# ECSE 551 Mini-Project 1: Implementing Logistic Regression From Scratch

**Yeo Kiah Huah**
McGill University
ID: 261252535

**Kaitlyn Pereira**
McGill University
ID: 261023292

**Melody Wang**
McGill University
ID: 261053910

## Abstract

In this project, we investigate the performance of linear classification models on two benchmark datasets: Chronic Kidney Disease (CKD) and Battery Quality. The CKD dataset consists of 28 numerical features representing biological measurements related to kidney function, while the Battery dataset contains 32 numerical features describing battery attributes. Our primary objective is to implement logistic regression from scratch and analyze its classification performance on both datasets. We employ data analysis techniques to explore the feature distributions and class imbalances. To evaluate our model, we implement 10-fold cross-validation, assessing different learning rates and feature selections to optimize accuracy. We find that logistic regression performance is highly dependent on the learning rate and that careful feature selection can enhance classification accuracy.

## 1 Introduction

Classification is a fundamental problem in machine learning, with applications spanning healthcare, manufacturing, finance, and beyond. Logistic regression, a widely used linear classifier, provides an interpretable and computationally efficient method for binary classification by modeling the probability of class membership using a sigmoid function. In this project, we implement logistic regression from scratch and apply it to two datasets: the Chronic Kidney Disease (CKD) dataset and the Battery dataset. The CKD dataset, consisting of 28 biological features, aims to predict whether a patient has CKD, while the Battery dataset, comprising 32 numerical features, is used to classify batteries as either normal or defective.

We begin by performing data analysis to examine the distributions of features and class labels. This allows us to gain insights into potential feature correlations, class imbalances, and preprocessing requirements. We then construct a logistic regression model from scratch, optimizing its parameters through gradient descent. To assess our model's robustness, we implement 10-fold cross-validation and evaluate performance using accuracy. Additionally, we experiment with different learning rates, feature selection techniques, and potential feature engineering strategies to determine their impact on classification accuracy.

## 2 Dataset

The datasets used for the following experiments were provided to us in myCourses by Professor Armanfard. Through data exploration shown in Fig. 1, we found that both datasets were sufficiently well distributed, and so no additional preprocessing was necessary.

### 2.1 Battery Dataset

The Battery Dataset consists of 32 real-valued features, labeled as feature_1 to feature_32, representing various attributes of batteries manufactured by an unspecified company including batch number, internal resistance, and nominal voltage. The entries fall into two categories, 'Normal' (mapped to a value of '0') and 'Defective (mapped to a value of '1'), each representing 50% of the total entries. There are 732 entries in total in this dataset. In section 3.5, we explore adding new features that are combinations of existing features, to increase the test accuracy.

### 2.2 Kidney Disease Dataset

The Kidney Disease Dataset consists of 28 real-valued features, labeled as feature_1 to feature_28, representing specific biological measurements related to a patient, such as creatinine levels, glomerular filtration rate (GFR), and urine protein levels. The entries fall into two categories, 'Normal' (mapped to a value of '0') and 'CKD' (Chronic Kidney Disease) (mapped to a value of '1'), each representing 50% of the total entries. There are 330 entries in total in this dataset. Again, in section 3.5, we explore adding new features that are combinations of existing features, to increase the test accuracy.

## 3 Methodology and Results

### 3.1 Data Exploration

We began our analysis by conducting a multivariate exploration using heat maps and pair plots to visualize the relationships among the independent variables, as shown in Fig. 2. Our initial strategy was to remove one variable from any pair that exhibited a correlation greater than 0.9, as high multicollinearity can inflate the standard errors of regression coefficients and undermine both the interpretability and reliability of the model. However, for the battery dataset, we found that all feature correlations were below 0.10, and for the CKD dataset, all correlations were below 0.16. Consequently, no features were removed, ensuring that the full set of predictors is retained for subsequent analyses.

### 3.2 Designing the Logistic Regression Class

The logistic regression model was implemented using a Python class that performs binary classification. The model fits the logistic regression model using gradient descent with two available convergence criteria: weight-based and loss-based convergence. The model is initialized with several parameters: the number of features, learning rate (default 0.1), convergence threshold epsilon (default 1e-4), maximum iterations (default 1000), and convergence type (default 'weight'). The weight vector is initialized as a zero vector with an additional element for the bias term. [1]

#### 3.2.1 .fit Method

The training process (fit method) begins by augmenting the input feature matrix with a column of ones to account for the bias term. The algorithm then iteratively updates the weights using gradient descent. For each iteration, the model:

1. Computes the linear combination of features and weights:
$$z = Xw \tag{1}$$
2. Applies the sigmoid activation function to obtain predictions:
$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} \tag{2}$$
3. Calculates the gradient of the loss with respect to the weights:
$$\nabla_w = \frac{1}{n}X^T(\hat{y} - y) \tag{3}$$
4. Updates the weights using the gradient and learning rate:
$$w_{new} = w - \alpha\nabla_w \tag{4}$$

Training continues until either the convergence threshold is met or the maximum number of iterations is reached. The convergence criterion depends on the specified convergence type, which can be modified through the inputs of the logistic regression class.

- For **weight-based convergence**, the algorithm monitors the L2 norm of the weight changes between iterations.
- For **loss-based convergence**, it tracks the change in binary cross-entropy loss between iterations:

$$L = -\frac{1}{n} \sum [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{5}$$

To prevent numerical instability in loss calculations, predictions are clipped to a small epsilon value ($10^{-15}$) away from 0 and 1. [1]

#### 3.2.2 .predict Method

For the predict function, the same feature augmentation and sigmoid transformation to new data was used to compute the probabilities, classifying inputs as positive (1) if the probability exceeds 0.5, and negative (0) otherwise.

### 3.3 Accuracy Evaluation

The function accu_eval was created to calculate the accuracy percentage of predictions made by a model. It takes the predicted labels and the true labels as input and compares them, outputting a percentage of the predictions that were correct.

#### 3.3.1 test_train_split Method

The test_train_split method is used to split a dataset into training and testing sets for machine learning tasks. By partitioning the input features (X) and target values (y), it ensures that a specified proportion (default 70% for training) is reserved for training a model, while the remaining data is used for testing and validation. The function also optionally shuffles the dataset, with a provided random seed for reproducibility, which is important to avoid bias in the training process. For the rest of the experiments, we fix the seed to 42 to ensure consistency when comparing results.

#### 3.3.2 10-fold Cross Validation

The class KFoldCrossValidation is used to evaluate a machine learning model's performance. It splits the dataset into 10 folds, optionally shuffling the data for randomness, and iteratively uses one fold for validation while training the model on the remaining folds. The model's accuracy is calculated for each fold, and the results are averaged to provide an overall performance metric, ensuring a robust assessment of the model's generalization ability.

Initial training of the model with the default parameters gave a 10-fold average training accuracy of **88.27%** and a test accuracy of **85.45%** for the Battery Dataset, and a 10-fold average training accuracy of **66.52%** and a test accuracy of **69.0%** for the CKD Dataset. There is a slight difference in test and train accuracies which is expected, but not enough to classify as overfitting. In the following sections, we discuss different methods to improve these accuracies.

### 3.4 Optimizing the Learning Rate

To evaluate the impact of learning rate on logistic regression performance, we systematically tested various values ranging from 0 to 1 in increments of 0.005 as the learning rate to compile the model. For each learning rate, we trained a logistic regression model, recorded the accuracy, and analyzed the trends. For this model, the epsilon and max_iterations were set as default constants of 1e-4 and 1000 respectively, to give the model ample time to converge at the given rate. We used the custom test_train_split function with a fixed seed, fit the model with the train set, used the test set for predicting, and Accu_eval for the accuracy score.

