ECSE 551 Group 16 Mini-Project 1

Jonathan Arsenault
260585289

Hung-Yang Chang
260899468

Anan Lu
260467054

Abstract

In this project, logistic regression is used to solve two separate binary classification problems, one involving bankruptcy data and the other involving hepatitis data. Simple models were first trained to establish benchmarks. Various experiments were then performed to improve on these results. These experiments can be classified into feature selection and optimization parameter selection. The results of these experiments are presented, and the best model for each task is identified.

1 Introduction

Logistic regression is a commonly used technique for performing classification. In logistic regression, the log-odds ratio of a given class is a linear combination of the features. This log-odds ratio is converted to a probability using the logistic function. In this report, logistic regression is used to perform two classification tasks. The first problem is to use patient information to determine the probability of a patient having hepatitis. The second problem is to use a set of unnamed numeric features to determine the probability of a bankruptcy occurring. To build the best possible models, the data sets are analyzed, feature engineering is performed, and the effect of the parameters of the optimization algorithm are investigated.

In order to build an effective machine learning model, it is important to have an understanding of the data set of interest. In Section 2, it is shown that certain numeric features are not normally distributed, meaning some type of normalizing transformation may be beneficial [5]. Furthermore, statistical techniques are used to determine whether the class label is independent of each feature [4]. In both data sets, these statistical techniques reveal that certain features are likely to be independent of the class label.

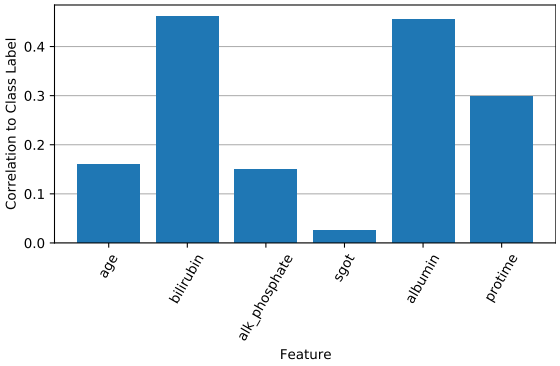
In Section 3 the feature engineering techniques discussed in Section 2 are tested to determine their efficacy. In this section, the effect of the tolerance, learning rate and initial weights on the performance of the logistic regression algorithm are also investigated. All algorithms are evaluated using 10-fold cross-validation. The metric of choice is the accuracy on the testing set. The number of iterations is used as a measure of convergence time. The report is brought to a close in Section 4, with a discussion of the key findings, along with possible avenues for future work.

2 Data Sets

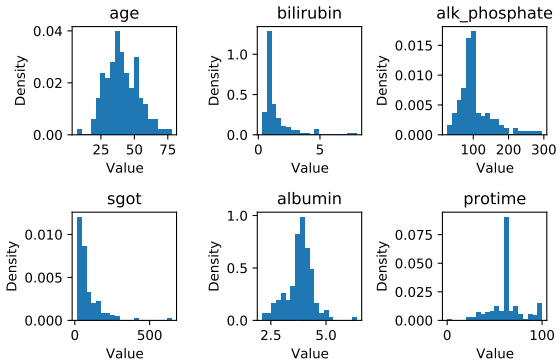
Before performing the logistic regression, the data sets are analyzed. A superficial exploration is performed to determine the distribution of the features and the classes. An attempt is also made to predict the relative importance of each feature in predicting the class of the sample. The Pearson correlation is used to measure the correlation between numeric features and the class label [4]. To measure the importance of categorical features, the Chi-squared test of independence is used [1].

2.1 Hepatitis

The hepatitis data set contains 142 samples. Each sample contains 19 features and a class label, with 1 being assigned to a patient with hepatitis and 0 being assigned to a patient without hepatitis. The data set is heavily biased towards patients with hepatitis, with 116 samples versus 26 samples of healthy patients. Of the 19 features, 6 are numeric features and 13 are categorical.



(a) Correlation of each numeric feature set to the class label.



(b) Distribution of each numeric feature.

Figure 1: Correlation and distribution of each numeric feature in the hepatitis data set to the class label.

The correlation of the numeric features to the class label is shown in Figure 1a. Most numeric features seem somewhat correlated to the class label. The only exception is the feature `sgot`, which has a correlation of only 0.02. It may therefore be reasonable to omit this feature from the logistic regression. This is attempted in Section 3.5.

The distributions of the numeric features are shown in Figure 1b. Ideally, all features would be normally distributed, with similar scales. However, the distributions of the `bilirubin` and `sgot` features are heavily skewed. Applying a transformation that would normalize these feature would possibly improve the performance of the model. A log transformation is one method of normalizing a skewed distribution [5]. Here, this is only possible as all samples have strictly positive values for these features. This possible improvement is further explored in Section 3.6.

To determine the importance of the categorical features on the class label, the Chi-squared test of independence is used. The results of the test are shown in Figure 2. With 95% certainty, it is possible to say that 5 features are independent from the class label. They are `sex`, `steroid`, `antivirals`, `liver_big`, and `liver_firm`. It is therefore possible that omitting these independent features from the data set will improve the performance of the model. This is attempted in Section 3.5.

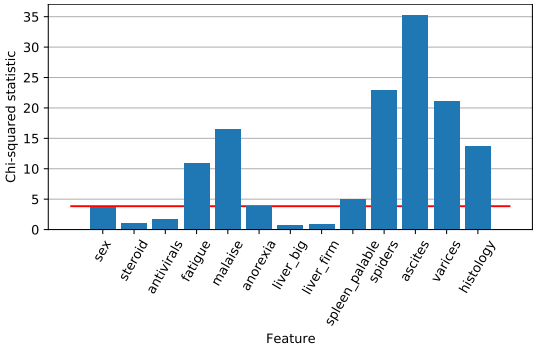


Figure 2: Chi-squared statistic for each categorical feature. The red line represents a 95% certainty bound

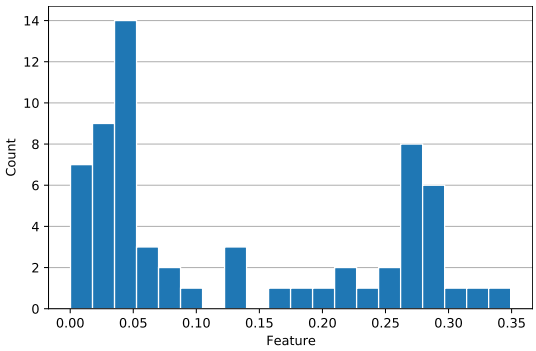


Figure 3: Distribution of correlation of the features in the bankruptcy data set

2.2 Bankruptcy

The bankruptcy data set contains 453 samples. Each sample contains 64 numeric features and a class label, where 1 and 0 correspond to a bankruptcy occurring or not occurring. Contrary to the hepatitis data set, the features do not have descriptive names. The data contains 250 instances of no bankruptcy and 203 bankruptcies.

As in the hepatitis data set, the correlation of each feature to the class label is of interest. Due to the amount of features, it is impractical to show the correlation of all the features individually. Figure 3 shows the distribution

	Hepatitis		Bankruptcy	
	accuracy (%)	iteration	accuracy (%)	iteration
w/ standardization	79.29	80.1	73.24	102
w/o standardization	84.02	1808	74.49	84.4

Table 1: Effect of standardization on results of 10-fold cross validation.

of the correlation of each feature to the class label. No feature has a correlation above 0.35. Of the 64 features only 35 have a correlation above 0.05. Removing the features with especially low correlations may improve the performance of the logistic regression, as is shown in Section 3.5.

