QBioS 2023 Spring Workshop

Tutorial 1: Introduction to PyMOL and to Machine Learning

May 15, 2023

Contents

1 Introduction to molecular visualization

In this tutorial, we will outline some basic methods of visualizing proteins, which will be crucial to better understand the context of the machine learning analysis that we will conduct and discuss over the course of this workshop.

1.1 Setting up PyMOL

PyMOL is a molecular visualization software that is simple to download and use. Instructions for download can be found here. When you first load the software, if everything has compiled correctly, you should see the interface in Figure 1. By default, you should get a 30-day free trial of this software, so don't worry if you are prompted for a license upon initialization.

The top part of the screen functions similarly to a terminal; that is, if you type

```
PyMOL> pwd
PyMOL> ls
```

you should see the path to your working directory followed by the list of files in your current directory. Since we will be writing and saving images within this tutorial, I recommend changing directories since, by default, PyMOL will load in your home directory. You can use basic Linux navigation commands within this interface, but you won't be able to create or modify directories and files here. Create a directory that you want to work in (using your terminal or file viewer) and then change into that directory by typing:

```
PyMOL> cd /path/to/your/directory
```

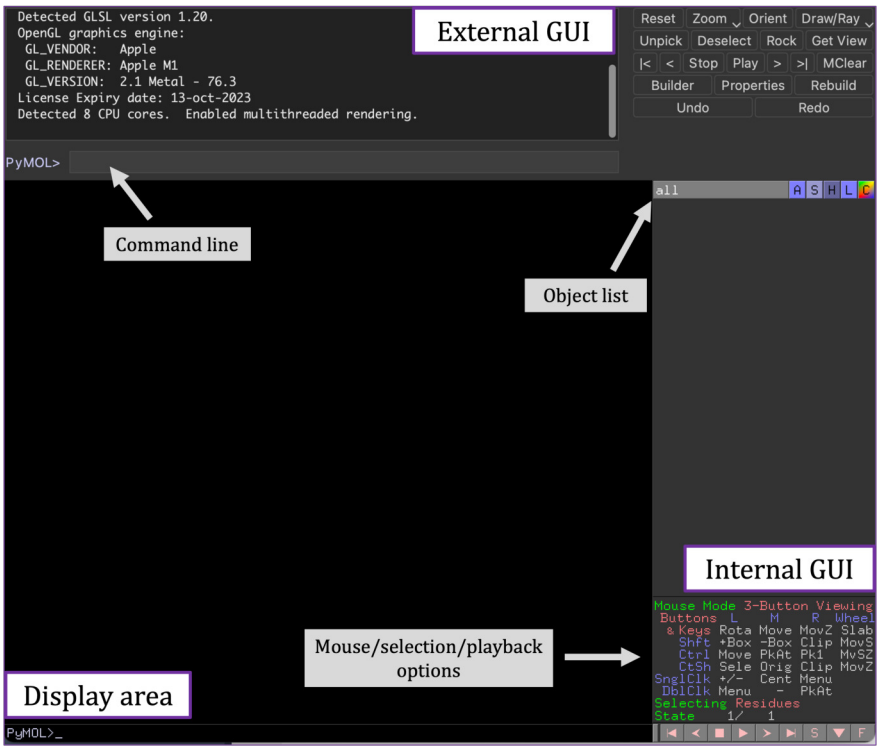


Figure 1: Annotated PyMOL user interface.

Table 1: Main PyMOL functions

Letter	Definition	Application
A	Action	general manipulation of objects
S	Show	change representation
H	Hide	hide atoms
L	Label	labelling
C	Color	change colors

1.2 Loading your first protein

As a basic introduction to PyMOL, we offer instructions to analyze a very basic protein.
In the PyMOL command line, type

```
PyMOL> fetch 1AKI.
```

This will load a protein structure, hen egg-white lysozyme, which is a very simple enzyme that we will look at before expanding to a more complicated viral system. This enzyme is (conveniently) quite short and functions to catalyze hydrolysis, i.e., the breaking apart of two molecules. Once loaded, a gray bar will appear on the right side of the screen with the Protein Data Bank (PDB) ID of the protein (in this case 1AKI). Each letter within this bar is associated with a different function (Table 1).

When loaded, the protein will be in a 'cartoon' visualization, where α helices and β sheets are shown as spirals and arrows, respectively. These are key components of protein secondary structure and are thus crucial to the structure and functionality of the molecule. For the sake of this tutorial, we will not go into more depth regarding the construction of these structural components, but it is important to note how amino acids can interact with each other via bonded and nonbonded interactions, which is dependent on the structure of the protein components.

Before we continue, it will be important for you to be able to easily manipulate proteins within the PyMOL interface. The bottom right of the screen (Figure 1) shows the current mouse mode and the options for editing or moving. Changing *mouse* $\rightarrow$ *1 button viewing* in the top toolbar can be useful to implement simpler mouse functionality for laptop trackpads. Otherwise, 3-button mouse features can often be implemented with the trackpad by using the control, command, shift, and alt keys. Once we load our first protein, experiment with navigating throughout the visual area using combinations of these keys in 3-button mouse viewing, and adjust to 1-button viewing if necessary. To begin, when using 1-button viewing, clicking anywhere and dragging will rotate the protein around its rotational axis. You can change this axis at any time by clicking anywhere on the protein and selecting $A \rightarrow$ *origin* or $A \rightarrow$ *zoom*. Holding control (PC/Linux) or command (Mac) and clicking and dragging will allow you to zoom in and out. Clicking alt (PC/Linux) or option (Mac) and dragging will allow you to translate your protein across the screen.

Click $C \rightarrow$ *by ss* $\rightarrow$ *choose any option here*. This will distinctly color the secondary structures and give you a better idea about where these different components begin and end. Your protein should now look like Figure 2. We can take a picture of this image by typing the following commands into the command window:

```
PyMOL> ray 1000,1000
PyMOL> png lysozyme.png
```

This will save a picture of the structure within the visual window as *lysozyme.png* into your previously chosen directory. The *ray* command will make an image of the specified height and width; in this case, we specified dimensions of 1000x1000 pixels. These values can be adjusted to image different-sized areas, but note that a very large area could take multiple minutes to render.

Click $S \rightarrow$ *as* $\rightarrow$ *licorice* $\rightarrow$ sticks. This will show explicitly the molecular structure of the amino acids within the protein. This representation, therefore, can contain information regarding which atoms are close to each other within the protein structure. This representation, however, does not contain any hydrogens

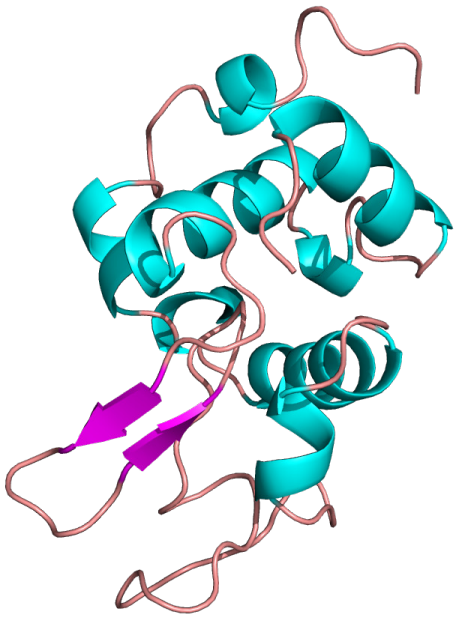


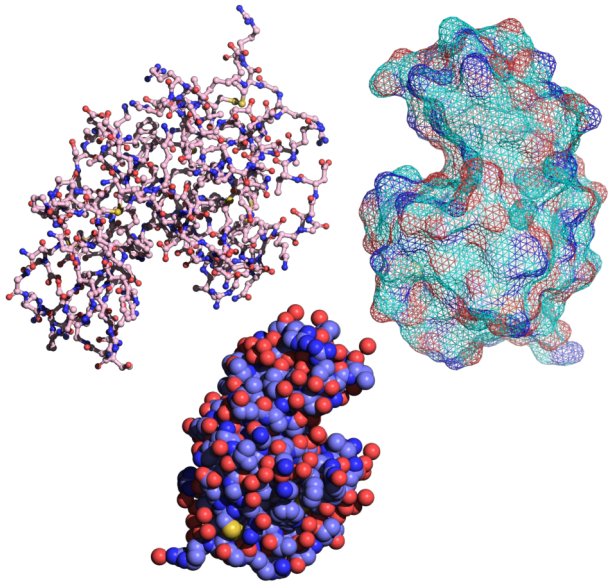
Figure 2: Hen egg white lysozyme (PDB: 1AKI).

(crystal structures typically do not include hydrogens). We can add these back in via the command $A \rightarrow$ *hydrogens* $\rightarrow$ *add*. You may notice that this command also makes it easier to see the water molecules in which the protein is solvated. To change back to the original viewing image, click $S \rightarrow$ *as* $\rightarrow$ *cartoon*. Lastly, to see a more holistic view of the entire protein, we can choose $S \rightarrow$ *as* $\rightarrow$ *surface*. This representation allows us to see the full globular form of the protein but isn't particularly useful for analysis of specific residue or atomic interactions.

