CHITTAGONG UNIVERSITY OF ENGINEERING & TECHNOLOGY

Lab Report
CSE-314
Data Communication (Sessional)
Department of Computer Science & Engineering

Report Title: Animal disease prediction and cattle disease prediction using various machine learning models

Section: B
Group: B2

Submitted By
Shahriar Mhamud Turab (2104113)
Tahmima Hoque Eid (2104114)
Chumui Tripura (2104131)

Submitted To
Dr. Mahfuzulhoq Chowdhury
(Professor, CSE, CUET)

Date of Submission
21 July, 2025

Introduction

Animal healthcare, particularly in veterinary diagnostics, plays a critical role in ensuring the well-being of livestock and pets. Early and accurate disease detection is essential for preventing outbreaks, improving animal welfare, and reducing economic losses in the agriculture and farming sectors. Traditional diagnostic approaches, while effective, are often time-consuming, require expert knowledge, and may not be feasible in all settings.

To address these challenges, machine learning (ML) offers powerful tools for automating disease prediction based on observable symptoms and physiological data. In this report, we explore two ML-driven systems:

1. **Animal Disease Prediction Using Symptoms**: A generalized system that leverages a rich dataset containing various animal types, breeds, symptoms, and biological parameters to predict the most likely disease.
2. **Cattle Disease Prediction**: A focused approach targeting cattle, where four classification algorithms—K-Nearest Neighbors (KNN), Naive Bayes, Decision Tree, and Random Forest—are applied to identify diseases based on key symptoms.

By combining domain knowledge in veterinary science with the analytical power of machine learning, these systems aim to support veterinarians, farmers, and researchers in making faster and more informed decisions.

Objectives

The main objectives of this report are:

- To understand the structure and significance of animal health datasets, including features such as symptoms, animal type, breed, physiological measures, and observed outcomes.
- To develop a machine learning model that can predict animal diseases based on input symptoms and parameters, improving the speed and accuracy of diagnosis.
- To implement and compare multiple ML algorithms (KNN, Naive Bayes, Decision Tree, and Random Forest) in the context of cattle disease prediction, evaluating each model's performance.
- To explore the practical applications of ML in veterinary diagnostics, especially for early detection and treatment planning in resource-constrained environments.
- To build a user-friendly prediction system, potentially deployable via GUI (e.g., Tkinter), to assist non-expert users in identifying possible diseases based on symptom inputs.

Source Code and Explanation:

Library Imports and Dataset Loading

In this section, we import essential libraries for data handling, model building, and evaluation. Key libraries include pandas and numpy for data manipulation, seaborn and matplotlib for visualization, and sklearn functions like train_test_split, cross_val_score, and RandomizedSearchCV for model training and evaluation. For feature selection, we use SelectKBest and VarianceThreshold, while classification_report, accuracy_score, and f1_score are used for performance metrics. We also handle class imbalance with compute_class_weight, and use RandomForestClassifier and XGBClassifier for modeling. chi2_contingency is used for statistical analysis, and joblib is utilized for model serialization. The dataset is loaded using pd.read_csv() and displayed with df.head() for verification.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns
import joblib
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, cross_val_score, RandomizedSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.metrics import classification_report, accuracy_score, f1_score, make_scorer
from sklearn.utils.class_weight import compute_class_weight
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import VarianceThreshold
from scipy.stats import chi2_contingency
from xgboost import XGBClassifier
```

```
df = pd.read_csv('cleaned_animal_disease_prediction.csv')
df.head()
```

Dataset Overview

The dataset contains 431 entries (rows) and 22 columns, as shown in the output of df.info(). Each column represents a different feature, with all entries being non-null, indicating that no missing data is present.

- **Data Types:**
    - Numerical Columns: Age, Weight, Duration, Body_Temperature, Heart_Rate, all of which are of type int64 or float64.
    - Categorical Columns: These include features such as Animal_Type, Breed, Gender, and various symptoms (e.g., Symptom_1, Vomiting, Lameness), which are of type object.

- **Target Variable:**
    - Disease_Prediction is the target variable, indicating the classification of whether the animal has a disease or not, which is of type object.

```
# Data shape and type
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 431 entries, 0 to 430
Data columns (total 22 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Animal_Type         431 non-null    object 
 1   Breed               431 non-null    object 
 2   Age                 431 non-null    int64  
 3   Gender              431 non-null    object 
 4   Weight              431 non-null    float64
 5   Symptom_1           431 non-null    object 
 6   Symptom_2           431 non-null    object 
 7   Symptom_3           431 non-null    object 
 8   Symptom_4           431 non-null    object 
 9   Duration            431 non-null    object 
 10  Appetite_Loss       431 non-null    object 
 11  Vomiting            431 non-null    object 
 12  Diarrhea            431 non-null    object 
 13  Coughing            431 non-null    object 
 14  Labored_Breathing   431 non-null    object 
 15  Lameness            431 non-null    object 
 16  Skin_Lesions        431 non-null    object 
 17  Nasal_Discharge     431 non-null    object 
 18  Eye_Discharge       431 non-null    object 
 19  Body_Temperature    431 non-null    object 
 20  Heart_Rate          431 non-null    int64  
 21  Disease_Prediction  431 non-null    object 
dtypes: float64(1), int64(2), object(19)
memory usage: 74.2+ KB
```

Chi-Square Test of Independence

A Chi-Square Test of Independence was conducted to assess the relationship between categorical features in the dataset. The results are as follows:

- Chi-Square Statistic: 1074.88
- P-value: 1.58e-160
- Degrees of Freedom: 105

The p-value is extremely small, much lower than the 0.05 threshold, indicating a significant association between the categorical variables. This suggests that the features are interdependent and relevant for the prediction task.

```
# Chi-square contigency of dataset
chi2, p_value, dof, expected = chi2_contingency(contingency_table.values)

print("Chi-square statistic:", chi2)
print("p-value:", p_value)
print("Degrees of freedom:", dof)
print("Expected frequencies:\n", expected)
```

```
Chi-square statistic: 1074.8845297668736
p-value: 1.5846329638707856e-160
Degrees of freedom: 105
Expected frequencies:
 [[ 1.33642691 23.55452436  0.16705336  2.00464037  1.16937355  2.67285383
   0.50116009  9.18793503  8.68677494  0.33410673  3.84222738  0.16705336
   5.51276102  1.33642691 10.35730858  1.16937355]
 [ 1.26218097 22.24593968  0.15777262  1.89327146  1.10440835  2.52436195
   0.47331787  8.6774942   8.20417633  0.31554524  3.6287703   0.15777262
   5.20649652  1.26218097  9.78190255  1.10440835]
 [ 1.39211137 24.53596288  0.17401392  2.08816705  1.21809745  2.78422274
   0.52204176  9.57076566  9.0487239   0.34802784  4.00232019  0.17401392
   5.7424594   1.39211137 10.78886311  1.21809745]
 [ 0.72389791 12.7587007   0.09048724  1.08584687  0.63341067  1.44779582
   0.27146172  4.97679814  4.70533643  0.18097448  2.0812065   0.09048724
   2.98607889  0.72389791  5.61020882  0.63341067]
 [ 1.225058   21.59164733  0.15313225  1.83758701  1.07192575  2.45011601
   0.45939675  8.42227378  7.96287703  0.3062645   3.52204176  0.15313225
   5.05336427  1.225058    9.49419954  1.07192575]
 [ 0.70533643 12.43155452  0.08816705  1.05800464  0.61716937  1.41067285
   0.26450116  4.84918794  4.58468677  0.17633411  2.02784223  0.08816705
   2.90951276  0.70533643  5.46635731  0.61716937]
 [ 0.63109049 11.12296984  0.07888631  0.94663573  0.55220418  1.26218097
   0.23665893  4.3387471   4.10208817  0.15777262  1.81438515  0.07888631
   2.60324826  0.63109049  4.89095128  0.55220418]
 [ 0.72389791 12.7587007   0.09048724  1.08584687  0.63341067  1.44779582
   0.27146172  4.97679814  4.70533643  0.18097448  2.0812065   0.09048724
   2.98607889  0.72389791  5.61020882  0.63341067]]
```

Animal Type Distribution and Gender Distribution by Animal Type

We analyzed the distribution of animal types and the relationship between Gender and Animal Type.

- **Gender Distribution by Animal Type:** We also calculated the gender distribution within each animal type:
    - For Cats, 44.4% are female and 55.6% male.
    - For Dogs, 54.4% are female and 45.6% male.
    - The rest of the animal types show varying gender distributions, with most having a slight male dominance.

These distributions provide insights into the dataset's composition, which can inform subsequent analysis and model development.

```python
# Animal distribution
animal_dist = df['Animal_Type'].value_counts(normalize=True) * 100
print("Animal Distribution (%):")
print(animal_dist)

# Animal Type vs Gender
gender_dist = pd.crosstab(df['Animal_Type'], df['Gender'], normalize='index') * 100
print("\nGender Distribution by Animal Type (%):")
print(gender_dist)
```

```
Animal Distribution (%):
Animal_Type
Dog       17.401392
Cat       16.705336
Cow       15.777262
Horse     15.313225
Sheep      9.048724
Goat       9.048724
Pig        8.816705
Rabbit     7.888631
Name: proportion, dtype: float64

Gender Distribution by Animal Type (%):
Gender          Female       Male
Animal_Type
Cat          44.444444  55.555556
Cow          54.411765  45.588235
Dog          46.666667  53.333333
Goat         64.102564  35.897436
Horse        45.454545  54.545455
Pig          47.368421  52.631579
Rabbit       47.058824  52.941176
Sheep        33.333333  66.666667
```

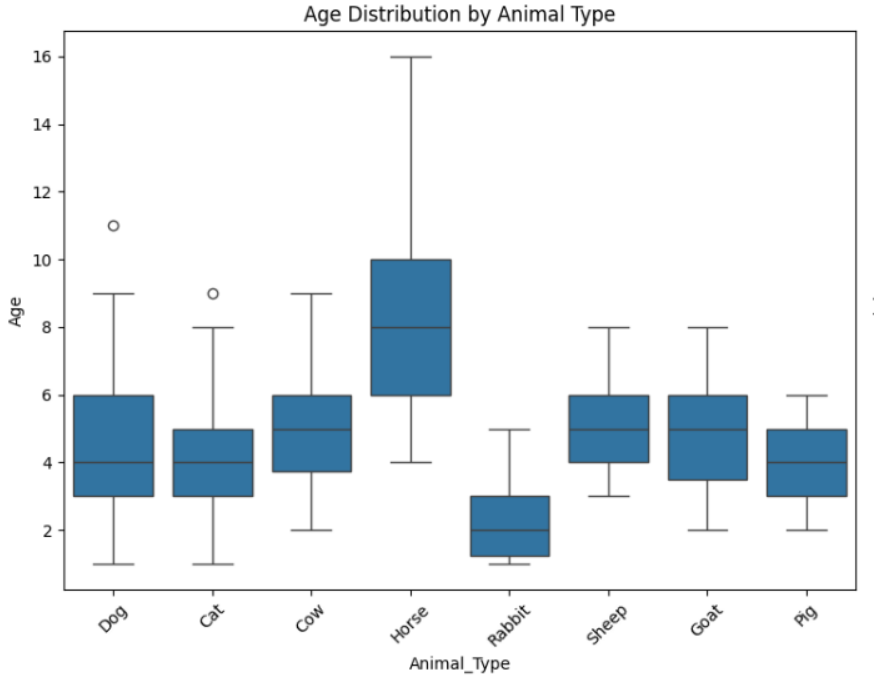
Age and Weight Distributions by Animal Type

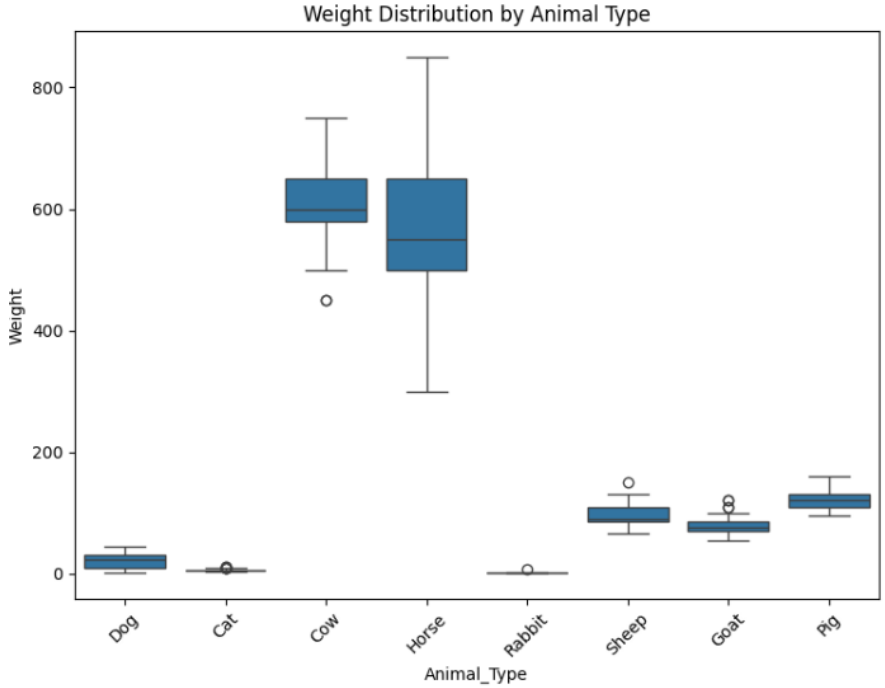
Boxplots were generated to visualize the Age and Weight distributions for each Animal Type.

- **Age Distribution**: A boxplot was created to show how the ages of different animal types vary. The plot helps in identifying potential outliers and the general age range for each animal category.
- **Weight Distribution**: A similar boxplot was plotted for Weight, highlighting the spread and central tendency of weights for each animal type.

These visualizations provide a clear view of the data's spread and potential differences between animal types, essential for understanding the dataset before model training.

