Machine Learning for Molecular Engineering
3/7/10/20.C01 (U) 3/7/10/20.C51 (G)
Spring 2025

Problem Set #1

Date: March 27, 2025
Due: Monday, April 7 @ 3pm ET

Instructions

- This problem set contains two modeling tasks, each containing several 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. Points associated with questions for both undergraduates and graduate are marked in blue; and in orange for the graduate-only version. The undergraduate version is out of 90 and the graduate version is out of 110 points.

- **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 we recommend setting it from the start to do this! Read Part 2.2 for any additional instructions.

- Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `pset1.ipynb`. We highly suggest using Google Colab for this course and using this template for the first problem set. Be sure to make and save a copy of the notebook with your changes; the template will not save them for you automatically! 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 `pset1.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.

- **LLM policy**: The use of large language models (LLMs) such as ChatGPT, Claude, Bard, Perplexity, etc. as learning tools is also permitted. We will not explicitly teach you how to use these tools most effectively, but they are part of the modern toolkit available to us as

researchers and software developers. LLMs are currently most reliable as code "co-pilots" and may be particularly helpful for debugging pieces of code. For the functions/scripts we ask you to write, these LLMs are often capable of composing the full code from scratch; using them to prepare the entire solution verbatim will be considered plagiarism but using them to prepare snippets of code is permitted. You are responsible for understanding the solutions you submit for problem sets and are fully accountable for any mistakes or errors. You are discouraged from using LLMs to help with conceptual/interpretation questions, as doing so will not prepare you for your future career; the risk of nonsensical answers for these conceptual questions is also higher than it is for code generation. If you receive assistance from an LLM or a comparable tool (including any utilities present within Google Colab), you must describe how you have used it to prepare your code in your submission.

- 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.

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).

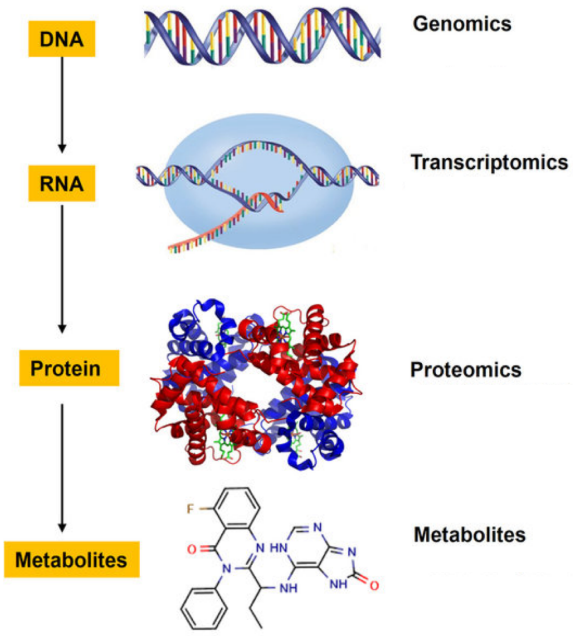

Figure 1: Figure adapted from Ref. [1]

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

a couple of machine learning algorithms to learn the correlation between certain metabolite levels and breast cancer using publicly available datasets.

Part 1: Preliminary modeling

You are receiving your first assignment as a data scientist: to detect breast cancer from patients' metabolite data collected from human plasma/serum. The detailed data description and collection protocols can be found here [4].

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

To perform a supervised machine learning task on vector-valued data, you will need a set of labeled examples $\{(\mathbf{x}, y)\}$ where $\mathbf{x}$ represents a vector of input features and y the known label. You are training a model $\hat{f}$ that maps the set of features to labels: $\hat{f}(\mathbf{x}) \approx y$. In the case of diagnosing breast cancers from metabolite data, the metabolite signal is your $\mathbf{x}$ and the binary labels of either positive or negative cases is your y. We have prepared these data for you in this question. Matrix $\mathbf{X}$ and vector $\mathbf{y}$ are provided as in `.csv` files which you can download using the `wget` command as implemented in the template notebook.

