLA-A, HLA-B, and HLA-C, each encoded by a different gene. (HLA stands for human leukocyte antigen, the human version of MHC). They are heterodimers consisting of a 44-kD α-chain associated with a 12-kD β_2 microglobulin (β_2m) protein. The α chain has three domains: $\alpha_1, \alpha_2, \alpha_3$; the β_2m domain interacts non-covalently with all three of these domains. Antigenic peptides bind to a groove between the α_1 and α_2 domains. The T cell receptor must contact both the antigenic peptide and the α_1 and α_2 domains, which ensures that T cells only recognize antigenic peptides that are bound to an MHC-I molecule, and not generic antigenic peptides floating around in the cytosol.

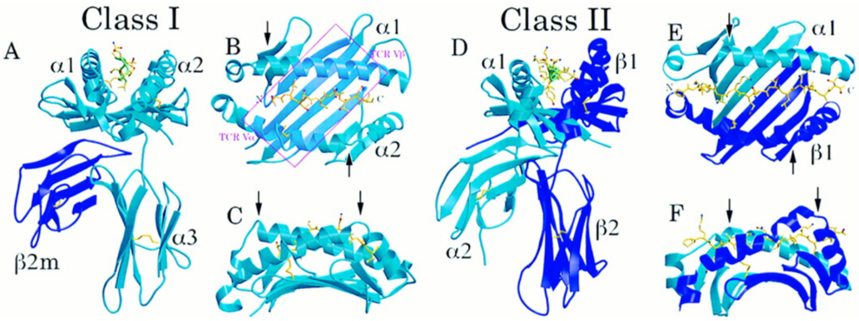

Figure 1: Structure of MHC molecule in complex with an antigenic peptide [1].

Each HLA class is highly structurally polymorphous, with potentially thousands of alleles of each. Most of the variation in the alleles is in the peptide binding groove, so each HLA allele corresponds to a different MHC-I molecule that has very different peptide binding specificity. A model that aims to predict binding specificity between MHC-I molecules and peptides must therefore account for the sequences of both the MHC-I molecule and the target peptide.

Data Generation

The data we will be using for this problem set is a subset of the IEDB, as processed by a previous machine learning paper [2]. This data is drawn from a number of qualitative and quantitative assays used to measure the affinity of MHC-I molecules for specific test peptides. We are not yet ready to build machine learning models that tackle the full dataset, so the data has been preprocessed and subsampled for our purposes. If you are a graduate student in the class and interested in pursuing immunology-related final projects, the full dataset is a good potential and paper linked above are good potential starting points.

Python packages required for this exercise

In this problem set, you will asked to use Pandas DataFrames. If you have not used Pandas before, please work through this tutorial.

Part 1: Modeling Binding Affinity

In this question, you are asked to predict the binding affinity of a MHC-I molecule to a given peptide. The binding affinities used here are measured by a large number of experiments and assembled into a single source by the IEDB. For convenience of comparing results from different experiments, they are converted to percentiles ranging from $[0, 1]$.

In practice, modeling binding affinity would require a model that understands both the sequence of the MHC-I molecule and the sequence of the peptide. We are not yet ready to tackle this full problem, so the data has been subsampled. Specifically, I have selected the MHC-I molecule HLA-B15:17 for you to work with, and only provided you with peptides of length 9. This means that your model will only need to featurize peptides of constant length.

Unlike pset 1, you will be performing regression tasks. For regression tasks, a metric we can use to evaluate model performance is the coefficient of determination, or the R^2 score. It is defined as:

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

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.

Sidenote: Since the output of this model is going to be a percentile, there are perhaps better ways to measure the performance and the loss function of the model, that are beyond the scope of what we are currently working on. Can you think of a better method?

You will first need to load the dataset using the code provided.

Part 1.1 (10 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. 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. We present simple examples on how you should encode peptide sequences on an abbreviated amino acid alphabet 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 vector representation of peptide sequences. We assume the amino acid alphabet is abbreviated to KSN for this example.

Use `preprocessing.LabelBinarizer()` to transform the peptide sequences into one-hot encoded numpy arrays. We provided the list of amino acids in `amino_acids.npy` which can be loaded using the code we provided to you. These amino acids are the “vocabulary” you need to use to construct your bit vectors. Next, convert the binding affinities to another array for use in regression.

Before you start, please read the documentation about `preprocessing.LabelBinarizer()` and understand the example code in the documentation. Your code should output two numpy arrays (`X` and `y`).

