Christopher Kevin Siswanto
2600220449-4
ISSE Course
AI Final Assignment: Scientific Abstract Classification

Scientific Abstract Classification

1. Introduction

Nowadays, Artificial Intelligence (AI) is becoming one of the most popular computer science branches, it all started with the emerging machine learning technology that utilizes a computer to execute tasks for humans. Machine learning uses algorithms developed to perform tasks without specific instructions, this leads to the ability of processing large amounts of documents and creating patterns that would otherwise be unrecognizable by humans.

In order to make computers more effective and intelligent, neural networks are introduced. Neural network is a method in AI where the computer is taught by humans to think and make actions that are similar to a human brain. However, the computer is only using logical thinking and not emotional thinking. Neural networks are important in this report since they can make intelligent decisions without human help, they can recognize texts and make general observations. This method will be used to gather crucial connections between data and is mainly used in document summarization.

Natural Language Processing (NLP) is a machine learning technology that makes computers recognize and understand human language. NLP has been used around the world to help with analyzing text and speech data. It has the ability to efficiently process large documents and one of the most prominent uses of NLP is to distinguish and correlate information where needed.

Information is constantly flooding the internet, these consist of audios, videos, and texts. Most of the data are in the form of texts, this is where text classifiers are important. Using machine learning to sift through information and let the machine organize and group these data that would’ve been

time consuming for a human's task. Text classification in machine learning labels the data into structured and categorized labels.

In this report, the objective is to create a machine learning model that can classify abstracts from a research paper and comprehend which categories it should be placed in. A machine learning model is used to categorize the texts, however, there are dozens of machine learning algorithms that can be utilized. The main programming language chosen for creating the model is python. Python is recognized to be one of the best languages in the fields of data science and machine learning, it contains vast options for libraries and is one of the easiest to use and efficient. Python 3 in particular is the language most developed and focuses on the future proofing by the countless developers working on it. Python is ideal for developing and implementing the text classification model, the libraries such as feedparser, NumPy, pandas, scikit-learn, and others are crucial components in machine learning model creation.

2. Methodology

2.1 Data Collection

The first step towards creating a text classifier using machine learning is to collect the texts that would eventually be used to train the classifier and produce the finished model. To ensure that the machine learning model is competent, the dataset chosen to feed into the model is 900 texts that would be processed. Research papers are used as the texts for this experiment due to the credibility that the information of the texts provide, texts or articles from the internet are rather hard to be trusted. Only the abstracts are chosen from the entirety of the research paper due to the concise and summarized information the paper provides, afterwards, the papers are collected automatically using an API from arXiv. ArXiv is a research-sharing platform that holds over two million scholar articles in eight major areas. arXiv offers its public API for users to utilize, however, it has a rate limit for non commercial projects. The API is used to gather the abstracts in three different categories, the chosen categories that would be used are math, physics, and economics.

2.2 Dataset Creation

The papers would be saved as a single csv file with their corresponding category, title, and abstract for each abstract. The category and title are important for the users to analyze the legitimacy, the category would be a main standpoint on how the model would be trained by analyzing the abstract and understanding patterns inside the text and would classify it to its corresponding categories. The category part would not be used in the testing phase since it should classify the categories by itself from the abstracts only and turned into numerical format using label encoding. Once the API fetches all the abstracts, they are saved as a file called abstracts.csv. Inside the csv file, the abstracts are ordered by one category at a time, their titles, and the abstract on the third column.