1.2.1 Challenge Problem 1: Exploring multiple ways to visualize a protein

Generate three distinct visual representations of lysozyme. Use the Settings tab in the toolbar to explore various options, rather than relying solely on default settings. Consider different ways to depict the enzyme's shape, including any binding pockets (easier to see in surface or mesh representation), as well as more detailed atomic representations. If needed, you can include the earlier cartoon representation as one of the three images.

Challenge Problem Solution



It is important to be able to see both the shape of the protein and the approximate proportion of atoms. For instance, it is clear that the majority of the protein appears to consist of nitrogen, oxygen, and carbon atoms, but sulfur atoms are also scattered throughout, albeit sparsely. This would not be able to be seen when only looking at the cartoon view.

Click on the *S*, shown in Figure 3 or go to *Display* → *sequence* in the top toolbar. This will show you the amino acid sequence of the protein. Highlighting any of these sequence strings will immediately show it within the visual interface. Highlighted residues will be temporarily stored in the variable *'sele'*, which can be manipulated in a similar way to the full protein.

1.2.2 Challenge Problem 2: Observing amino acids

Identify and select the amino acids located within the beta-sheet of the hen lysozyme protein, represented by the two arrows in the cartoon depiction. Once you have visually identified your residues of interest, click on them to temporarily store them into the 'sele' variable. Copy these residues into a separate object (on your *sele* object, click *A* → *copy to object*), display them in a licorice format, and color by element *C* → *by element*. Additionally, add hydrogen atoms to the structure and show any hydrogen bonds between the residues. To reveal the hydrogen bonds, navigate to the *A* → *Find* → *Polar Contacts* → *Intra-Main Chain* option on your created object. Use the measurement wizard located in the top toolbar to calculate the length of the hydrogen bonds present in this section of the protein structure.

1.2.3 Tips for using the measurement wizard

The measurement wizard is useful for measuring distances, angles, dihedrals, and more. To measure the distance between two atoms click *Wizard* → *measurement* → *(click on the first atom)* → *(click on the second*

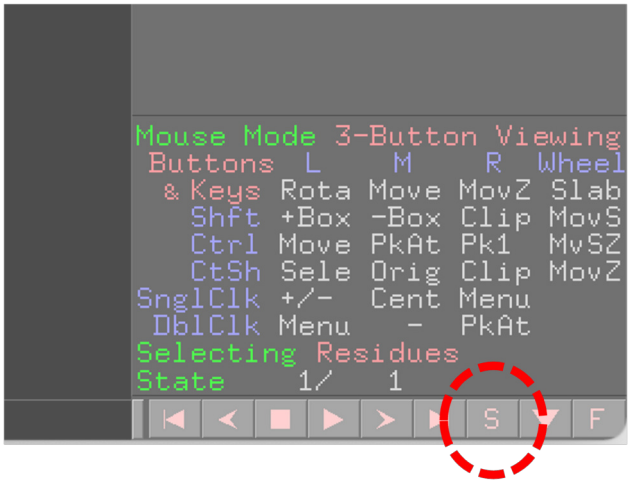
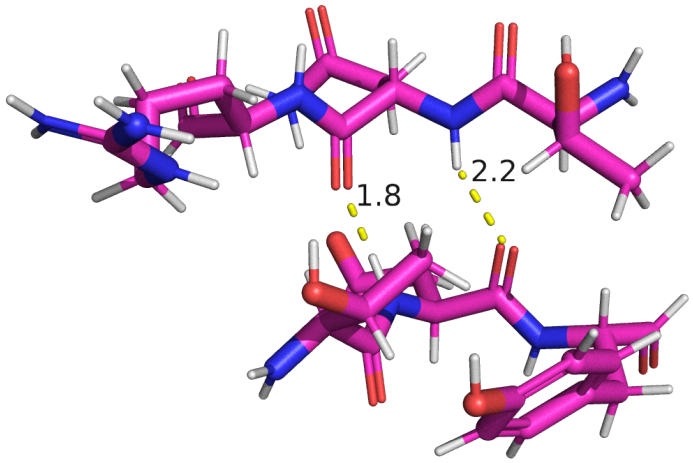


Figure 3: PyMOL mouse option descriptions. The red circle indicates the location of the sequence viewer. Clicking the button which says 'Residues' will also allow you to change the default selection from residues to atoms, chains, etc. Clicking the button next to 'mouse mode' will quickly allow you to change between 3-button viewing, 3-button editing, and 1-button viewing, among other settings.

atom). This will generate a new 'measurement' object, which shows a dashed line between the two chosen atoms and its length in angstroms, saved as 'measure01'. This object can be manipulated in a similar way to your protein; click the object name to hide the measurement, or delete it completely by clicking $A \rightarrow$ *delete*.

Challenge Problem Solution

This can be accomplished manually by clicking on residues and selecting (which can be good for visual understanding), but also quickly done via the command 'sele chain A and ss S' (it is convenient to use logical arguments within PyMOL). Although understanding hydrogen bonding is outside the scope of this tutorial, this exercise provides an example to visualize potential intermolecular interactions.



We will finish up this section by manipulating the representation of the enzyme around a certain point of interest. In doing so, it will also be very useful to learn how to manipulate temporary selections. Click on any residue within the protein. This will temporarily store the selection into the variable 'sele'. Then click *A* → *modify* → *around* → *residues within 12 angstroms*. Once done, a much larger number of residues should be selected. You can then change the representation of this selection. This technique will be very useful when analyzing amino acids near particular ligands, ions, or residues.

Throughout the first part of this tutorial, we have covered basic visual manipulation of proteins using PyMOL along with some brief basics on protein structure. The next sections will continue this trend and allow you to gain an intuition for observing potential intermolecular interactions and differences between proteins with similar structures.

1.3 Adding some complexity

We will now investigate a protein in a complex with a substrate. This tutorial will help you better understand intermolecular interactions that can occur between proteins of within a protein-substrate complex.

Create a new PyMOL window (or delete everything in your current session). In the PyMOL command line, type

```
PyMOL> fetch 4I8N
```

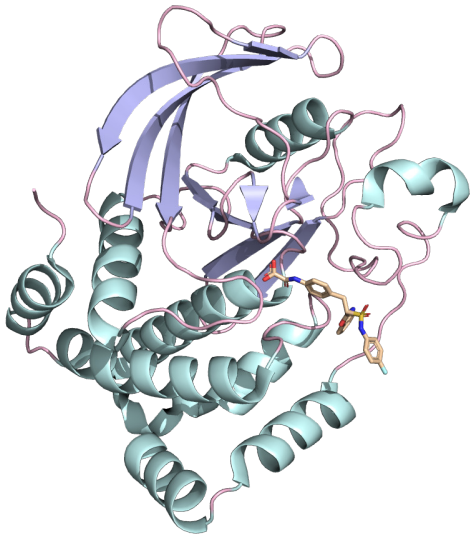


Figure 4: PTP1B protein (PDBID: 4I8N).

Color by secondary structure and observe your protein, which should look like Figure 4 (note: the colors used here are different from the default; feel free to explore different color palettes for your protein structure). This protein is called protein tyrosine phosphatase 1B (PTP1B), and functions to remove phosphate groups from molecules. For the sake of this tutorial, it is important only to know that this protein can be inhibited, and that an inhibitor molecule is also found within this pdb file. Typically, when loading in a pdb file containing both a protein and a ligand, the protein will be automatically visualized in 'cartoon' format and the ligand in 'licorice' format. This makes it easy to identify the ligand upon first loading of the molecule. PTP1B is an interesting protein because we can directly compare it to other proteins within the same PTP family. For instance, type into the command line

```
PyMOL> fetch 1L8K
PyMOL> align 4I8N, 1L8K
```

Here we have fetched another protein, a T-cell protein tyrosine phosphatase (TC-PTP), which is known to be homologous, or containing very similar secondary and tertiary structure, to PTP1B.[1] We have also aligned the two proteins so that we can better see their similarity. It is easier to view the similarities by first hiding any ligands and then making sure that each protein is colored one color (rather than having differentiated colors for secondary structure, atom types, etc.). Your alignment should look like Figure 5. When we align these proteins, PyMOL will immediately quantify how close the objects are via their root-mean-square deviation (RMSD), distance discrepancy, calculation; the lower this number, the closer the two structures. The RMSD for this alignment is 0.556 Å (0.0556 nm), which is very small, as we would expect. It is a useful exercise to pick out which components of the proteins appear the most dissimilar. With proteins that have a high sequence conservation, discrepancies are often due to missing residues within the pdb file (these will show as greyed letters within the sequence view in PyMOL). For instance, there is an α helix contained within the pdb file for PTP1B (4I8N) that is not present within the TC-PTP structure (1L8K). Can you identify this helix?