Hint: There are many ways to do it. One way is to loop over rows (the peptide sequences) and then loop over characters within each row, converting each character using `prepocessing.LabelBinarizer()`.

After obtaining bit vector representations for the amino acid sequences, randomly split your data into train/test with an 80%/20% ratio.

Part 1.2 Graduate (5 points) Modeling with ridge regression

Train a ridge regressor (linear regression with L2 regularization) to predict binding affinity (column name: `Binding Affinity`) using the dataset split from 1.1. Report your test score. Plot your prediction using a scatter plot (`matplotlib.pyplot.scatter`) for both train and test data.

Please first read the documentation on `sklearn.linear_model.Ridge` before training the model. We have also provided a code snippet for creating a scatter plot.

Part 1.3 (6 points) Modeling with a multi-layer perceptron

Using the same train/test split, train a model to predict binding affinity using a multi-layer perceptron with the following parameters in table 2. Visualize your prediction for train and test data with a scatter plot. Briefly comment on the meaning of (512, 256, 128) which is the input you will use for the `hidden_layers_sizes` option as it pertains to the intercepts and coefficients of the MLP. Also comment in one sentence on the difference between hyperparameters and optimized parameters with examples for both.

Please first read the documentation on `sklearn.neural_network.MLPRegressor` and understand the meaning of each hyperparameter before training the model.

‘`hidden_layer_sizes`’	(512, 256, 128)
‘`activation`’	’relu’
‘`alpha`’	0.16
‘`solver`’	’adam’
‘`early_stopping`’	False

Table 2: Hyperparameters to be used in Part 1.3

Part 1.4 Graduate (5 points) Compute model size

Calculate the number of parameters used in your MLP given the provided ‘`hidden_layer_sizes`’.

Part 1.5 (5 points) Chemical transferability of one-hot representations

We prepared a special holdout set for you to test your model performance. This holdout set contains amino acids that are not present in given positions in the training set. Load the holdout set using the code provided. Featurize the data with label encoder defined in Part 1.1. Validate your trained ridge regressor and MLP by computing the R^2 score for both of them. Visualize your prediction with a scatter plot for the two models you trained. Briefly describe explain what you observed. Do either of these models generalize well to this new data?

Part 1.6 Graduate (5 points) Featurize amino acids with physical descriptors

Now load the data for physical descriptors stored in `amino_acid.csv` with the code we provided. These physical descriptors were generated by the `peptide` package in Python and should provide physiochemical descriptions of various properties of amino acids. Use all the numerical features to construct a new feature set for prediction; 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. You need to split the data into a training (80%) set and a test (20%) set and then scale the data with `preprocessing.StandardScalar` as shown in problem set 1. Train on the training set with a MLP with the same hyperparameters we provided to you in Part 1.3. Visualize your prediction with scatter plots for both train and test data.

Part 1.7 Graduate (5 points) Chemical transferability of physical descriptors

Report the R^2 score on your holdout set using the model trained in Part 1.6 and visualize your prediction with a scatter plot like you did before. Has your prediction on the holdout set improved? Briefly explain why.

Part 2: Hyperparameter Tuning and Modeling Eluted Ligand Data

Physical descriptors for amino acids do provide performance improvements when some amino acids are missing from your dataset, but in general they are still imperfect and most prominent protein models use one-hot descriptions of amino acids. Another way to get around the problem of missing amino acids is to simply gather a better experimental dataset that does not have such egregious holes. We will assume that your experimental partners did this offscreen, so stack your train and holdout set together to form a new training set for the remainder of this problem set. You should use one-hot descriptions of the amino acids for the rest of the problem set, not the physical descriptors you used in Part 1.5.

Part 2.1 (15 points) Optimize your MLP architecture with HyperOpt

In PSet 1, you used grid search for model hyperparameters. However, grid search in many cases is not tractable due to combinatorial explosion. In practice, there exist more efficient algorithms to improve exploration efficiency. Tree of Parzen Estimators (TPE) is such a more advanced search method based on sequential model-based optimization. The details of TPE algorithm are beyond the scope of this class, but you are encouraged to read more about it in ref. [3]. TPE has been implemented in the `Hyperopt` package. In this part, you will try to use `Hyperopt` to explore the space of hyperparamters for MLP.