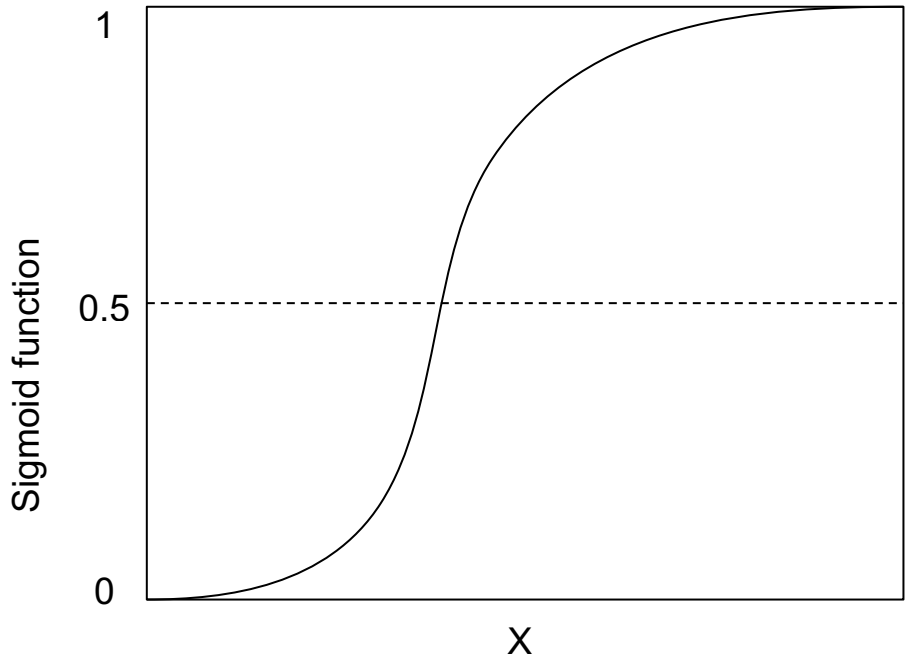
2.3 Preprocessing

Before training the model itself, there should be a preprocessing stage for the datasets created. Preprocessing is a very crucial part of data preparation and the main purpose is to clean and convert raw data gathered to a clean sheet. In the csv file, lower casing is performed using Python to ensure that all the letters in the alphabet are equally lower cased. Afterwards, tokenization is performed, this method is used to change all the text's data into smaller units called token. Tokenization is performed to enable the model to understand the human language by separating the words. Stop-word removal is also done to remove any unnecessary information, this includes low-level English words such as conjunction, and other unmeaningful words. Lemmatization is added to convert words back to the base dictionary form without tenses. Label encoding is interpreted for NLP to understand using numerical format. Finally, delete the title column to achieve raw data and the category. After preprocessing steps are executed, a new file called preprocessed_abstracts.csv is created that is ready for the training model to use.

2.4 Model Selection

Using Python 3 and the combination of libraries from the internet, the algorithm that suits the task most is the Logistic Regression method as shown in the diagram below. The algorithm is a part of the supervised learning method that uses a logistic function to operate and takes any real-valued integer and translates it to a value between 0 and 1. This specific method is chosen due to its output value between 0 and 1, when the output is closer towards 1 it means the result is a

positive sample, otherwise shifting to 0 means closer to a negative sample. The algorithm first converts the result of the sigmoid function into a probability of 0 to 1, for binary predictions the population is cut at 0.5 split into two groups that would be further analyzed and the categories can be predicted using its decision function and activation function when learning. Next it then proceeds with the cost function to measure the difference of predicted categories and true categories. Afterwards it trains the model, sets the decision boundary, and finally predicts the outcomes of the categories.



