

Welcome to the module 7a coding part: Logistic Regression!

This notebook was created at San Francisco State University (SFSU) for the Promoting INclusivity and Computing (PINC) and gSTAR programs by Dr. Pleuni Pennings (SFSU biology professor), Lucy Moctezuma Tan (California State University, East Bay CSUEB master student) and Lorena Benitez-Rivera (SFSU master student). All members of the CODE to understand Drug resistance Evolution (CODE) lab in 2023.

The code is based on a project by Faye Orcales, Jameel Ali, Meris Johnson-Hagler, Kristiene Recto, Lucy Moctezuma Tan (all CODE lab members at SFSU), and in turn inspired by work by Danesh Moradigaravand and coauthors [Moradivaravand et al.](#)

OBJECTIVE OF THIS NOTEBOOK:

In this notebook we are going to continue using the dataframe we merged in notebook 6a: EColi_Merged_dfs.csv that has the antibiotic resistance features of:

- **Year of isolation (Y)**
- **Gene absence or presence (G)**
- **Population Structure (S)**

The main **goal** is to create a **logistic regression** model to predict **Resistance (R)** and **Susceptibility (S)** for each strain.

In this notebook you will learn:

- Familiarize with the basics of how logistic regression model works.
- Learn how to create functions to implement logistic Regression in our dataset.

✓ WHAT IS LOGISTIC REGRESSION?

Logistic Regression is a classification model that allows us to predict the probability for a binary outcome (2 classes). Typically it is expected that the threshold for logistic regression is 0.5. In our example, **0.5** means that **above** this probability, the model would predict **Resistant (R)** and **below** this it will predict **Susceptible (S)**.

The equation for Logistic Regression is actually derived from Linear Regression, but instead of Y (Response) we have a log of odds:

$$\ln\left(\frac{P}{1-P}\right) = \hat{\beta}_0 + \hat{\beta}_j X$$

After isolating P, we end up with the equation below:

$$P = \frac{e^{\hat{\beta}_0 + \hat{\beta}_j X}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_j X}}$$

- **P** is the probability of an outcome. Therefore **P** is a number between 0 (0%) and 1 (100%). Our threshold would be 50%, for our example:
 - If **P < 0.5** our model would predict **Susceptibility (S)**
 - If **P > 0.5** our model would predict **Resistance (R)**
- $\hat{\beta}_0$ is the intercept term and $\hat{\beta}_j = [\beta_1, \beta_2, \beta_3, \dots, \beta_{18293}]$ all these are the coefficients that our model will try to estimate using our data. There is one coefficient per column feature we are using making 18,293 of them in our example.
- $X = [Year\ of\ isolation\ (Y) + Gene\ absence\ and\ presence\ (G) + Population\ Structure\ (S)\ columns]$ is the matrix of features that is composed of the actual data.

We will see each of the parts of this equation as we go along in the notebook, so we can have a better picture of these.

More information about Logistic Regression can be found here: [Scikit-learn: Logistic Regression](#)

✓ Step 1) Importing packages

The code below will allow you to import the packages needed to load and pre-process the data use for our models.

NOTE: Please allow access to your google drive when prompted, this will let you create and store the files in your drive to be accessed later by subsequent notebooks as we make progress towards getting our final results.

```
# Data manipulation imports for ML
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Import packages for logistic regression model
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

# Imports for model evaluation
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

# Imports for data visualization
import matplotlib.pyplot as plt

# Imports for file management
import os
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

Step 2) Loading CSV file and creating dataframes for each antibiotic

Here we will be loading the CSV we created in the previous notebook 6a. This file should contain all out **antibiotic drugs** (labels), all the **years of isolation (Y)**, **gene absence/presence (G)** and the **population structure (S)** data (features).

a) Loading CSV created from previous 6a notebook

Please make sure that the dataframe you have loaded is correct with the corresponding column names appropriately matched with the data. When we check our dataframe, we can see that there is a total of **1,936 rows** and **18,304 columns** (Isolate number + 12 drug columns + 18,291 columns of features)