For the battery dataset, the accuracy peaked at **87.27%** with a learning rate of 0.03, 0.035 and 0.4, after which it dropped a bit then remained constant, as seen in the graph shown in the Appendix Fig. 3. This is because if learning rate is too large, updates might be too aggressive, and the logistic regression model could reach a point where updates stop changing much. For the CKD Dataset, accuracy peaked at **70%** with learning rates from 0.04 to 0.05, again decreasing before remaining constant.The full distribution can be seen in the Appendix Fig. 4.

We also measured the time it took for the model to converge for each fit call. As can be seen in Figure 3b), the time has a downward trend. This is because larger rates make for larger updates, meaning it reaches the convergence condition faster. However, due to these larger updates it might miss the optimal weights.

### 3.5 Optimizing Maximum Number of Iterations

In a similar fashion, we also examined the impact of the maximum number of iterations on accuracy. The expectation was that accuracy would improve with more iterations until reaching a convergence point. We used the best learning rate as determined above, and tested values from 0 to 5000 at intervals of 100.

In both cases, the model converged in less than 1000 iterations, meaning the accuracies found before with the default parameters are still the best so far. The full distributions can be seen in Fig. 5 and 6. For the Battery dataset, the maximum accuracy was found to be **87.27%** with 700-900 iterations. For the Kidney Disease dataset, the maximum accuracy was found to be **70%** which occurred after 300 iterations.

### 3.6 Feature and Model Selection

For this part, we wanted to explore whether accuracy could be improved by modifying the feature set, both removing unnecessary or redundant features and combining good ones to create new, higher-order features. [2] In order to test this, we adopted a systematic approach:

Baseline Model: We first trained the logistic regression model using the regular, unmodified dataset. We saved this test accuracy score to use as a baseline.

Feature Dropping: We then iterated through the dataset, removing one feature at a time and observe the change in accuracy. If the accuracy increased without this feature, we drop it from the improved dataset. The full distributions can be seen in Fig. 7a) and 8a).

Feature Engineering: Next, we iterate through the dataset features again, this time squaring the feature individually and adding it as a new feature. The full distributions can be seen in Fig. 7b) and 8b). Then, taking the new features that resulted in the greatest increase in accuracy, we experiment with different combinations of these new features

For the Battery dataset, feature engineering improved accuracy from a test accuracy of **87.27%** (baseline) to **88.18%**, with squared features contributing positively. For the Kidney dataset, we were able to increase the test accuracy from **70.0%** to **74.0%**. These experiments demonstrate that feature selection and transformation can significantly impact model performance, but effectiveness depends on dataset characteristics.

### 3.7 Stopping criteria

Finally, different stopping criteria for linear regression were investigated: the loss-based convergence model using the cross-entropy loss and the weight-based convergence model using the L2 norm of the weight changes. [3] The accuracies of both models were compared for both datasets. For the battery dataset, the weight-based convergence model had an accuracy of **87.27%** while the loss-based convergence model had an accuracy of **86.36%**. For the kidney disease data, the weight-based convergence model had an accuracy of **69.0%** while the loss-based convergence model had an accuracy of **65.0%** with the same initializing inputs. When compared to the weight-based convergence model, the loss-based convergence model had a decrease in accuracy for both datasets.

Below is a table summarizing the test accuracies at each stage of our investigations:

| Method | Test Accuracy (Battery Dataset) | Test Accuracy (CKD Dataset) |
|---|---|---|
| Baseline | 85.45% | 69.0% |
| Learning Rate | 87.27% | 70.0% |
| Num. Iterations | 87.27% | 70.0% |
| Dropping Bad Features | 87.72% | 73.0% |
| Squaring Good Features | 88.18% | 74.0% |
| Cross-entropy Loss | 86.36% | 65.0% |

Table 1: Summary of Test Accuracy Results for Different Methods

## 4 Discussions and Conclusions

### 4.1 Discussions

Overall, in this project we implemented logistic regression from scratch and evaluated its performance on two real-world datasets: Chronic Kidney Disease (CKD) and Battery Dataset. Our primary objectives were to analyze the classification performance, study the effect of different learning rates, explore feature selection strategies, and implement k-fold cross-validation for robust performance evaluation.

Our findings indicate that logistic regression performs well on both datasets, achieving reasonable classification accuracy. However, the choice of learning rate significantly impacts model convergence and stability. A too-large learning rate led to instability, while a very small learning rate resulted in slow convergence. Through feature selection, we observed that removing certain features improved classification performance, suggesting that some attributes may introduce noise rather than useful information. Additionally, runtime analysis demonstrated that logistic regression is computationally efficient, making it a viable option for real-time or large-scale applications.

### 4.2 Future Directions

There are several potential areas for future exploration:

Alternative Optimization Methods – While gradient descent was used for training, exploring other optimization techniques such as stochastic gradient descent (SGD), Adam, or Newton's method could lead to faster convergence and potentially better performance.

Feature Engineering and Dimensionality Reduction – Investigating more advanced feature selection methods, such as Principal Component Analysis (PCA) [2] or mutual information-based feature selection, could help enhance model interpretability and efficiency.

Comparison with Other Models – Future work could compare logistic regression with other machine learning models such as decision trees, support vector machines, or deep learning approaches to assess performance trade-offs in terms of accuracy, interpretability, and computational cost.

### 4.3 Conclusions

In conclusion, this project provided valuable insights into the practical implementation of logistic regression and its application to real-world datasets. While logistic regression remains a powerful and interpretable baseline classifier, exploring more sophisticated techniques could further improve predictive performance and expand its applicability to more complex classification problems.

## 5 Statement of Contributions

The work was divided equally between each group member:

**Yeo** - Data analysis, accuracy and k-fold validation, writing report.

**Kaitlyn** - Exploration of learning rate/iterations/feature engineering, writing and formatting report.

**Melody** - Logistic Regression class, exploration of different stopping criteria, writing report.

## References

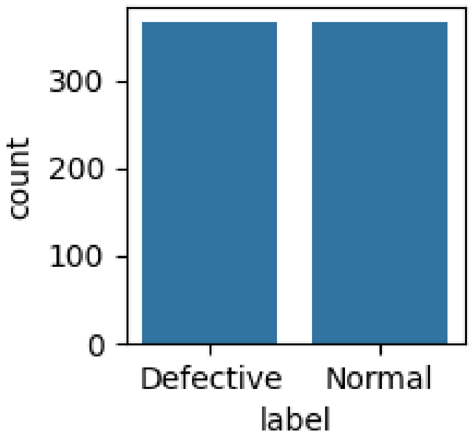
[1] Lecture Slides from ECSE-551: Machine Learning for Engineers by Prof. Armanfard

[2] GeeksforGeeks, "What is Feature Engineering?," GeeksforGeeks, Dec. 3, 2021. [Online]. Available: https://www.geeksforgeeks.org/what-is-feature-engineering/

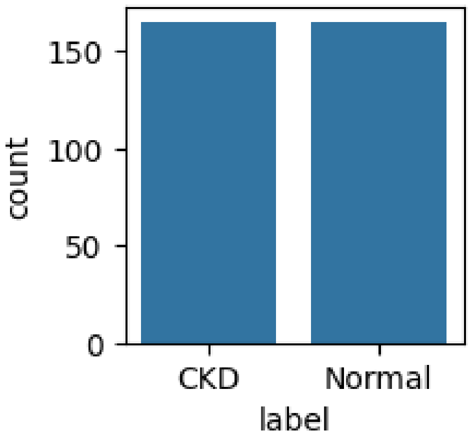
[3] Golik, P., Doetsch, P., & Ney, H. "Cross-entropy vs. squared error training: a theoretical and experimental comparison". In Interspeech, August 2013 (Vol. 13, pp. 1756-1760)

## Appendix A: Report Figures

The following section contains all figures referenced in the report. This figures are also available in better quality in the attached .ipynb file.

