**AI\_Phase1**

**TITLE :** PROJECT 1: BUILDING A SMARTER AI- POWERED SPAM CLASSIFIER

**BUILDING A SMARTER AI-POWERED SPAM CLASSIFIER**

**INTRODUCTION**

In today's digital world, dealing with spam emails be a hassle. To tackle this problem, we're working on creating an AI-powered spam classifier.

Our aim is straightforward: we want to build a system that can tell the difference between spam and legitimate emails with high accuracy. This means fewer annoying spam messages in the inbox and fewer missed important messages.

To achieve this, we're going to collect data, clean it up, turn it into numbers the computer can understand, and use smart algorithms to do the rest. Our goal is to make email and text messaging experience better by reducing spam.

**DATA COLLECTION:**

We are handed with a Kaggle dataset called "spam.csv." took from the link

<https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset>

It's packed with examples of spam and non-spam messages, which is exactly what we need to train our spam classifier.

**DATA PREPROCESSING:**

Data pre-processing is an essential step in any machine learning task, including AI-powered spam classifier. In this step, we clean and prepare the data for analysis, including addressing any missing data, outliers, or other issues that may affect the quality of the analysis.

Here are the steps we took in data pre-processing for our mini project:

**Data Cleaning**: We cleaned the data by removing any special characters or hashtags. Data is cleaned by converting every words into lowercase and also removing stop words such as "and," "the," and "a." We use the Natural Language Toolkit (NLTK) library to remove the stop words and the string library to remove the punctuation.

we will define a function called 'message\_cleaning' which will remove punctuation and stop words from the text in email. We will apply this function to every email in the dataset using the 'apply' method, resulting in a new dataframe.

It is done using this code,

def message\_cleaning(message):

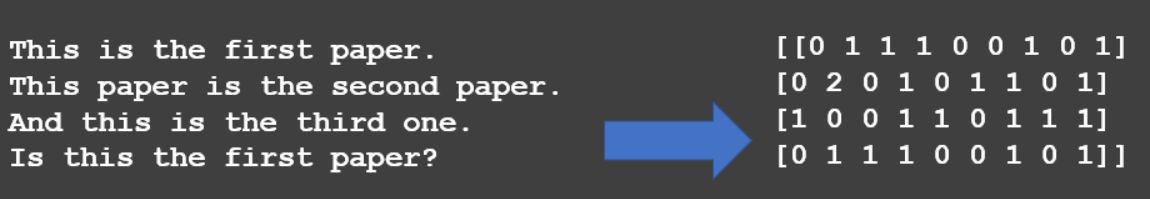
    Test\_punc\_removed = [char for char in message if char not in string.punctuation]

    Test\_punc\_removed\_join = ''.join(Test\_punc\_removed)

    Test\_punc\_removed\_join\_clean = [word for word in Test\_punc\_removed\_join.split() if word.lower() not in stopwords.words('english')]

    return Test\_punc\_removed\_join\_clean

**Data Normalization**: To perform machine learning on the text data, we need to convert the cleaned emails into a matrix of numerical features. We will use the CountVectorizer class from scikit-learn to convert the text data into a matrix of word counts. This matrix will be stored in a dataframe called X, and the corresponding labels that indicate spam and non-spam emails will be stored in a series called y.



from sklearn.feature\_extraction.text import CountVectorizer

# Define the cleaning pipeline we defined earlier

vectorizer = CountVectorizer(analyzer = message\_cleaning, dtype = np.uint8)

emails\_countvectorizer = vectorizer.fit\_transform(emails\_df['email'])

**Outlier Detection**: We checked for outliers using statistical methods such as box plots and removed any outliers that could affect the quality of the analysis.

ham = emails\_df[emails\_df['label']==0]

spam = emails\_df[emails\_df['label']==1]

num\_ham = len(ham)