Once again, due to the large amount of features, displaying their distributions is impractical. Investigating the distribution of each feature independently reveals some interesting information. First, most of the features have similar scales. The features are, on average, close to zero mean, with standard deviations varying between 0 and approximately 2. This indicates that a z-score standardization may have minimal effect on the results. There are however, skewed distributions for which a log transformation may improve the results, with necessary treatment of negative values. Contrary to the hepatitis data set, the bankruptcy data set has several samples with outliers. This may further complicate the machine learning problem, potentially leading to greater variance in the model.

3 Results

The logistic regression model is constructed with a `fit` function to find weights from a training set by minimizing cross-entropy loss, a `predict` function to predict results with a validation set, and an `accu_eval` function to compare the predicted results with actual data. The code is presented in the Appendix. The feature engineering techniques discussed in Section 2 are tested. The effect of weight initialization, learning rate and stopping criteria are also investigated. A standard 10-fold cross-validation is used to assess the performance of the logistic regression in these various experiments. The benchmark used for comparison utilizes the features as given in the raw data, with the weights initialized to zero, the tolerance set to $\epsilon = 10^{-2}$ and the learning rate set to

$$\alpha_k = \frac{1}{1+k}. \tag{1}$$

Acknowledging that there are various methods of evaluating the performance of the model, the chosen metric is the accuracy on the training set. The number of iterations to convergence is also recorded for each experiment to get a measure of the convergence time of the model training.

3.1 Standardization

Standardization, or normalization, is a standard feature processing technique used to improve the performance of machine learning techniques. This section investigates how a z-score standardization of the features may improve the classifier's performance. According to [3], standardization is particularly useful when data is of varying scale or the model has an underlying Gaussian assumption. Despite not requiring Gaussian data, standardization may still be useful for logistic regression. The effect of z-score standardization on both classifiers is summarized in Table 1. On both data sets, the net impact is a decrease in accuracy. As mentioned in Section 2.1, the decrease in accuracy on the hepatitis classifier may be due to the skewed nature of some of the distributions. However, the scaling provided by the standardization clearly leads to an improvement in convergence time, with an order of magnitude decrease in the number of iterations required to converge. The results are much less stark in the case of the bankruptcy classifier, as most of the features are already of similar scales. In this case, the decrease may be attributable to the presence of outliers. If outliers are present in the training set, the mean and standard deviation used to standardize the testing data may lead to erroneous results.

3.2 Initial Weight

It is cited commonly that the initial weights for linear logistic regression could be set to zero [2]. Three other options are tested, namely drawing the weights from a uniform distribution defined between 0 and 1, denoted $\mathcal{U}(0,1)$, drawing the weights from $\mathcal{U}(0,100)$ and drawing the weights from a standard normal, $\mathcal{N}(0,1)$. The results are summarized in Table 2. It is shown that random initial weights decreased the accuracy for the bankruptcy classifier, and increased it slightly for the hepatitis classifier. For best results, it is therefore recommended to draw the initial weights from $\mathcal{U}(0,1)$ for the hepatitis classifier, and using $\mathbf{w}_0 = \mathbf{0}$ for the bankruptcy classifier.

	Hepatitis		Bankruptcy	
	accuracy (%)	iteration	accuracy (%)	iteration
$\mathbf{w}_0 = \mathbf{0}$	84.02	1808	73.69	84
$\mathbf{w}_0 \sim \mathcal{U}(0,1)$	85.54	1804	73.01	103
$\mathbf{w}_0 \sim \mathcal{U}(0,100)$	85.14	2073	70.65	232
$\mathbf{w}_0 \sim \mathcal{N}(0,1)$	84.73	1788	71.93	85

Table 2: Effect of initial weight selection on results of 10-fold cross validation.

	Hepatitis		Bankruptcy	
	accuracy (%)	iteration	accuracy (%)	iteration
$\epsilon = 10^{-2}$	84.02	1808	73.69	84
$\epsilon = 10^{-6}$	84.73	14408	75.03	3709
$\epsilon = 10^{-9}$	84.73	failed	76.33	failed

Table 3: Effect of changing stopping criteria on results of 10-Fold cross validation.

3.3 Learning rate

At first, a large number of learning rates are tested, including setting it to a constant 1/2, 1/4, or 1/8, and setting it to 1/2 before 250 iterations while reducing it to 1/4 afterward. In all these cases, the model failed to converge in a reasonable amount of iterations. For the purpose of demonstration, experiments with the learning rate set at 0.5 and 0.1 are included in the code, both are able to present an accuracy close to the benchmark but fail to converge at 15000 iterations. In the end, Equation (1) is implemented. In this way, a larger gradient can help the model converge faster in the beginning, and as the iterations increase, smaller local search can be realized to achieve finer change. However, this does mean that the number of iterations grows exponentially with more strict stopping conditions.

3.4 Stopping Condition

For this project, the weights are trained until $||\mathbf{w}_k - \mathbf{w}_{k-1}|| < \epsilon$, where ϵ is the tolerance. In this section, different values for ϵ are tested. The results are summarized in Table 3. For the hepatitis classifier, the improvement in accuracy is marginal, accompanied by significant increase in the number of iterations. For the bankruptcy classifier, setting $\epsilon = 10^{-6}$ could increase the accuracy by over 1%, at the cost of just a few thousands iterations. Therefore, for better performance, stopping condition could be set at $\epsilon = 10^{-2}$ for the hepatitis classifier, and $\epsilon = 10^{-6}$ for the bankruptcy classifier.

3.5 Removing Independent Features

In Section 2, it is demonstrated that certain features are, to some extent, statistically independent of the class label using the Pearson correlation coefficient and the Chi-squared test of independence. These features are referred to as independent features. In the hepatitis data set, the features most likely to be independent of the class label were `sgot`, `sex`, `steroid`, `antivirals`, `liver_big`, and `liver_firm`. In the bankruptcy data set, features with a Pearson correlation coefficient to the class label less than 0.15 were determined to be worth further investigation. There are 17 features that satisfy this criteria.

Experiments are run where each of the possibly independent features are removed one at a time, and then all at once. The features that led to an increase in accuracy when removed are identified and recorded. Removing only these "accuracy-decreasing" features leads to a 2.14% improvement in accuracy compared to the benchmark hepatitis classifier and a 2.67% improvement as compared to the benchmark bankruptcy classifier. The results are shown in Table 4.

3.6 Log Transformation

As mentioned in Section 2, the distributions of certain features are skewed. Applying standardization to skewed distributions may partly explain the results of Section 3.1. To remedy this, one option is to transform these features to Gaussian-like ones by taking the log of the values [5]. Therefore, an experiment was constructed to observe if a log transformation of selected features could improve the accuracy of the logistic regression model.

	Hepatitis		Bankruptcy	
	accuracy (%)	iteration	accuracy (%)	iteration
All features	84.02	1808	74.49	84.4
w/o independent features	83.93	1200.6	72.90	102
w/o "accuracy-decreasing" features	86.16	1643	77.15	84

Table 4: Effect of removing independent features on results of 10-fold cross validation.

Determining which features had skewed distributions was based on a simple test to see if the median of a feature is far away from the mean, a common measure of skewness. This test was first applied on the numeric features of the hepatitis data set to show that it matches the observation from Section 2.1. This criterion is then applied to both data sets. For the bankruptcy data set, a threshold value is added to treat negative numbers.

As a result, for the hepatitis data set, only adding log transformed features without standardization improved the training accuracy by 0.98%. On the other hand, for bankruptcy data, almost all cases with log transform result in increased accuracy, with the maximum resulting from replacing the features with its log without standardization that improved accuracy by 1.78%. Contrary to the expectations, log transform does not improve the result of standardization. One explanation is that log of small values creates discrepancies and incompatibility with features that remain unchanged. Nevertheless, cases with improved accuracy are included in the final best result presented in Section 4.

