Automatic Textile Fabric Defect Detection with Real Time Implementation

Master of Science in Data Analytics

Dublin Business School

Full-time 2022-2023

Candidate Name: Saima Saleem (10603707)

Supervisor name: Satya Prakash

Institute address: 13/14 Aungier St, Dublin, D02 WC04

Ireland

Word Count: 9668

Declaration

I declare that this Applied Research Project that I have submitted to Dublin Business School for the award of (B9DA113) Master of Science in Data Analytics Analytics is my own investigations, except where otherwise stated, where it is clearly acknowledged by references. Furthermore, this work has not been submitted for any other degree.

Signed: Saima Saleem

Student Number: 10603707

Date: 05/01/2023

Acknowledgment

I am thankful to my lectures for giving me the knowledge to work on this project. Furthermore, I would really like to extend my gratitude to my supervisor who gave me the idea and chance to work on this novel project. Without his encouragement, it would have been difficult to select the novel idea and work on it. A special thanks to Peiran Peng, author of Automatic Fabric Defect Detection Method Using PRAN-Net and Kaggle for their support in the data collection.

Signature: Saima Saleem

Date: 05/01/2023

Contents

List of Tables

Title

An automatic fabric defect detection system with high accuracy and high speed for real time implementation in the textile industry using machine learning algorithm.

Abstract

In textile industry, manual fabric quality inspection is a challenging task. A quality inspector keeps an eye on the fabric for several hours. Incomplete and faulty inspection compromises cost and quality of the product. Various complex machine learning algorithms developed so far have certain limitations including real time implementation. An automatic machine learning algorithm is required that should be accurate and fast for real-time detection in industrial setup. This research has successfully developed a simple Convolutional Neural Network machine learning algorithm. It was experimented on three different sizes of images i.e., 200x800, 150x700 and 245x345. It was found out that image size has a great impact on the performance of the model. The developed CNN model gave the best performance with small image size of 245x345. Different sampling techniques were experimented and the best performance was achieved with SMOTENN. A method is proposed to deploy the algorithm and automate the whole process of quality inspection in the textile industry.

Key Words: Fabric defect detection, fabric inspection, deep learning, convolutional neural networks.

1. Introduction

Textile fabric manufacturing process is shown in the fig. 1 below:

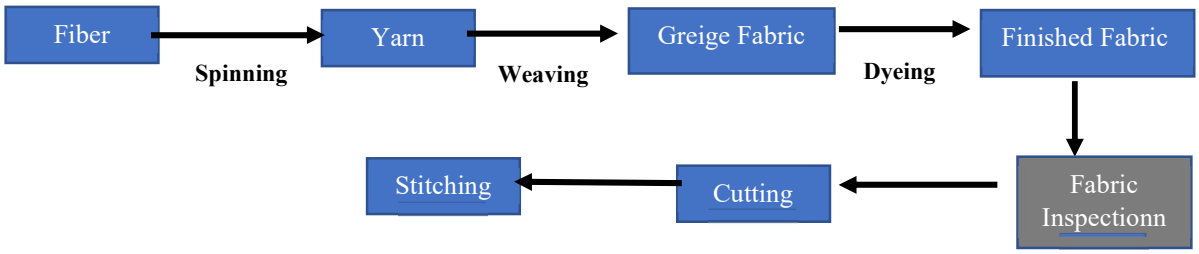


Fig. 1 Fabric manufacturing process

Fabric inspection is done at each step of textile manufacturing process i.e., weaving, dyeing, printing and finishing. This is done to ensure quality of the product to avoid any loss (Li et al., 2021, p. 1). A fabric's structure is repetitive. Hence, a defective region is a destruction of this repetitive structure (Gandelsman, Shocher and Irani, 2019, cited in Wang and Jing, 2020, p. 161318). An early detection of defects is necessary to avoid wastage in the final product (Bullon et al., 2017, cited in Li et al., 2021, p. 1). Jing et al. (2022, pp. 2-3) have divided the task of defect detection into four levels. First is to identify the categories of defects i.e., classification. Second is to detect and locate the defects. Third is to identify the defective pixels in images i.e., segmentation. Fourth level is defect semantic segmentation and that combines segmentation and classification.

2. Literature Review

2.1. Main Theories

Fabric defect detection algorithms are of two types i.e., traditional and learning based algorithms. Traditional algorithms include statistical, spectral, structural, and model based methods. These algorithms deal with a single image and there is manual selection of features. On the other hand, in learning-based methods, feature selection is done automatically. Learning based algorithms are subdivided into classical machine learning algorithms and deep learning algorithms. These methods use mathematical algorithms to learn. Deep Learning methods extract features automatically. However, these methods require a large amount of data as compared to classical machine learning algorithms which in turn need a high amount of parameter settings (Li et al., 2021, p. 1-2; Wang and Jing, 2020, p. 161317).

Therefore, for this research on the fabric defect detection method, working on deep learning algorithms was selected. Significant research work is available on the development of a suitable deep learning-based algorithm but there are various limitations and until now there has not been a system which can be deployed in the industry.

2.2. Data Preprocessing

2.2.1. Imbalance Dataset

The dataset in which classification categories are not equal is an imbalance dataset. Accuracy is not a useful performance measurement in case of imbalanced dataset. Resampling the dataset by different sampling techniques is used to tackle this issue of imbalance dataset (Chawla, V., N. et al., 2002, pp. 322). The dataset should be balanced in which all sample sizes of positive and negative examples are roughly equivalent. This balanced dataset is required to avoid systematic error and bias (Howarth et al. 2020, pp.8).

Data Sampling

Imbalanced dataset can either be absolute or relative. (Letteri et al., 2020, p. 3-4).

- Absolute: minority class samples are less and not well represented.
- Relative: minority samples are well represented but these are larger in number as compared to majority class samples.

Imbalance can be:

- Between-class: number of samples representing a class differs from the number of samples representing the other class.
- Within Class: when a class is composed of several different subclusters which, in turn, do not contain the same number of samples (Letteri et al., 2020, p. 3-4).

Problems of Data Imbalance

- The classifier is biased towards the majority classes.
- Class imbalance hinders the recognition of minority classes since the minority class samples may be insufficient to represent the boundaries between the two classes.
- Imbalanced datasets are more deeply impacted by noisy data

(Letteri et al., 2020, pp. 4)

Resampling is commonly used to adjust the class distribution when dealing with unbalanced datasets (Letteri et al., 2020, pp. 4).

2.2.2. Resampling Techniques

The most common methods are:

- Oversampling means that we increase the number of samples in the minor classes so that the number of samples in different classes become equal or close to it thus get more balanced (Vladimir, 2020).

- Undersampling methods resample the data to reduce some samples from the majority class (Letteri et al., 2020, pp. 5) but this results in removal of useful data samples. To cope with this issue, different techniques are used as:

Instance selection algorithms

In this unimportant samples are eliminated. This algorithm selects a subset of the samples that preserves the underlying distribution, so that the remaining data is still representative of the characteristics of the overall data (Letteri et al., 2020, pp. 5).

Combination Method

Ensemble method combines several weak classifiers to get a better and more comprehensive ensemble classifier (Duan et al., 2020, p.1-2).

Oversampling Methods

Synthetic Minority Oversampling Technique SMOTE

It is a technique to over sample the minority class by creating more examples that are slightly different from the original data points (Chawla, V., N. et al., 2002, pp. 328). SMOTE chooses a minority class instance at random and finds its k-nearest neighbors. One of these k-nearest neighbors is then selected at random (Pahren et al., 2022, pp.75).

Random over-sampling

This method repeats some samples and balances the number of samples. Samples are selected at random from minority classes with replacement. In case of multiple classes, each class is sampled independently (imbalanced-learn, n.d., i).

Borderline SMOTE

This finds borderline samples to generate new synthetic samples. Model learns the borderline of each class in the training process. Borderline of the minority class is determined and then synthetic examples are generated to add to the original training set (Han et al., 2005).

Synthetic Minority Over-sampling Technique for Nominal and Continuous (SMOTENC)

In this method, median of standard deviations of the minority class (continuous) is calculated. Euclidean distance is calculated between minority of one class for which k-nearest neighbors are being identified and the other minority class samples using the continuous feature space (Chawla et al. 2002, pp. 348).

Synthetic Minority Over-sampling Technique for Nominal (SMOTEN)

It is used to resample the categorical data features. The nearest neighbors are computed using the modified version of Value Difference Metric. The Value Difference Metric (VDM) looks at the overlap of feature values over all feature vectors. A matrix defining the distance between corresponding feature values for all feature vectors is created (Chawla et al. 2002, p. 349-351)

SVM SMOTE

It is a kind of SMOTE that uses support vector machine algorithm to select samples. In SVM-SMOTE, borderline is determined by the support vectors after training SVMs classifier on the original training set (Imbalanced learn, n.d., a).

Drawback of SMOTE

The oversampling of SMOTE ignores within-class imbalance. Algorithm does not enforce the decision boundary. Sample instances far from the border are oversampled with the same probability as those close to the boundary (Letteri et al., 2020, pp. 6)

SMOTE produces new samples with certain blindness and may make class overlapping more serious (Duan et al., 2020, pp. 2).

Adaptive Synthetic (ADASYN) algorithm

It is like SMOTE but it generates a different number of samples depending on an estimate of the local distribution of the class to be oversampled. ADASYN adds a random value and the samples are somewhat scattered (Letteri et al., 2020, p. 6-7)

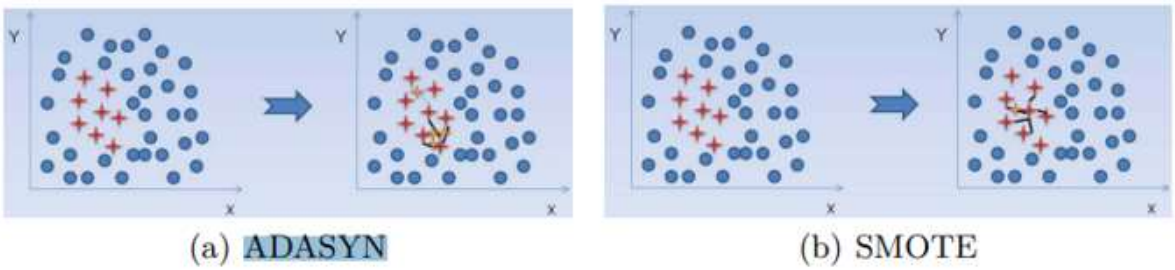


Fig. 2 Difference between ADASYN and SMOTE algorithms (Letteri et al., 2020, p. 7)

It generates minority data samples according to their distributions. Thus, more minority class samples are generated that learn hardly as compared to those minority samples that are easier to learn. The ADASYN method can not only reduce the learning bias introduced by the original imbalance data distribution but can also adaptively shift the decision boundary to focus on those difficult to learn sample (He et al., 2008, pp. 1323).

K-Means SMOTE Oversampling

K-Means SMOTE works in three steps:

1. Cluster the entire input space using k-means.
2. Distribute samples to:
    - Select clusters which have a high number of minority class samples.
    - Assign more synthetic samples to clusters where minority class samples are sparsely distributed.
    3. Oversample each filtered cluster using SMOTE.

The method implements SMOTE and random oversampling as limit cases (kmeans-smote.readthedocs.io, n.d.).

Undersampling

Cluster Centroids

It reduces the majority class by replacing a cluster of majority samples by the cluster centroid of a KMeans algorithm. This algorithm keeps N majority samples by fitting the KMeans algorithm with N cluster to the majority class and using the coordinates of the N cluster centroids as the new majority samples (imbalanced learn, n.d., b).

Condensed Nearest Neighbour

Condensed Nearest Neighbor reduces the dataset for k-NN classification by using a subset of examples (Gowda and Krishna, 1979, pp.488). Nearest neighbor rule decides how to remove a sample or not. The algorithm is runs as given below:

1. Get all minority samples in a set.
2. Add a sample from the targeted class.
3. Classify each sample using nearest neighbor rule.
4. Add a sample if it is misclassified.
5. Repeat the procedure until there are no samples to be added.

(imbalanced learn, n.d. f)

Edited Nearest Neighbor

This method cleans dataset by removing samples close to the decision boundary (imbalanced learn, n.d., c).

Samples are classified using nearest neighbor rule and then classified using single nearest neighbor rule of the pre-classified samples (Wilson, 1972, pp.408).

Repeated Edited Nearest Neighbor

This repeats the Edited Nearest Neighbor many times (imbalanced learn, n.d., d).

AIIKNN

This method applies Edited Nearest Neighbor several times and will vary the number of nearest neighbours (imbalanced learn, n.d. e). AIIKNN differs from the previous Repeated Nearest Neighbor as the number of neighbors of the internal nearest neighbors algorithm is increased at each iteration (imbalanced learn, n.d. e).

Instance Hardness Threshold

Samples of the class with low probabilities are removed from the dataset. Sampling strategy is based on the result. If the result is float, then it is the ratio of majority to minority class. There is a problem of class overlap (imbalanced learn, n.d. f; Sisters, 2020).

NearMiss Undersampling Technique

This method randomly eliminates samples from the larger class. When two points belonging to different classes are very close to each other in the distribution, this algorithm eliminates the samples of the larger class to balance the distribution. The steps taken by this algorithm are:

1. Calculates the distance between all the points in the larger class and compares them with the points in the smaller class.
2. Samples of the larger class that have the shortest distance with the smaller class are selected. These n classes need to be stored for elimination.
3. If there are m instances of the smaller class, then the algorithm will return m*n instances of the larger class.

(Madhukar, 2020)

Tomek Links

Tomek Links is a modification from Condensed Nearest Neighbors. CNN method only randomly selects the samples with its k nearest neighbors from the majority class that are removed. Tomek Links method uses the rule to selects the pair of observation (**a** and **b**) that are fulfilled these properties:

1. The observation **a**'s nearest neighbor is **b**.
2. The observation **b**'s nearest neighbor is **a**.
3. Observations **a** and **b** belong to a different class. That is, **a** and **b** belong to the minority and majority class vice versa.

(Viadinugroho, 2021)

Neighbourhood Cleaning Rule Undersampler

This class uses Edited Nearest Neighbor and a k-NN to remove irrelevant samples from the datasets (imbalanced learn, n.d. g). Neighborhood Cleaning Rule (NCL) is an undersampling method to overcome imbalance class distribution by reducing the data based on cleaning. Cleaning process is improvised by removing the three closest neighbors from the data which are incorrectly classified. The data cleaning process is for both majority and minority class. Basically, the principle of NCL is based on the concept of One-Sided Selection (OSS), which is one technique for reducing data based on the instances to reduce classes carefully. (Agustianto and Destarianto, 2019, pp. 87)

Hybrid Sampling: Oversampling combined with Undersampling

This method can generate noisy samples which is solved by cleaning the space resulting from over-sampling. After SMOTE over-sampling, Tomek's link and edited nearest-neighbours are the two cleaning methods are applied to the pipeline.The two ready-to use classes imbalanced-learn implements for combining over- and undersampling methods are: (i) SMOTETomek and (ii) SMOTEENN (imbalanced learn, n.d. h)

2.3. Variables and Models

Convolutional Neural Network (CNN) Model

Convolutional neural network CNN consists of convolution layers, pooling layers, and fully connected layers. A convolution layer is a basic layer of the CNN which is used for feature extraction (Yamashita et al., 2018, pp. 612).

A Two-dimensional (2D) grid is an array used for storing the pixels of the images (Yamashita et al., 2018, pp. 612).

Kernel is a small grid of parameters that is used for the feature extraction (Yamashita et al., 2018, pp. 612).

As the output of one layer is fed to the next layer, extracted features can hierarchically and progressively become more complex. Training process minimises the difference between the output and ground truth by means of backpropagation and gradient descent. Training process identifies best kernels. Kernels learn automatically during the training process. Hyperparameters that need to be adjusted are the size of the kernels, number of kernels, padding, and stride (Yamashita et al., 2018, pp. 612).

Convolution is a linear operation used for feature extraction. In this process the kernel is applied on the input number array called tensors. A product between each element of the kernel and the input tensor is calculated at each location of the tensor and then summed. The output is called a feature map. Two key hyperparameters are the size and number of kernels. These are usually 3 × 3, 5 × 5 or 7 × 7 (Yamashita et al., 2018, pp. 614).

Padding process reduces height and width of the output feature map by overlapping the centre of each kernel with the outermost element of the input tensor (Yamashita et al., 2018, pp.614).

Stride is the distance between two successive kernels. A stride that is larger than 1 is used to downsample the feature maps (Yamashita et al., 2018, pp.614).

Nonlinear activation function

The outputs of a linear operation such as convolution are passed through a nonlinear activation function. These are sigmoid or hyperbolic tangent (tanh) functions, rectified linear unit (RELU) (Yamashita et al., 2018, pp. 615).

Pooling layer

A pooling layer reduces the dimensionality of the feature maps and decreases the number of subsequent learnable parameters (Yamashita et al., 2018, pp. 615).

Weight Sharing is used for the following purpose:

- To allow the local feature patterns extracted by kernels to travel across all the images for the detection of patterns.
- To learn spatial hierarchies of feature patterns by downsampling in conjunction with a pooling operation, resulting in capturing an increasingly larger field of view.
- To increase model efficiency by reducing the number of parameters to learn in comparison with fully connected neural networks.

(Yamashita et al., 2018)

Max pooling extracts patches from the input feature maps, discards the other and outputs the maximum value in each patch. A max pooling with a filter of size 2×2 with a stride of 2 is commonly used. This down samples the dimension of feature maps by a factor of 2 (Yamashita et al., 2018, p. 615-616)

Fully connected layer

The output feature maps of the final convolution or pooling layer is flattened by means of transformation into a one-dimensional (1D) array of number and then connected to one or more fully connected layers. This fully connected layer is also known as dense layers, in which every input is connected to every output by a learnable weight. The final fully connected layer typically has the same number of output nodes as the number of classes. Each fully connected layer is followed by a nonlinear function, such as RELU (Yamashita et al., 2018, pp. 616).

Last layer activation function

The activation function applied to the last fully connected layer is usually different from the others. An appropriate activation function needs to be selected according to each task. An activation function applied to the multiclass classification task is a SoftMax function which

normalizes output real values from the last fully connected layer to target class probabilities, where each value ranges between 0 and 1 and all values sum to 1 (Yamashita et al., 2018, pp. 616).

Loss function or Cost Function

It measures the compatibility between output predictions of the network through forward propagation and given ground truth labels. Commonly used loss function for multiclass classification is cross entropy (Yamashita et al., 2018, pp. 617).

Gradient descent

It is used to update the learnable parameters kernels and weights of the network to minimize the loss (Yamashita et al., 2018, pp. 617).

Data and ground truth labels

As a famous proverb says, "Garbage in, garbage out", data and ground truth labels are collected with which to train and test a model for a successful deep learning project. Available data are divided into three sets: a training, a validation, and a test set (Yamashita et al., 2018, pp. 617). .

Overfitting

It occurs when a model memorises irrelevant data instead of learning the signal, and, therefore, performs less well on a subsequent new dataset. An overfitted model does not generalize to new data. If the model performs well on the training set compared to the validation set, then the model has likely been overfit to the training data. The best solution for reducing overfitting is to obtain more training data. The other solutions including dropout or batch normalisation and data augmentation as well as reducing architectural complexity. During dropout randomly selected activations are set to 0 during the training, so that the model becomes less sensitive to specific weights in the network. Weight decay reduces overfitting by penalizing the model's weights so that the weights take only small values. Batch normalization is a type of supplemental layer which adaptively normalizes the input values of the following layer, mitigating the risk of overfitting, as well as improving gradient flow through the network,

allowing higher learning rates, and reducing the dependence on initialization. Data augmentation is also effective for the reduction of overfitting, which is a process of modifying the training data through random transformations, such as flipping, translation, cropping, rotating, and random erasing, so that the model will not see the same inputs during the training iterations (Yamashita et al., 2018, p. 619-620).

2.4. Performance Measures

Peng et al. (2020, pp. 7) divided the detection results into true positive (TP), false positive (FP), true negative (TN) and false negative (FN).

Intersection over Union (IoU)

It is calculated by dividing the area of overlap by the area of union between bounding boxes. The area of overlap is between predicted and ground truth bounding box and the area of union is area covered by both bounding boxes (Peng et al., 2020, p. 7).

Precision (p)

It is the ratio of the detected true defects to all detected defects (Peng et al., 2020, p. 7).

Precision= TP/(TP+FP) Precision is how good a model identifies true positives (Chawla, V., N. et al., 2002, pp. 326)

Recall (r)

It is the ratio of the detected true defects to all true defects (Peng et al., 2020, p. 7).

Recall = TP/(TP+FN).

It shows the ability of a model to determine all positive examples (Chawla, V., N. et al., 2002, pp. 326).

Average Recall (AR)

AR is the ratio of the detected true defects of all classes to the true defects of all classes (Peng et al., 2020, p. 7).

Mean Accuracy Precision (mAP)

It is the average of the precision values corresponding to the different recall (Peng et al., 2020, p. 7).

F1 Score

This is the harmonic average of precision and recall (Minhas and Zelek, 2020, p. 510; Zhao et al., 2020, p. 23)

F-value is a combination of recall and precision. F-value is high when both recall and precision are high and can be adjusted through changing the value of β , where β corresponds to relative importance of precision vs. recall and it is usually set to 1.

F-value= $((1+\beta^2)$. Recall. Precision)/(β^2.Recall+Precision) (Han et al., 2005, pp. 880)

AUROC

AUC is the area under curve (AUC) measurement of the receiver operating characteristics (ROC). It measures the degree of the separability of a binary classifier. For an ideal model, its value is 1 (Minhas and Zelek, 2020, p. 510).

Confusion Matrix

Columns represent the predicted class and rows represent actual class.

True Negatives (TN) = number of negative examples correctly classified.

False Positive (FP) = number of negative examples incorrectly classified.

False Negatives (FN)= number of positive examples incorrectly classified.

True Positive (TP)= number of positive examples correctly classified

Accuracy

It is calculated as:

Accuracy = (TP+TN) / (TP+FP+TN+FN)

If the dataset is imbalanced, even when the classifier classifies all the majority examples correctly and misclassified all the minority examples, the accuracy is still high because there are much more majority examples than minority examples. Then accuracy is not a reliable prediction for the minority class (Han et al., 2005, pp. 879).

When dataset is balanced, error rate is used as a performance measurement.

Error Rate = 1- Accuracy (Chawla, V., N. et al., 2002, pp. 322)

When a dataset is imbalanced, error rate is not an appropriate measure of performance. For the imbalanced datasets, precision and recall are the useful measures (Chawla, V., N. et al., 2002, pp. 326).

2.5. Principal Component Analysis

It is a statistical method to reduce the dimensionality of dataset. It determines the direction where the variation in the dataset is maximum. This direction is called "principal component". It determines the different principal components (directions) within the dataset and then uses these principal components to represent the samples. In this way samples can be plotted to check the similarities or differences between the samples. This enables to group the samples. Principal components are the linear combinations (new variables) of the original variables (Ringner, 2008, pp. 303). PCA is a technique for feature extraction. It combines all the variables and then drops less important variables. In this way new independent variables are created. It identifies the relationships between variables and then determines direction of dispersion in the dataset (Brems, 2017).

2.6. Research of Interest

According to Liu et al. (2019, pp. 2), convolution neural network (CNN) models are accurate for the fabric defect detection tasks.

Peng et al. (2020) proposed a Priori Anchor Convolutional Neural Network (PRAN-Net) model that has given a high accuracy for tiny defects. Instead of fixed anchors, selected multi-scale

feature maps Feature Pyramid Network (FPN) were used to generate sparse priori anchors based on fabric defects ground truth boxes in images. A deep residual network ResNet-101-FPN was used for feature extraction.

Almeida, Moutinho and Matos-Carvalho (2021) proposed an operator assisted CNN model based on the fact that undetected defects (False Negative - FN) usually cost higher than non-defective defect being classified as defective (false positive). This model provided an average accuracy of 75% but when assisted by an operator, 95% an accuracy was achieved.

Zhao et al. (2019) designed a visual Long-short-term memory based integrated CNN model that uses visual perception (VP) for classification. Features are extracted by stacked convolutional autoencoders (SCAE). They experimented on three types of datasets i.e. dataset with 500, 1000 and 10500 images and found that the dataset of 500 images gave the highest performance with accuracy of 99.47%, while the accuracy of datasets with 1000 and 1050 images was 97.73% and 95.73%. Image sizes were 224x224. This is a complex method and is sensitive to complex fabric backgrounds.

Zhao et al. (2020) used K-means clustering method called Multi-scale Convolutional Neural Network (MSCNN with K-clustering) to define defect bounding boxes of known sizes. They claimed that the method is fast and accurate due to the application pyramid features method. However, the proposed method is data driven and needs plenty of labelled fabric images to train models. Besides, the developed model is for only five types of fabric defects.

A faster RCNN model consisting of convolution layers, RPN (Region Proposal Network), ROI pooling and classification was designed by Wei et al. (2019). VGG16 was used for feature extraction. For each anchor, the output score of the classification layer of each anchor represents background. The output of the regression layer indicates the coordinate position of the fabric defect. This method suffers from the time limitation as training of the model is a lengthy process.

Chen et al. (2020) proposed an improved faster Faster R-CNN (Faster GG R-CNN) model. This model used Gabor kernels embedded into Faster R-CNN. Training of the model is by a two-stage backpropagation method.

Another Faster RCNN algorithm was proposed by He et al. (2021) that used Convolutional Block Attention Module (CBAM) along with Resnet50. Resnet50 was used for feature extraction, classification, and regression. the extracted feature map was fed in the RoI pooling and RPN. Soft-NMS and RPN remove the extra boxes. This model can be integrated with CNN and can be trained end-to-end together with the basic CNN. The Channel Attention Module compresses the feature map in the spatial dimension to obtain a one-dimensional vector before performing the operation. This method is time consuming.

Zhang et al. (2018) made a comparison between YOLO 9000, YOLO-VOC, and Tiny-YOLO models and devised an improved YOLO-VOC. This model used super-parameter optimization for defect classification and localization. Image prediction was performed by a single stage CNN that increased the speed of detection. However, the proposed method is less accurate as compared to the other methods.

Liu et al. (2018) suggested a method using an improved single shot detector (SSD) with lower convolution feature layer. They combined single detection deep neural networks with the idea of YOLO's regression and anchors principle of faster R-CNN. Regression simplified the complexity and improved the efficiency of the algorithm. Anchors efficiently extracted the characteristics of scales and aspect ratio. SSD used a multiscale object feature extraction method. However, this method could not accurately detect tiny defects.

