

REPORT NO.1

Group 23: Mini-Project 1



ECSE551

MASOUMEH SHAFIEI, MIKAEEL MAYELI, MOHAMMAD AMIN NAJAFLOU

McGill University

1 Abstract

In this project, we implemented logistic regression model and investigated its performance for binary classification on two distinct datasets - Hepatitis for disease diagnosis and Mushrooms for edibility prediction. Our goal was to optimize logistic regression through iterative feature engineering and hyperparameter tuning. The Hepatitis dataset contained blood test and medical record data, with a target variable indicating hepatitis diagnosis. For the Mushrooms dataset, the task was to predict if a mushroom sample is edible based on physical characteristics provided.

Our analysis revealed that smaller learning rates result in more stable optimization and better generalization of logistic regression. Rigorous cross-validation was employed to reduce overfitting and estimate model accuracy. For the Hepatitis data, accuracy improved from 71% to 80% after expanding the feature space using some polynomial terms and feature interactions. Further gains arose from eliminating noisy features through selection techniques. Experiments with mushrooms dataset showed that by using engineered features, such as polynomials and square roots, and incorporating feature selection, the average accuracy achieved was 70%.

In conclusion, our comprehensive analysis and optimization steps including feature engineering, selection, and hyperparameter tuning enhanced the predictive capabilities of logistic regression on these datasets. Key results were the importance of proper learning rates, value of validation, and substantial gains from an expanded and refined feature space. The work provides insights for developing accurate and interpretable healthcare and botany classification models.

2 Introduction

Machine Learning (ML) approaches has revolutionized various fields by providing powerful computational tools for identifying patterns, making predictions, and facilitating decision-making [1]. Among the many tools in machine learning, logistic regression is a key method designed mainly for binary classification. While relatively simple, logistic regression can provide classifications with high accuracy, making it a popular tool [2].

The primary aim of this project was to deepen our understanding of logistic regression by implementing this classifier from scratch, and exploring the utility and performance of logistic regression classification in different areas including medical diagnostics and botany. To achieve this, two distinct datasets, Hepatitis and Mushroom, are employed.

The initial dataset concerns Hepatitis, a type of liver disease. Diagnosing this illness through blood tests and patient information presents a significant machine learning challenge. Models capable of effectively predicting hepatitis based on test outcomes can aid physicians in early detection and intervention.

This analysis explores using logistic regression combined with data analysis and optimization techniques to diagnose hepatitis.

Through iterative feature engineering including polynomial terms, variable interactions, and eliminating uninformative features, the baseline model accuracy was improved. Additional gains are realized by tuning hyperparameters like learning rate using a validation set approach. The final optimized model leverages a narrowed engineered feature set and optimal learning rate to classify liver disease with 77% cross-validated accuracy.

This analysis highlights the importance of thorough data exploration to guide modeling, validation to avoid overfitting, and optimization techniques like feature engineering and tuning to improve model performance. The techniques demonstrated provide a blueprint for developing accurate predictive models from healthcare data.

Following on, our second attempt was on Mushroom dataset. With thousands of mushroom species, being able to accurately determine if a mushroom is poisonous or edible is extremely important. Consuming toxic mushrooms can cause severe illnesses and even death in some cases. However, identifying mushroom species based solely on visible characteristics requires extensive expertise. Automated classification through machine learning can therefore be very useful for mushroom safety.

For the Mushroom dataset, the objective was to determine whether a specific mushroom sample is edible or not, based on its key physical attributes like shape, color, bruising, odor etc. The data consists of over 8000 hypothetical mushroom samples, labeled as poisonous or edible. Each sample is described by 22 feature attributes capturing major visual properties.

We implemented logistic regression and enhanced its performance and accuracy through feature engineering, selection and hyperparameter tuning. Polynomial (square roots) and interaction features were derived to expand the feature space. Careful subset selection helped reduce overfitting. Moreover, a properly set learning rate and other hyperparameters ensured that the model converges efficiently to a solution that minimizes the cost function, leading to better generalization on unseen data.

Comprehensive experiments reveal how our methods boost logistic regression performance on this dataset. We achieve over 75% accuracy in distinguishing poisonous from edible varieties. The work provides insights into data-driven modeling of mushroom toxicity prediction using interpretable machine learning. With some adaptation, the techniques can extend to real-world species identification and safety applications.

3 Datasets

3.1 Hepatitis

The dataset used in this analysis contains medical records for 344 individuals. Each sample has 8 features capturing important blood test results and medical history. These features are Ascites, Varices, Bilirubin, Alk Phosphate, Sgot, Albumin, Protime, and Histology. The target variable, the last column of each data entry, is a binary label indicating whether that patient has hepatitis or not.

In exploring this dataset, we found that approximately one-thirds of individuals are hepatitis patients. All the features are continuous in nature and have been normalized to fall between 0 and 1. While features such as Bilirubin and Albumin followed a normal distribution, other features like Sgot and Varices exhibited skewed or gaussian mixture distributions (suggesting two different behavior). Features like Albumin and Protime have a tighter distribution, evident from their lower standard deviations. On the other hand, Ascites and Histology showed wider variability. Moreover, some features such as Alk Phosphate and protime had similar distributions within each class.

Correlation analysis of the features showed that some features are correlated (positively or negatively). For instance, Alk Phosphate has a positive correlation with Sgot and Albumin. This suggests that as the value of one feature increases, the value of the other tends to increase as well. Features that are highly correlated might potentially be combined or one might be removed to reduce redundancy.

Features such as Ascites, Varices, Albumin, and Histology have moderately strong correlation with the classes, while Alk Phosphate has a weak correlation with the classes. Considering this initial exploration of the dataset and the relevance of features in the context of hepatitis, we assumed that Ascites [4], Varices [8], and Albumin [3] are important and discriminative features for hepatitis prediction, while features such as Alk Phosphate are less important.

- **New Features Description:**

To improve the model performance on this dataset, the original features were engineered into new polynomial and interaction features.

- **Polynomial Features:** Polynomial features are an effective strategy to introduce a non-linear dimension to our dataset, potentially improving the performance of our model. By squaring the original features (features^2), we've added an additional layer of complexity that can help the model grasp more intricate patterns, particularly where the relationship between a feature and the outcome isn't strictly linear. For instance, a particular biomarker's effect on Hepatitis risk might not increase linearly with its value; it might increase exponentially after a certain threshold. Incorporating polynomial features can help the model identify such relationships.
- **Interaction Features:** Interaction features capture the dependencies between features and combined effect of two or more features on the target variable. In the context of our dataset, it's plausible that the combined effect of two biomarkers might be more (or less) than the sum of their individual effects on the Hepatitis risk. By creating interaction terms, we're allowing our model to identify scenarios where, for example, high values of both Ascites and Alk Phosphate together have a particularly strong indication of Hepatitis, even if their individual effects aren't as pronounced. This enables the model to harness synergistic or antagonistic relationships between features, potentially improving its predictive accuracy.
- **Ascites and Albumin:** Ascites results from high pressure in certain veins of the liver (portal hypertension) and low blood levels of a protein called Albumin. So potentially, the relationship between ascites and albumin appears to be antagonistic.
- **Ascites and Varices:** Portal hypertension, which leads to ascites, can also lead to the development of varices. Therefore, the relationship between ascites and varices can be considered synergistic. The presence of one (either ascites or varices) might indicate a higher likelihood of the presence of the other due to the common underlying cause.
- **Albumin and Protime:** Protime is a test that measures how long it takes for your blood to clot. If the liver is damaged or not functioning properly, it may not produce enough clotting proteins, leading to an increased prothrombin time. [6] If both albumin levels are low and prothrombin time is prolonged, it can indicate severe liver dysfunction.

By integrating both polynomial and interaction features, we aim to provide our logistic regression model with a richer, more detailed representation of the data, equipping it to identify and leverage complex relationships that might otherwise go unnoticed.

- **Feature selection using Recursive Feature Elimination (RFE):** Then considering all the new potential features, we implemented a RFE function to identify the most impactful features for our classification, thus potentially improving the performance of the model. RFE is a feature selection method used to find the optimal number of features for a given model, by recursively

removing the least important features based on their weights (or importance scores) in the model. [5]

The final set of engineered features contained 7 variables: Varices, Albumin², Ascites, AlbuminXProtime, AscitesXVarices, Varices², Albumin. These features were chosen based on extensive data exploration, cross validation, RFE approach and were justified by domain knowledge as explained before.

3.2 Mushroom

The Mushroom dataset is a collection of 1623 observations, primarily intended for the classification task of determining the edibility of a given mushroom sample. Each sample has 11 features and a target label (edible=0, poisonous=1). The features are naturally categorical but, in our dataset, they have continuous values. Some features such as Odor and Gill attachment are skewed. Most of the features have quite similar distribution within each class. Odor and Cap color don't have wide variation. Based on the standard deviation and distribution.

Correlation analysis shows strong correlation between some features, such as Poisonous and Cap-surface, and Gill-size, implying potential relationships that could be significant when determining the edibility of a mushroom. Stalk-color-below-ring has a strong correlation with the class making it potentially an important discriminative feature.

Moreover, in the context of our classification problem and the related domain knowledge, we assume that Poisonous and Odor are important features for identifying potentially poisonous mushrooms. The gills of the mushroom can carry significant information about its edibility. Some toxic mushrooms have specific gill colors or configurations that distinguish them from edible counterparts [7]. Therefore, the gill characteristics might be important features for our model. On the other hand, since many edible mushrooms have toxic look-alikes that can be very similar in cap appearance, features such as cap shape and color might not be very discriminative.

