

AI Project

Automated Plagiarism Detector

Group 7

Namit Shrivastava	2020A2PS1767P
Mohit Gupta	2020A3PS1021P
Sahil Shah	2020A7PS0140P

Problem Statement

Electronic submissions have become increasingly common for student assignments, however this opens the door to more instances of plagiarism. It is relatively simple to copy data from a variety of sources and then paste it into a single work without giving any credit to the original authors of the material. This is made possible by the proliferation of information across the globe through the internet. It's a loss on both fronts as students who engage in such behaviour will not learn as much as they could and authors who put hours of effort into their work are not given due credit. Consequently, there is a requirement for identifying instances of plagiarism in order to heighten and enhance the quality of a student's educational experience and have uniformity in evaluating different students by professors. Detecting instances of plagiarism manually is a challenging and time-consuming endeavour. Because of this, there is a pressing need to develop an automated system that can both detect instances of plagiarism and enhance the overall quality of the educational experience for the student.

PLAGIARISM PLUG-IN PROTOTYPE

Selecting the algorithm for document comparison and incorporating it into a new framework combining local and global search is necessary for developing the prototype for anti-plagiarism plug-in. The goal of this project is to create an anti-plagiarism module that may be used with a variety of assignment types, including but not limited to free-form essays, worksheet-based projects, and even audio podcasts. The following characteristics of the algorithm were chosen for incorporation into an anti-plagiarism add-on in light of this need.

- The programme needs to process various inputs that can be transformed into text.
- The algorithm's one-versus-all check performance needs to be robust even while processing massive data sets.

The Winnowing Algorithm developed by Schleimer, Wilkerson, and Aiken was chosen for use because it best meets these criteria. The approach relies on generating fingerprints (hashes) from the input texts, which are then used to compare the texts. In particular, Winnowing's benefits include:

- One, it takes any form of input and works with it. All that's required is some sort of plain text (like code, an essay, or a podcast script) to feed into the system.
- The second benefit is how quickly it processes massive data volumes. By adjusting the size of the grammar, a fast comparison can be made, and then a more in-depth one can be performed.
- Student work submitted for grading can have the "template fingerprint" removed without any effort.

Proposed System

As part of the proposed system, we plan to create software capable of identifying instances of plagiarism within academic assignments containing visual content. This will reduce instances of students plagiarising the work of their peers, boost the standard of education, and give each student an opportunity to develop his or her own abilities. The similarity between photos is used by the system to identify instances of plagiarism. The suggested approach employs reverse image search to identify plagiarised content.

The integration with Moodle CMS is a primary goal of this add-on (MOODLE is an acronym for "Modular Object-Oriented Dynamic Learning Environment,"). The Figure 1 loosely connected architecture was created to test how well Winnowing and global search function together.

We've built a prototype of the system's software on this architecture. The prototype evaluates student work in light of both a local document database and the results produced by a global search engine. Two main steps are required to complete a local search, while a third is recommended:

1. The system performs tokenization to strip the content of any meaningless characters (such as punctuation, whitespace, etc.)
2. The system computes the fingerprints of the provided document and saves the results to a database.
3. The fingerprint of the worksheet is being scrubbed from the submission if necessary.