A special VGG16 model is proposed by Minhas and Zelek (2020) that optimises VGG convolutional network by a deconvolutional network. They used a transfer learning-based mechanism along with CNN to achieve a high accuracy with a small amount of data. The model gave a high accuracy but was time consuming. The study by Minhas and Zelek (2020)

employed two techniques i.e., Fixed Feature Extraction and Full Network Fine Tuning. It was found that the full network fine tuning performed better than the fixed feature extraction approach. In the full network fine tuning, parameters of the entire network are updated using the SoftMax classifier during training. As pretrained weights are good therefore a lower learning rate is used. In this paper, detection speed performance of the model was not studied. According to Tan et al. (2018, cited in Minhas and Zelek, 2020, p. 507), among the four classes of transfer learning i.e., instance-based, mapping-based, network-based and adversarial based, only network-based transfer learning is of practical use. In this case, the target network is built from the source network that is composed of two sub-networks i.e., feature extractor and classification subnetwork. This model can be used for defect classification even if there is less labelled data available.

Liu et al. (2019) emphasised that special VGG16 could be used to detect defects accurately by optimising the VGG convolutional network. A new model called LZFNet was developed by attaching a deconvolutional network to each layer of VGG16 that provided a continuous path back to the image pixel space. The modified model had few parameters. CNN models require a large amount of training labelled data that is difficult to obtain in real world scenarios.

2.7. A Summary of Performance Comparisons of Models from Literature Review

The following picture gives a brief view of the evaluation of these models discussed.

	Intersection Over Union IoU %	mAP %	Acc %	Precision %	Recall %	AR %	F1 Score	Detection Time	Frame Per second	NP Parameter #	Time Loss	Model Size MB	AUROC
PRAN-NET Denim	73	62.3	92			53			9.7				
PRAN-NET Plain	96	93	99			70			25				
CNN-FN Reduction			75	72	88			5.7					
CNN-VLSTM			96-99	96-100	96-100		96-100	1.8-3.5 e2					
MSCNN-K-clustering		94-96							29				
Faster RCNN			96					0.28 sec					
Faster GG-RCNN		95											
Faster RCNN-CBAM		71 Hole-95 Needle-61											
CNN YOLOV2	69			87	88			0.02 sec					
CNN SSD											2.5		
VGG16			97.7							134 M		537	
LZFNet			98					13.8 ms		12 M		51	
CNN+Transfer Learning							89						98

Table 1 Performance comparison of different models from literature review

Where IoU is an intersection over Union, mAP means average precision; AR is average recall, AUROC is the area under curve (AUC) measurement of the receiver operating characteristics (ROC).

All models discussed are slow, time consuming and are not suitable for all types of fabric defects. However, as shown in table 1, CNN based on visual long, short memory has proved better in terms of accuracy and speed, but this is too sensitive to complex fabric structures. Therefore, for this research, CNN model will be researched using different types of sampling techniques.

2.8. Research Gap

Until today, fabric defect detection tasks is done manually. Although a lot of research work is available on defect detection algorithms but so far, almost all of them have limitations like slow speed, not suitable for all kinds of defects due to which the methods could not be deployed in the textile industry.

2.9. Research Question

The aim of this research is to solve the following research question:

“To develop a deep learning method for the detection of fabric defects. Method should have real time application in terms of speed, accuracy, should be fully trained for all types of fabric defects with a balanced approach with low false negative detection rate”.

3. Methodology

The life cycle of a machine learning model consists of six iterative phases as given below:

Business Understanding

This is first and the most important step. In this phase, the business problem is analyzed to determine business objectives and goals.

Data Understanding

Data is checked to find out the inconsistency within data. Segmentation is the task to divide the data into classes.

Data Preparation

Data is prepared for analysis. Selected data undergoes the process of scaling and transformation. New records are generated by combining the information available from multiple tables.

Modelling

This phase involves the task of selecting models which are further analysed by using the techniques of training and testing the datasets.

Evaluation

Evaluation is done to check the quality of the trained models and to find the completeness of tasks.

Deployment

Finally, the developed model is implemented. Model is monitored for its performance.

4. Implementation

A deep learning algorithm based on convolutional neural network (CNN) was used to build model. The dataset was composed of images collected from different sources including the authors of different research papers in the literature review. The model was built on three image sizes i.e. 200 x 800, 150 x700, 245 x 345. This dataset was not balanced. Therefore, different types of sampling techniques were used available from the Imbalanced-Learn module. The sampling technique that performed best was then selected. Accuracy, precision, recall, F1 score and confusion matrix were used as performance measuring metrics. Principal component analysis was used for the data visualization before and after sampling.

4.1. Data Collection

The dataset was collected from the following sources as given below:

Aitex fabric image database (Silvestre-Blanes et al., 2019).

This textile fabric dataset consisted of 245 images of 7 different fabrics with image sizes of 4096×256 pixels. There were 140 defect-free images, 20 for each type of fabric. In each of the defected category, there were 105 images.

Fabric Stain Dataset

This was taken from Kaggle (Pathirana, 2020). The dataset was built as a part of the fabric defect detection project of the Intelligence Lab of University of Moratuwa, Sri Lanka. The dataset consisted of images with resolution of 1920x1080 or 1080x1920. It consisted of 398 defected images with different types of stains and 68 defect free images.

Fabric Defect Dataset from Kaggle (Ranathunga, 2020)

The dataset was also taken from Kaggle and it was supplied by the Intelligence Lab, Department of computer science and Engineering, University of Moratuwa.

This dataset consisted of 3 classes of defective images namely horizontal, vertical and holes along with 3 mask images for each defective image sample. The folder named 'captured' consisted of raw images. Images have a size of 640x360.

Dataset from the author of the literature review paper (Peng et. al, 2020)

Authors of the research papers in the literature review were contacted through emails for the dataset. With the consent of the author of the paper "Automatic fabric defect detection method using pran-net" (Peng, 2020) named Troy Peng, the annotated data was used.

Thus, the dataset collected from all the sources consisted of the images from the following:

- **Fabric Defect Dataset** from Kaggle (Ranathunga, 2020)
- **Fabric Stain Dataset** from Kaggle (Pathirana, 2020).
- **Aitex fabric image database** (Silvestre-Blanes et al., 2019)
- **Dataset from the author of the literature review paper** (Peng, 2020): only the non-defect images from the dataset.

4.2. Dataset

After collecting the images from all these sources, the distribution of the images is as given below:

- Defect free 417
- Defected hole 281
- Defected horizontal 136
- Defected lines images 157
- Defected stain images 398
- Defected verticle images 101

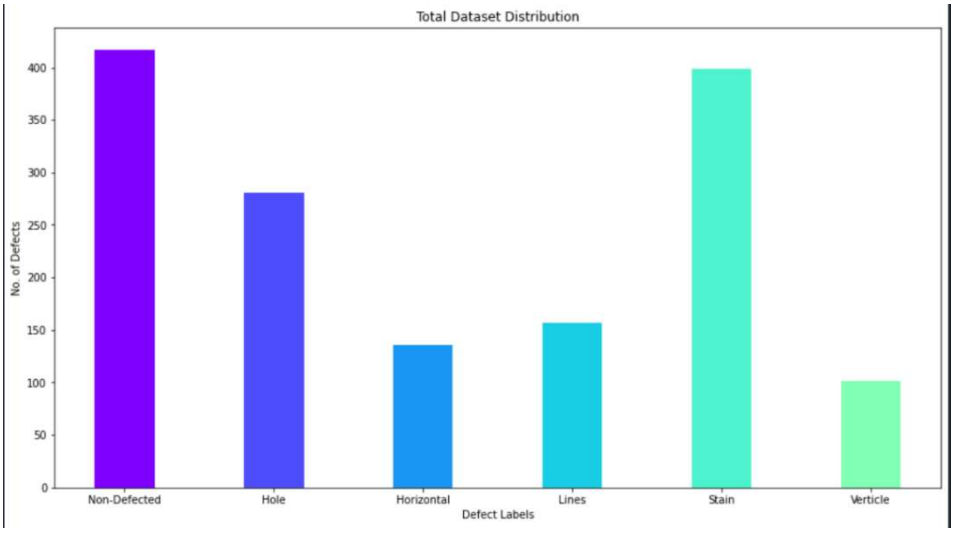


Fig. 3 Number of images in each class of the dataset

Total number of defected images = 1073

Total number of non-defected images = 417

Total Images = 1,490

The dataset was loaded on the Google Drive and the link is given below:

https://drive.google.com/drive/folders/1TjR9E86cab-W_e7enYmc_IxHgR2gExRF?usp=sharing

4.3. Data Preprocessing

The collected dataset was not balanced. Therefore, different types of sampling techniques available in imbalanced-learn module were experimented to select the best sampling technique with high performance.

4.4. Modelling

CNN model was built for multi classification problem that consisted of the five two dimensional convolutional layers for feature extraction.

The first convolutional layer consisted of a set of 16 filters width and length size of 7 referred to as kernel size.

The second convolutional layer consisted of a set of 32 filters of kernel size width 5 and length 5.

The third and fourth layers consisted of a set 64 filters with kernel size of 3x3.

The fifth layer consisted of a set of 128 filters with kernel size of 7x7.

Padding with hyperparameter same was employed to facilitate the filter kernels to move across the images with the addition of pixels. The use of same as hyperparameter in padding enabled to keep the size of the output feature map equal to the input feature map.

Each successive convolutional layer was followed by batch normalization, activation function, pooling (max pooling) and dropout layers. To maximize training and learning speed, Batch normalization was used to standardise and normalise the dataset in batches.

A Rectified linear activation function RELU was used because CNN was multi-layered. Sigmoid or tangent activation could not work for the multi-layered CNN due to vanishing gradient problem. RELU function works by multiplying the input images by the weights and then those are summed together to give the specific output.

Max pooling was used to reduce the size of the images and for downsampling to create the lower resolution images with main features. Max pooling was applied with filter shape of 2 x 2 because selected size should be less than than the size of the feature map. This pooling layer created new feature maps. To avoid overfitting, dropout layers were added.

In the final stage of the model, a fully connected layer consisting of a dense layer was used to match the output from the pooling layers with the labels of the dataset. Because there were many pooling layers employed, therefore, to get these into a sequential manner in the form of a vector of inputs, a flattening function layer was used. After passing through all the convolutional layers, the output was in the form of a multidimensional array that should be passed through a dense layer. To interpret the result in the form of probability distribution, a SoftMax function was used in the fully connected layer.

4.5. Model Validation

Training and testing were performed at 80:20. Epochs used were 100 with a batch size of 16.

Evaluation metrics used are accuracy, precision, recall, F1 score, confusion matrix.

For data visualisation before sampling and after sampling, dimension reduction technique was performed using principal component analysis.

Three sizes are selected i.e., 200x800, 150x700, 245x345.

5. Results

Modelling was performed for the three selected sizes of images dataset without any sampling technique. The results were as given in table 2.

Sizes	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
200x800 (Appendix 1-1)	0..81	0.84	0.84	0.84	1,373,638	1,372,902	736	0.58	0.08	8
150x700 (Appendix 1-2)	0.89	0.88	0.88	0.88	832,966	832,230	736	0.18	0.07-0.1	6.65
245x345 (Appendix 1-3)	0.85	0.86	0.84	0.85	718,278	717,542	736	0.17	0.06	4.75

Table 2 Performance of the model experimented with the three sizes of the images in dataset

The link to the folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/16nup58ZMk8kJ9g3fOxfIx6TfoLdPoEbR?usp=sharing

5.1. Data Preprocessing Implementation

Oversampling (Appendix 3)

The link to the oversampling folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/1b3bP8USc2V9WepNLWCHnngxdBqD3W0Fb?usp=share_link

Results Oversampling Size 200x800

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	PredictS	ValidS
ADASYN App. 3-1.1	0.83	0.85	0.84	0.74	1,373,64	1,372,902	736	0.5	0.08	10.55
Random Oversampling Appendix 3-2.1	0.89	0.89	0.89	0.863	1,373,64	1,372,902	736	0.46	0.07	13.37
Borderline SMOTE Appendix 3-3.1	0.89	0.90	0.89	0.88	1,373,64	1,372,902	736	0.4	0.08	13.40
KMeans SMOTE Appendix. 3-4.1	0.96	0.96	0.96	0.96	1,373,64	1,372,902	736	0.5	0.07	18.27
SMOTENN Appendix . 3-5.1	0.82	0.85	0.81	0.82	1,373,64	1,372,902	736	0.43	0.078	16.8
SMOTENC Appendix .3-7.1					1,373,64	1,372,902	736			
SVMSMOTE Appendix 3-6.1	0.87	0.88	0.88	0.87	1,373,64	1,372,902	736	0.58	0.07	18.22

Table 2.1 Performance metrics of Oversampling Techniques for the size 200x800

Oversampling Size 150x700

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	ValidS
ADASYN Appendix. 3-1.2	0.88	0.90	0.89	0.89	832,966	832, 230	736	0.195	0.06-0.07	6.7
Random Oversampling Appendix 3-2.2	0.92	0.92	0.92	0.92	832,966	832, 230	736	0.53	0.07	8.88
Borderline SMOTE Appendix 3-3.2	0.95	0.95	0.95	0.95	832,966	832, 230	736	0.75	0.07-0145	11.13
KMeans SMOTE Appendix 3-4.2	0.92	0.92	0.92	0.92	832,966	832, 230	736	0.68	0.06-0.07	9.2
SMOTENN Appendix 3-5.2	0.88	0.90	0.87	0.88	832,966	832, 230	736	0.54	0.06-0.07	12.6
SMOTENC Appendix 3-7.2	0.96	0.96	0.96	0.96	832,966	832, 230	736	0.65	0.06-0.07	8.95
SVMSMOTE Appendix 3-6.2	0.95	0.95	0.95	0.95	832,966	832, 230	736	0.57	0.06-0.07	8.87

Table 2.2 Performance metrics of Oversampling Techniques for the size 150x700

Oversampling Size 245x345

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
ADASYN Appendix 3-1.3	0.85	0.84	0.87	0.85	718,278	717,542	736	0.16	0.067-0.09	5.2
Random Oversampling Appendix 3-2.3	0.95	0.95	0.95	0.95	718,278	717,542	736	0.25	0.06 0.07	10.15
Borderline SMOTE Appendix 3-3.3	0.91	0.91	0.91	0.91	718,278	717,542	736	0.41	0.07	7,6
KMeans SMOTE Appendix 3-4.3	0.95	0.95	0.95	0.95	718,278	717,542	736	0.46	0.065	7
SMOTENN Appendix 3-5.3	0.77	0.82	0.77	0.77	718,278	717,542	736	0.36	0.14-0.09	13.94
SMOTENC Appendix 3-7.3	0.86	0.88	0.85	0.84	718,278	717,542	736	0.52	0.06	8.05
SVMSMOTE Appendix 3-6.3	0.95	0.95	0.95	0.95	718,278	717,542	736	0.283	0.065-0.1	8

Table 2.3 Performance metrics of Oversampling Techniques for the size 245x345

Undersampling (Appendix 2)

The link to the undersampling folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/1F9SBxMXpmYYK_UjqifzkFt9DHibUypWX?usp=sharing

Undersampling Size: 200X800

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	PredictS	ValidS
Cluster Centroids Appendix 2-1.1	0.79	0.72	0.75	0.72	1,373,638	1,372,902	736	0.36	0.07	3.77
Condensed Nearest Neighbor Appendix 2-2.1	0.75	0.86	0.63	0.63	1,373,638	1,372,902	736	0.18	0.08	2.37
Edited Nearest Neighbour Appendix 2-3.1	0.92	0.91	0.90	0.91	1,373,638	1,372,902	736	0.28	0.08	4.85
Repeated Edited Nearest Neighbour Appendix 2-4.1	0.81	0.85	0.78	0.77	1,373,638	1,372,902	736	0.18	0.08	4.42
AIKNN Appendix 2-5.1	0.81	0.83	0.83	0.82	1,373,638	1,372,902	736	0.99	0.07-0.24	5.83
Instance Hardness Threshold Appendix 2-6.1	0.93	0.94	0.94	0.94	1,373,638	1,372,902	736	0.3	0.08-0.09	3.99
Near Miss Appendix 2-7.1	0.88	0.90	0.86	0.87	1,373,638	1,372,902	736	0.29	0.08	3.9
Neighborhood Cleaning Rule Appendix 2-8.1	0.72	0.78	0.74	0.73	1,373,638	1,372,902	736	0.18	0.09-0.1	7.7
Random Under Sampler Appendix 2-10.1	0.77	0.81	0.77	0.77	1,373,638	1,372,902	736	0.28	0.08	4.3
Tomek Links Appendix 2-9.1	0.84	0.86	0.85	0.5	1,373,638	1,372,902	736	0.56	0.08-0.25	9.05

Table 3.1 Performance metrics of Undersampling Techniques for the size 200x800

Undersampling Size 150x700

Sizes 150x700	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	models	Predicts	Valids
Cluster Centroids Appendix 2-1.2	0.80	0.81	0.79	0.79	832,966	832,230	736	0.26	0.07	2.73
Condensed Nearest Neighbour Appendix 2-2.2	0.65	0.55	0.55	0.53	832,966	832,230	736	0.2	0.09-0.12	1.66
Edited Nearest Neighbour Appendix 2-3.2	0.92	0.92	0.93	0.92	832,966	832,230	736	0.46	0.06-0.08	5.06
Repeated Edited Nearest Neighbour Appendix 2-4.2	0.94	0.94	0.96	0.95	832,966	832,230	736	0.18	0.06-0.07	3.43
AIKNN Appendix 2-5.2	0.96	0.96	0.95	0.96	832,966	832,230	736	0.4	0.07-0.2	4
Instance Hardness Threshold Appendix 2-6.2	0.95	0.96	0.94	0.95	832,966	832,230	736	0.18	0.06-0.07	2.5
Near Miss Appendix 2-7.2	0.68	0.79	0.68	0.68	832,966	832,230	736	0.17	0.065-0.068	2.28
Neighborhood Cleaning Rule Appendix 2-8.2	0.88	0.90	0.89	0.89	832,966	832,230	736	0.38	0.07	4.9
Random Under Sampler Appendix 2-10.2	0.71	0.73	0.70	0.68	832,966	832,230	736	0.19	0.07	3.2
Tomek Links Appendix 2-9.2	0.77	0.78	0.80	0.77	832,966	832,230	736	0.29	0.07-0.2	10.55

Table 3.2 Performance metrics of Undersampling Techniques for the size 150x700

Undersampling Size 245x345

Sizes 245x345	Accuracy	Precision	Recall	F1	Parameters			Model	Predict	Valid
					Total	Trainable	Non-trainable	sec	sec	sec
Cluster Centroids Appendix 2-1.3	0.70	0.71	0.70	0.69	718,278	717,542	736	0.18	0.06-0.07	1.9
Condensed Nearest Neighbour Appendix 2-2.3	0.63	0.57	0.53	0.52	718,278	717,542	736	0.17	0.06-0.07	1.27
Edited Nearest Neighbour Appendix 2-3.3	0.94	0.94	0.95	0.94	718,278	717,542	736	0.18	0.06	2.55
Repeated Edited Nearest Neighbour Appendix 2-4.3	0.96	0.96	0.96	0.96	718,278	717,542	736	0.17	0.06	2.42
AIKNN Appendix 2-5.3	0.94	0.94	0.93	0.93	718,278	717,542	736	0.165	0.07	3.3
Instance Hardness Threshold Appendix 2-6.3	0.90	0.91	0.90	0.90	718,278	717,542	736	0.17	0.06-0.07	2.1
Near Miss Appendix 2-7.3	0.70	0.80	0.71	0.73	718,278	717,542	736	0.19	0.06	1.94
Neighborhood Cleaning Rule Appendix 2-8.3	0.90	0.90	0.91	0.90	718,278	717,542	736	0.17	0.065	3.87
Random Under Sampler Appendix 2-10.3	0.75	0.75	0.74	0.72	718,278	717,542	736	0.27	0.06-0.07	2.65
Tomek Links Appendix 2-9.3	0.83	0.86	0..87	0.85	718,278	717,542	736	0.17	0.06-0.2	4.9

Table 3.3 Performance metrics of Undersampling Techniques for the size 245x345

Combination of Oversampling and Undersampling Size 200x800 (Appendix 4)

The link to the folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/1oi6M0aC9bbq6w0lE7CXiOPh6n-Ij_BiV?usp=sharing

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	PredictS	Valids
SMOTEENN Appendix 4-1.1	0.98	0.98	0.98	0.98	1,373,638	1,372,902	736	0.48	0.07	10.33
SMOTETomek Appendix 4-2.1	0.93	0.93	0.93	0.93	1,373,638	1,372,902	736	0.72	0.07-0.09	12.97

Table 4.1 Performance metrics of SMOTEENN and SMOTETomek for the size 200x800

Combination of Oversampling and Undersampling Size 150x700

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Model S	Predict S	Valid s
SMOTEENN Appendix 4-1.2	0.97	0.97	0.98	0.97	832,966	832,2390	736	0.15	0.12-0.13	8.29
SMOTETomek Appendix 4-2.2	0.89	0.91	0.89	0.89	832,966	832,2390	736	0.48	0.065-0.07	8.73

Table 4.2 Performance metrics of SMOTEENN and SMOTETomek for the size 150x700

Combination of Oversampling and Undersampling Size 245x345

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	PredictS	Valids
SMOTEENN Appendix 4-1.3	0.99	1.00	0.99	0.99	718,278	717,542	736	0.15	0.06-0.07	7.68
SMOTETomek Appendix 4-2.3	0.95	0.95	0.95	0.94	718,278	717,542	736	0.43	0.06	7.9

Table 4.3 Performance metrics of SMOTEENN and SMOTETomek for the size 245x345

Ensemble (Appendix 5)

The link to the Ensemble folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/1w_Cg8W23IHCdvc81IXBU8tkHJYY11S7X?usp=sharing

Ensemble Size 200 x800

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Pipeline Appendix 5-1.1	0.97	0.98	0.97	0.97	1,373,638	1,372,902	736	0.58	0.08	9.8

Table 5.1 Performance metrics of Pipeline for the size 200x800

Ensemble Size 150 x700

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Pipeline Appendix 5-1.2	0.95	0.94	0.92	0.93	832,966	832,230	736	0.4		6.62

Table 5.2 Performance metrics of Pipeline for the size 150x700

Ensemble Size 245 x345

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Pipeline Appendix 5-1.3	92	0.93	0.91	0.91	718,278	717,542	736	0.22	0.06	5.56

Table 5.3 Performance metrics of Pipeline for the size 245x345

Miscellaneous (Appendix 6)

The link to the folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/110f6tfFU_8UY5qtJRkePAtPXmVPwmklW?usp=sharing

Miscellaneous Size 200x800

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Function Sampling Appendix 6-1.1	0.69	0.71	0.76	0.70	1,373,638	1,372,902	736	0.19	0.08	8

Table 6.1 Performance metrics of Function Sample for the size 200x800

Miscellaneous Size 150x700

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Function Sampling Appendix 6-1.2	0.89	0.84	0.91	0.87	832,966	832,2302	736	0.16	0.07	5.2

Table 6.2 Performance metrics of Function Sample for the size 150x700

Miscellaneous Size 245x345

	Accuracy	Precision	Recall	F1	Parameters			Time		
					Total	Trainable	Non-trainable	Models	Predicts	Valids
Function Sampling Appendix 6-1.3	0.71	0.71	0.80	0.72	718,278	717,542	736	0.26	0.06	3.55

Table 6.3 Performance metrics of Function Sampler for the size 245x345

6. Discussion

As shown in table 2, for all the three sizes, CNN model could not perform well and accuracy was below 90%. However, the performance of 150x700 size was better than the large size of 200x800 and very small size of 245x345. Among all sizes, the performance of 200x800 was inferior as compared to the other two sizes.

Oversampling

Oversampling			
	200x800	150x700	245x345
ADASYN	0.83	0.88	0.85
Random Oversampling	0.89	0.92	0.95
Borderline SMOTE	0.89	0.95	0.91
KMeans SMOTE	0.96	0.92	0.95
SMOTEN	0.82	0.88	0.77
SMOTENC		0.96	0.86
SVMSMOTE	0.87	0.95	0.95

Table 7.1 Accuracy comparison of the three sizes for Oversampling

As shown in table 7.1, large size 200x800 image could not perform well in terms of accuracy, precision, recall, F1, confusion matrix.

SMOTENC could not run due to memory size limitation. Performance of medium sized 150x700 and small size images 245x345 was almost equal and in fact, was much better than large sized images.

Performance of KMeans Smote was much better performance in case of all the three sizes. SVM SMOTE stood second as compared to KMeans SMOTE.

Performance of ADASYN and SMOTEN was not good.

Random oversampling and Borderline smote stood third in the list and their performance was quite similar for both the sizes i.e., 150x700 and 245x345.

As shown in table 7.2, validation time for 200x800 was the highest in case of all sampling as compared to 150x700 and 245x345. 150x700 took high validation time in case of SMOTEN and Borderline SMOTE. In case of 245x345, validation time remained the highest for

SMOTEN and Random Sampling. SMOTEN validation time remained the highest in case of all the three sizes.

	245x345			150x700			200x800		
	Models	**Predicts**	**Valids**	**Models**	**Predicts**	**Valids**	**Models**	**Predicts**	**Valids**
ADASYN	0.16	0.067-0.09	5.2	0.195	0.06-0.07	6.7	0.5	0.08	**10.55**
Random Oversampling	0.25	0.06 0.07	**10.15**	0.53	0.07	8.88	0.46	0.07	**13.37**
Borderline SMOTE	0.41	0.07	7,6	0.75	0.07-0145	**11.13**	0.4	0.08	**13.40**
KMeans SMOTE	0.46	0.065	7	0.68	0.06-0.07	9.2	0.5	0.07	**18.27**
SMOTEN	0.36	0.14-0.09	**13.94**	0.54	0.06-0.07	**12.6**	0.43	0.078	**16.8**
SMOTENC	0.52	0.06	8.05	0.65	0.06-0.07	8.95			
SVMSMOTE	0.283	0.065-0.1	8	0.57	0.06-0.07	8.87	0.58	0.07	**18.22**

Table 7.2 Time Comparison of the three sizes for Oversampling

Undersampling

Undersampling			
	200x800	150x700	245x345
Cluster Centroids	0.79	0.80	0.70
Condensed Nearest Neighbour	0.75	0.65	0.63
Edited Nearest Neighbour	0.92	0.92	0.94
Repeated Edited Nearest Neighbour	0.81	0.94	0.96
AIKNN	0.81	0.96	0.94
Instance Hardness Threshold	0.93	0.95	0.90
Near Miss	0.88	0.68	0.70
Neighborhood Cleaning Rule	0.72	0.88	0.90
Random Under Sampler	0.77	0.71	0.75
Tomek Links	0.84	0.77	0.83

Table 8.1 Accuracy comparison of the three sizes for undersampling

Like oversampling, as shown in table 8.1, performance of undersampling technique for the 200x800 size remained inferior as compared to 150x700 and 245x345 sizes. Performance of 150x700 size was quite like and in fact, almost equal to 245x345.