Similar to the Hepatitis dataset, we have explored addition of different polynomial and interaction features to be able to capture potential nonlinear or complex relations with the target variable. Moreover, we also investigated the potential impact of square root of the features. RFE analysis revealed the most important features. The final set of engineered features contained: 'Stalk-color-below-ring', 'sqrt Gill-color', 'Poisonous', 'Gill-color X Stalk-color-below-ring', 'Gill-size X Gill-color', 'sqrt Odor', 'sqrt Stalk-color-below-ring', 'Cap-surface', 'Gill-color', 'sqrt Poisonous', 'sqrt Cap-surface', 'Gill-size'.

4 Results

4.1 Hepatitis

Several experiments were conducted to evaluate and optimize the logistic regression model's performance. First, 10-fold cross-validation was used to assess accuracy of the baseline model on the original 8 blood test features. This achieved 0.70 accuracy on average across the folds.

To improve on this, polynomial features were engineered and added to model potential non-linear relationships. The cross-validation accuracy increased to 0.75 by employing the square of the features, indicating these engineered features better captured the outcome patterns. However, incorporating the cubic transformations of the original features did not improve the accuracy suggesting that the added complexity from the cubic terms did not provide any additional meaningful information for the model. Finally, the

addition of Interaction features increased the accuracy of cross validation suggesting dependencies between features and combined effect of two or more features on the Hepatitis risk.

Further, we implemented a Recursive Feature Elimination (RFE) function. The process starts by training the model utilizing all available features and then determining the importance of each feature, by either its corresponding coefficient value in the logistic regression model. Subsequently, the least significant feature was removed, and the model was retrained. This recursive elimination continued until no feature is left. By narrowing down to the most impactful features, we ensured a more interpretable model. Reducing the feature set gave further gains, increasing accuracy to around 0.8. This shows eliminating noisy, less informative or redundant features improves generalization. The final 7 feature model gave the best accuracy of 0.81.

Hyperparameter tuning is also essential for configuring machine learning models to maximize performance. Models were trained on the training data across learning rates from 0.1 to 0.00001 and resulting log loss between predictions and true labels was measured, with lower loss indicating better model calibration.

A figure plotting learning rate vs. log loss revealed 0.0001 achieved optimal loss. The optimal 0.0001 rate was used in refitting the final model on all training data before final evaluation. This smaller learning rate led to slower convergence during training, but resulted in more stable optimization and better generalization. Incorporating both engineered features and optimal hyperparameters increased model accuracy substantially compared to the baseline.

4.2 Mushroom

Similar to the previous dataset, we've implemented all the steps above to the mushroom dataset. The initial accuracy of model was around 70%. To improve this accuracy, we tried cubic, square root, square, and interaction of the features. Contrary to the previous dataset, we didn't observe a significant increase in our accuracy and it remained at 70%. Finally, by selecting the features determined by RFE, we tried to increase our accuracy, but again, we didn't have any performance improvement to our model. Plotting the log loss rate for different epsilons and learning rates, showed that the optimal learning rate is 0.001 which we applied as the default value to our fit function.

5 Discussion and conclusion

In this project, we tried several methods to develop accurate and interpretable Logistic Regression machine learning models. We've taken advantage of data analysis, domain knowledge exploration, and extensive feature exploration. We have also tried to leverage feature engineering, which improved our model accuracy on Hepatitis dataset significantly, increasing accuracy from 71% to around 80%. Hyperparameter tuning also proved important, as different learning rates for different datasets achieved optimal performance by enabling stable convergence.

Finally, by inspecting the importance of features through analyzing the models, we could pick and choose the most critical sets of features to improve accuracy.

For future works, we can incorporate other algorithms like random forests to evaluate and compare the performance and provide more feature insights, especially on Mushroom dataset since we didn't observe any meaningful changes to the accuracy of the model. Regularization techniques like LASSO could further reduce overfitting. Overall, the analysis demonstrated the substantial gains in accuracy and interpretability possible from thoughtful data analysis, feature engineering, validation, and tuning.

6 Statement of Contributions

Masoumeh Shafiei helped in data analysis, algorithm and code development, interpretation, domain knowledge, and report writing.

Mikael Mayeli helped in code development, debugging, and overcoming implementation challenges.

Mohammad Amin Najafloo helped in report writing, analyzing data, and improving code readability and comprehension.

7 Appendix

7.1 Hepatitis

Logistic Regression model

```
class LogisticRegression:
    def __init__(self):
        self.w = None # Initialize weights to None

    def predict(self, X):
        """
        Predicts binary class labels for input data X.

        :param X: Input data of shape (n_samples, n_features).
        :return: Binary class labels (True or False) for each sample.
        """
        ones_column = np.ones((X.shape[0], 1))
        X_with_bias = np.concatenate((ones_column, X), axis=1)
        return (np.dot(X_with_bias, self.w)) > 0

    def sigmoid(self, z):
        """
        Sigmoid activation function.

        :param z: Input value or array.
        :return: Sigmoid function output.
        """
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y, learning_rate=0.001, e=0.0001, max_iter=100000):
        """
        Fit the logistic regression model to the training data.

        :param X: Input data of shape (n_samples, n_features).
        :param y: Binary target labels of shape (n_samples,).
        :param learning_rate: Learning rate for gradient descent (default: 0.001).
        :param e: Convergence threshold (default: 0.0001).
        :param max_iter: Maximum number of iterations (default: 100000).
        :return: None
        """
        if self.w is None: # If weights are None, initialize them based on X's shape
            num_features = X.shape[1]
            self.w = np.zeros(num_features + 1)

        ones_column = np.ones((X.shape[0], 1))
        X_with_bias = np.concatenate((ones_column, X), axis=1)
        n_samples = len(X)
        old_w = np.ones(self.w.shape)
        while np.linalg.norm(self.w - old_w, ord=2) > e and max_iter > 0:
            old_w = self.w.copy()
            predictions = self.sigmoid(np.dot(X_with_bias, self.w))
            gradient = np.dot(X_with_bias.T, (y - predictions))
            self.w += learning_rate * gradient
            max_iter -= 1
```


Train test split function

```
def my_train_test_split(data, labels, test_size=0.2, random_state=None):  
    """  
    Custom train-test split function for data and labels.  
  
    :param data: The input data.  
    :param labels: The corresponding labels for the data.  
    :param test_size: The proportion of the data to include in the test split (default: 0.2).  
    :param random_state: Seed for random shuffling (default: None).  
  
    :return: X_train, X_test, y_train, y_test  
        - X_train: The training data.  
        - X_test: The testing data.  
        - y_train: The labels for the training data.  
        - y_test: The labels for the testing data.  
    """  
    assert len(data) == len(labels), "Data and labels must have the same length!"  
    if random_state is not None:  
        random.seed(random_state)  
  
    # Shuffle indices.  
    indices = list(data.index)  
    random.shuffle(indices)  
  
    # Calculate split index.  
    split_idx = int(len(data) * (1 - test_size))  
  
    # Split the data and labels.  
    train_indices = indices[:split_idx]  
    test_indices = indices[split_idx:]  
  
    X_train = data.loc[train_indices]  
    y_train = labels.loc[train_indices]  
  
    X_test = data.loc[test_indices]  
    y_test = labels.loc[test_indices]  
  
    return X_train, X_test, y_train, y_test
```

K fold cross validation function



```
def Accu_eval(true_labels, predictions):  
    """  
    Calculate the accuracy of a classification model.  
  
    :param true_labels: The true labels (ground truth) for the data.  
    :param predictions: The predicted labels made by a classification model.  
  
    :return: The accuracy, which is the proportion of correctly predicted labels.  
    """  
    return np.mean(true_labels == predictions)
```

```
def k_fold_split(data, k=10, random_state=None):  
    """  
    Splits the data into k folds for cross-validation.  
  
    Parameters:  
    - data: DataFrame of data points.  
    - k: Number of folds.  
    - random_state: Seed for random number generator.  
  
    Returns:  
    - folds: List of k DataFrames.  
    """  
  
    if random_state is not None:  
        np.random.seed(random_state)  
  
    # Randomly shuffle the data
```

```

    shuffled_data = data.sample(frac=1,
random_state=random_state).reset_index(
    drop=True
)

# Split the data into k roughly equal parts
n = len(shuffled_data)
fold_size = n // k
folds = []
for i in range(k):
    if i < k - 1:
        fold = shuffled_data.iloc[i * fold_size : (i + 1) * fold_size]
    else:
        # The last fold might have more than fold_size elements
        fold = shuffled_data.iloc[i * fold_size :]
    folds.append(fold)

return folds

def k_fold_cross_validation(model, data, labels, k=5, random_state=None):
    """
    Conducts k-fold cross-validation on the data with the given model.

    Parameters:
    - model: A machine learning model with fit and predict methods.
    - data: DataFrame of data points.
    - labels: DataFrame or Series of labels corresponding to the data.
    - k: Number of folds.
    - random_state: Seed for random number generator.

    Returns:
    - scores: List of accuracy scores for each fold.
    """

    data_folds = k_fold_split(data, k, random_state)
    label_folds = k_fold_split(labels, k, random_state)

    scores = []

    for i in range(k):
        # Split the data into training and validation sets
        train_data = pd.concat(
            [fold for j, fold in enumerate(data_folds) if j != i],
            ignore_index=True