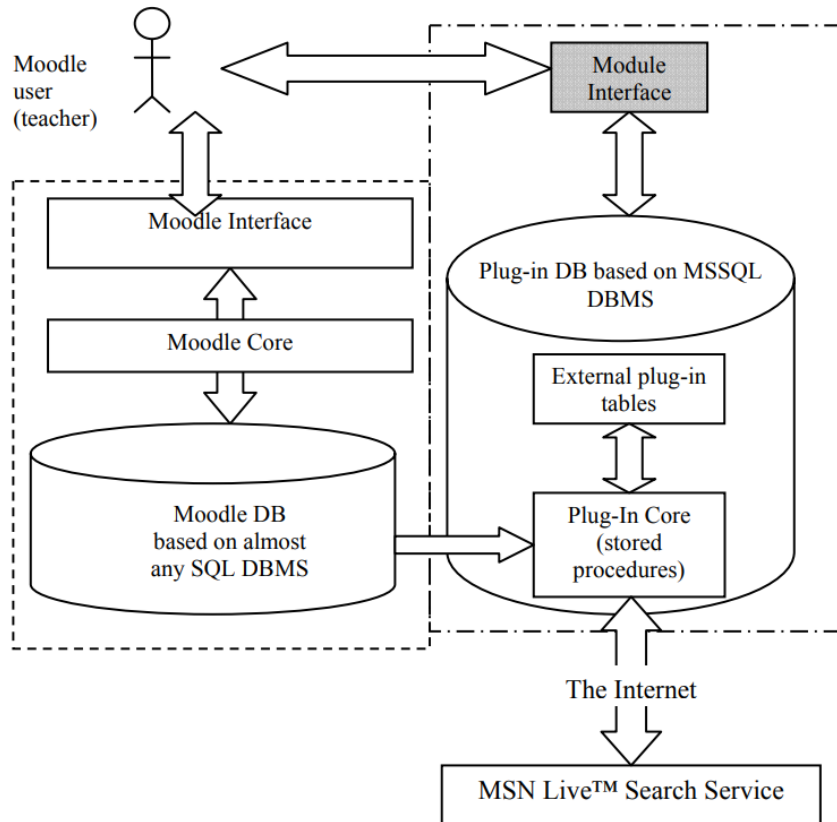


Figure 1. The simplified data flows for software prototype.

The global search requires an additional four steps to complete:

1. Conduct an online search for key terms (or even individual sentences) from the content. At this point, the system establishes a free connection to MSN Live Search (the service developed by Microsoft).
2. The system uses the search results to generate fingerprints of the most relevant documents.
3. Check the input against "local" and "global" document fingerprints.
4. Determine whether or not the work contains "local" or "global" plagiarised content. This choice determines how the supplementary materials are displayed to the educator (like fragments of original and copied paper). Figures 2 and 3 show screenshots containing this data for local and global searches, respectively.