When looking at an enzyme, we often care about the residues within the active site which are able to interact with associated ligand(s). Although interaction distances can vary across proteins due to potential catalytic movements, we can make a general estimate of how close a residue needs to be in order to have

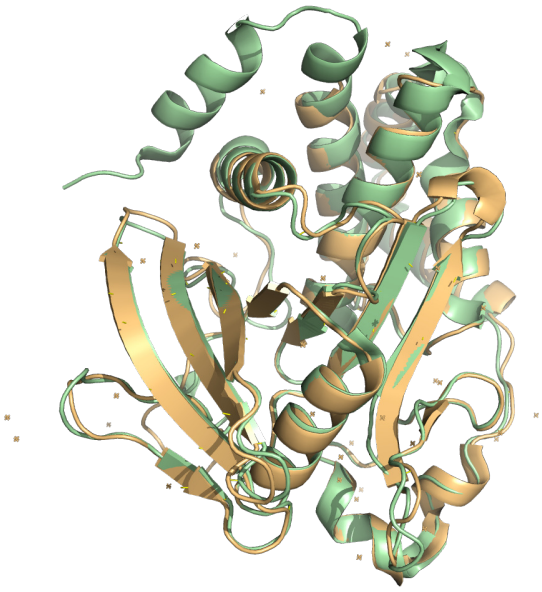


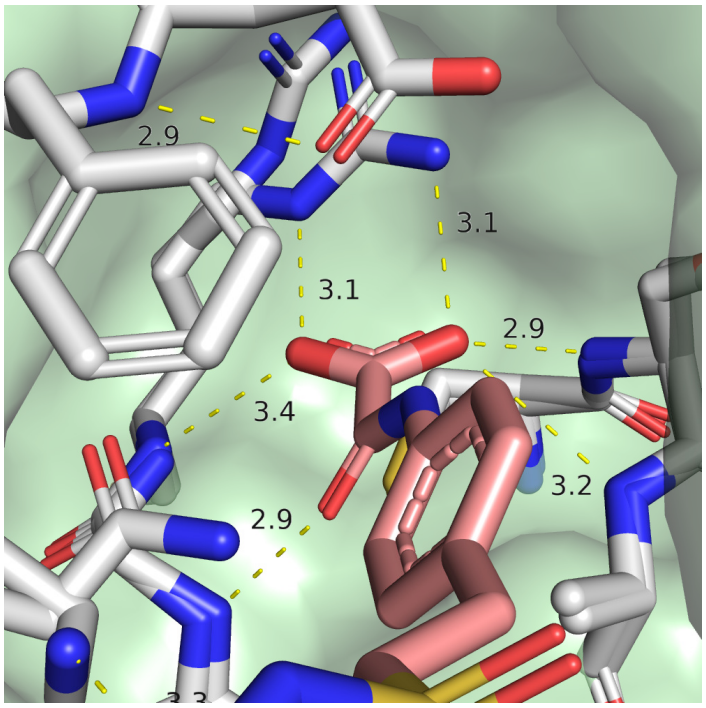
Figure 5: Alignment of PTP1B (green) and TC-PTP (yellow).

a potential interaction. For the sake of this tutorial, we will claim that any protein residue within 4 Å (1 Å $= 10^{-10}$ m) of the ligand could potentially interact, and then we will visually investigate to see which ones are more likely candidates.

1.3.1 Challenge Problem 3: Isolating a ligand

Copy the inhibitor (the residues shown in sticks when you initially fetch the structure) to a new object and rename. Change the color of this ligand and identify the amino acids within the PTP1B protein that could potentially interact with it by using the *modify around* command after selecting the ligand. Change the representation of these amino acids to licorice format. Find several residues that appear closest to the ligand (they will be on the inside of the protein). Use PyMOL to find any nearby polar contacts within the selection. Measure distances among the nearest neighbors. Save a png image of the ligand, its closest neighbors, and the corresponding atom distances.

Challenge Problem Solution



Machine learning can be used to provide an avenue for quantifying these potential interactions. As of right now, we are only predicting interactions based on static images.

Once we have a list of residues that may be interacting with the ligand, we should inspect them more thoroughly to determine what sorts of interactions could occur. Different amino acids exhibit different chemical properties. Although a more thorough explanation of chemical interactions is out of the scope of this tutorial, it is worth noting specifically that hydrophobic species will not interact with polar or charged species. Polar species will be more likely to form hydrogen bonds, while charged species can form electrostatic interactions. Determine what types of interactions can be possible within the PTP1B protein-ligand complex. A figure containing amino acid properties is provided for your reference (Figure 6).

1.4 Mini project: Understanding the SARS main protease

Now that you have sufficient skills in molecular visualization and a better idea for how atoms can interact within a larger protein, we will look at some further applications into understanding viral efficacy.

We will be looking at the main SARS-CoV protease. This enzyme is responsible for processing the majority of the replication proteins generated from SARS-CoV.[2] As a result, inhibiting the functionality of this enzyme could serve as a method for mitigating SARS-CoV. Through this mini project, you will analyze this protein and associated mutants to make observations on how single point mutations can alter the functionality of the key enzyme.

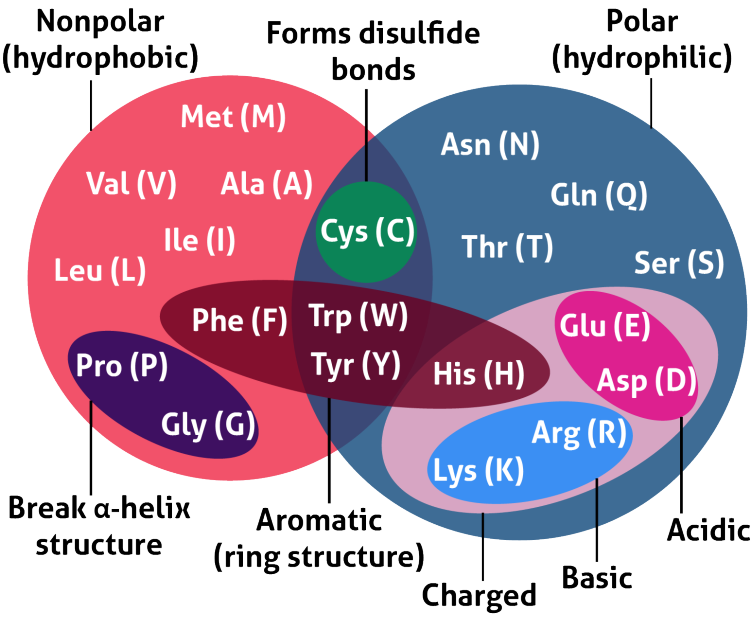


Figure 6: Venn diagram illustrating key chemical characteristics of amino acids. This figure can be used as a reference when looking into protein-protein interactions.

1.4.1 Challenge Problem 4: Characterizing mutants

Fetch the pdb 6Y2E and download the mutants of this protein, chosen previously,[2] from here. Find the mutations by comparing the sequences to the original structure, and use any PyMOL skills that you desire to determine how these mutations could affect the functionality of the protein. As a tip, the easiest way to quickly identify the differences visually would be to manipulate the coloring and representation of each protein, ensuring that all structures are aligned. Take a picture of each mutated residue aligned with the original structure (there is only one per structure). Can you determine where potential active sites may be located? If so, take a picture of the potential sites (it may be easier to change the protein to a surface representation for this). Label each mutant object in the format AYYYB, where A residue at position YYY within the wild type species is replaced by a mutated residue, B (ex: A611V would mean the alanine at position 611 was mutated to a valine).

Now that we have a grasp on how our mutants differ from each other, we will now look at how the protease (and its mutants) will interact with an inhibitor, nirmatrelvir, the key component of the drug Paxlovid.

1.4.2 Challenge Problem 5: Visualizing different interactions

Fetch the pdbs 8DZ2, 7U28, 7U29, and 7TLL. Determine which mutants these pdbs correspond to from the last challenge problems. Examine interactions within the active site between each protein and the inhibitor and determine whether it is possible to see an immediate difference between the potential effectivity of the inhibitor across the different proteins. Then, expand your gaze across the protein. These structures have a greater discrepancy than the previous ones, as they have been experimentally visualized rather than generated solely using PyMOL mutations. Take pictures of any differences among the mutants that are of interest to you. In summary, after aligning the structures, find areas that do not appear the same and determine if these differences can be directly traced to an amino acid mutation or rather if these differences occur despite identical residue sequences within these locations.

Lastly, we will explore the interactions between this protease and nirmatrelvir along with the mechanistic impact of the mutations that we have chosen.

1.4.3 Challenge Problem 6: Putting it together

Do some research and find how SARS interacts with nirmatrelvir and how mutating specific amino acids can alter these interactions (a good place to begin would be a recent study by Ullrich et al.[2]). Go back into the pdbs from earlier and try and pinpoint these locations/residues of interest. Do these locations line up with the discrepancies that you imaged for the last problem?

2 Introduction to machine learning

2.1 Background and fundamentals

Machine learning is a branch of artificial intelligence that focuses on teaching computers to learn and improve without explicit programming. Implementation of machine learning models allows us to design systems that can automatically identify patterns, make predictions, and take action based on data, rather than relying on human instructions. Later, we'll use one particular Python code suite *sklearn* to analyze the differences between SARS-CoV and SARS-CoV-2. But before we can handle that, we need to cover the basics of machine learning.

In this project, we will be using supervised machine learning, meaning that each of our data points will have a distinct label, which we can then differentiate. This process is analagous to the way we distinguish between a dog and a bird, where we use the features of each animal to determine the classification. Once the algorithm learns how to distinguish between different inputs, it can then predict the category of a novel input.