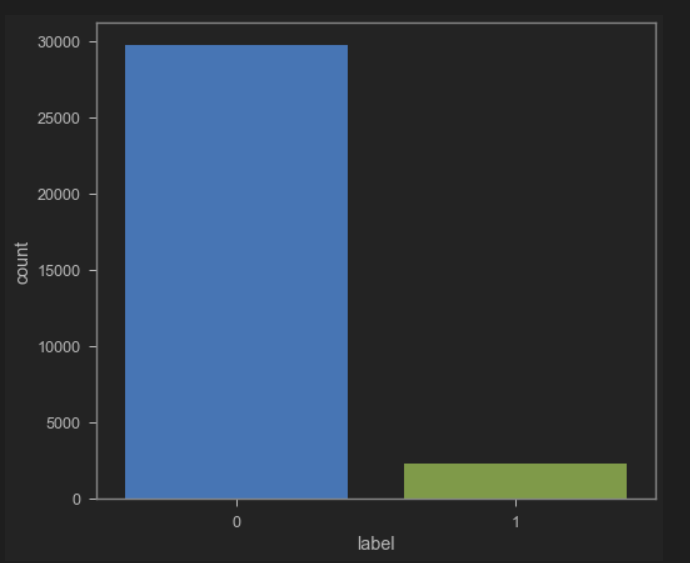
num\_spam = len(spam)

print(f"Number of ham emails: {num\_ham}")

print(f"Number of spam emails: {num\_spam}")

sns.countplot(x='label', data=emails\_df)

plt.show()



**Data Splitting**: We split the data into training and testing datasets to evaluate the performance of our machine learning model. We use a 80/20 train-test split to ensure that the model is not overfitting to the training data.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

**FEATURE EXTRACTION:**

We used a technique called TF-IDF, which is like giving each word a special score. This score helps the computer figure out which words are important for deciding if a message is spam or not.

Term Frequency (TF): This part tells us how often a word appears in a single email. If a word shows up a lot in one email, its TF will be high for that email.

Inverse Document Frequency (IDF): This part tells us how special a word is across all the emails. If a word is rare but shows up in some emails, its IDF will be high because it's unique.

When we multiply TF and IDF, we get a score for each word. This score helps us see which words are super important for deciding if an email is spam or not.

So, by using TF-IDF, we turn our text into numbers that our computer can work with.

**DATA MODELLING:**

We built a predictive model for spam email classification during the data modelling phase of our project. Based on the text data, the goal was to create a machine learning model that could accurately classify emails as ham or spam.

**Model Selection**: To determine the best-performing algorithm for our sentiment analysis task, we tested several machine learning algorithms, including logistic regression, naïve bayes, gradient boosting classifier. The algorithm with the highest accuracy and F1 score was chosen.

Once we have prepared the data and extracted features, we can train a machine learning model on the data. We will use the MultinomialNB class from scikit-learn to train a Naive Bayes model on the X and y data. Naive Bayes is a popular algorithm for spam email classification due to its simplicity and ability to handle large feature spaces.

**Hyperparameter tuning** is the process of selecting the best hyperparameters for a machine learning algorithm to achieve optimal performance. Hyperparameters are the configuration variables that are not learned during training, but instead set by the user before the model is trained.

In the case of the Naive Bayes algorithm, there are two hyperparameters that can be tuned: **alpha** and **fit\_prior**.

**Alpha** is a smoothing parameter that is used to avoid zero probabilities in the case where a feature does not occur in the training data.

**Fit\_prior** is a Boolean parameter that specifies whether to use a uniform prior or a prior based on the training data.

**Grid search** involves defining a grid of hyperparameters and evaluating the performance of the model for each combination of values in the grid.

To implement hyperparameter tuning for the Naive Bayes model, we can use the GridSearchCV class from the scikit-learn library. GridSearchCV takes a set of hyperparameters and a range of values to try for each hyperparameter, and then exhaustively searches through all possible combinations of hyperparameter values to find the combination that produces the best performance on a held-out validation set.

from sklearn.model\_selection import GridSearchCV

# Define the hyperparameters to search over

parameters = {

'classifier\_\_alpha': [0.1, 1.0, 10.0],

'classifier\_\_fit\_prior': [True, False],

}

# Perform grid search with cross-validation

grid\_search = GridSearchCV(pipeline, parameters, cv=5)