```

```

    )
    train_labels = pd.concat(
        [fold for j, fold in enumerate(label_folds) if j != i],
        ignore_index=True
    )

    validation_data = data_folds[i]
    validation_labels = label_folds[i]

    # Train the model
    model.fit(train_data, train_labels)

    # Predict on the validation set
    predictions = model.predict(validation_data)

    # Calculate accuracy
    accuracy = Accu_eval(validation_labels, predictions)
    scores.append(accuracy)

```

RFE function

```

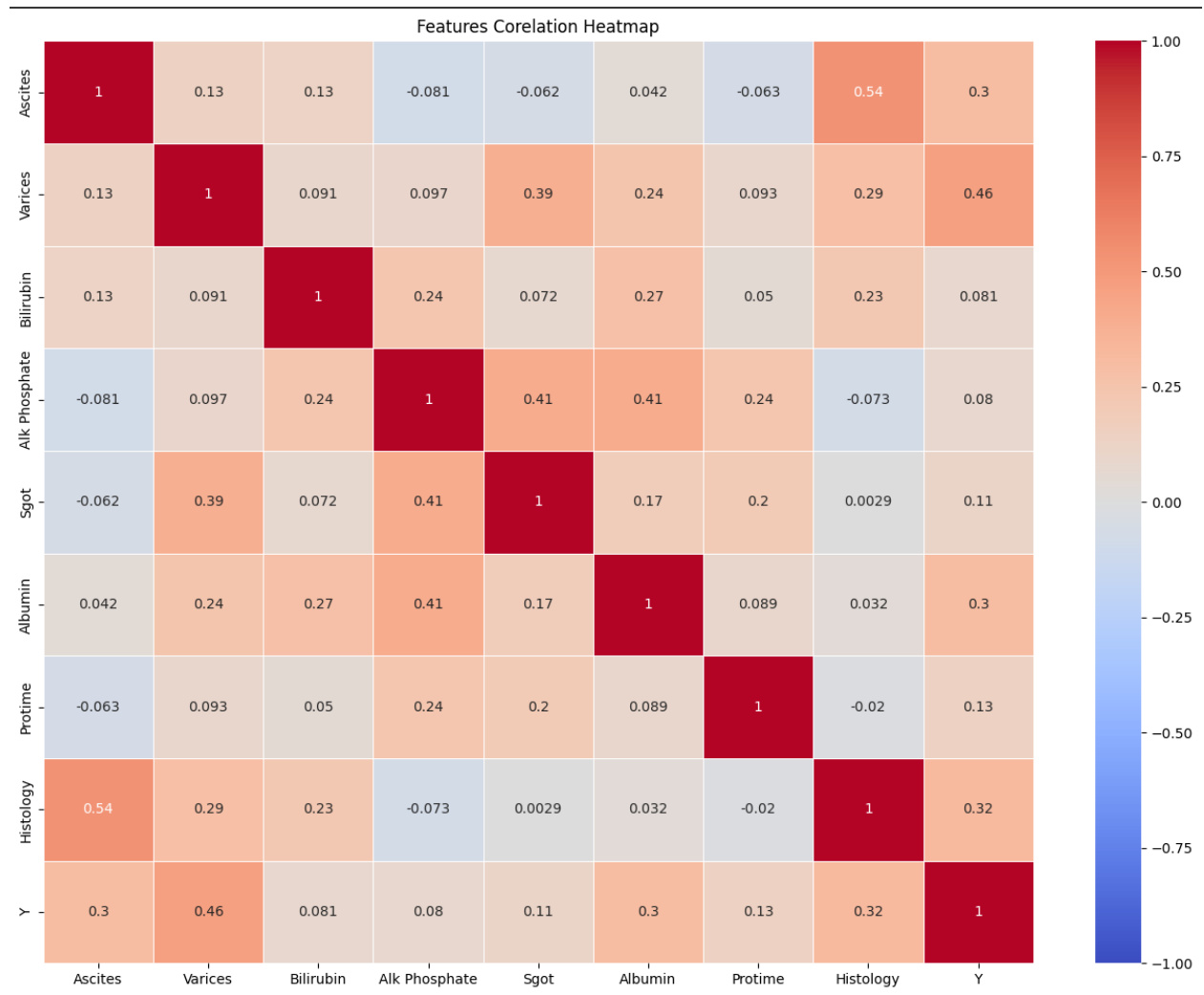
def rfe(Model_Class, X, y, train_test_split):
    """
    Recursive Feature Elimination (RFE) using a specified model class.

    :param Model_Class: The class of the model to be used for feature elimination.
    :param X: The input features.
    :param y: The target labels.
    :param train_test_split: A function for splitting the data into training and testing sets.

    :return: A dictionary containing feature names ordered by their elimination sequence.
    """
    X = X.copy()
    weights_ordered = dict()
    while True:
        X_train, _, y_train, _ = train_test_split(
            X, y, test_size=0.2, random_state=42)
        model = Model_Class()
        model.fit(X_train, y_train)
        weights = model.w[1:] # The first weight is the bias we added
        smallest_weight_index = np.argmin(weights)
        smallest_weight_name = X.columns[smallest_weight_index]
        X = X.drop(columns=smallest_weight_name)
        weights_ordered[smallest_weight_name] = len(weights)
        if len(weights) == 1:
            break
    weights_ordered_descending = dict(sorted(weights_ordered.items(), key=lambda item: item[1]))
    return weights_ordered_descending

return scores, model

```




feature Selection


basic features

```
[ ] # First we select all of the given features
    all_features = [
        "Ascites",
        "Varices",
        "Bilirubin",
        "Alk Phosphate",
        "Sgot",
        "Albumin",
        "Protime",
        "Histology",
    ]
```

Splitting data into train and test data

```
[ ] X = df.loc[:, all_features]
    y = df.loc[:, "Y"]
    X_train, X_test, y_train, y_test = my_train_test_split(
        X, y, test_size=0.2, random_state=42
    )
```

```
 weight_importance = rfe(LogisticRegression, X, y, my_train_test_split)
    print(weight_importance)
```

```
 {'Varices': 1, 'Ascites': 2, 'Albumin': 3, 'Protime': 4, 'Histology': 5, 'Alk Phosphate': 6, 'Bilirubin': 7, 'Sgot': 8}
```

Validating model with cross validation

```
[ ] model = LogisticRegression()
    scores, trained_model = k_fold_cross_validation(
        model, X_train, y_train, k=10, random_state=42
    )
    best_score_mean = np.mean(scores)
    print("best score mean: ", best_score_mean)
```

```
best score mean:  0.728587962962963
```

```
[ ] model = LogisticRegression()
    model.fit(X_train, y_train)
    Accu_eval(y_test, model.predict(X_test))
```

```
0.7101449275362319
```

Feature engineering

1. basis expansion (order 2)

```
# Extract the original features from the DataFrame
X_engineered = df.loc[:, all_features]

# Create an empty DataFrame to store the new squared features
new_features_df = pd.DataFrame()

# Iterate through the original features and create squared features
for i in range(len(all_features)):
    new_col_name = all_features[i] + "^2" # Name for the new squared feature
    new_features_df[new_col_name] = X_engineered[all_features[i]] ** 2 # Calculate the squared feature

# Append the new squared features to the original DataFrame
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# Display the first few rows of the engineered DataFrame
X_engineered.head()
```

	Ascites	Varices	Bilirubin	Alk Phosphate	Sgot	Albumin	Prottime	Histology	Ascites^2	Varices^2	Bilirubin^2	Alk Phosphate^2	Sgot^2	Albumin^2	Prottime^2	Histology^2
0	0.117650	0.77889	0.42623	0.27273	0.638300	0.57675	0.069172	0.066667	0.013842	0.606670	0.181672	0.074382	0.407427	0.332641	0.004785	0.004444
1	0.176470	0.40201	0.67213	0.31313	0.082742	0.50969	0.518360	0.100000	0.031142	0.161612	0.451759	0.098050	0.006846	0.259784	0.268697	0.010000
2	0.058824	1.00000	0.62295	0.43434	0.000000	0.63934	0.561910	0.016667	0.003460	1.000000	0.388067	0.188651	0.000000	0.408756	0.315743	0.000278
3	0.470590	0.83920	0.86885	0.46465	0.273050	0.56036	0.037148	0.366670	0.221455	0.704257	0.754900	0.215900	0.074556	0.314003	0.001380	0.134447
4	0.529410	0.72864	0.65574	0.46465	0.153660	0.56483	0.238680	0.316670	0.280275	0.530916	0.429995	0.215900	0.023611	0.319033	0.056968	0.100280

```
# Split the engineered data into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Create a LogisticRegression model instance
model = LogisticRegression()

# Perform k-fold cross-validation and get scores for each fold
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of the best scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean of the best scores
print("best score mean: ", best_score_mean)
```

best score mean: 0.7155892592592593

2. basis expansion (order 3)

```
# Extract the original features from the DataFrame
X_engineered = df.loc[:, all_features]

# Create an empty DataFrame to store the new cubic features
new_features_df = pd.DataFrame()

# Iterate through the original features and create cubic features
for i in range(len(all_features)):
    new_col_name = all_features[i] + "^3" # Name for the new cubic feature
    new_features_df[new_col_name] = X_engineered[all_features[i]] ** 3 # Calculate the cubic feature

# Append the new cubic features to the original DataFrame
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# Display the first few rows of the engineered DataFrame
X_engineered.head()
```