Performance of Instance Hardness Threshold was the best amongst all the undersampling techniques. Edited Nearest Neighbour, Repeated Edited Nearest Neighbour and AIKNN stood second and third respectively. Performance of Edited Nearest Neighbour and AIKNN was almost similar, Both Edited Nearest Neighbour and AIKNN performed well for 150x700 and

245x345 as compared to 200x800 in terms of accuracy, precision, recall, F1 and also in terms of validation time.

As shown in table 8.2, validation time of 200x800 was the highest as compared to 150x700 and 245x345.

Tomek link's validation time was the highest amongst all the undersampling techniques.

	245x345			150x700			200x800		
	Models	**Predicts**	**Valids**	**Models**	**Predicts**	**ValidS**	**Models**	**Predicts**	**Valids**
Cluster Centroids	0.18	0.06-0.07	1.9	0.26	0.07	2.73	0.36	0.07	**3.77**
Condensed Nearest Neighbour	0.17	0.06-0.07	1.27	0.2	0.09-0.12	1.66	0.18	0.08	**2.37**
Edited Nearest Neighbour	0.18	0.06	2.55	0.46	0.06-0.08	**5.06**	0.28	0.08	**4.85**
Repeated Edited Nearest Neighbour	0.17	0.06	2.42	0.18	0.06-0.07	**3.43**	0.18	0.08	**4.42**
AIKNN	0.165	0.07	**3.3**	0.4	0.07-0.2	4	0.99	0.07-0.24	**5.83**
Instance Hardness Threshold	0.17	0.06-0.07	2.1	0.18	0.06-0.07	2.5	0.3	0.08-0.09	**3.99**
Near Miss	0.19	0.06	1.94	0.17	0.065-0.068	2.28	0.29	0.08	**3.9**
Neighborhood Cleaning Rule	0.17	0.065	**3.87**	0.38	0.07	4.9	0.18	0.09-0.1	**7.7**
Random Under Sampler	0.27	0.06-0.07	2.65	0.19	0.07	3.2	0.28	0.08	**4.3**
Tomek Links	0.17	0.06-0.2	**4.9**	0.29	0.07-0.2	**10.55**	0.56	0.08-0.25	**9.05**

Table 8.2 Time Comparison of the three sizes for Undersampling

Combined Undersampling and Oversampling

As shown in table 9.1, performance of all the three sizes remained almost similar.

Both SMOTEENN and SMOTETomek performed well as compared to Oversampling and Undersampling alone. SMOTENN performance was better as compared to SMOTETomek.

Undersampling & Oversampling			
	200x800	150x700	245x345
SMOTEENN	**0.98**	**0.97**	**0.99**
SMOTETomek	0.93	0.89	0.95

Table 9.1 Accuracy comparison of three sizes for SMOTEENN and SMOTETomek

	245x345			150x700			200x800		
	Models	**Predicts**	**Valids**	**ModelS**	**PredictS**	**Valids**	**ModelS**	**Predicts**	**ValidS**
SMOTEENN	0.15	0.06-0.07	7.68	0.15	0.12-0.13	8.29	0.48	0.07	**10.33**
SMOTETomek	0.43	0.06	7.9	0.48	0.065-0.07	8.73	0.72	0.07-0.09	**12.97**

Table 9.2 Time Comparison of the three sizes for SMOTEENN and SMOTETomek

Validation time was the highest for 200x800 while it is almost equal in case of 150x700 and 245x345 as shown in table 9.2.

Validation time for both SMOTEENN and SMOTETomek was the same for both 150x700 and 245x345.

Ensemble

Ensemble			
	200x800	150x700	245x345
Pipeline	0.97	0.95	0.92

Table 10.1 Accuracy comparison of three sizes of Pipeline

	245x345			150x700			200x800		
	Models	**Predicts**	**Valids**	**ModelS**	**Predicts**	**Valids**	**Models**	**PredictS**	**Valids**
Pipeline	0.22	0.06	5.56	0.4		6.62	0.58	0.08	**9.8**

Table 10.2 Time Comparison of Pipeline

Pipeline performed well in case of all the three sizes as shown in table 10.1.

Performance of Pipeline was the best and was like SMOTENN and SMOTomek. However, validation time taken by Pipeline is lesser and much better as compared to SMOTENN and SMOTETomek as shown in table 10.1.

Miscellaneous

Miscellaneous			
	200x800	150x700	245x345
Function Sampler	0.69	0.89	0.71

Table 11.1 Accuracy of Function Sampler

	245x345			150x700			200x800		
	Models	**Predicts**	**Valids**	**Models**	**PredictS**	**Valids**	**ModelS**	**PredictS**	**ValidS**
Function Sampler	0.26	0.06	3.55	0.16	0.07	5.2	0.19	0.08	8

Table 11.2 Time Comparison of Function Sampler

Performance of the Function sampler was similar and almost equal in case of all the three sizes as shown in table 11.1. In terms of validation time, 245x345 performance was better as compared to large and medium sized images as shown in table 11.2.

6.1. Results Verification

From the above experiments, it was found out that the CNN model gave the best performance with SMOTEENN sampling. Although the performance was good with all the three sizes i.e. 200x800, 150x700 and 245 x345 but the best was with 245x345 image size. Therefore, this model was further selected to run for at least five times to check the consistency of the results (appendix 7). Table 12 shows the results obtained from running the model five times.

	Accuracy	Precision	Recall	F1 Score
Model Run 1 Appendix 7.1	0.99	0.98	0.99	0.98
Model Run 2 Appendix 7.2	0.98	0.97	0.97	0.97
Model Run 3 Appendix 7.3	0.99	0.99	0.99	0.99
Model Run 4 Appendix 7.4	0.99	0.99	0.99	0.99
Model Run 5 Appendix 7.5	0.99	0.99	0.98	0.98

Table 12 Performance metrics of SMOTEENN for the models run 5 times

The link to the folder uploaded on the google drive is given below:

https://drive.google.com/drive/folders/1LzaRMymVdk-wViJFK00aPHCRtVDGWar1?usp=sharing

6.2. Verification from Literature Review

From the above experimentation it was found that results of this research were in accordance with what was concluded by Luke, Joseph and Balaji (2019, pp. 70) as mentioned below:

- CNN has a significant impact on the accuracy of image resolution.
- Image sizes have a significant effect on the accuracy of any model. Any change in the sizes may degrade or enhance the performance of the model.
- Accuracy increases on increasing the image sizes but decreases on increasing beyond a certain level because of the decrease in the level of receptive field.
- As image size increases, the ability to learn medium and high-level features is affected.
- To achieve higher accuracy, optimal image size lies between default size and 512.
- Resizing images affect low level features due to information loss. Accuracy is affected as inter-class similarities are highly dependent on low-level features (Luke, Joseph and Balaji, 2019, pp.79).

Image resolution has a major influence on the performance of the model. Thambawita et al. (2021) found the performance of CNN best with a large image size of 512x512 as compared to 32x32. (Thambawita, et al., 2021).

According to Horwath et al. (2020, pp. 4), image segmentation is a challenging task for high resolution images and hinders the recognition of interface pixels as these are not considering the background. There is an increase in the complexity of the features.

According to Sabottke and Spieler (2020, pp.2), lesser the number of parameters, the higher is the model performance, because they reduce the chances of overfitting. However, a high level of resolution reduction may result useful information loss. Sabottke and Spieler found that

CNN performed better between resolutions of 256x256 and 448x448 image sizes. They concluded that increasing image resolution has a trade-off with the maximum possible batch size. Large number of parameters affect the performance of the model and that is not mainly because of overfitting but also because of the complexity involved for the optimization (Sabottke and Spieler, 2020, pp. 2)

Rukundo (2021, pp. 17) has written that selection of the best image size for training datasets is a big challenge. He found the performance of size 256x256 superior to 128 × 128. This is because of the low cross-entropy loss in U-net trained on the training set of images of the size 256 × 256 as compared to U-net trained on the training set of images of the size 128 × 128. He stressed the fact that the lower the loss, the more accurate the model.

6.3. Result comparison with models from literature review

The performance of the model developed by Zhao et al. (2019) is given in the table 13.

Dataset	Accuracy	Precision	Recall	F1
	%	%	%	%
500	99.5	100	100	100
1000	97.7	99.26	99.20	99.20
10500	95.7	95.74	95.73	95.73

Table 13 Performance metrics of Zhao et al. (2019) model

The model was based on visual long- short term memory. In comparison to the model developed by Zhao et al. (2019) model, the present model is quite simple where there is no integration of CNN architecture with any other mechanism. Zhao et. al. (2019) experimented on images of varying numbers i.e. 500, 1000 and 10500 and found the performance of dataset with only 500 images best while the performance of other two datasets with larger images of 1000 was low. However, the CNN model developed in this research performed the best with

large dataset of 1,490 images and that too with the image size of 245x345. Hence, this model is most suitable for the deployment in the textile industry.

7. Conclusion

In this research, a novel method is developed for the automatic detection of fabric defects in the textile industry. In the textile industry, yarn from spinning department comes to weaving unit where greige fabric is weaved. The greige fabric is scoured, bleached, dyed and finished. After fabric has been finished, quality of the fabric is checked before the fabric is sent to stitching department for garment making. During quality inspection, fabric is run over the quality tables and quality inspectors check the fabric and mark the fabric if any fabric defect is found.

The aim of the research was to find a deep learning algorithm and that can be deployed in the textile industry to replace the manual tedious job of fabric quality inspection with an automated system. A deep learning algorithm based on convolution neural network was selected based on the literature review. Three image sizes i.e. 200x800, 150x700 and 245x345 were experimented to find the best size with optimum model performance. As dataset was imbalanced, different types of sampling techniques were experimented. It was also found out that CNN model performed best with SMOTEENN sampling technique.

It was found that image size has a great impact on the performance of the model. The developed CNN model showed best performance with SMOTEENN sampling technique and from among the three image sizes 200x800, 150x700 and 245x345, the best performance was with 245x345. This model can be deployed in the industry as the model took only 0.15 sec for validation and 0.06 sec for prediction as compared to the 200x800 that too 10.33 and 0.07 sec for validation and prediction and 150x700 that took 8.29 and 0.12 sec for validation and prediction.

Hence, a method is proposed to automate the whole system of fabric quality inspection. In this method, an infrared camera is used that traverses across the whole width of the fabric as the fabric is run over the table lengthwise. The infrared high-resolution camera takes pictures and the developed machine learning algorithm detects the defects in the fabric. If a defect is found,

the defect is marked by an automatic labelling machine. The whole process is shown in the fig. C below:

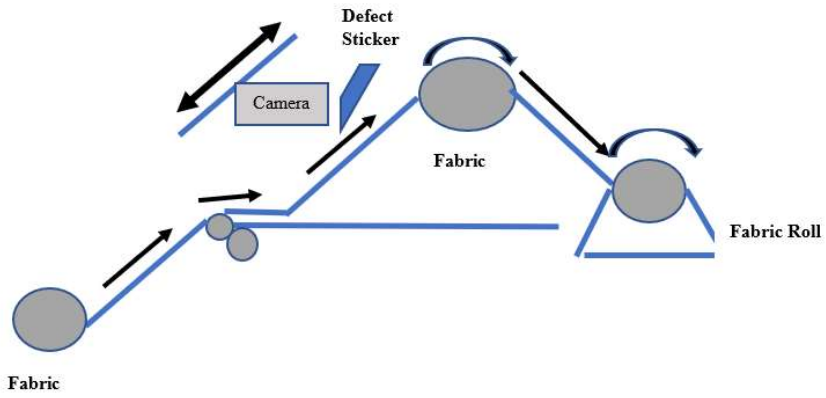


Fig. C Automated Fabric Defect Detection

8. Future Research

More research is required by increasing the number of images and types of defects for multiclassification.

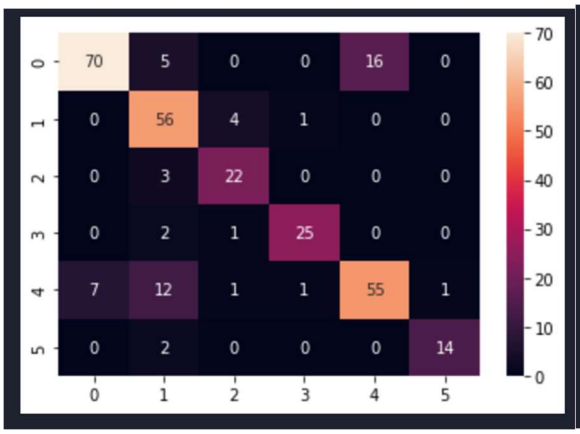
More research is required to deploy this model in the textile industry with the installation of infrared cameras and defect labelling machine to automate the whole system of quality inspection.

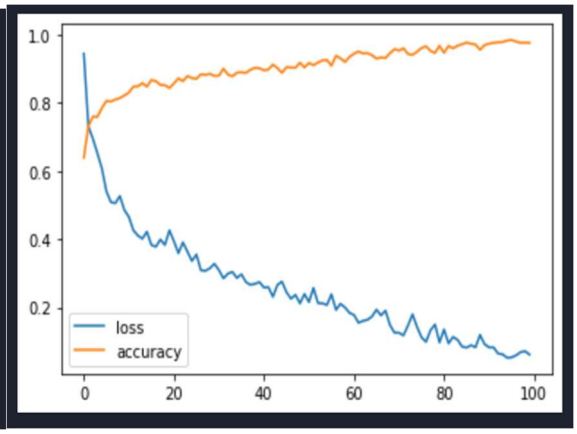
Appendix

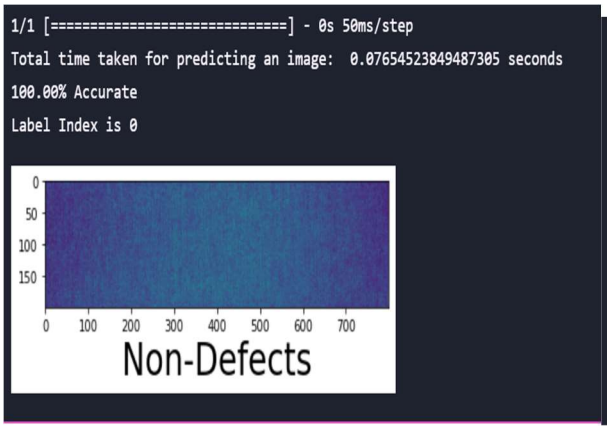
Appendix 1

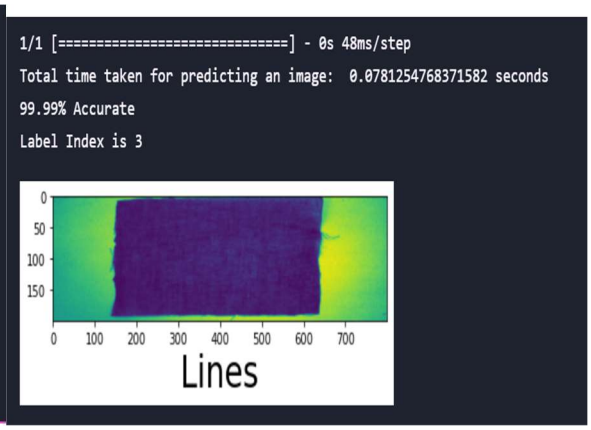
Appendix 1-1 Size: 200 * 800

```
10/10 [==============================] - 7s 714ms/step
              precision    recall  f1-score   support

           0       0.91      0.77      0.83        91
           1       0.70      0.92      0.79        61
           2       0.79      0.88      0.83        25
           3       0.93      0.89      0.91        28
           4       0.77      0.71      0.74        77
           5       0.93      0.88      0.90        16

    accuracy                           0.81       298
   macro avg       0.84      0.84      0.84       298
weighted avg       0.82      0.81      0.81       298
```

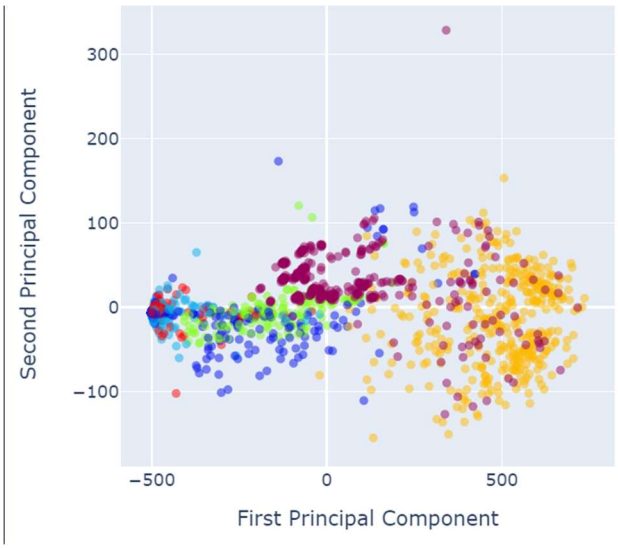




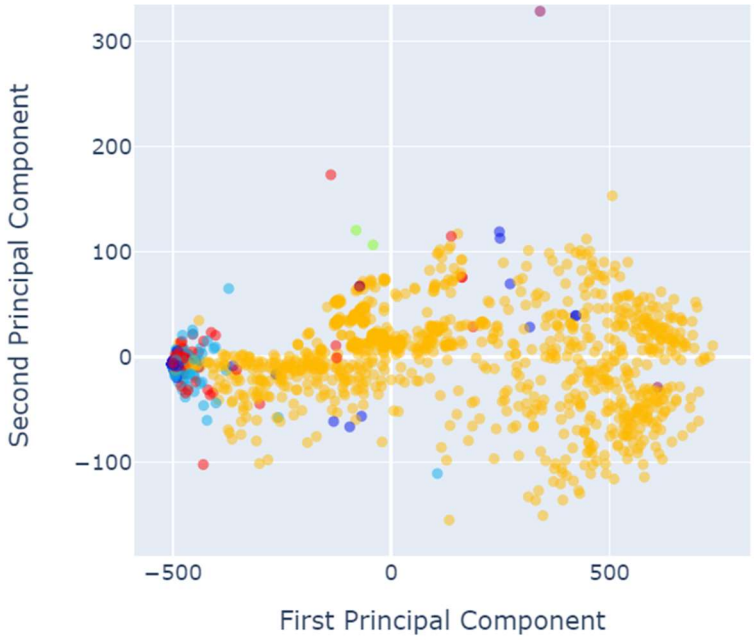




PCA Before CNN

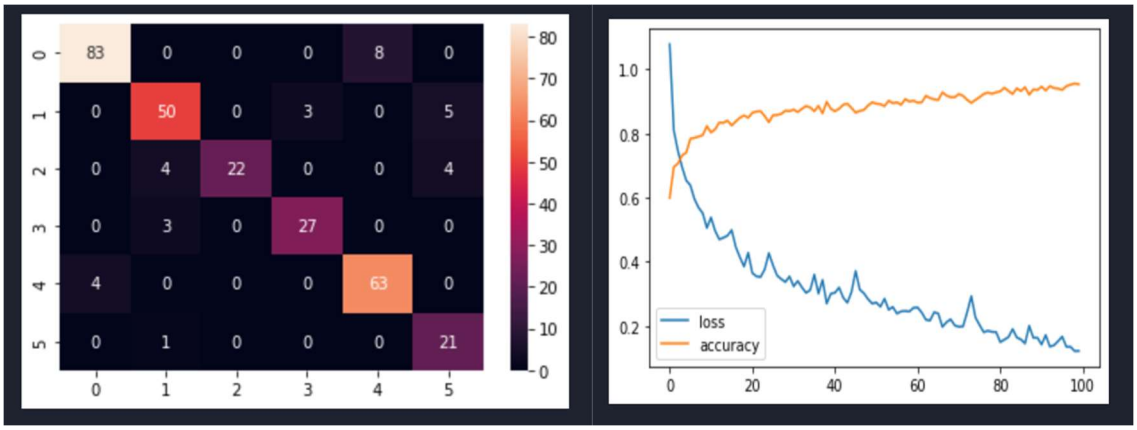


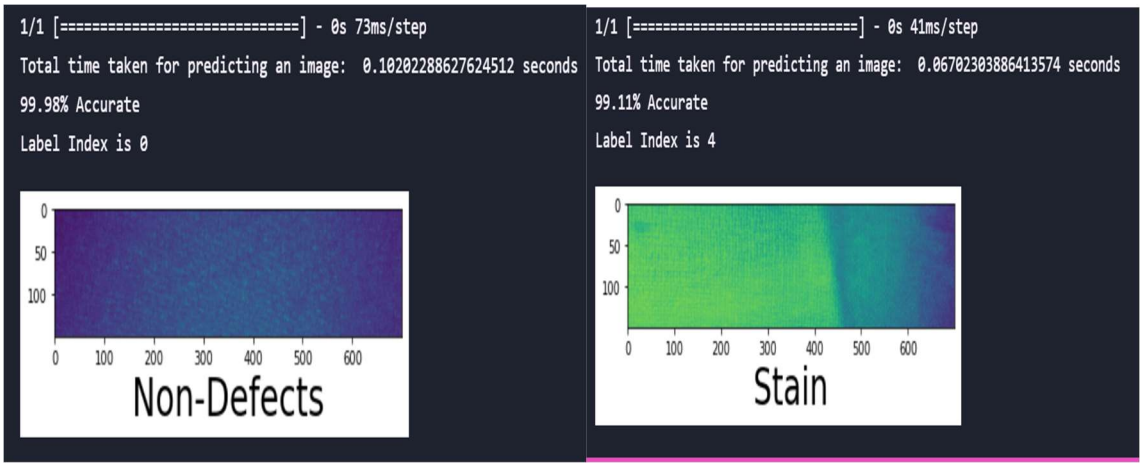
PCA After CNN



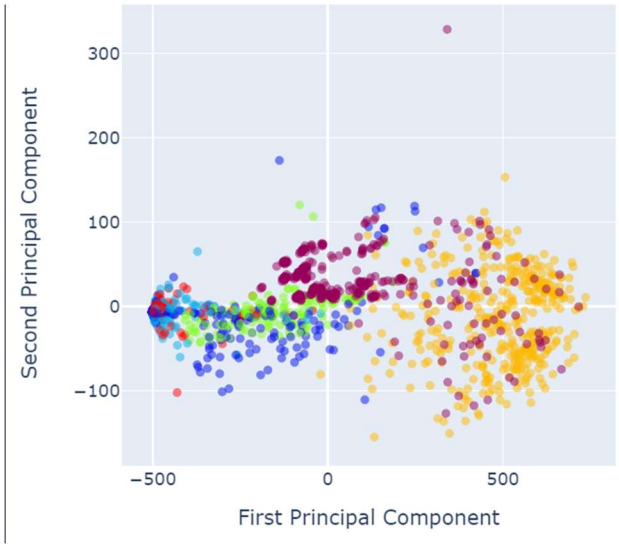
Appendix 1-2 Size: 150*700

```
10/10 [==============================] - 8s 474ms/step
              precision    recall  f1-score   support

           0       0.95      0.91      0.93        91
           1       0.86      0.86      0.86        58
           2       1.00      0.73      0.85        30
           3       0.90      0.90      0.90        30
           4       0.89      0.94      0.91        67
           5       0.70      0.95      0.81        22

    accuracy                           0.89       298
   macro avg       0.88      0.88      0.88       298
weighted avg       0.90      0.89      0.89       298
```

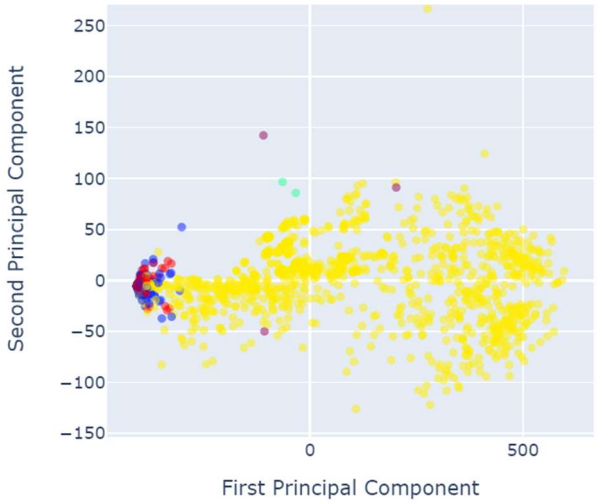




PCA Before CNN

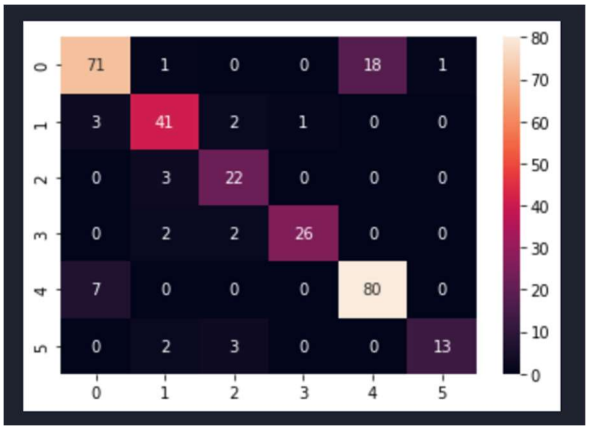


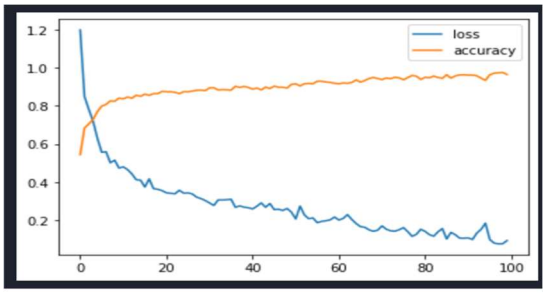
PCA After CNN

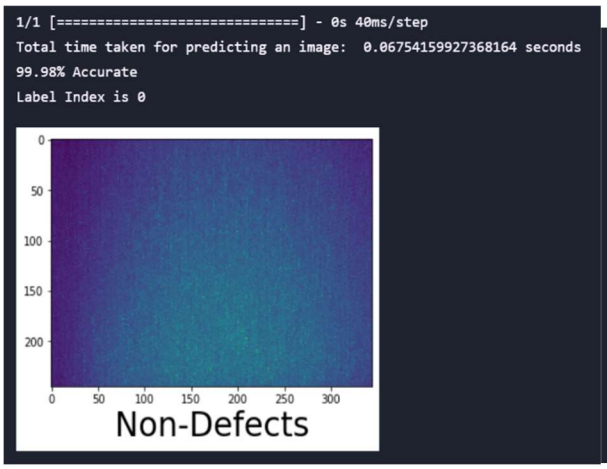


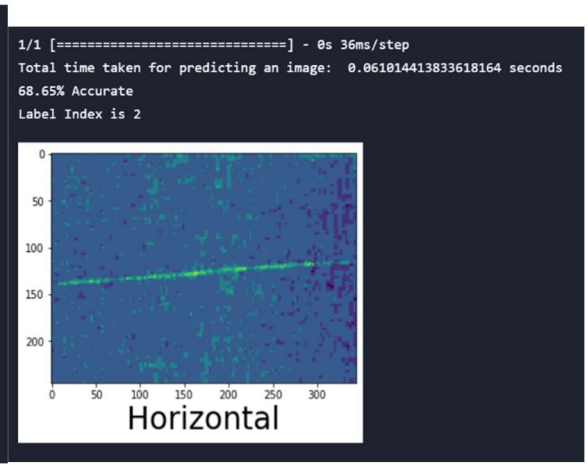
Appendix 1-3 Size 245x345

```
10/10 [==============================] - 4s 392ms/step
              precision    recall  f1-score   support

           0       0.88      0.78      0.83        91
           1       0.84      0.87      0.85        47
           2       0.76      0.88      0.81        25
           3       0.96      0.87      0.91        30
           4       0.82      0.92      0.86        87
           5       0.93      0.72      0.81        18

    accuracy                           0.85       298
   macro avg       0.86      0.84      0.85       298
weighted avg       0.85      0.85      0.85       298
```