```
# Age and weight distributions by animal
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

sns.boxplot(x='Animal_Type', y='Age', data=df, ax=ax1)
ax1.set_title('Age Distribution by Animal Type')
ax1.tick_params(axis='x', rotation=45)

sns.boxplot(x='Animal_Type', y='Weight', data=df, ax=ax2)
ax2.set_title('Weight Distribution by Animal Type')
ax2.tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()
```

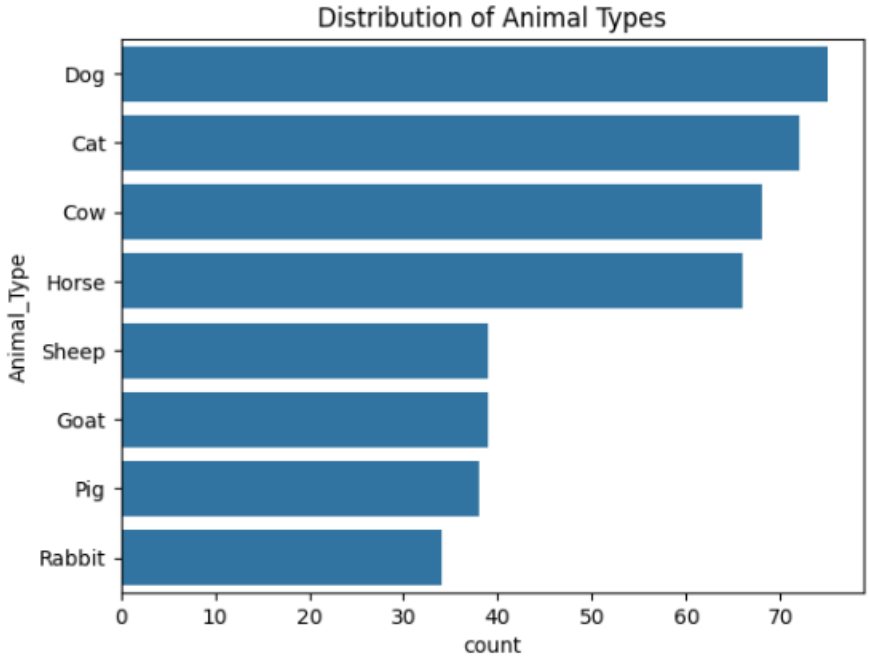




Distribution of Animal Types

The code uses the sns.countplot() function to create a horizontal bar plot of the Animal_Type column, displaying the frequency of each animal type. The order argument sorts the animal types in descending order based on their count. The plot shows that Dog has the highest count, followed by Cat and Cow, while Rabbit has the lowest count. This visualization helps to quickly assess the distribution of animal types in the dataset.

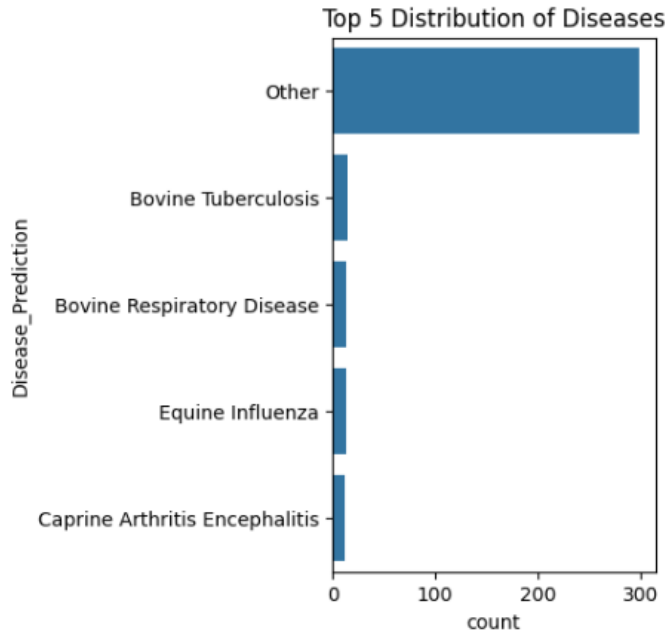
```
# Distribution of animals
sns.countplot(data=df, y='Animal_Type', order=df['Animal_Type'].value_counts().index)
plt.title('Distribution of Animal Types')
plt.show()
```



Top 5 Distribution of Diseases

The code generates a horizontal bar plot to visualize the distribution of the Top 5 most frequent diseases in the dataset. It first sets the figure size with plt.figure(figsize=(3, 5)), then uses sns.countplot() to plot the count of each disease in the Disease_Prediction column, ordered by the most frequent occurrences. The order=df['Disease_Prediction'].value_counts().index[:5] argument ensures that only the top 5 diseases are displayed. Finally, the title "Top 5 Distribution of Diseases" is added with plt.title(). The output reveals that "Other" disease has the highest count, followed by Bovine Tuberculosis, Bovine Respiratory Disease, Equine Influenza, and Caprine Arthritis Encephalitis, providing a clear view of the most common diseases in the dataset.

```
# Top 5 diseases
plt.figure(figsize=(3, 5))
sns.countplot(data=df, y='Disease_Prediction', order=df['Disease_Prediction'].value_counts().index[:5])
plt.title('Top 5 Distribution of Diseases')
plt.show()
```



Distribution of Numerical Variables

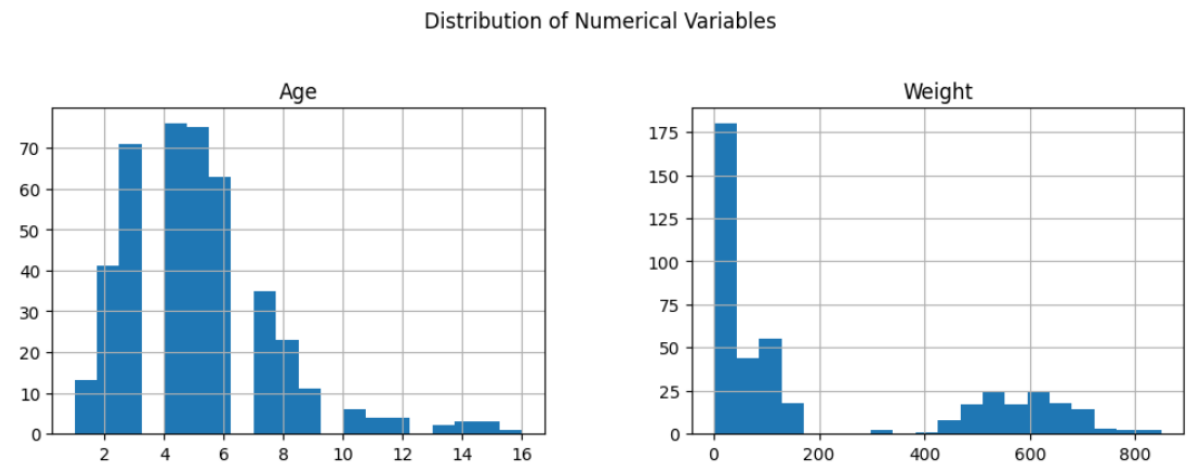
The code generates histograms to visualize the distribution of the numerical variables in the dataset, specifically Age, Weight, Heart Rate, and Body Temperature. The df[['Age', 'Weight', 'Heart_Rate', 'Body_Temperature']].hist() function is used to plot these variables, with 20 bins specified for the histograms and a figure size of (12, 8). The histograms display the frequency distribution for each variable.

In the output:

- Age shows a somewhat normal distribution with a peak around 4-5 years.
- Weight shows a heavily skewed distribution with a high frequency of low-weight animals and a long tail on the higher end.

These visualizations provide insight into the spread and skewness of the numerical features in the dataset, which is important for further analysis and model training.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Distribution of numerical columns
df[['Age', 'Weight', 'Heart_Rate', 'Body_Temperature']].hist(bins=20, figsize=(12, 8))
plt.suptitle('Distribution of Numerical Variables')
plt.show()
```

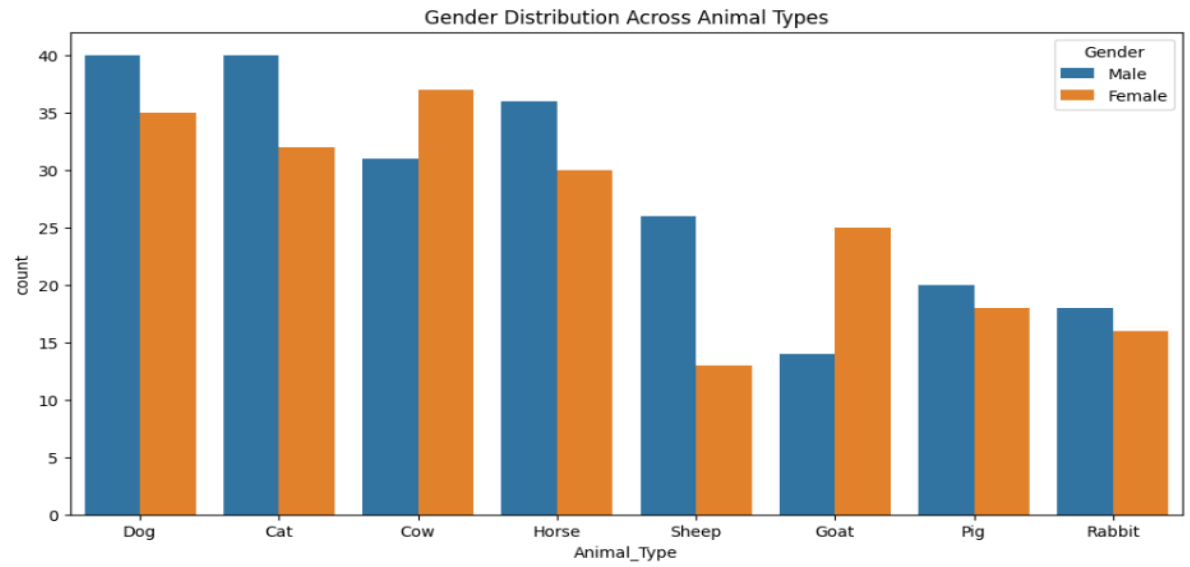


Gender Distribution Across Animal Types

The code creates a grouped count plot to visualize the distribution of gender across different animal types in the dataset. Using sns.countplot(), the plot shows the count of Male and Female animals for each Animal_Type. The hue='Gender' argument colors the bars by gender, while the order=df['Animal_Type'].value_counts().index sorts the animal types based on their frequency.

The plot shows that for most animal types, there are nearly equal numbers of Male and Female animals, but for some, like Horse and Dog, the gender distribution is more skewed, with one gender being more frequent.

```
plt.figure(figsize=(12, 6))
sns.countplot(data=df, x='Animal_Type', hue='Gender', order=df['Animal_Type'].value_counts().index)
plt.title('Gender Distribution Across Animal Types')
plt.show()
```

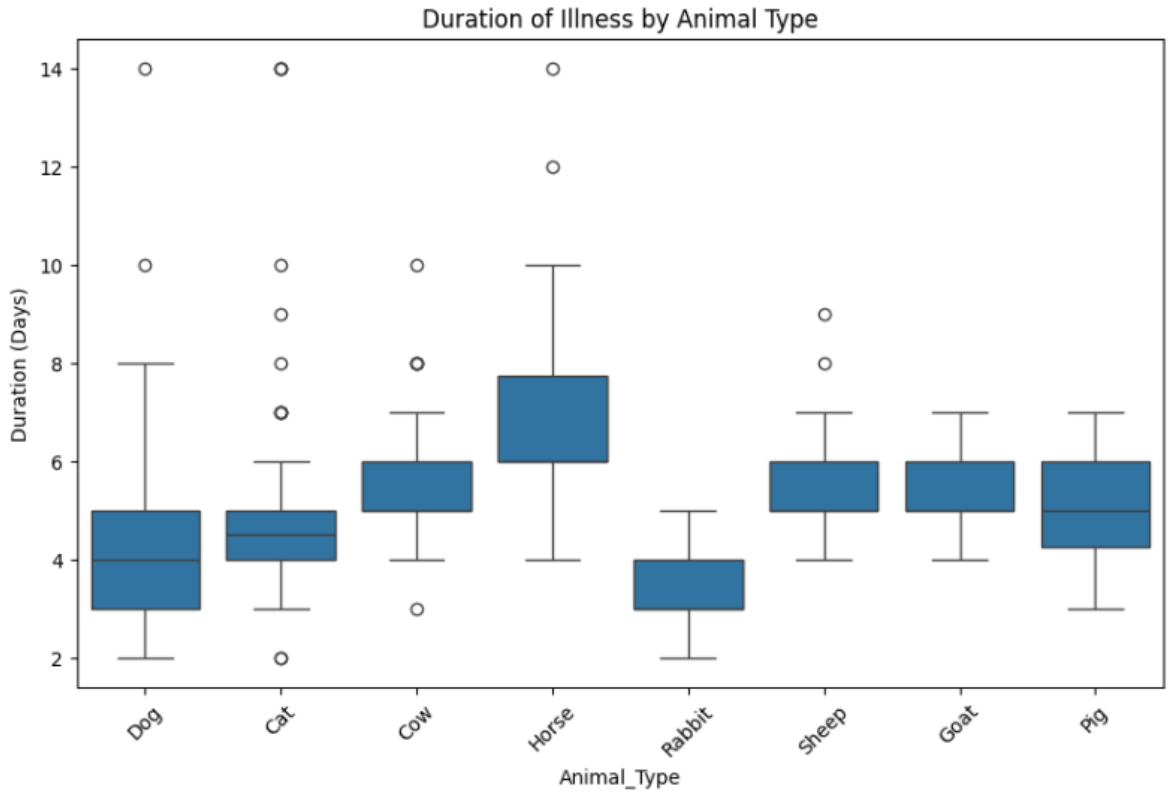


Duration of Illness by Animal Type

This code first converts the Duration column, which contains the duration in weeks or days, into a numeric format for further analysis. The conversion process is done by extracting the numeric part using str.extract() and converting it into a float type. Additionally, if the duration is provided in weeks, it is multiplied by 7 to convert it into days.