Logistic Regression Model Diagram

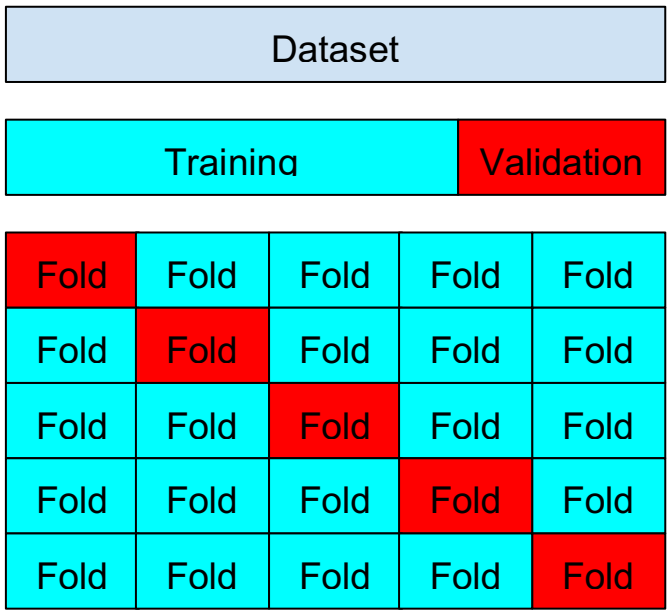
2.5 Model Training

After selecting the logistic regression model, we begin training the model to predict the abstracts given to it and classify the categories correctly. Start by determining the abstract label as the x-axis and the category as y-axis, perform word vectorization which is one of the methods in NLP is performed to change the words inside a text to a vector of numbers that the computer understands to eventually find word similarities and semantics. Afterwards we initialize the logistic regression model using a library that contains the algorithm created by sklearn models. Sklearn provides supervised training such as Naive Bayes, Random Forest, Nearest Neighbors, and others. The algorithms are free to use and built using NumPy, SciPy, and matplotlib which are

free softwares for the python programming language. Amongst the model, there are parameters that can be set according to the needs of the model such as the maximum number of iterations the model is allowed to converge, random state ensures that every time the script is run it will produce the same train test split.

After initializing the training model, a pipeline is created using the previous library. Pipelines are created to ensure that the sequence of the steps are created and executed in order, this is crucial for the reproducibility aspect of the code. The pipeline facilitates the model by encapsulating the development phases starting from preprocessing up until training and testing. Afterwards, cross validation is implemented. Overall, pipelines are created to ease the entire machine learning process.

As seen in the diagram below, a k-fold cross validation is then performed on the entire dataset using each fold with a training and validation set separated by the amount of folds. The diagram below showcases a 5 k-fold cross validation, which results in the 80:20 split of training and validation set. A k-fold cross validation trains and evaluates the model chosen based on the average accuracy for each fold and other metrics such as precision, recall, and f1-score. For each fold, an early stopping method is implemented to stop the fold after a certain number of iterations where the accuracy does not improve.



K-fold Cross Validation Diagram

3. Experiment

This section will show the rundown of the entire code and how it runs alongside what can be changed to expect a different outcome. Firstly, download and import all the prerequisite libraries or software mentioned in the experiment. The first step is data collection where we define a function called collect_papers with default parameters to obtain the abstracts. Since the data is collected from ArXiv API queries are created for each specific category that is available from the ArXiV database. Using the ArXiv API to request a search query for the categories and the maximum results specified is 300, parsing the atom response using the feedparser import that would later be saved as a csv file. (Figure 1.1)

In [3]:

```
%%time
import csv
import feedparser

def collect_papers(categories, num_papers=300):
    """
    Collects papers based on the queried category and amount written as a response in the Atom 1.0 format

    Parameters:
    categories: a string that specifies which category should be queired.
    num_papers: the number of papers to collect from each of the defined categories.

    """

    all_papers = []

    for category in categories:
        # Define the query for each category that would be queried using ArXiv API(string)
        if category == 'math':
            query_str = 'cat:math.*'
        elif category == 'physics':
            query_str = 'cat:physics.*'
        elif category == 'economics':
            query_str = 'cat:econ.*'

        # The ArXiv API URL for getting the responses
        api_url = f'http://export.arxiv.org/api/query?search_query={query_str}&max_results={num_papers}'

        # Parse the feed from Atom format
        feed = feedparser.parse(api_url)
        papers = feed.entries
        all_papers.extend(papers)

    return all_papers
```