	Ascites	Varices	Bilirubin	Alk Phosphate	Sgot	Albumin	Protine	Histology	Ascites^3	Varices^3	Bilirubin^3	Alk Phosphate^3	Sgot^3	Albumin^3	Protine^3	Histology^3
0	0.117650	0.77889	0.42623	0.27273	0.638300	0.57675	0.069172	0.066667	0.001628	0.472529	0.077434	0.020286	0.260061	0.191850	0.000331	0.000296
1	0.176470	0.40201	0.67213	0.31313	0.082742	0.50969	0.518360	0.100000	0.005496	0.064970	0.303641	0.030703	0.000566	0.132409	0.139282	0.001000
2	0.058824	1.00000	0.62295	0.43434	0.000000	0.63934	0.561910	0.016667	0.000204	1.000000	0.241746	0.081939	0.000000	0.261334	0.177419	0.000005
3	0.470590	0.83920	0.86885	0.46465	0.273050	0.56036	0.037148	0.366670	0.104214	0.591012	0.655895	0.100318	0.020358	0.175955	0.000051	0.049298
4	0.529410	0.72864	0.65574	0.46465	0.153660	0.56483	0.238680	0.316670	0.148380	0.386847	0.281965	0.100318	0.003628	0.180199	0.013597	0.031756

```
# Split the engineered data into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Create a LogisticRegression model instance
model = LogisticRegression()

# Perform k-fold cross-validation and get scores for each fold
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of the scores obtained from the cross-validation
best_score_mean = np.mean(scores)

# Print the mean score as an estimate of model performance
print("best score mean: ", best_score_mean)
```

best score mean: 0.7155092592592593

3. interaction terms

```
# Extract the original features from the DataFrame
X_engineered = df.loc[:, all_features]

# Create an empty DataFrame to store the new interaction features
new_features_df = pd.DataFrame()

# Iterate through pairs of original features to create interaction features
for i in range(len(all_features)):
    for j in range(i + 1, len(all_features)):
        new_col_name = all_features[i] + "x" + all_features[j] # Name for the new interaction feature
        new_features_df[new_col_name] = (
            X_engineered[all_features[i]] * X_engineered[all_features[j]] # Calculate the interaction feature
        )

# Append the new interaction features to the original DataFrame
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# To update a selected_features list (not shown in the code, assuming you have this list)

# Display the first few rows of the engineered DataFrame
X_engineered.head()
```

```
8
```

	Ascites	Varices	Bilirubin	Alk Phosphate	Sgot	Albumin	Protine	Histology	AscitesXVarices	AscitesXBilirubin	...	Alk PhosphateXSgot	Alk PhosphateXAlbumin	Alk PhosphateXProtine	Alk PhosphateXHistology	SgotXAlbumin	SgotXProtine	SgotXHistology	AlbuminXProtine	AlbuminXHistology	ProtineXHistology
0	0.117650	0.77889	0.42623	0.27273	0.638300	0.57675	0.069172	0.066667	0.091636	0.050146	...	0.174084	0.157297	0.018865	0.018182	0.368140	0.044152	0.042554	0.039895	0.038450	0.004611
1	0.176470	0.40201	0.67213	0.31313	0.082742	0.50969	0.518360	0.100000	0.070943	0.118611	...	0.025909	0.159999	0.162314	0.031313	0.042173	0.042890	0.008274	0.264203	0.050969	0.051836
2	0.058824	1.00000	0.62295	0.43434	0.000000	0.63934	0.561910	0.016667	0.058824	0.036644	...	0.000000	0.277691	0.244060	0.007239	0.000000	0.000000	0.000000	0.359252	0.010656	0.009365
3	0.470590	0.83920	0.86885	0.46465	0.273050	0.56036	0.037148	0.366670	0.394919	0.408872	...	0.126873	0.260371	0.017261	0.170373	0.153006	0.010143	0.100119	0.020816	0.205467	0.013621
4	0.529410	0.72864	0.65574	0.46465	0.153660	0.56483	0.238680	0.316670	0.305749	0.347155	...	0.071398	0.262448	0.110903	0.147141	0.086792	0.036676	0.048660	0.134814	0.178865	0.075583

5 rows x 36 columns

```
[ ] # Split the engineered data into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Create a LogisticRegression model instance
model = LogisticRegression()

# Perform k-fold cross-validation and get scores for each fold
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of the scores obtained from the cross-validation
best_score_mean = np.mean(scores)

# Print the mean score as an estimate of model performance
print("best score mean: ", best_score_mean)
```

best score mean: 0.7558925925925926

4. Removing unnecessary features

```
[ ] # based on the distribution and correlation of the features with the classes, we assume Bilirubin,  
# Alk Phosphate, and sgot are not that informative and discriminative. So, we remove them  
selected_features = ["Ascites", "Varices", "Albumin", "Protime", "Histology"]
```

```
X_engineered = df.loc[:, selected_features]  
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(  
    X_engineered, y, test_size=0.2, random_state=42  
)  
X_engineered.head()
```

	Ascites	Varices	Albumin	Protime	Histology
0	0.117650	0.77889	0.57675	0.069172	0.066667
1	0.176470	0.40201	0.50969	0.518360	0.100000
2	0.058824	1.00000	0.63934	0.561910	0.016667
3	0.470590	0.83920	0.56036	0.037148	0.366670
4	0.529410	0.72864	0.56483	0.238680	0.316670

```
[ ] model = LogisticRegression()  
  
scores, trained_model = k_fold_cross_validation(  
    model, X_engineered_train, y_train, k=10, random_state=42  
)  
best_score_mean = np.mean(scores)  
print("best score mean: ", best_score_mean)
```

best score mean: 0.7291666666666666

```
[ ] weight_importance = rfe(LogisticRegression, X_engineered, y, my_train_test_split)  
print(weight_importance)  
  
{'Varices': 1, 'Ascites': 2, 'Albumin': 3, 'Protime': 4, 'Histology': 5}
```

```
[ ] # Create an empty list to store the names of the new features  
new_features = []  
  
# Iterate through pairs of selected features to create interaction features  
for i in range(len(selected_features)):  
    for j in range(i + 1, len(selected_features)):  
        new_col_name = selected_features[i] + "X" + selected_features[j] # Name for the new interaction feature  
        new_features.append(new_col_name)  
        X_engineered[new_col_name] = (  
            X_engineered[selected_features[i]] * X_engineered[selected_features[j]] # Calculate the interaction feature  
        )  
  
# Iterate through selected features to create squared features  
for i in range(len(selected_features)):  
    new_col_name = selected_features[i] + "^2" # Name for the new squared feature  
    new_features.append(new_col_name)  
    X_engineered[new_col_name] = X_engineered[selected_features[i]] ** 2 # Calculate the squared feature  
  
# Update the selected_features list by adding the new features  
selected_features += new_features
```