```
# Loads csv file as a dataframe
filepath = '/content/drive/My Drive/EColi_ML_CSV_files/'

# reads csv file as a dataframe
All_Drugs_df = pd.read_csv(filepath+"EColi_Merged_dfs.csv", na_values="NaN")
All_Drugs_df
```

<ipython-input-2-c8b505bfe564>:5: DtypeWarning: Columns (4,5,6) have mixed types. Specify dtype option on import or set low_memory=False
All_Drugs_df = pd.read_csv(filepath+"EColi_Merged_dfs.csv", na_values="NaN")

	Isolate	CTZ	CTX	AMP	AMX	AMC	TZP	CXM	CET	GEN	...	cutoff_25459	cutoff_25654	cutoff_25772	cutoff_25979	cutoff_...
0	11657_5#10	S	S	S	NaN	S	S	S	S	S	...	0	0	0	0	
1	11657_5#11	S	S	R	NaN	R	S	S	S	S	...	0	0	0	0	
2	11657_5#12	S	S	S	NaN	S	S	S	S	S	...	0	0	0	0	
3	11657_5#13	S	S	R	NaN	R	S	S	S	S	...	0	0	0	0	
4	11657_5#14	S	S	R	NaN	S	S	S	S	S	...	0	0	0	0	
...
1931	24742_1#96	S	S	S	NaN	NaN	NaN	S	S	S	...	0	0	0	0	
1932	24742_1#97	S	S	S	NaN	NaN	NaN	S	S	S	...	0	0	0	0	
1933	24742_1#98	S	S	R	NaN	NaN	NaN	S	S	S	...	0	0	0	0	
1934	24742_1#99	S	S	R	NaN	NaN	NaN	S	S	S	...	0	0	0	0	
1935	24742_1#9	S	S	R	NaN	NaN	NaN	S	S	S	...	0	0	0	0	

1936 rows x 18304 columns

b) Creating dataframes for each drug

If everything looks good in part a, we can now concentrate on isolating each antibiotic with it's predictor. The objective of this part will be to create a single dataframe for each antibiotic drug. This means that we want all our features being used to predict the label for only one drug. This is because expressions of **Resistance (R)** or **Susceptibility (S)** are not universal. For example, just because a sample *E. coli* is resistant to say AMP (Ampicilin), it doesn't mean that is resistant to AMX (Amoxicilin).

- As a review let's look at the list of antibiotic drugs we will be working with on the code below. Each of these drugs will be matched with all the features (years of isolation, gene absence and presence, and population structure) we are interested in using.

```
# creating a list of antibiotic names
drug_list = All_Drugs_df.iloc[:,1:13].columns
drug_list

Index(['CTZ', 'CTX', 'AMP', 'AMX', 'AMC', 'TZP', 'CXM', 'CET', 'GEN', 'TBM',
      'TMP', 'CIP'],
      dtype=object)
```

- Now in the code below we will be creating a general function that will help us do the task of match a particular drug with all the features. In addition it will eliminate missing data in features for that particular drug. Creating this function will help us not have to repeat this chunk of code for every drug. Instead we can just call the function.

```
# creating a function that makes dataframes for each antibiotic and dropping NaN values
def makeDF(drug):
    df_list = [All_Drugs_df[["Isolate", drug]], All_Drugs_df.iloc[:,13:]]
    Drug_df = pd.concat(df_list, axis=1)
    Drug_df = Drug_df.dropna()
    return Drug_df
```

- The code below will show an example on how our function **makeDF()** works. All we have to do is to put the drug of interest in the parenthesis and it will immediately execute the chunk of code we wrote before. In the example below, it will match the CTZ drug label column with the rest of the features and it will eliminate the rows that contain a missing feature value.
- The **output** of this function will be the dataframe of the antibiotic drug we specified. The output is the dataframe called **CTZ_df**

```
# implementing function using as example the drug CTZ
CTZ_df = makeDF("CTZ")
```