Figure 1.1

Next step is to save the response into the csv file in the right formats, by opening a file called abstract.csv. If the file does not exist it will create a new file, the append mode is used to not overwrite the contents and instead puts it at the end of the document to not disturb other documents

if they are the same filename accidentally. The responses that were requested are then analyzed and only the category, title, and abstract are saved into the file. After defining all the functions, the process of collecting and saving the papers is executed, it might take some time to collect the data from the API. Users that use the api must read arXiv code of conducts and adhere to the policies such as the rate limit for the api that is limited to once every three seconds and other responsibilities that the developers need to follow. (Figure 1.2)

```
def save_to_csv(papers, categories, csv_filename='abstract.csv'):
    """
    Saves the papers from the API's responses collected as a csv file named asbtract.csv using the csv library.

    Parameters:
    papers: the list of papers that would be saved eventually in the csv file.
    categories: a string that specifies which category should be queired.
    csv_filename: the name of the csv file.
    """
    # opens a csv file called abstract.csv, creates it if it doesn't exist using append mode
    with open(csv_filename, 'a', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)

        # Write the header for the columns
        if csvfile.tell() == 0:
            writer.writerow(['category', 'title', 'abstract'])

        # Write to CSV file
        for category, paper in zip(categories, papers):
            abstract = paper.get('summary', paper.get('description', ''))
            writer.writerow([category, paper.title, abstract])

if __name__ == "__main__":
    categories = ['math', 'physics', 'economics']
    num_papers_per_category = 300

    # Collect and save papers for each category
    papers = collect_papers(categories, num_papers_per_category)
    save_to_csv(papers, categories)
```

```
CPU times: user 1.3 s, sys: 46.5 ms, total: 1.34 s
Wall time: 10.6 s
```

Figure 1.2

The next step in the experiment as seen at figure 1.3 is to initiate the preprocessing of the data collected. To read and write the csv file the library pandas are utilized which is the common library to edit a csv file. The sklearn library contains the needed preprocessing steps integrated and made for python. The function lemmatize_text is used to transform a word in the WordNetLemmatizer library list of English words back to its original form. (Figure 1.3)

In [4]:

```
%%time
import pandas as pd
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
from sklearn.preprocessing import LabelEncoder
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
def lemmatize_text(text):
    """
    Turns all words back to its base dictionary word.

    Parameter:
    text: the input text that is turned into the base form.

    Example: lemmatize_text("Sleeping")
    "Sleep"
    """
    lemmatizer = WordNetLemmatizer()
    tokens = word_tokenize(text)
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
    return ' '.join(lemmatized_tokens)
```

Figure 1.3

Continuing with figure 1.4, the category and title of the paper are required for the purpose where users are able to check and analyze the data collected itself and verify whether the papers collected are legitimate. Stop words removal are a set of words predetermined to remove the most common words from the data, during this phase they are considered to be insignificant and occur in all the abstracts such as conjunction words. The import WordNetLemmatizer is then used for lemmatization which converts words into their dictionary based translation, this helps with normalization of the words in the text and is defined in the lemmatize_text function. The tokenization is implemented using Natural Language Toolkit (NLTK) word tokenization library for Python, word tokenization is used to break down the text data into individual words called tokens. Lastly, label encoding is performed to the category column which defines 0 as math, 1 as physics, and 2 as economics. Finally, the title column is removed, this helps the machine learning model since it processes a numerical input rather than a text and the titles are only required for validating the abstracts. After defining all the functions for preprocessing our dataset, we call the function and execute the preprocessing steps.

```
def preprocess_csv(csv_filename='abstract.csv', processed_filename='preprocessed_abstract.csv'):
    """
    Preprocesses the scv file using necessary steps. 1st step is lower casing, 2nd is stop word removal,
    3rd is lemmatization, 4th is tokenization, 5th is label encoding, 6th is removing unnecessary labels.

    Parameters:
    csv_filename: the name of the csv file
    processed_filename: the name of the preprocessed file

    Explanations:
    str.lower(): turns all the string into lowercase
    stop_words: all the stop words from the english stop words in the sklearn library
    str.split: splits the string on text into substrings based on the blank spaces in between texts

    """

    # Read the CSV file
    df = pd.read_csv(csv_filename)

    # Lowercase conversion to the abstracts
    df['abstract'] = df['abstract'].str.lower()

    # Remove stop words from the abstracts
    stop_words = ENGLISH_STOP_WORDS
    df['abstract'] = df['abstract'].apply(lambda x: ' '.join(word for word in x.split() if word.lower() not in stop_words))

    # Apply lemmatization function
    df['abstract'] = df['abstract'].apply(lemmatize_text)

   # Tokenization using string split
    df['abstract'] = df['abstract'].apply(str.split)

    # Label encoding for the categories
    category_mapping = {'math': 0, 'physics': 1, 'economics': 2}
    df['category'] = df['category'].map(category_mapping)

    # Remove title column
    df.drop('title', axis=1, inplace=True)

    # Save to the file
    df.to_csv(processed_filename, index=False)

# Execute the function to perform preprocessing
preprocess_csv()
```

```
CPU times: user 3.8 s, sys: 2.18 s, total: 5.98 s
Wall time: 4.67 s
```