Testing different learning rates

```
[ ] def compute_log_loss(y_true, y_pred_prob):
    """
    Calculate the log loss (logarithmic loss) between true binary labels and predicted probabilities.

    :param y_true: True binary labels (0 or 1).
    :param y_pred_prob: Predicted probabilities of class membership (between 0 and 1).

    :return: Log loss value.
    """
    epsilon = 1e-15 # A small constant to avoid division by zero or taking the logarithm of zero
    y_pred_prob = np.clip(y_pred_prob, epsilon, 1 - epsilon) # Clip probabilities to avoid extreme values
    loss = -y_true * np.log(y_pred_prob) - (1 - y_true) * np.log(1 - y_pred_prob) # Calculate log loss element-wise
    return np.mean(loss) # Return the mean log loss value


[ ] def evaluate_learning_rate(model, X, y, learning_rate, e=0.00001, max_iter=10000):
    """
    Evaluate a logistic regression model's performance for a specific learning rate.

    :param model: An instance of the LogisticRegression class.
    :param X: Input features.
    :param y: True binary labels (0 or 1).
    :param learning_rate: Learning rate for gradient descent.
    :param e: Convergence threshold for the model (default: 0.00001).
    :param max_iter: Maximum number of iterations for gradient descent (default: 10000).

    :return: Log loss value for the given learning rate.
    """
    model.fit(X, y, learning_rate=learning_rate, e=e, max_iter=max_iter) # Train the model with the specified learning rate
    predictions_prob = model.sigmoid(np.dot(X, model.w[1:]) + model.w[0]) # Calculate predicted probabilities
    return compute_log_loss(y, predictions_prob) # Compute and return the log loss value
```

```
# List of learning rates to try
learning_rates = [0.1, 0.01, 0.001, 0.0001, 0.00001]
epsilon = [0.1, 0.01, 0.001, 0.0001, 0.00001]
lrXe = []

losses = []

# Iterate through different learning rates and epsilon values
for lr in learning_rates:
    for e in epsilon:
        lrXe.append(str(lr) + "X" + str(e)) # Combine lr and e as a label
        model = LogisticRegression()
        loss = evaluate_learning_rate(model, X_final_train, y_train, learning_rate=lr, e=e)
        losses.append(loss)

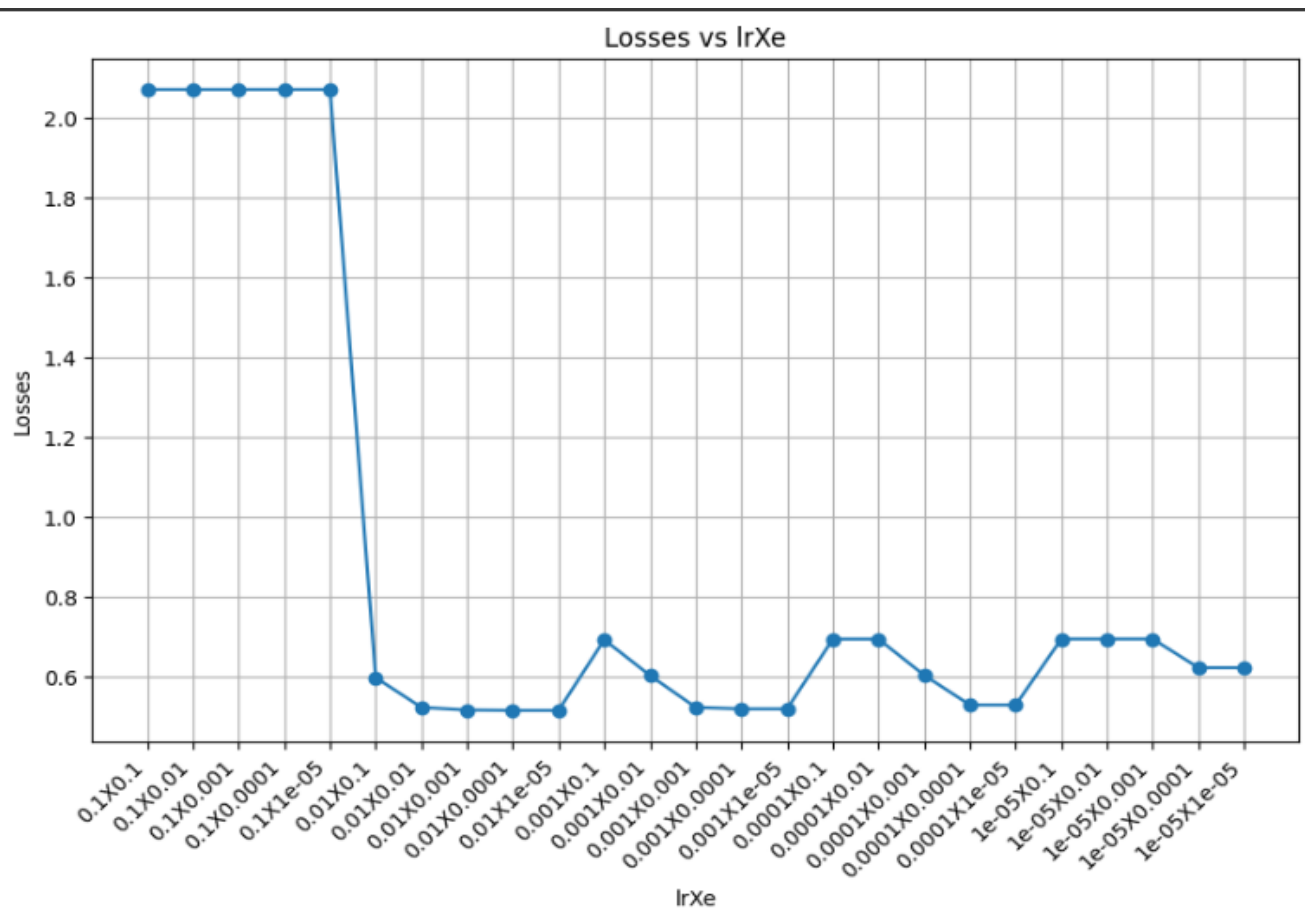
# Plotting the results
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(lrXe, losses, marker='o') # Plotting lrXe against losses

# Adding title and labels
plt.title('Losses vs lrXe')
plt.xlabel('lrXe')
plt.ylabel('Losses')

# Rotate x labels for better readability
plt.xticks(rotation=45, ha="right")

# Display the plot
plt.grid(True)
plt.show()
```



7.2 Mushrooms

Logistic Regression model

```
# Define a LogisticRegression class for binary classification
class LogisticRegression:
    def __init__(self):
        self.w = None # Initialize model weights to None

    def predict(self, X):
        """
        Predict binary class labels (0 or 1) based on input features.

        :param X: Input feature matrix.
        :return: Binary predictions.
        """
        ones_column = np.ones((X.shape[0], 1)) # Add a bias column
        X_with_bias = np.concatenate((ones_column, X), axis=1) # Add bias term to input
        return (np.dot(X_with_bias, self.w)) > 0 # Predict based on dot product with model weights

    def sigmoid(self, z):
        """
        Calculate the sigmoid function.

        :param z: Input values.
        :return: Sigmoid function output.
        """
        return 1 / (1 + np.exp(-z)) # Calculate sigmoid of input

    def fit(self, X, y, learning_rate=0.001, e=0.001, max_iter=200000):
        """
        Train the logistic regression model using gradient descent.

        :param X: Input feature matrix.
        :param y: True binary labels (0 or 1).
        :param learning_rate: Learning rate for gradient descent (default: 0.001).
        :param e: Convergence threshold (default: 0.001).
        :param max_iter: Maximum number of iterations (default: 200,000).
        """
        if self.w is None: # If weights are None, initialize them based on the number of features
            num_features = X.shape[1]
            self.w = np.zeros(num_features + 1)

        ones_column = np.ones((X.shape[0], 1)) # Add a bias column
        X_with_bias = np.concatenate((ones_column, X), axis=1) # Add bias term to input
        old_w = np.ones(self.w.shape)
        while np.linalg.norm(self.w - old_w, ord=2) > e and max_iter > 0:
            old_w = self.w.copy()
            predictions = self.sigmoid(np.dot(X_with_bias, self.w)) # Calculate predicted probabilities
            gradient = np.dot(X_with_bias.T, (y - predictions)) # Compute gradient
            self.w += learning_rate * gradient # Update model weights using gradient descent
            max_iter -= 1
        return
```

Train test split function

```
def my_train_test_split(data, labels, test_size=0.2, random_state=None):
    """
    Custom train-test split function for data and labels.

    :param data: Input data (features).
    :param labels: Corresponding labels.
    :param test_size: Fraction of the data to be allocated as the test set (default: 0.2).
    :param random_state: Seed for random number generation (default: None).

    :return: X_train, X_test, y_train, y_test
    """
    assert len(data) == len(labels), "Data and labels must have the same length!" # Ensure data and labels have the same length

    if random_state is not None:
        random.seed(random_state) # Set a random seed if provided

    # Shuffle indices.
    indices = list(data.index)
    random.shuffle(indices) # Shuffle the indices randomly

    random.seed(10) # Reset the random seed (not needed)
    random.shuffle(indices) # Shuffle the indices again (possibly redundant)

    # Calculate the split index based on the test_size fraction.
    split_idx = int(len(data) * (1 - test_size))

    # Split the data and labels based on the calculated indices.
    train_indices = indices[:split_idx]
    test_indices = indices[split_idx:]

    # Create training and testing sets using the selected indices.
    X_train = data.loc[train_indices]
    y_train = labels.loc[train_indices]

    X_test = data.loc[test_indices]
    y_test = labels.loc[test_indices]

    return X_train, X_test, y_train, y_test
```

K fold cross validation function

```
def Accu_eval(true_labels, predictions):
    """
    Calculate the accuracy of classification predictions.

    :param true_labels: True labels or ground truth.
    :param predictions: Predicted labels made by the model.

    :return: Accuracy value (proportion of correct predictions).
    """
    return np.mean(true_labels == predictions) # Calculate accuracy as the proportion of correct predictions
```

```

def k_fold_split(data, k=10, random_state=None):
    """
    Splits the data into k folds for cross-validation.

    Parameters:
    - data: DataFrame of data points.
    - k: Number of folds.
    - random_state: Seed for random number generator.

    Returns:
    - folds: List of k DataFrames.
    """

    if random_state is not None:
        np.random.seed(random_state)

    # Randomly shuffle the data
    shuffled_data = data.sample(frac=1,
                                random_state=random_state).reset_index(
                                    drop=True
                                )

    # Split the data into k roughly equal parts
    n = len(shuffled_data)
    fold_size = n // k
    folds = []
    for i in range(k):
        if i < k - 1:
            fold = shuffled_data.iloc[i * fold_size : (i + 1) * fold_size]
        else:
            # The last fold might have more than fold_size elements
            fold = shuffled_data.iloc[i * fold_size :]
        folds.append(fold)

    return folds

def k_fold_cross_validation(model, data, labels, k=10, random_state=None):
    """
    Conducts k-fold cross-validation on the data with the given model.

    Parameters:
    - model: A machine learning model with fit and predict methods.
    - data: DataFrame of data points.
    - labels: DataFrame or Series of labels corresponding to the data.