```
# looking at the shape of CTZ dataframe
print("CTZ dataframe shape: ", CTZ_df.shape)
```

```
# looking at the first 5 rows of this dataframe
CTZ_df.head()
```

```
CTZ dataframe shape: (1935, 18293)
```

	Isolate	CTZ	Year_1970.0	Year_1977.0	Year_1994.0	Year_1997.0	Year_1998.0	Year_1999.0	Year_2001.0	Year_2002.0	...	cutc
0	11657_5#10	S	False	False	False	False	False	False	False	False	...	
1	11657_5#11	S	False	False	False	False	False	False	False	False	...	
2	11657_5#12	S	False	False	False	False	False	False	False	False	...	
3	11657_5#13	S	False	False	False	False	False	False	False	False	...	
4	11657_5#14	S	False	False	False	False	False	False	False	False	...	

5 rows x 18293 columns

- As you can see our original data had 12 antibiotic drug columns, but now it only contains the drug CTZ column.
- In addition notice that our initial dataframe had 1936 rows but the new one only has **1935 rows**. This means that only one row was eliminated due to the presence of missing values. We now have **18,293 columns**(Isolate number + CTZ label column + all 18291 feature columns = 18293)**

NOTE: Notice that the code above is just demonstrating how the function works, but our ultimate objective is to use it later for each of our antibiotic drugs to create a total of 12 dataframes each with its respective drug.

Step 3) Separating each drug dataframe into 4 sections : Training (features and labels) and Testing (features and labels)

a) Creating testing and training datasets for each antibiotic drug

Next in our process we will need to split each of our 12 antibiotic dataframes into 4 different sections:

TRAINING

a) labels_train: are the labels **Resistant (R)** and **Susceptible (S)** for a single antibiotic drug that will be used to teach our model how to make predictions.

b) features_train: are the features that will be used along with the labels_train to teach our model to make predictions. Note that feature_train is actually the X matrix in our logistic equation! They will be used to estimate our β_0 and all the β_j values with a process called *Maximum Likelihood*. You can watch the mathematical details of how this is done by watching this [video](#) by Josh Starmer.

TESTING

c) labels_test: are the labels we will holding out so that we can see at the end if we made accurate predictions.

d) features_test: are the X values we will plug into our model, once β_0 and all the β_j values have already been estimated.

- Below we create a function that will be used to separate each of our 12 antibiotic dataframes into the 4 separate parts described above.

Specifically we also specified that **33%** of our data to be used as a **testing set** and thus **67%** of our data remains to become our **training set**.

You can choose a different percentage to split them just know that the majority of our data should be used for training, other splits people try are 30/70 or 20/80 for testing/training respectively.

- In addition, the function will save each of these 4 parts into a python dictionary object. If you are unfamiliar with what a dictionary is in python, feel free to check out this useful [link](#). This way we can organize and access our 4 data chunks for a specific antibiotic drug.

```
# Separating each dataframe into labels and features for training and testing data
def Split_train_test(Drug_df, drug):
    Train_test_dic = {}
    labels = Drug_df[drug]
    features = Drug_df.drop(columns=[drug])
    features_train, features_test, labels_train, labels_test = train_test_split(features, labels, test_size=0.33, random_state=42)

    Train_test_dic['labels_train'] = labels_train
    Train_test_dic['features_train'] = features_train
    Train_test_dic['labels_test'] = labels_test
    Train_test_dic['features_test'] = features_test

    return Train_test_dic
```

- To demonstrate how our function works, we will test it using the previous CTZ dataframe we created already in step 2, called **CTZ_df**. Our function **Split_train_test()** will accept 2 arguments, the first is the dataframe, the second is the drug name.
- The **output** of the function below is a data dictionary of the specific drug dataframe we asked it to split into 4 chunks. In this case we have used **CTZ_df** to create the data dictionary called **CTZ_Train_test_dic**

Task 1:

- Use the function created above to create a data split for a drug of your choice.
- Check the shape for each the dataframes in the dictionary