We have provided the skeleton code for you with predefined ranges of hyperparameters in the template file, but you will have to write your own `model_eval()` function which should take a suggested hyperparameter set (and `‘early_stopping‘` set to `False`) and return the resulting 5-fold cross-validation (CV) score on the training data you used previously, plus the holdout data from Part 1.4 (which you should have already added to your training set). Remember to use the one-hot features, not the physical descriptors from Part 1.5. Run the program for 40 evaluations. The `hyperopt` program might take a while to run, and you can check the progress bar to monitor the program executions. If it takes longer than a couple of hours, consult with your classmates or the TAs.

Finally, use the optimized hyperparameter set to train a MLP with all the training data again, and report your validation score on the test data you used previously. Have you seen any improvement?

Caveats What Hyperopt will do is to optimize your hyperparameters given the range you provide and minimize your loss function using `hyperopt.fmin`. Please note that because a higher R^2 indicate better performance, you should minimize the negative R^2 score with `fmin` (We have implemented this for you).

Part 2.2 Graduate (5 points) Applying MLPs to classifying eluted ligands

Another common type of data used for MHC binding is eluted ligand data. This dataset tells you whether a ligand is eluted along with a particular MHC molecule. Since the output variable here is whether or not a particular ligand is eluted, this is now a classification problem. As before, I've selected HLA-C07:01 as the only MHC-I molecule for you to work with and filtered to peptides of length 9. We will make an MLP-based classifier to tackle this problem.

Load the new dataset as directed and encode features as you did in Part 1.1. Please use the same MLP architecture and hyperparameters you obtained from hyperopt in Part 2.1. The class labels are in the `Eluted Ligand` column in your loaded dataframe. Randomly split your data into 80% train and 20% test data and train a classifier on your training data.

Please use the `metrics.ConfusionMatrixDisplay` class to visualize the classification result for both train and test data.

Before you start, please read documentation on `sklearn.neural_network.MLPClassifier` to understand the examples.

Part 3: Accelerating neural networks with GPUs

You may have noticed that the neural network as implemented by sklearn was both relatively inflexible and rather slow to train. Sklearn is great for using common algorithms and predefined model types, but cannot be used to build a fully custom model for end-to-end training. PyTorch (and other frameworks like TensorFlow, JAX or CNTK) is built for easily training and implementing deep learning models. The backbone of PyTorch is reverse-mode automatic differentation (AD), which is a programmatic framework to compute the gradients of your model output (a loss function) with respect to model parameters and inputs. Reverse-mode AD provides an easy way for machine learning researchers to construct models without having to think about explicit writing the gradient programs. These gradients are vital for optimizing (*i.e.* fitting) the weights efficiently. PyTorch also allows you to perform your training process on a GPU (Graphical Processing Unit), which will significantly accelerate computation and decrease training time.

In this part, you will re-implement the same neural network as in Part 2.1 using PyTorch's API and using a GPU for training. To speed up training, you should be sure to use a graphical processing unit (GPU) on Colab. PyTorch is pre-installed on Google Colab, so you can import it directly. We've provided some skeleton code in the template file, but you will of course need to fill in the details. 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. We suggest taking a moment to go through their examples to gain familiarity with basic PyTorch use cases and rely upon these official guides to fill in any details which may be not explicitly stated in this problem and in Problem Set 3, where you'll be training a more complicated neural network using a similar framework. Likewise for debugging we suggest leafing through this guide paying special attention to the advice surrounding size mismatch (when in doubt inspect shapes) and device mismatch. Note tensor operations generally require all tensors to be on the same device (cpu or cuda), you can check the device attribute of tensors as you debug and shuttle data to specific devices with '.to(device)'.

Part 3.1 (2 points) Request a GPU on Google Colab

In Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select GPU as your hardware accelerator. Check that you successfully requested a GPU using the code we provided. This will reset the notebook state so you may need to rerun a few cells. You do not need to recalculated the best hyparameters, just create a dictionary called best with the values you got in 2.1.

Part 3.2 (12 points) Build Datasets and DataLoaders in PyTorch

Code for processing data samples can get messy. From the perspective of good software engineering, it is good to modularize the dataloading procedure. In PyTorch, you will find two data-related classes, `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`, which are used to manage data when training a model.

From Part 2.1, you should already have the data featurized and split into a train and test set. Retrieve 10% of your training data as the validation set to be used to check for training convergence. Because training a deep neural network model takes time, so we won't perform cross validation or hyperparameter optimization as you did in previous psets. However, you might still want to do it in your final project if you decide to use a neural network based model.

Next, you need to construct your dataset and loader, with `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. A `DataLoader` is an object that iterates over your dataset and we provided an example of how to loop over your data. By using `DataLoader` you can utilize useful features such as memory pinning, multi-processing, and automatic batching when loading your data. You can also customize your data loading procedure to perform any on-the-fly preprocessing calculations.