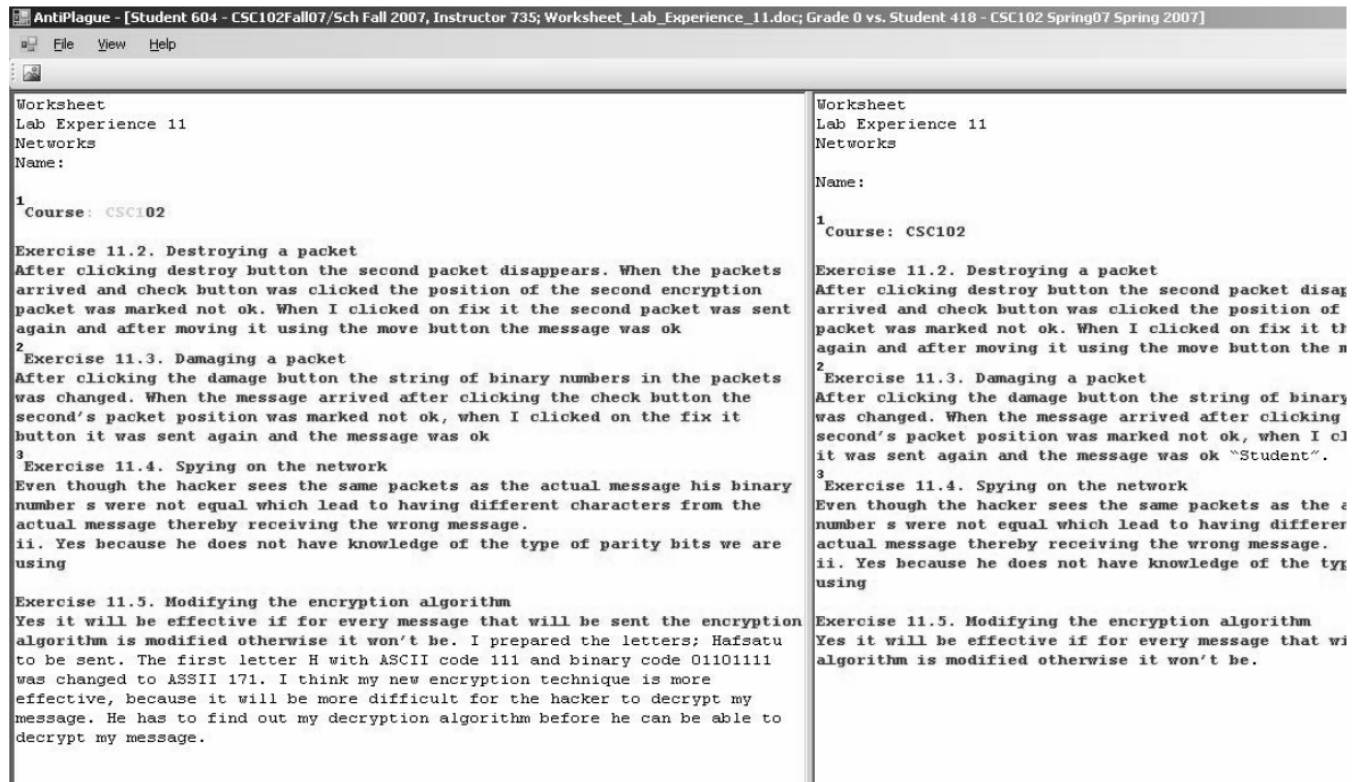


Figure 2. Screenshot comparing the submission with similar document from local database