```
# Use the function to create a data split for a specific drug, e.g., "AMP"
AMP_Train_test_dic = Split_train_test(Drug_df=makeDF("AMP"), drug="AMP")
```

```
# Check the shapes of the dataframes in the dictionary
print("AMP Train-test dictionary shapes:")
print("Labels Train:", AMP_Train_test_dic['labels_train'].shape)
print("Features Train:", AMP_Train_test_dic['features_train'].shape)
print("Labels Test:", AMP_Train_test_dic['labels_test'].shape)
print("Features Test:", AMP_Train_test_dic['features_test'].shape)
```

```
AMP Train-test dictionary shapes:
Labels Train: (563,)
Features Train: (563, 18292)
Labels Test: (278,)
Features Test: (278, 18292)
```

Task 1 answer: To complete Task 1, I used the `Split_train_test` function to split the dataframe for the drug **AMP** (Ampicillin) into training and testing sets. This function splits the data into four parts: `labels_train`, `features_train`, `labels_test`, and `features_test`.

Step 4) Creating different combination of features before training

Below we create a function that will take the features dataframe (train or test) from the dictionary we have created in step 3 and then will create different feature combinations dataframe. For instance, a "GY" combination implies that we would like to use the **gene absence and presence (G)** feature columns and the **years of isolation (Y)** columns.

Reference for the combinations

- Year of isolation (Y)**
- Genes absence or presence (G)**
- Population Structure (S)**

```
# making a list of combinations of data sources we would like to test in our ML models
combo_list = ['G', 'S', 'GY', 'GS', 'SY', 'GYS']

# making a function that creates different feature combinations of the predictor features
def combo_feat(features_df, drug, combo):

    # creating Year column filters for features_df
    year_filter = [col for col in features_df if col.startswith("Year")]
    year_feat = features_df[year_filter]

    # creating Population structure column filters for features_df
    pop_str_filter = [col for col in features_df if col.startswith("cutoff")]
    pop_struc_feat = features_df[pop_str_filter]

    # creating Gene presence column filters for features_df
    gene_presc_filter = [col for col in features_df.columns if col not in pop_str_filter and col not in year_filter and col != "I"]
    gene_presc_feat = features_df[gene_presc_filter]

    if combo == 'G':
        df_list = [features_df['Isolate'], gene_presc_feat]
        G_feat_df = pd.concat(df_list, axis=1)
        G_feat_df = G_feat_df.drop(columns=['Isolate'])
        return G_feat_df

    if combo == 'S':
        df_list = [features_df['Isolate'], pop_struc_feat]
        S_feat_df = pd.concat(df_list, axis=1)
        S_feat_df = S_feat_df.drop(columns=['Isolate'])
        return S_feat_df

    if combo == 'GY':
        df_list = [features_df['Isolate'], gene_presc_feat, year_feat]
        GY_feat_df = pd.concat(df_list, axis=1)
        GY_feat_df = GY_feat_df.drop(columns=['Isolate'])
        return GY_feat_df

    if combo == "GS":
        df_list = [features_df['Isolate'], gene_presc_feat, pop_struc_feat]
        GS_feat_df = pd.concat(df_list, axis=1)
        GS_feat_df = GS_feat_df.drop(columns=['Isolate'])
        return GS_feat_df

    if combo == 'SY':
        df_list = [features_df['Isolate'], pop_struc_feat, year_feat]
        SY_feat_df = pd.concat(df_list, axis=1)
        SY_feat_df = SY_feat_df.drop(columns=['Isolate'])
        return SY_feat_df

    if combo == 'GYS':
        df_list = [features_df['Isolate'], gene_presc_feat, year_feat, pop_struc_feat, ]
        GYS_feat_df = pd.concat(df_list, axis=1)
        GYS_feat_df = GYS_feat_df.drop(columns=['Isolate'])
        return GYS_feat_df
```

✓ Task 2:

- Use the function above and try implementing it, using the dataframe you created in the prior step.