The central object in Pytorch is `torch.Tensor`. They are very similar to the `np.array` objects you are already familiar with. You can easily transform a `np.array` to `torch.Tensor`, and send the `torch.Tensor` to a Nvidia GPU which is very fast at parallelized tensor operations. A nice tutorial can be found here.

We have provided you with a `SequenceDataset` class that implements `Dataset` for this problem set. Using this, fill in the skeleton code to construct your train, validation and test dataloaders with `batch_size=256`, using your train, validation and test datasets; remember to set `shuffle=True` to shuffle the dataset. Ensure your implementation is correct by iterating over your dataloader to retrieve batches of data (using our provided code). What is the shape of each batch? And how many batches are there in your dataset?

Part 3.3 (15 points) Define the MLP in PyTorch

There are many ways to define a neural network in PyTorch. For the purpose of this exercise, we would like you to use the `nn.Module` class to implement your MLP from Part 2.1. This method is a little bit more complicated than some other approaches but is more generalizable to the more complex architectures we will be discussing later in the course.

The `nn.Module` class requires two methods: an `__init__` method where you initialize any components of the neural network, and a `forward` class where you define the forward computation associated with the given network, i.e. how to compute a prediction given input data. Fortunately, many standard layers, including the feedforward layer used in the MLP, are already implemented in PyTorch, and you can call them to simplify your code. In particular, you will want an `nn.Linear` layer for each hidden layer and either the `nn.ReLU` or `nn.Tanh` layer for your activation function after each linear layer. You may also use a `nn.Sequential` module to stack layers; the output of the first layer within a nn.Sequential module will be fed as the input to the second, and so on. Look at our code snippets to understand how all of these layers work; you can use these to help you implement your MLP. Ensure that the layers of your model are attributes of your class, so PyTorch will know to auto-differentiate through the parameters of those layers.

We have provided the skeleton code for implementing your MLP in PyTorch as a nn.Module. Fill in the remainder of the code, remembering to use the optimal hyperparameters you found in Part 2.1.

Part 3.4 (10 points) Implement functions for training and testing

Now that you have defined the model architecture and a DataLoader, you can use them to train the model. First you need to choose an optimizer that takes the gradients computed from backpropagation to update model parameters through a variant of stochastic gradient descent. Depending on the hyperparameter you found as optimal in Part 2.1, you should use either stochastic gradient descent (torch.optim.SGD) or the Adam optimizer (torch.optim.Adam). The optimizer object takes the model parameters which you can retrieve with model.parameters(). You also need to specify a starting learning rate lr, which you should set to 1e-3. Do not implement L2 regularization; don't copy the alpha parameter into PyTorch.

Next, you will want to write a training loop. We have provided the skeleton code of a training loop; this is essentially standard, so you can copy it for virtually any neural network. We use the DataLoader to loop over the training data and use the model to predict the binding probabilities for each minibatch. You will now need to compute a loss; in this case, we want the mean-squared error loss implemented as nn.functional.mse_loss(); this requires both your model output and the ground-truth data. Once this is done, PyTorch will compute $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ to find the gradient step direction. This is done for you when you call loss.backward(). The optimizer receives the gradient and optimize your model by then calling optimizer.step(). Always remember to remove the accumulated gradient from a previous calculation (minibatch) by calling optimizer.zero_grad() at the beginning of the minibatch processing. Please take a moment to reflect on this procedure: all of the complicated gradient calculation/update is nicely wrapped by these function calls so that you do not need to think about the numerics behind the scenes. Complete the train() function following these guidelines. You will next want to complete the validate() function which uses the validation DataLoader and computes the loss on the validation set; it is otherwise very similar.

Looping over the entire training data set once is called an *epoch*. The train() and validate() function you will implement is only for one epoch, and should return a train loss and validation loss computed as averages over all the batches. Train and validate your model for 400 epochs.

Record the average train and validation loss for each epoch and plot these on a single graph (the code to do this has already been provided). Finally report your test R^2 of your trained model on the test data. You would need to transform your prediction from torch.Tensor to np.array to compute the R^2 score and use sklearn.metrics.r2_score.

References

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

[2] 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).

[3] Bergstra, J., Bardenet, R., Bengio, Y. & Kégl, B. Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NeurIPS 2011)*, vol. 24 (Neural Information Processing Systems Foundation, 2011).