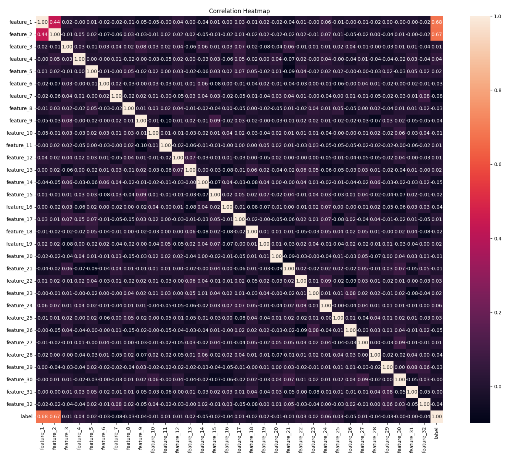


(a) Distribution for Battery Dataset

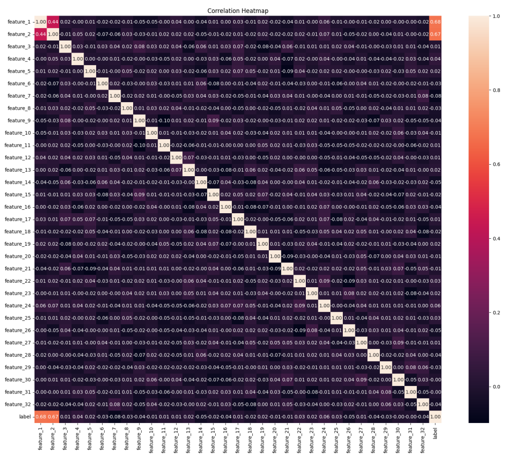


(b) Distribution for Kidney Dataset

Figure 1: Label Distributions for both Datasets

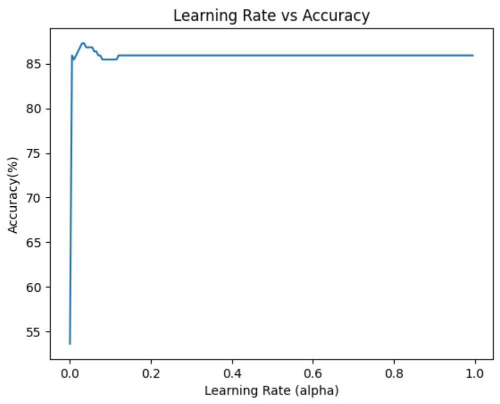


(a) Correlation Matrix for Battery Dataset

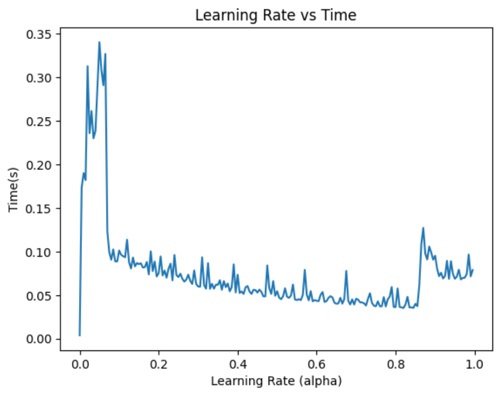


(b) Correlation Matrix for Kidney Dataset

Figure 2: Correlation Matrices for both Datasets

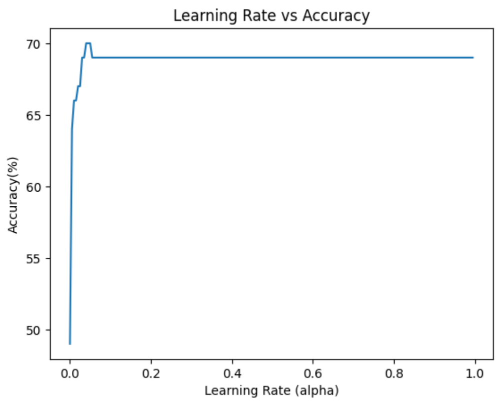


(a) Test accuracy for varying learning rates

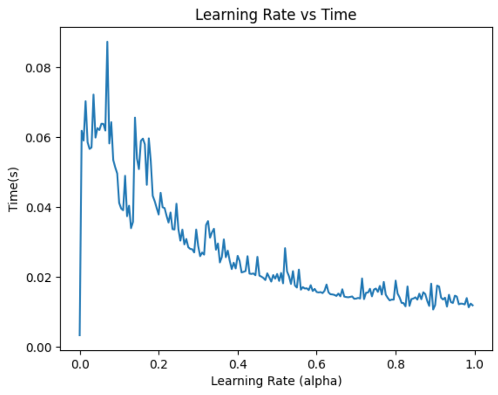


(b) Elapsed time for varying learning rates

Figure 3: Results of Learning Rate Optimization the Battery Dataset

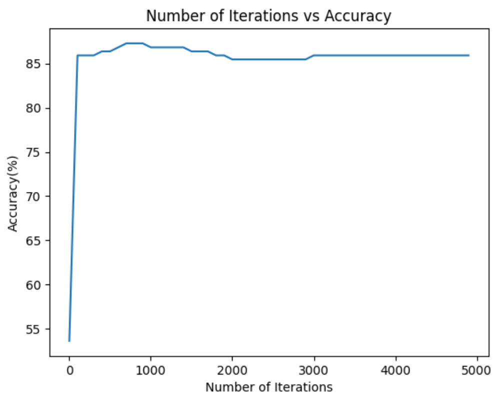


(a) Test accuracy for varying learning rates

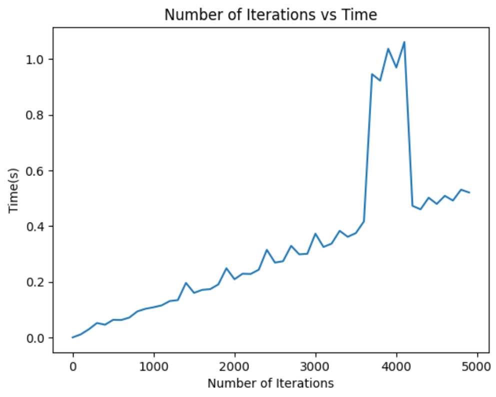


(b) Elapsed time for varying learning rates

Figure 4: Results of Learning Rate Optimization the Kidney Disease Dataset

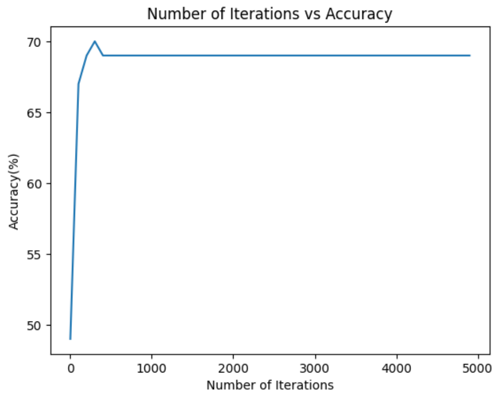


(a) Test accuracy for varying number of iterations

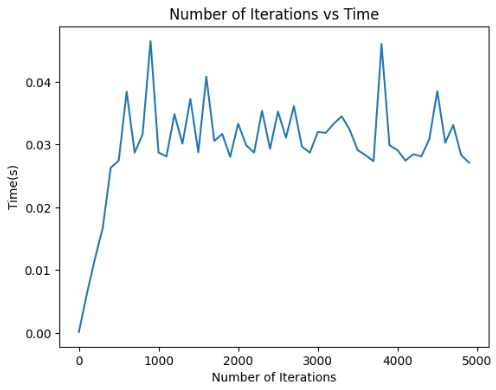


(b) Elapsed time for varying number of iterations

Figure 5: Results of Max. Iteration Optimization the Battery Dataset

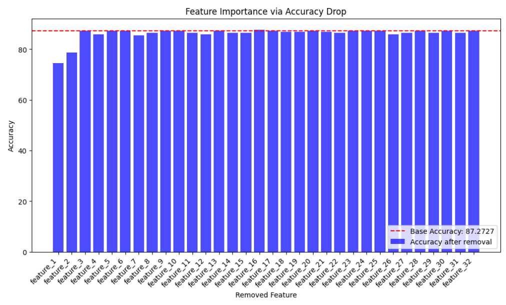


(a) Test accuracy for varying number of iterations

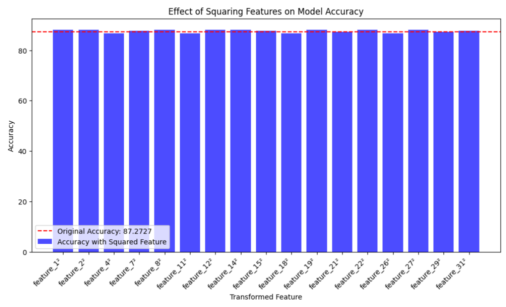


(b) Elapsed time for varying number of iterations

Figure 6: Results of Max. Iteration Optimization the Kidney Disease Dataset

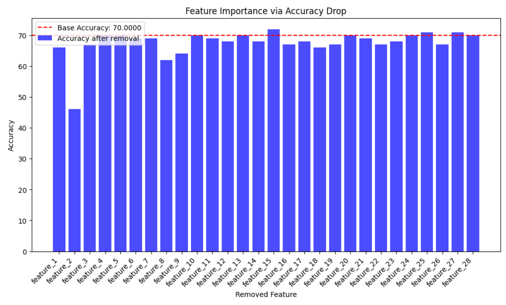


(a) Accuracy after dropping each feature

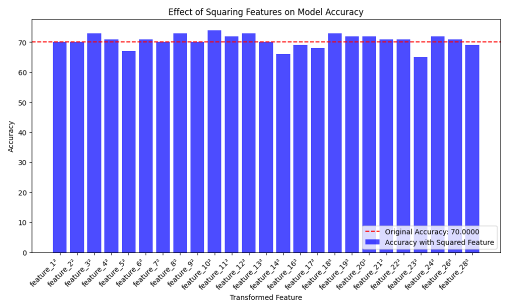


(b) Accuracy after squaring each feature

Figure 7: Accuracy Comparisons for Feature Engineering on the Battery Dataset



(a) Accuracy after dropping each feature



(b) Accuracy after squaring each feature

Figure 8: Accuracy Comparisons for Feature Engineering on the Kidney Disease Dataset

## Appendix B: Code

The following section contains an abridged version of the code without the outputs, in the form of a pdf file of the Jupiter Notebook, as some of the outputs were much too long for a non-scrollable text block. We recommend opening the actual code file, submitted alongside this report, to view the full output blocks and all relevant code/analysis.

# ECSE551Assignment1

February 16, 2025

## 1 Data Exploration

```
[ ]: from google.colab import drive
     drive.mount('/content/drive')

     #replace with wherever you saved the csv files
     path = '/content/drive/My Drive/Colab Notebooks/ECSE551/'