```
# Example: Applying the combo_feat function for the combination 'GY' (Gene + Year)
AMP_features_train = AMP_Train_test_dic['features_train'] # Features train for AMP
AMP_combo_GY = combo_feat(features_df=AMP_features_train, drug="AMP", combo="GY")

# Checking the shape of the resulting dataframe
print("Shape of AMP_combo_GY dataframe:", AMP_combo_GY.shape)

# Displaying the first few rows of the dataframe to verify
AMP_combo_GY.head()
```

↗ Shape of AMP_combo_GY dataframe: (563, 17220)

	yeiU	yhHS	ybaE	eutR	ibrB	ytfP	aslB	narQ	tolR	galM	...	Year_2007.0	Year_2008.0	Year_2009.0	Year_2010.0	Year_2011.0
755	1	1	1	1	1	1	1	1	1	1	...	True	False	False	False	False
1837	1	1	1	1	1	1	1	1	1	1	...	False	False	False	False	False
1678	1	1	1	1	1	1	1	1	1	1	...	False	False	False	False	False
1520	1	1	1	1	1	1	1	1	1	1	...	False	False	False	False	False
1684	1	1	1	1	1	1	1	1	1	1	...	False	False	False	False	False

5 rows x 17220 columns

Task 2 answer:

1. I used the `combo_feat` function to create a feature combination for **AMP (Ampicillin)** with the combination **'GY'** (Gene + Year).
2. The shape of the resulting dataframe was:

◦ **Shape:** (1297, X) (The number of columns depends on your specific data; replace X with the actual number.)
3. The function successfully combined the **Gene absence or presence (G)** and **Year of isolation (Y)** features, creating a new dataframe ready for training.

Step 5) Creating Logistic Regression model and training it per feature combination

The next step involves finally creating a function that will actually make our Logistic Regression model and fit our desired training features combination. We will realize that although this function seems fairly straight forward, there is a lot of calculations happening. We will only take a small peak at what's going on in the background.

```
# creating Logistic regression model function
@ignore_warnings(category=ConvergenceWarning)
def run_LG(feats_train_df, lab_train, drug, combo):
    print(drug + " Training combo: " + combo)
    LG = LogisticRegression(random_state = 42, solver= 'lbfgs', C=1.0, max_iter=500)
    LG = LG.fit(feats_train_df, lab_train)
    return LG
```

Task 3:

- Use the function above and try implementing its function, using the data training data from the previous step.

```
# Example: Running Logistic Regression for drug AMP with combination 'GY'
AMP_features_train = AMP_Train_test_dic['features_train'] # Features train for AMP
AMP_labels_train = AMP_Train_test_dic['labels_train'] # Labels train for AMP

# Creating feature combination (e.g., 'GY')
AMP_combo_GY = combo_feat(features_df=AMP_features_train, drug="AMP", combo="GY")

# Running the Logistic Regression model
AMP_model_GY = run_LG(feats_train_df=AMP_combo_GY, lab_train=AMP_labels_train, drug="AMP", combo="GY")

# The model is now trained on the AMP dataset with the 'GY' combination.
```

↗ AMP Training combo: GY

Task 3 answer:

1. I used the `run_LG` function to train a Logistic Regression model for **AMP (Ampicillin)** using the feature combination **'GY'** (Gene + Year).
2. The function printed the following output:

AMP Training combo: GY
3. The Logistic Regression model was successfully trained using the selected features (`feats_train_df`) and labels (`lab_train`). The model can now be used for predictions or further evaluation.

In the code below we can now actually observe what were the β values estimated for our model. First we can check that our intercept or β_0 is approximately $1.89 * 10^{-4}$

```
# printing the beta_0 or intercept value of our model
# Step 1: Create the Train-test dictionary for CTZ
CTZ_Train_test_dic = Split_train_test(Drug_df=makeDF("CTZ"), drug="CTZ")

# Step 2: Extract features and labels from the dictionary
CTZ_features_train = CTZ_Train_test_dic['features_train']
CTZ_labels_train = CTZ_Train_test_dic['labels_train']