4 Discussion and Conclusions

The goal of this project was to build the best possible logistic regression model through creative feature engineering and careful selection of the parameters of the gradient descent algorithm. This proved to be a complex task, with a quasi-infinite set of possible experiments to run. The focus was put on creating a systematic way of obtaining the best possible classification models, where "best" was defined as the highest mean testing accuracy obtained from 10-fold cross-validation.

For the hepatitis data set, the best possible results were obtained by simply using the benchmark classifier while removing the features deemed to be independent of the class label. This leads to an accuracy of 86.16%, compared to the benchmark value of 84.02%. The best results were obtained on the bankruptcy classifier by removing the independent features, appending the log transformation of the skewed variables and setting the tolerance to 10^{-6}. This led to an increase of 3.75 % in accuracy, from 73.69% to 77.44%. This came at the cost of an increase in the number of iterations to convergence, however, from 84.1 to 3919.3. Due to the inherent randomness involved in these experiments, it is likely that the proposed modifications improve on the benchmark classifiers, but more rigorous experimentation would be required to provide a definitive answer.

4.1 Future Investigation

Future work on these classification algorithms can be in the feature engineering and in the optimization algorithm. Due to the small sizes of the data sets, the results are particularly sensitive to the way the data is split in the 10-fold cross-validation. More experiments would be required to eliminate the effects of randomness. These small sample sizes also amplify the effect of outliers. A scheme to account for these outliers could also be implemented. These models are trained using a standard gradient descent algorithm. However, there exist many alternative algorithms that could be used to improve the optimization. Lastly, there are several other metrics that could be used to evaluate the performance of these models. Depending on the potential uses of these classification algorithms, alternative evaluation metrics may be more appropriate than accuracy in determining how well these models perform.

5 Contribution Statement

1. Jonathan Arsenault: Data pre-processing, implementing of algorithm, write-up contribution.
2. Hung-Yang Chang: Data pre-processing, implementing of algorithm, write-up contribution.
3. Anan Lu: Data pre-processing, implementing of algorithm, write-up contribution.

References

[1] Agresti, A. (2013). Categorical Data Analysis (3rd ed.). John Wiley and Sons, Inc.

[2] Lee, Elisa T. "A computer program for linear logistic regression analysis." Computer programs in biomedicine 4.2 (1974): 80-92.

[3] A. Gelman and J. Hill. "Data analysis using regression and multilevel/hierarchical models" vol. Analytical methods for social research. New York: Cambridge University Press (2007)

[4] Ozdemir, S., & Susarla, D. (2018). Feature Engineering Made Easy. Packt Publishing.

[5] Changyong, F. E. N. G., et al. "Log-transformation and its implications for data analysis." Shanghai archives of psychiatry 26.2 (2014): 105.