After the data is processed, a boxplot is created using sns.boxplot() to show the duration of illness (in days) for each animal type. The plot provides a visual representation of the distribution of illness duration, highlighting the median, interquartile range (IQR), and outliers for each animal type.

```
# Convert duration to numeric
df['Duration_Num'] = df['Duration'].str.extract('(\d+)').astype(float)
df.loc[df['Duration'].str.contains('week'), 'Duration_Num'] *= 7

plt.figure(figsize=(10, 6))
sns.boxplot(x='Animal_Type', y='Duration_Num', data=df)
plt.title('Duration of Illness by Animal Type')
plt.ylabel('Duration (Days)')
plt.xticks(rotation=45)
plt.show()
```



Data Preprocessing and Feature Engineering

This code segment performs several preprocessing tasks:

1. **Class Frequency**: Rare disease classes (less than 2% of data) are grouped under "Other" to balance the dataset.
2. **Binary Flags for Symptoms**: Converts symptoms (e.g., Appetite_Loss, Vomiting) to binary values (1 for "Yes", 0 for others).
3. **Temperature Conversion**: Cleans and converts Body Temperature values by removing '°C' and changing them to float.
4. **Duration Conversion**: Converts Duration from weeks to days.
5. **Age Grouping**: The Age variable is categorized into groups (e.g., Infant, Young, Adult) using pd.cut().

```
# Compute class frequencies
target_col = 'Disease_Prediction'
value_counts = df[target_col].value_counts(normalize=True)

# Group rare classes
rare_classes = value_counts[value_counts < 0.02].index

# Replace rare classes with 'Other'
df[target_col] = df[target_col].replace(rare_classes, 'Other')
```

```
# Binary flags for symptoms
symptom_cols = ['Appetite_Loss', 'Vomiting', 'Diarrhea', 'Coughing',
                'Labored_Breathing', 'Lameness', 'Skin_Lesions',
                'Nasal_Discharge', 'Eye_Discharge']
for col in symptom_cols:
    df[col] = (df[col] == 'Yes').astype(int)

# Temperature conversion
df['Body_Temperature'] = df['Body_Temperature'].astype(str).str.replace('°C', '', regex=False).astype(float)


# Duration conversion
df['Duration_Num'] = df['Duration'].str.extract('(\d+)').astype(float)
df.loc[df['Duration'].str.contains('week'), 'Duration_Num'] *= 7

# Age groups
df['Age_Group'] = pd.cut(df['Age'],
                         bins=[0, 1, 3, 5, 10, float('inf')],
                         labels=['Infant', 'Young', 'Adult', 'Middle_Aged', 'Senior'])
```

Weight Categorization, Train-Test Split, and Feature Scaling

Weight Categorization: The categorize_weight() function categorizes the weight of animals into bins based on their type (Dog or Cat). Dogs have weight bins like [0, 10, 25, 40, ∞], while Cats have bins [0, 4, 6, 8, ∞]. The categorized values are stored in a new column, Weight_Category.

Train-Test Split: The dataset is split into features (X) and target (y), and the target column is dropped from X. The train_test_split() function then splits the data into 80% training and 20% testing, ensuring the split is reproducible using a fixed random state and maintaining the same proportion of target classes in both sets (stratified split).

Feature Scaling: The StandardScaler is applied to scale numerical features such as Age, Weight, Body_Temperature, and Duration_Num. The scaler is fitted on the training set and then applied to both the training and test sets to ensure consistent scaling.

```
# Weight categories
def categorize_weight(row):
    if row['Animal_Type'] == 'Dog':
        bins = [0, 10, 25, 40, float('inf')]
        labels = ['Small', 'Medium', 'Large', 'Giant']
    elif row['Animal_Type'] == 'Cat':
        bins = [0, 4, 6, 8, float('inf')]
        labels = ['Small', 'Medium', 'Large', 'Giant']
    else:
        return 'NA'
    return pd.cut([row['Weight']], bins=bins, labels=labels)[0]

df['Weight_Category'] = df.apply(categorize_weight, axis=1)
```

```
# Train-test split
X = df.drop(columns=[target_col])
y = df[target_col]
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42,
                                                    stratify=y)
```

```
# Feature scaling
scaler = StandardScaler()
num_features = ['Age', 'Weight', 'Body_Temperature', 'Duration_Num']
X_train[num_features] = scaler.fit_transform(X_train[num_features])
X_test[num_features] = scaler.transform(X_test[num_features])
```

Encoding Categorical Features and Target Variable

Label Encoding: The categorical features (Animal_Type, Breed, Gender, etc.) are encoded into numerical values using LabelEncoder.

One-Hot Encoding: One-Hot Encoding is applied to object-type features, converting them into binary columns using pd.get_dummies().

Align Columns: The columns of X_train and X_test are aligned to ensure consistency, with missing columns in X_test filled with 0s.

Target Encoding: The target variable (y_train and y_test) is also label-encoded using LabelEncoder.

Class Weights: Class weights are computed to handle class imbalance, with less frequent classes assigned higher weights.

```
# Categorical features
cat_features = ['Animal_Type', 'Breed', 'Gender', 'Age_Group', 'Weight_Category']
encoders = {}

for feature in cat_features:
    X_train[feature] = X_train[feature].astype(str)
    X_test[feature] = X_test[feature].astype(str)
    all_unique_values = pd.concat([X_train[feature], X_test[feature]]).unique()

    enc = LabelEncoder().fit(all_unique_values)
    encoders[feature] = enc

    X_train[feature] = enc.transform(X_train[feature])
    X_test[feature] = X_test[feature].apply(
        lambda x: enc.transform([x])[0] if x in enc.classes_ else -1
    )

# One-hot encode
objs_train = X_train.select_dtypes(['object']).columns
X_train = pd.get_dummies(X_train, columns=objs_train)
X_test = pd.get_dummies(X_test, columns=objs_train)

# Align columns
X_train, X_test = X_train.align(X_test, axis=1, join='left', fill_value=0)
```

```
# Encode y_train and y_test
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_test_encoded = le.transform(y_test)

# Compute class weights
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train_encoded),
    y=y_train_encoded
)

# Store class weights
weights_dict = dict(zip(np.unique(y_train_encoded), class_weights))
```

Removing Constant Features and Dimensionality Reduction

1. **Remove Constant Features**: The code uses VarianceThreshold to remove features that have little to no variance (constant features). Features with variance below 0.01 are

removed using constant_filter.get_support() to identify the non-constant features. This step reduces unnecessary columns that don't contribute to the model's performance.

2. **Filter Train and Test Datasets**: After identifying the non-constant features, both the training and test datasets are filtered to retain only these features. The number of removed constant features is displayed using print(f"Removed {<non_constant_features.sum()} constant features.").
3. **Reduce Dimensions with SelectKBest**: The SelectKBest method is applied to reduce the feature space by selecting the top 20 features based on the f_classif statistical test. This helps improve model performance by keeping the most informative features.

```
# Remove constant features
constant_filter = VarianceThreshold(threshold=0.01)
constant_filter.fit(X_train)

# Get mask of non-constant features
non_constant_features = constant_filter.get_support()

# Filter train and test datasets
X_train = X_train.loc[:, non_constant_features]
X_test = X_test.loc[:, non_constant_features]

print(f"Removed {~non_constant_features.sum()} constant features.")
```

```
Removed -83 constant features.
```

```
# Reduce dimensions
selector = SelectKBest(f_classif, k=20)
X_train_fs = selector.fit_transform(X_train, y_train_encoded)
X_test_fs = selector.transform(X_test)
```

Hyperparameter Tuning with RandomizedSearchCV

1. **Scoring Metric**: The F1 macro score is used for model evaluation.
2. **Base Estimators**: RandomForestClassifier and XGBClassifier are initialized with random seeds.
3. **Hyperparameter Grids**: Defines possible values for Random Forest and XGBoost hyperparameters (e.g., n_estimators, max_depth).
4. **Randomized Search**: RandomizedSearchCV performs 8 iterations, using 3-fold cross-validation to find the best hyperparameters for both models.
5. **Fitting and Output**: Models are fitted on training data, and the best hyperparameters are printed.

```python
# Define scoring
f1_macro_scorer = make_scorer(f1_score, average='macro')

# Base estimators
rf_base = RandomForestClassifier(
    random_state=42
)

xgb_base = XGBClassifier(
    use_label_encoder=False,
    eval_metric='mlogloss',
    random_state=42
)

# Param grids
rf_param_dist = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5, 10],
    'class_weight': ['balanced']
}
xgb_param_dist = {
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 6, 10],
    'n_estimators': [50, 100, 200],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

# Random Search for RF and XGB
rf_random_search = RandomizedSearchCV(
    estimator=rf_base,
    param_distributions=rf_param_dist,
    n_iter=8,
    scoring=f1_macro_scorer,
    cv=3,
    n_jobs=-1,
    random_state=42
)
```

```python
xgb_random_search = RandomizedSearchCV(
    estimator=xgb_base,
    param_distributions=xgb_param_dist,
    n_iter=8,
    scoring=f1_macro_scorer,
    cv=3,
    n_jobs=-1,
    random_state=42
)

# Fit
rf_random_search.fit(X_train_fs, y_train_encoded)
xgb_random_search.fit(X_train_fs, y_train_encoded)

print("Best RF params:", rf_random_search.best_params_)
print("Best XGB params:", xgb_random_search.best_params_)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarning: [15:15:17] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
Best RF params: {'n_estimators': 50, 'min_samples_split': 5, 'max_depth': None, 'class_weight': 'balanced'}
Best XGB params: {'subsample': 1.0, 'n_estimators': 100, 'max_depth': 6, 'learning_rate': 0.2, 'colsample_bytree': 0.8}
```

Model Evaluation and Reporting

1. **Get Best Estimators**: The best models from RandomizedSearchCV for Random Forest and XGBoost are retrieved using best_estimator_.
2. **Model Evaluation**: For each model, predictions are made on the test data (X_test_fs), and accuracy is computed by comparing the predicted labels (y_pred) with the true labels (y_test_encoded).
3. **Classification Report**: A classification report is generated for each model, showing metrics like precision, recall, and F1-score, using classification_report().
4. **Output**: The accuracy for both Random Forest (73.56%) and XGBoost (75.86%) is printed, along with the classification report.

```
# Get best estimators
rf_best = rf_random_search.best_estimator_
xgb_best = xgb_random_search.best_estimator_

# Evaluate models
for model_name, model in [("Random Forest", rf_best), ("XGBoost", xgb_best)]:
    y_pred = model.predict(X_test_fs)
    accuracy = accuracy_score(y_test_encoded, y_pred)

    test_classes = np.unique(y_test_encoded)
    target_names = le.inverse_transform(test_classes)

    # Classification report
    report = classification_report(
        y_test_encoded,
        y_pred,
        labels=test_classes,
        target_names=target_names,
        zero_division=1
    )

    print(f"=== {model_name} ===")
    print(f"Accuracy: {accuracy:.4f}")
```

```
=== Random Forest ===
Accuracy: 0.7356
=== XGBoost ===
Accuracy: 0.7586
```

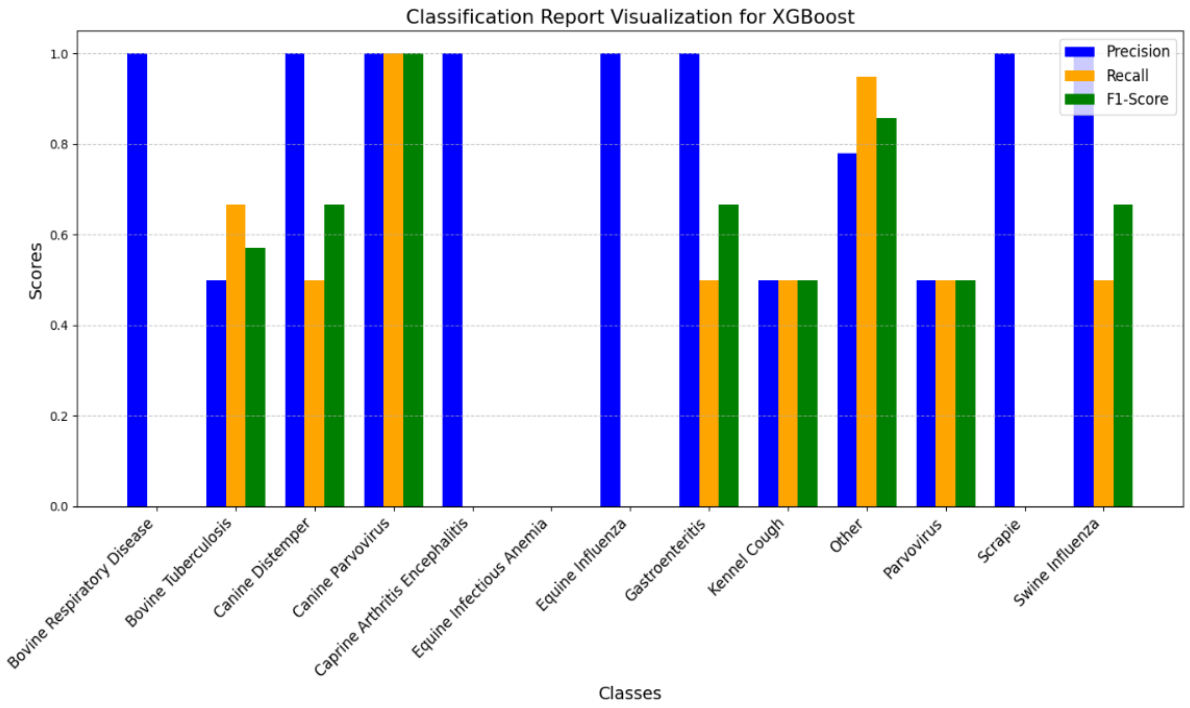
Visualization of Classification Report

1. **Generate Classification Report**: The classification_report() function is used to generate a detailed report, which includes precision, recall, and F1-score for each class. The output_dict=True argument stores the report in a dictionary (report_dict).