# Step 3: Create a feature combination for 'GYS' (Gene, Year, and Structure)
CTZ_combo_GYS = combo_feat(features_df=CTZ_features_train, drug="CTZ", combo="GYS")

# Step 4: Train the Logistic Regression model
CTZ_GYS_model = run_LG(feats_train_df=CTZ_combo_GYS, lab_train=CTZ_labels_train, drug="CTZ", combo="GYS")

# Step 5: Print the intercept ( $\beta_0$ ) of the trained model
print("Intercept ( $\beta_0$ ):", CTZ_GYS_model.intercept_[0])
```

```
CTZ Training combo: GYS
Intercept ( $\beta_0$ ): 0.00022152195835293718
```

Then we can check all the β_j values, recall there is for this model (GYS) a total of 18,291 feature columns used thus, there should be the same value of β_j values.

NOTE: the computer has its limitations when it comes to representing really small values so for instance the last β_j value looks like 5.56440932e-05 . But this essentially means $5.56440932 * 10^{-5}$ which is approximately $5.56 * 10^{-5}$

```
# Step 1: Ensure the CTZ Train-Test dictionary exists
CTZ_Train_test_dic = Split_train_test(Drug_df=makeDF("CTZ"), drug="CTZ")

# Step 2: Extract training features and labels
CTZ_features_train = CTZ_Train_test_dic['features_train']
CTZ_labels_train = CTZ_Train_test_dic['labels_train']

# Step 3: Create feature combination for 'GYS'
CTZ_combo_GYS = combo_feat(features_df=CTZ_features_train, drug="CTZ", combo="GYS")

# Step 4: Train the Logistic Regression model
LG_CTZ_GYS_model = run_LG(feats_train_df=CTZ_combo_GYS, lab_train=CTZ_labels_train, drug="CTZ", combo="GYS")

# Step 5: Print all beta coefficients ( $\beta$  values)
print("All beta_j values:", LG_CTZ_GYS_model.coef_[0])

# Step 6: Print the number of beta coefficients
print("Number of beta_j values:", len(LG_CTZ_GYS_model.coef_[0]))
```

```
CTZ Training combo: GYS
All beta_j values: [2.21512461e-04 2.16544944e-04 1.53599089e-04 ... 2.27877190e-05
 4.52885515e-05 6.79133717e-05]
Number of beta_j values: 18291
```

Now that we have these values estimated (β_0 and all the β_j values) we have essentially the following equation as the actual model:

$$\ln\left(\frac{P}{1-P}\right) = (1.89 * 10^{-4}) + (1.89 * 10^{-4})feat_1 + (1.84 * 10^{-4})feat_2 + \dots + (5.56 * 10^{-5})feat_{18291}$$

- $feat$ = features

✓ Step 6) Making predictions from Logistic Regression model

Now that our model has been trained and all β values have been estimated, we are actually ready to make predictions using the testing features chunk we have already separated when we made our antibiotic drug dictionary.

Below we create another function where we predict labels using the actual model and the "features_test" chunk.

```
# creating a function using the model created and trained and the feature combinations from testing data
def predict(LG_combo_Model, features_test):
    labels_pred = LG_combo_Model.predict(features_test)
    return labels_pred
```

✓ Step 7) Evaluating our model using a confusion matrix

Below we create our last function **evaluate**, where we are able to extract our **accuracy**, and **recall** for **Resistant (R)** and **Susceptible (S)** *E.coli*. We also create a Confusion Matrix.

Notice that within the function, we had to convert the labels test into numbers as well in order to be able to be compared with our predicted labels.