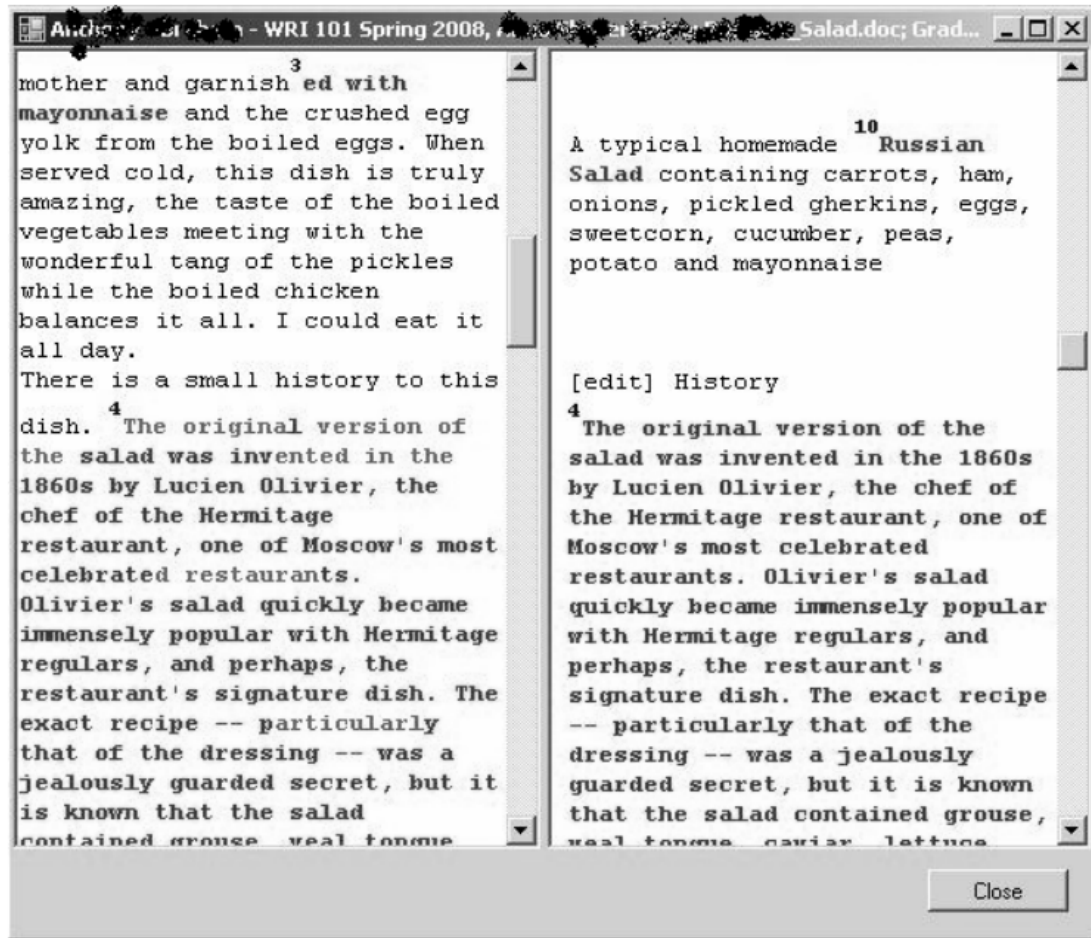


Figure 3. Screenshot comparing the submission with similar document from the Internet.

The prototype that was made doesn't meet all the requirements for Moodle plug-ins because it uses some Microsoft technologies. However, the same algorithms could be easily put into place using PHP, which is what Moodle needs.

The new open architecture for an anti-plagiarism plug-in was made after prototyping showed that it worked well. It can be seen in Figure 2. The main idea behind this architecture is that the anti-plagiarism plug-in should have a tokenization request broker added to it. With this kind of addition, the system will be able to handle almost any kind of soft submission. The broker will figure out what kind of file was sent and then call the right server to tokenize the submission.

As a result of the tokenization process, the system will return a set of symbols (words, sentences, etc.). After this point, it doesn't matter to the anti-plagiarism algorithm what

kind of assignment a student originally turned in, because the algorithm will now work with the plain tokenized text.