Classification algorithms can take a probabilistic approach, rule-based approach, or combination of both. The process of selecting a specific machine learning model is often done in a subjective manner, and the effectiveness of different models can vary depending on the relationships within the data. In this tutorial, we will explore three types of classification models: Logistic Regression model, Random Forest model, and Multilayer Perceptron.

This tutorial will make use of the scikit-learn Python package. Instructions for installation can be found here. We recommend creating a virtual environment to install this package, but that decision is up to the user. For reference, instructions for how to generate and manage virtual conda environments can be found here. You should also have installed the matplotlib, numpy, and pandas packages.

2.2 Decision tree model

A random forest model is an estimator that fits a number of decision tree classifiers on sub-samples of a dataset, averaging to increase the accuracy of the predictive fitting. Therefore, in order to understand the random forest classifier, we first need to investigate decision trees more simply. A decision tree is a basic rule-based classifier where data is organized into a tree-like structure before classes are assigned to all data points in each leaf node of the tree. To construct the tree structure, you can use dataset features to generate yes/no questions (rules) and repeatedly split the data into smaller subsets until you don't have any more rules to apply or every instance is assigned to a specific class.

The performance of the tree is quantified using a metric called Gini impurity where a lower Gini impurity corresponds to greater variance captured using the given classifier.

2.2.1 Example: Know your flowers

Here, we examine an example created from sci-kit learn. This example will produce a graphic illustrating how a decision tree is generated from a dataset.

```
    # Load libraries
from sklearn import tree

# Load data
from sklearn.datasets import load_iris
iris = load_iris()

# Divide the data into features and target
X, y = iris.data, iris.target

# Building decision tree model
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
```

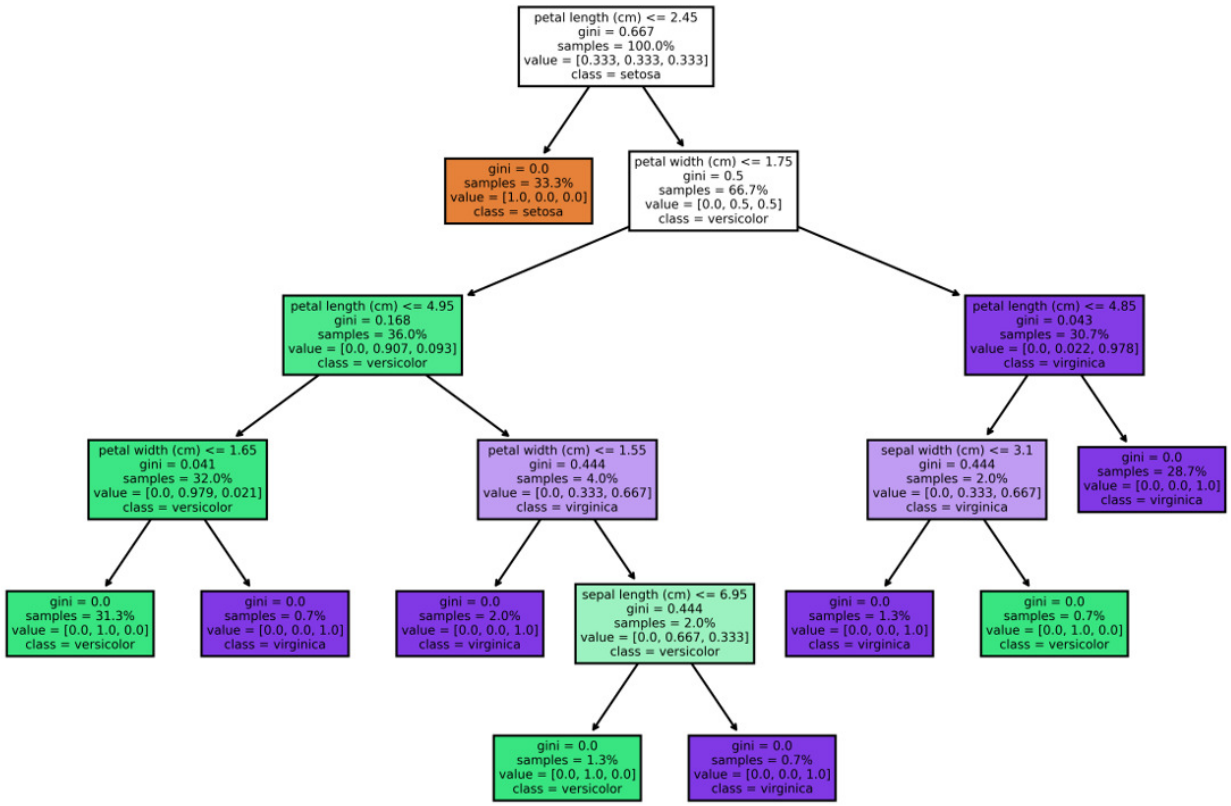


Figure 7: Decision tree sourced from sklearn's *iris* data.

```
# Visualize
plt.figure(figsize=(10, 7))
tree.plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.
                                                target_names, proportion=True)
plt.show()
```

From these results, we can see that conditions of petal and sepal length/width were used to make decisions. Picking the best split is not an easy task and will significantly affect the performance of the classifier. Solving this problem is out of the scope of the tutorial; here, we are only focusing on the concept of the decision tree algorithm to prepare for the following explanation of the Random Forest model.

2.3 Random Forest classifier

A random forest (RF) classifer is composed of multiple decision trees. The classification result of one random forest is decided by the majority vote from all decision trees (Figure 8). During the training process, the decision tree algorithm iteratively searches features and determines a threshold value to split the dataset in such a way that the maximum amount of information possible is conveyed through each node.

Scikit-learn provides dozens of built-in machine-learning algorithms and models, called estimators. Each estimator can be fitted to some data using its fit method. For example, follow along with the code below to set up a basic Random Forest Classifier.

```
    from sklearn.ensemble import RandomForestClassifier
```

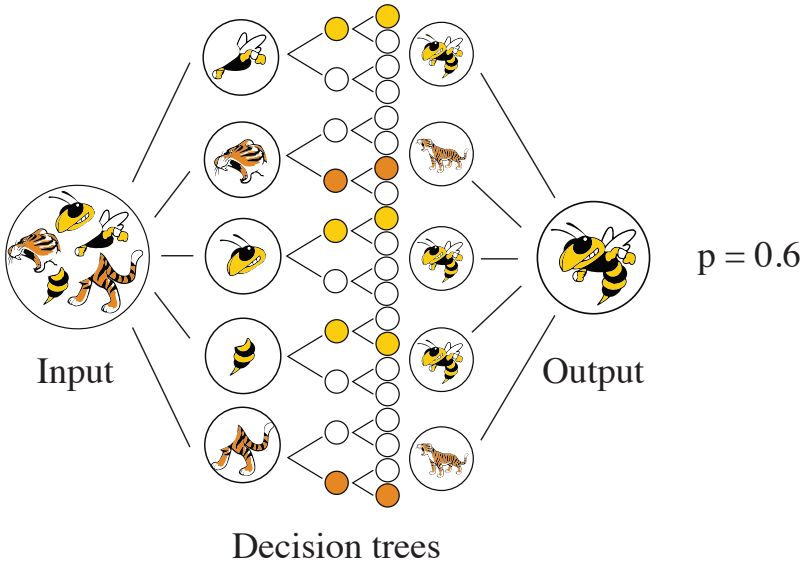


Figure 8: Simplified representation of a random forest classifier.

```
clf = RandomForestClassifier(random_state=0)
X = [[ 1,  2,  3], [11, 12, 13]]  # 2 samples, 3 features
y = ['0', '1']  # classes of each sample
clf.fit(X, y)
```

Here we define a data set, X, where the rows represent samples and the columns features of the sample. Here the samples are the vectors [1, 2, 3] and [11, 12, 13]. We then define which classes/groups these samples belong to by defining y, where we say the first sample belongs to '0' and the latter belongs to '1'. This information is now used to fit and train a RF model, clf. The argument random_state=0 is used to ensure that every time we run this function, we will obtain the same result by using the same random seed, which will be helpful for the extent of this tutorial. We will adjust this in the next tutorial in order to increase the resulting accuracy of our model.

Now that we have a trained model, we can use it to predict a classification output for new entries. First, let's test our model to make sure that it returns the same y given our initial X:

```
clf.predict(X)  # predict classes of the training data
----
Out: array(['0', '1'])
```

As desired, given our training matrix X, our classifier will return y precisely. Since we have validated our model, we can input other datasets to see what the predicted classifications would be. For instance, let's use a very similar, but noticeably different, matrix as input.

```
clf.predict([[4, 5, 6], [14, 15, 16]])  # predict classes of new data
----
Out: array(['0', '1'])
```

As you may have expected, the [4,5,6] data entry returned 0, and the [14,15,16] data entry returned 1. However, this could be for any number of reasons. Maybe the estimator assumed that single-digit numbers should return 0. Maybe it wanted the numbers to be greater than 7 for being classified as 1. In any case, this model isn't going to be very good for a larger variety of datasets. This is primarily because the model was trained on only two data points. To help resolve this problem, we will train a new model using a larger

dataset.