     import pandas as pd, numpy as np
     bd = pd.read_csv(path+'Battery_Dataset.csv')
     kd = pd.read_csv(path+'CKD.csv')
```

### 1.1 Distribution of Classes

#### Battery Data

Investigate the distribution of labels

```
[ ]: import matplotlib.pyplot as plt
     import seaborn as sns
     print(bd["label"].value_counts()/len(bd))

     f, ax = plt.subplots(figsize=(2, 2))
     ax = sns.countplot(x="label", data=bd)
     plt.show()
```

The dataset is very evenly distributed.

```
[ ]: bd.info()
     # Class distribution for Battery dataset
     print("Battery Class Distribution:")
     print(bd.iloc[:, -1].value_counts(), "\n")

     # Convert categorical labels to numerical
     bd.iloc[:, -1] = bd.iloc[:, -1].map({'Normal': 0, 'Defective': 1})

     # Summary statistics
     print("Battery Dataset Statistics:\n", bd.describe(), "\n")
     print("Head:")
     bd.head()
     bd = bd.drop(bd.columns[0], axis=1)
```

**Kidney Data**

Investigate the distribution of labels

[ ]:
```
print(kd["label"].value_counts()/len(kd))

f, ax = plt.subplots(figsize=(2, 2))
ax = sns.countplot(x="label", data=kd)
plt.show()
```

The dataset is very evenly distributed

[ ]:
```
kd.info()
# Class distribution for Battery dataset
print("Battery Class Distribution:")
print(kd.iloc[:, -1].value_counts(), "\n")

# Convert categorical labels to numerical
kd.iloc[:, -1] = kd.iloc[:, -1].map({'Normal': 0, 'CKD': 1})

# Summary statistics
print("Battery Dataset Statistics:\n", kd.describe(), "\n")


print("Head:")
kd.head()
kd = kd.drop(kd.columns[0], axis=1)
```

## 1.2 Investigate Correlations among Variables

**Battery Data**

[ ]:
```
correlation = bd.corr()
plt.figure(figsize=(20,15))
plt.title('Correlation Heatmap')
ax = sns.heatmap(correlation, square=True, annot=True, fmt='.2f',␣
 ↪linecolor='red')
ax.set_xticklabels(ax.get_xticklabels())
ax.set_yticklabels(ax.get_yticklabels())
plt.show()
```

[ ]:
```
sns.pairplot(bd)
plt.show()
```

There are no multicollinearity found among variables, so there is no need to remove any features.

**Kidney Disease Data**

[ ]:
```
correlation_2 = kd.corr()
plt.figure(figsize=(20,15))
plt.title('Correlation Heatmap')
ax = sns.heatmap(correlation_2, square=True, annot=True, fmt='.2f',␣
 ↪linecolor='red')
ax.set_xticklabels(ax.get_xticklabels())
ax.set_yticklabels(ax.get_yticklabels())
plt.show()
```

[ ]:
```
sns.pairplot(kd)
plt.show()
```

There are no multicollinearity found among variables, so there is no need to remove any features.