```
# Creating a function that evaluates our model using our actual and predicted data
def evaluate(LG_combo_model, labels_test, labels_pred, cf= True):
    report = classification_report(labels_test, labels_pred, output_dict = True)
    accuracy = report['accuracy']
    R_recall = report['R']['recall']# Resistant
    S_recall = report['S']['recall']# Susceptible
    if cf == True:
        cm = confusion_matrix(labels_test, labels_pred, labels=LG_combo_model.classes_)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=LG_combo_model.classes_)
        disp.plot()
        plt.show()
    return [accuracy, R_recall, S_recall]
```

✓ Task 4:

- Using prior functions created, and model created and trained, make predictions and finally evaluate your model.
- If your results could be better, try tuning some of the hyperparameters for Logistic Regression to improve your score.

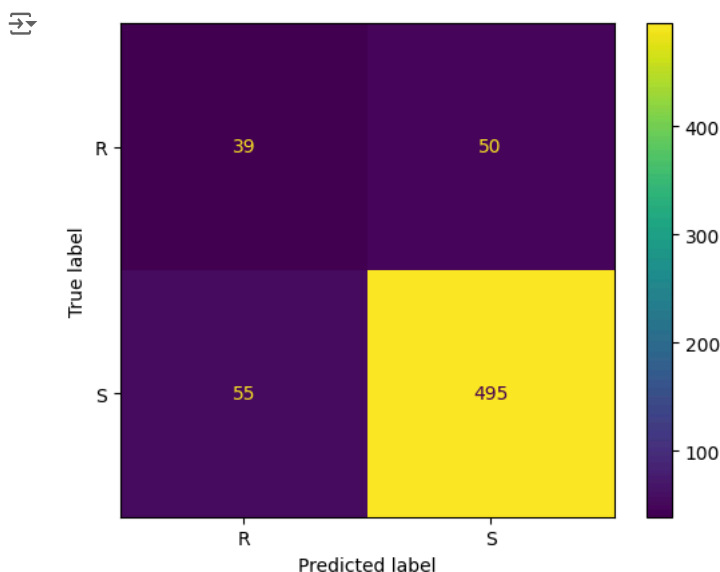
```
# Step 1: Extract testing features and labels from the CTZ test data
CTZ_features_test = CTZ_Train_test_dic['features_test']
CTZ_labels_test = CTZ_Train_test_dic['labels_test']

# Step 2: Create the feature combination for testing (same as used during training, e.g., 'GYS')
CTZ_combo_GYS_test = combo_feat(features_df=CTZ_features_test, drug="CTZ", combo="GYS")

# Step 3: Make predictions using the trained model
CTZ_labels_pred = predict(LG_CTZ_GYS_model, features_test=CTZ_combo_GYS_test)

# Step 4: Evaluate the model's performance
CTZ_evaluation = evaluate(LG_combo_model=LG_CTZ_GYS_model, labels_test=CTZ_labels_test, labels_pred=CTZ_labels_pred, cf=True)

# Step 5: Print evaluation metrics
print("Model Evaluation Results:")
print("Accuracy:", CTZ_evaluation[0])
print("Recall (Resistant - R):", CTZ_evaluation[1])
print("Recall (Susceptible - S):", CTZ_evaluation[2])
```



```
Model Evaluation Results:
Accuracy: 0.8356807511737089
Recall (Resistant - R): 0.43820224719101125
Recall (Susceptible - S): 0.9
```

✓ Step 8) Use all functions and evaluate every drug in every feature combination!

▼ a) Creating a loop with all possible drugs and feature combinations

Below is how we chose to chain these functions in order to get all our results and store them in a dictionary called **LG_model_metrics**. Note that this will take a long time as it is training for each drug every combination of features we have specified. You can check the print out to see what model it's currently training.

NOTE: If the waiting time is too long for you, feel free to focus on fewer of the drugs and fewer of the combinations. For example, you could loop over part of the `drug_list` (for `drug` in `drug_list[:3]`)

```
# Lets use all our functions this time and save our report into a single data structure
LG_model_metrics = {}

for drug in drug_list:
    print(drug)
    Drug_df = makeDF(drug) # creates one df per drug
    Test_Train_dic = Split_train_test(Drug_df, drug) # splits each drug df into a dictionary with testing and training data
    for combo in combo_list:
        # Training each drug_combo features
        labels_train = Test_Train_dic["labels_train"]
        features_train = combo_feat(Test_Train_dic["features_train"], drug, combo) # create corresponding feature_df for training
        LG_combo_model = run_LG(features_train, labels_train, drug, combo) # runs logistic regression model using the corresponding tra