6 Appendix

Code1:hepatitis_feature_analysis.py

```
# -*- coding: utf-8 -*-
"""hepatitis_feature_analysis.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1wMzHN4FVGqzykvITvI_Jolj8Kbi25Mof

# Perform initial data analysis on hepatitis data set
"""

# Import relevant modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load hepatitis data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/hepatitis.csv"
hep_data = pd.read_csv(url)

# Display some of the data
print(hep_data.head())

# Print size of data
hep_data.shape

# Plot distribution of class labels
label_counts = hep_data["ClassLabel"].value_counts()
print(label_counts)
unique_labels = hep_data["ClassLabel"].unique()

fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(unique_labels, label_counts, zorder=2)
ax.set_xticks([0, 1])
ax.set_xticklabels(unique_labels)
ax.set_xlabel("Class Label")
ax.set_ylabel("Count")

# Mean and standard deviation of numeric features

# List of column names that are numeric
num_columns = [
    "age",
    "bilirubin",
    "alk_phosphate",
    "sgot",
    "albumin",
    "protime",
]

# Indices of features
index_cat_columns = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18])
index_numeric_columns = np.array([0, 13, 14, 15, 16, 17])

# Compute mean of each numerical feature
means = hep_data.loc[:, num_columns].mean()

# Compute standard deviation of each feature
std = hep_data.loc[:, num_columns].std()
```

```
# Plot mean and standard deviation of each numerical feature
features = np.arange(1, len(num_columns) + 1)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, means, zorder=2)
ax.set_xticks(np.arange(1, len(num_columns) + 1))
ax.set_xticklabels(num_columns, rotation=60)
ax.set_xlabel("Feature")
ax.set_ylabel("Mean")

fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, std, zorder=2)
ax.set_xticks(np.arange(1, len(num_columns) + 1))
ax.set_xticklabels(num_columns, rotation=60)
ax.set_xlabel("Feature")
ax.set_ylabel("Standard Deviation")

# Compute correlation of numeric features
save_fig = False

# Compute correlation only for numerical features
correlation = np.abs(hep_data.loc[:, num_columns + ["ClassLabel"]].corr())
correlation_arr = correlation.to_numpy()
for i in range(correlation_arr.shape[0]):
    for j in range(i):
        correlation_arr[i, j] = np.nan

# Plot as image
fig, ax = plt.subplots()
im = ax.imshow(correlation_arr, vmin=0, vmax=1.0)
plt.colorbar(im)
ax.set_xticks(np.arange(len(num_columns) + 1))
ax.set_yticks(np.arange(len(num_columns) + 1))
ax.set_xticklabels(num_columns + ["ClassLabel"], rotation=60)
ax.set_yticklabels(num_columns + ["ClassLabel"])
ax.set_xlabel("Feature")
ax.set_ylabel("Feature")

# Plot bar graph of correlation to class label
features = np.arange(len(num_columns))
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, correlation_arr[:-1, -1], zorder=2)
ax.set_xticks(np.arange(0, len(num_columns)))
ax.set_xticklabels(num_columns, rotation=60)
ax.set_xlabel("Feature")
ax.set_ylabel("Correlation to Class Label")
if save_fig:
    plt.tight_layout()
    plt.savefig("hep_correlation_bar.pdf", bbox_inches="tight", pad_inches=0)

print(correlation_arr[:-1, -1])

save_fig = False

fig_all, ax = plt.subplots(2, 3)
col = 0
for x, axis in enumerate(ax.ravel()):
    axis.hist(hep_data.loc[:, num_columns[col]], bins=20, density=True)
    axis.set_title(num_columns[col])
    axis.set_xlabel("Value")
    axis.set_ylabel("Density")
    col += 1
```

```
fig_all.tight_layout()
if save_fig:
    plt.savefig("hep_data_distributions.pdf", bbox_inches="tight", pad_inches=0)

features_to_log = ["bilirubin", "alk_phosphate", "sgot"]
fig, ax = plt.subplots(1, 3)
col = 0
for x, axis in enumerate(ax.ravel()):
    axis.hist(np.log(hep_data.loc[:, features_to_log[col]]), bins=20, density=True)
    axis.set_title(features_to_log[col])
    axis.set_xlabel("Value")
    axis.set_ylabel("Density")
    col += 1
fig.tight_layout()

cat_columns = [
    "sex",
    "steroid",
    "antivirals",
    "fatigue",
    "malaise",
    "anorexia",
    "liver_big",
    "liver_firm",
    "spleen_palable",
    "spiders",
    "ascites",
    "varices",
    "histology",
]

# Compute the value of each categorical feature when label is 1
label_one = hep_data.loc[hep_data["ClassLabel"] == 1, cat_columns]
count_two = label_one[label_one == 2].count()
count_one = label_one[label_one == 1].count()

features = np.arange(1, len(cat_columns) + 1)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, count_one, zorder=2)
ax.bar(features, count_two, bottom=count_one, zorder=2)
ax.set_xticks(np.arange(1, len(cat_columns) + 1))
ax.set_xticklabels(cat_columns, rotation=60)
ax.set_xlabel("Feature")
ax.set_ylabel("Count")
ax.set_title("Distribution of categorical features given a class label of 1.")

# Compute the value of each categorical feature when label is 1
label_two = hep_data.loc[hep_data["ClassLabel"] == 0, cat_columns]

count_two = label_two[label_two == 2].count()
count_one = label_two[label_two == 1].count()

features = np.arange(1, len(cat_columns) + 1)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, count_one, zorder=2)
ax.bar(features, count_two, bottom=count_one, zorder=2)
ax.set_xticks(np.arange(1, len(cat_columns) + 1))
ax.set_xticklabels(cat_columns, rotation=60)
ax.set_xlabel("Feature")
ax.set_ylabel("Count")
ax.set_title("Distribution of categorical features given a class label of 0.")

# Chi-squared test of independence
```

```
# https://www.youtube.com/watch?v=L1QPBGoDmT0
##            class label
##              0   | 1       total
##  feature 1  01   | 11    feature 1
##  feature 2  02   | 12    feature 2
##           class0  class1

from scipy import stats


def Chi_squared_test(hep_data, cat_columns):
    """Perform a chi-squared test of independance on hepatitis data set"""

    # Separate data by class label
    label_one = hep_data.loc[hep_data["ClassLabel"] == 1, cat_columns]
    label_zero = hep_data.loc[hep_data["ClassLabel"] == 0, cat_columns]

    # Loop over all categorical features
    independent_list = []
    dependent_list = []
    chi_squared_list = []
    for i in cat_columns:

        feature = i
        num_samples = hep_data.shape[0]

        # prob of label = 0 or = 1
        probability_label_0 = label_zero.shape[0] / num_samples
        probability_label_1 = label_one.shape[0] / num_samples

        # prob of feature =1 or =2 (for categorical data)
        probability_feature_1 = hep_data[feature].value_counts()[1] / num_samples
        probability_feature_2 = hep_data[feature].value_counts()[2] / num_samples

        # number of label = 0 when feature =1 or =2
        if i == "sex":
            # Exception for sex where no 02 values exist
            observed_01 = label_zero[feature].value_counts()[1]
            observed_02 = 0
        else:
            observed_01 = label_zero[feature].value_counts()[1]
            observed_02 = label_zero[feature].value_counts()[2]

        # number of label = 1 when feature = 1 or = 2
        observed_11 = label_one[feature].value_counts()[1]
        observed_12 = label_one[feature].value_counts()[2]

        # If it's independent, expected number of label = 0 when feature = 1 or = 2
        expected_01 = probability_label_0 * probability_feature_1 * num_samples
        expected_02 = probability_label_0 * probability_feature_2 * num_samples

        # If it's independent, expected number of label = 1 when feature = 1 or = 2
        expected_11 = probability_label_1 * probability_feature_1 * num_samples
        expected_12 = probability_label_1 * probability_feature_2 * num_samples

        # Compute difference between observed and expected values
        diff_01 = observed_01 - expected_01
        diff_02 = observed_02 - expected_02
        diff_11 = observed_11 - expected_11
        diff_12 = observed_12 - expected_12

        # Compute Chi-squared test variable
        squared_diff_norm_01 = (diff_01) ** 2 / expected_01
        squared_diff_norm_02 = (diff_02) ** 2 / expected_02
```

```
        squared_diff_norm_11 = (diff_11) ** 2 / expected_11
        squared_diff_norm_12 = (diff_12) ** 2 / expected_12
        chi_squared = (
            squared_diff_norm_12
            + squared_diff_norm_11
            + squared_diff_norm_02
            + squared_diff_norm_01
        )

        # df=(r-1)(c-1)
        p_value = stats.chi2.ppf(0.95, df=1)

        chi_squared_list.append(chi_squared)

        # Add feature to appropriate list
        if chi_squared < p_value:
            independent_list.append(feature)
        else:
            dependent_list.append(feature)

    return independent_list, dependent_list, chi_squared_list


# Run chi-squared test
independent_list, dependent_list, chi_squared_list = Chi_squared_test(
    hep_data, cat_columns
)
print("independent:", independent_list)
print("dependent:", dependent_list)

p_value = stats.chi2.ppf(0.95, df=1)
print(p_value)
save_fig = True

# Plot chi-squared statistic
features = np.arange(0, len(cat_columns))
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, chi_squared_list, zorder=2)
ax.set_xticks(np.arange(0, len(cat_columns)))
ax.set_xticklabels(cat_columns, rotation=60)
ax.hlines(p_value, *ax.get_xlim(), color="r")
ax.set_xlabel("Feature")
ax.set_ylabel("Chi-squared statistic")
if save_fig:
    plt.tight_layout()
    plt.savefig("hep_chi_squared.pdf", bbox_inches="tight", pad_inches=0)
```

Code2:bankruptcy_feature_analysis.py

```
# -*- coding: utf-8 -*-
"""bankruptcy_feature_analysis.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1u4tJZJ-rWS6FW7Psh_sQY6eMp2mZXLO5

# Perform initial data analysis on bankruptcy data set
"""

# Import relevant modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load bankruptcy data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/bankrupcy.csv"
bank_data = pd.read_csv(url)

# Display some of the data
print(bank_data.head())

# Print size of data
bank_data.shape

# Plot distribution of class labels

label_counts = bank_data["ClassLabel"].value_counts()
print(label_counts)
unique_labels = bank_data["ClassLabel"].unique()

fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(unique_labels, label_counts, zorder=2)
ax.set_xticks([0, 1])
ax.set_xticklabels(unique_labels)
ax.set_xlabel("Class Label")
ax.set_ylabel("Count")

# Print summary statistics of features
bank_data.iloc[:, :-1].describe()

# Compute mean of each feature
means = bank_data.iloc[:, :-1].mean()

# Compute standard deviation of each feature
std = bank_data.iloc[:, :-1].std()

# Plot mean and standard deviation of each attribute
attributes = np.arange(1, 65)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(attributes, means, zorder=2)
ax.set_xlabel("Attribute #")
ax.set_ylabel("Mean")

attributes = np.arange(1, 65)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(attributes, std, zorder=2)
ax.set_xlabel("Attribute #")
ax.set_ylabel("Standard Deviation")
```

```
# Compute correlation of features only
correlation = bank_data.corr()
correlation_arr = np.abs(correlation.to_numpy())

# upper right and lower left represent exactly the same thing,
# so fill nan to lower left
for i in range(correlation_arr.shape[0]):
    for j in range(i):
        correlation_arr[i, j] = np.nan

# Plot as image
fig, ax = plt.subplots(figsize=(8, 8))
im = ax.imshow(correlation_arr, vmin=0.0, vmax=1.0)
plt.colorbar(im)
ax.set_xlabel("Attribute")
ax.set_ylabel("Attribute")

# correalation btw Class and all attribute
print(correlation_arr[:, -1])


save_fig = True
# Plot bar graph of correlation to class label
features = np.arange(len(bank_data.columns[:-1]))
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(features, correlation_arr[:-1, -1], zorder=2)
ax.set_xlabel("Feature")
ax.set_ylabel("Correlation to Class Label")
if save_fig:
    plt.tight_layout()
    plt.savefig("bank_correlation_bar.pdf", bbox_inches="tight", pad_inches=0)

save_fig = True
# Plot histogram of correlation to class label
features = np.arange(len(bank_data.columns[:-1]))
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.hist(correlation_arr[:-1, -1], bins=20, zorder=2, edgecolor="w")
ax.set_xlabel("Feature")
ax.set_ylabel("Count")
if save_fig:
    plt.tight_layout()
    plt.savefig("bank_correlation_hist.pdf", bbox_inches="tight", pad_inches=0)

print(np.count_nonzero(bank_data.columns[correlation_arr[:, -1] < 5e-2]))

# Compute mean of each attribute for each class
class_label_1 = bank_data.loc[
    bank_data.iloc[:, -1] == 1,
]
class_label_0 = bank_data.loc[
    bank_data.iloc[:, -1] == 0,
]

mean_1 = class_label_1.iloc[:, :-1].mean()
mean_0 = class_label_0.iloc[:, :-1].mean()

difference = mean_1 - mean_0

attributes = np.arange(1, 65)
fig, ax = plt.subplots()
ax.grid(zorder=1, axis="y")
ax.bar(attributes, difference, zorder=2)
```

```
ax.set_xlabel("Attribute #")
ax.set_ylabel("difference")
```

