Email Spam Detection Assignment: Detailed Explanation

Problem Statement

This assignment focuses on classifying emails as either "Spam" or "Not Spam" using binary classification. The algorithms to be employed are:

1. **K-Nearest Neighbors (KNN)**

2. **Support Vector Machine (SVM)**

The goal is to build models with these algorithms, predict spam, and evaluate their performances on a dataset of emails available on Kaggle. The performance comparison will include metrics like R-squared (R^2) and Root Mean Squared Error (RMSE).

Objectives

1. **Dataset Understanding and Cleanup**: Explore the dataset for understanding its structure, identifying missing data, and handling anomalies if necessary.

2. **Model Building with KNN and SVM**: Develop two classification models using KNN and SVM.

3. **Model Evaluation and Comparison**: Assess the models based on their accuracy, R^2, RMSE, and other relevant metrics.

Dataset Overview

The dataset includes 5172 emails stored as a CSV file, with each row corresponding to an email. It has 3002 columns:

- The first column is the email ID.

- The last column contains the label (1 = Spam, 0 = Not Spam).

- The middle 3000 columns represent word counts for the 3000 most frequent words across all emails.

Algorithms for Spam Classification

1. **K-Nearest Neighbors (KNN)**

The **KNN algorithm** is a simple, yet powerful, supervised learning algorithm. It classifies data points based on the majority class among the "K" closest neighbors. It works as follows:

1. **Distance Calculation**: For any new data point (i.e., a test email), KNN calculates the distance between this point and all training data points.

2. **Choosing Neighbors**: It selects the "K" closest points (neighbors) based on the calculated distances.

3. **Class Prediction**: The new data point is classified into the majority class among its "K" nearest neighbors.

K Selection:

- Choosing "K" is crucial. A small "K" may result in overfitting (high variance), while a larger "K" may lead to underfitting (high bias).

- Cross-validation is typically used to determine the best "K" value for the dataset.

KNN Steps:

1. **Data Preprocessing**: Normalize or scale features to ensure distance calculations are meaningful.

2. **Training**: Store the training data without explicit learning.

3. **Prediction**: Classify a new email based on the majority class among its nearest neighbors.

4. **Evaluation**: Use confusion matrices, accuracy scores, and other metrics to assess performance.

2. **Support Vector Machine (SVM)**

The **SVM algorithm** is a supervised learning model used for classification tasks. Its goal is to identify a decision boundary (or hyperplane) that maximizes the margin between two classes. Support vectors, or the closest points to the hyperplane, help determine the best separation.

Types of SVM:

- **Linear SVM**: Used when data is linearly separable.

- **Non-linear SVM**: Employs kernels for more complex, non-linear data.

Kernel Trick:

SVM can use kernel functions to transform data into higher dimensions, making it possible to separate non-linearly separable data. Common kernels include:

- **Linear Kernel**: Suitable for high-dimensional data.

- **Polynomial Kernel**: Less common due to computational cost.

- **Gaussian RBF Kernel**: Effective for non-linear data.

- **Sigmoid Kernel**: Occasionally used, but more common in neural networks.

SVM Steps:

1. **Import and Preprocess Data**: Handle scaling for better hyperplane positioning.

2. **Train-Test Split**: Divide data into training and testing sets.

3. **Train SVM Model**: Define the kernel and train the SVM.

4. **Prediction and Evaluation**: Measure model performance using confusion matrices, accuracy, and hinge loss.

Questions and Explanations

1. **Why should we not use the KNN algorithm for large datasets?**

   - KNN is computationally expensive for large datasets since it computes the distance from each test point to all training points. This can be slow and memory-intensive, making KNN less suitable for high-volume data.

2. **Is Feature Scaling required for the KNN Algorithm? Explain.**

   - Yes, feature scaling is crucial for KNN. KNN relies on distance calculations, and without scaling, features with larger ranges could dominate the distance metric, leading to biased predictions.