```

```

- k: Number of folds.
- random_state: Seed for random number generator.

Returns:
- scores: List of accuracy scores for each fold.
"""

data_folds = k_fold_split(data, k, random_state)
label_folds = k_fold_split(labels, k, random_state)

scores = []

for i in range(k):
    # Split the data into training and validation sets
    train_data = pd.concat(
        [fold for j, fold in enumerate(data_folds) if j != i],
        ignore_index=True
    )
    train_labels = pd.concat(
        [fold for j, fold in enumerate(label_folds) if j != i],
        ignore_index=True
    )

    validation_data = data_folds[i]
    validation_labels = label_folds[i]

    # Train the model
    model.fit(train_data, train_labels)

    # Predict on the validation set
    predictions = model.predict(validation_data)

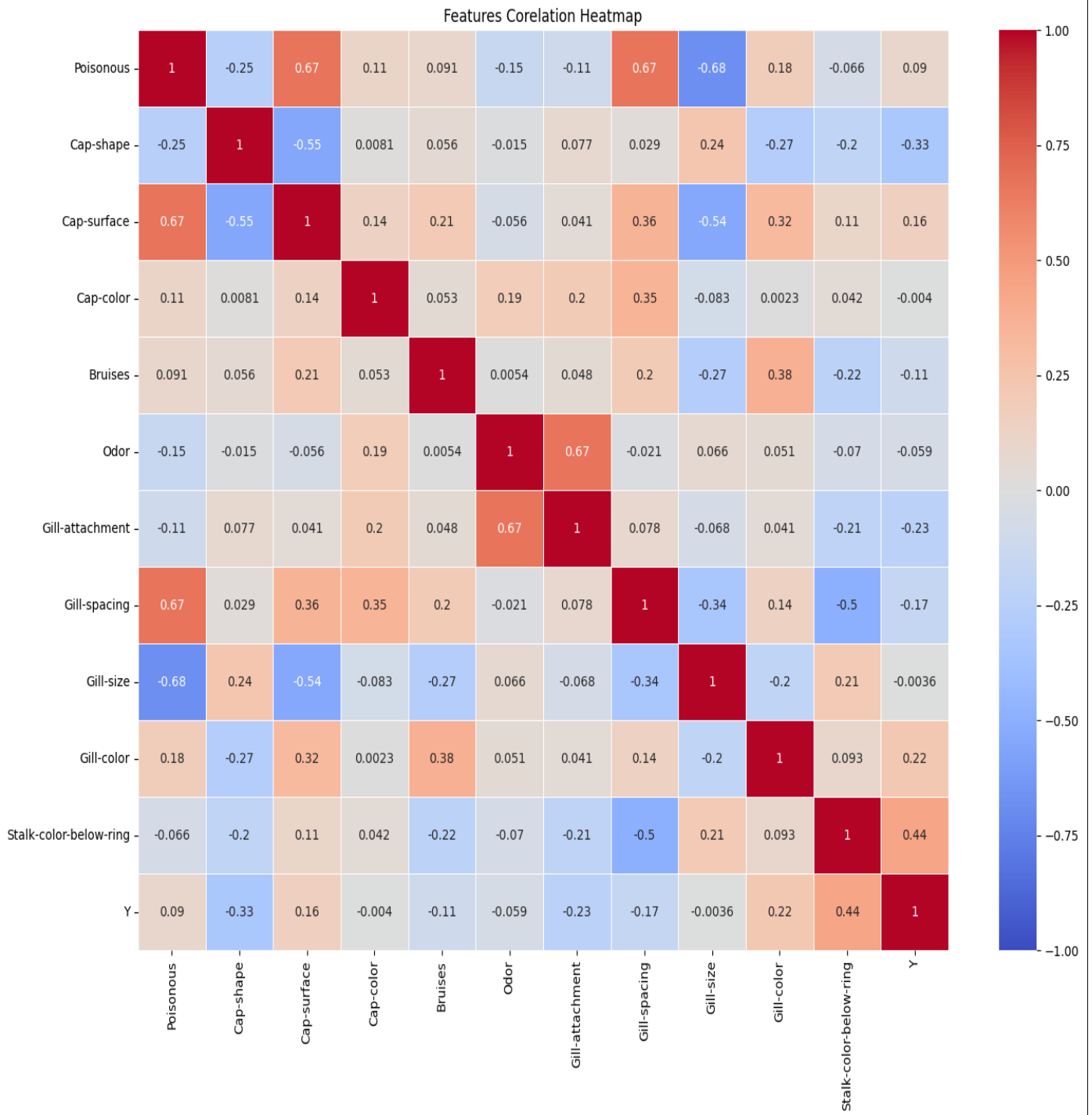
    # Calculate accuracy
    accuracy = Accu_eval(validation_labels, predictions)
    scores.append(accuracy)

return scores, model

```


RFE function

```
def rfe(Model_Class, X, y, train_test_split):  
    """  
    Perform Recursive Feature Elimination (RFE) to rank and select features.  
  
    Parameters:  
    - Model_Class: A machine learning model class with fit and predict methods.  
    - X: Input features (DataFrame or array-like).  
    - y: Target labels.  
    - train_test_split: A function for splitting the data into training and testing sets.  
  
    Returns:  
    - weights_ordered_descending: A dictionary of feature names ordered by importance (descending).  
    """  
  
    X = X.copy() # Create a copy of the input features  
    weights_ordered = dict() # Initialize a dictionary to store feature weights  
  
    while True:  
        # Split the data into a training set (80%) and an unused test set (20%)  
        X_train, _, y_train, _ = train_test_split(  
            X, y, test_size=0.2, random_state=42  
        )  
  
        # Create an instance of the provided machine learning model  
        model = Model_Class()  
  
        # Fit the model to the training data  
        model.fit(X_train, y_train)  
  
        # Get the feature weights (excluding the bias)  
        weights = model.w[1:]  
  
        # Find the index of the smallest weight  
        smallest_weight_index = np.argmin(weights)  
  
        # Get the name of the feature with the smallest weight  
        smallest_weight_name = X.columns[smallest_weight_index]  
  
        # Drop the feature with the smallest weight from the dataset  
        X = X.drop(columns=smallest_weight_name)  
  
        # Store the name and length of remaining features  
        weights_ordered[smallest_weight_name] = len(weights)  
  
        # If only one feature remains, exit the loop  
        if len(weights) == 1:  
            break  
  
        # Sort the feature names by their lengths (importance) in descending order  
        weights_ordered_descending = dict(sorted(weights_ordered.items(), key=lambda item: item[1], reverse=True))  
  
    return weights_ordered_descending
```



feature Selection

basic features

```
[ ] # First we select all of the given features
all_features = [
    "Poisonous",
    "Cap-shape",
    "Cap-surface",
    "Cap-color",
    "Bruises",
    "Odor",
    "Gill-attachment",
    "Gill-spacing",
    "Gill-size",
    "Gill-color",
    "Stalk-color-below-ring"
]
```

Splitting data into train and test data

```
[ ] X = df.loc[:, all_features]
y = df.loc[:, "V"]
X_train, X_test, y_train, y_test = my_train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

[+ Code](#)[+ Text](#)

Validating model with cross validation

```
model = LogisticRegression()
scores, trained_model = k_fold_cross_validation(
    model, X_train, y_train, k=10, random_state=42
)
best_score_mean = np.mean(scores)
print("best score mean: ", best_score_mean)
```

best score mean: 0.7584450857239857

```
[ ] model = LogisticRegression()
model.fit(X_train, y_train)
print(Accu_eval(y_test, model.predict(X_test)))
```

0.7046153846153846

```
[ ] weight_importance = rfe(LogisticRegression, X, y, my_train_test_split)
print(weight_importance)
```

{'Cap-shape': 11, 'Gill-attachment': 10, 'Bruises': 9, 'Gill-spacing': 8, 'Cap-color': 7, 'Gill-size': 6, 'Odor': 5, 'Cap-surface': 4, 'Poisonous': 3, 'Gill-color': 2, 'Stalk-color-below-ring': 1}

▼ Feature engineering

1. basis expansion (order 2)

```
# Select the desired features from the DataFrame 'df'
X_engineered = df.loc[:, all_features]

# Initialize an empty DataFrame to store the new squared features
new_features_df = pd.DataFrame()

# Iterate over each feature in 'all_features'
for i in range(len(all_features)):
    # Define a new column name by appending "^2" to the original feature name
    new_col_name = all_features[i] + "^2"

    # Calculate the squared values of the current feature and add it as a new column
    new_features_df[new_col_name] = X_engineered[all_features[i]] ** 2

# If you want to append these new features to the original DataFrame, you can concatenate them
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# Display the first few rows of the updated 'X_engineered' DataFrame
X_engineered.head()
```

	Poisonous	Cap-shape	Cap-surface	Cap-color	Bruises	Odor	Gill-attachment	Gill-spacing	Gill-size	Gill-color	...	Cap-shape*2	Cap-surface*2	Cap-color*2	Bruises*2	Odor*2	Gill-attachment*2	Gill-spacing*2	Gill-size*2	Gill-color*2	Stalk-color-below-ring*2
0	0.24779	0.35274	0.10	0.102740	0.113520	0.211270	0.095406	0.53485	0.66142	0.21557	...	0.124426	0.0100	0.010556	0.012887	0.044635	0.009102	0.286065	0.437476	0.046470	0.136331
1	0.19469	0.32192	0.06	0.349320	0.080134	0.140850	0.042403	0.44974	0.52756	0.15569	...	0.103632	0.0036	0.122024	0.006421	0.019839	0.001798	0.202266	0.278320	0.024239	0.136331
2	0.19469	0.32192	0.06	0.349320	0.080134	0.140850	0.042403	0.44974	0.52756	0.15569	...	0.103632	0.0036	0.122024	0.006421	0.019839	0.001798	0.202266	0.278320	0.024239	0.136331
3	0.13274	0.30137	0.00	0.089041	0.111850	0.070423	0.010601	0.34703	0.66829	0.12575	...	0.090824	0.0000	0.007928	0.012510	0.004959	0.000112	0.120430	0.447949	0.015813	0.227453
4	0.24779	0.27397	0.13	0.102740	0.110180	0.464790	0.194350	0.38225	0.54331	0.15569	...	0.075060	0.0169	0.010556	0.012140	0.216030	0.037772	0.146115	0.295186	0.024239	0.136331

5 rows x 22 columns