2.3.1 Challenge Problem 1: Bigger numbers

> Create a random 2D array of data points between 0 and 100 (we suggest using the numpy randint function). We want each row within our resulting dataset to have three features, following the same format as the previous training data we have been using. However, instead of only two samples, increase this number to 2000 (you should end up with a matrix consisting of random numbers of size 2000×3).
>
> Then, create a classification array with the value 'True' if a data point's first value is greater than the second, and 'False' otherwise. The third value for each data point is a red herring! Once you have fit your model, create a new set of random data points (of the same size as your training data) and test them against the model for accuracy!
>
> **BONUS:** Now switch your random integer number generator for one that produces random numbers pulled from a Gaussian (normal) distribution. Which estimator yielded better accuracy?

Challenge Problem Solution

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier

### Creating a random set of 3000 data points, each using 3
### integer values between 0 and 100
data = np.random.randint(0, 100, size=(2000, 3))
# data = np.random.normal(50, 10, size=(2000, 3))

y = data[:,0]>data[:,1] ### Classifier
clf = RandomForestClassifier(random_state=0)
clf.fit(data, y)

### Testing new data ###
dataTest = np.random.randint(0,100,size=(1000,3))
model_prediction = clf.predict(dataTest)

actual_result = dataTest[:,0]>dataTest[:,1]
difference = model_prediction[:] == actual_result[:]
accuracy = np.count_nonzero(difference)/np.size(difference)
print('Accuracy of classifier trained on uniform random data =', accuracy)

#Gaussian bonus problem
ata2 = np.random.normal(50, 10, size=(2000, 3))
y2 = data2[:,0]>data2[:,1]
clf2 = RandomForestClassifier(random_state=0)
clf2.fit(data2, y2)
model2_prediction= clf2.predict(dataTest)
difference = model2_prediction[:] == actual_result[:]
accuracy = np.count_nonzero(difference)/np.size(difference)
print('Accuracy of classifier trained on Gaussian data =', accuracy)
```

If you did the bonus, you'll find that your estimator may have yielded a decently high accuracy ($> 95\%$) when using the random distribution method. However, you probably found that your test had lower accuracy when using a normalized Gaussian distribution ($\approx 85\%$). Further quantifying this level of accuracy will be the focus of the next challenge problem.

2.3.2 Challenge Problem 2: Quantifying accuracy dynamics

Using the code you developed above, determine the average accuracy of the random forest classifier over 50 trials for a sample data size ranging from 2 to 100 data points. Graph the results of this analysis. It may be helpful to define the accuracy output from the random forest model within a contained function.

Challenge Problem Solution

```
def RandomForest(data):
    y = data[:,0]>data[:,1]
    clf = RandomForestClassifier(random_state=0)
    clf.fit(data,y)

    dataTest = np.random.randint(0,100,size=(1000,3))
    prediction = clf.predict(dataTest)

    trueResult = dataTest[:,0]>dataTest[:,1]
    difference = prediction[:] == trueResult[:]

    Accuracy = np.count_nonzero(difference)/np.size(difference)

    return(Accuracy)

numtrials = 50
sizerange = np.arange(2,100)
accuracymat = np.zeros((len(sizerange),numtrials))

for i_ind, i in enumerate(sizerange):
    for j in range(0,numtrials):
        data = np.random.randint(0,100,size=(i,3))
        accuracymat[i_ind,j] = RandomForest(data)*100


plt.plot(accuracymat.mean(axis=1),linewidth=3,color='k')
plt.xlabel('Number of sample datapoints',fontsize=14)
plt.title('Random Forest Classifier Accuracy',fontsize=16)
plt.ylabel('Accuracy, %',fontsize=14)
```

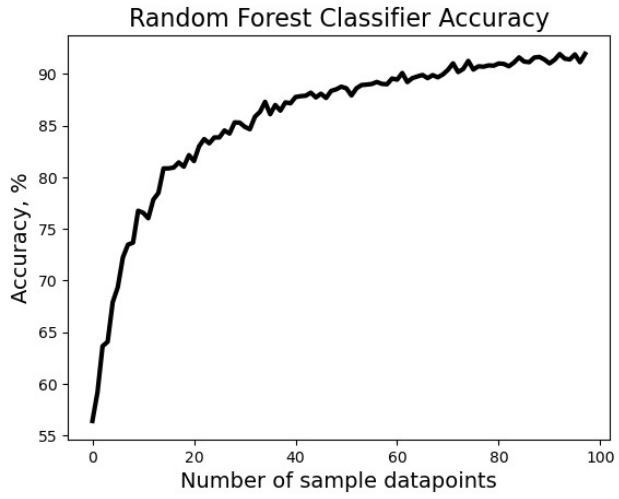


Figure 9: Random forest classifier average accuracy as a function of initial training sample size over 50 trials.

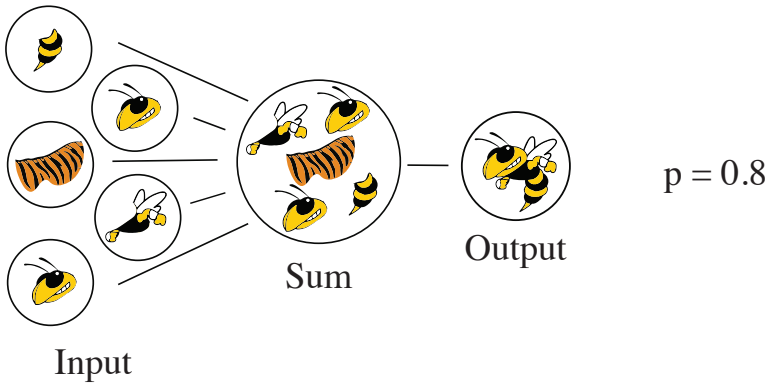


Figure 10: Simplified schematic to describe a logistic regression classifier.

2.4 Logistic Regression

We will now look into another common machine learning model. A logistic regression functions by summing input features with a bias term and returning a value between 0 and 1.

In other words, a logistic regression model is a binary classifier, which is different from our previous RF classifier, which could classify among multiple groups. Logistic regression is a statistical model and hence, uses a probabilistic approach. It models the log-odds (natural log of the ratio of probabilities) as a linear sum (*z*) of the various features (X_i) of the sample:

$$\ln\left(\frac{p(X)}{1-p(X)}\right) = z = \beta_0 + \beta_1 X_1 + ... + \beta_n X_n$$

Here, $p(x)$ is the probability of X belonging to one of the groups (say, group A), X_i is the value of the i^{th} feature of the sample, and n is the number of features.

A Logistic Regression classifier uses the given training data to estimate each of the coefficients involved in calculating the score z using MLE (Maximum Likelihood estimates). In simple terms, it finds values such that the likelihood of the given data following the above equation is maximized. Once it is trained, it can use the coefficients to calculate a score for any input. Then it can calculate the probability using the logistic function:

$$p(X) = \frac{1}{1+e^{-z}}$$

If $p(X) \geq 0.5$ then it classifies the input in group A, otherwise it is classified into group B (Figure 10).

We will explore this using an example and also learn how to utilize python for Logistic Regression.

2.4.1 Example: Will it rain tomorrow?

Let's assume we have a dataset where each sample represents a day using two features: humidity and temperature. We want to use such data to be able to predict if it will rain tomorrow. As you can see, this is a binary classification problem where the classes are "it will rain tomorrow" or "it will not rain tomorrow". In the following code, we will create a mock dataset that will represent our training data:

```
    import pandas as pd

# Imagine you have a DataFrame `df` with 'Temperature', 'Humidity', and 'WillRainTomorrow
                                                    ' columns
```

```
# 'WillRainTomorrow' is 1 if it will rain tomorrow, and 0 otherwise.

df = pd.DataFrame({
    'Temperature': [23, 30, 25, 28, 27, 32, 24, 22, 33, 29],
    'Humidity': [70, 55, 75, 60, 80, 50, 68, 65, 45, 58],
    'WillRainTomorrow': [1, 0, 1, 0, 1, 0, 1, 0, 0, 1]
})

# Split the data into features (X) and target (y)
X = df[['Temperature', 'Humidity']]
y = df['WillRainTomorrow']

print(df)
```

Now, we will use this data to train a Logistic regression model available in python as part of the sklearn.linear_model package (more info) Once, we train the model, we can check what coefficients it calculates using the provided data.

```
    from sklearn.linear_model import LogisticRegression

    #Define a Logistic Regression model
    model=LogisticRegression()

    #Train the model using X and y
    model.fit(X,y)

    print("Intercept: ", model.intercept_)
    print("Coefficients: ", model.coef_)
    ----
out: Intercept:  [-53.32358036]
    Coefficients:  [[0.65449537 0.58065493]]
```

Thus, using our provided data, the model has learned to model the probability of raining as logistic function of z which is the linear weighted sum of the features: Temperature and humidity.

$$z = -53.32 + 0.65 \cdot Temperature + 0.58 \cdot Humidity$$

Now, we can use our model to predict if it will rain tomorrow given temperature and humidity of today. Try it with today's temperature and humidity below.