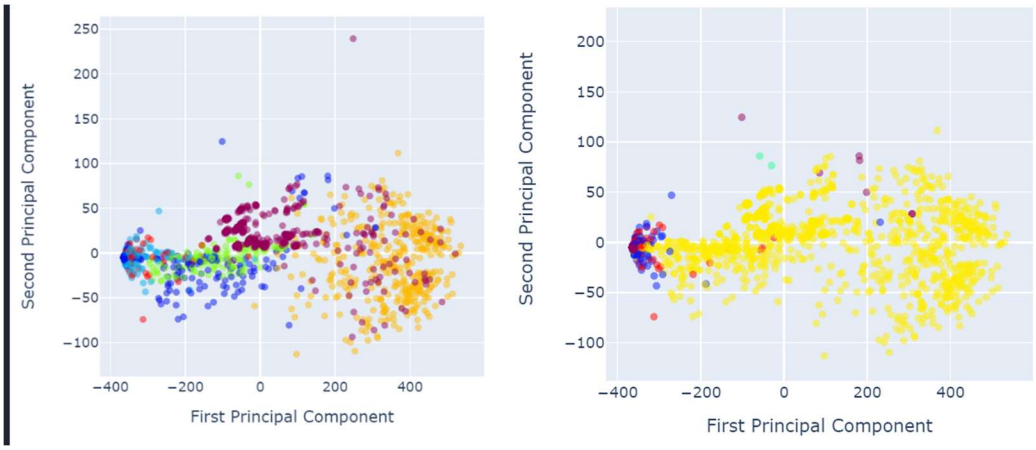








PCA Before and After CNN



Appendix 2 Undersampling

Appendix 2-1.1: Cluster Centroids Size 200X800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

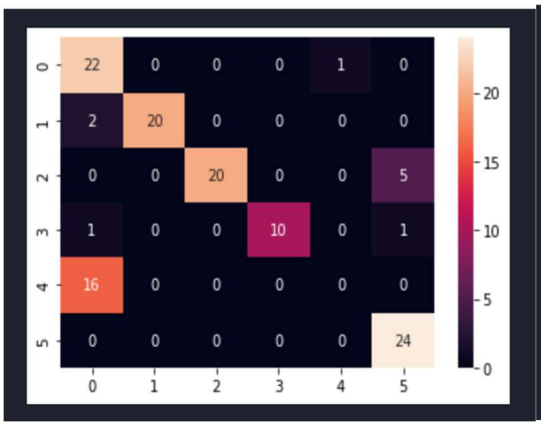
```
cc = ClusterCentroids(random_state=42)
X_cus, y_cus = cc.fit_resample(X_res, y_res)
```

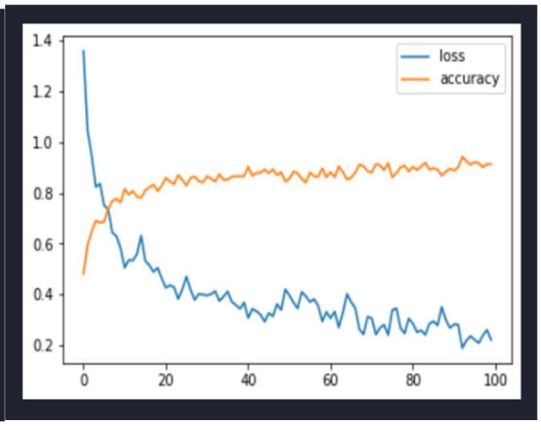
```
print(f'Resampled dataset shape {Counter(y_cus)}')
```

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

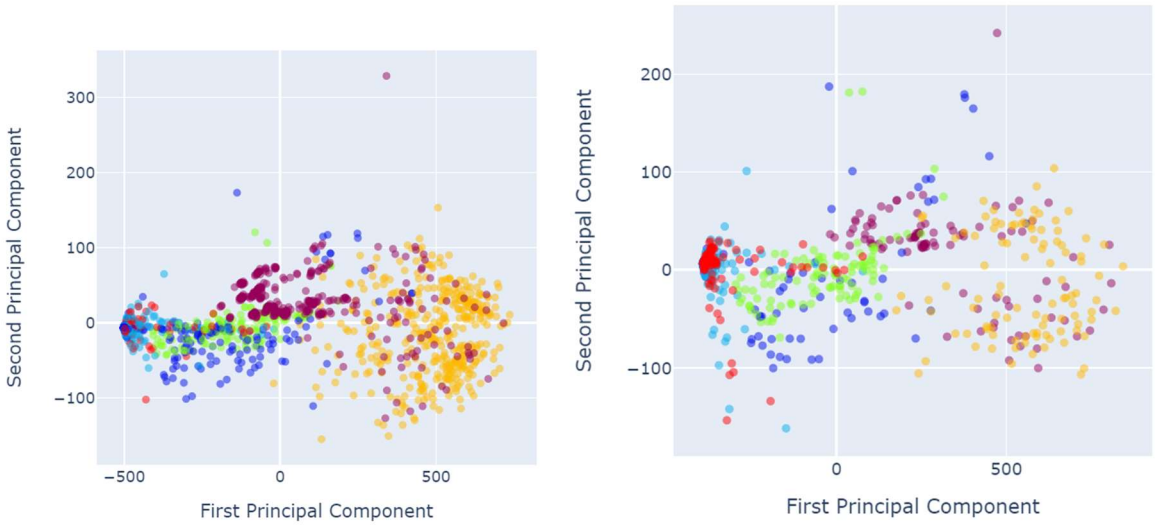
```
4/4 [==============================] - 3s 757ms/step
```

	precision	recall	f1-score	support
0	0.54	0.96	0.69	23
1	1.00	0.91	0.95	22
2	1.00	0.80	0.89	25
3	1.00	0.83	0.91	12
4	0.00	0.00	0.00	16
5	0.80	1.00	0.89	24
accuracy			0.79	122
macro avg	0.72	0.75	0.72	122
weighted avg	0.74	0.79	0.75	122

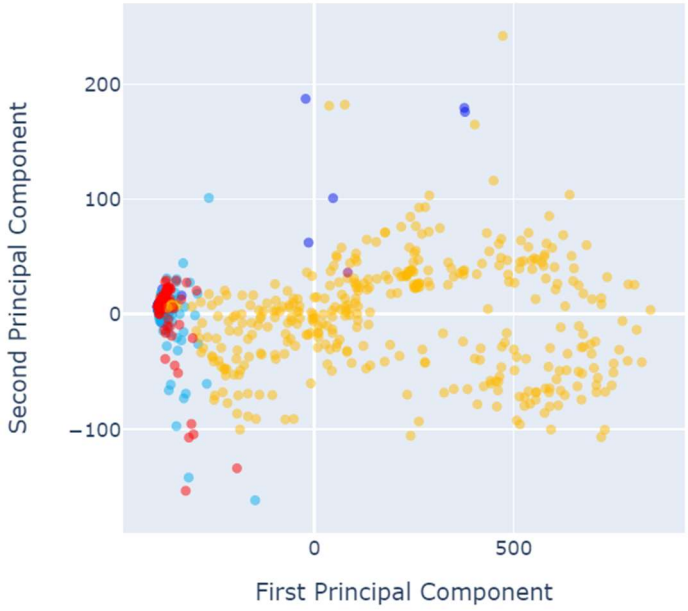




PCA Before and After Sampling



PCA after CNN



Appendix 2-1.2: Cluster Centroids 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

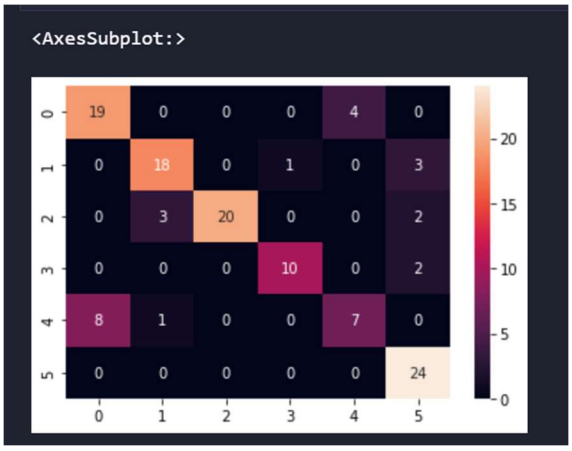
```
cc = ClusterCentroids(random_state=42)
X_cus, y_cus = cc.fit_resample(X_res, y_res)
```

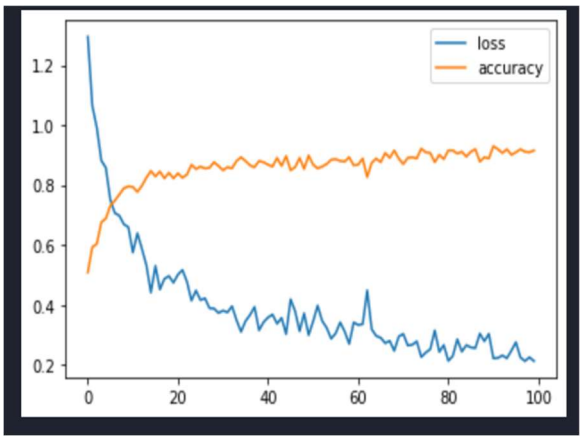
```
print(f'Resampled dataset shape {Counter(y_cus)}')
```

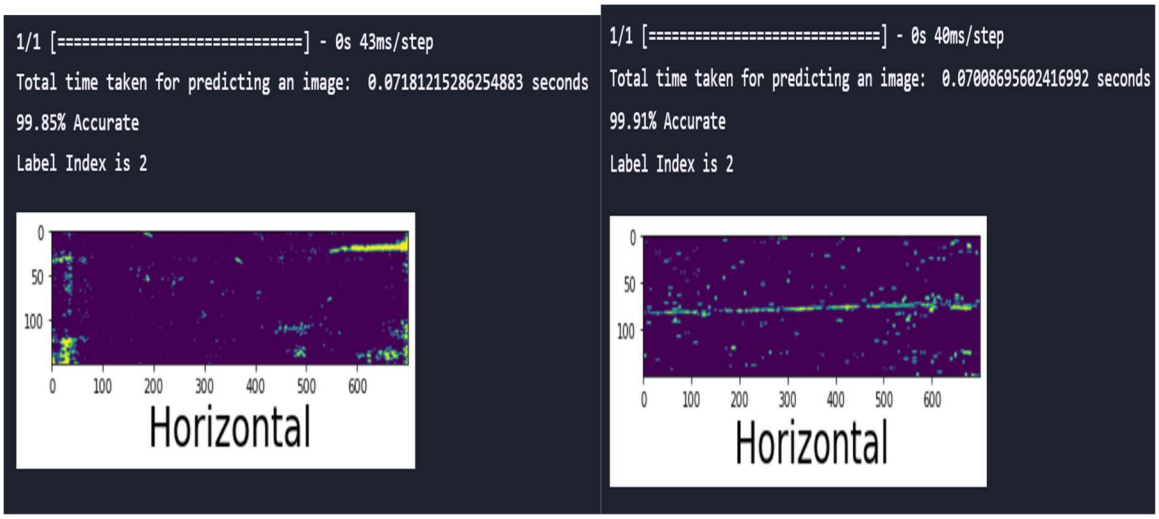
```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

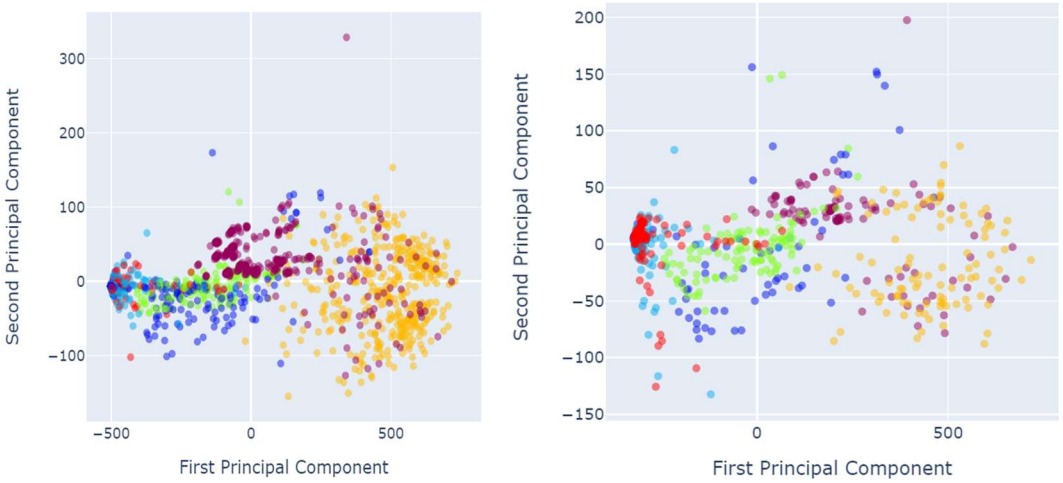
```
4/4 [==============================] - 2s 482ms/step
```

	precision	recall	f1-score	support
0	0.70	0.83	0.76	23
1	0.82	0.82	0.82	22
2	1.00	0.80	0.89	25
3	0.91	0.83	0.87	12
4	0.64	0.44	0.52	16
5	0.77	1.00	0.87	24
accuracy			0.80	122
macro avg	0.81	0.79	0.79	122
weighted avg	0.81	0.80	0.80	122

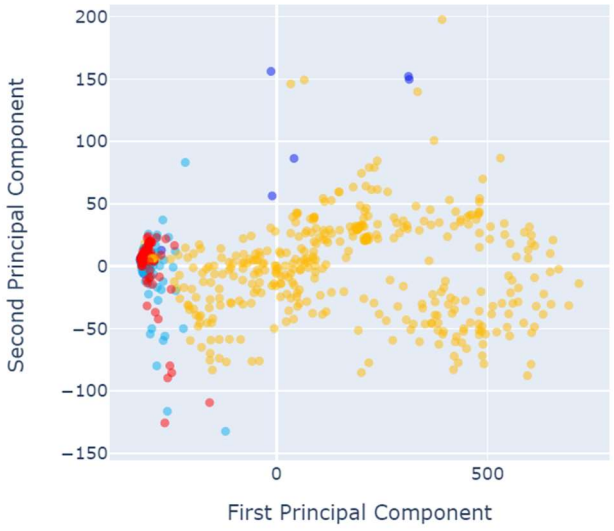








PCA After CNN



Appendix 2-1.3: Cluster Centroids 245x345

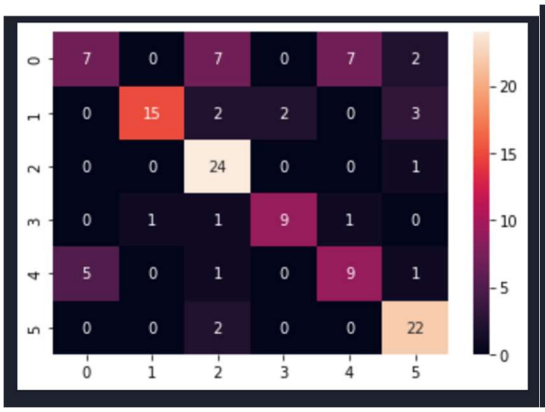
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

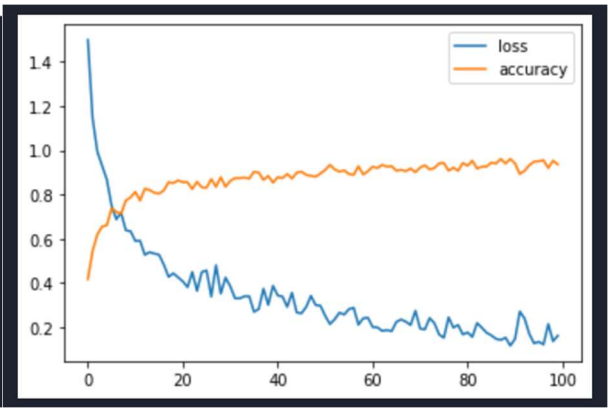
```
cc = ClusterCentroids(random_state=42)
X_cus, y_cus = cc.fit_resample(X_res, y_res)
```

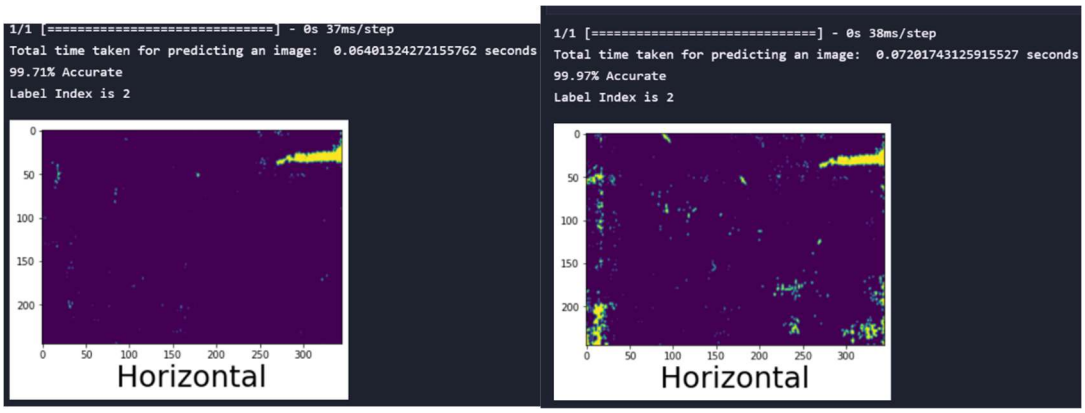
```
print(f'Resampled dataset shape {Counter(y_cus)}')
```

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

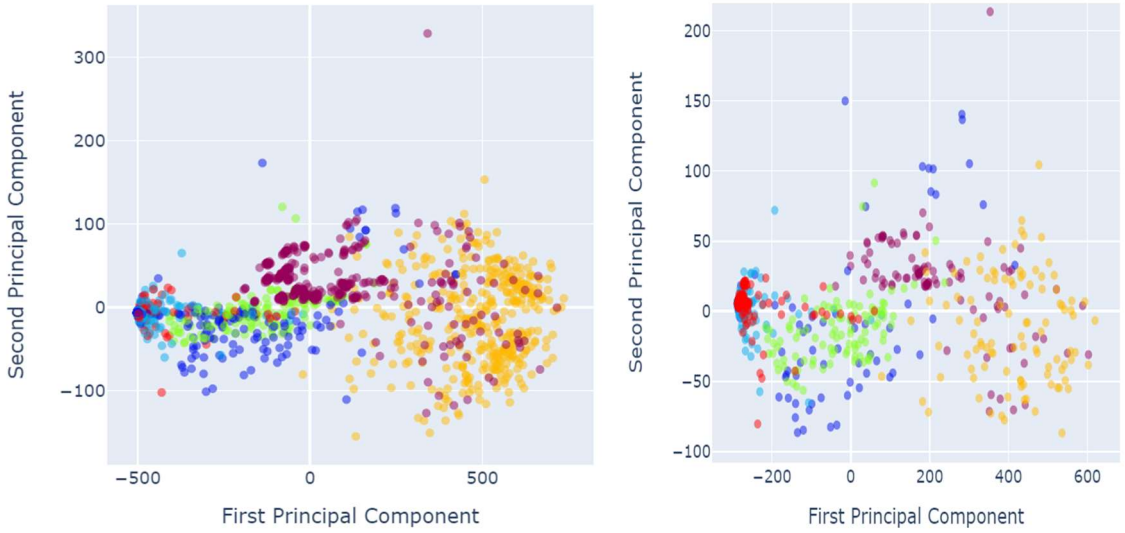
```
4/4 [==============================] - 2s 392ms/step
              precision    recall  f1-score   support

           0       0.58      0.30      0.40        23
           1       0.94      0.68      0.79        22
           2       0.65      0.96      0.77        25
           3       0.82      0.75      0.78        12
           4       0.53      0.56      0.55        16
           5       0.76      0.92      0.83        24

    accuracy                           0.70       122
   macro avg       0.71      0.70      0.69       122
weighted avg       0.71      0.70      0.69       122
```



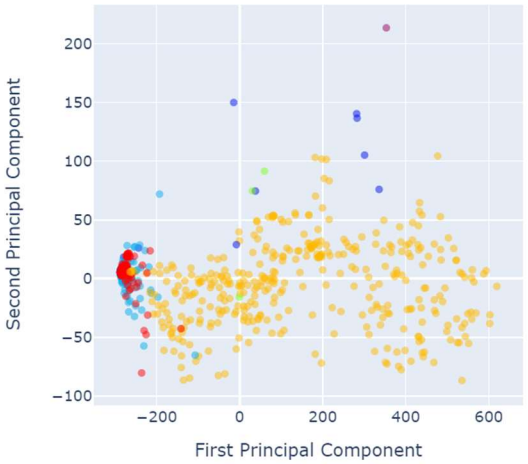




PCA Before and After Sampling



PCA After CNN



Appendix 2-2.1: Condensed Nearest Neighbor 200x800

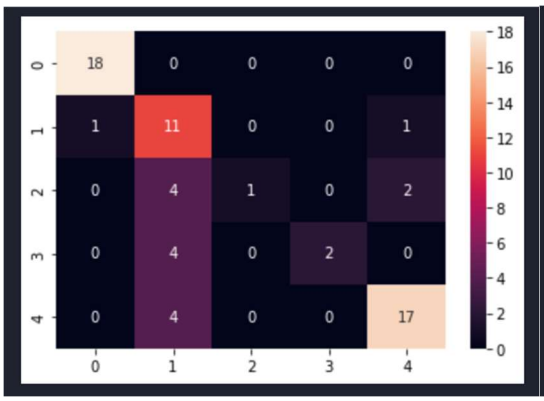
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

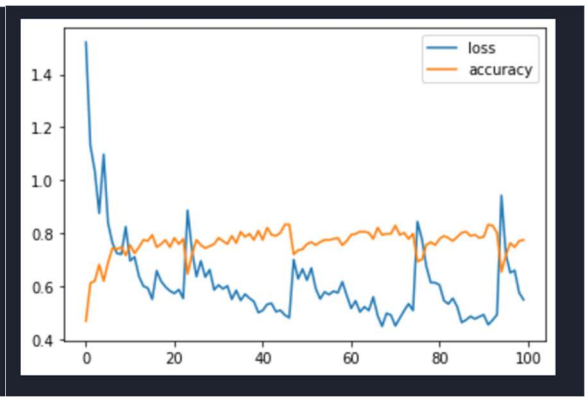
```
cn = CondensedNearestNeighbour(random_state=42)
X_cus, y_cus = cn.fit_resample(X_res, y_res)
```

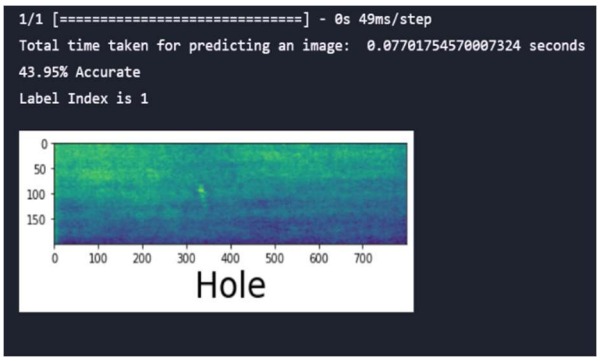
```
print(f'Resampled dataset shape {Counter(y_cus)}')
```

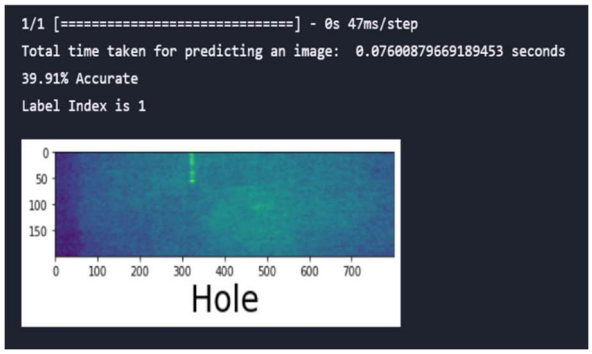
```
Resampled dataset shape Counter({5: 101, 0: 81, 1: 61, 2: 52, 3: 24, 4: 4})
```

