Machine Learning for Molecular Engineering

Problem Set 1

Date: February 15, 2023
Due: February 24, 2023 at 10pm

Instructions 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 first 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.

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).

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 two types of 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

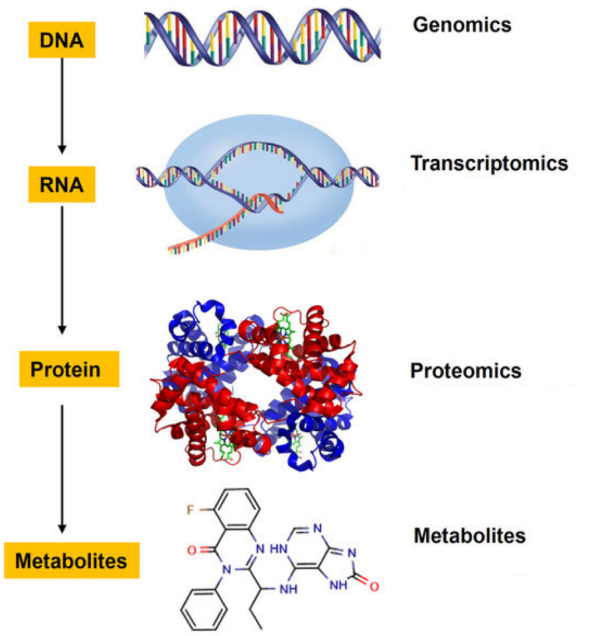

Figure 1: Figure adapted from Ref. [1]

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 of each feature are calculated as:

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

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 scaler transform 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 (15 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. 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.

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. **Show your code for running the Logistic Regressions and report the AUC score for both training and testing data. Generate plots for confusion matrices and ROC curve for both training and testing data using the provided helper functions.**

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) Perform a more thorough cross validation

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. Qualitatively supply some**

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.

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 (10 points) Quantify the effects of a reduced feature space

The representation we choose for each input x can have a significant effect on performance. Here, consider a situation in which our metabolomics pipeline wasn't able to quantify the abundance of all 128 species. What if, for example, we only had access to 25 species? Or 50? Loop over hypothetical feature subsets comprising a random 5%, 10%, 20%, 40%, 60%, 80%, and 100% subset of metabolites. This will require selecting random columns of `p1_X`, for which you can adapt this example which shows how to select random rows.

For each hypothetical featurization, obtain an average AUC-ROC remembering to scale the training set before fitting your model just as you did in Part 1.5. Because quantitative performance will depend on whether we get lucky or unlucky with what features are selected, run each regression 10 times with different feature subsets and calculate the average and standard deviation of AUC-ROC across feature subsets. **Use `matplotlib` to generate one plot showing these values (y-axis) as a function of the different feature subset sizes (x-axis) with error bars corresponding to the standard deviation.**

Part 2: Adding regularization to reduce overfitting

In this part, you will explore the effect of regularization on model performance. We will move on to a new dataset describing metabolite levels in patients with and without hepatocellular carcinoma, which is a common type of primary liver cancer. Your model training will be based on metabolite data collected from patients' body fluids. The data description and collection protocol can be found here. The data files you need are `liver_X.csv` and `liver_y.csv` in the GitHub repo here. You can download the dataset by using the provided `wget` commands.

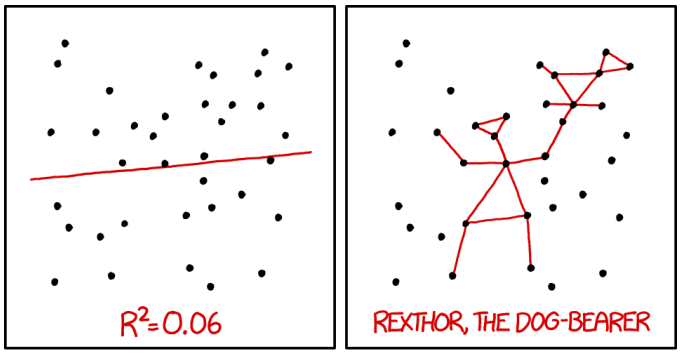

Figure 2: source: xkcd.com

Part 2.1 (10 points) Unregularized logistic regression

Follow the same procedure from Part 1 to load the data, split the data, scale the data, train a logistic regression model, and test its performance. Use an 80%/20% train/test split. Perform

logistic regression with `penalty='none'` to ensure there is no model regularization applied. **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. Provide a brief comment on your results—does the model perform better on training or testing data?**