## 2 Model Implementation

**Logistic Regression**

[ ]:
```
class LogisticRegression:
    def __init__(self, numFeatures, learning_rate=0.1, epsilon=1e-4,␣
 ↪max_iterations = 1000, convergence_type = 'weight'):
        self.weights = np.zeros((numFeatures + 1, 1))  # +1 for bias term
        self.epsilon = epsilon
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.convergence_type = convergence_type

    def sigmoid(self, z):
        return 1/(1 + np.exp(-z))

    def fit(self, X, y):
        X = np.column_stack([np.ones(X.shape[0]), X])  # add bias term
        y = y.reshape(-1, 1)

        num_iterations = 0
        history = []

        if self.convergence_type == 'weight':
            current_err = float('inf')
        else:  # loss-based
            prev_loss = float('inf')
        for i in range(self.max_iterations):
            z = np.dot(X, self.weights)
            y_pred = self.sigmoid(z)

            gradient = np.dot(X.T, (y_pred - y)) / y.shape[0]

            if self.convergence_type == 'weight':
                new_weights = self.weights - self.learning_rate * gradient
                current_err = np.linalg.norm(new_weights - self.weights, 2)
                history.append(current_err)

                if current_err < self.epsilon:
                    break

                self.weights = new_weights

            else:  # loss-based
                self.weights = self.weights - self.learning_rate * gradient

                epsilon = 1e-15  # Small constant to prevent log(0)
                y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
                current_loss = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1␣
↪- y_pred))
                history.append(current_loss)

                loss_change = abs(prev_loss - current_loss)
                if loss_change < self.epsilon:
                    break

                prev_loss = current_loss

            num_iterations += 1

        print(f"\nConverged after {num_iterations} iterations")
        return history

    def predict(self, X):
        X = np.column_stack([np.ones(X.shape[0]), X])
        z = np.dot(X, self.weights)
        probabilities = self.sigmoid(z)
        predictions = (probabilities >= 0.5).astype(int)
        return predictions
```

**Accu-Eval Function**

```
[ ]: def Accu_eval(y_pred, y_test):
         return (np.sum(y_pred == y_test) / len(y_test)) * 100
```

**Test-train-split**

```
[ ]: def train_test_split(X, y, train_size=0.7, shuffle=True, random_state=None):
         n_samples = X.shape[0]
         indices = np.arange(n_samples)

         if shuffle:
             np.random.seed(random_state)
             np.random.shuffle(indices)

         split_idx = int(n_samples * train_size)
         train_indices, test_indices = indices[:split_idx], indices[split_idx:]

         X_train, X_test = X[train_indices], X[test_indices]
         y_train, y_test = y[train_indices], y[test_indices]

         return X_train, X_test, y_train, y_test
```

**K-fold Cross Validation Class**

```
[ ]: class KFoldCrossValidation:
         def __init__(self, k=10, shuffle=True, random_state=None):
             self.k = k
             self.shuffle = shuffle
             self.random_state = random_state

         def split(self, X, y):
             """Generates indices for K-Fold splits."""
             n_samples = X.shape[0]
             indices = np.arange(n_samples)

             if self.shuffle:
                 np.random.seed(self.random_state)
                 np.random.shuffle(indices)

             fold_sizes = np.full(self.k, n_samples // self.k, dtype=int)
             fold_sizes[:n_samples % self.k] += 1  # Distribute remainder
             current = 0

             for fold_size in fold_sizes:
                 val_indices = indices[current:current + fold_size]
                 train_indices = np.concatenate([indices[:current], indices[current␣
      ↪+ fold_size:]])
                 yield train_indices, val_indices
                 current += fold_size
```

```
    def evaluate(self, model, X, y):
        """Trains and evaluates the model on each fold."""
        accuracies = []

        for train_idx, val_idx in self.split(X, y):
            X_train, X_val = X[train_idx], X[val_idx]
            y_train, y_val = y[train_idx], y[val_idx]

            model.fit(X_train, y_train)  # Train model
            y_pred = model.predict(X_val)  # Predict

            acc = Accu_eval(y_pred, y_val)
            accuracies.append(acc)

        print("Accuracy scores for each fold:", accuracies)
        print("Average Accuracy:", np.mean(accuracies))
```

## 3 Model Training

**Battery Data**

```
[ ]: X_bd = bd.iloc[:, :-1].values  # Features
     y_bd= bd.iloc[:, -1].values   # Target

     # Convert y to numeric and reshape
     y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)
     from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_bd_scaled = scaler.fit_transform(X_bd)
     X_bd_train, X_bd_test, y_bd_train, y_bd_test = train_test_split(X_bd_scaled,␣
      ↪y_bd, train_size=0.7, shuffle=True, random_state=42)

     # Train Model on Training Set
     log_reg = LogisticRegression(numFeatures=X_bd.shape[1])
     log_reg.fit(X_bd_train, y_bd_train)

     y_pred = log_reg.predict(X_bd_test)
     accuracy = Accu_eval(y_bd_test, y_pred)
     print("Test Accuracy: ", accuracy)

     # Perform Custom 10-Fold Cross-Validation
     kf = KFoldCrossValidation(k=10, shuffle=True, random_state=42)
     kf.evaluate(log_reg, X_bd_train, y_bd_train)  # Train & Evaluate
```

**Kidney Disease Data**

```
[ ]: X_kd = kd.iloc[:, :-1].values  # Features
     y_kd= kd.iloc[:, -1].values   # Target

     # Convert y to numeric and reshape
     y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)
     from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_kd_scaled = scaler.fit_transform(X_kd)
     X_kd_train, X_kd_test, y_kd_train, y_kd_test = train_test_split(X_kd_scaled,␣
      ↪y_kd, train_size=0.7, shuffle=True, random_state=42)

     # Train Model on Training Set
     log_reg = LogisticRegression(numFeatures=X_kd.shape[1])
     log_reg.fit(X_kd_train, y_kd_train)

     y_pred = log_reg.predict(X_kd_test)
     accuracy = Accu_eval(y_kd_test, y_pred)
     print("Test Accuracy: ", accuracy)

     # Perform Custom 10-Fold Cross-Validation
     kf = KFoldCrossValidation(k=10, shuffle=True, random_state=42)
     kf.evaluate(log_reg, X_kd_train, y_kd_train)  # Train & Evaluate
```

Use scikit-learn package to confirm model is working correctly

```
[ ]: from sklearn.model_selection import KFold, cross_val_score
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score

     df = kd # change to kd if wanna test for kidney disease data
     X = df.iloc[:, :-1].values  # All columns except last
     y = df.iloc[:, -1].values   # Last column (target)
     y = pd.to_numeric(y, errors='coerce')
     log_reg = LogisticRegression(max_iter=10000, solver='lbfgs')

     # Set up 10-fold cross-validation
     kf = KFold(n_splits=10, shuffle=True, random_state=42)
     accuracy_scores = cross_val_score(log_reg, X, y, cv=kf, scoring='accuracy')
     print("Accuracy scores for each fold:", accuracy_scores)
     print("Average accuracy:", np.mean(accuracy_scores))
```