Figure 1.4

Figure 1.5 shows the model selection, the preprocessed_abstracts.csv is loaded using the pandas library that provides the ability to read the file. The dataframes otherwise known as the tables in the csv file are then divided into the X and Y labels. The X label represents the abstracts as features and the Y label represents the category for training and evaluating the model. Afterwards vectorization is initialized using TF_IDF, vectorization is a process in which the textual data is converted into a numerical format called vectors. It is one of the NLP contexts in which vectorization is done for the computers to analyze the words and find patterns and correlations between certain words in the abstract. TF_IDF specifically comes from Term Frequency (TF) which measures the number of times a word appears while Inverse Document Frequency (IDF) measures the importance of a word from the whole text, the resulting TF-IDF vectorization would be perfect for capturing unique words and the importance of that word in the abstract for classification. The next step is to initialize the logistic regression model imported from sklearn library and provide the parameter maximum iterations. Finally a pipeline is created for an encapsulated and reproducible code.

```
import pandas as pd
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.pipeline import make_pipeline
import numpy as np

# Load preprocessed data
df = pd.read_csv('preprocessed_abstract.csv')

# Split data into feature X and label y
X = df['abstract']
y = df['category']

# Vectorize abstract using TF-IDF
vectorizer = TfidfVectorizer()

# Load the Logistic Regression model
logreg_model = LogisticRegression(max_iter=1000)

# Create a pipeline for model and vectorizer
model_pipeline = make_pipeline(vectorizer, logreg_model)
```

Figure 1.5

Figure 1.6 shows inside the loop of k-fold cross validation. Using the Stratified k-fold method, the fold's distribution is ensured to be randomly shuffled using the parameter shuffle and the training begins for each fold up until 5. The k value can be alternated into other values to redistribute the training and validation set, if the value is changed to 10 the training and validation set split would be a 9:1 ratio. Users can change this value to achieve a different result for cross validation. An early stopping method is implemented inside the loop to ensure the best accuracy is saved by stopping the fold after a determined number of iterations are ran with no improvements in the accuracy factor. At the final line, it collects the predictions and labels for each category

```python
for i, (train_idx, val_idx) in enumerate(StratifiedKFold(n_splits=k, random_state=42, shuffle=True).split(X, y)):
    """
    Perform the k-fold cross-validation using the logistic regression model and evaluates the model's
    performance using metrics and presents the confusion matrix.

    Parameters:
    train_idx: an index of the training set for the fold
    val_idx: an index of the validation set for the fold

    StratifiedKFold: a variation of k-fold
    Parameters:
    n_splits: the number of splits
    random_state: set the random number generator seed to 42 to get the same splits and reproducibility.
    shuffle: shuffles the contents of the dataset
    """

    # Reset early stopping parameters for each fold
    best_accuracy = 0
    no_improvement = 0

    # Create training and validation sets for split i
    X_train_fold, X_val_fold = X.iloc[train_idx], X.iloc[val_idx]
    y_train_fold, y_val_fold = y.iloc[train_idx], y.iloc[val_idx]

    # Train the model on the training set for the current fold with early stopping
    for epoch in range(100):
        model_pipeline.fit(X_train_fold, y_train_fold)
        val_predictions = model_pipeline.predict(X_val_fold)
        val_accuracy = accuracy_score(y_val_fold, val_predictions)

        if val_accuracy > best_accuracy:
            best_accuracy = val_accuracy
            no_improvement = 0
        else:
            no_improvement += 1

        if no_improvement >= max_no_improvement:
            print(f'Early stopping in fold {i + 1} with accuracy {best_accuracy:.4f}')
            break

    # Append predictions and true labels for the current fold
    all_val_predictions.extend(val_predictions)
    all_val_labels.extend(y_val_fold)
```