We have provided the code to load the data with pandas and transform it into numpy arrays. Make sure you understand what each line of code is doing. **Briefly explain what each line of the code is doing Use `X.shape` to report how many examples are in this dataset and the number of features for each data point.**

Part 1.2 (5 points) Generate train/test splits

To fairly evaluate the performance, you will need to split your data into two parts, a training data set and testing data set. Training data are used to train the model and the testing data are for unbiased evaluations of the 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. As the result of splitting, you will get two feature arrays, and the corresponding two label arrays. **Show your code for splitting the dataset, print the shapes of your four variables, `X_train`, `X_test`, `y_train`, and `y_test` and ensure sure that the dimensions match your expectations.**

Part 1.3 (5 points) Preprocess the data through scaling

In your data set, each feature *might* use different units with varying magnitudes. Consider a different dataset where we might describe an atom in terms of two features: its molecular weight (1 - 294 grams/mol) and covalent radius 30 - 250 $\times 10^{-12}$ meters)–these are wildly different ranges and scales! We don't want our model to unfairly weight different features without considering the relative scaling. Therefore, it is important to transform all features into a consistent scale.

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] of each feature are

[1]Note that scikit-learn uses 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, which you will use later in the course, defaults to the sample standard deviation.

calculated as:

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

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 (means that you can get the original feature vector back if you know σ_i and μ_i), and does not lead to information loss for you data.

You do not need to implement these calculations from scratch. Instead, use scikit-learn's `preprocessing.StandardScaler` to process your input features, `X`, into a new variable `X_train_scaled` by subtracting by the mean value and scaling by its standard deviation for each feature. You can use `np.mean` and `np.var` to check that each feature has a mean of 0 and variance of 1 (or close to it) after scaling. Also apply your scaler transform to `X_test` to produce the transformed test feature array, `X_test_scaled`. Note: the ScalerTransform should *only* be fit upon `X_train` and applied to `X_test`.

Show your code for transforming the arrays and print the mean/variance for each transformed feature. Discuss in a sentence why it is important to fit the scaler transform only upon `X_train` / what information leak would occur from the test to train dataset if one did 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 library of machine learning models.

Confusion matrices: For a classification task, the *confusion matrix* is a useful way to understand the model performance: we are interested not only in how many samples are classified correctly, but also in the number of samples that are mis-classified (and how). The confusion matrix includes information about *true positives* (positive samples we predict to be positive), *false positives* (negative samples we predict to be positive), *true negatives* (negative samples we predict to be negative), and *false negatives* (positive samples we predict to be negative).

These statistics can provide a more complete view of performance. For example, for coronavirus testing, we care less about the overall accuracy. Why? Let us say only 5% of the population are truly positive for Covid, an (abominable) statistician can simply predict negative for everyone without running any tests. This "lazy" model achieves 95% accuracy on but is clearly not useful. We might care more about those false negative cases because these may contribute more to the spread of the disease if people do not know that they might be contagious.

There are two other visualizations for the accuracy of binary classification that are commonly used. One is the receiver operating characteristic (ROC) curve which plots the true positive rate (TPR) against the false positive rate (FPR) at different decision thresholds. Read more about ROC score here. We usually report the area under the curve (AUC) of the ROC (AUC-ROC) to describe the performance of the classifier as a single scalar metric.

The other is the precision-recall (PR) curve, which plots the precision (or proportion of true positives among all predicted positives) against the recall (or true positive rate). Read more about the PR curve here. We report the area under the PR curve (AUPRC) as a single metric from the PR curve. It is frequently used 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. You should use scikit-learn's `LogisticRegression` class. Use the helper function `plot_clf` provided in the template notebook to plot your prediction results; it plots both the ROC curve and confusion matrix. **Outputs:**

- Show your code for running the Logistic Regression models with `scikit-learn`.
- Report the AUC score on both the training and test datasets.
- Generate plots for confusion matrices and the ROC curve for both training and testing data, using the provided helper functions.
- Plot a histogram of the model coefficients to get an understanding of the parameters learned for the dataset's features.