3. **What is Hinge Loss?**

   - Hinge loss is a cost function used in SVMs to maximize the margin between classes. It penalizes misclassifications or points inside the margin, aiming for a zero loss when points are correctly classified with a significant margin.

4. **What's the "kernel trick" and how is it useful?**

   - The kernel trick allows SVM to operate in a higher-dimensional space without explicitly transforming data. By applying a kernel function, SVM finds a hyperplane in this new space to separate classes more effectively, especially for non-linear data.

This assignment combines both KNN and SVM models for an in-depth comparison and performance analysis, contributing to a deeper understanding of supervised machine learning techniques for text classification.

…………………………………………………………………….code explanation……………………………………………………………..

Here's a more detailed explanation of each part of the code, with a deeper look into the concepts and specific functions used.

Step 1: Importing Libraries

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn import preprocessing
```

Each library serves a specific purpose in data analysis, machine learning, or visualization:

- **`pandas`**: Used for data manipulation. With functions like `pd.read_csv` and `DataFrame` operations, it provides an easy way to load and explore data.

- **`numpy`**: Helps with numerical operations, especially arrays and matrices, which are efficient for handling large datasets.

- **`seaborn`** and **`matplotlib`**: These libraries are used for data visualization, allowing us to visualize relationships in data and understand distributions before training models.

- **`sklearn` (scikit-learn)**: A machine learning library offering tools for model building, preprocessing, and evaluation. We specifically import functions for splitting the data (`train_test_split`), creating models (`SVC`, `LinearSVC`, `KNeighborsClassifier`), and evaluating models (`metrics`).

Step 2: Loading the Dataset

```python
df = pd.read_csv('emails.csv')
```

This line loads a CSV file named `emails.csv` into a pandas DataFrame `df`. A **DataFrame** is a table-like structure where rows represent samples, and columns represent features. Here, each row represents an email, and the columns capture various properties or words in the email, which serve as features.

Step 3: Exploring the Dataset

```python
df.info()
```

The `info()` function provides a summary of the DataFrame, showing the number of rows, columns, data types, and memory usage. This function is essential for:

- **Understanding Data Types**: Allows us to identify numerical and categorical data.

- **Identifying Missing Values**: Missing data is indicated by columns where the number of non-null entries is less than the total entries.

Example Output:

```

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 5172 entries, 0 to 5171

Columns: 3002 entries, Email No. to Prediction

dtypes: int64(3001), object(1)

memory usage: 118.5+ MB

```

- **Analysis**: Here, `Email No.` is an identifier, and `Prediction` is the target variable (1 for spam, 0 for non-spam). The other columns likely represent word occurrences or presence (common in spam filtering).

Step 4: Data Inspection

```python

df.head()

```

The `head()` function displays the first few rows of the DataFrame. This allows us to inspect the structure and type of data we're dealing with. For example, if the columns contain word frequencies or binary indicators, it helps confirm our assumptions and visualize a few sample rows.

Step 5: Cleaning the Dataset

Dropping Unnecessary Columns

```python
df.drop(columns=['Email No.'], inplace=True)
```

- **Dropping Columns**: The `Email No.` column, which serves as an identifier, does not contribute to the predictive model, so it’s dropped using `drop()` to streamline the dataset.

Checking for Missing Values

```python
df.isna().sum()
```

- **`isna().sum()`**: Checks for missing values. Handling missing values is crucial because machine learning algorithms often cannot handle null values. If any columns had nulls, options would include filling them or dropping the rows/columns.

Descriptive Statistics

```python
df.describe()
```

- **Descriptive Statistics**: The `describe()` function calculates summary statistics, which help in understanding the distribution of numerical columns (e.g., mean, standard deviation, min, max, quartiles). This helps identify any outliers or anomalies.

Step 6: Separating Features and Labels

```python
X = df.iloc[:, :df.shape[1] - 1]
```

y = df.iloc[:, -1]

X.shape, y.shape

```

- **Feature (`X`) and Label (`y`) Separation**:

  - **Features (`X`)**: All columns except the last one (`Prediction`) are selected as features.