Figure 1.6

The next step is the calculation of each metric for each fold. All the metrics mentioned such as accuracy, precision, recall, and f1-score are obtained using the sklearn metrics import. Specifically, the accuracy is calculated by the number of correct predictions divided by the total number of predictions.

$$Accuracy = \frac{Correct\ Number\ of\ Predictions}{Total\ Number\ of\ Predictions}$$

The precision is calculated using the true positive over the positive predictions that includes true and false positives, true positive occurs when the model correctly predicts the positive class, while false positive is when the model incorrectly predicts the positive class when it is a negative class. On the other hand, false negative is when the model fails to predict the negative class when it is a positive one.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

The recall is calculated by the true positive over by the true positive and false negative, recall is used to showcase how correctly the model identifies all relevant instances from the true positives.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

Finally the f1-score metric is obtained by two times the precision multiplied by recall over the precision in addition with recall, this produces the mean of the precision and recall.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precession + Recall}$$

After each metric is obtained for each fold it is appended after each fold and kept until the folds are finished, once finished the information for each fold and the metrics are printed. All the metrics are then compiled and shown in a classification report based on the metrics obtained from all the folds. This allows a constructed report for the performance assessment. A confusion matrix is also utilized to show the model's performance across all the categories, using the seaborn library a heatmap of the model's accuracy for each category is created. Each row should add up to 1 or 100 percent, in the experiment the values chosen are percentages for readability.

```
    # Calculate metrics for the current fold
    fold_accuracy = accuracy_score(y_val_fold, val_predictions)
    fold_precision = precision_score(y_val_fold, val_predictions, average='weighted')
    fold_recall = recall_score(y_val_fold, val_predictions, average='weighted')
    fold_f1 = f1_score(y_val_fold, val_predictions, average='weighted')

    # Append metrics for the current fold
    fold_accuracies.append(fold_accuracy)
    fold_precisions.append(fold_precision)
    fold_recalls.append(fold_recall)
    fold_f1_scores.append(fold_f1)

    print(f'Fold {i + 1}: Accuracy: {fold_accuracy:.4f}, Precision: {fold_precision:.4f}, Recall: {fold_recall:.4f}, F1 Score: {fold_f1:.4f}')

# Calculate total confusion matrix
overall_cm = confusion_matrix(all_val_labels, all_val_predictions, labels=df['category'].unique())

# Calculate average from appended metrics
average_accuracy = np.mean(fold_accuracies)
average_precision = np.mean(fold_precisions)
average_recall = np.mean(fold_recalls)
average_f1 = np.mean(fold_f1_scores)

# Display classification report for all folds with original labels
class_labels = ['math', 'physics', 'economics']
print('\nClassification Report - All Folds:\n', classification_report(all_val_labels, all_val_predictions, labels=df['category'].unique(), target_names=class_labels))

# Plot the confusion matrix for all folds with original labels as percentages
plt.figure(figsize=(8, 6))
sns.heatmap(overall_cm / overall_cm.sum(axis=1)[:, np.newaxis], annot=True, fmt='.2%', cmap='Blues', xticklabels=class_labels, yticklabels=class_labels)
plt.title('Overall Confusion Matrix - All Folds (Percentages)')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

Figure 1.7

4. Results

This section will present the results created from the code which includes the information regarding the metrics, and the confusion matrix itself. Figure 2.1 below represents the execution results of each fold from the k-fold cross validation, in the first fold the training stops with the accuracy of 0.98, and all the other metrics rounds up to 0.98. On the second fold the metrics dropped to 0.97 across the board. This might be caused by the shuffled dataset that affects the training process. Amongst the next fold, the results are very similar with 0.97 and 0.98 being the most prominent, however, the last fold has metrics evaluation of 0.95 which is unique from the others.