Code3:Miniproject1.py

```
# -*- coding: utf-8 -*-
"""Miniproject1.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1qTlbmhtUVG5661LoSdJAizRf6rw2OuRH

# Logistic Regression Code
"""

# Import relevant modules

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time

"""# Read Data Sets (Bankrupcy and Hepatitis)"""

# Load bankruptcy data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/bankrupcy.csv"
bank_data = pd.read_csv(url)

# Display some of the data
print(bank_data.head())

# Print size of data
print(bank_data.shape)

# Load hepatitis data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/hepatitis.csv"
hep_data = pd.read_csv(url)

# Display some of the data
print(hep_data.head())

# Print size of data
print(hep_data.shape)

# Indices of numerical and categorical features (identified from data analysis)
index_cat_columns_raw = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18])
index_num_columns_raw = np.array([0, 13, 14, 15, 16, 17])

"""# Define utility functions for logistic regression
- shuffle_data
- splitdata
- standardization
- sigmoid
- log_transform
"""


def shuffle_data(X, y, random_seed=None):
    """Shuffle the data to randomize tests"""

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

    # Copy data
    X_original_copy = X.copy()
    y_original_copy = y.copy()
```

```
    # Concatenate and shuffle
    full_array = np.concatenate((X_original_copy, y_original_copy), axis=1)
    np.random.shuffle(full_array)

    # Split into features and labels
    X_shuffle = full_array[:, :-1]
    y_shuffle = full_array[:, [-1]]

    return X_shuffle, y_shuffle


def splitdata(X, y, perc_training, random_flag=True):
    """Split data into training and testing set (by rows(observations)) by taking a constant
        set"""

    num_rows = X.shape[0]
    num_rows_train = int(num_rows * perc_training)
    num_rows_test = num_rows - num_rows_train

    X_train = X[:num_rows_train, :]
    X_test = X[num_rows_train:, :]
    y_train = y[:num_rows_train]
    y_test = y[num_rows_train:]

    return X_train, y_train, X_test, y_test


def standardization(data, training_data):
    """Standardize each column of input data"""

    data_standardized = (data - training_data.mean(axis=0)) / training_data.std(axis=0)

    return data_standardized


def sigmoid(x):
    """Apply logistic sigmoid function to input"""

    return 1 / (1 + np.exp(-x))


def log_transform(X_data, feature, replace=True, bank=False):
    """Perform log transform if feature may not be normally distributed"""

    if replace:
        # Features to be replaced with their log transform

        for i in feature:
            # Check if mean is close to median
            variance = np.absolute(
                np.mean(X_data[:, i]) - np.median(X_data[:, i])
            ) / np.mean(X_data[:, i])
            if variance > 0.1:
                if bank:
                    # For bankruptcy data, ensure data is positive
                    X_data[:, i] = (
                        X_data[:, i] + np.absolute(np.min(X_data[:, i])) + 0.5
                    )
                    X_data[:, i] = np.log(X_data[:, i])
                else:
                    X_data[:, i] = np.log(X_data[:, i])

    else:
        # Append log transform to exisitng features
```

```
        for i in feature:
            # Check if mean is close to median
            variance = np.absolute(
                np.mean(X_data[:, i]) - np.median(X_data[:, i])
            ) / np.mean(X_data[:, i])
            if variance > 0.1:
                if bank:
                    # For bankruptcy data, ensure data is positive
                    X_data[:, i] = (
                        X_data[:, i] + np.absolute(np.min(X_data[:, i])) + 0.5
                    )
                    X_data = np.insert(
                        X_data, X_data.shape[1] - 1, np.log(X_data[:, i]), axis=1
                    )
                else:
                    X_data = np.insert(
                        X_data, X_data.shape[1] - 1, np.log(X_data[:, i]), axis=1
                    )

    return X_data


"""# Logistic Classifier Class"""


class LogisticClassifier:
    """Class defining a logistic classifier"""

    def __init__(self, weights):
        """Constructor"""

        self._w = weights

    def fit(
        self,
        x_train,
        y_train,
        max_iters,
        tolerance=1e-2,
        print_results=True,
        store_w_iteration=False,
        learning_rate="dependent",
    ):
        """Fit a logistic regression model to data"""

        # Lists to store (1) weights at each iteration or (2) just last weights
        weight_store_fit = []

        iteration = 1
        delta_weights = 1e6

        while (delta_weights > tolerance) & (iteration < max_iters):

            # Set learning rate for this iteration
            if learning_rate == "dependent":
                alpha = 1 / (1 + iteration)
            elif learning_rate == "small":
                alpha = 0.1
            else:
                alpha = 0.5

            # Store current weights before updating
            weights_previous = self._w

            # Compute gradient of cross-entropy loss
```

```
            gradient = np.sum(
                x_train * (y_train - sigmoid(np.dot(x_train, weights_previous))), axis=0
            ).reshape(-1, 1)

            # Update weights
            self._w = weights_previous + alpha * gradient

            # Compute change in weights
            delta_weights = np.linalg.norm(self._w - weights_previous) ** 2

            if store_w_iteration == True:
                # Store weights at each iteration
                weight_store_fit.append(self._w.flatten()) # TODOO

            iteration += 1

        if store_w_iteration == False:
            # Store final weights
            weight_store_fit.append(self._w.flatten())
            # print ("Final weights:", weight_store_fit)

        # Compute training accuracy
        y_pred_train = self.predict(x_train)

        accuracy = self.accu_eval(y_train, y_pred_train)

        if print_results:
            if iteration == max_iters:
                print(f"Failed to converge in {max_iters} iterations")
            else:
                print(f"Model converged in {max_iters} iterations")
            print(f"Training accuracy: {100*accuracy:.2f}")

        return accuracy, iteration, weight_store_fit

    def predict(self, X):
        """Predict the class labels of a given set of samples"""

        # Decision boundary
        decision_boundary = 0.5

        # Obtain probability of each sample
        y_pred_prob = sigmoid(np.dot(X, self._w))

        # Assign class labels based on decision boundary
        y_pred = np.where(y_pred_prob < decision_boundary, 0, 1)

        return y_pred

    def accu_eval(self, y_true, y_pred):
        """Compute accuracy of model"""

        accuracy = np.count_nonzero(y_true == y_pred) / len(y_true)

        return accuracy

    def cross_entropy_loss(self, x_test, y_test):
        """Compute cross entropy loss"""

        y_pred_prob = sigmoid(np.dot(x_test, self._w))
        y_pred_prob_m1 = 1 - y_pred_prob

        # Replace small values in both with 1e-5 to avoid NAN (log0)
        y_pred_prob = np.where(y_pred_prob < 1e-5, 1e-5, y_pred_prob)
        y_pred_prob_m1 = np.where(y_pred_prob_m1 < 1e-5, 1e-5, y_pred_prob_m1)
```

```
        loss_0 = y_test * np.log(y_pred_prob)
        loss_1 = (1 - y_test) * np.log(y_pred_prob_m1)

        loss = -np.sum(loss_0 + loss_1)

        return loss


"""# K-fold Cross Validation"""


def kfold_cross_validation(
    model,
    X_train,
    y_train,
    k=10,
    tolerance=1e-2,
    standardize_idx=None,
    printresult=True,
    learning_rate="dependent",
):
    """Perform k-fold cross validation"""

    # Compute the amount of samples in each fold

    fold_size = int(len(X_train) / k)
    if printresult:
        print(
            "Now doing k-fold cross validation, fold size= {}, x train shape = {}, y train
                shape = {}".format(
                fold_size, X_train.shape, y_train.shape
            )
        )

    accuracy_train_store = []
    accuracy_test_store = []
    iteration_store = []
    weight_store_fold = []
    fold_CE_store = []

    # Store initial weights to use the same for each fold
    initial_weights = model._w

    for fold_number in range(k):

        # Reset weights
        model._w = initial_weights

        # Split data
        index_start = fold_size * fold_number
        index_end = fold_size * (fold_number + 1)