Note: When computing the AUC-ROC score, you should use the predicted probabilities rather than binary labels. To do that, use `roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])`. For more information, you can read the documentation here. We use predicted probabilities for the AUC-ROC since the ROC is drawn by computing true and false positive rates at multiple decision thresholds, and the predicted probability determines the class assigned to a sample at a particular decision threshold.

Part 1.5 (5 points) Introduce L1 regularization

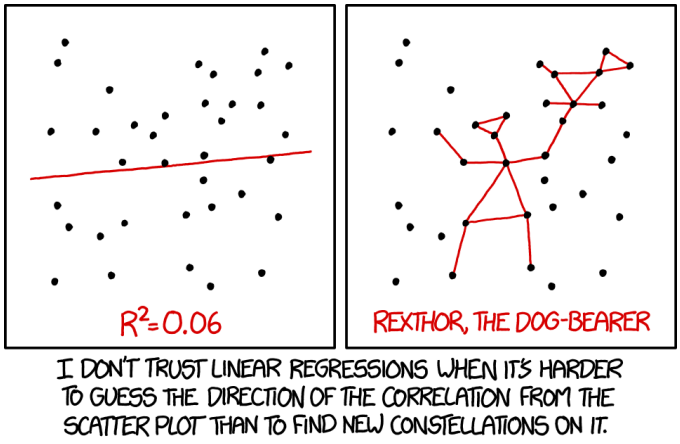

Figure 2: source: xkcd.com

In the previous part, you visualized the distribution of model coefficients, which should not be too large in magnitude. We will now consider how these change in response to regularizing the model with L1, or lasso regularization. In a later part of this pset, you'll explore L2/ridge regression. 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. **Show your code for all of these steps. Print the AUC-ROC score, plot the ROC curve, and plot the confusion matrix for both training and testing data.**

To further probe the effect of regularization, look at the histogram of the model coefficients to see if there are any qualitative changes. L1 tends to have a "sparsifying" effect on model parameters, i.e., it encourages a smaller fraction of them to be non-zero. **Use `matplotlib.pyplot.hist` to plot the distribution of model coefficients for both unregularized and L1 regularized training.** You can retrieve model coefficients from `model.coef_` if `model` is your classifier. **Comment on any differences in the distribution of model coefficients between the two models thinking specifically about the *geometric* interpretation of L1 regularization.**

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

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. You might want to repeat this analysis for multiple random splits to see if the most important metabolites are consistent.

Part 1.7 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 more rigorously report the model performance statistics, we need to make sure that the model does not merely perform well out of "luck". Therefore, we test it on multiple different splits of the data. To do that, you can run a K-fold cross-validation. This will divide the dataset into K equal-sized folds; in each experiment, we train on $K-1$ of these folds and test on the remaining one. Note one can think of cross validation as running the same experiment K times each time adopting a different non-overlapping test / train split to improve confidence in final results. From cross validation, we end up with K different performance metrics, which can be summarized as a mean and standard deviation. Use the function `cross_val_score` to perform a 5-fold cross validation using `scoring='roc_auc'` as the metric to report. **Show your code for running the cross validation and report the cross validated AUC-ROC score in terms of its mean and standard deviation.**

As before, we want to fit our scaler to only the training set in each cross-validation run. `sklearn` has a very useful API for chaining together multiple steps in a machine learning model (in this case the scaler normalization and then the logistic regression model); this is the `Pipeline` API, which you can read more about here. You will want to feed `cross_val_score` a `Pipeline` so that it correctly fits the scaler to only the training set within each cross-validation fold. A code snippet has been provided for you to learn how to set up `Pipeline` to do this.

Use the same data split to train a Random Forest classifier using `RandomForestClassifier` from the scikit-learn library. **Show your code and report the AUC-ROC results of a 5-fold cross validation (both mean and standard deviation).** Use the following parameter set: `{max_depth=2, n_estimators=20}`.

Problem 2 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 [5].

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 [6].

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.