## 4 Testing different learning rates

```
[ ]: import time
     import matplotlib.pyplot as plt

     def test_learning_rate(X, Y, start, stop, step, iterations):
       rates_arr = np.arange(start, stop, step)
       accuracy_arr = np.zeros(len(rates_arr))
       time_arr = np.zeros(len(rates_arr))

       for i in range(len(rates_arr)):
         rate = rates_arr[i]
         print("fitting model with learning rate = ", rate)
         model = LogisticRegression(numFeatures=X.shape[1], learning_rate=rate,␣
      ↪epsilon=1e-4, max_iterations = iterations)
         scaler = StandardScaler()
         X_scaled = scaler.fit_transform(X)
         X_train, X_test, y_train, y_test = train_test_split(X_scaled, Y,␣
      ↪train_size=0.7, shuffle=True, random_state=42)
         t1 = time.time()
         model.fit(X_train, y_train)
         t2 = time.time()
         y_pred = model.predict(X_test)
         accuracy_arr[i] = Accu_eval(y_test, y_pred)
         time_arr[i] = t2-t1

       plt.plot(rates_arr,accuracy_arr);
       plt.xlabel('Learning Rate (alpha)'); plt.ylabel('Accuracy(%)')
       plt.title('Learning Rate vs Accuracy')
       plt.show()
       plt.plot(rates_arr,time_arr)
       plt.xlabel('Learning Rate (alpha)'); plt.ylabel('Time(s)')
       plt.title('Learning Rate vs Time')
       plt.show()

       ind = np.where(accuracy_arr == max(accuracy_arr))
       print("Maximum Accuracy:",max(accuracy_arr),"%","for Learning Rate:
      ↪",list(rates_arr[ind]))
```

```
[ ]: X_bd = bd.iloc[:, :-1].values  # Features
     y_bd= bd.iloc[:, -1].values   # Target

     # Convert y to numeric and reshape
     y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)

     # Train Model on Training Set
     test_learning_rate(X_bd ,y_bd,0,1,0.005,1000)
```

```
[ ]: X_kd = kd.iloc[:, :-1].values  # Features
     y_kd= kd.iloc[:, -1].values    # Target

     # Convert y to numeric and reshape
     y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)

     # Train Model on Training Set
     test_learning_rate(X_kd,y_kd,0,1,0.005,1000)
```

## 5 Testing different num. iterations

```
[ ]: def test_iterations(X, Y, rate, start, stop, step):
       iterations_arr = np.arange(start, stop, step)
       accuracy_arr = np.zeros(len(iterations_arr))
       time_arr = np.zeros(len(iterations_arr))

       for i in range(len(iterations_arr)):
         iterations = iterations_arr[i]
         print("fitting model with ", iterations, " iterations")
         model = LogisticRegression(numFeatures=X.shape[1], learning_rate=rate,␣
      ↪epsilon=1e-4, max_iterations = iterations)
         scaler = StandardScaler()
         X_scaled = scaler.fit_transform(X)
         X_train, X_test, y_train, y_test = train_test_split(X_scaled, Y,␣
      ↪train_size=0.7, shuffle=True, random_state=42)
         t1 = time.time()
         model.fit(X_train, y_train)
         t2 = time.time()
         y_pred = model.predict(X_test)
         accuracy_arr[i] = Accu_eval(y_test, y_pred)
         time_arr[i] = t2-t1

       plt.plot(iterations_arr,accuracy_arr);
       plt.xlabel('Number of Iterations'); plt.ylabel('Accuracy(%)')
       plt.title('Number of Iterations vs Accuracy')
       plt.show()
       plt.plot(iterations_arr,time_arr)
       plt.xlabel('Number of Iterations'); plt.ylabel('Time(s)')
       plt.title('Number of Iterations vs Time')
       plt.show()

       ind = np.where(accuracy_arr == max(accuracy_arr))
       print("Maximum Accuracy:",max(accuracy_arr),"%","for Number of Iterations:
      ↪",list(iterations_arr[ind]))
```

**Battery Data**

[ ]:
```
X_bd = bd.iloc[:, :-1].values  # Features
y_bd= bd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)

# Train Model on Training Set
test_iterations(X_bd,y_bd,0.04,0,5000,100)
```

**Kidney Disease Data**

[ ]:
```
X_kd = kd.iloc[:, :-1].values  # Features
y_kd= kd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)

# Train Model on Training Set
test_iterations(X_kd,y_kd,0.17,0,5000,100)
```

## 6 Testing feature importance/new features

Approach: - train model using all features - drop one feature at a time, observe change in accuracy - determine important features (where dropping said feature causes the largest drop in accuracy) - try squaring them and combining to make new features

**Battery Data**

[ ]:
```
X_bd = bd.iloc[:, :-1].values  # Features
y_bd= bd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_bd_scaled = scaler.fit_transform(X_bd)
X_bd_train, X_bd_test, y_bd_train, y_bd_test = train_test_split(X_bd_scaled,␣
 ↪y_bd, train_size=0.7, shuffle=True, random_state=42)

# Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_bd.shape[1], learning_rate=0.03,␣
 ↪epsilon=1e-4, max_iterations = 1000)
log_reg.fit(X_bd_train, y_bd_train)

# Accuracy Evaluation
test_y = log_reg.predict(X_bd_test)
```

```
base_acc = Accu_eval(test_y, y_bd_test)
print("Baseline Accuracy: ", base_acc)
```

[ ]:
```
feature_accuracies = {}
features = bd.columns[:-1]
selected_features = []
accuracies = []  # To store accuracy values
removed_features = []  # To store removed feature names

for i, feature in enumerate(features):
    X_bd_train_mod = np.delete(X_bd_train, i, axis=1)
    X_bd_test_mod = np.delete(X_bd_test, i, axis=1)
    log_reg = LogisticRegression(numFeatures=X_bd_train_mod.shape[1],␣
 ↪learning_rate=0.03, epsilon=1e-4, max_iterations=1000)
    log_reg.fit(X_bd_train_mod, y_bd_train)
    y_pred_mod = log_reg.predict(X_bd_test_mod)
    acc = Accu_eval(y_pred_mod, y_bd_test)

    feature_accuracies[feature] = acc
    accuracies.append(acc)
    removed_features.append(feature)  # Store the feature removed

    print(f"Accuracy after removing {feature}: {acc:.4f}")

    # Keep features if accuracy doesn't drop significantly
    if acc < base_acc:
        selected_features.append(i)

# Plot accuracy vs. removed feature
plt.figure(figsize=(12, 6))
plt.bar(removed_features, accuracies, color='b', alpha=0.7, label="Accuracy␣
 ↪after removal")
plt.axhline(y=base_acc, color='r', linestyle='--', label=f"Base Accuracy:␣
 ↪{base_acc:.4f}")
plt.xticks(rotation=45, ha='right')
plt.xlabel("Removed Feature")
plt.ylabel("Accuracy")
plt.title("Feature Importance via Accuracy Drop")
plt.legend(loc='lower right')
plt.show()

print("\nAll features: ")
print(features)

print("\nSelected features: ")
print(selected_features)
```

We drop any features where the accuracy does not change when removed

[ ]:
```
X_train_reduced = X_bd_train[:, selected_features]
X_test_reduced = X_bd_test[:, selected_features]

log_reg = LogisticRegression(numFeatures=X_train_reduced.shape[1],␣
 ↪learning_rate=0.03, epsilon=1e-4, max_iterations=1000)
log_reg.fit(X_train_reduced, y_bd_train)

y_pred_reduced = log_reg.predict(X_test_reduced)
acc_reduced = Accu_eval(y_bd_test, y_pred_reduced)
print(f"Accuracy with selected features: {acc_reduced:.4f}")
```