        if fold_number == (k - 1):
            # For final fold
            X_train_fold = X_train[:index_start, :]
            y_train_fold = y_train[:index_start, :]
            X_validation_fold = X_train[index_start:, :]
            y_validation_fold = y_train[index_start:, :]

        else:
            # For all other folds fold
            X_train_fold = np.concatenate(
                (X_train[:index_start, :], X_train[index_end:, :]), axis=0
            )
```

```
            y_train_fold = np.concatenate(
                (y_train[:index_start, :], y_train[index_end:, :]), axis=0
            )
            X_validation_fold = X_train[index_start:index_end, :]
            y_validation_fold = y_train[index_start:index_end, :]

        # Standardize if required
        X_train_fold_original = X_train_fold.copy()
        if standardize_idx == "all":
            # Standardize all columns
            X_train_fold[:, 1:] = standardization(
                X_train_fold[:, 1:], X_train_fold_original[:, 1:]
            )
            X_validation_fold[:, 1:] = standardization(
                X_validation_fold[:, 1:], X_train_fold_original[:, 1:]
            )
        elif standardize_idx is not None:
            # Standardize subset of columns
            X_train_fold[:, standardize_idx] = standardization(
                X_train_fold[:, standardize_idx],
                X_train_fold_original[:, standardize_idx],
            )
            X_validation_fold[:, standardize_idx] = standardization(
                X_validation_fold[:, standardize_idx],
                X_train_fold_original[:, standardize_idx],
            )
        else:
            # Do not standardize
            pass

        # Fit the model
        t_start = time.time()
        accuracy_train, iteration, weight_store = model.fit(
            X_train_fold,
            y_train_fold,
            max_iters=15000,
            tolerance=tolerance,
            print_results=False,
            learning_rate=learning_rate,
        )
        t_end = time.time()

        # Compute accuracy and cross-entropy loss
        y_pred_test = model.predict(X_validation_fold)
        accuracy_test = model.accu_eval(y_validation_fold, y_pred_test)
        cross_entropy = model.cross_entropy_loss(X_validation_fold, y_validation_fold)

        # Store values
        accuracy_train_store.append(accuracy_train)
        accuracy_test_store.append(accuracy_test)
        iteration_store.append(iteration)
        fold_CE_store.append(cross_entropy)
        weight_store_fold.append(weight_store)

        if printresult:
            print(f"### Fold number {fold_number+1} ###")
            print(f"Execution time: {t_end-t_start:.3f}s")
            print(f"Training Accuracy: {100*accuracy_train:.2f} %")
            print(f"Testing Accuracy: {100*accuracy_test:.2f} %")
            print(f"Cross-entropy loss CE: {cross_entropy}")

    cross_accuracy = np.mean(accuracy_test_store)
    if printresult:
        print("############################")
        print(f"Mean testing accuracy is {cross_accuracy*100:.2f} %")
```

```
    return accuracy_train_store, accuracy_test_store, iteration_store, weight_store_fold


"""# Define a function to easily run experiments"""


def test_classifier(
    model,
    X,
    y,
    num_folds=None,
    num_loops=1,
    standardize_idx=None,
    tolerance=1e-2,
    max_iters=1000,
    random_seed=None,
    print_results=False,
    learning_rate="dependent",
):
    """Test the logistic regression"""

    if num_folds is None:
        # Do not perform k-fold cv

        # Shuffle data
        if random_seed is not None:
            X, y = shuffle_data(X, y, random_seed)

        X_train, y_train, X_test, y_test = splitdata(X, y, 0.8)

        X_train_original = X_train.copy()
        if standardize_idx == "all":
            # Standardize all columns
            X_train = standardization(X_train, X_train_original)
            X_test = standardization(X_test, X_train_original)
        elif standardize_idx is not None:
            # Standardize subset of columns
            X_train[:, standardize_idx] = standardization(
                X_train_original[:, standardize_idx],
                X_train_original[:, standardize_idx],
            )
            X_test[:, standardize_idx] = standardization(
                X_test[:, standardize_idx], X_train_original[:, standardize_idx]
            )
        else:
            # Do not standardize
            pass

        accuracy_train, iterations, weight_store = model.fit(
            X_train, y_train, max_iters, tolerance, print_results
        )
        accuracy_test = model.accu_eval(y_test, model.predict(X_test))
    else:
        # Perform k-fold cross-validation
        iterations = []
        accuracy_train = []
        accuracy_test = []
        weight_store = []
        for i in range(num_loops):

            if random_seed is not None:
                X, y = shuffle_data(X, y, random_seed * (i + 1))

            (
```

```
                accuracy_train_iter,
                accuracy_test_iter,
                iterations_iter,
                weight_store_iter,
            ) = kfold_cross_validation(
                model,
                X,
                y,
                num_folds,
                tolerance,
                standardize_idx=standardize_idx,
                printresult=print_results,
                learning_rate=learning_rate,
            )

            accuracy_train.append(accuracy_train_iter)
            accuracy_test.append(accuracy_test_iter)
            iterations.append(iterations_iter)
            weight_store.append(weight_store_iter)

    return accuracy_train, accuracy_test, iterations, weight_store


"""# Set baseline accuracy and iterations for hepatitis data
- Initial weights: zeros
- Learning rate = 1/1+k
- Stopping criteria: epsilon = 1e-2
- No standardization
"""

# RIGHT NOW, BASELINE IS DEFINED AS THE RESULT OF
# (1)ONE RANDOM 10-FOLD CV, (2)NO STANDARDIZATION, (3)WITH SHUFFLE (random_seed)

# Set random seed, num_fold, num_loops
seed = 0
fold = 10
loops = 1

# Create original set of features
X_hep_original_no_bias = hep_data.iloc[:, :-1].to_numpy()
X_hep_original = np.insert(
    X_hep_original_no_bias, 0, np.ones(X_hep_original_no_bias.shape[0]), axis=1
)
y_hep_original = hep_data.iloc[:, -1].to_numpy().reshape(-1, 1)

# Account for extra column for bias
index_num_columns = index_num_columns_raw + 1
index_cat_columns = index_cat_columns_raw + 1

# Create an instance of the model object
initial_weights = np.zeros((X_hep_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="dependent",
)
```

```
baseline_accuracy_hep = np.mean(accuracy_test)
baseline_iterations_hep = np.mean(iterations)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Number of iterations: {baseline_iterations_hep}")

plt.title("k-fold accuracy")
plt.xlabel("fold number"), plt.ylabel("accuracy")
plt.plot(accuracy_test[0])
plt.show()

plt.title("k-fold iteration")
plt.xlabel("fold number"), plt.ylabel("iteration")
plt.plot(iterations[0])
plt.show()

"""# Hepatitis Test 0: learning rate"""

# Learning rate = 0.5
print(f"-----------------------------------------------")
print(f"Learning rate = 0.5")
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="large",
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

# Learning rate = 0.1
print(f"-----------------------------------------------")
print(f"Learning rate = 0.1")
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="small",
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Hepatitis Test 1: Standardization"""

# Create an instance of the model object
initial_weights = np.zeros((X_hep_original.shape[1], 1))
```

```
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=index_num_columns,
    random_seed=seed,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Hepatitis Test 2: Initial Weights"""

# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.random.rand(X_hep_original.shape[1], 1)
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print("#### Test group 1: Initial weight random 0 to 1 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")


# Create an instance of the model object, weights set between 0 and 100
initial_weights = np.random.rand(X_hep_original.shape[1], 1) * 100
model = LogisticClassifier(initial_weights)


# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print("#### Test group 2: Initial weight random 0 to 100 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
```

```