```
3/3 [==============================] - 2s 416ms/step
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        18
           1       0.48      0.85      0.61        13
           2       1.00      0.14      0.25         7
           3       1.00      0.33      0.50         6
           5       0.85      0.81      0.83        21

    accuracy                           0.75        65
   macro avg       0.86      0.63      0.63        65
weighted avg       0.83      0.75      0.73        65
```

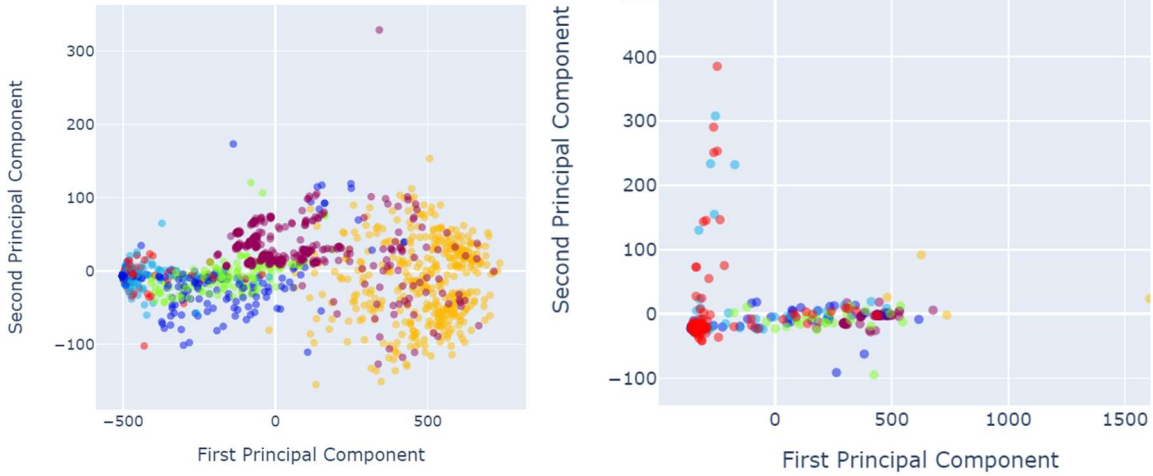




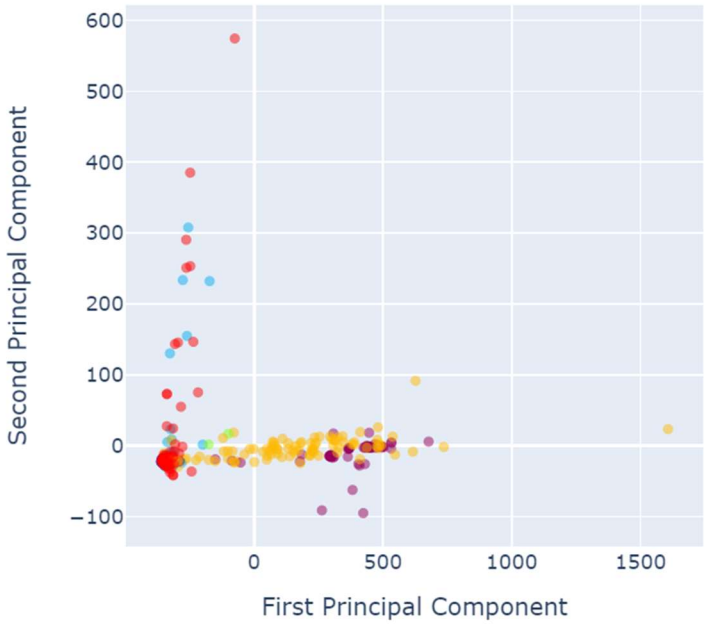




PCA Before and After Sampling



PCA After CNN



Appendix 2-2.2: Condensed Nearest Neighbor 150x700

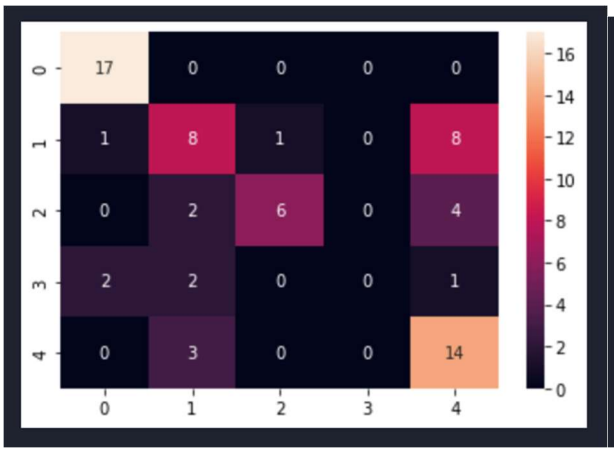
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

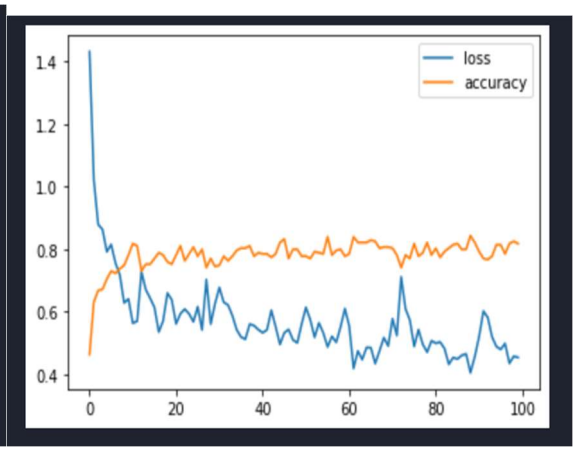
```
    cn = CondensedNearestNeighbour(random_state=42)
    X_cus, y_cus = cn.fit_resample(X_res, y_res)
```

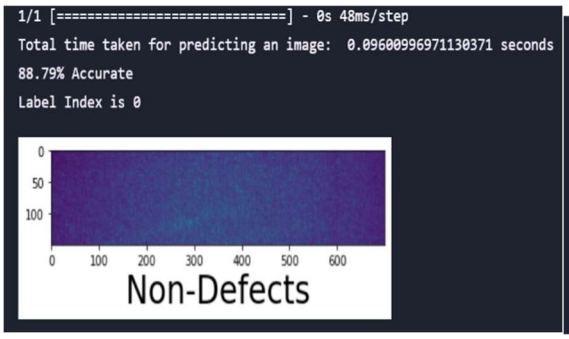
```
    print(f'Resampled dataset shape {Counter(y_cus)}')
```

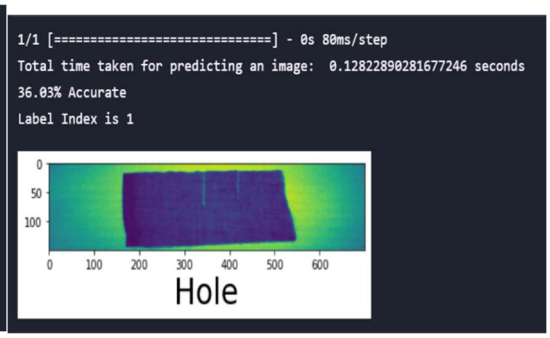
```
Resampled dataset shape Counter({5: 101, 0: 76, 1: 76, 2: 58, 3: 28, 4: 4})
```

```
3/3 [==============================] - 1s 308ms/step
              precision    recall  f1-score   support

           0       0.85      1.00      0.92        17
           1       0.53      0.44      0.48        18
           2       0.86      0.50      0.63        12
           3       0.00      0.00      0.00         5
           5       0.52      0.82      0.64        17

    accuracy                           0.65        69
   macro avg       0.55      0.55      0.53        69
weighted avg       0.63      0.65      0.62        69
```

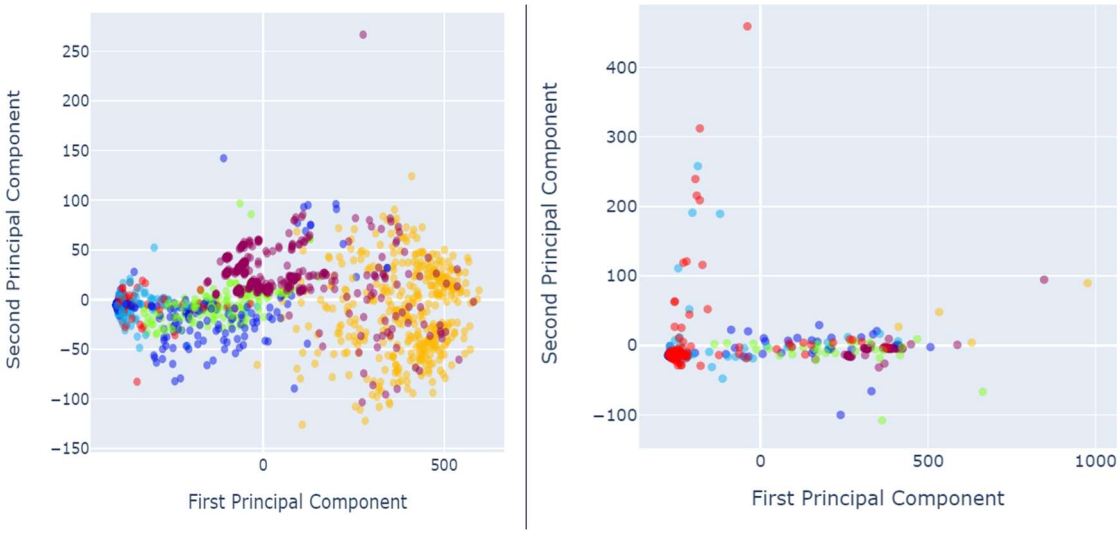




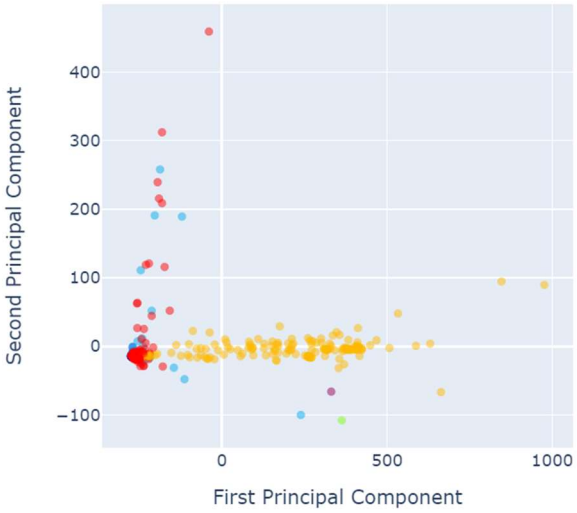




PCA Before and After Sampling



PCA After CNN



Appendix 2-2.3: Condensed Nearest Neighbour Size 245x345

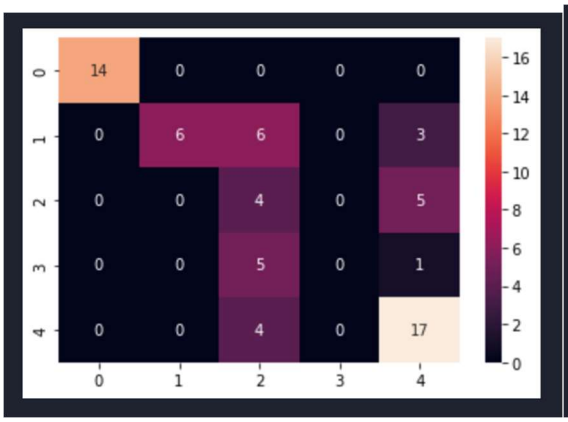
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

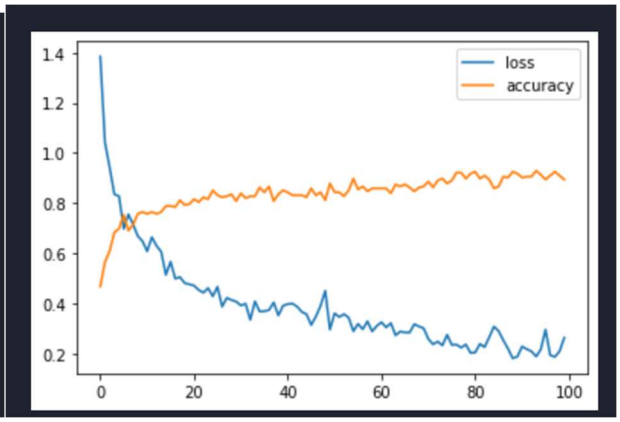
```
cn = CondensedNearestNeighbour(random_state=42)
X_cus, y_cus = cn.fit_resample(X_res, y_res)
```

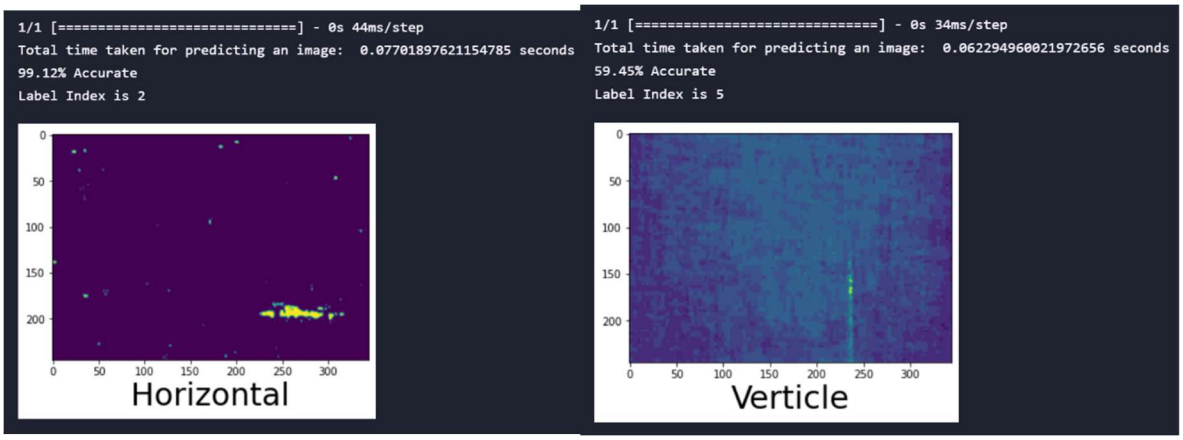
```
print(f'Resampled dataset shape {Counter(y_cus)}')
```

```
Resampled dataset shape Counter({5: 101, 0: 68, 2: 62, 1: 57, 3: 28, 4: 5})
```

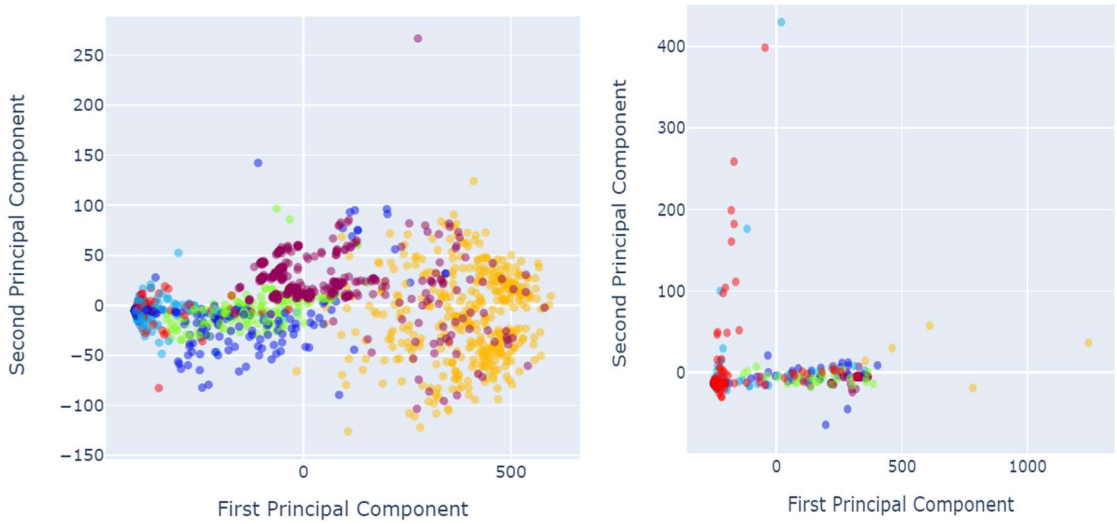
```
3/3 [==============================] - 1s 248ms/step
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        14
           1       1.00      0.40      0.57        15
           2       0.21      0.44      0.29         9
           3       0.00      0.00      0.00         6
           5       0.65      0.81      0.72        21

    accuracy                           0.63        65
   macro avg       0.57      0.53      0.52        65
weighted avg       0.69      0.63      0.62        65
```



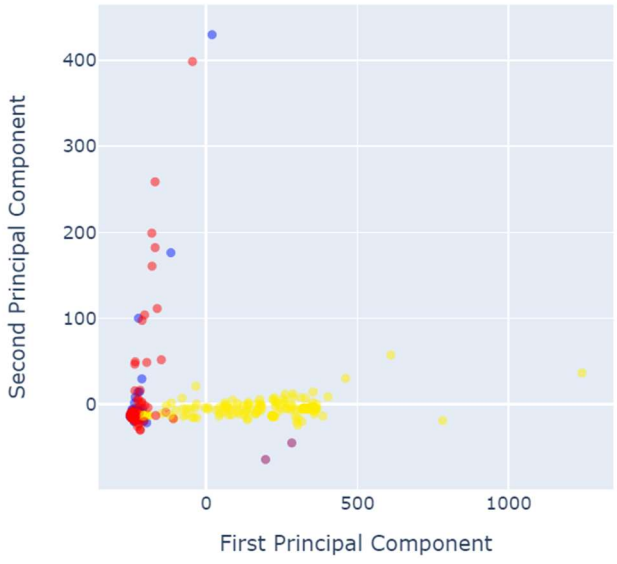




PCA Before and After Sampling



PCA After CNN



Appendix 2-3.1: Edited Nearest Neighbor 200x800

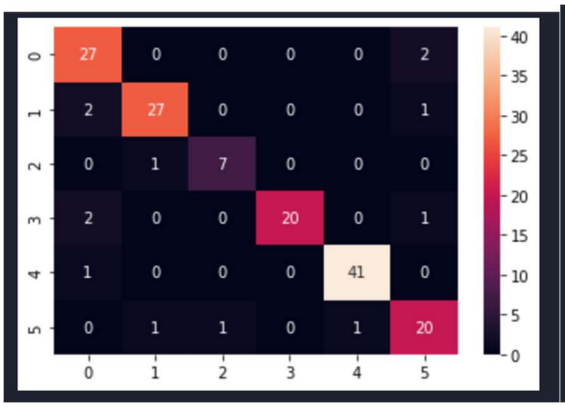
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

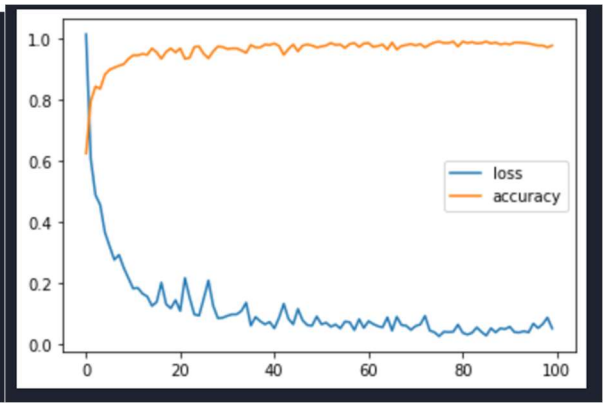
```
ust = EditedNearestNeighbours()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

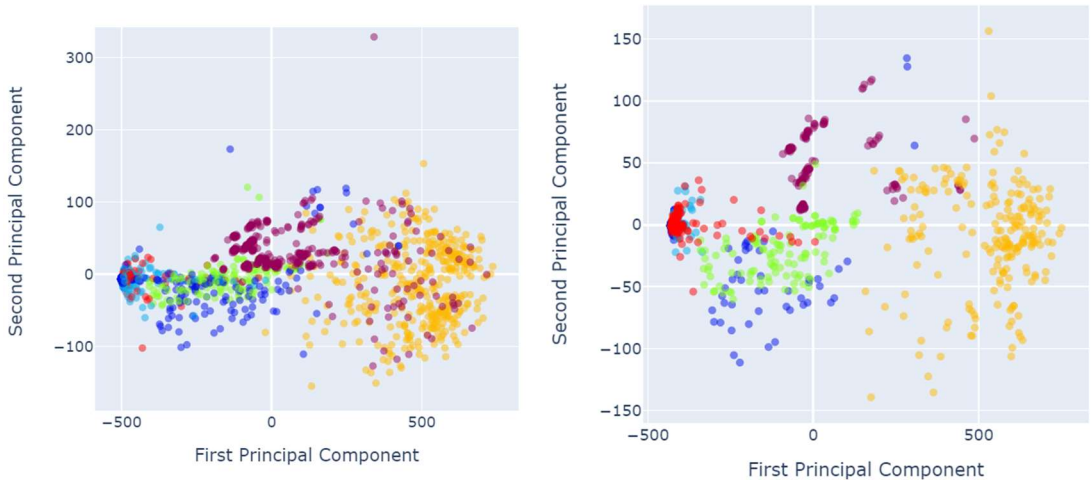
```
Resampled dataset shape Counter({4: 216, 1: 163, 0: 137, 3: 118, 5: 101, 2: 39})
```

```
5/5 [==============================] - 4s 775ms/step
              precision    recall  f1-score   support

           0       0.84      0.93      0.89        29
           1       0.93      0.90      0.92        30
           2       0.88      0.88      0.88         8
           3       1.00      0.87      0.93        23
           4       0.98      0.98      0.98        42
           5       0.83      0.87      0.85        23

    accuracy                           0.92       155
   macro avg       0.91      0.90      0.91       155
weighted avg       0.92      0.92      0.92       155
```

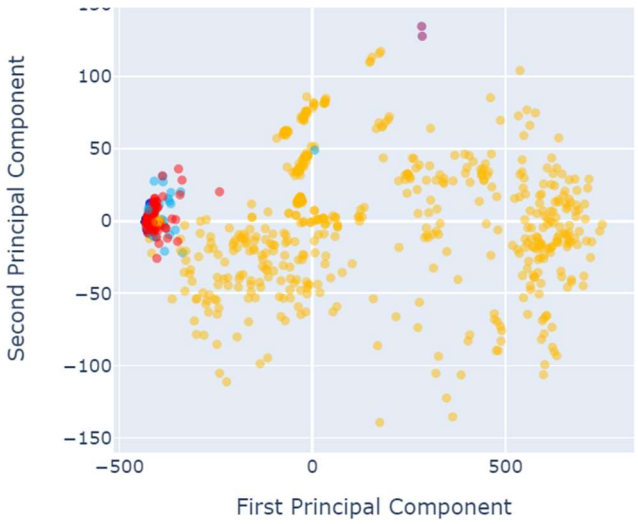




PCA Before and After Sampling



PCA After CNN



Appendix 2-3.2: Edited Nearest Neighbor 150x700

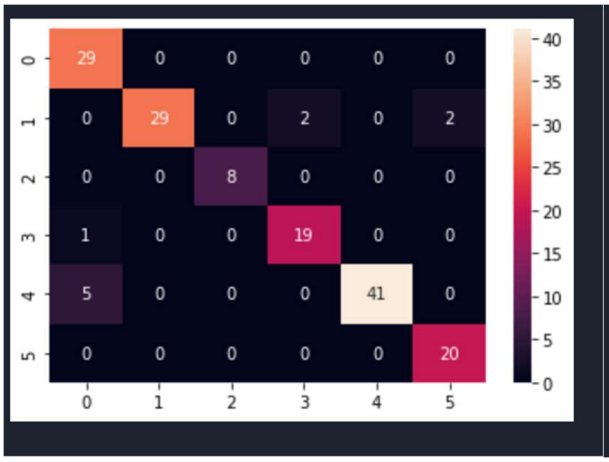
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

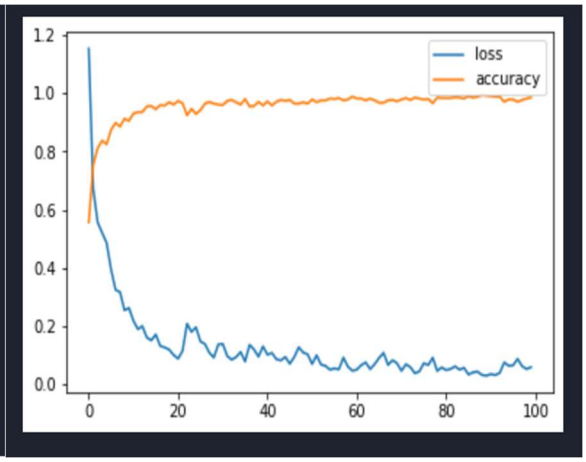
```
ust = EditedNearestNeighbours()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

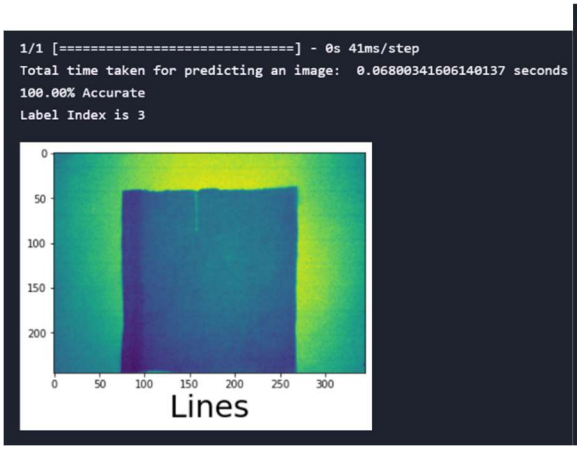
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

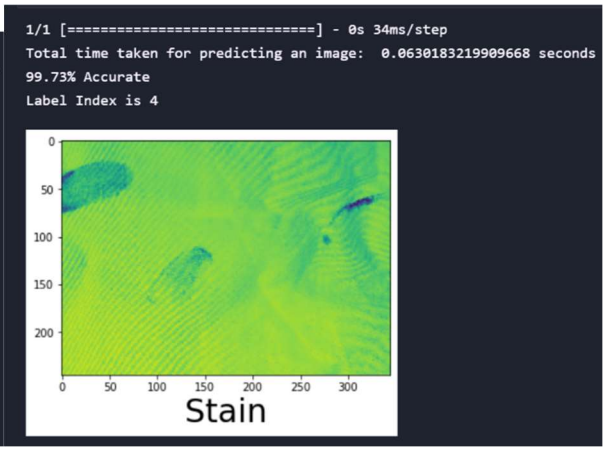
```
Resampled dataset shape Counter({4: 212, 1: 164, 0: 146, 3: 118, 5: 101, 2: 38})
```

```
5/5 [==============================] - 2s 412ms/step
              precision    recall  f1-score   support