```
    todays_Temp=24
    todays_Humidity=80
    X=[[todays_Temp, todays_Humidity]]

    R=model.predict(X)
    if R[0]==0:
      print("It will not rain tomorrow")
    else:
      print("It will rain tomorrow")
```

Logistic Regression can be used to make a non-binary classification using a method called One vs All (OvA) where a binary model is created for each group and then the classifier that gives the highest probabibility dictates which group the sample belongs in.

Let's say we have three groups 'A', 'B' and 'C'. The classifier will create 3 models -

- Model that classifies 'A' and not 'A' by calculating $P_A(X)$
- Model that classifies 'B' and not 'B' by calculating $P_B(X)$
- Model that classifies 'C' and not 'C' by calculating $P_C(X)$

The classifier will use these three models to calculate the probability for any given input and assign the class based on the highest probability, i.e, if $P_C(X)$ is the highest, then the input will be categorized into group C. The *LogisticRegression()* function by default uses this algorithm when provided training data containing more than 2 classes.

2.4.2 Challenge Problem 3: Logs versus forests

> Using the code you developed in the previous challenge problem, determine the average accuracy of the random forest classifier and logistic regression model over 10 trials for a sample data size ranging from 10 to 100 data points. Graph the results of this analysis and compare the results from both methods.

Challenge Problem Solution

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

def RandomForest(data):
    y = data[:,0]>data[:,1]

    clf = RandomForestClassifier(random_state=0)
    clf.fit(data,y)

    dataTest = np.random.randint(0,100,size=(1000,3))
    prediction = clf.predict(dataTest)

    trueResult = dataTest[:,0]>dataTest[:,1]
    difference = prediction[:] == trueResult[:]

    Accuracy = np.count_nonzero(difference)/np.size(difference)
    return(Accuracy)

def LogRegress(data):
    y = data[:,0]>data[:,1]
    if all(y==True) or all(y==False):
        return(float("nan"))
    clf = LogisticRegression(random_state=0)
    clf.fit(data,y)

    dataTest = np.random.randint(0,100,size=(1000,3))
    prediction = clf.predict(dataTest)

    trueResult = dataTest[:,0]>dataTest[:,1]
    difference = prediction[:] == trueResult[:]

    Accuracy = np.count_nonzero(difference)/np.size(difference)
    return(Accuracy)

numtrials = 10
sizerange = np.arange(10,101,2)
accuracymat = np.zeros((len(sizerange),numtrials))

for i_ind, i in enumerate(sizerange):
  for j in range(0,numtrials):
    data = np.random.randint(0,100,size=(i,3))
    accuracymat[i_ind,j] = RandomForest(data)*100
```

```
    plt.plot(sizerange,accuracymat.mean(axis=1),linewidth=3,color='k',label='Random
                                                    Forest')
    plt.xlabel('Number of sample datapoints',fontsize=14)
    plt.title('Classifier Accuracy',fontsize=16)
    plt.ylabel('Accuracy, %',fontsize=14)
    plt.ylim((50,100))

    for i_ind, i in enumerate(sizerange):
        for j in range(0,numtrials):
            data = np.random.randint(0,100,size=(i,3))
            accuracymat[i_ind,j] = LogRegress(data)*100


    plt.plot(sizerange,accuracymat.mean(axis=1),linewidth=3,color='r',label='Logistic
                                                    regression')
    plt.legend()
    plt.show()
```

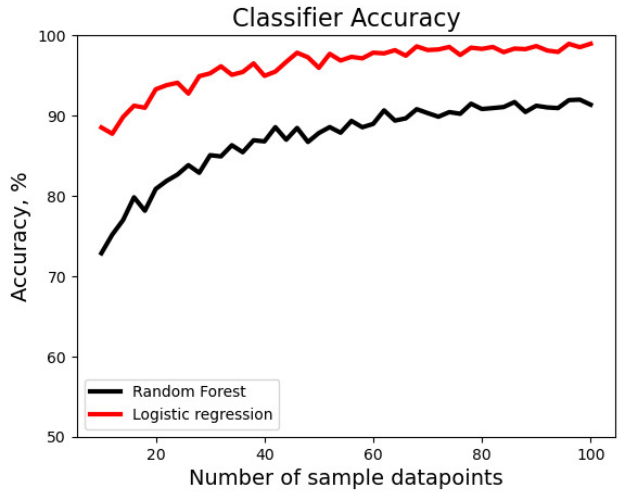


Figure 11: Comparison of accuracy comparison between logistic regression and random forest machine learning algorithms.

2.4.3 Challenge Problem 4: Too many imposters

> Repeat the previous challenge problem with a dataset of size 100 but increase the number of features from 3 to 103 (intervals of 10). Define the classification as before (determined by the first two features). Essentially, this will provide an increasing number of uninformative features (noise) in the dataset.

Challenge Problem Solution

```
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.linear_model import LogisticRegression

    def RandomForest(data,i):
        y = data[:,0]>data[:,1]
```

```
        clf = RandomForestClassifier(random_state=0)
        clf.fit(data,y)

        dataTest = np.random.randint(0,100,size=(1000,2+i))
        prediction = clf.predict(dataTest)

        trueResult = dataTest[:,0]>dataTest[:,1]
        difference = prediction[:] == trueResult[:]

        Accuracy = np.count_nonzero(difference)/np.size(difference)
        return(Accuracy)

    def LogRegress(data,i):
        y = data[:,0]>data[:,1]
        if all(y==True) or all(y==False):
            return(float("nan"))
        clf = LogisticRegression(random_state=0)
        clf.fit(data,y)

        dataTest = np.random.randint(0,100,size=(1000,2+i))
        prediction = clf.predict(dataTest)

        trueResult = dataTest[:,0]>dataTest[:,1]
        difference = prediction[:] == trueResult[:]

        Accuracy = np.count_nonzero(difference)/np.size(difference)
        return(Accuracy)

    numtrials = 10
    sizerange = np.arange(1,102,10)
    accuracymat = np.zeros((len(sizerange),numtrials))

    for i_ind, i in enumerate(sizerange):
      for j in range(0,numtrials):
        data = np.random.randint(0,100,size=(100,2+i))
        accuracymat[i_ind,j] = RandomForest(data,i)*100


    plt.plot(sizerange,accuracymat.mean(axis=1),linewidth=3,color='k',label='Random
                                                    Forest')
    plt.xlabel('Number of uninformative features',fontsize=14)
    plt.title('Classifier Accuracy',fontsize=16)
    plt.ylabel('Accuracy, %',fontsize=14)
    plt.ylim((50,100))

    for i_ind, i in enumerate(sizerange):
        for j in range(0,numtrials):
            data = np.random.randint(0,100,size=(100,2+i))
            accuracymat[i_ind,j] = LogRegress(data,i)*100


    plt.plot(sizerange,accuracymat.mean(axis=1),linewidth=3,color='r',label='Logistic
                                                    regression')
    plt.legend()
    plt.show()
```

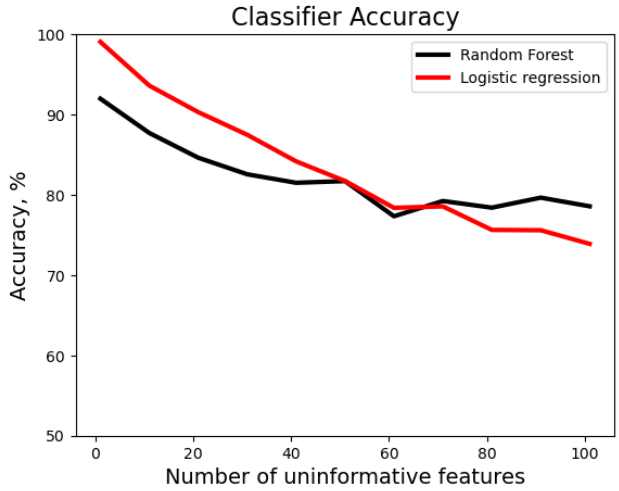


Figure 12: Comparison of LR with RF accuracy for data with a large amount of uninformative features.

2.5 Multilayer Perceptron

A Multilayer Perceptron (MLP) is a supervised neural network model that learns a transformation function to convert input arrays into output arrays based on a training dataset. This sounds similar to the logistic regression model, which transforms the data using a linear sum to predict output. This is done using multiple layers of nodes (also referred to as neurons) in a directed graph, with each layer fully connected to the next one. The nodes in the MLP are organized into an input layer, one or more hidden layers, and an output layer (Figure 13).

- **Input Layer:** This layer consists of nodes corresponding to the input features. Each node in this layer represents an individual feature.
- **Hidden Layers:** These layers are between the input and output layers. Each node in these layers uses a nonlinear function with the purpose of transforming the inputs into something that the output layer can use. (This is simlar to how in Logistic regression the input features are converted into a z-score as a linear sum and then into a probability).
- **Output Layer:** The final layer is the output layer, which is the class label for the sample.

Each node in a layer is connected to all the nodes in the next layer (shown in the below figure). These connections are weighted, and the training data is used to determine weights. There are various algorithms that MLPs can use to solve for the weights. In the example, we will use *lbfgs* because it is efficient for small samples.