```
# Split the engineered feature dataset into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

2. basis expansion (order 3)

```
# Create a new dataframe 'new_features_df' to store cubed features
new_features_df = pd.DataFrame()

# Iterate through each feature in 'all_features'
for i in range(len(all_features)):
    # Generate a new column name for cubed features
    new_col_name = all_features[i] + "^3"

    # Calculate and add cubed values as a new column in 'new_features_df'
    new_features_df[new_col_name] = X_engineered[all_features[i]] ** 3

# Append the newly created cubed features to the original dataframe 'X_engineered'
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# Display the first few rows of the updated 'X_engineered' dataframe
X_engineered.head()
```

```

Poisonous  Cap-shape  Cap-surface  Cap-color  Bruises  Odor  Gill-attachment  Gill-spacing  Gill-size  Gill-color  ...  Cap-shape^3  Cap-surface^3  Cap-color^3  Bruises^3  Odor^3  Gill-attachment^3  Gill-spacing^3  Gill-size^3  Gill-color^3  Stalk-color-below-ring^3
0      0.24779      0.35274         0.10    0.102740    0.113520    0.211270         0.095406      0.53485      0.66142      0.21557  ...      0.043890      0.001000      0.001084      0.001463      0.009430      0.000868      0.153002      0.289356      0.010018      0.050337
1      0.19469      0.32192         0.06    0.349320    0.080134    0.140850         0.042403      0.44974      0.52756      0.15569  ...      0.033361      0.000216      0.042626      0.000515      0.002794      0.000076      0.090967      0.146830      0.003774      0.050337
2      0.19469      0.32192         0.06    0.349320    0.080134    0.140850         0.042403      0.44974      0.52756      0.15569  ...      0.033361      0.000216      0.042626      0.000515      0.002794      0.000076      0.090967      0.146830      0.003774      0.050337
3      0.13274      0.30137         0.00    0.089041    0.111850    0.070423         0.010601      0.34703      0.66929      0.12575  ...      0.027372      0.000000      0.000706      0.001399      0.000349      0.000001      0.041793      0.299808      0.001988      0.108477
4      0.24779      0.27397         0.13    0.102740    0.110180    0.464790         0.194350      0.38225      0.54331      0.15569  ...      0.020564      0.002197      0.001084      0.001338      0.100408      0.007341      0.055852      0.160377      0.003774      0.050337

```

5 rows x 33 columns

```
# Split the engineered feature dataset into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

best score mean: 0.7598144061562836

```
[ ] # Create a new dataframe 'new_features_df' to store interaction features
new_features_df = pd.DataFrame()

# Iterate through each pair of features in 'all_features'
for i in range(len(all_features)):
    for j in range(i + 1, len(all_features)):
        # Generate a new column name for interaction features
        new_col_name = all_features[i] + "x" + all_features[j]

        # Calculate and add interaction values as a new column in 'new_features_df'
        new_features_df[new_col_name] = (
            X_engineered[all_features[i]] * X_engineered[all_features[j]]
        )

# Append the newly created interaction features to the original dataframe 'X_engineered'
X_engineered = pd.concat([X_engineered, new_features_df], axis=1)

# Update the 'selected_features' list with the new interaction features
X_engineered.head()
```

	Poisonous	Cap- shape	Cap- surface	Cap- color	Bruises	Odor	Gill- attachment	Gill- spacing	Gill- size	Gill- color	...	Gill-attachmentxGill- spacing	Gill-attachmentxGill- size	Gill-attachmentxGill- color	Stalk-color- below-ring	Gill-spacingxGill- size	Gill-spacingxGill- color	Gill-spacingxStalk-color- below-ring	Gill-sizexGill- color	Gill-sizexStalk-color- below-ring	Gill-colorxStalk-color- below-ring
0	0.24779	0.35274	0.10	0.102740	0.113520	0.211270	0.095406	0.53405	0.66142	0.21557	...	0.051028	0.063103	0.020557	0.035227	0.353760	0.115298	0.197483	0.142582	0.244216	0.079595
1	0.19469	0.32192	0.06	0.349320	0.080134	0.140860	0.042403	0.44974	0.52756	0.15569	...	0.019070	0.022370	0.006602	0.015656	0.237265	0.070020	0.166058	0.082136	0.194791	0.057485
2	0.19469	0.32192	0.06	0.349320	0.080134	0.140860	0.042403	0.44974	0.52756	0.15569	...	0.019070	0.022370	0.006602	0.015656	0.237265	0.070020	0.166058	0.082136	0.194791	0.057485
3	0.13274	0.30137	0.00	0.089041	0.111050	0.070423	0.010601	0.34703	0.66829	0.12575	...	0.003679	0.007095	0.001333	0.005056	0.232264	0.043639	0.165506	0.084163	0.319198	0.059973
4	0.24779	0.27397	0.13	0.102740	0.110180	0.454790	0.194350	0.38225	0.54331	0.15569	...	0.074290	0.105592	0.030258	0.071760	0.207680	0.059513	0.141138	0.084588	0.200606	0.057485

5 rows × 88 columns

```
0 # Split the engineered feature dataset into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data (using 10 folds for illustration)
# Note: The code mentions using 10 folds but actually uses 5 for the sake of time
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

best score mean: 0.7656848893792791

4. Sqrt terms

```
# Create an empty list to store new feature names
new_features = []

# Create a new dataframe 'new_features_df' to store square root features
new_features_df = pd.DataFrame()

# Iterate through each feature in 'all_features'
for i in range(len(all_features)):
    # Generate a new column name for square root features
    new_col_name = "sqrt" + all_features[i]

    # Append the new feature name to the 'new_features' list
    new_features.append(new_col_name)

    # Calculate and add square root values as a new column in 'new_features_df'
    X[new_col_name] = np.sqrt(X[all_features[i]])

# Append the newly created square root features to the original dataframe 'X_engineered'
X_engineered = pd.concat([X, new_features_df], axis=1)
```

```
[ ] # Split the engineered feature dataset into training and testing sets
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)

# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data (using 10 folds for illustration)
# Note: The code mentions using 10 folds but actually uses 5 for the sake of time
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

```
best score mean: 0.7607254003281843
```

5. Removing unnecessary features

```
[ ] # based on the distribution and correlation of the features with the classes, we assume Stalk-color-below-ring,
# Cap-shape, Gill-color, Stalk-color-below-ring, and Gill-attachment are not that informative and discriminative. So, we remove them
selected_features = [
    'Stalk-color-below-ring',
    'Gill-color',
    'Poisonous',
    'Gill-size',
    'Odor',
    'Cap-surface'
]
```

```
X_engineered = df.loc[:, selected_features]
X_engineered_train, X_engineered_test, y_train, y_test = my_train_test_split(
    X_engineered, y, test_size=0.2, random_state=42
)
X_engineered.head()
```

	Stalk-color-below-ring	Gill-color	Poisonous	Gill-size	Odor	Cap-surface
0	0.36923	0.21557	0.24779	0.66142	0.211270	0.10
1	0.36923	0.15569	0.19469	0.52756	0.140850	0.06
2	0.36923	0.15569	0.19469	0.52756	0.140850	0.06
3	0.47692	0.12575	0.13274	0.66929	0.070423	0.00
4	0.36923	0.15569	0.24779	0.54331	0.464790	0.13

```
# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data (using 10 folds for illustration)
# Note: The code mentions using 10 folds but actually uses 5 for the sake of time
scores, trained_model = k_fold_cross_validation(
    model, X_engineered_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

```
best score mean: 0.7199569965484072
```



```
[ ] new_features = []

new_features.append("sqrtOdor")
X_engineered[new_features[-1]] = np.sqrt(X_engineered["Odor"])

new_features.append("sqrtGill-color")
X_engineered[new_features[-1]] = np.sqrt(X_engineered["Gill-color"])

new_features.append("Gill-colorXStalk-color-below-ring")
X_engineered[new_features[-1]] = X_engineered["Gill-color"] * X_engineered["Stalk-color-below-ring"]

new_features.append("Gill-sizeXGill-color")
X_engineered[new_features[-1]] = X_engineered["Gill-size"] * X_engineered["Gill-color"]

new_features.append("sqrtStalk-color-below-ring")
X_engineered[new_features[-1]] = np.sqrt(X_engineered["Stalk-color-below-ring"])

new_features.append('sqrtPoisonous')
X_engineered[new_features[-1]] = np.sqrt(X_engineered['Poisonous'])

new_features.append('sqrtCap-surface')
X_engineered[new_features[-1]] = np.sqrt(X_engineered['Cap-surface'])

[ ] selected_features = [
    'Stalk-color-below-ring', 'sqrtGill-color', 'Poisonous', 'Gill-colorXStalk-color-below-ring', 'Gill-sizeXGill-color', 'sqrtOdor',
    'sqrtStalk-color-below-ring', 'Cap-surface', 'Gill-color', 'sqrtPoisonous', 'sqrtCap-surface', 'Gill-size'
]

# Create a final dataset 'X_final' containing only the selected features
X_final = X_engineered.loc[:, selected_features]

# Split 'X_final' into training and testing sets
X_final_train, X_final_test, y_train, y_test = my_train_test_split(
    X_final, y, test_size=0.2, random_state=42
)

# Instantiate a logistic regression model
model = LogisticRegression()

# Perform k-fold cross-validation on the training data (using 10 folds for illustration)
# Note: The code mentions using 10 folds but actually uses 5 for the sake of time
scores, trained_model = k_fold_cross_validation(
    model, X_final_train, y_train, k=10, random_state=42
)

# Calculate the mean of accuracy scores obtained from cross-validation
best_score_mean = np.mean(scores)

# Print the mean accuracy score as the evaluation result
print("best score mean: ", best_score_mean)
```