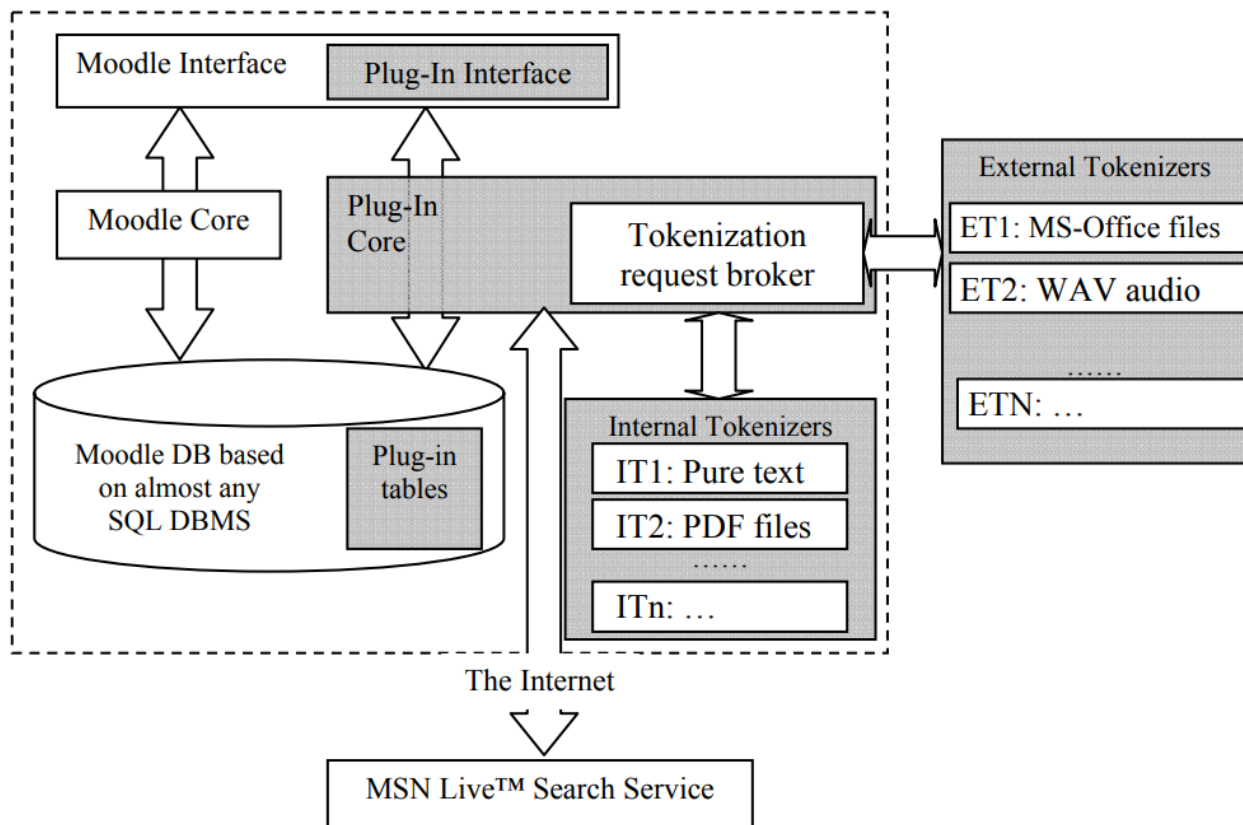


Figure 4. The preliminary architecture with tokenization request broker

Figure 2 shows that something like this will take the project to the next level. If the right tokenizer is connected to the system, the anti-plagiarism module will be able to be used to evaluate audio podcasts. Using speech recognition software, the audio information could be turned into text, which could then be used with the above algorithm.

Working Flow For Image Plagiarism

Collection of assignments

Each student's name and ID number will be gathered digitally along with any assignments or papers they may have submitted. In order to effectively detect instances

of plagiarism. A client-side and server-side validation UI for files (pdf, docs.) is developed. The submitted assignments are temporarily stored on local storage until they can be processed, at which point they are erased.

Pre-processing

In the first, crucial stage, called "pre-processing," all submitted assignments are transformed into the correct format. The format of all assignments has been standardised. After being extracted from the text, the photos are then stored in a database for further use. Only image files in the.jpeg,.png, or.jpg format are gathered; text, numbers, and other symbols are ignored.

Pseudocode for pre-processing –

```
1. OPEN THE FILE PREPROCESSED USING fitz.open
2. FOR EVERY PAGE IN OPENED PDF :
    a. EXTRACT THE LIST OF IMAGES USING getImageList()
    b. FOR EVERY IMAGE IN THE EXTRACTED LIST :
        i. GET THE IMAGE BYTES USING extractImage()
        ii. GET THE IMAGE EXTENSION AND LOAD IT TO PIL
        iii. SAVE THE IMAGE TO LOCAL DISK
```

Hash value computation

Find the hash values of images that are already in the database and save those values. Do the same thing for any new images that are added. The new image is compared to every other image in the database, and it is put into a category based on the percentage of hash values that match.

A-Hash Value Computation

This algorithm works very quickly, but it is not sensitive to changes like scaling the original image, compressing and stretching it, or changing its brightness and contrast. It is based on the average value, so it is sensitive to operations that change this average value (like changing the levels or color balance).

The instructions below detail how to construct an A-Hash:

Reducing the size of the image: The original picture is shrunk down to a more manageable size (usually it is 8x8 or 16x16 points which will show a 64 or 256 bytes respectively).

Image grey-scaling: This change aids in decreasing the hash size by a factor of 3 by describing the number of components from three values of RGB to one level of grey.

Computing the average value: The next step is to determine the average value of all of the picture points.

Simplifying the image: In this system, each pixel reports a value of 0 if its value is below the average and a value of 1 if it is above the mean. As a result, we can think of the image as a collection of bits. The hash is calculated by reading each line sequentially and combining the values.