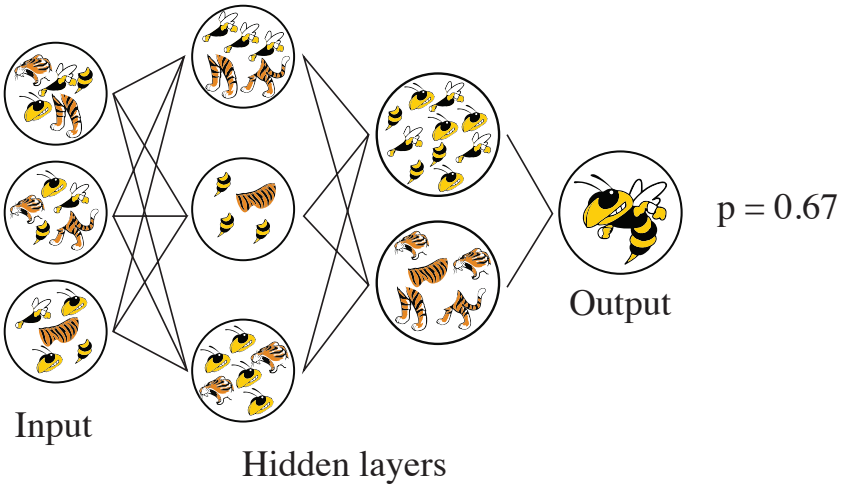


Figure 13: Two hidden layer MLP.

Look more closely at the example flowchart (Figure 13). On the lefthand side, we have the input layer where data (input as some array of size n = 3 here) is fed to the MLP. Each data point represents a different feature of the system. If you were making a script to estimate a human's weight, you might include input variables like height, age, exercise, number of cinnamon rolls consumed per hour, etc. This data is passed to the hidden layers, which apply a weighted sum of each of the previous layer's cells (also called Neurons), then transforms them with a non-linear function. This is repeated for the second hidden layer. Finally, the output later transforms the values in the second hidden layer to a usable output. In our example, it would hopefully line up with an expected weight.

MLPs have an advantage over classifiers and other regression models in that they are powerful tools for learning non-linear models and can be modified to adapt to train to data in real-time. However, MLPs can be sensitive to different random initializations as well as feature scaling. Furthermore, MLPs often require tuning parameters such as the number of hidden layers and neurons to obtain high degrees of accuracy. Give the following example a try:

```
from sklearn.neural_network import MLPClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
# the hidden layer size variable tells us that the first hidden layer
# has 3 neurons, the second has 2. (like in the diagram)
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(3, 2), random_state=1)

clf.fit(X, y)

# From here, we can make predictions just like using the other models
clf.predict([[2., 2.], [-1., -2.]])
```

We now offer a slightly more complex example that you can try. Notice the new functions imported here:

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a random n-class classification problem
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=
                                                0, random_state=1)
```

```
# Split the dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=1)

# Create a MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(5,2,), random_state=0)

# Train the model
mlp.fit(X_train, y_train)

# Predict the response for test dataset
y_pred = mlp.predict(X_test)

# Model Accuracy
print("Accuracy:",accuracy_score(y_test, y_pred))
```

Before we analyze the MLP results, let's look at new functions that can make life easier when testing various ML classifiers:

- **Generating random datasets:** An easy way of creating random datasets to test classifiers using the *make_classification* function of *sklearn.datasets* module. Here we create a dataset of 100 samples with 2 features, where both are informative and have no redundancy amongst them.
- **Creating training and testing datasets:** The *train_test_split* from the *sklearn.model_selection* module is a function that helps split our data into a set for training the classifier and a set for testing the classifier. The stratify=y parameter makes sure that the proportion of classes in each set is the same as the proportion of classes in the original data provided.
- **Measuring accuracy:** Previously, we have manually calculated the accuracy of models. A function in python that does the same is *accuracy_score* from the *sklearn.metrics* module.

  For using MLP, we use the *MLPclassifier* function of *sklearn.neural_network* module. We define the hidden layer sizes to be used as (5,2) which means that it will have 2 hidden layers; the first layer will have 5 nodes and the second layer will have 2 nodes. There is no set rule when it comes to deciding the size of the hidden layers. For many problems, a single large hidden layer is sufficient.

2.6 Transformers and pre-processors

Machine learning algorithms often use multiple steps instead of singular estimators to predict results. Usually, there are many pre-processing steps that transform the data in some way which makes it more accessible to the estimator, leading to more accurate results.

Let's cover the StandardScaler which centers data points and transforms them based on their mean and standard deviation. Let us consider a dataset where one feature is "Years of experience" and another feature is "Salary" in a job. You can imagine that experience will range from 1 to 40 years and salary can range from 10,000 to 1,000,000. Thus, salary will have a greater variance. In this scenario, we know that these differences in variance are due to the different scales at which the features are measured. Hence, to adjust for this we can rescale our data. We rescale the data such that the mean of the feature is 0 and the variance is 1 as follows:

$$Z_i = \frac{X_i - \bar{X}}{S_X} \tag{1}$$

where X_i is the i^{th} value of the feature X. By subtracting each value with the average value of the feature ($\bar{X}$), we rescale it such that the new mean of the feature is 0. Now we can divide it by the

standard deviation, S_X to reduce the variance to 1. Hence, we obtain the standardized value for that element, which is Z_i. By doing these for all our features, we can remove any potentially confounding effects of scaling.

```
from sklearn.preprocessing import StandardScaler

data = [[0,0],[1,1],[1,0],[2,1]]
scaler = StandardScaler()
scaler.fit(data)
print(scaler.transform(data))
----
[[-1.41421356 -1.        ]
 [ 0.          1.        ]
 [ 0.         -1.        ]
 [ 1.41421356  1.        ]]
```

Now we're going learn how to use a pipeline to apply this transformation (and other transformations as needed) to an estimator.

2.7 Pipelines

Pipelines are end-to-end constructs of various sequential steps that can lead to a desired result. For example, the pipeline to bring you bottled water goes from a natural source to a filtering facility to a packaging facility to a distributor and then finally to you. In machine learning, you can think of the natural source as your data and your result is the bottle of water you receive. The facilities in between represent the various pre-processors, transformers, and classifiers that you use. This essentially allows us to automate a set of steps such that we can enter the data into the pipeline and then obtain our result without having to manually complete each step. The next example will help explain this further.

2.7.1 Example: Using a pipeline

We will try to define and utilize a pipeline, *pipe*, which takes the data and does the following:

1. Standardizes the data (using *StandardScaler*)
2. Uses a logistic regression classifier

This will be done using the *sklearn.pipeline* module.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# load the iris dataset and split it into train and test sets
X , y = load_iris(return_X_y=True)
X_train , X_test , y_train , y_test = train_test_split (X , y , random_state=0)
```

For this example, we will use a botanical dataset available in the sklearn package. We load and split the dataset into a training set that can be used to develop and train our ML classifier and a test set which can be used to check the accuracy of the trained model.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

# create a pipeline object
pipe = make_pipeline(StandardScaler() , LogisticRegression())
pipe.fit(X_train, y_train)
```

Now, we define a pipeline as we had described above using the make_pipeline function. This module uses the same API as the classifiers we used previously. Hence we use the pipe.fit(X_train,y_train) using the previously created training dataset to train the model. When we enter our data into our pipeline, the program first standardizes the data using *StandardScaler()* and then uses the transformed data to train a *LogisticRegression()* model.

```
    from sklearn.metrics import accuracy_score

    # we can now use it like any other estimator
    accuracy_score(pipe.predict(X_test) , y_test)
```

Now, we can use a function from the sklearn.metrics module called accuracy_score to test the trained model using the previously created training datasets.

2.7.2 Challenge Problem 5: Down the pipeline

Define two pipelines:

- Pipeline 1: Standardization followed by a RF classifier
- Pipeline 2: Standardization followed by a LR model

Now use two types of random data from a(n):

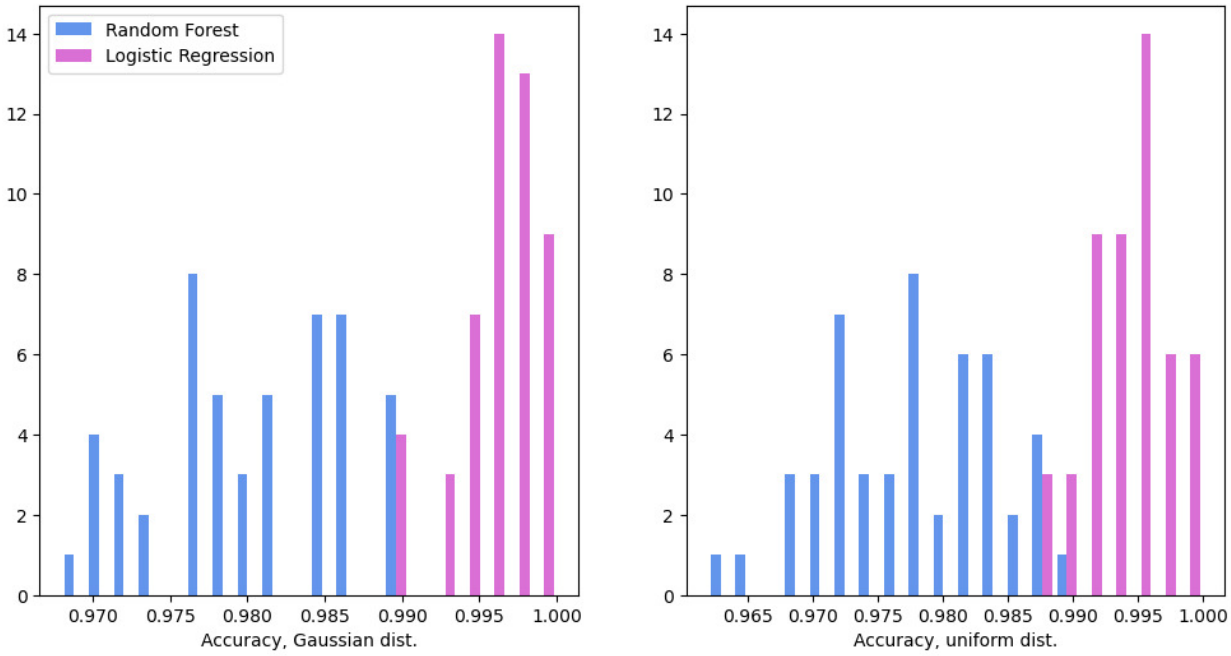
- Gaussian distribution
- Uniform distribution

to train the models. Compare your model accuracy for the trials. Repeat the measurement several times and plot the resulting accuracies.

Challenge Problem Solution