           0       0.83      1.00      0.91        29
           1       1.00      0.88      0.94        33
           2       1.00      1.00      1.00         8
           3       0.90      0.95      0.93        20
           4       1.00      0.89      0.94        46
           5       0.91      1.00      0.95        20

    accuracy                           0.94       156
   macro avg       0.94      0.95      0.94       156
weighted avg       0.94      0.94      0.94       156
```

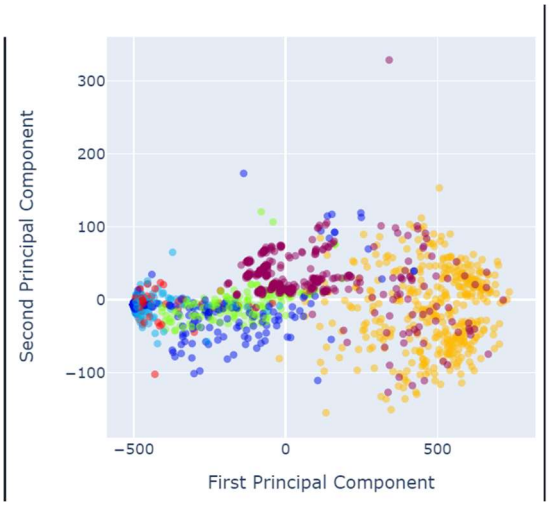


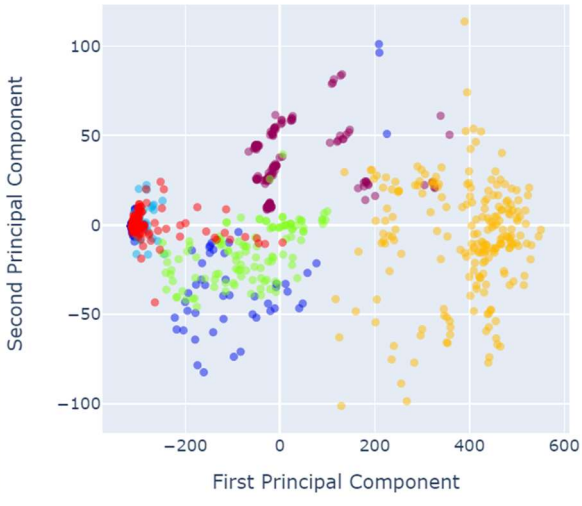




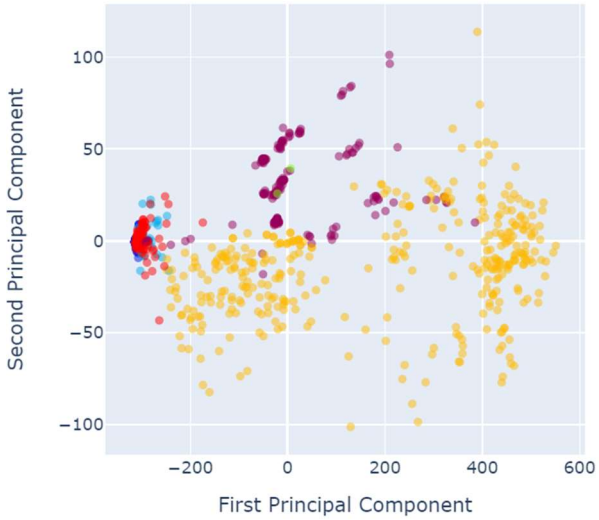


PCA Before and After Sampling





PCA After CNN



Appendix 2-3.3: Edited Nearest Neighbor Size 245x345

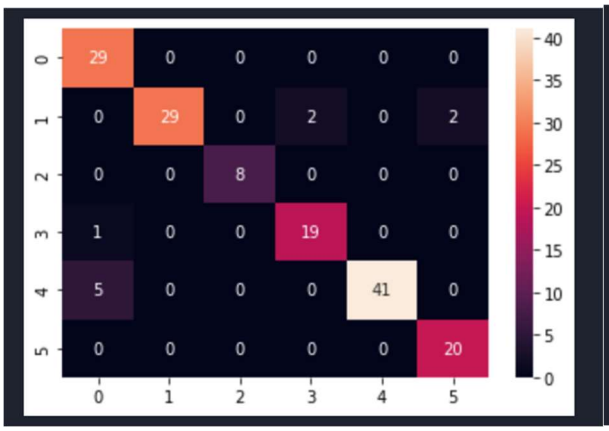
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

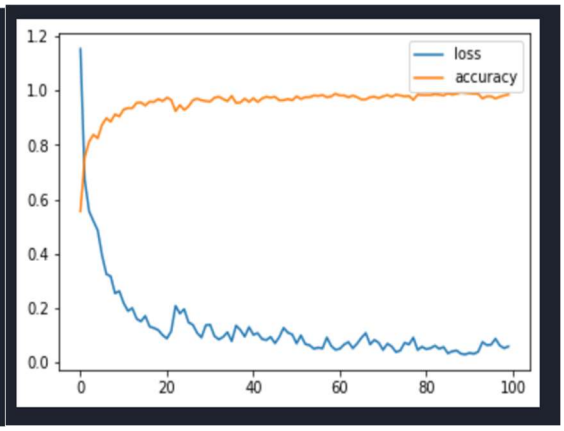
```
    ust = EditedNearestNeighbours()
    X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

```
    print(f'Resampled dataset shape {Counter(y_ust)}')
```

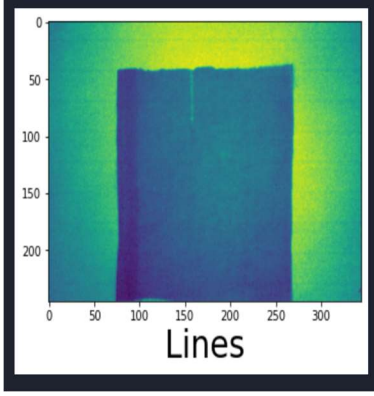
```
Resampled dataset shape Counter({4: 212, 1: 164, 0: 146, 3: 118, 5: 101, 2: 38})
```

```
5/5 [==============================] - 2s 412ms/step
              precision    recall  f1-score   support

           0       0.83      1.00      0.91        29
           1       1.00      0.88      0.94        33
           2       1.00      1.00      1.00         8
           3       0.90      0.95      0.93        20
           4       1.00      0.89      0.94        46
           5       0.91      1.00      0.95        20

    accuracy                           0.94       156
   macro avg       0.94      0.95      0.94       156
weighted avg       0.94      0.94      0.94       156
```

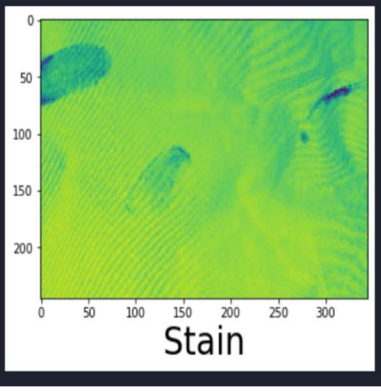




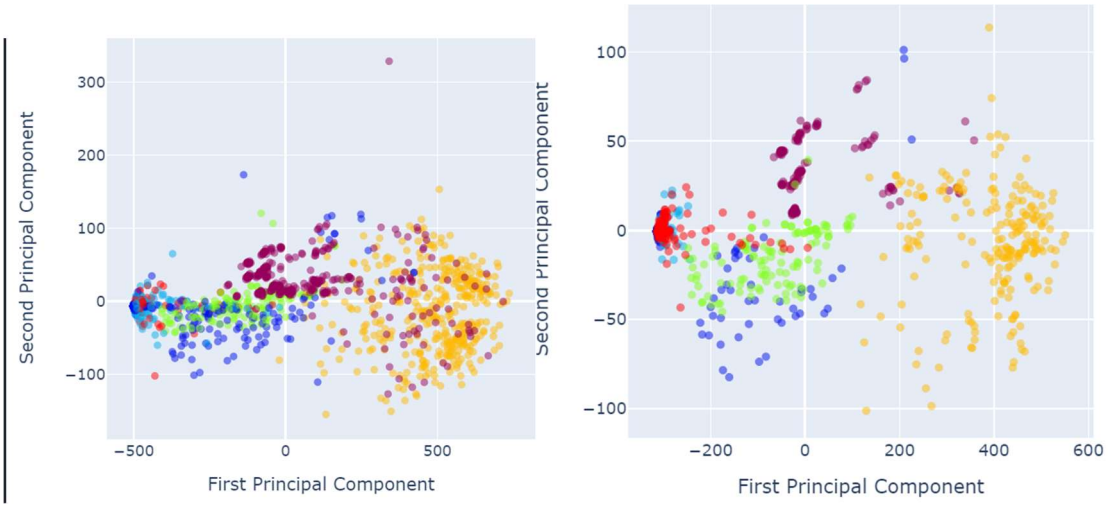

```
1/1 [==============================] - 0s 41ms/step
Total time taken for predicting an image:  0.06800341606140137 seconds
100.00% Accurate
Label Index is 3
```



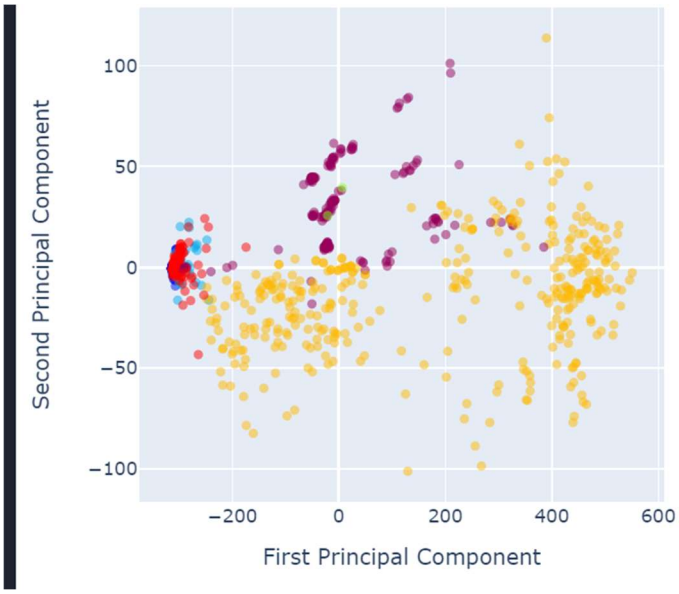

```
1/1 [==============================] - 0s 34ms/step
Total time taken for predicting an image:  0.0630183219909668 seconds
99.73% Accurate
Label Index is 4
```



PCA Before and After Sampling



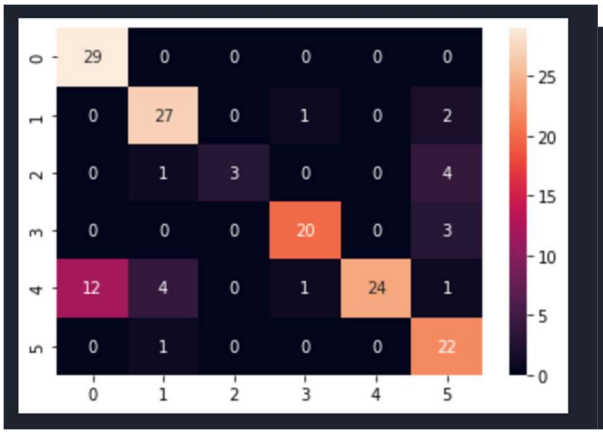
PCA After CNN

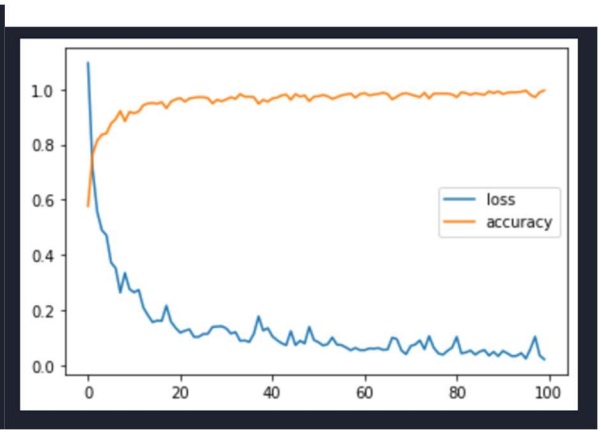


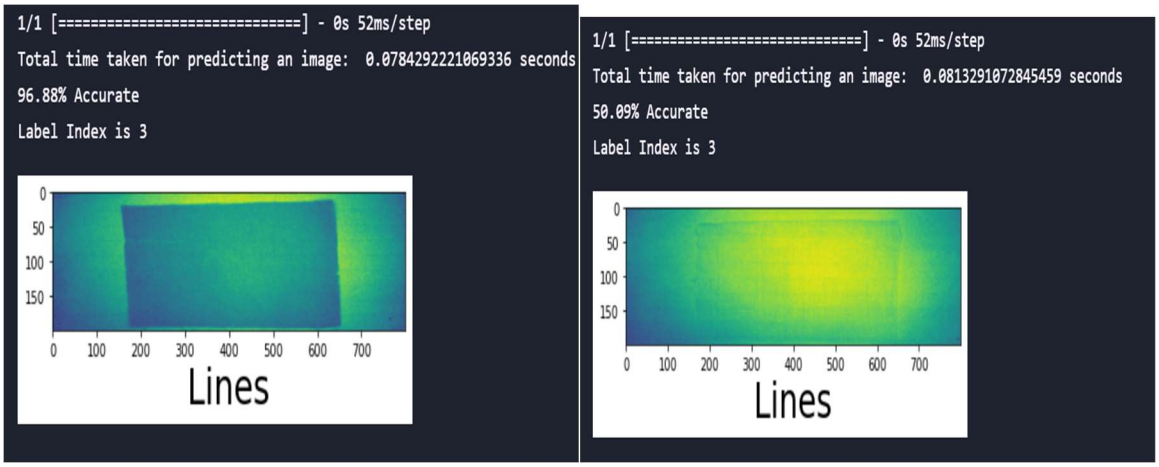
Appendix 2-4.1: Repeated Edited Nearest Neighbors Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})

    ust = RepeatedEditedNearestNeighbours()
    X_ust, y_ust = ust.fit_resample(X_res, y_res)

    print(f'Resampled dataset shape {Counter(y_ust)}')

Resampled dataset shape Counter({4: 216, 1: 163, 0: 137, 3: 118, 5: 101, 2: 39})
```

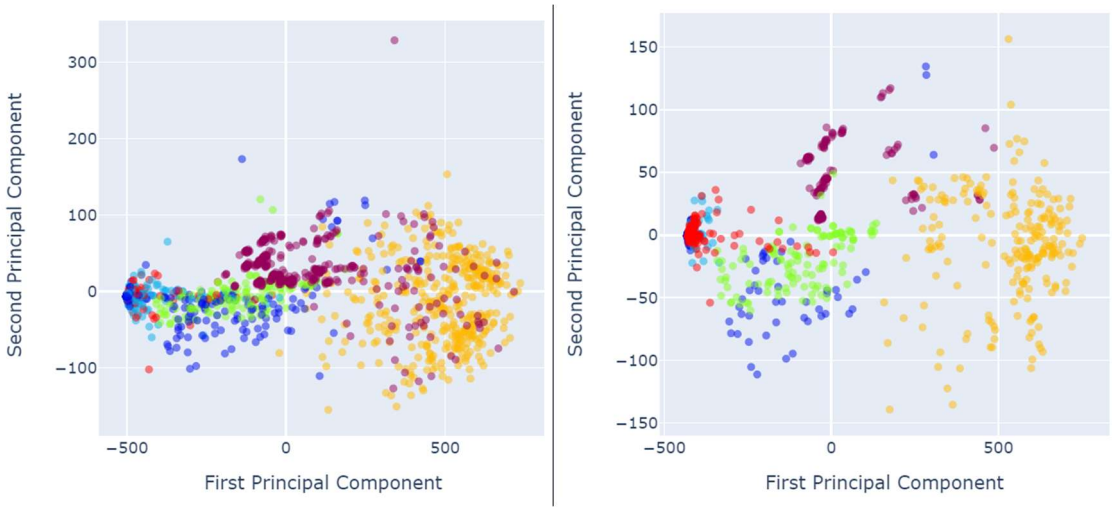
```
5/5 [==============================] - 4s 772ms/step
              precision    recall  f1-score   support

           0       0.71      1.00      0.83        29
           1       0.82      0.90      0.86        30
           2       1.00      0.38      0.55         8
           3       0.91      0.87      0.89        23
           4       1.00      0.57      0.73        42
           5       0.69      0.96      0.80        23

    accuracy                           0.81       155
   macro avg       0.85      0.78      0.77       155
weighted avg       0.85      0.81      0.80       155
```



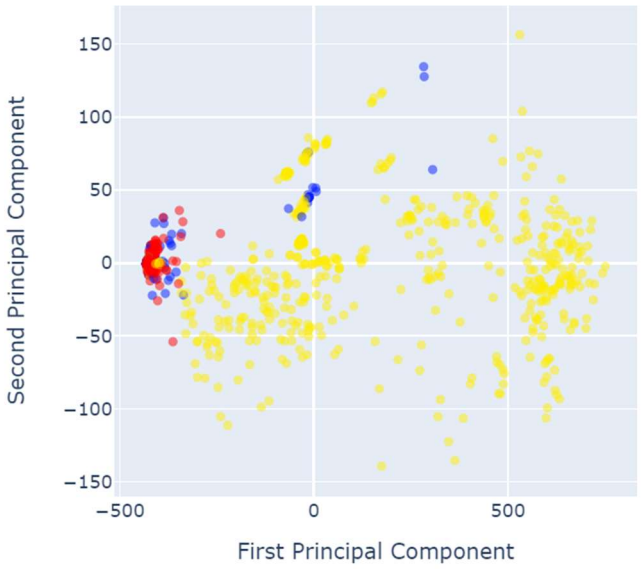




PCA Before and After Sampling



PCA After CNN



Appendix 2-4.2: Repeated Edited Nearest Neighbour Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = RepeatedEditedNearestNeighbours()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

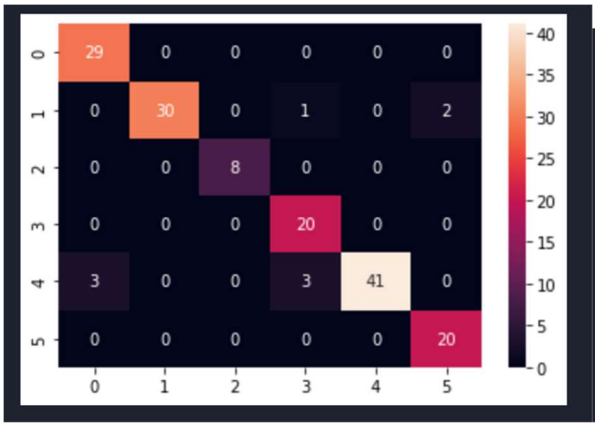
✓ 7.3s

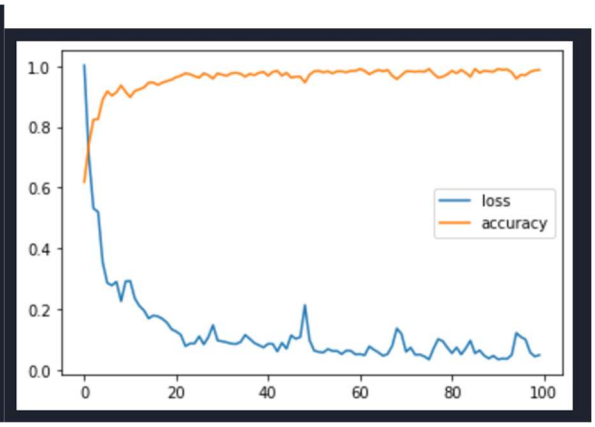
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

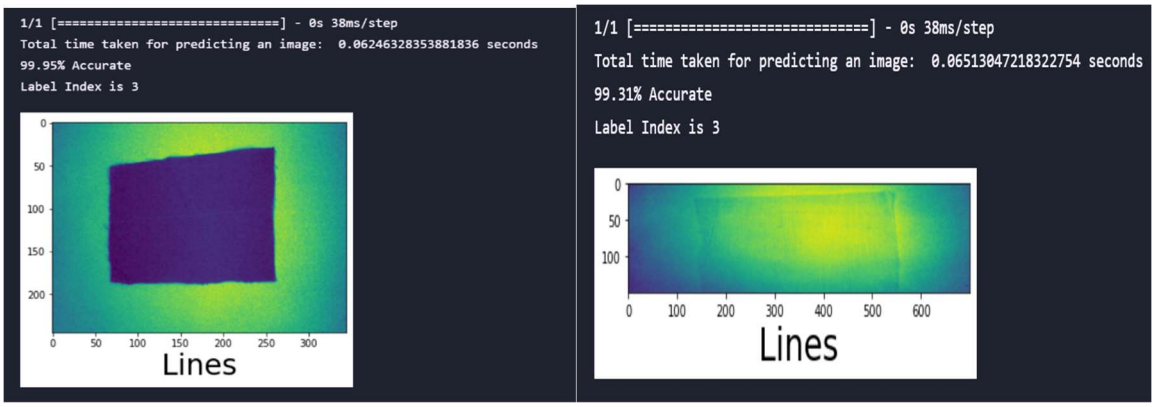
✓ 0.4s

```
Resampled dataset shape Counter({4: 212, 1: 164, 0: 146, 3: 118, 5: 101, 2: 38})
```

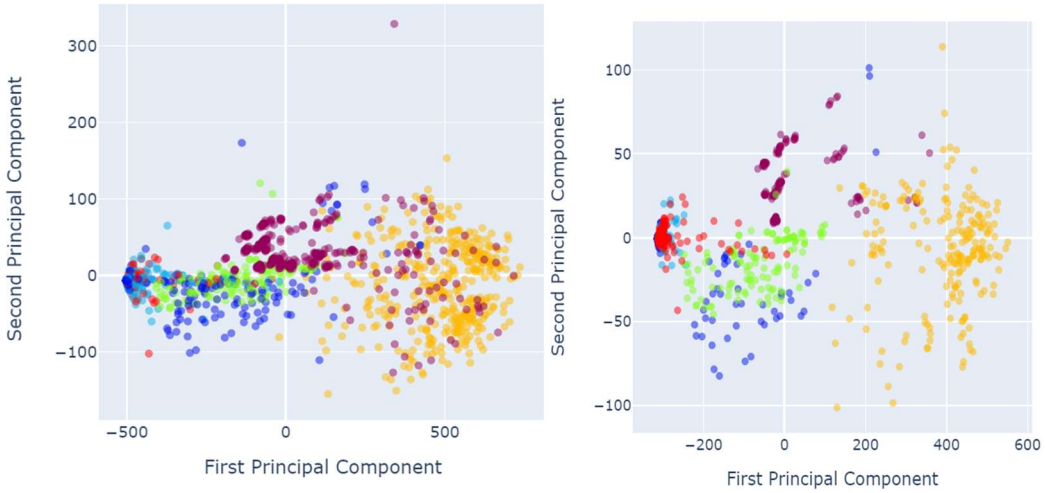
```
5/5 [==============================] - 3s 527ms/step
              precision    recall  f1-score   support

           0       0.91      1.00      0.95        29
           1       1.00      0.91      0.95        33
           2       1.00      1.00      1.00         8
           3       0.83      1.00      0.91        20
           4       1.00      0.87      0.93        47
           5       0.91      1.00      0.95        20

    accuracy                           0.94       157
   macro avg       0.94      0.96      0.95       157
weighted avg       0.95      0.94      0.94       157
```



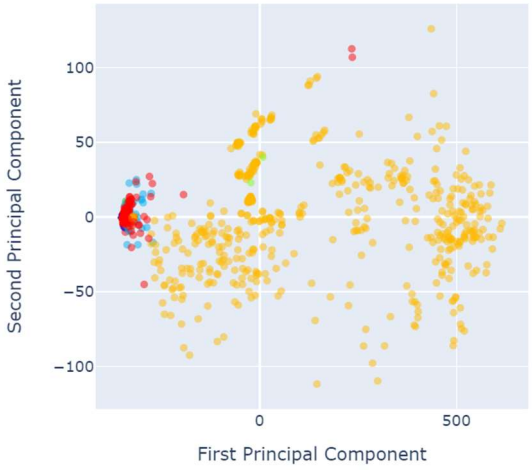




PCA Before and After Sampling



PCA After CNN



Appendix 2-4.3: Repeated Edited Nearest Neighbour Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = RepeatedEditedNearestNeighbours()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

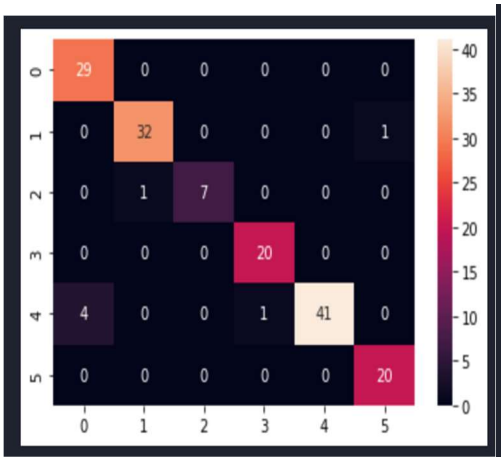
✓ 7.3s

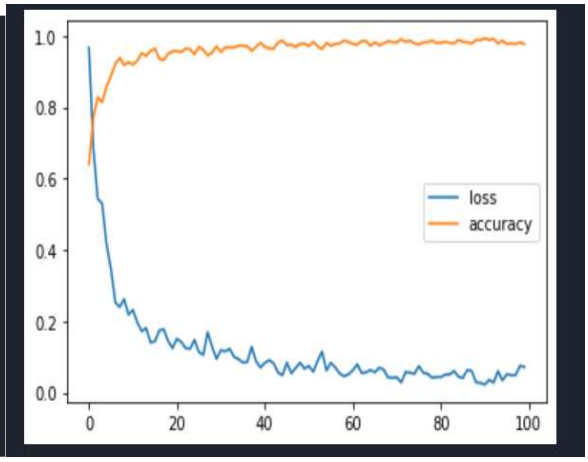
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

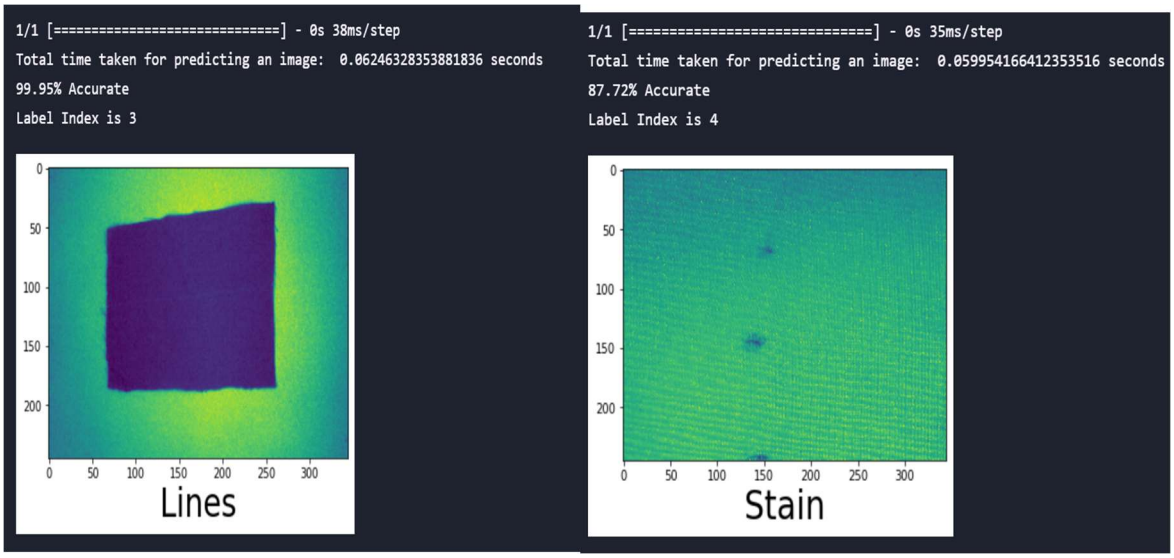
✓ 0.4s