2. **Extract Precision, Recall, and F1-Score**: The code extracts the precision, recall, and F1-score for each class from report_dict and stores them in separate lists for plotting.
3. **Plot Categories**: The categories (classes) are extracted and stored in the list categories. The x-axis (x) represents the different classes, and bar width is set to 0.25.
4. **Plot the Bars**: Bar plots are created for precision, recall, and F1-score for each category using ax.bar(), with different colors assigned to each metric (blue, orange, green).
5. **Label the Plot**: The plot is labeled with axis labels for classes and scores, and a title indicating the model name. The x-tick labels are rotated for better readability.
6. **Show the Plot**: The plot layout is adjusted using plt.tight_layout(), and the plot is displayed with plt.show().

```
# Generate classification report
report_dict = classification_report(
    y_test_encoded,
    y_pred,
    labels=test_classes,
    target_names=target_names,
    zero_division=1,
    output_dict=True
)

# Visualize report
categories = list(report_dict.keys())[:-3]
precision = [report_dict[category]['precision'] for category in categories]
recall = [report_dict[category]['recall'] for category in categories]
f1_score = [report_dict[category]['f1-score'] for category in categories]
```

```
# Plot categories
x = np.arange(len(categories))
bar_width = 0.25

# Plot the bars
fig, ax = plt.subplots(figsize=(14, 8))
ax.bar(x - bar_width, precision, bar_width, label='Precision', color='blue')
ax.bar(x, recall, bar_width, label='Recall', color='orange')
ax.bar(x + bar_width, f1_score, bar_width, label='F1-Score', color='green')

# Label the plot
ax.set_xlabel('Classes', fontsize=14)
ax.set_ylabel('Scores', fontsize=14)
ax.set_title(f'Classification Report Visualization for {model_name}', fontsize=16)
ax.set_xticks(x)
ax.set_xticklabels(categories, rotation=45, ha='right', fontsize=12)
ax.legend(fontsize=12)
ax.grid(axis='y', linestyle='--', alpha=0.7)

# Show the plot
plt.tight_layout()
plt.show()
```



Library and Package Imports

In this project, several essential Python libraries and modules were imported to support data preprocessing, machine learning, visualization, and user interface development. The pandas library was used to load and manipulate the dataset in a structured tabular format, while numpy provided support for numerical operations and array handling. For preparing the data for machine learning algorithms, the StandardScaler module from sklearn.preprocessing was used to standardize features by removing the mean and scaling to unit variance. This is crucial for algorithms sensitive to feature magnitudes.

To enable data visualization, particularly for plotting graphs and model behavior, the matplotlib.pyplot library was utilized, and mpl_toolkits.mplot3d.Axes3D was specifically imported to create 3D plots when necessary. The math module provided additional mathematical functions needed for data transformations. For building a graphical user interface (GUI), the tkinter library was imported, allowing the creation of an interactive front end where users can input symptoms and receive predicted diseases. Finally, the os library was used for interacting with the operating system, such as handling file paths and directory management. Together, these libraries formed the foundation for implementing and deploying the animal disease prediction system effectively.

```
#Importing libraries and packages
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import math
from tkinter import *
import numpy as np
import pandas as pd
import os
```

Symptom and Disease List Definition

In this section of the code, two important lists are defined: one for symptoms and another for cattle diseases. These lists form the foundation of the cattle disease prediction model.

- This list l1 contains 94 unique symptoms that may be observed in cattle.
- These symptoms represent the input features that can be selected or entered by the user during prediction.
- They cover a wide range of physical, behavioral, and clinical signs commonly associated with cattle diseases.
- This list is used later in the project to build input vectors for ML models (e.g., converting selected symptoms into binary format: 1 if symptom is present, 0 if not).
- This list disease includes 26 cattle-specific diseases that are being predicted by the model.
- Each disease corresponds to a target label in the classification problem.
- The machine learning algorithms are trained to map symptom patterns from l1 to one of these disease labels.
- The model outputs a prediction by identifying which disease is most likely associated with the given symptom set.

```
#List of the all symptoms is listed here in list l1.

l1=['anorexia','abdominal_pain','anaemia','abortions','acetone','aggression','arthrogyposis',
    'ankylosis','anxiety','bellowing','blood_loss','blood_poisoning','blisters','colic','Condemnation_of_livers',
    'coughing','depression','discomfort','dyspnea','dysentery','diarrhoea','dehydration','drooling',
    'dull','decreased_fertility','diffculty_breath','emaciation','encephalitis','fever','facial_paralysis','frothing_of_mouth',
    'frothing','gaseous_stomach','highly_diarrhoea','high_pulse_rate','high_temp','high_proportion','hyperaemia','hydrocephalus',
    'isolation_from_herd','infertility','intermittent_fever','jaundice','ketosis','loss_of_appetite','lameness',
    'lack_of-coordination','lethargy','lacrimation','milk_flakes','milk_watery','milk_clots',
    'mild_diarrhoea','moaning','mucosal_lesions','milk_fever','nausea','nasel_discharges','oedema',
    'pain','painful_tongue','pneumonia','photo_sensitization','quivering_lips','reduction_milk_vields','rapid_breathing',
    'rumenstasis','reduced_rumination','reduced_fertility','reduced_fat','reduces_feed_intake','raised_breathing','stomach_pain',
    'salivation','stillbirths','shallow_breathing','swollen_pharyngeal','swelling','saliva','swollen_tongue',
    'tachycardia','torticollis','udder_swelling','udder_heat','udder_hardeness','udder_redness','udder_pain','unwillingness_to_move',
    'ulcers','vomiting','weight_loss','weakness']
```

```
#List of Diseases (26 Cattle Diseases Mention) is listed in list disease.

disease=['mastitis','blackleg','bloat','coccidiosis','cryptosporidiosis',
       'displaced_abomasum','gut_worms','listeriosis','liver_fluke','necrotic_enteritis','peri_weaning_diarrhoea',
       'rift_valley_fever','rumen_acidosis',
       'traumatic_reticulitis','calf_diphtheria','foot_rot','foot_and_mouth','ragwort_poisoning','wooden_tongue','infectious_bovine_rhinotracheitis',
'acetonaemia','fatty_liver_syndrome','calf_pneumonia','schmallen_berg_virus','trypanosomosis','fog_fever']

print(disease)
```

Initialization of Symptom Vector

- This code creates a list l2 filled with zeros.
- The length of l2 is the same as l1, which contains all 94 symptoms.
- So, l2 becomes a zero-initialized vector of length 94.

```
l2=[]
for i in range(0,len(l1)):
    l2.append(0)
print(l2)
```

Dataset Reading and Preprocessing

Two copies of the dataset training.csv are loaded using Pandas:

- df: Used for model training and preprocessing.
- DF: Same data, but the prognosis column is set as the index. This is helpful for viewing class-specific data conveniently.

This code converts the categorical disease labels (stored in the prognosis column) into numerical labels (0–25). This transformation is necessary because most machine learning models expect numerical inputs and outputs.

```
#Reading the Cattle training Dataset .csv file
df=pd.read_csv("training.csv")
DF= pd.read_csv('training.csv', index_col='prognosis')
#Replace the values in the imported file by pandas by the inbuilt function replace in pandas.

df.replace({'prognosis':{'mastitis':0,'blackleg':1,'bloat':2,'coccidiosis':3,'cryptosporidiosis':4,
'displaced_abomasum':5,'gut_worms':6,'listeriosis':7,'liver_fluke':8,'necrotic_enteritis':9,
'peri_weaning_diarrhoea':10,
'rift_valley_fever':11,'rumen_acidosis':12,
'traumatic_reticulitis':13,'calf_diphtheria':14,'foot_rot':15,'foot_and_mouth':16,
'ragwort_poisoning':17,'wooden_tongue':18,'infectious_bovine_rhinotracheitis':19,
'acetonaemia':20,'fatty_liver_syndrome':21,'calf_pneumonia':22,'schmallen_berg_virus':23,
'trypanosomosis':24,'fog_fever':25}},inplace=True)
df.head()
df = df.infer_objects(copy=False)
DF.head()
DF.info()
```

The dataset has 2044 rows (samples) and 93 columns (symptoms). Each column represents a symptom with binary values. 1 if the symptom is present, and 0 if absent. The index is set to the disease name (in DF only), making it easier to inspect data by disease class. All features are of type int64, which confirms they are numeric and suitable for classification models.

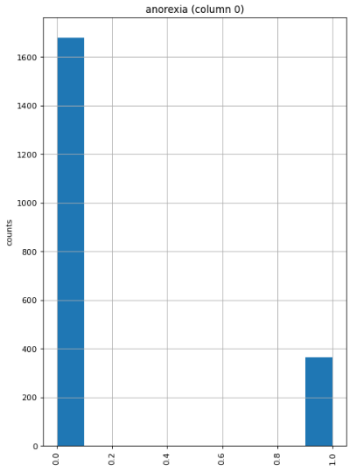
```
Index: 2044 entries, mastitis to traumatic_reticulitis
Data columns (total 93 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   anorexia                2044 non-null   int64
 1   abdominal_pain          2044 non-null   int64
 2   anaemia                 2044 non-null   int64
 3   abortions               2044 non-null   int64
 4   acetone                 2044 non-null   int64
 5   aggression              2044 non-null   int64
 6   arthrogyposis           2044 non-null   int64
 7   ankylosis               2044 non-null   int64
 8   anxiety                 2044 non-null   int64
 9   bellowing               2044 non-null   int64
 10  blood_loss              2044 non-null   int64
 11  blood_poisoning         2044 non-null   int64
 12  blisters                2044 non-null   int64
 13  colic                   2044 non-null   int64
 14  Condemnation_of_livers  2044 non-null   int64
 15  conjunctivae            2044 non-null   int64
 16  coughing                2044 non-null   int64
 17  depression              2044 non-null   int64
 18  discomfort              2044 non-null   int64
 19  dyspnea                 2044 non-null   int64
...
 91  weight_loss             2044 non-null   int64
 92  weakness                2044 non-null   int64
dtypes: int64(93)
memory usage: 1.5+ MB
```

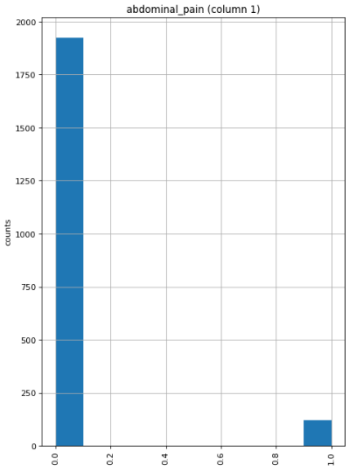
Column-Wise Distribution Visualization

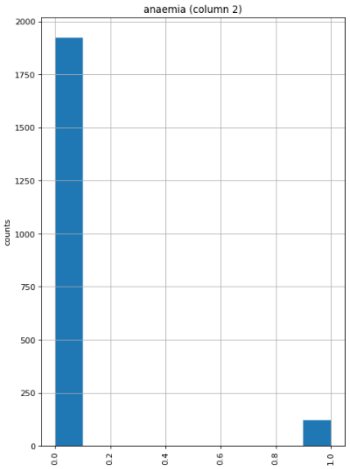
The plotPerColumnDistribution function is used to visualize the distribution of values (primarily symptoms) across the dataset. It selects columns that have more than one unique value and fewer than 50 unique values—typically binary or categorical features—and then generates a set of bar charts or histograms to display how often each value occurs. This is helpful for understanding the frequency and balance of symptoms in the dataset. The function takes three parameters: the dataset, the number of graphs to display, and the number of graphs per row for layout. This type of visualization supports exploratory data analysis (EDA) and

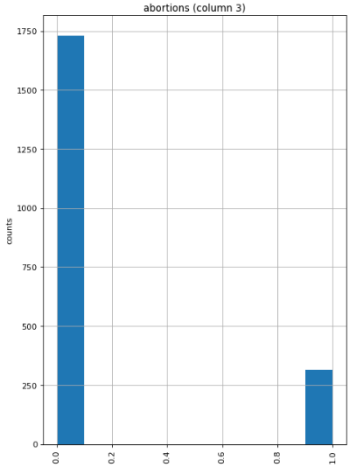
helps identify symptoms that may be too rare or overly common, both of which can impact model performance.