print(f"Number of iterations: {np.mean(iterations)}")

# Create an instance of the model object, weights set between 0 and 1, normalized
initial_weights = np.random.randn(X_hep_original.shape[1], 1)
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print("#### Test group 3: Initial weight drawn from standard normal ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Hepatitis Test 3: Stopping Condition"""

# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.zeros((X_hep_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    tolerance=1e-6,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print("#### Test group 1: Tolerance 1e-6 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")


# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.zeros((X_hep_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_original,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    tolerance=1e-9,
)
```

```
print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print("#### Test group 2: Tolerance 1e-9 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Hepatitis Test 4: Removing independent features"""

# indices of independent features to remove
features_to_remove = [2, 3, 4, 8, 9, 16]
Improve_list_removing = []
accuracy_compare = []

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

for i in features_to_remove:
    print(f"Removing {hep_data.columns[i-1]}...")
    X_hep_remove = np.delete(X_hep_original, i, axis=1)

    initial_weights = np.zeros((X_hep_remove.shape[1], 1))
    model = LogisticClassifier(initial_weights)

    accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
        model,
        X_hep_remove,
        y_hep_original,
        num_folds=fold,
        standardize_idx=None,
        random_seed=seed,
        learning_rate="dependent",
    )
    print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
    print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
    print(f"Number of iterations: {np.mean(iterations)}")
    if np.mean(accuracy_test) > baseline_accuracy_hep:
        Improve_list_removing.append(i)
    accuracy_compare.append(100 * (np.mean(accuracy_test) - baseline_accuracy_hep))

print(f"-----------------------------------------------")
# Remove all independent features
print(f"Removing all independent features...")
X_hep_remove = np.delete(X_hep_original, features_to_remove, axis=1)

initial_weights = np.zeros((X_hep_remove.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_remove,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="dependent",
)
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
accuracy_compare.append(100 * (np.mean(accuracy_test) - baseline_accuracy_hep))

print(f"-----------------------------------------------")
```

```
# Remove accu-improved independent features
print(
    "List of removing independent feature which accuracy improvement",
    Improve_list_removing,
)
X_hep_remove = np.delete(X_hep_original, Improve_list_removing, axis=1)

initial_weights = np.zeros((X_hep_remove.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_remove,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="dependent",
)
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)
accuracy_compare.append(100 * (np.mean(accuracy_test) - baseline_accuracy_hep))

print(accuracy_compare)

"""# Hepatitis Test 5: Log transform"""

feature_to_test = index_num_columns

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

# Test 1: replace selected features with log, no standardization
print(f"-----------------------------------------------")
print(f"Test 1: replace selected features with log, w/o standardization")
X_hep_log = X_hep_original.copy()
X_hep_log_1 = log_transform(X_hep_log, feature_to_test)
initial_weights = np.zeros((X_hep_log_1.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_log_1,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)

# Test 2: replace selected features with log, no standardization
print(f"-----------------------------------------------")
print(f"Test 2: replace selected features with log, w/ standardization")
```

```
X_hep_log = X_hep_original.copy()
X_hep_log_2 = log_transform(X_hep_log, feature_to_test)
initial_weights = np.zeros((X_hep_log_2.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_log_2,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=index_num_columns,
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)

# Test 3: append selected features with log, no standardization
print(f"-----------------------------------------------")
print(f"Test 3: append log transform of selected features, w/o standardization")
X_hep_log = X_hep_original.copy()
X_hep_log_3 = log_transform(X_hep_log, feature_to_test, replace=False)
initial_weights = np.zeros((X_hep_log_3.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_log_3,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)


print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)

# Test 4: append selected features with log, with standardization
print(f"-----------------------------------------------")
print(f"Test 4: append log transform of selected features, w standardization")
X_hep_log = X_hep_original.copy()
X_hep_log_4 = log_transform(X_hep_log, feature_to_test, replace=False)
initial_weights = np.zeros((X_hep_log_4.shape[1], 1))
model = LogisticClassifier(initial_weights)
# add log transformed columns to the list of standardization
index_std = [1, 14, 15, 16, 17, 18, 19, 20]

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_log_4,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=index_std,
```

```
    random_seed=seed,
)


print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)

"""# Hepatitis Test 6: Final results

- Delete independent features
- Initial weight: random 0s
- Stopping condition: error=1e-2
- No standardization
"""

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

X_hep_final = X_hep_original.copy()

# Remove features
X_hep_final = np.delete(X_hep_final, [2, 3, 9], axis=1)

# New locations of numerical features
index_num_columns_new = np.array([1, 11, 12, 13, 14, 15])

# initial_weights = np.random.rand(X_hep_final.shape[1], 1)
initial_weights = np.zeros((X_hep_final.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_hep_final,
    y_hep_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    tolerance=1e-2,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Test accuracy change: {100*(np.mean(accuracy_test)-baseline_accuracy_hep):.2f} %"
)

"""# Set baseline accuracy and iterations for bankruptcy data
- Initial weights: zeros
- Learning rate = 1/1+k
- Stopping criteria: epsilon = 1e-2
- No standardization
"""

# RIGHT NOW, BASELINE IS DEFINED AS THE RESULT OF
# (1)ONE RANDOM 10-FOLD CV, (2)NO STANDARDIZATION, (3)WITH SHUFFLE (random_seed)

# Set random seed, num_fold, num_loops
seed = 10
fold = 10
```

```
loops = 1

# Create original set of features
X_bank_original_no_bias = bank_data.iloc[:, :-1].to_numpy()
X_bank_original = np.insert(
    X_bank_original_no_bias, 0, np.ones(X_bank_original_no_bias.shape[0]), axis=1
)
y_bank_original = bank_data.iloc[:, -1].to_numpy().reshape(-1, 1)


# Create an instance of the model object
initial_weights = np.zeros((X_bank_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

baseline_accuracy_bank = np.mean(accuracy_test)
baseline_iterations_bank = np.mean(iterations)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Number of iterations: {baseline_iterations_bank}")

plt.title("k-fold accuracy")
plt.xlabel("fold number"), plt.ylabel("accuracy")
plt.plot(accuracy_test[0])
plt.show()


plt.title("k-fold iteration")
plt.xlabel("fold number"), plt.ylabel("iteration")
plt.plot(iterations[0])
plt.show()

"""# Bankruptcy Test 0: learning rate"""

# Learning rate = 0.5
print(f"-----------------------------------------------")
print(f"Learning rate = 0.5")
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="large",
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
```

```
# Learning rate = 0.1
print(f"-----------------------------------------------")
print(f"Learning rate = 0.1")
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    learning_rate="small",
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_hep:.2f} %")
print(f"Baseline iterations: {baseline_iterations_hep}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Bankruptcy Test 1: Standardization"""

# Create an instance of the model object
initial_weights = np.zeros((X_bank_original.shape[1], 1))
model = LogisticClassifier(initial_weights)


# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx="all",