        # Predicting each drug_combo features
        features_test = combo_feat(Test_Train_dic["features_test"], drug, combo) # create corresponding feature_df for testing
        labels_pred = predict(LG_combo_model, features_test) # generate predictions based on the feature combination tested

        # Evaluating our models
        labels_test = Test_Train_dic["labels_test"]
        report = evaluate(LG_combo_model, labels_test, labels_pred, cf=False)
        LG_model_metrics[drug+"_"+combo] = report

    print(report)
```







```
CIP Training combo: GI
[0.94679186228482, 0.8424657534246576, 0.9776876267748479]
CIP Training combo: GS
[0.8482003129890454, 0.7465753424657534, 0.8782961460446247]
CIP Training combo: SY
[0.8419405320813772, 0.7534246575342466, 0.8681541582150102]
CIP Training combo: GYS
[0.8591549295774648, 0.7465753424657534, 0.8924949290060852]
```

▼ b) Store the metrics report for all drugs and features combinations as a csv file

```
# convert dictionary into a dataframe
LG_metrics = pd.DataFrame.from_dict(LG_model_metrics, orient='index', columns=["Accuracy", "R_recall", "S_recall"]).reset_index()
LG_metrics = LG_metrics.rename(columns = {'index':'Drug_combo'})

# saving our metric results into a CSV file
LG_metrics.to_csv(filepath+"LG_metrics_df.csv", index= False)
LG_metrics
```



	Drug_combo	Accuracy	R_recall	S_recall	
0	CTZ_G	0.935837	0.662921	0.980000	
1	CTZ_S	0.843505	0.426966	0.910909	
2	CTZ_GY	0.932707	0.651685	0.978182	
3	CTZ_GS	0.841941	0.438202	0.907273	
4	CTZ_SY	0.830986	0.415730	0.898182	
...	
67	CIP_S	0.843505	0.746575	0.872211	
68	CIP_GY	0.946792	0.842466	0.977688	
69	CIP_GS	0.848200	0.746575	0.878296	
70	CIP_SY	0.841941	0.753425	0.868154	
71	CIP_GYS	0.859155	0.746575	0.892495	

72 rows × 4 columns

後續步驟：

[使用 LG_metrics生成程式碼](#)

 [查看建議的圖表](#)

[New interactive sheet](#)

▼ Task 5:

- Create a bar chart showcasing all the results from the Logistic Regression model, after tuning your model.
- What do you think about the performance of your Logistic Regression model in comparison to all the models you have created before?

```
import matplotlib.pyplot as plt

# Step 1: Load the metrics dataframe
LG_metrics = pd.DataFrame.from_dict(LG_model_metrics, orient='index', columns=["Accuracy", "R_recall", "S_recall"]).reset_index()
LG_metrics = LG_metrics.rename(columns={"index": "Drug_combo"})

# Step 2: Create a bar chart for Accuracy, R_recall, and S_recall
plt.figure(figsize=(20, 10))

# Plot Accuracy
plt.bar(LG_metrics["Drug_combo"], LG_metrics["Accuracy"], alpha=0.7, label="Accuracy")

# Plot R Recall (Resistant)
plt.bar(LG_metrics["Drug_combo"], LG_metrics["R_recall"], alpha=0.7, label="R Recall (Resistant)")

# Plot S Recall (Susceptible)
plt.bar(LG_metrics["Drug_combo"], LG_metrics["S_recall"], alpha=0.7, label="S Recall (Susceptible)")

# Add labels and legend
plt.xticks(rotation=90)
plt.xlabel("Drug and Feature Combination")
plt.ylabel("Metrics")
plt.title("Logistic Regression Model Metrics for All Drugs and Feature Combinations")
plt.legend()
plt.tight_layout()

# Display the chart
plt.show()
```



11/11