Pseudocode for hash value computation

```
1. FUNCTION LOAD_IMAGES(FOLDER) - TO EXTRACT THE LIST OF IMAGES IN THE
   GIVEN FOLDER
2. ANSWER LIST = []
   a. FOR EVERY FILE IN THE FOLDER :
       i. IF FILE HAS EXTENSION FROM (.JPEG , .JPG , .PNG)
           1. ADD TO THE ANSWER LIST
   b. RETURN ANSWER LIST

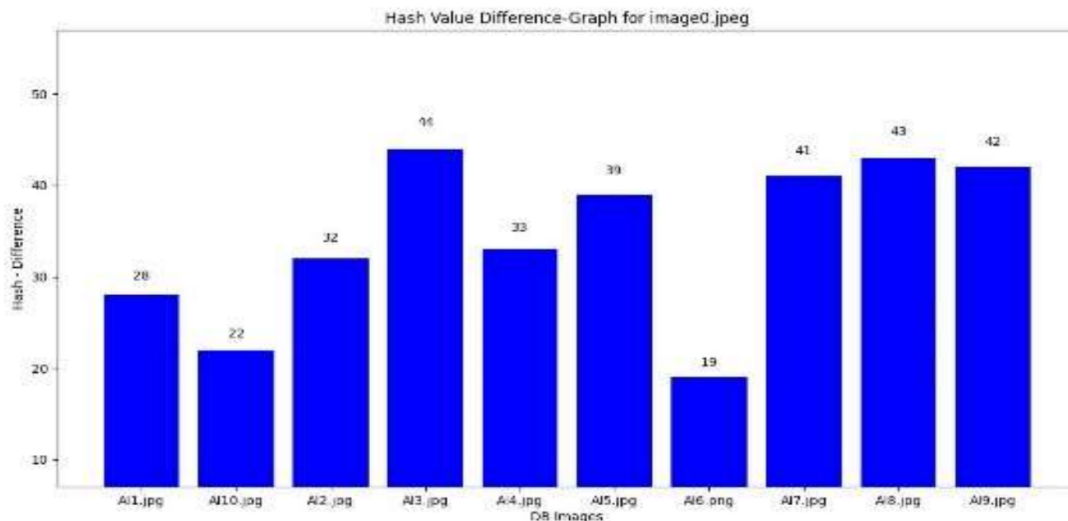
1. EXTRACT THE IMAGES IN THE DATABASE AND USER INPUT USING LOAD_IMAGES
2. ANSWER LIST = []
3. FOR EVERY IMAGE IN USER INPUT :
   a. CALCULATE THE HASH VALUE USING imagehash.average_hash() AS HASH1
   b. FOR EVERY IMAGE IN DATABASE :
       i. CALCULATE THE HASH VALUE USING imagehash.average_hash() AS
          HASH2
       ii. IF HASH1 == HASH2
           1. ADD THE IMAGE IN USER INPUT TO THE ANSWER LIST
```

Comparison and classification: After a hash is computed, it is compared to the hash values of all photos in the database. All photos are checked for similarities using their hash values, and those that match are marked as plagiarised.

Result: Both the pirated text and the plagiarised photos are stored in a Word document and an Excel sheet, respectively. A bar chart can also be used to depict the same set of data.

Excel Sheet: Differences in image hashes between uploaded photographs and those already in the database are recorded in an Excel spreadsheet for further use. A hash difference value for every image in the database is stored in its own Excel sheet.

Bar graph: The difference in hash values and their impact on database pictures is depicted in the following graph. Database image on the x-axis and hash value difference for image0 on the y-axis.



Distinction in hash values for picture 0 displayed as a bar chart

Resultant document: These are the results of a plagiarism scan done on the supplied assignment document, which revealed instances of copied photos. This refers to duplicate photos, which are visually similar to preexisting photographs in a database but have a different hash value. The system has now produced its final result.