  - **Labels (`y`)**: The last column, `Prediction`, is the target variable.

- **Shape Check**: Displays the shape of `X` and `y` to ensure they contain the expected number of samples and features.

Example Output:

```

((5172, 3000), (5172,))

This output confirms that `X` has 5172 samples with 3000 features, and `y` has 5172 labels.

Step 7: Splitting the Dataset

```python

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=8)

```

- **Train-Test Split**:

  - **Training Set**: 85% of data, used to train the model.

  - **Testing Set**: 15% of data, used to evaluate the model's performance.

  - **`random_state=8`**: Ensures that the split is reproducible, so running the code multiple times yields the same split.

Step 8: Defining Machine Learning Models

```python
models = {
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=2),
    "Linear SVM": LinearSVC(random_state=8, max_iter=900000),
    "Polynomial SVM": SVC(kernel="poly", degree=2, random_state=8),
    "RBF SVM": SVC(kernel="rbf", random_state=8),
    "Sigmoid SVM": SVC(kernel="sigmoid", random_state=8)
}
```

Each model is defined with a specific algorithm and relevant parameters:

- **K-Nearest Neighbors (KNN)**:
  - Uses distance-based voting from `n_neighbors=2` closest neighbors to classify data points.
  - Suited for local patterns but can struggle with high-dimensional data.

- **Linear SVM**:
  - Uses a linear boundary to separate classes.
  - **`max_iter=900000`**: Sets the maximum number of iterations to avoid stopping too early.

- **Polynomial SVM**:
  - Uses a polynomial kernel to create a non-linear decision boundary.
  - **`degree=2`**: Sets the polynomial degree, creating a quadratic decision surface.

- **RBF (Radial Basis Function) SVM**:

  - Uses an RBF kernel, well-suited for non-linear boundaries.

  - Effective in high-dimensional spaces and more flexible.

- **Sigmoid SVM**:

  - Uses a sigmoid kernel to capture complex patterns but can be less effective for some datasets.

Step 9: Training and Evaluating Each Model

```python
for model_name, model in models.items():
    y_pred = model.fit(X_train, y_train).predict(X_test)
    print(f"Accuracy for {model_name} model \t: {metrics.accuracy_score(y_test, y_pred)}")
```

Each model is trained, and predictions are made on the test set:

1. **Model Training**:

   - **`model.fit(X_train, y_train)`**: Trains each model using the training data.

2. **Model Prediction**:

   - **`.predict(X_test)`**: Predicts the class labels for the test set.

3. **Accuracy Calculation**:

   - **`metrics.accuracy_score(y_test, y_pred)`**: Calculates accuracy, which is the proportion of correctly predicted labels.

Sample Output:

```
Accuracy for K-Nearest Neighbors model : 0.8879
Accuracy for Linear SVM model          : 0.9755
Accuracy for Polynomial SVM model      : 0.7616
Accuracy for RBF SVM model            : 0.8183
Accuracy for Sigmoid SVM model        : 0.6237
```

Step 10: Analyzing Results

Each model's accuracy helps determine which algorithm is most effective. For this dataset:

- **Linear SVM** performs best, with ~97.6% accuracy, suggesting that the data is largely linearly separable.
- **K-Nearest Neighbors (KNN)** achieves ~88.8% accuracy, indicating it captures local patterns but is outperformed by the linear boundary.
- **Polynomial and RBF SVMs** show moderate accuracy, meaning a more complex decision surface doesn’t significantly improve results.
- **Sigmoid SVM** has the lowest accuracy (~62.4%), suggesting that this kernel doesn’t suit the dataset.

Summary

The code:

1. Loads and preprocesses the dataset.
2. Splits data into training and test sets.
3. Compares five classifiers on spam detection.
4. Reveals that a **linear SVM** model performs best, effectively classifying spam vs. non-spam emails.