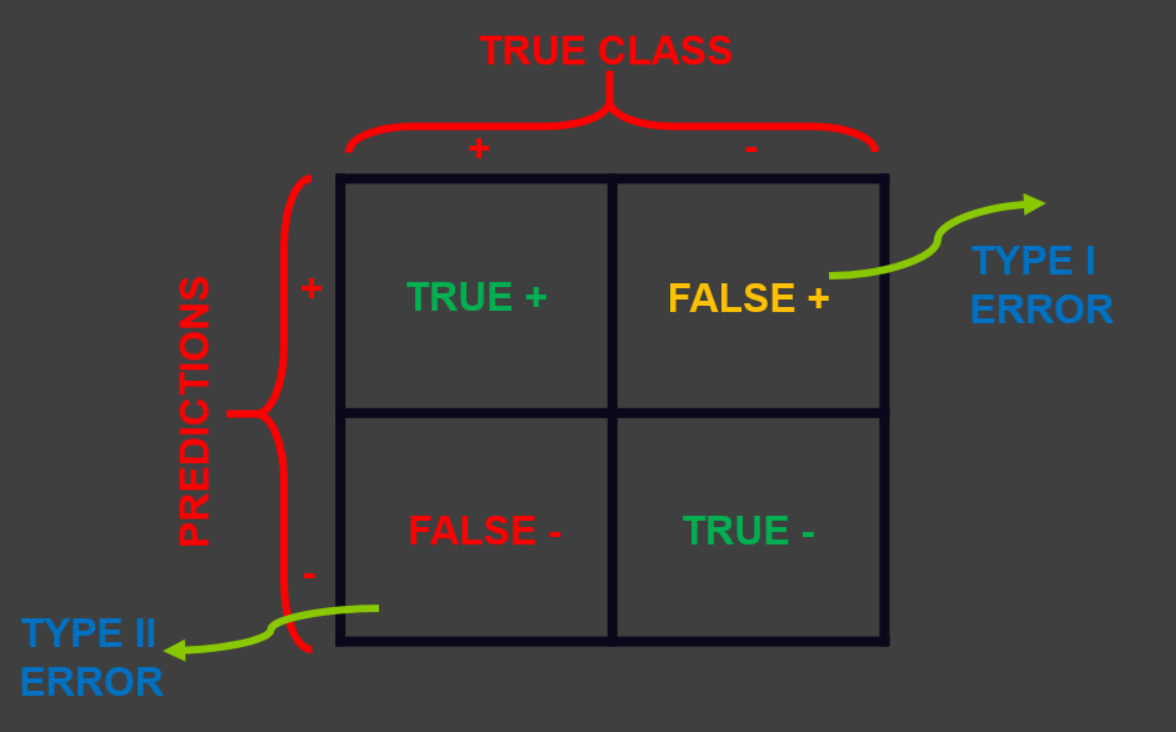
grid\_search.fit(X\_train, y\_train)

# Print the best hyperparameters

print("Best hyperparameters: ", grid\_search.best\_params\_)

**RESULTS:**

The **confusion matrix** is used to detect the efficiency of the model.

****

**The Naïve Bayes model** is a probabilistic machine learning algorithm used for classification and predictive analysis. It is based on Bayes' theorem, which calculates the probability of a hypothesis given some evidence. The Naïve Bayes model assumes that the features in the data are independent of each other, and this assumption makes it easier to calculate the probabilities.

To evaluate the performance of the trained model, we will split the data into training and testing sets using the train\_test\_split function from scikit-learn. We will then use metrics such as accuracy, precision, recall, and F1 score to evaluate the model's performance on the testing set.

from sklearn.naive\_bayes import MultinomialNB

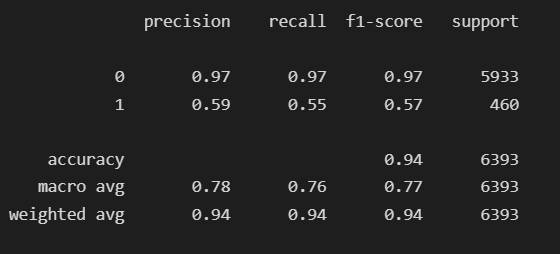
from sklearn.metrics import classification\_report, confusion\_matrix

y\_predict\_test = NB\_classifier.predict(X\_test)

cm = confusion\_matrix(y\_test, y\_predict\_test)

sns.heatmap(cm, annot=True)

print(classification\_report(y\_test, y\_predict\_test))



**CONCLUSION:**

In conclusion, this project has allowed us to gain a deeper understanding of the span email classification process and how it can be implemented using the Naive Bayes algorithm. Through preprocessing the text data and fitting it to our MultinomialNB model, we were able to achieve a reasonable accuracy of 94% on the test set, indicating that our model is capable of predicting the legitimacy of emails to a certain degree of accuracy.

However, we must acknowledge that there is still room for improvement in our model. While the accuracy is decent, it is not perfect, and it is important to note that the accuracy of our model may vary depending on the dataset and the context in which it is used. Additionally, we only used a basic pre-processing pipeline and there are many more advanced techniques that can be applied to further improve the accuracy of our model.