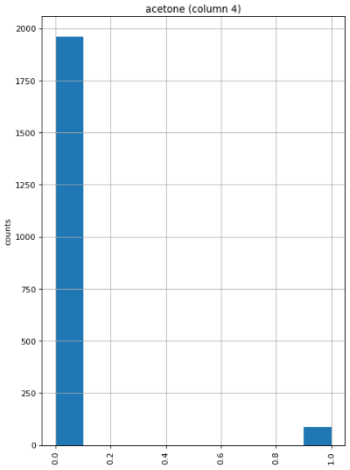
```
#·Distribution·graphs·(histogram/bar·graph)·of·column·data
def·plotPerColumnDistribution(df1,·nGraphShown,·nGraphPerRow):
····nunique·=·df1.nunique()
····df1·=·df1[[col·for·col·in·df1·if·nunique[col]·>·1·and·nunique[col]·<·50]]··#·Fix·here
····nRow,·nCol·=·df1.shape
····columnNames·=·list(df1)
····nGraphRow·=·math.ceil((nCol·+·nGraphPerRow·-·1)·/·nGraphPerRow)··#·Fix·here
····plt.figure(num=None,·figsize=(6·*·nGraphPerRow,·8·*·nGraphRow),·dpi=80,·facecolor='w',·edgecolor='k')

····for·i·in·range(min(nCol,·nGraphShown)):
········plt.subplot(nGraphRow,·nGraphPerRow,·i·+·1)
········columnDf·=·df1.iloc[:,·i]··#·Fix·here
········if·not·np.issubdtype(type(columnDf.iloc[0]),·np.number):
············valueCounts·=·columnDf.value_counts()
············valueCounts.plot.bar()
········else:
············columnDf.hist()
········plt.ylabel('counts')
········plt.xticks(rotation=90)
········plt.title(f'{columnNames[i]}·(column·{i})')

····plt.tight_layout(pad=1.0,·w_pad=1.0,·h_pad=1.0)
····plt.show()
plotPerColumnDistribution(df, 10, 3)
```

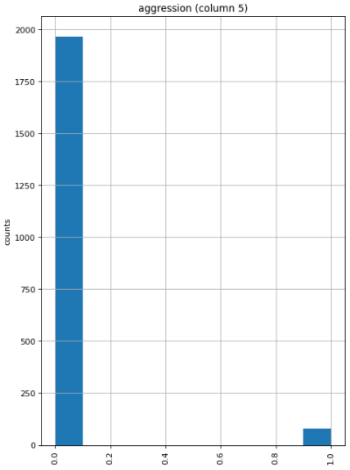

anorexia (column 0)
counts

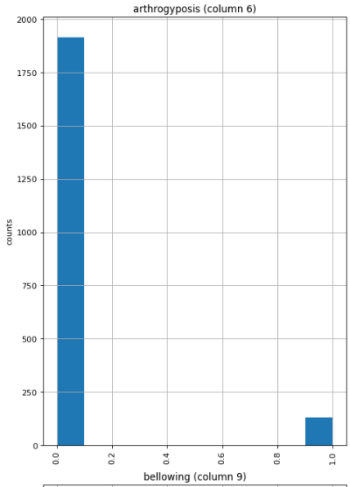

abdominal_pain (column 1)
counts

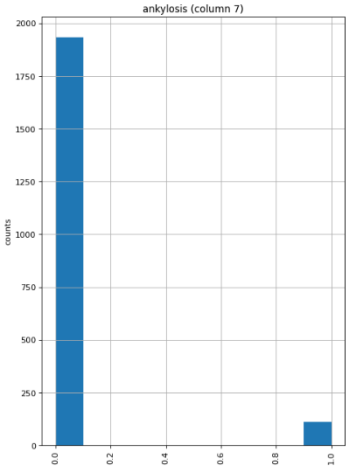

anaemia (column 2)
counts

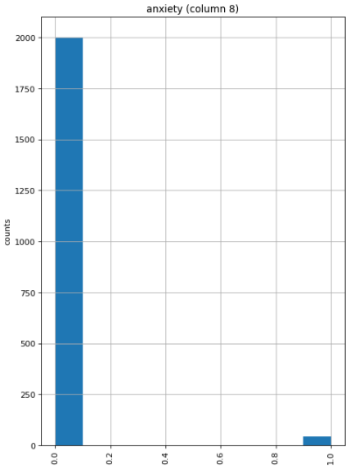

abortions (column 3)
counts

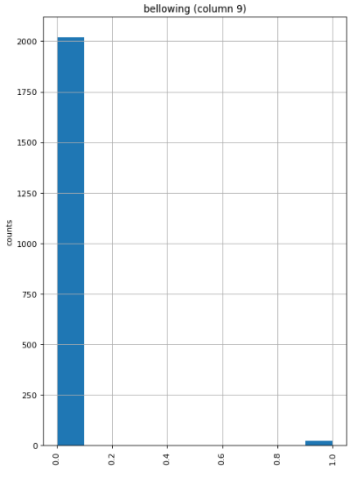

acetone (column 4)
counts


aggression (column 5)
counts


arthrogyposis (column 6)
counts


ankylosis (column 7)
counts

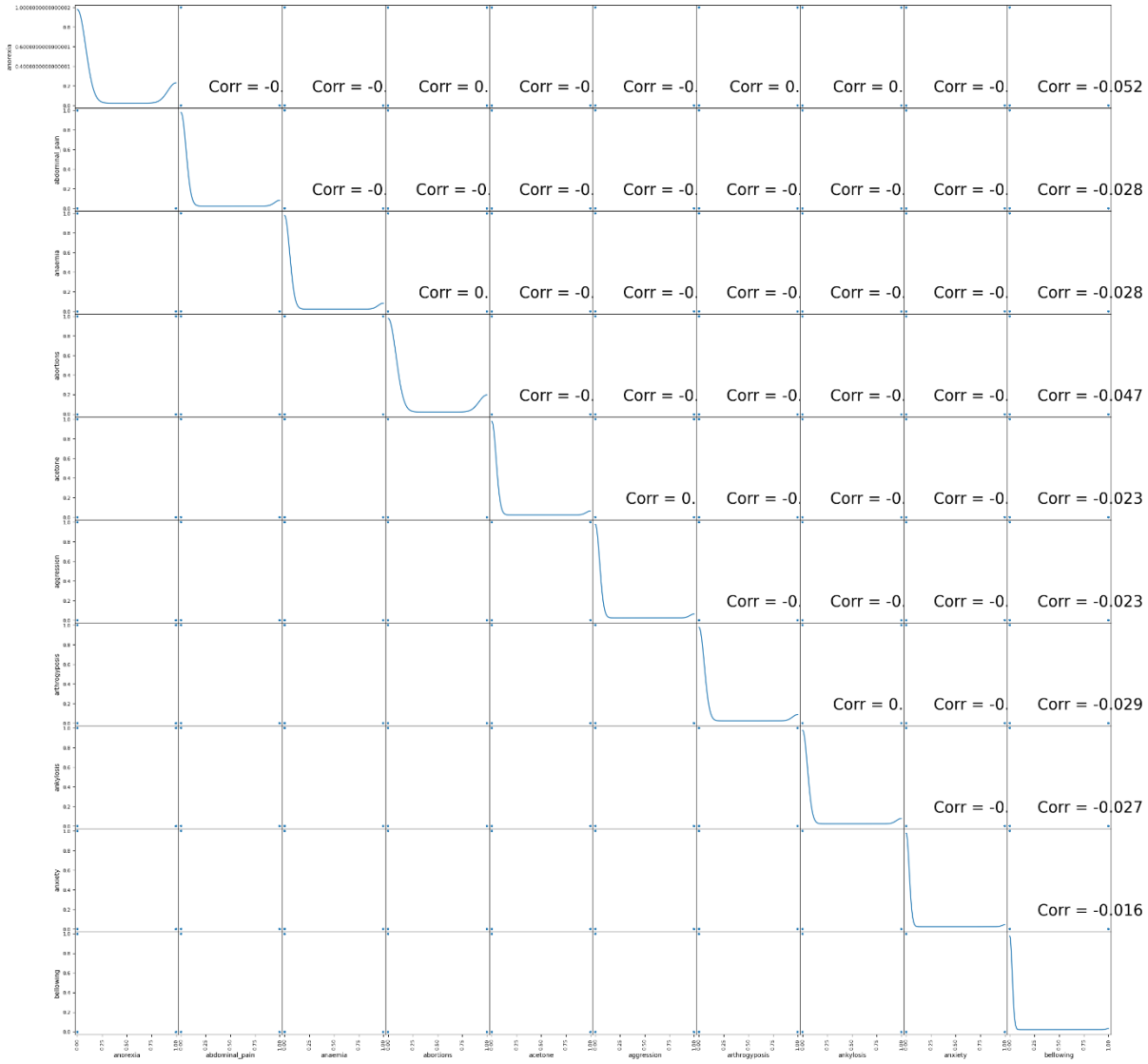

anxiety (column 8)
counts


bellowing (column 9)
counts

Scatter Matrix and Correlation Analysis of Numerical Features

The plotScatterMatrix() function is designed to generate a matrix of scatter plots and kernel density plots (KDEs) to visualize relationships between numeric features in the dataset. It starts by selecting only numerical columns and removes any columns that contain missing values or have only one unique value, as they do not contribute meaningful variance for analysis. To prevent the plot from becoming overcrowded, it limits the number of columns to a maximum of 10. The function then uses pandas' built-in scatter_matrix() to create a grid of scatter plots for all pairs of features, with density plots on the diagonals. Additionally, it calculates the Pearson correlation coefficients between feature pairs and displays them as annotations on the upper triangle of the scatter matrix. This helps in visually identifying strong or weak correlations between symptoms, which is valuable for understanding feature relationships and potential redundancies in the dataset before applying machine learning models.

```
def plotScatterMatrix(df1, plotSize, textSize):
    df1 = df1.select_dtypes(include=[np.number])  # keep only numeric columns
    df1 = df1.dropna(axis='columns')  # Drop columns with NaN

    # Only keep columns with more than 1 unique value
    df1 = df1[[col for col in df1 if df1[col].nunique() > 1]]
    columnNames = list(df1)

    # Limit to 10 columns to avoid overplotting
    if len(columnNames) > 10:
        columnNames = columnNames[:10]

    df1 = df1[columnNames]
    ax = pd.plotting.scatter_matrix(df1, alpha=0.75, figsize=[plotSize, plotSize], diagonal='kde')

    corrs = df1.corr().values
    for i, j in zip(*np.triu_indices_from(ax, k=1)):
        ax[i, j].annotate(f'Corr = {corrs[i, j]:.3f}', (0.8, 0.2),
                          xycoords='axes fraction', ha='center', va='center', size=textSize)

    plt.suptitle('Scatter and Density Plot')
    plt.show()

plotScatterMatrix(df, 30, 25)
```



Feature and Target Variable Selection

In this portion of the code, the dataset is split into input features and the target variable to prepare for machine learning model training:

- X = df[l1] Selects all columns listed in l1 (the symptom list) from the DataFrame df. This creates the feature matrix X, where each row corresponds to a sample, and each column represents a symptom as a binary feature (0 or 1 indicating absence or presence).
- y = df[["prognosis"]] Extracts the target column prognosis (the disease label) into a separate DataFrame y. This column contains the numeric disease codes assigned earlier.