```
Early stopping in fold 1 with accuracy 0.9889
Fold 1: Accuracy: 0.9889, Precision: 0.9891, Recall: 0.9889, F1 Score: 0.9888
Early stopping in fold 2 with accuracy 0.9778
Fold 2: Accuracy: 0.9778, Precision: 0.9783, Recall: 0.9778, F1 Score: 0.9779
Early stopping in fold 3 with accuracy 0.9889
Fold 3: Accuracy: 0.9889, Precision: 0.9890, Recall: 0.9889, F1 Score: 0.9889
Early stopping in fold 4 with accuracy 0.9778
Fold 4: Accuracy: 0.9778, Precision: 0.9781, Recall: 0.9778, F1 Score: 0.9778
Early stopping in fold 5 with accuracy 0.9500
Fold 5: Accuracy: 0.9500, Precision: 0.9537, Recall: 0.9500, F1 Score: 0.9502
```

Figure 2.1

The next figure shows the classification report gathered from all the folds which shows the average metrics assessments created from the k-fold cross validation. The confusion matrix is created using sklearn's confusion matrix library and the different shades of blue are to represent a heatmap. Since the accuracy of the model is quite high with an average of 98% accuracy, the heatmap doesnt show a gradient of blue colors due to the high accuracies.

```
Classification Report - All Folds:
              precision    recall  f1-score   support

        math       0.98      0.97      0.97       300
     physics       0.96      0.99      0.97       300
   economics       0.99      0.97      0.98       300

    accuracy                           0.98       900
   macro avg       0.98      0.98      0.98       900
weighted avg       0.98      0.98      0.98       900
```

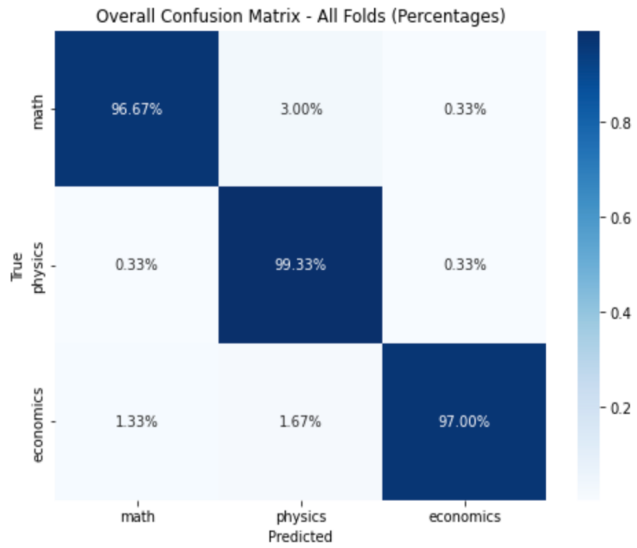


Figure 2.2

5. Conclusion

To summarize what have been done so far, the Logistic Regression model makes us of the preprocessed abstract as the input data to classify the texts into the three predetermined categories. The accuracy of the model from a 5 k-fold cross validation yielded a 98% average accuracy. To change the results of the model's performance, there are a few things that could be altered for the purpose of achieving a better performance across different model such as the amount of k folds that can be increased or decreased. The logistic regression model is mainly chosen due to its simplicity and efficiency in classifying. In other words, there are many other models that could be utilized, and each type of model contains their own excellence and weakness at some parts, such as the number of classes that the model should classify, and the size of datasets to yield a better text classifier.