```
Resampled dataset shape Counter({4: 212, 1: 164, 0: 146, 3: 118, 5: 101, 2: 38})
```

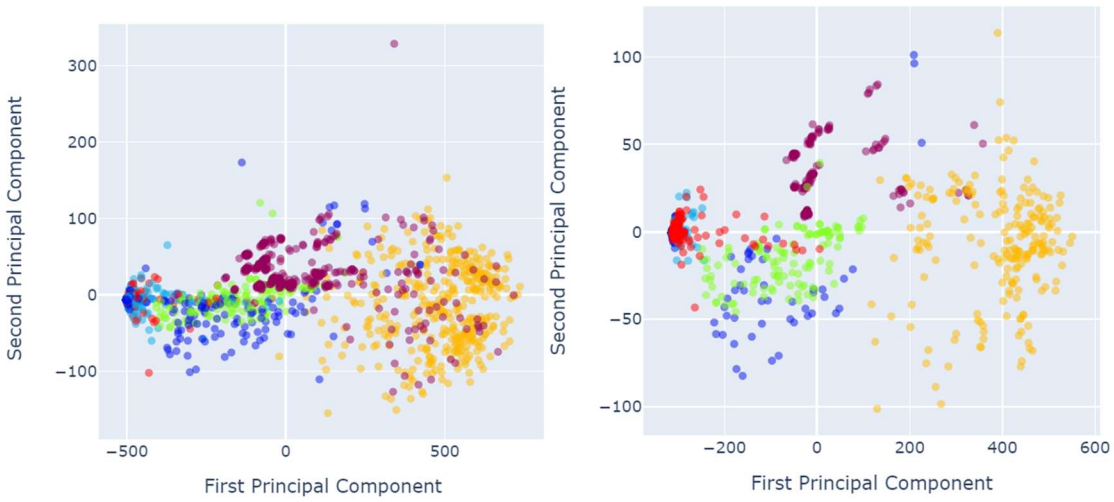
```
5/5 [==============================] - 8s 2s/step
              precision    recall  f1-score   support

           0       0.88      1.00      0.94        29
           1       0.97      0.97      0.97        33
           2       1.00      0.88      0.93         8
           3       0.95      1.00      0.98        20
           4       1.00      0.89      0.94        46
           5       0.95      1.00      0.98        20

    accuracy                           0.96       156
   macro avg       0.96      0.96      0.96       156
weighted avg       0.96      0.96      0.95       156
```



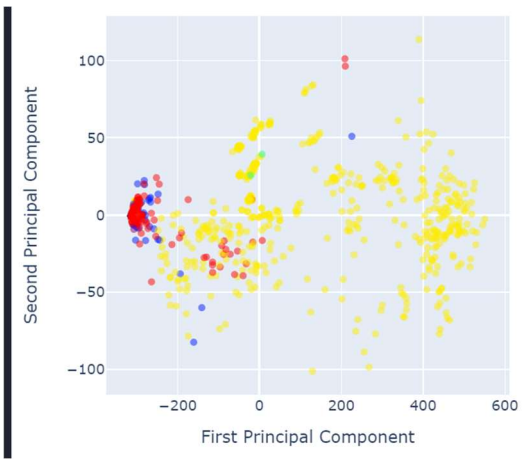




PCA Before and After Sampling



PCA After CNN



Appendix 2-5.1: AIKNN Size 200x800

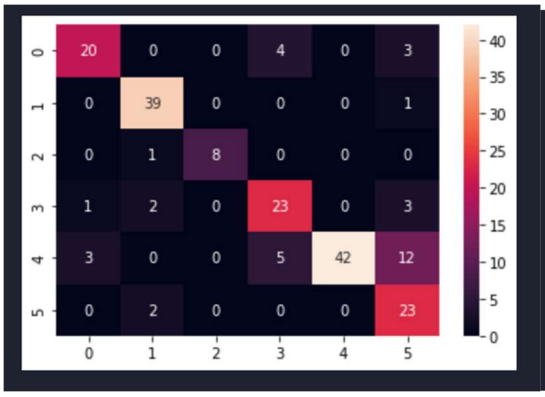
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

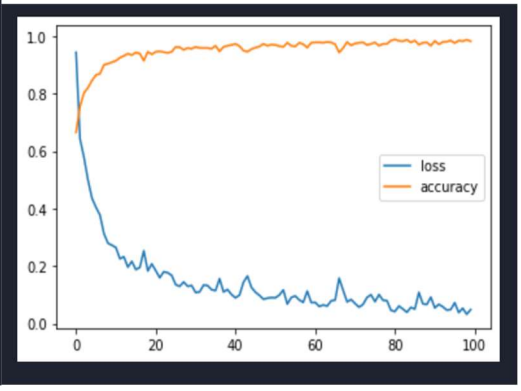
```
ust = AllKNN()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

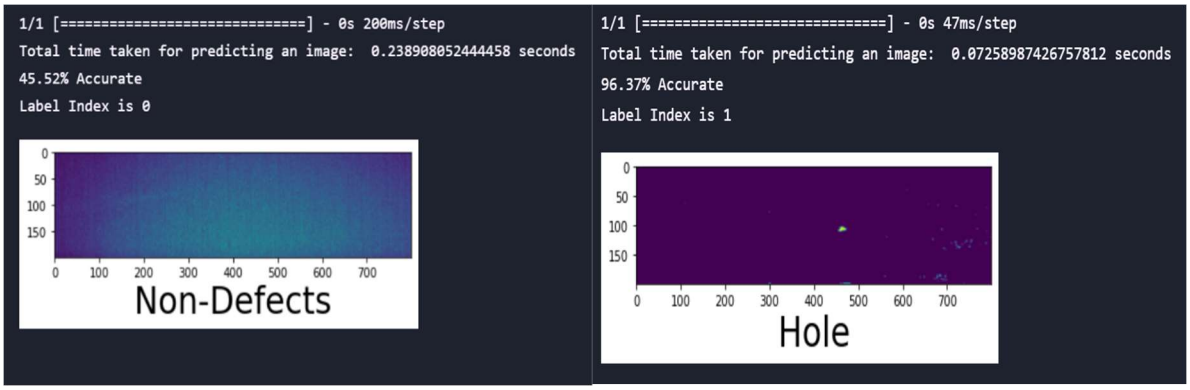
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({4: 304, 1: 204, 0: 152, 3: 129, 5: 101, 2: 70})
```

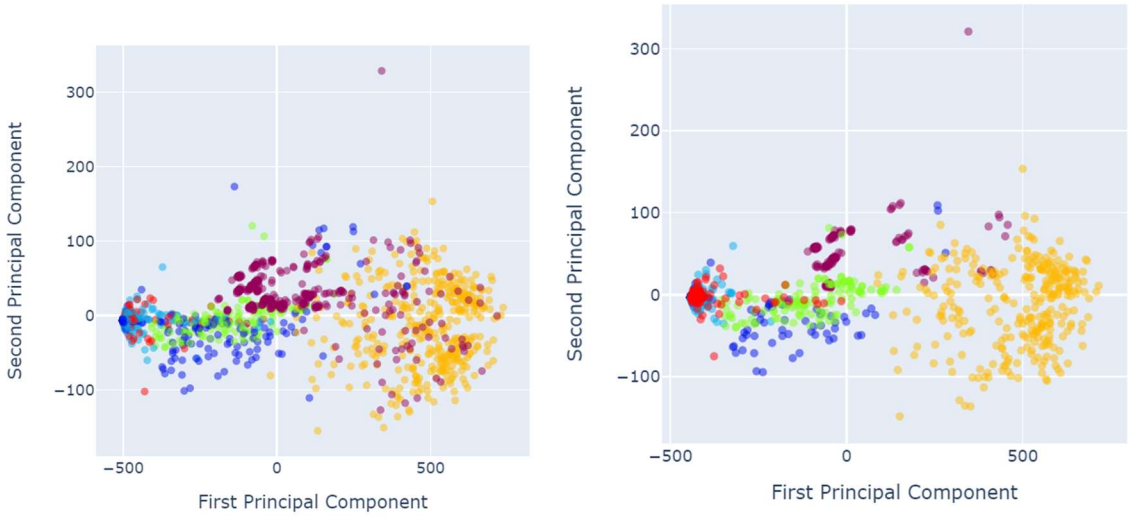
```
6/6 [==============================] - 5s 833ms/step
              precision    recall  f1-score   support

           0       0.83      0.74      0.78        27
           1       0.89      0.97      0.93        40
           2       1.00      0.89      0.94         9
           3       0.72      0.79      0.75        29
           4       1.00      0.68      0.81        62
           5       0.55      0.92      0.69        25

    accuracy                           0.81       192
   macro avg       0.83      0.83      0.82       192
weighted avg       0.85      0.81      0.81       192
```



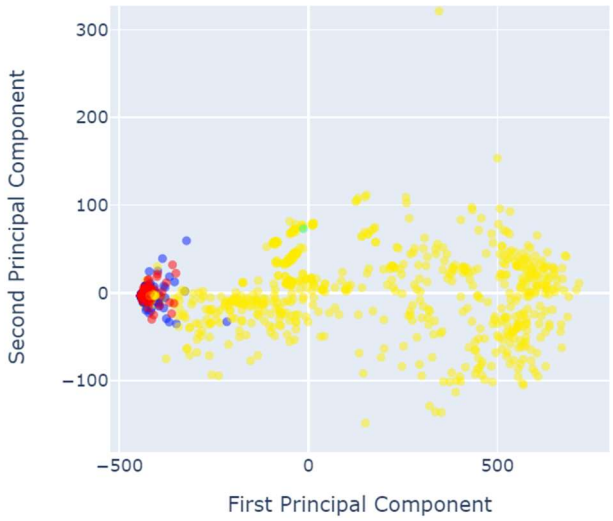




PCA Before and After Sampling



PCA After CNN



Appendix 2-5.2: AIKNN Size 150x700

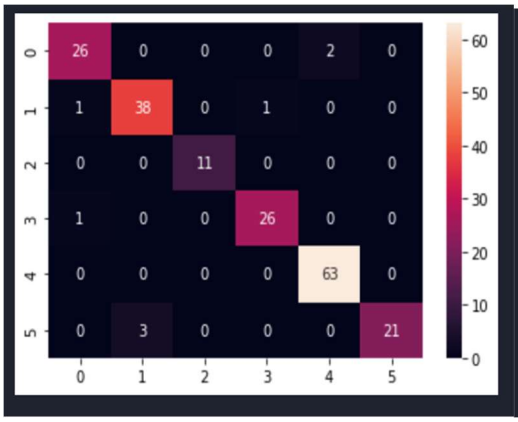
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

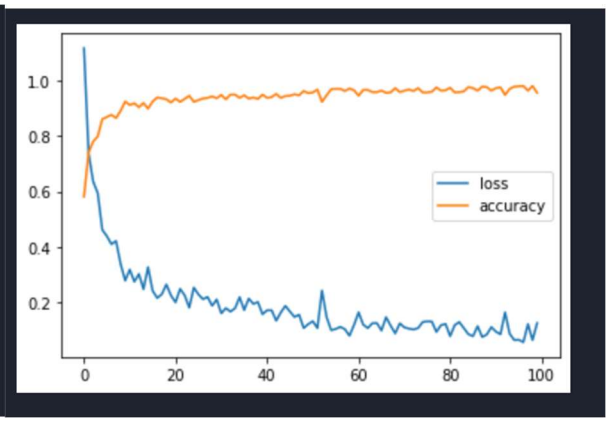
```
ust = AllKNN()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

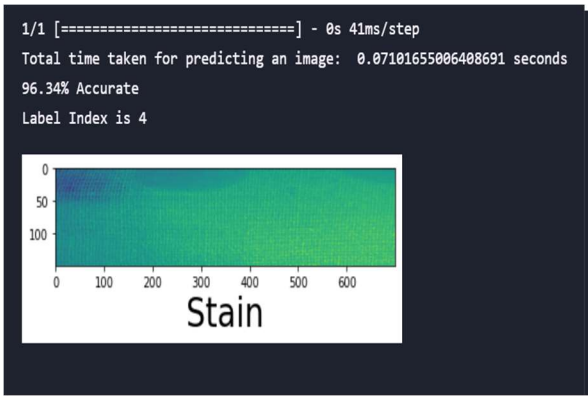
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

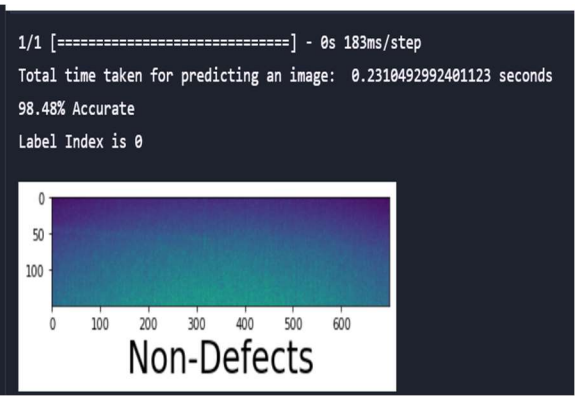
```
Resampled dataset shape Counter({4: 301, 1: 204, 0: 160, 3: 129, 5: 101, 2: 70})
```

```
7/7 [==============================] - 3s 440ms/step
              precision    recall  f1-score   support

           0       0.93      0.93      0.93        28
           1       0.93      0.95      0.94        40
           2       1.00      1.00      1.00        11
           3       0.96      0.96      0.96        27
           4       0.97      1.00      0.98        63
           5       1.00      0.88      0.93        24

    accuracy                           0.96       193
   macro avg       0.96      0.95      0.96       193
weighted avg       0.96      0.96      0.96       193
```

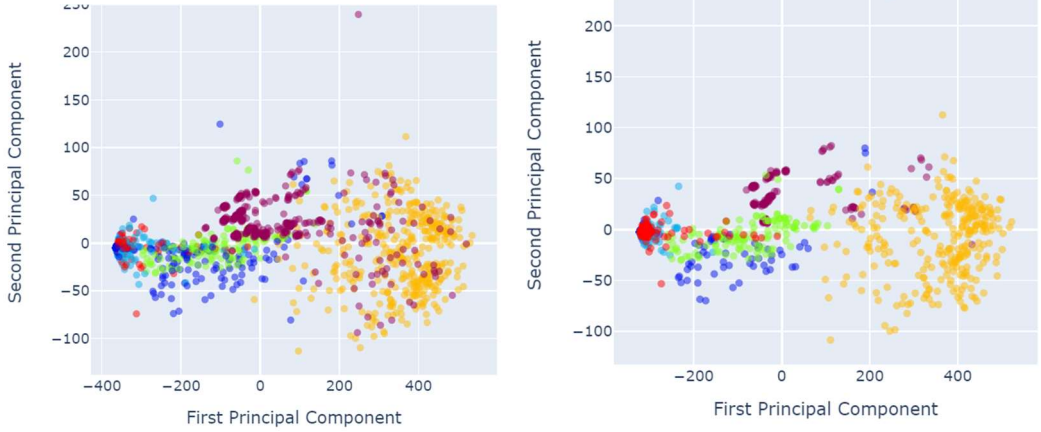




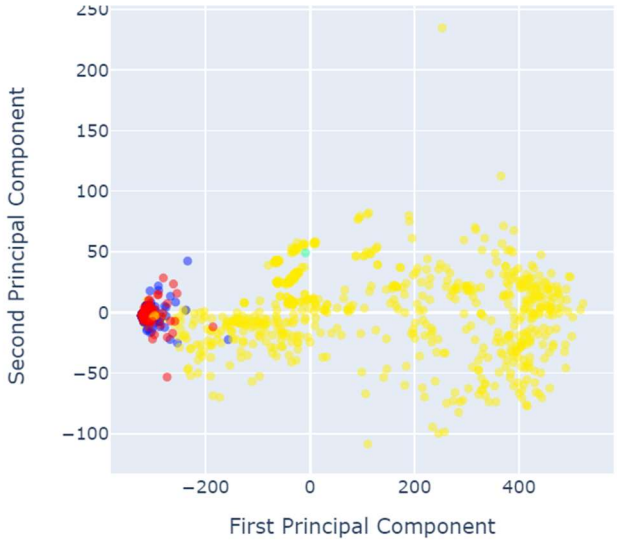




PCA Before and After Sampling



PCA After CNN



Appendix 2-5.3: AIKNN Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = AllKNN()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

✓ 17.9s

+ Code + Markdown

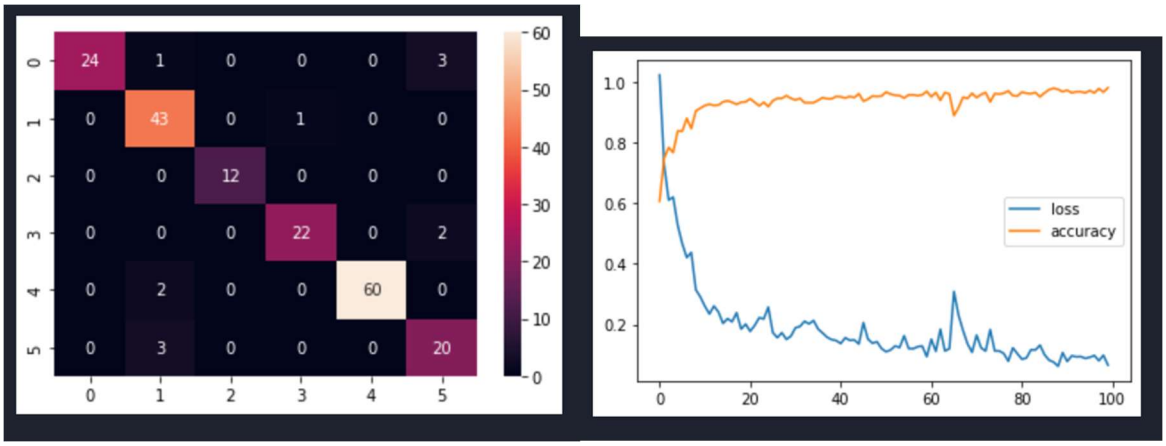
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

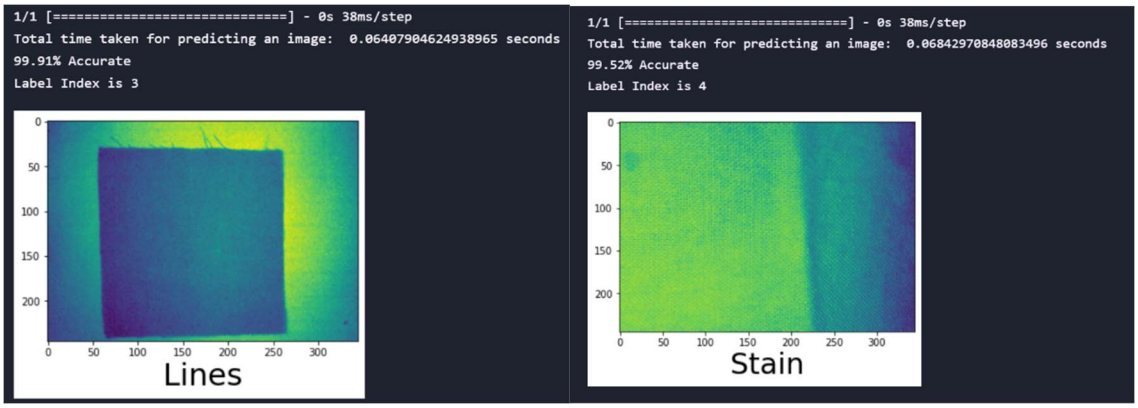
✓ 0.1s

```
Resampled dataset shape Counter({4: 301, 1: 204, 0: 160, 3: 129, 5: 101, 2: 70})
```

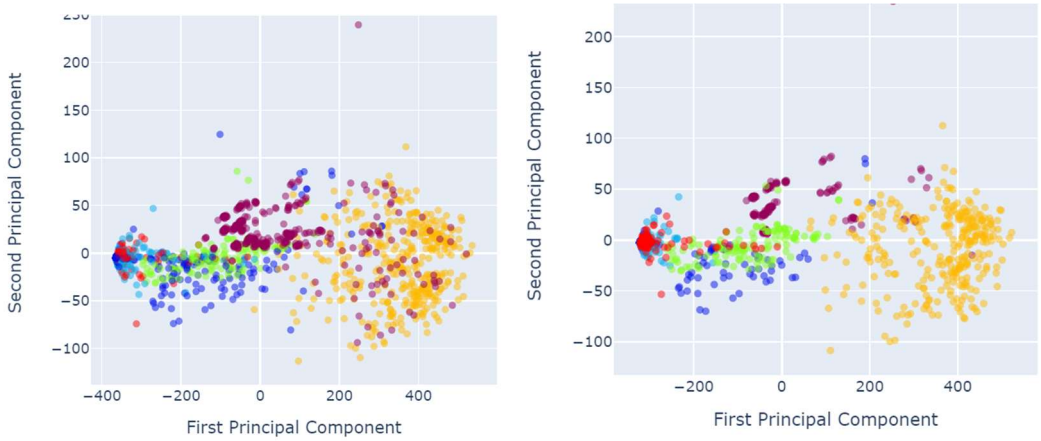
```
7/7 [==============================] - 3s 346ms/step
```

	precision	recall	f1-score	support
0	1.00	0.86	0.92	28
1	0.88	0.98	0.92	44
2	1.00	1.00	1.00	12
3	0.96	0.92	0.94	24
4	1.00	0.97	0.98	62
5	0.80	0.87	0.83	23
accuracy			0.94	193
macro avg	0.94	0.93	0.93	193
weighted avg	0.94	0.94	0.94	193

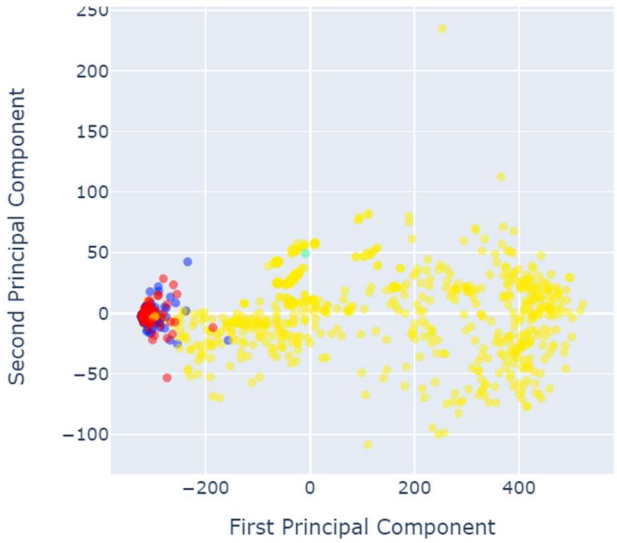




PCA Before and After Sampling



PCA After CNN



Appendix 2-6.1: Instance Hardness Threshold 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

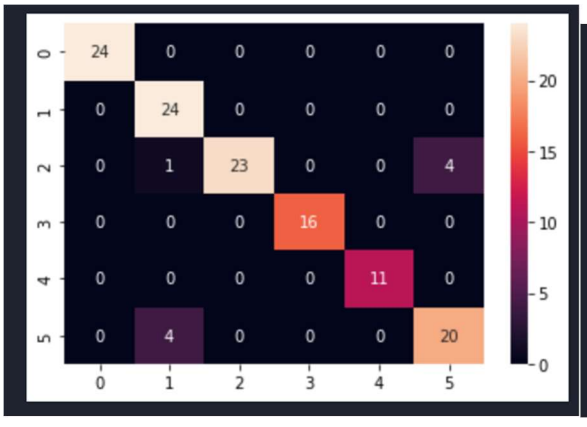
```
ust = InstanceHardnessThreshold(random_state=42)
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

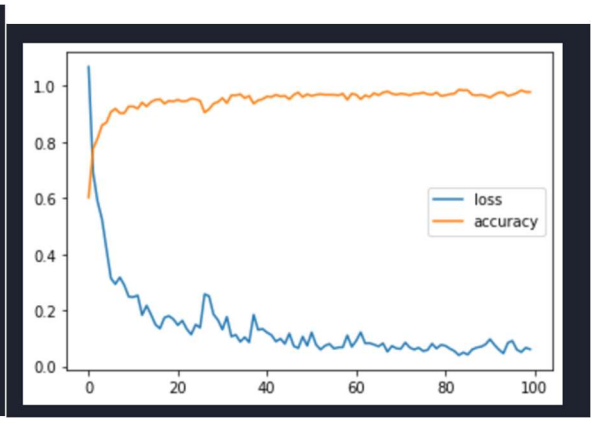
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

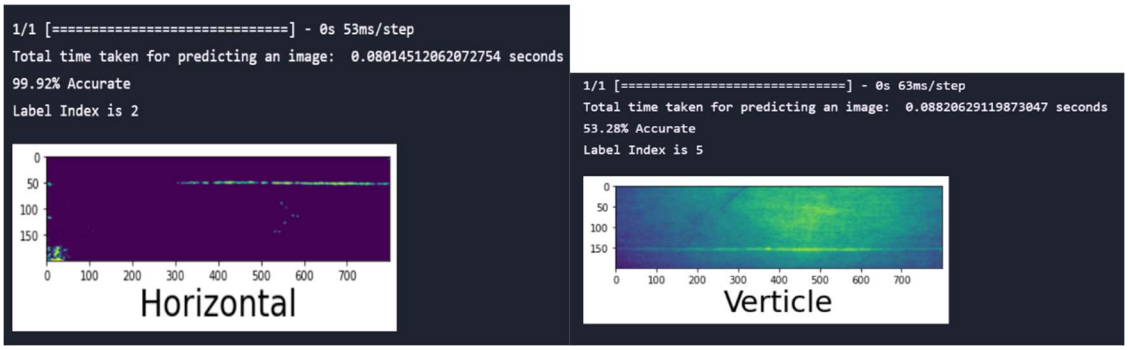
```
Resampled dataset shape Counter({1: 111, 0: 110, 4: 108, 2: 101, 3: 101, 5: 101})
```