```
    from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt

numtrials=50

#Test using Gaussian data
test_acc_GauRFC = np.zeros(numtrials)
test_acc_GauLog = np.zeros(numtrials)
for i in range(0, numtrials):
    pipe_RF = make_pipeline(StandardScaler() , RandomForestClassifier(random_state=0) )
    pipe_Log = make_pipeline(StandardScaler() , LogisticRegression() )
    X = np.random.normal(50,10,size=(2000,3))
    y = X[:,0] > X[:,1]
    X_train , X_test , y_train , y_test = train_test_split (X, y, random_state=0 )
    pipe_RF.fit(X_train , y_train)
    pipe_Log.fit(X_train , y_train)
    test_acc_GauRFC[i] = accuracy_score(pipe_RF.predict(X_test) , y_test)
    test_acc_GauLog[i] = accuracy_score(pipe_Log.predict(X_test) , y_test)
```

```
fig, axs = plt.subplots(1,2, figsize=(12,6))
axs[0].hist([ test_acc_GauRFC , test_acc_GauLog ], label = ['Random Forest','Logistic
                                                Regression'], color = ['cornflowerblue',
                                                'orchid'], bins=20)
axs[0].set_xlabel('Accuracy, Gaussian dist.')
axs[0].legend()

#Test using uniform data
test_acc_GauRFC = np.zeros(numtrials)
test_acc_GauLog = np.zeros(numtrials)
for i in range(0, numtrials):
    pipe_RF = make_pipeline(StandardScaler() , RandomForestClassifier(random_state=0) )
    pipe_Log = make_pipeline(StandardScaler() , LogisticRegression() )
    X = np.random.uniform(0,100,size=(2000,3))
    y = X[:,0] > X[:,1]
    X_train , X_test , y_train , y_test = train_test_split (X, y, random_state=0 )
    pipe_RF.fit(X_train , y_train)
    pipe_Log.fit(X_train , y_train)
    test_acc_GauRFC[i] = accuracy_score(pipe_RF.predict(X_test) , y_test)
    test_acc_GauLog[i] = accuracy_score(pipe_Log.predict(X_test) , y_test)

axs[1].hist([ test_acc_GauRFC , test_acc_GauLog ], label = ['Random Forest','Logistic
                                                Regression'], color = ['cornflowerblue',
                                                'orchid'], bins=20)
axs[1].set_xlabel('Accuracy, uniform dist.')
plt.show()
```



2.7.3 Challenge Problem 6: Again with MLP

Just as you can use the LogisticRegression and RandomForestClassifies tools in a pipeline, you can also use the MLPClassifier in the same scenarios. Starting from the previous challenge problem, augment your code to compare the accuracy between a RandomForestClassifier and the MLPClassifier. Try to make adjustments to increase the accuracy. *Hint: Consider the size and number of hidden layers, the accuracy parameter alpha, and the random state.*

Challenge Problem Solution

```
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
## Begin Test for Gaussian RFC
test_acc_GauRFC = np.zeros(50)
for i in range(len(test_acc_GauRFC)):
    pipe = make_pipeline(
        StandardScaler(),
        RandomForestClassifier(random_state=0)
    )
    X = np.random.normal(50, 10, size=(2000, 3))
    #X = np.random.randint(0, 100, size=(2000, 3))
    y = X[:,0]>X[:,1]
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

    pipe.fit(X_train, y_train)
    test_acc_GauRFC[i] = accuracy_score(pipe.predict(X_test), y_test)

## Begin Test for Gaussian MLPClassifier
test_acc_GauMLP = np.zeros(50)
for i in range(len(test_acc_GauMLP)):
    pipe = make_pipeline(
        StandardScaler(),
        MLPClassifier(
                solver="lbfgs",
                alpha=1e-5,
                random_state=1,
                max_iter=2000,
                early_stopping=True,
                hidden_layer_sizes=(3,2)
            )
    )
    X = np.random.normal(50, 10, size=(2000, 3))
    #X = np.random.randint(0, 100, size=(2000, 3))
    y = X[:,0]>X[:,1]
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

    pipe.fit(X_train, y_train)

    test_acc_GauMLP[i] = accuracy_score(pipe.predict(X_test), y_test)
plt.hist([test_acc_GauRFC,test_acc_GauMLP],label=['Random Forest','MLP'],bins=20)
plt.xlabel('Accuracy')
plt.legend()
```

2.7.4 Challenge Problem 7: The only alcohol you will get served

Create three pipelines that standardize data and classify them using each of the three classifiers taught today.
Use the dataset found here to train your model. It has 11 variables reporting various features of red wine samples. We will treat quality as the dependent variable.
Now use the dataset found here which reports the same variables for white wine sample. Predict the quality of these samples and crosscheck with the reported quality. How do the various models compare to each other? Comment on the accuracies.
Hint: Load the data using pandas with the code: pd.read_csv(url,delimiter=';')

Challenge Problem Solution

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.pipeline import make_pipeline
import pandas as pd

url='https://raw.githubusercontent.com/Sdatta4556/Workshop2023_ML1/main/winequality-red
                                                    .csv'
data = pd.read_csv(url, delimiter=';')
print(data)

col=data.columns
X_train=data[col[:-1]]
y_train=data['quality']

pipe_RF = make_pipeline(StandardScaler() , RandomForestClassifier())
pipe_Log = make_pipeline(StandardScaler() , LogisticRegression(max_iter=1000) )
pipe_MLP = make_pipeline(StandardScaler(), MLPClassifier(hidden_layer_sizes=(20, 10,),
                                                    max_iter=10000))

pipe_RF.fit(X_train,y_train)
pipe_Log.fit(X_train,y_train)
pipe_MLP.fit(X_train,y_train)

url='https://raw.githubusercontent.com/Sdatta4556/Workshop2023_ML1/main/winequality-
                                                    white.csv'
data = pd.read_csv(url, delimiter=';')

col=data.columns
X_test=data[col[:-1]]
y_test=data['quality']

acc_RF=accuracy_score(pipe_RF.predict(X_test),y_test)
acc_Log=accuracy_score(pipe_Log.predict(X_test),y_test)
acc_MLP=accuracy_score(pipe_MLP.predict(X_test),y_test)

print("Accuracy of trained model on red wine dataset using RF:", acc_RF)
print("Accuracy of trained model on red wine dataset using Logistic Regression:",
                                                    acc_Log)
print("Accuracy of trained model on red wine dataset using MLP:", acc_MLP)
```

You will notice that our models have low accuracy in predicting the quality of white wine even though it was trained on a large dataset (>1500 samples). This can be due to two reasons:

- Red wine is not a good predictor of white wine
- There are too many unrelated features in our dataset

There is nothing that can be done to solve the first problem as it is an intrinsic property of the system being studied. However, we can try solving the latter by choosing features that are informative and removing noise from our data. Tomorrow, we will learn how to do so using the protein data of SARS-CoV-2 and SARS-CoV.

References

(1) Iversen, L. F.; Møller, K. B.; Pedersen, A. K.; Peters, G. H.; Petersen, A. S.; Andersen, H. S.; Branner, S.; Mortensen, S. B.; Møller, N. P. H. Structure Determination of T Cell Protein-tyrosine Phosphatase*. *Journal of Biological Chemistry* **2002**, *277*, 19982–19990, DOI: https://doi.org/10.1074/jbc.M200567200.

(2) Ullrich, S.; Ekanayake, K. B.; Otting, G.; Nitsche, C. Main protease mutants of SARS-CoV-2 variants remain susceptible to nirmatrelvir. *Bioorganic & Medicinal Chemistry Letters* **2022**, *62*, 128629, DOI: https://doi.org/10.1016/j.bmcl.2022.128629.