[ ]:
```
accuracy_results = {}
features = bd.columns[:-1]
transformed_features = []  # To store transformed feature names
accuracies = []  # To store accuracy values

X_train_reduced = X_bd_train[:, selected_features]
X_test_reduced = X_bd_test[:, selected_features]

# Iterate through each feature and square it, then add to the model
for i in selected_features:
    X_train_squared = X_bd_train[:, i] ** 2
    X_test_squared = X_bd_test[:, i] ** 2

    X_train_expanded = np.hstack((X_train_reduced, X_train_squared.reshape(-1,␣
 ↪1)))
    X_test_expanded = np.hstack((X_test_reduced, X_test_squared.reshape(-1, 1)))

    log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
 ↪learning_rate=0.03, epsilon=1e-4, max_iterations=1000)
    log_reg.fit(X_train_expanded, y_bd_train)

    y_pred_expanded = log_reg.predict(X_test_expanded)
    acc_expanded = Accu_eval(y_bd_test, y_pred_expanded)

    feature_name = f"{features[i]}^2"
    accuracy_results[feature_name] = acc_expanded
    transformed_features.append(feature_name)
    accuracies.append(acc_expanded)

    print(f"Accuracy with {feature_name}: {acc_expanded:.4f}")

# Plot the accuracy changes
plt.figure(figsize=(12, 6))
plt.bar(transformed_features, accuracies, color='b', alpha=0.7, label="Accuracy␣
↪with Squared Feature")
plt.axhline(y=base_acc, color='r', linestyle='--', label=f"Original Accuracy:␣
↪{base_acc:.4f}")
plt.xticks(rotation=45, ha='right')
plt.xlabel("Transformed Feature")
plt.ylabel("Accuracy")
plt.title("Effect of Squaring Features on Model Accuracy")
plt.legend()
plt.show()

# Step 4: Print all results
print("\nAccuracy results for all squared features:")
for feature, accuracy in accuracy_results.items():
    print(f"{feature}: {accuracy:.4f}")
```

```
[ ]: print(selected_features)

X_train_reduced = X_bd_train[:, selected_features]
X_test_reduced = X_bd_test[:, selected_features]

X_train_squared_0 = X_train_reduced[:, 0] ** 2
X_test_squared_0 = X_test_reduced[:, 0] ** 2

X_train_squared_1 = X_train_reduced[:, 1] ** 2
X_test_squared_1 = X_test_reduced[:, 1] ** 2

X_train_interaction = X_train_reduced[:, 0] * X_train_reduced[:, 1]
X_test_interaction = X_test_reduced[:, 0] * X_test_reduced[:, 1]

X_train_expanded = np.hstack((
    X_train_reduced,
    X_train_squared_0.reshape(-1, 1),
    X_train_squared_1.reshape(-1, 1),
    X_train_interaction.reshape(-1, 1)
))

X_test_expanded = np.hstack((
    X_test_reduced,
    X_test_squared_0.reshape(-1, 1),
    X_test_squared_1.reshape(-1, 1),
    X_test_interaction.reshape(-1, 1)
))

log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
↪learning_rate=0.03, epsilon=1e-4, max_iterations=1000)
log_reg.fit(X_train_expanded, y_bd_train)
```

```
y_pred_expanded = log_reg.predict(X_test_expanded)
acc_expanded = Accu_eval(y_bd_test, y_pred_expanded)
print(f"Accuracy with feature engineering: {acc_expanded:.4f}")
```

With feature engineering, we were able to increase the accuracy from the baseline of **87.2727** to **88.1818**

```
[ ]: # Train Model on Training Set
     log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
      ↪learning_rate=0.03, epsilon=1e-4, max_iterations=1000)
     log_reg.fit(X_train_expanded, y_bd_train)

     # Perform Custom 10-Fold Cross-Validation
     kf = KFoldCrossValidation(k=10, shuffle=True, random_state=42)
     kf.evaluate(log_reg, X_train_expanded, y_bd_train)  # Train & Evaluate
```

**Kidney Disease Data**

```
[ ]: X_kd = kd.iloc[:, :-1].values  # Features
     y_kd= kd.iloc[:, -1].values   # Target

     # Convert y to numeric and reshape
     y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)
     from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_kd_scaled = scaler.fit_transform(X_kd)
     X_kd_train, X_kd_test, y_kd_train, y_kd_test = train_test_split(X_kd_scaled,␣
      ↪y_kd, train_size=0.7, shuffle=True, random_state=42)

     # Train Model on Training Set
     log_reg = LogisticRegression(numFeatures=X_kd.shape[1], learning_rate=0.04,␣
      ↪epsilon=1e-4, max_iterations = 1000)
     log_reg.fit(X_kd_train, y_kd_train)

     # Accuracy Evaluation
     test_y = log_reg.predict(X_kd_test)
     base_acc = Accu_eval(y_kd_test, test_y)
     print("Baseline Accuracy: ", base_acc)
```

```
[ ]: feature_accuracies = {}
     features = kd.columns[:-1]
     selected_features = []
     accuracies = []  # To store accuracy values
     removed_features = []  # To store removed feature names
```

```
for i, feature in enumerate(features):
    X_kd_train_mod = np.delete(X_kd_train, i, axis=1)
    X_kd_test_mod = np.delete(X_kd_test, i, axis=1)
    log_reg = LogisticRegression(numFeatures=X_kd_train_mod.shape[1],␣
 ↪learning_rate=0.04, epsilon=1e-4, max_iterations=1000)
    log_reg.fit(X_kd_train_mod, y_kd_train)
    y_pred_mod = log_reg.predict(X_kd_test_mod)
    acc = Accu_eval(y_pred_mod, y_kd_test)

    feature_accuracies[feature] = acc
    accuracies.append(acc)
    removed_features.append(feature)  # Store the feature removed

    print(f"Accuracy after removing {feature}: {acc:.4f}")

    # Keep features if accuracy doesn't drop significantly
    if acc < base_acc+0.1:
        selected_features.append(i)

# Plot accuracy vs. removed feature
plt.figure(figsize=(12, 6))
plt.bar(removed_features, accuracies, color='b', alpha=0.7, label="Accuracy␣
 ↪after removal")
plt.axhline(y=base_acc, color='r', linestyle='--', label=f"Base Accuracy:␣
 ↪{base_acc:.4f}")
plt.xticks(rotation=45, ha='right')
plt.xlabel("Removed Feature")
plt.ylabel("Accuracy")
plt.title("Feature Importance via Accuracy Drop")
plt.legend(loc='lower right')
plt.show()

print("\nAll features: ")
print(features)

print("\nSelected features: ")
print(selected_features)
```