- y = np.ravel(y) Converts y from a 2D array (DataFrame) into a 1D array, which is required by most machine learning models in libraries like scikit-learn for the target variable.
- print(X) and print(y) Outputs the features and target arrays to the console for verification, allowing the user to check the data before proceeding to model training.

```
X=df[l1]
y=df[["prognosis"]]
y=np.ravel(y)
print(X)
print(y)
```

```
      anorexia  abdominal_pain  anaemia  abortions  acetone  aggression
0            0               0        0          0        0           0
1            0               0        0          0        0           0
2            0               0        0          0        0           0
3            0               0        0          0        0           0
4            0               0        0          0        0           0
...        ...             ...      ...        ...      ...         ...
2039         0               0        0          0        0           0
2040         0               0        0          0        0           0
2041         0               0        0          0        0           0
2042         0               0        0          0        0           0
2043         0               0        0          0        0           0

      arthrogyposis  ankylosis  anxiety  bellowing  ...  udder_swelling
0                 0          0        0          0  ...               0
1                 0          0        0          0  ...               0
2                 0          0        0          0  ...               1
3                 0          0        0          0  ...               1
4                 0          0        0          0  ...               1
...             ...        ...      ...        ...  ...             ...
2039              0          0        0          0  ...               0
2040              0          0        0          0  ...               0
2041              0          0        0          0  ...               0
2042              0          0        0          0  ...               0
2043              0          0        0          0  ...               0
...
2043                    0       0        0           0        0

[2044 rows x 92 columns]
[ 0  0  0 ... 12 12 13]
```

Reading and Preprocessing the Cattle Disease Testing Dataset

This portion of the code handles loading and preparing the testing dataset for model evaluation:

- tr = pd.read_csv("testing.csv"): This line reads the testing dataset from a CSV file named testing.csv into a pandas DataFrame tr. This dataset is assumed to contain the same structure as the training set, including symptom columns and a prognosis column with disease names.
- tr.replace({...}, inplace=True): This line uses the pandas .replace() function to map disease names (strings) in the prognosis column to their corresponding integer labels (0–25). This numeric encoding is essential for classification algorithms, as they require numerical target labels.
- tr = tr.infer_objects(copy=False): This converts the datatypes of the columns in the DataFrame to the most appropriate types (e.g., converting object types to int64 or float64) without making an unnecessary copy of the data. It ensures cleaner and more consistent numeric data types for machine learning.
- tr.head(): Displays the first 5 rows of the processed testing dataset to verify that the data has been correctly loaded and preprocessed.

	anorexia	abdominal_pain	anaemia	abortions	acetone	aggression	arthrogyposis	ankylosis	anxiety
0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

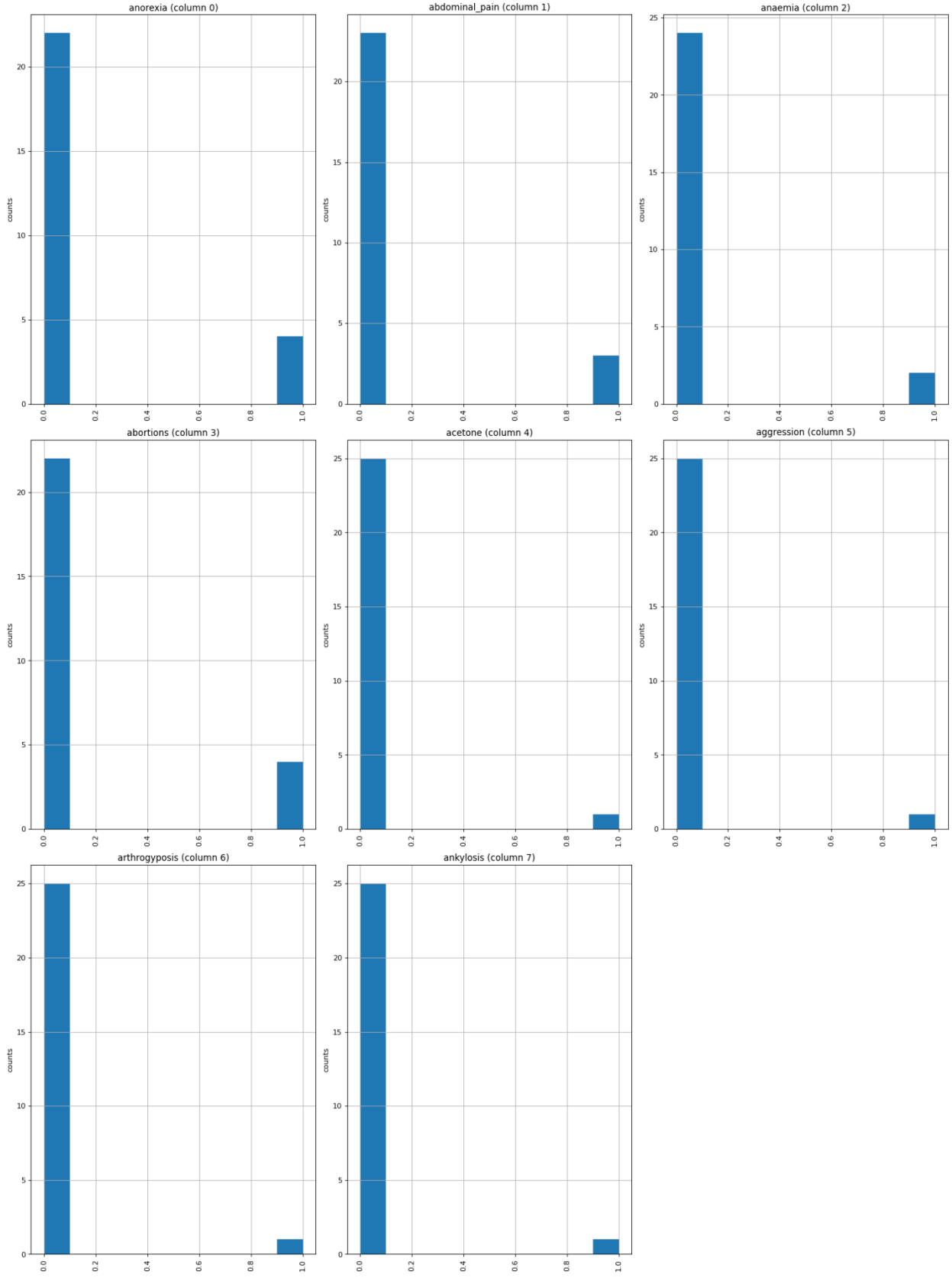
Visualizing the Testing Dataset

This section creates two types of visualizations to help understand the distribution and relationships of features in the cattle disease testing dataset:

plotPerColumnDistribution(tr, 10, 3)

- This function generates histograms or bar graphs for up to 10 columns from the testing dataset (tr) arranged in 3 graphs per row.
- It selects only those columns with a number of unique values between 2 and 49 to avoid plotting constant or high-cardinality columns.
- Histograms are used for numeric features, while bar charts are used for categorical or discrete features.
- These plots give a quick view of data distributions, frequency of symptoms, and help detect imbalances or anomalies in the testing data.

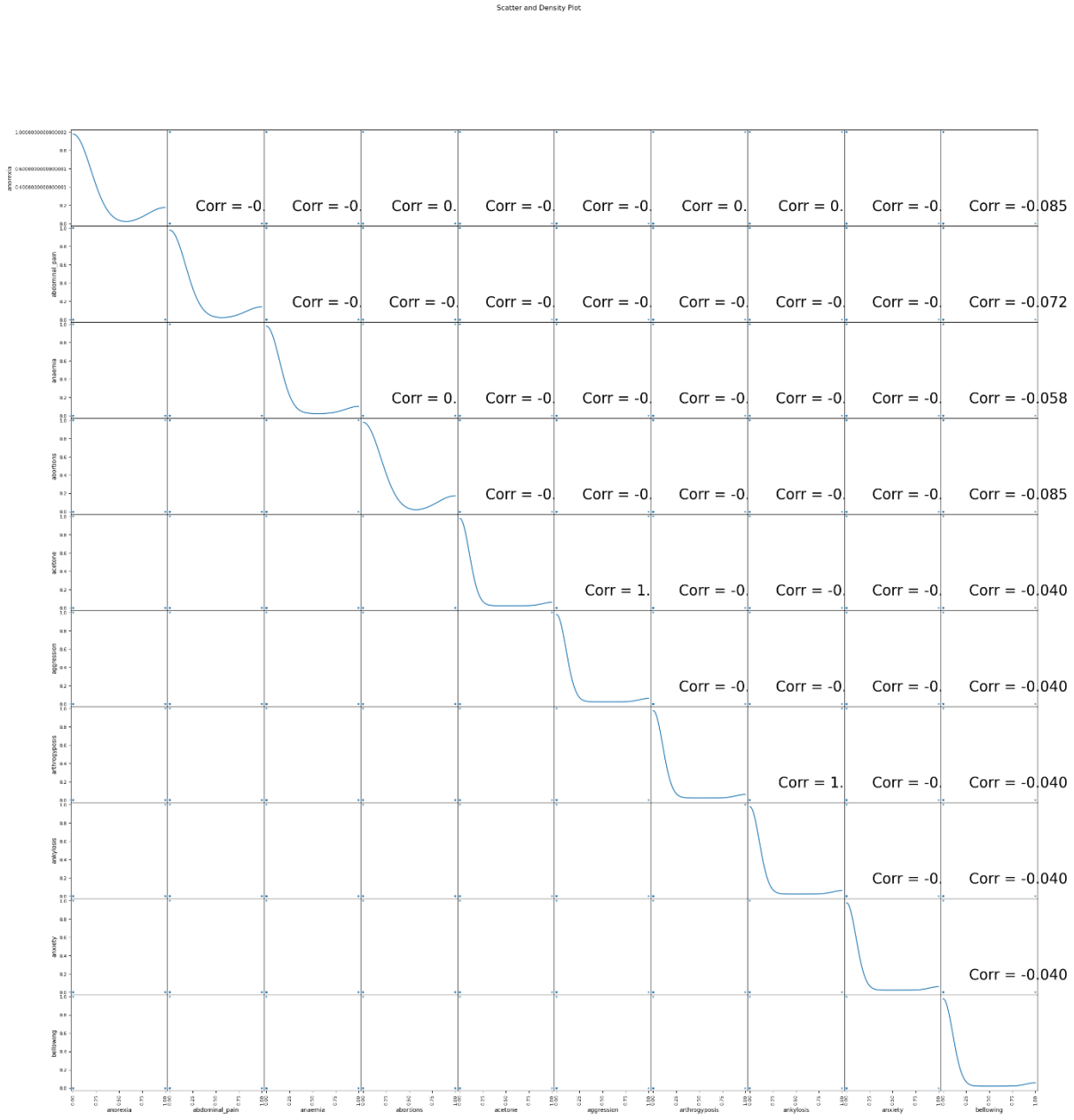
```
plotPerColumnDistribution(tr, 8, 3)
```



plotScatterMatrix(tr, 30, 25)

- This function creates a scatter matrix plot (also known as pair plot) of selected numeric columns (up to 10).
- It visualizes pairwise relationships between features using scatter plots and includes diagonal kernel density plots to show feature distributions.
- Additionally, it calculates and displays correlation coefficients between feature pairs on the plots.
- This helps understand feature interactions, clustering, and collinearity, which are important in evaluating how the features relate in the test dataset.

```
plotScatterMatrix(tr, 30, 25)
```



Splitting Testing Data into Features and Labels

In this section, the cattle disease testing dataset (tr) is prepared for model evaluation by separating the input features (symptoms) and the target output (disease label).

- X_test = tr[l1]: This line selects all the symptom columns listed in the previously defined list l1 from the testing dataset tr. It stores these columns into a new DataFrame X_test, where each row represents a single test sample and each column represents a binary symptom (0 = absent, 1 = present).
- y_test = tr[["prognosis"]]: This extracts the target column named "prognosis" from the testing data, which contains the integer-encoded disease labels for each sample.
- y_test = np.ravel(y_test): The np.ravel() function is used to flatten the 2D DataFrame into a 1D NumPy array so it can be passed directly to model evaluation functions like accuracy_score().
- print(X_test): Displays the binary symptom values (features) of the 26 test samples across 92 symptom columns.
- print(y_test): Displays the true class labels (integer values ranging from 0 to 25) corresponding to each test sample in X_test.

```
X_test= tr[l1]
y_test = tr[["prognosis"]]
y_test = np.ravel(y_test)
print(X_test)
print(y_test)
```

Output Summary

X_test is a 26 × 92 matrix where:

- Rows represent individual cattle cases in the test set.
- Columns represent presence (1) or absence (0) of each of the 92 symptoms.

y_test is a 1D array of length 26, where:

- Each number corresponds to the actual disease label (e.g., 0 = mastitis, 1 = blackleg, ..., 25 = fog_fever).

```
    anorexia  abdominal_pain  anaemia  abortions  acetone  aggression  \
0          0               0        0          0        0           0
1          1               0        0          0        0           0
2          0               0        0          0        0           0
3          0               0        0          0        0           0
4          0               0        0          0        0           0
5          0               0        0          0        0           0
6          0               1        0          0        0           0
7          0               0        0          1        0           0
8          0               0        1          0        0           0
9          0               0        0          0        0           0
10         0               0        0          0        0           0
11         1               0        0          1        0           0
12         0               0        0          0        0           0
13         0               1        0          0        0           0
14         0               0        0          0        0           0
15         1               0        0          0        0           0
16         0               0        0          0        0           0
17         0               1        0          0        0           0
18         0               0        0          0        0           0
19         0               0        0          1        0           0
20         0               0        0          0        1           1
21         0               0        0          0        0           0
22         0               0        0          0        0           0
23         1               0        0          0        0           0
...

[26 rows x 92 columns]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25]
```

Symptom Visualization Functions

This section defines two separate visualization functions used to explore symptom patterns for a specific disease and to visualize user-selected symptoms.

scatterplt(disea)

This function visualizes the distribution of symptoms for a specific disease label passed in the disea parameter.

- It filters the main dataset DF to include only the rows (patients) where prognosis == disea.
- It sums the values in all symptom columns (l1) to get the total number of times each symptom occurred for that specific disease.
- Any symptom that never appears (count = 0) is dropped to avoid clutter in the plot.
- A scatter plot is generated:

    - X-axis: names of symptoms that appeared at least once for that disease.
    - Y-axis: count of occurrences of each symptom.
- It helps in understanding which symptoms are dominant for a particular disease.

scatterinp(sym1, sym2, sym3, sym4, sym5)

This function is designed to visualize user-selected symptoms in binary form.