Conclusion and Results: Accuracy Scores of Different Classifiers

The code we used to implement our model is [here](#). Its most important snippets help us conclude our findings. In its essence, our code accepts strings in small tokens and compares these small tokens with each other and then with a source file for similarity. The following results show how often the final verdict of the plagiarism checker, based on similarity between tokens, is correct.

The dataset we are using has taken 2 documents and made all possible pairs of sentences. It then makes a moving window of size 3 words and compares the sentences in each pair and based on a threshold, decides if they are plagiarism or not, assigning values 1 and 0 for the same in the column labeled '0'.

We used multiple standard accuracy testing functions such as SVM Classifier, Random Forest Classifier, Decision Classifier, Neural Network Classifier to test aspects of methodology in multiple ways. Its results are given below.

```
SVM Classifier

#SVM Classifier

from sklearn.svm import LinearSVC

clf_SVM = LinearSVC(random_state=0, tol=1e-5, dual = False)
clf_SVM.fit(X_train, y_train)

LinearSVC(dual=False, random_state=0, tol=1e-05)

[79] y_pred = clf_SVM.predict(X_test)

[80] print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])

Accuracy: 0.6805219079765178
```

```
Random Forest Classifier

[19] #Random Forest Classifier

from sklearn.ensemble import RandomForestClassifier

clf_Rf = RandomForestClassifier(max_depth=2, random_state=0)
clf_Rf.fit(X_train, y_train)

RandomForestClassifier(max_depth=2, random_state=0)

[20] y_pred = clf_Rf.predict(X_test)

[21] print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])


Accuracy: 0.61015134468116
```

Decision Trees Classifier

✓ [38] #Decision Trees Classifier

```
from sklearn import tree

clf_DT = tree.DecisionTreeClassifier()
clf_DT = clf_DT.fit(X_train, y_train)
```

✓  y_pred = clf_DT.predict(X_test)
print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])


Accuracy: 0.6339962980437702

Neural Network Based Classifier

✓ [42] #NN Classifier

```
from sklearn.neural_network import MLPClassifier

clf_NN = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 2), random_state=1)
clf_NN = clf_NN.fit(X_train, y_train)
```

✓  y_pred = clf_NN.predict(X_test)
print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])

Accuracy: 0.5025587050411934

Nearest Neighbours

```
[44] #Nearest Neighbours Classifier

from sklearn.neighbors import NearestCentroid

clf_neighbour = NearestCentroid()
clf_neighbour.fit(X_train, y_train)

NearestCentroid()

y_pred = clf_neighbour.predict(X_test)
print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])

Accuracy: 0.5981744274670635
```

Stochastic Gradient Descent

```
[46] #Stochastic Gradient Descent Classifier

from sklearn.linear_model import SGDClassifier

clf_SG = SGDClassifier(loss="hinge", penalty="l2", max_iter=5)
clf_SG.fit(X_train, y_train)

/usr/local/lib/python3.8/dist-packages/sklearn/linear_model/_stochastic_g
warnings.warn(
SGDClassifier(max_iter=5)

y_pred = clf_SG.predict(X_test)
print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])

Accuracy: 0.6325808441911952
```

Naive Bayes Classifier

```
[49] # Naive Bayes Classifier - Took too long to run on our dataset

# Naive Bayes Classifier

from sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
nb.fit(X_train, y_train)

GaussianNB()

y_pred = nb.predict(X_test)
print('Accuracy: ', np.sum(y_pred == y_test)/X_test.shape[0])

Accuracy: 0.609062534025333
```

From the above accuracy (and other trials with different parameter values), it is clear that for our model's training, the best classifier model will be **SVM Classifier**.

Despite having low overall accuracies, we can only assume that this would get better with time invested into this model and makes want to invest more time into this field.

References

<https://www.degruyter.com/document/doi/10.1515/jisys-2014-0146/html?lang=en>

<https://dl.acm.org/doi/abs/10.1145/3345317>

<https://www.irjet.net/archives/V9/i3/IRJET-V9I3197.pdf>

<https://www.ijcaonline.org/research/volume125/number11/naik-2015-ijca-906113.pdf>