Part 2.2 (5 points) Introduce L1 regularization

Now include L1 regularization in your logistic regression model using the keyword arguments `penalty='l1'`, `solver='saga'`. Use the same pre-processed training and testing data from Part 2.1. **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 2.3 (5 points) Introduce L2 regularization

Now include L2 regularization in your logistic regression model using the keyword arguments `penalty='l2'`, `solver='saga'`. Another argument you can change is `C` which controls the regularization strength. A smaller value in `C` means stronger regularization; the default value is 1. Set `C=0.1` for your L2 regularization. Use the same pre-processed training and testing data from Part 2.1. **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. Generate histograms for the distribution of coefficients in this L2-regularized case and again comment on any differences relative to the unregularized model thinking specifically now about the geometric interpretation of L2 regularization.**

Part 2.4 (10 points) Quantify the extent of overfitting as a function of training set size

In this problem, we want to explore the benefit of using L1-regularization from the "small data" regime to the larger data regime. At one extreme, imagine that when we train on just one sample, our model will simply memorize the result and will have nearly zero chance of generalizing to other samples. There will be a large gap between training and testing performance. As we add more and more data, it will tend to have better generalization, and therefore decrease the gap between training and testing performance. We are going to explore how regularization helps mitigate this generalization gap. **Run un-regularized and L1-regularized logistic regressions for the following random train/test splits:** $\{20\%/30\%, 30\%/30\%, 40\%/30\%, 50\%/30\%, 60\%/30\%\}$. Use `train_test_split` to generate these splits; remember to scale your data after generating the splits.

Because quantitative performance will depend on whether we get lucky or unlucky with each split, run each regression at least 10 times with different random splits of the same ratio; record the mean AUC-ROC score for both training and testing data. **For both unregularized and L1-regularized regressions, use `matplotlib` to generate one plot showing the mean**

train AUC-ROC score and mean test AUC score (y-axis) as a function of the different training data sizes (x-axis). Please also mark error bars corresponding to the standard deviation similar to Part 1.7. Comment on when the model is most prone to poor generalization (i.e., a large gap between training and testing performance): with or without regularization? For large or small training set sizes?

Part 3: Random forests and hyperparameter optimization

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. Use logistic regression as an example, the hyperparameter choices here include `{'penalty', 'C', 'solver'}`. These hyperparameters can have a significant effect on model performance. In this part, we will explore hyperparameter optimization through a *grid search* for a Random Forest classifier which you can learn how to use here. A Random Forest classifier has a fairly large set of parameters to optimize over, *e.g*, the number of trees (`n_estimators`) , the maximum depth of each tree (`max_depth`), etc. You can read the documentation to know all the possible hyperparameters.

Part 3.1 (5 points) Train a Random Forest classifier

Use the data from Part 2 and train a Random Forest classifier. **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}`.

Part 3.2 (15 points) Perform a hyperparameter optimization with a grid search

Randomly split the data into train and test sets with a 80%/20% ratio. We will use the training data to identify what hyperparameters might lead to a model that generalizes well. 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 over the following parameter grid with an internal 5-fold cross validation:

```
model__n_estimators: [20, 40, 80, 160, 320, 640, 1280]
model__min_samples_split: [8, 10, 12, 24]
model__max_depth: [2, 4, 8]
```

For the scoring function, use `scoring='roc_auc'`. As before, you'll need to fit your scaler to the internal training data, so use a `Pipeline`; the `model__` prefixes on each of the parameters tell the `Pipeline` which method to assign the parameters to. **Describe the total number of possible parameter combinations given the parameter ranges given above: how many different settings are there?** Evaluate model performance on the test data. **Show your code and report the AUC for both cross-validation and the held-out test data. What are the best hyperparameters identified through this approach?**

Everyone might get different answers from this optimization due to the randomness of data splitting, but the model should likely achieve better performance than your baseline model from

Part 3.1. You can follow one or more of the examples listed here. If you use the `GridSearchCV` class, you can set the `verbose` keyword argument to an integer greater than 0 to see more information during the calculation.

Part 4: Conclusion

Part 4.1 (5 points) Feedback survey

You will get 5 points for completing the survey posted on Canvas. This will help us improve future problem sets.

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).