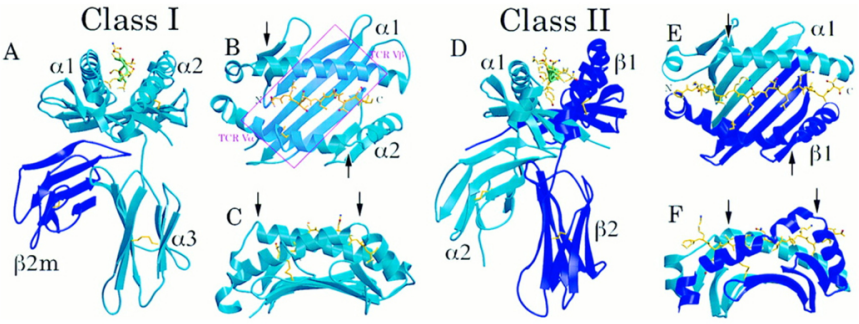

Figure 3: Structure of MHC molecule in complex with an antigenic peptide [5].

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 [6]. 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 2: 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:

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

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.

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 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. 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.

Accelerating neural networks with GPUs

You may have noticed that the architectures implemented by sklearn were both relatively inflexible and can be 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 2 and onward, 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 2.2 (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 T4 GPU as your hardware accelerator. You can also do this from `Runtime` > `Change runtime type`, or in the top right corner, hover over the three dots and select `Change runtime type`. 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, including import calls and code written for 2.1 (you don't need anything from Question 1 for Question 2, so do not (re)run anything there provided that your cell output has been preserved).

Run the `assert` call we provide for you to show you successfully requested a GPU.

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. In the second cell, we demonstrate how to make `Tensor` from a `np.array`; demonstrate that you can successfully move this tensor to and from the GPU.

Part 2.3 (3 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.

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 2.4 (10 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. 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. Use three hidden layers, of widths 512, 256, and 128; and ReLu activation layers after each layer.

Part 2.5 (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`.For this problem, additionally implement L2 regularization with an alpha of 0.01; this can be done by either writing a custom loss function or by modifying the optimizer to take in a `weight_decay` paramter of 0.01.

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.

Part 2.6 (5 points) Train the multi-layer perceptron

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 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`. 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.

'hidden_layer_sizes'	(512, 256, 128)
'activation'	'relu'
'alpha'	0.16
'solver'	'adam'
'early_stopping'	False

Table 2: Hyperparameters to be used

Part 2.7 Graduate (5 points) Compute model size

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

Part 2.8 (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 2.1. 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 2.9 Graduate (10 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 2.6 Visualize your prediction with scatter plots for both train and test data.

Part 2.10 Graduate (5 points) Chemical transferability of physical descriptors

Report the R^2 score on your holdout set using the model trained in Part 2.9 and visualize your prediction with a scatter plot like you did before. Has your prediction on the holdout set improved? Briefly explain why.

Part 2.11 (15 points) Hyperparameter Tuning with a grid search and cross validation

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 2.9.

A hyperparameter is a parameter whose value is used to control some aspect of the model or the learning process, and is not directly optimized during the training. These might include the layer widths, the number of layers, the learning rate, optimizer, and more. During this process of hyperparameter optimization, we will further split the training data into K folds for a K-fold cross-validation. The performance on the *development set* (the internal test set) for each fold will be used as the criteria to select hyperparameters. Once the best set of hyperparameters is identified, the model is trained on the full training set. The final model performance should be reported based on the 20% from the original split.

Perform an exhaustive grid search (that is, all combinations) over the provided hyperparameters and use `KFold` from the `scikit-learn` library to help you perform 5-fold cross-validation. Remember, you will need to "learn" and apply a scaling transform for *each training fold* that gets applied

to your validation fold. 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?

```
            layers: [[512, 256, 128], [256, 128, 64], [128, 64, 32]]
            learning_rate: [1e−3, 1e−4, 1e−5]
            weight_decay: [0.01, 0.001, 0.0001]
```

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).