- Takes 5 symptom names (sym1 to sym5) as input.
- Creates a binary vector (y) where each entry is:
    - 1 if the symptom is selected (i.e., not "Select Here")
    - 0 otherwise.
- Plots a scatter graph where:
    - X-axis: the symptom names selected (or placeholders).
    - Y-axis: binary values showing whether the symptom was selected (1) or not (0).
- This helps visualize user choices for further processing or prediction.

```
#list1 = DF['prognosis'].unique()
def scatterplt(disea):
    x = DF[DF['prognosis'] == disea][l1].sum()
    x.drop(x[x == 0].index, inplace=True)
    print(x.values)
    y = x.keys()
    print(len(x))
    print(len(y))
    plt.title(disea)
    plt.scatter(y, x.values)
    plt.xticks(rotation=90)
    plt.ylabel("Count")
    plt.xlabel("Symptoms")
    plt.tight_layout()
    plt.show()


def scatterinp(sym1, sym2, sym3, sym4, sym5):
    x = [sym1, sym2, sym3, sym4, sym5]
    y = [0, 0, 0, 0, 0]

    for i, sym in enumerate(x):
        if sym != 'Select Here':
            y[i] = 1

    print("Symptoms:", x)
    print("Binary Vector:", y)

    plt.scatter(x, y)
    plt.title("Selected Symptoms")
    plt.xlabel("Symptom")
    plt.ylabel("Selected (1 = Yes)")
    plt.ylim(-0.5, 1.5)
    plt.tight_layout()
    plt.show()
```

DecisionTree() Function Overview

This function acts as the core backend logic for the Decision Tree-based disease prediction in your Tkinter app. It:

- Validates user input
- Trains and evaluates the model
- Predicts based on symptom input
- Logs the prediction to a local database
- Produces useful plots for user interaction and analysis

```
root = Tk()
pred1=StringVar()
def DecisionTree():
    if len(NameEn.get()) == 0:
        pred1.set(" ")
        comp=messagebox.askokcancel("System","Kindly Fill the Name")
        if comp:
            root.mainloop()
    elif((Symptom1.get()=="Select Here") or (Symptom2.get()=="Select Here")):
        pred1.set(" ")
        sym=messagebox.askokcancel("System","Kindly Fill atleast first two Symptoms")
        if sym:
            root.mainloop()
    else:
        from sklearn import tree

        clf3 = tree.DecisionTreeClassifier()
        clf3 = clf3.fit(X,y)

        from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
        y_pred=clf3.predict(X_test)
        print("Decision Tree")
        print("Accuracy")
        print(accuracy_score(y_test, y_pred))
        print(accuracy_score(y_test, y_pred,normalize=False))
        print("Confusion matrix")
        conf_matrix=confusion_matrix(y_test,y_pred)
        print(conf_matrix)

        psymptoms = [Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),Symptom5.get()]

        for k in range(0,len(l1)):
            for z in psymptoms:
                if(z==l1[k]):
                    l2[k]=1

        inputtest = [l2]
        predict = clf3.predict(inputtest)
        predicted=predict[0]
```

```
        h='no'
        for a in range(0,len(disease)):
            if(predicted == a):
                h='yes'
                break
```

```
    if (h=='yes'):
        pred1.set(" ")
        pred1.set(disease[a])
    else:
        pred1.set(" ")
        pred1.set("Not Found")
    #Creating the database if not exists named as database.db and creating table if not exists
    named as DecisionTree using sqlite3
    import sqlite3
    conn = sqlite3.connect('database.db')
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS DecisionTree(Name StringVar,Symtom1 StringVar,Symtom2
    StringVar,Symtom3 StringVar,Symtom4 TEXT,Symtom5 TEXT,Disease StringVar)")
    c.execute("INSERT INTO DecisionTree(Name,Symtom1,Symtom2,Symtom3,Symtom4,Symtom5,Disease) VALUES
    (?,?,?,?,?,?,?)",(NameEn.get(),Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),
    Symptom5.get(),pred1.get()))
    conn.commit()
    c.close()
    conn.close()

    #printing scatter plot of input symptoms
    #printing scatter plot of disease predicted vs its symptoms
    scatterinp(Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),Symptom5.get())
    scatterplt(pred1.get())
```

RandomForest() Function Overview

This function serves as the backend logic for Random Forest-based disease prediction in the Tkinter application. It:

- Validates user input by checking if the name and at least two symptoms are provided
- Trains a RandomForestClassifier model using the training dataset
- Evaluates model performance using accuracy score and confusion matrix
- Predicts the disease based on the selected symptoms provided by the user
- Stores the user's name, symptoms, and predicted disease in a local SQLite database
- Displays a scatter plot showing the symptoms related to the predicted disease for visualization

```
pred2=StringVar()
def randomforest():
    if len(NameEn.get())==0:
        pred1.set(" ")
        comp=messagebox.askokcancel("System","Kindly Fill the Name")
        if comp:
            root.mainloop()
    elif((Symptom1.get()=="Select Here") or (Symptom2.get()=="Select Here")):
        pred1.set(" ")
        sym=messagebox.askokcancel("System","Kindly Fill atleast first two Symptoms")
        if sym:
            root.mainloop()
    else:
        from sklearn.ensemble import RandomForestClassifier
        clf4 = RandomForestClassifier(n_estimators=100)
        clf4 = clf4.fit(X,np.ravel(y))

        # calculating accuracy
        from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
        y_pred=clf4.predict(X_test)
        print("Random Forest")
        print("Accuracy")
        print(accuracy_score(y_test, y_pred))
        print(accuracy_score(y_test, y_pred,normalize=False))
        print("Confusion matrix")
        conf_matrix=confusion_matrix(y_test,y_pred)
        print(conf_matrix)
```

```
    psymptoms = [Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),Symptom5.get()]

    for k in range(0,len(l1)):
        for z in psymptoms:
            if(z==l1[k]):
                l2[k]=1

    inputtest = [l2]
    predict = clf4.predict(inputtest)
    predicted=predict[0]

    h='no'
    for a in range(0,len(disease)):
        if(predicted == a):
            h='yes'
            break
    if (h=='yes'):
        pred2.set(" ")
        pred2.set(disease[a])
    else:
        pred2.set(" ")
    #Creating the database if not exists named as database.db and creating table if not exists
     named as RandomForest using sqlite3
    import sqlite3
    conn = sqlite3.connect('database.db')
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS RandomForest(Name StringVar,Symtom1 StringVar,Symtom2
    StringVar,Symtom3 StringVar,Symtom4 TEXT,Symtom5 TEXT,Disease StringVar)")
    c.execute("INSERT INTO RandomForest(Name,Symtom1,Symtom2,Symtom3,Symtom4,Symtom5,Disease) VALUES
    (?,?,?,?,?,?,?)",(NameEn.get(),Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),
    Symptom5.get(),pred2.get()))
    conn.commit()
    c.close()
    conn.close()
    #printing scatter plot of disease predicted vs its symptoms
    scatterplt(pred2.get())
```

KNN() Function Overview

This function serves as the backend logic for K-Nearest Neighbors (KNN)-based disease prediction in the Tkinter application. It:

- Validates user input by checking if the name and at least two symptoms are provided
- Trains a KNeighborsClassifier model using the training dataset
- Evaluates model performance using accuracy score and confusion matrix
- Predicts the disease based on the selected symptoms provided by the user
- Stores the user's name, symptoms, and predicted disease in a local SQLite database
- Displays a scatter plot showing the symptoms related to the predicted disease for visualization.

```
pred4=StringVar()
def KNN():
    if len(NameEn.get()) == 0:
        pred4.set(" ")
        comp=messagebox.askokcancel("System","Kindly Fill the Name")
        if comp:
            root.mainloop()
```

```
elif((Symptom1.get()=="Select Here") or (Symptom2.get()=="Select Here")):
    pred4.set(" ")
    sym=messagebox.askokcancel("System","Kindly Fill atleast first two Symptoms")
    if sym:
        root.mainloop()
else:
    from sklearn.neighbors import KNeighborsClassifier
    knn=KNeighborsClassifier(n_neighbors=5,metric='minkowski',p=2)
    knn=knn.fit(X,np.ravel(y))

    from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
    y_pred=knn.predict(X_test)
    print("kNearest Neighbour")
    print("Accuracy")
    print(accuracy_score(y_test, y_pred))
    print(accuracy_score(y_test, y_pred,normalize=False))
    print("Confusion matrix")
    conf_matrix=confusion_matrix(y_test,y_pred)
    print(conf_matrix)

    psymptoms = [Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),Symptom5.get()]

    for k in range(0,len(l1)):
        for z in psymptoms:
            if(z==l1[k]):
                l2[k]=1
    inputtest = [l2]
    predict = knn.predict(inputtest)
    predicted=predict[0]

    h='no'
    for a in range(0,len(disease)):
        if(predicted == a):
            h='yes'
            break


    if (h=='yes'):
        pred4.set(" ")
        pred4.set(disease[a])
    else:
        pred4.set(" ")
        pred4.set("Not Found")
     #Creating the database if not exists named as database.db and creating table if not exists
     named as KNearestNeighbour using sqlite3
    import sqlite3
    conn = sqlite3.connect('database.db')
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS KNearestNeighbour(Name StringVar,Symtom1 StringVar,
    Symtom2 StringVar,Symtom3 StringVar,Symtom4 TEXT,Symtom5 TEXT,Disease StringVar)")
    c.execute("INSERT INTO KNearestNeighbour(Name,Symtom1,Symtom2,Symtom3,Symtom4,Symtom5,Disease)
    VALUES(?,?,?,?,?,?,?)",(NameEn.get(),Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get
    (),Symptom5.get(),pred4.get()))
    conn.commit()
    c.close()
    conn.close()
    #printing scatter plot of disease predicted vs its symptoms

    scatterplt(pred4.get())
```

NaiveBayes() Function Overview

This function serves as the backend logic for Naive Bayes-based disease prediction in the Tkinter application. It:

- Validates user input by checking if the name and at least two symptoms are provided
- Trains a Gaussian Naive Bayes model using the training dataset
- Evaluates model performance using accuracy score and confusion matrix
- Predicts the disease based on the selected symptoms provided by the user
- Stores the user's name, symptoms, and predicted disease in a local SQLite database
- Displays a scatter plot showing the symptoms related to the predicted disease for visualization

```
pred3=StringVar()
def NaiveBayes():
    if len(NameEn.get()) == 0:
        pred1.set(" ")
        comp=messagebox.askokcancel("System","Kindly Fill the Name")
        if comp:
            root.mainloop()
    elif((Symptom1.get()=="Select Here") or (Symptom2.get()=="Select Here")):
        pred1.set(" ")
        sym=messagebox.askokcancel("System","Kindly Fill atleast first two Symptoms")
        if sym:
            root.mainloop()
    else:
        from sklearn.naive_bayes import GaussianNB
        gnb = GaussianNB()
        gnb=gnb.fit(X,np.ravel(y))

        from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
        y_pred=gnb.predict(X_test)
        print("Naive Bayes")
        print("Accuracy")
        print(accuracy_score(y_test, y_pred))
        print(accuracy_score(y_test, y_pred,normalize=False))
        print("Confusion matrix")
        conf_matrix=confusion_matrix(y_test,y_pred)
        print(conf_matrix)

        psymptoms = [Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),Symptom5.get()]
        for k in range(0,len(l1)):
            for z in psymptoms:
                if(z==l1[k]):
                    l2[k]=1
        inputtest = [l2]
        predict = gnb.predict(inputtest)
        predicted=predict[0]

        h='no'
        for a in range(0,len(disease)):
            if predicted in disease:
                h='yes'
                break
```

```
    if (h=='yes'):
        pred3.set(" ")
        pred3.set(disease[a])
    else:
        pred3.set(" ")
        pred3.set("Not Found")
     #Creating the database if not exists named as database.db and creating table if not exists
     named as NaiveBayes using sqlite3
    import sqlite3
    conn = sqlite3.connect('database.db')
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS NaiveBayes(Name StringVar,Symtom1 StringVar,Symtom2
    StringVar,Symtom3 StringVar,Symtom4 TEXT,Symtom5 TEXT,Disease StringVar)")
    c.execute("INSERT INTO NaiveBayes(Name,Symtom1,Symtom2,Symtom3,Symtom4,Symtom5,Disease) VALUES
    (?,?,?,?,?,?,?)",(NameEn.get(),Symptom1.get(),Symptom2.get(),Symptom3.get(),Symptom4.get(),
    Symptom5.get(),pred3.get()))
    conn.commit()
    c.close()
    conn.close()
    #printing scatter plot of disease predicted vs its symptoms
    scatterplt(pred3.get())
```

Tkinter GUI Setup and Reset Function Overview

This portion of the code sets up the main window and manages the state of the symptom inputs and predictions in the Tkinter application. Here's the explanation:

- The root window is configured with a background color ('Whitesmoke'), a title ("Cattle disease prediction using Machine Learning"), and is set to a fixed size (non-resizable).
- Five StringVar() variables (Symptom1 to Symptom5) are created to hold the selected symptoms from dropdown menus, all initialized to "Select Here" as the default placeholder.
- A StringVar() named Name is created to store the user's name input.
- A global variable prev_win is initialized as None to keep track of any previous popup or window that may need to be closed.
- The Reset() function clears the user input and resets the GUI state by:
    - Setting all symptom variables back to "Select Here"
    - Clearing the text in the name entry field (NameEn)
    - Resetting all prediction output variables (pred1, pred2, pred3, pred4) to empty strings
    - Attempting to close and destroy any previously opened popup window stored in prev_win to avoid clutter, handling the case if prev_win does not exist.

```
#Create a root window
root.configure(background='Whitesmoke')
root.title('Cattle disease prediction using Machine Learning')
root.resizable(0,0)
```

```
Symptom1 = StringVar()
Symptom1.set("Select Here")

Symptom2 = StringVar()
Symptom2.set("Select Here")

Symptom3 = StringVar()
Symptom3.set("Select Here")

Symptom4 = StringVar()
Symptom4.set("Select Here")

Symptom5 = StringVar()
Symptom5.set("Select Here")
Name = StringVar()
prev_win=None
def Reset():
    global prev_win

    Symptom1.set("Select Here")
    Symptom2.set("Select Here")
    Symptom3.set("Select Here")
    Symptom4.set("Select Here")
    Symptom5.set("Select Here")
    NameEn.delete(first=0,last=100)
    pred1.set(" ")
    pred2.set(" ")
    pred3.set(" ")
    pred4.set(" ")
    try:
        prev_win.destroy()
        prev_win=None
    except AttributeError:
        pass
```

GUI Exit Handling and Header Setup Overview

Exit Function:

- Defines a function Exit() that shows a confirmation dialog box asking the user, “Do you want to exit the system?” using messagebox.askyesno.
- If the user clicks “Yes,” the main Tkinter window (root) is destroyed, and the program exits cleanly.