best score mean: 0.7267979403610026

Testing different learning rates

```
[ ] def compute_log_loss(y_true, y_pred_prob):  
    # Set a small epsilon value to avoid numerical instability  
    epsilon = 1e-15  
  
    # Clip predicted probabilities to ensure they are within a valid range  
    y_pred_prob = np.clip(y_pred_prob, epsilon, 1 - epsilon)  
  
    # Compute the log loss using the formula for binary cross-entropy  
    loss = -y_true * np.log(y_pred_prob) - (1 - y_true) * np.log(1 - y_pred_prob)  
  
    # Calculate the mean log loss over all samples  
    return np.mean(loss)
```

```
▶ def evaluate_learning_rate(model, X, y, learning_rate, e=0.00001, max_iter=10000):  
    # Train the model with the specified learning rate, early stopping criteria, and max iterations  
    model.fit(X, y, learning_rate=learning_rate, e=e, max_iter=max_iter)  
  
    # Calculate the predicted probabilities using the trained model  
    predictions_prob = model.sigmoid(np.dot(X, model.w[1:]) + model.w[0])  
  
    # Compute and return the log loss between true labels and predicted probabilities  
    return compute_log_loss(y, predictions_prob)
```

```

# List of learning rates to try
learning_rates = [0.1, 0.01, 0.001, 0.0001, 0.00001]

# List of epsilon (early stopping criteria) values to try
epsilon = [0.1, 0.01, 0.001, 0.0001, 0.00001]

# Initialize an empty list to store the combinations of learning rates and epsilon values
lrXe = []

# Initialize an empty list to store the corresponding losses
losses = []

# Iterate over each learning rate and epsilon combination
for lr in learning_rates:
    for e in epsilon:
        # Create a string representation of the combination (e.g., "0.1X0.1")
        lrXe.append(str(lr) + "X" + str(e))

        # Instantiate a logistic regression model
        model = LogisticRegression()

        # Evaluate the model's performance for the current combination of lr and e
        loss = evaluate_learning_rate(model, X_final_train, y_train, learning_rate=lr, e=e)

        # Append the loss to the 'losses' list
        losses.append(loss)

# Create a figure for the plot
plt.figure(figsize=(10,6))

# Plot the losses against the lrXe combinations, using markers for data points
plt.plot(lrXe, losses, marker='o')

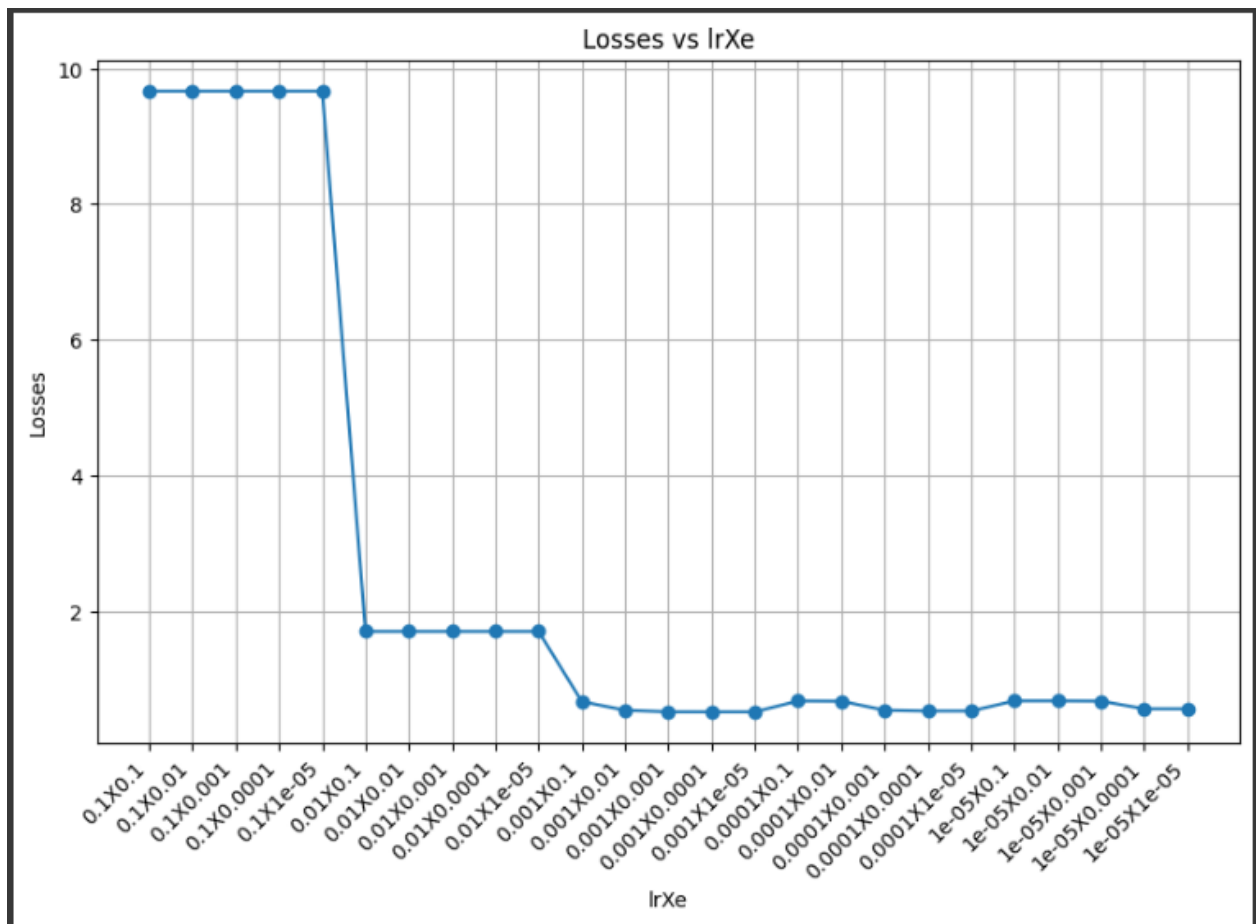
# Adding a title and labels to the plot
plt.title('Losses vs lrXe')
plt.xlabel('lrXe')
plt.ylabel('Losses')

# Rotate x-axis labels for better readability
plt.xticks(rotation=45, ha="right")

# Display grid lines on the plot
plt.grid(True)

# Show the plot
plt.show()

```



8 References

- [1] R. M. K. Williams, "Machine Learning Paradigms: A Comprehensive Overview," *Journal of Computational Intelligence*, pp. 32(2), 128-140., 2020.
- [2] L. Z. T. Johnson, "Linear Classifiers in Machine Learning: Emphasizing Logistic Regression," *Journal of Data Science and Applications*, Vols. 6(3), 231-242., 2019.
- [3] Y. Z. Y. L. H. H. Q. C. F. Y. Caiyun Tian, "High Albumin Level Is Associated With Regression of Glucose Metabolism Disorders Upon Resolution of Acute Liver Inflammation in Hepatitis B-Related Cirrhosis," *Frontiers in Cellular and Infection Microbiology*, 2022.
- [4] "What is Ascites?," [Online]. Available: <https://www.pennmedicine.org/for-patients-and-visitors/patient-information/conditions-treated-a-to-z/ascites#:~:text=Ascites%20results%20from%20high%20pressure,hepatitis%20C%20or%20B%20infection>. [Accessed 15 10 2023].
- [5] "Recursive Feature Elimination," [Online]. Available: [https://www.scikit-yb.org/en/latest/api/model_selection/rfe.html#:~:text=Recursive%20feature%20elimination%20\(RFE\)%20is,number%20of%20features%20is%20reached](https://www.scikit-yb.org/en/latest/api/model_selection/rfe.html#:~:text=Recursive%20feature%20elimination%20(RFE)%20is,number%20of%20features%20is%20reached). [Accessed 10 10 2023].
- [6] "Prothrombin time (PT)," [Online]. Available: <https://medlineplus.gov/ency/article/003652.htm>. [Accessed 15 10 2023].
- [7] "How to Tell the Difference Between Poisonous and Edible Mushrooms," [Online]. Available: <https://www.wildfooduk.com/articles/how-to-tell-the-difference-between-poisonous-and-edible-mushrooms/>. [Accessed 9 10 2023].
- [8] "Esophageal varices," [Online]. Available: <https://www.mayoclinic.org/diseases-conditions/esophageal-varices/symptoms-causes/syc-20351538>. [Accessed 15 10 2023].