    random_seed=seed,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Bankruptcy Test 2: Initial Weights"""

# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.random.rand(X_bank_original.shape[1], 1)
model = LogisticClassifier(initial_weights)


# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)
```

```
print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

print("#### Test group 1: Initial weight random 0 to 1 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")


# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.random.rand(X_bank_original.shape[1], 1) * 100
model = LogisticClassifier(initial_weights)


# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print("#### Test group 2: Initial weight random 0 to 100 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.random.randn(X_bank_original.shape[1], 1)
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print("#### Test group 3: Initial weight drawn from standard normal ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Bankruptcy Test 3: Stopping Condition"""

# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.zeros((X_bank_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
```

```
    tolerance=1e-6,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

print("#### Test group 1: Tolerance 1e-6 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")


# Create an instance of the model object, weights set between 0 and 1
initial_weights = np.zeros((X_bank_original.shape[1], 1))
model = LogisticClassifier(initial_weights)

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
    tolerance=1e-9,
)

print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

print("#### Test group 2: Tolerance 1e-9 ####")
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

"""# Bankruptcy Test 4: Removing independent features"""

# generate indices of independent features to remove
Threshold = 0.15
correlation = bank_data.corr()
correlation_arr = np.abs(correlation.to_numpy())
features_to_remove = []
count = 0

for i in correlation_arr[:, -1]:
    if i < Threshold:
        features_to_remove.append(count)
    count += 1


initial_weights = np.zeros((X_bank_original.shape[1], 1))
model = LogisticClassifier(initial_weights)
Improve_list_removing = []

# Run one k-fold cross validation
accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_original,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)
```

```
print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

for i in features_to_remove:
    print(f"Removing {bank_data.columns[i-1]}...")
    X_bank_remove = np.delete(X_bank_original, i, axis=1)

    initial_weights = np.zeros((X_bank_remove.shape[1], 1))
    model = LogisticClassifier(initial_weights)

    accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
        model,
        X_bank_remove,
        y_bank_original,
        num_folds=fold,
        num_loops=loops,
        standardize_idx=None,
        random_seed=seed,
    )
    print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
    print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
    print(f"Number of iterations: {np.mean(iterations)}")
    if np.mean(accuracy_test) > baseline_accuracy_bank:
        Improve_list_removing.append(i)

print(f"-----------------------------------------------")
# Remove all independent features
print(f"Removing all independent features...")
X_bank_remove = np.delete(X_bank_original, features_to_remove, axis=1)

initial_weights = np.zeros((X_bank_remove.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_remove,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")

print(f"-----------------------------------------------")
# Remove accu-improved independent features
print(
    "List of removing independent feature which accuracy improvement",
    Improve_list_removing,
)
print(len(Improve_list_removing))
X_bank_remove = np.delete(X_bank_original, Improve_list_removing, axis=1)

initial_weights = np.zeros((X_bank_remove.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_remove,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
```

```
    standardize_idx=None,
    random_seed=seed,
)
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)

"""# Bankruptcy Test 5: Log transform"""

# Set random seed, num_fold, num_loops
seed = 10
fold = 10
loops = 1

# Create original set of features
X_bank_original_no_bias = bank_data.iloc[:, :-1].to_numpy()
X_bank_original = np.insert(
    X_bank_original_no_bias, 0, np.ones(X_bank_original_no_bias.shape[0]), axis=1
)
y_bank_original = bank_data.iloc[:, -1].to_numpy().reshape(-1, 1)


feature_to_test = range(1, X_bank_original.shape[1] - 1)

# Baseline result

print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

# shift all values up to remove negative values
# X_bank_norm = X_bank_original.copy()
# for i in feature_to_test:
#  X_bank_norm[:,i] = X_bank_norm[:,i] + np.absolute(np.min(X_bank_norm[:,i]))

# Test 1: replace selected features with log, no standardization
print(f"-----------------------------------------------")
print(f"Test 1: replace selected features with log, w/o standardization")
X_bank_log = X_bank_original.copy()
X_bank_log_1 = log_transform(X_bank_log, feature_to_test, bank=True)
initial_weights = np.zeros((X_bank_log_1.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_log_1,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    tolerance=1e-6,
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)

# Test 2: replace selected features with log, no standardization
print(f"-----------------------------------------------")
```

```
print(f"Test 2: replace selected features with log, w/ standardization")
X_bank_log = X_bank_original.copy()
X_bank_log_2 = log_transform(X_bank_log, feature_to_test, bank=True)
initial_weights = np.zeros((X_bank_log_2.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_log_2,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx="all",
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)

# Test 3: replace selected features with log, no standardization
print(f"-----------------------------------------------")
print(f"Test 3: append log transform of selected features, w/o standardization")
X_bank_log = X_bank_original.copy()
X_bank_log_3 = log_transform(X_bank_log, feature_to_test, replace=False, bank=True)
initial_weights = np.zeros((X_bank_log_3.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_log_3,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)

# Test 4: replace selected features with log, with standardization
print(f"-----------------------------------------------")
print(f"Test 4: append log transform of selected features, w standardization")
X_bank_log = X_bank_original.copy()
X_bank_log_4 = log_transform(X_bank_log, feature_to_test, replace=False, bank=True)
initial_weights = np.zeros((X_bank_log_4.shape[1], 1))
model = LogisticClassifier(initial_weights)

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_log_4,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx="all",
    random_seed=seed,
)
```

```
print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)

"""# Bankruptcy Test 6: Final Results

- Delete independent features
- Append log transform of selected features
- Initial weight: random 0s
- Stopping condition: error=1e-6
- No standardization
"""

features_to_remove = np.array(
    [4, 12, 19, 20, 22, 26, 33, 35, 36, 43, 48, 51, 53, 59, 60, 61, 63]
)
# Set random seed, num_fold, num_loops
seed = 10
fold = 10
loops = 1

# Baseline result

print(f"Baseline testing accuracy: {100*baseline_accuracy_bank:.2f} %")
print(f"Baseline iterations: {baseline_iterations_bank}")

X_bank_final = X_bank_original.copy()

# Remove features
X_bank_final = np.delete(X_bank_final, features_to_remove, axis=1)
feature_to_test = range(1, X_bank_final.shape[1] - 1)
X_bank_final = log_transform(X_bank_final, feature_to_test, replace=False, bank=True)
initial_weights = np.zeros((X_bank_final.shape[1], 1))
model = LogisticClassifier(initial_weights)

tolerance_final = 1e-6

accuracy_train, accuracy_test, iterations, weight_store = test_classifier(
    model,
    X_bank_final,
    y_bank_original,
    num_folds=fold,
    num_loops=loops,
    standardize_idx=None,
    tolerance=tolerance_final,
    random_seed=seed,
)

print(f"Training accuracy: {100*np.mean(accuracy_train):.2f} %")
print(f"Testing accuracy: {100*np.mean(accuracy_test):.2f} %")
print(f"Number of iterations: {np.mean(iterations)}")
print(
    f"Training accuracy improve: {100*(np.mean(accuracy_test)-baseline_accuracy_bank):.2f} %"
)
```