GUI Header Label:

- Creates a label widget w2 at the top of the window displaying the text “Cattle disease prediction using Machine Learning.”
- The label has indigo-colored text on a ‘Whitesmoke’ background.
- Font is set to a large, bold italic Times font (size 30).
- The label is positioned using the grid layout manager, spanning two columns with padding on the sides.

Name Input Label:

- Creates a label NameLb for the user to identify the input field for the “Cattle ID/Name,” marked as required by the asterisk (*).
- The label text is black on a white background, styled with a bold italic Times font of size 15.
- Positioned in row 6, column 0 of the grid with vertical padding and left alignment.

```
from tkinter import messagebox
def Exit():
    qExit=messagebox.askyesno("System","Do you want to exit the system")

    if qExit:
        root.destroy()
        exit()
#Headings·for·the·GUI·written·at·the·top·of·GUI
w2·=·Label(root,·justify=LEFT,·text="·Cattle·disease·prediction·using·Machine·Learning·",·fg="indigo",·
bg="Whitesmoke")
w2.config(font=("Times",30,"bold·italic"))
w2.grid(row=1,·column=0,·columnspan=2,·padx=100)
#w2·=·Label(root,·justify=LEFT,·text="Contributors:·Created·By·Thyagarajan·K",·fg="Pink",·bg="Ivory")
#w2·=·Label(root,·justify=LEFT,·text="GitHub:https://github.com/thyagarajank/
Cattle-disease-prediction-using-Machine-Learning",·fg="Pink",·bg="Ivory")
w2.config(font=("Times",30,"bold·italic"))
w2.grid(row=2,·column=0,·columnspan=2,·padx=100)
#Label for the name
NameLb = Label(root, text="Cattle ID/Name *", fg="Black", bg="White")
NameLb.config(font=("Times",15,"bold italic"))
NameLb.grid(row=6, column=0, pady=15, sticky=W)
```

GUI Components Setup: Symptom Labels, Algorithm Labels, and User Inputs

This code segment builds the core user interface components of the Tkinter application, focusing on symptom input fields, algorithm labels, and user data entry controls.

Symptom Labels:

Five labels (Symptom 1 to Symptom 5) are created to guide users in selecting symptoms. The first four symptoms are marked as mandatory with an asterisk (*), while the fifth is optional. Each label is styled with a bold italic Times font and positioned vertically in the GUI with consistent spacing.

Algorithm Labels:

Four labels identify the different machine learning algorithms used for disease prediction: Decision Tree, Random Forest, Naive Bayes, and K-Nearest Neighbour. These labels use white text on a deepskyblue background, each with a fixed width and similar font styling, laid out vertically below the symptom inputs to clearly separate the algorithm results section.

User Input Fields:

- An entry box allows users to input the cattle’s ID or name.

- Dropdown menus (OptionMenu) let users select symptoms from a sorted list of available symptoms (l1). These dropdowns correspond to the symptom labels and ensure structured, error-free symptom input.

```
#Creating Labels for the symtoms
S1Lb = Label(root, text="Symptom 1 *", fg="Black", bg="White")
S1Lb.config(font=("Times",15,"bold italic"))
S1Lb.grid(row=7, column=0, pady=10, sticky=W)

S2Lb = Label(root, text="Symptom 2 *", fg="Black", bg="White")
S2Lb.config(font=("Times",15,"bold italic"))
S2Lb.grid(row=8, column=0, pady=10, sticky=W)

S3Lb = Label(root, text="Symptom 3 *", fg="Black",bg="White")
S3Lb.config(font=("Times",15,"bold italic"))
S3Lb.grid(row=9, column=0, pady=10, sticky=W)

S4Lb = Label(root, text="Symptom 4 *", fg="Black", bg="White")
S4Lb.config(font=("Times",15,"bold italic"))
S4Lb.grid(row=10, column=0, pady=10, sticky=W)

S5Lb = Label(root, text="Symptom 5", fg="Black", bg="White")
S5Lb.config(font=("Times",15,"bold italic"))
S5Lb.grid(row=11, column=0, pady=10, sticky=W)
#Labels for the different algorithms
lrLb = Label(root, text="DecisionTree", fg="white", bg="deepskyblue", width = 20)
lrLb.config(font=("Times",15,"bold italic"))
lrLb.grid(row=15, column=0, pady=10,sticky=W)

destreeLb = Label(root, text="RandomForest", fg="White", bg="deepskyblue", width = 20)
destreeLb.config(font=("Times",15,"bold italic"))
destreeLb.grid(row=17, column=0, pady=10, sticky=W)


knnLb = Label(root, text="Naive Bayes", fg="White", bg="deepskyblue", width = 20)
knnLb.config(font=("Times",15,"bold italic"))
knnLb.grid(row=19, column=0, pady=10, sticky=W)
OPTIONS = sorted(l1)

ranfLb = Label(root, text="K-Nearest Neighbour", fg="White", bg="deepskyblue", width = 20)
ranfLb.config(font=("Times",15,"bold italic"))
ranfLb.grid(row=21, column=0, pady=10, sticky=W)
#Taking name as input from user
NameEn = Entry(root, textvariable=Name)
NameEn.grid(row=6, column=1)

#Taking Symptoms as input from the dropdown from the user
S1 = OptionMenu(root, Symptom1,*OPTIONS)
S1.grid(row=7, column=1)

S2 = OptionMenu(root, Symptom2,*OPTIONS)
S2.grid(row=8, column=1)

S3 = OptionMenu(root, Symptom3,*OPTIONS)
S3.grid(row=9, column=1)

S4 = OptionMenu(root, Symptom4,*OPTIONS)
S4.grid(row=10, column=1)

S5 = OptionMenu(root, Symptom5,*OPTIONS)
S5.grid(row=11, column=1)
```

Prediction Buttons and Output Display Setup Overview

This section adds interactive buttons to trigger disease prediction and manages how the results are shown in the Tkinter GUI. It:

Adds Buttons to run each of the four ML models:

- Prediction 1: Runs DecisionTree() model
- Prediction 2: Runs randomforest() model
- Prediction 3: Runs KNN() model
- Prediction 4: Runs NaiveBayes() model

Adds Utility Buttons:

- Reset Inputs: Clears all input fields using the Reset() function
- Exit System: Asks for exit confirmation and closes the application with Exit()

Displays Output for each algorithm:

- Shows prediction results in separate styled labels using pred1, pred2, pred3, and pred4 variables
- Each output label has consistent styling and is visually matched to its corresponding prediction button

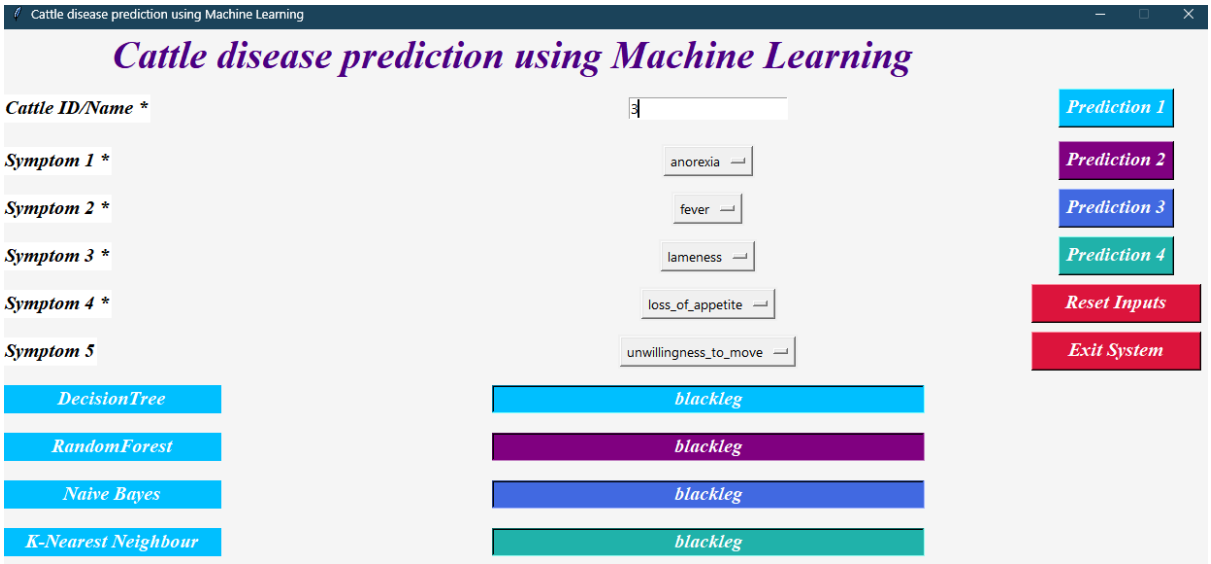
Final Line:

- root.mainloop() starts the Tkinter event loop, allowing the GUI to run and respond to user actions.

```
#Buttons for predicting the disease using different algorithms
dst = Button(root, text="Prediction 1", command=DecisionTree,bg="deepskyblue",fg="White")
dst.config(font=("Times",15,"bold italic"))
dst.grid(row=6, column=3,padx=10)

rnf = Button(root, text="Prediction 2", command=randomforest,bg="Purple",fg="White")
rnf.config(font=("Times",15,"bold italic"))
rnf.grid(row=7, column=3,padx=10)

lr = Button(root, text="Prediction 4", command=NaiveBayes,bg="royalblue",fg="white")
lr.config(font=("Times",15,"bold italic"))
lr.grid(row=9, column=3,padx=10)

kn = Button(root, text="Prediction 3", command=KNN,bg="lightseagreen",fg="White")
kn.config(font=("Times",15,"bold italic"))
kn.grid(row=8, column=3,padx=10)

rs = Button(root,text="Reset Inputs", command=Reset,bg="crimson",fg="White",width=15)
rs.config(font=("Times",15,"bold italic"))
rs.grid(row=10,column=3,padx=10)

ex = Button(root,text="Exit System", command=Exit,bg="crimson",fg="White",width=15)
ex.config(font=("Times",15,"bold italic"))
ex.grid(row=11,column=3,padx=10)
```

```
#Showing the output of different algorithms
t1=Label(root,font=("Times",15,"bold italic"),text="Decision Tree",height=1,bg="deepskyblue"
        ,width=40,fg="white",textvariable=pred1,relief="sunken").grid(row=15, column=1, padx=10)

t2=Label(root,font=("Times",15,"bold italic"),text="Random Forest",height=1,bg="Purple"
        ,width=40,fg="white",textvariable=pred2,relief="sunken").grid(row=17, column=1, padx=10)

t3=Label(root,font=("Times",15,"bold italic"),text="Naive Bayes",height=1,bg="royalblue"
        ,width=40,fg="white",textvariable=pred3,relief="sunken").grid(row=21, column=1, padx=10)

t4=Label(root,font=("Times",15,"bold italic"),text="K-Nearest Neighbour",height=1,bg="lightseagreen"
        ,width=40,fg="white",textvariable=pred4,relief="sunken").grid(row=19, column=1, padx=10)
#Mainloop the application is ready to run
root.mainloop()
```

Input Taken From User And Output



Accuracy and Confusion Matrix of the Algorithms

Decision Tree

```
Decision Tree
Accuracy
1.0
26.0
Confusion matrix
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
Symptoms: ['anorexia', 'fever', 'lameness', 'loss_of_appetite', 'unwillingness_to_move']
Binary Vector: [1, 1, 1, 1, 1]
```

Random Forest

```
Random Forest
Accuracy
1.0
26.0
Confusion matrix
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
...
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

Naïve Bayes

```
Naive Bayes
Accuracy
1.0
26.0
Confusion matrix
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
...
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

KNearest Neighbour

```
kNearest Neighbour
Accuracy
1.0
26.0
Confusion matrix
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
...
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

Conclusion

This report demonstrates the practical potential of machine learning in enhancing veterinary diagnostics, particularly in predicting animal and cattle diseases based on symptom inputs. By leveraging algorithms such as K-Nearest Neighbors, Naive Bayes, Decision Tree, and Random Forest, the system achieves automated, fast, and reasonably accurate predictions without requiring deep domain expertise at the user end.

The successful implementation and comparison of multiple classifiers revealed that different algorithms offer varying strengths depending on the dataset and complexity of symptoms. Among them, tree-based models like Random Forest often showed robust performance. The integration of a GUI through Tkinter further enhanced usability, allowing farmers and field practitioners to interact with the system intuitively.

Overall, the project bridges the gap between veterinary knowledge and accessible digital tools, offering a scalable foundation for future improvements such as deeper datasets, real-time monitoring, or mobile deployment. This supports timely interventions, ultimately contributing to better animal health management and economic resilience in livestock-based industries.