We drop any features where the accuracy increases when removed

```
[ ]: X_train_reduced = X_kd_train[:, selected_features]
     X_test_reduced = X_kd_test[:, selected_features]

     log_reg = LogisticRegression(numFeatures=X_train_reduced.shape[1],␣
      ↪learning_rate=0.04, epsilon=1e-4, max_iterations=1000)
     log_reg.fit(X_train_reduced, y_kd_train)
```

```
y_pred_reduced = log_reg.predict(X_test_reduced)
acc_reduced = Accu_eval(y_kd_test, y_pred_reduced)
print(f"Accuracy with selected features: {acc_reduced:.4f}")
```

```
[ ]: accuracy_results = {}
features = kd.columns[:-1]
transformed_features = []  # To store transformed feature names
accuracies = []  # To store accuracy values

X_train_reduced = X_kd_train[:, selected_features]
X_test_reduced = X_kd_test[:, selected_features]

# Iterate through each feature and square it, then add to the model
for i in selected_features:
    X_train_squared = X_kd_train[:, i] ** 2
    X_test_squared = X_kd_test[:, i] ** 2

    X_train_expanded = np.hstack((X_train_reduced, X_train_squared.reshape(-1,␣
 ↪1)))
    X_test_expanded = np.hstack((X_test_reduced, X_test_squared.reshape(-1, 1)))

    log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
 ↪learning_rate=0.04, epsilon=1e-4, max_iterations=1000)
    log_reg.fit(X_train_expanded, y_kd_train)

    y_pred_expanded = log_reg.predict(X_test_expanded)
    acc_expanded = Accu_eval(y_kd_test, y_pred_expanded)

    feature_name = f"{features[i]}^2"
    accuracy_results[feature_name] = acc_expanded
    transformed_features.append(feature_name)
    accuracies.append(acc_expanded)

    print(f"Accuracy with {feature_name}: {acc_expanded:.4f}")

# Plot the accuracy changes
plt.figure(figsize=(12, 6))
plt.bar(transformed_features, accuracies, color='b', alpha=0.7, label="Accuracy␣
 ↪with Squared Feature")
plt.axhline(y=base_acc, color='r', linestyle='--', label=f"Original Accuracy:␣
 ↪{base_acc:.4f}")
plt.xticks(rotation=45, ha='right')
plt.xlabel("Transformed Feature")
plt.ylabel("Accuracy")
plt.title("Effect of Squaring Features on Model Accuracy")
plt.legend(loc='lower right')
plt.show()
```

```
print("\nAccuracy results for all squared features:")
for feature, accuracy in accuracy_results.items():
    print(f"{feature}: {accuracy:.4f}")
```

```
[ ]: print(selected_features)

X_train_reduced = X_kd_train[:, selected_features]
X_test_reduced = X_kd_test[:, selected_features]

X_train_squared_0 = X_train_reduced[:,2] ** 2
X_test_squared_0 = X_test_reduced[:, 2] ** 2

X_train_squared_1 = X_train_reduced[:, 9] ** 2
X_test_squared_1 = X_test_reduced[:, 9] ** 2

X_train_squared_2 = X_train_reduced[:, 11] ** 2
X_test_squared_2 = X_test_reduced[:, 11] ** 2

X_train_expanded = np.hstack((
    X_train_reduced,
    X_train_squared_0.reshape(-1, 1),
    X_train_squared_1.reshape(-1, 1),
    X_train_squared_2.reshape(-1, 1)
))

X_test_expanded = np.hstack((
    X_test_reduced,
    X_test_squared_0.reshape(-1, 1),
    X_test_squared_1.reshape(-1, 1),
    X_test_squared_2.reshape(-1, 1)
))


log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
 ↪learning_rate=0.04, epsilon=1e-4, max_iterations=1000)
log_reg.fit(X_train_expanded, y_kd_train)

y_pred_expanded = log_reg.predict(X_test_expanded)
acc_expanded = Accu_eval(y_kd_test, y_pred_expanded)
print(f"Accuracy with feature engineering: {acc_expanded:.4f}")
```

```
[ ]: # Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_train_expanded.shape[1],␣
 ↪learning_rate=0.04, epsilon=1e-4, max_iterations=1000)
log_reg.fit(X_train_expanded, y_kd_train)
```

```
# Perform Custom 10-Fold Cross-Validation
kf = KFoldCrossValidation(k=10, shuffle=True, random_state=42)
kf.evaluate(log_reg, X_train_expanded, y_kd_train)  # Train & Evaluate
```

Adding the interaction term actually makes the accuracy worse. so the best we can do with feature engineering is an accuracy of **74.0**, up from the baseline of **70.0**

## 7 Testing different stopping criterion: Loss-based gradient descent:

The logistic regression was tested using two different stopping criterion.

Weight-Based Convergence: Stops training when the difference between consecutive weight updates is below the defined epsilon threshold.

Loss-Based Convergence: Terminates when the absolute difference between consecutive cross-entropy loss values is less than epsilon.

**Battery Data**

[ ]:
```
X_bd = bd.iloc[:, :-1].values  # Features
y_bd= bd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_bd_scaled = scaler.fit_transform(X_bd)
X_bd_train, X_bd_test, y_bd_train, y_bd_test = train_test_split(X_bd_scaled,␣
 ↪y_bd, train_size=0.7, shuffle=True, random_state=42)

# Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_bd.shape[1], learning_rate=0.03,␣
 ↪epsilon=1e-4, max_iterations = 1000)
log_reg.fit(X_bd_train, y_bd_train)

# Accuracy Evaluation
test_y = log_reg.predict(X_bd_test)
acc = Accu_eval(y_bd_test, test_y)
print("Accuracy with weight-based gradient descent: ", acc)
```

[ ]:
```
X_bd = bd.iloc[:, :-1].values  # Features
y_bd= bd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_bd = pd.to_numeric(y_bd, errors='coerce').reshape(-1, 1)
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
X_bd_scaled = scaler.fit_transform(X_bd)
X_bd_train, X_bd_test, y_bd_train, y_bd_test = train_test_split(X_bd_scaled,␣
 ↪y_bd, train_size=0.7, shuffle=True, random_state=42)

# Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_bd.shape[1], learning_rate=0.04,␣
 ↪epsilon=1e-4, max_iterations = 1000, convergence_type = 'loss' )
log_reg.fit(X_bd_train, y_bd_train)

# Accuracy Evaluation
test_y = log_reg.predict(X_bd_test)
acc = Accu_eval(y_bd_test, test_y)
print("Accuracy with loss-based gradient descent: ", acc)
```

For the battery data, the loss-based gradient descent gives an accuracy of 86.36 compared to the weight-based gradient descent which gives 87.27.

**Kidney Disease Data**

[ ]:
```
X_kd = kd.iloc[:, :-1].values  # Features
y_kd= kd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)

scaler = StandardScaler()
X_kd_scaled = scaler.fit_transform(X_kd)
X_kd_train, X_kd_test, y_kd_train, y_kd_test = train_test_split(X_kd_scaled,␣
 ↪y_kd, train_size=0.7, shuffle=True, random_state=42)

# Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_kd.shape[1], learning_rate=0.03,␣
 ↪epsilon=1e-4, max_iterations = 1000 )
log_reg.fit(X_kd_train, y_kd_train)

# Accuracy Evaluation
test_y = log_reg.predict(X_kd_test)
acc = Accu_eval(y_kd_test, test_y)
print("Accuracy with weight-based gradient descent: ", acc)
```

[ ]:
```
X_kd = kd.iloc[:, :-1].values  # Features
y_kd= kd.iloc[:, -1].values   # Target

# Convert y to numeric and reshape
y_kd = pd.to_numeric(y_kd, errors='coerce').reshape(-1, 1)
```

```
scaler = StandardScaler()
X_kd_scaled = scaler.fit_transform(X_kd)
X_kd_train, X_kd_test, y_kd_train, y_kd_test = train_test_split(X_kd_scaled,␣
 ↪y_kd, train_size=0.7, shuffle=True, random_state=42)

# Train Model on Training Set
log_reg = LogisticRegression(numFeatures=X_kd.shape[1], learning_rate=0.04,␣
 ↪epsilon=1e-4, max_iterations = 1000, convergence_type = 'loss' )
log_reg.fit(X_kd_train, y_kd_train)

# Accuracy Evaluation
test_y = log_reg.predict(X_kd_test)
acc = Accu_eval(y_kd_test, test_y)
print("Accuracy with loss-based gradient descent: ", acc)
```

For the kidney disease data, the loss-based gradient descent gives an accuracy of 65.0 compared to the weight-based gradient descent which gives 69.0.

(functions to get code in pdf form)

```
[ ]: !apt-get install -y pandoc
     !apt-get install -y texlive-xetex texlive-fonts-recommended␣
      ↪texlive-plain-generic
```

```
[ ]: !jupyter nbconvert --to pdf "/content/drive/My Drive/Colab Notebooks/ECSE551/
      ↪ECSE551Assignment1.ipynb"
```