```
4/4 [==============================] - 3s 802ms/step
```

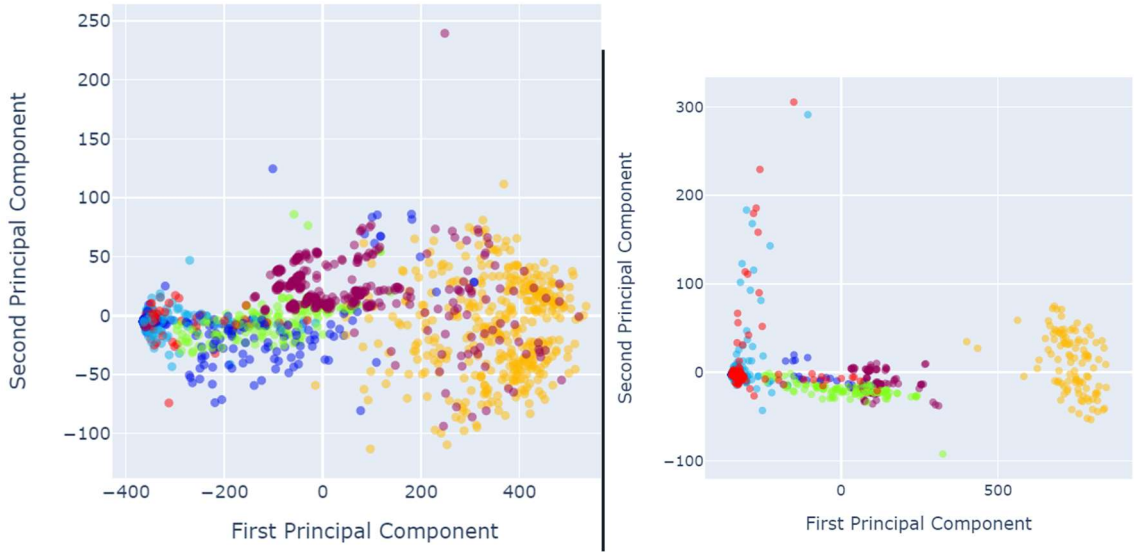
	precision	recall	f1-score	support
0	1.00	1.00	1.00	24
1	0.83	1.00	0.91	24
2	1.00	0.82	0.90	28
3	1.00	1.00	1.00	16
4	1.00	1.00	1.00	11
5	0.83	0.83	0.83	24
accuracy			0.93	127
macro avg	0.94	0.94	0.94	127
weighted avg	0.94	0.93	0.93	127



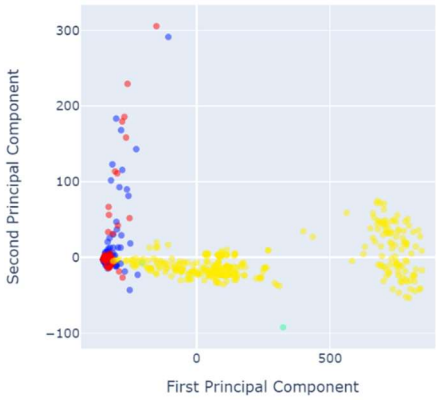




PCA Before and After Sampling



PCA After CNN



Appendix 2-6.2: Instance Hardness Threshold Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

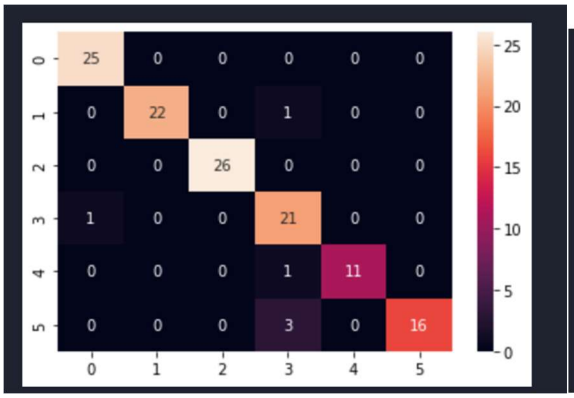
```
ust = InstanceHardnessThreshold(random_state=42)
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

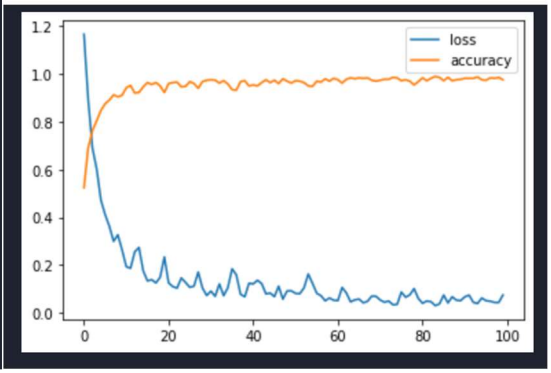
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({0: 111, 1: 108, 4: 108, 3: 104, 2: 101, 5: 101})
```

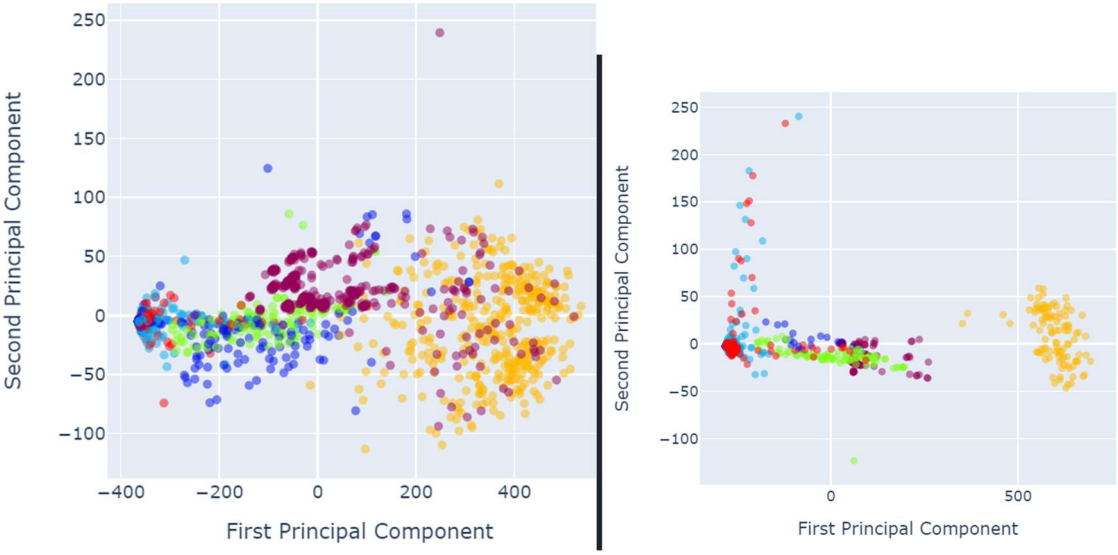
```
4/4 [==============================] - 2s 513ms/step
```

	precision	recall	f1-score	support
0	0.96	1.00	0.98	25
1	1.00	0.96	0.98	23
2	1.00	1.00	1.00	26
3	0.81	0.95	0.88	22
4	1.00	0.92	0.96	12
5	1.00	0.84	0.91	19
accuracy			0.95	127
macro avg	0.96	0.94	0.95	127
weighted avg	0.96	0.95	0.95	127

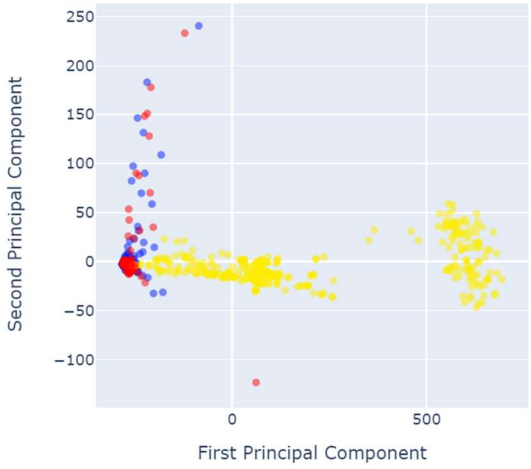




PCA Before and After Sampling



PCA After CNN



Appendix 2-6.3: Instance Hardness Threshold Size 245x345

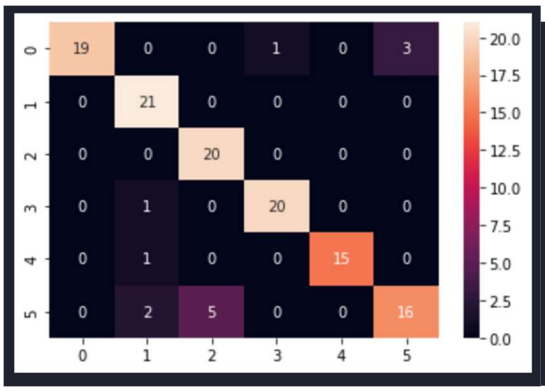
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

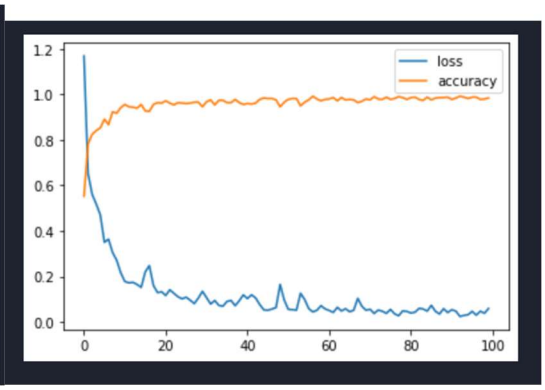
```
ust = InstanceHardnessThreshold(random_state=42)
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

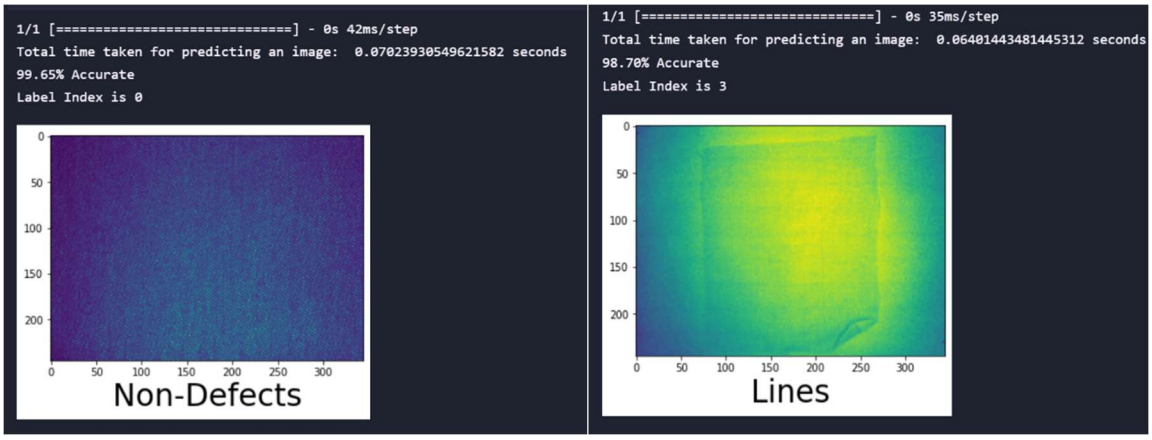
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({1: 108, 0: 107, 4: 102, 2: 101, 3: 101, 5: 101})
```

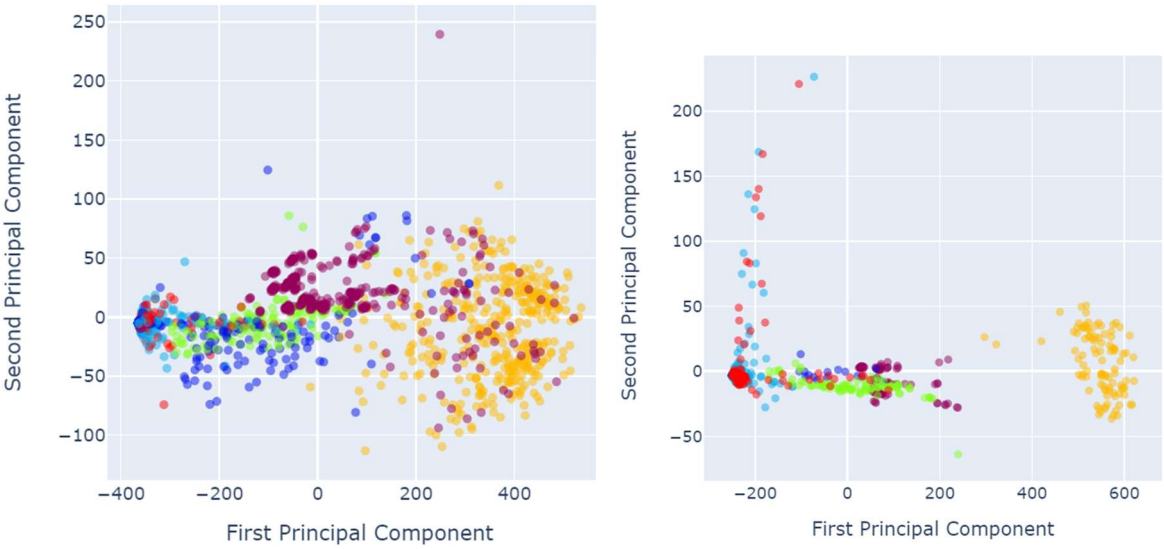
```
4/4 [==============================] - 2s 398ms/step
              precision    recall  f1-score   support

           0       1.00      0.83      0.90        23
           1       0.84      1.00      0.91        21
           2       0.80      1.00      0.89        20
           3       0.95      0.95      0.95        21
           4       1.00      0.94      0.97        16
           5       0.84      0.70      0.76        23

    accuracy                           0.90       124
   macro avg       0.91      0.90      0.90       124
weighted avg       0.90      0.90      0.89       124
```



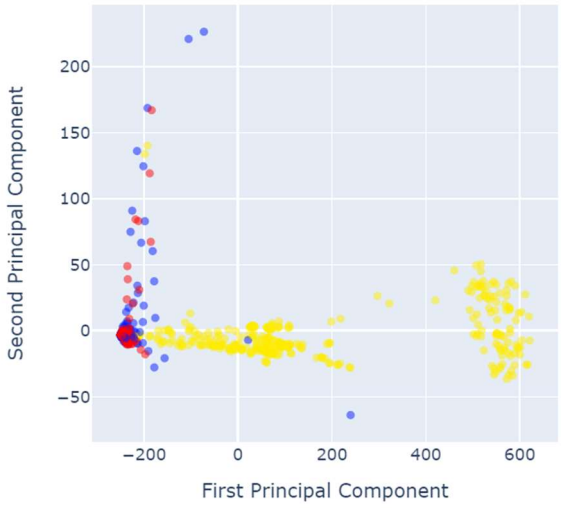




PCA Before and After Sampling



PCA After CNN



Appendix 2-7.1: Near Miss Undersampling Size 200x800

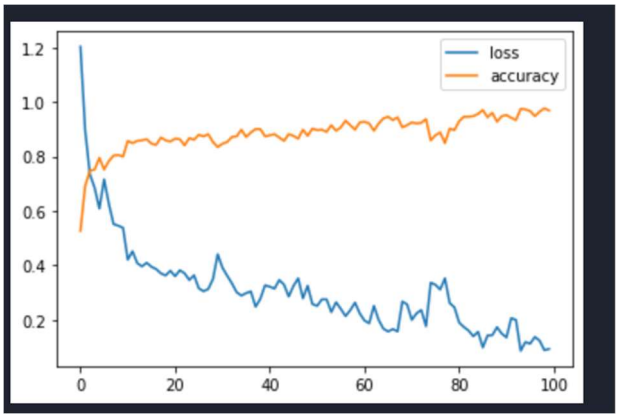
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = NearMiss ()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

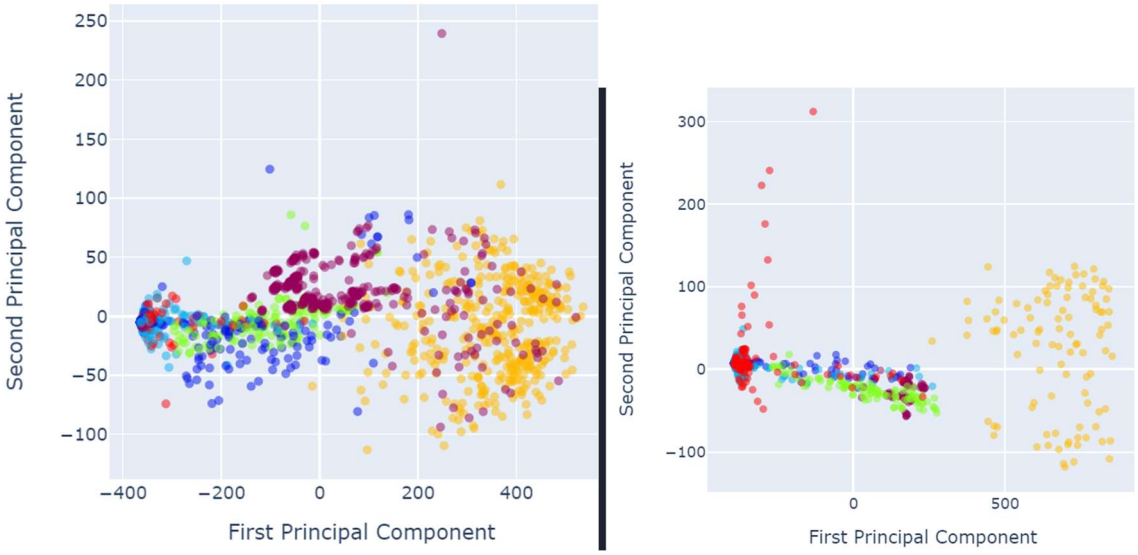
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

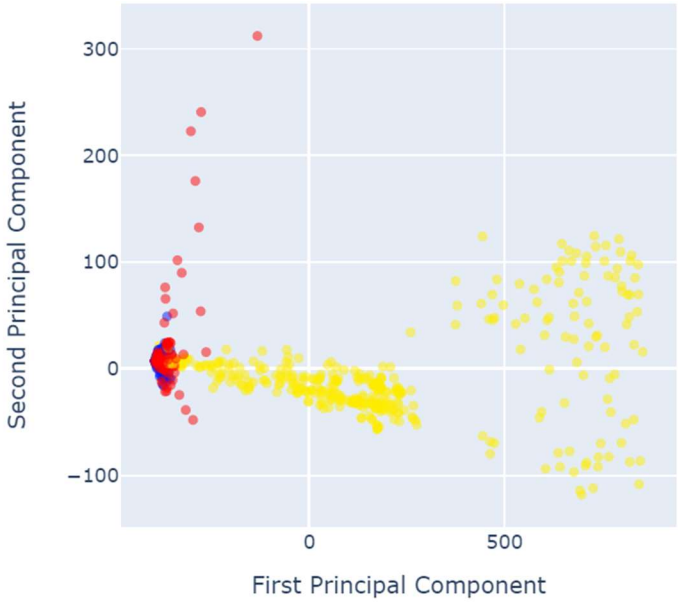
```
4/4 [==============================] - 3s 762ms/step
              precision    recall  f1-score   support

           0       0.95      0.91      0.93        23
           1       0.55      0.95      0.70        22
           2       0.89      0.68      0.77        25
           3       1.00      0.75      0.86        12
           4       1.00      0.62      0.77        16
           5       0.83      0.83      0.83        24

    accuracy                           0.80       122
   macro avg       0.87      0.79      0.81       122
weighted avg       0.86      0.80      0.81       122
```



PCA Before and After Sampling



PCA After CNN



Appendix 2-7.2: Near Miss Undersampling Size 150x700

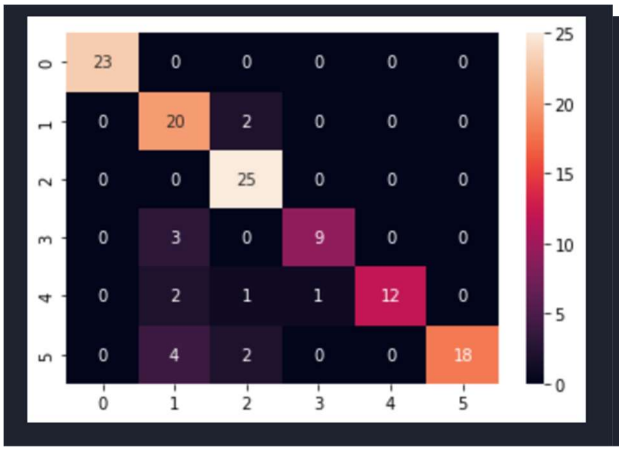
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

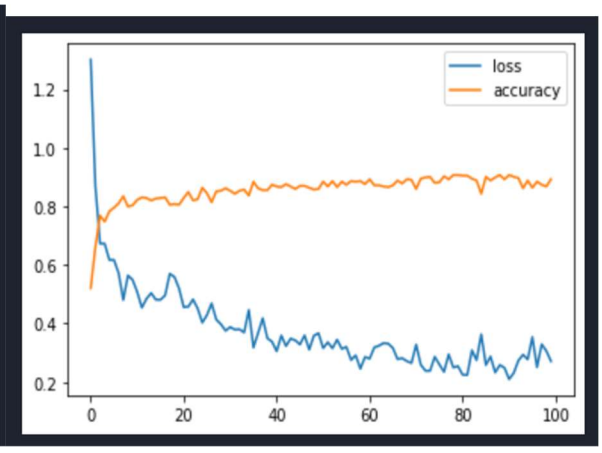
```
ust = NearMiss ()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

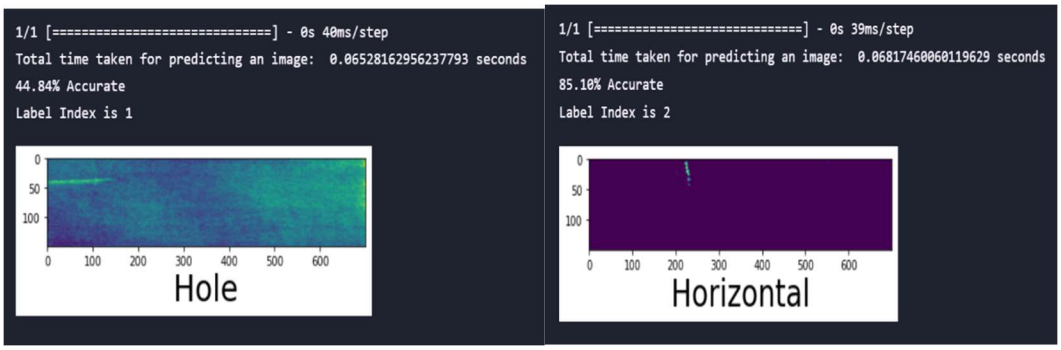
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

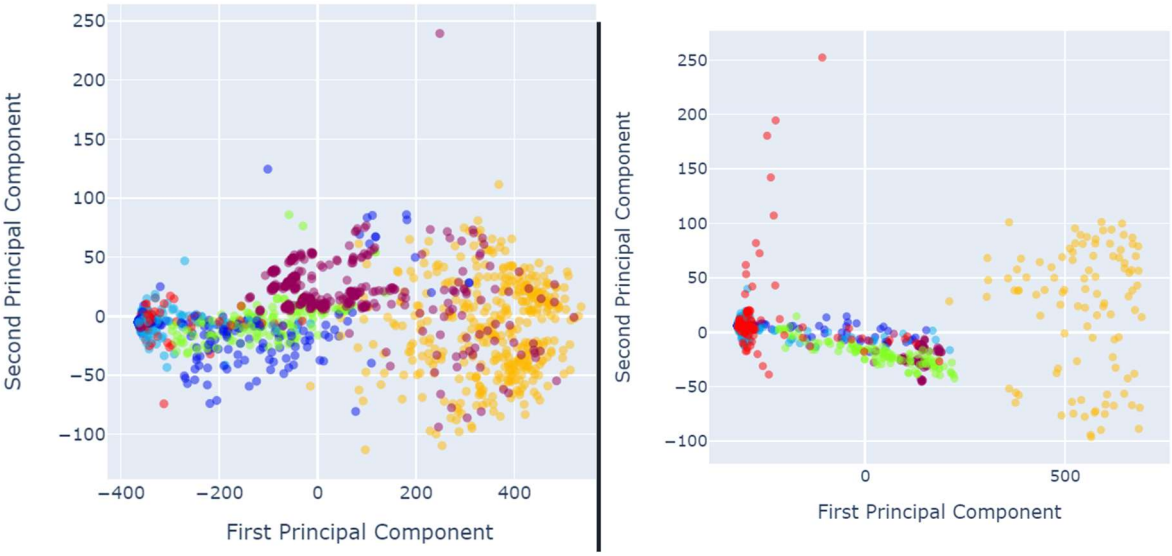
```
4/4 [==============================] - 3s 745ms/step
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        23
           1       0.69      0.91      0.78        22
           2       0.83      1.00      0.91        25
           3       0.90      0.75      0.82        12
           4       1.00      0.75      0.86        16
           5       1.00      0.75      0.86        24

    accuracy                           0.88       122
   macro avg       0.90      0.86      0.87       122
weighted avg       0.90      0.88      0.88       122
```



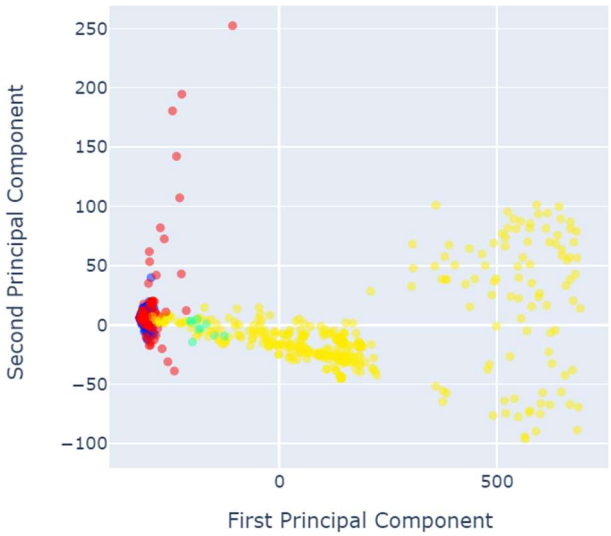




PCA Before and After Sampling



PCA After CNN



Appendix 2-7.3: Near Miss Undersampling Size 245x345

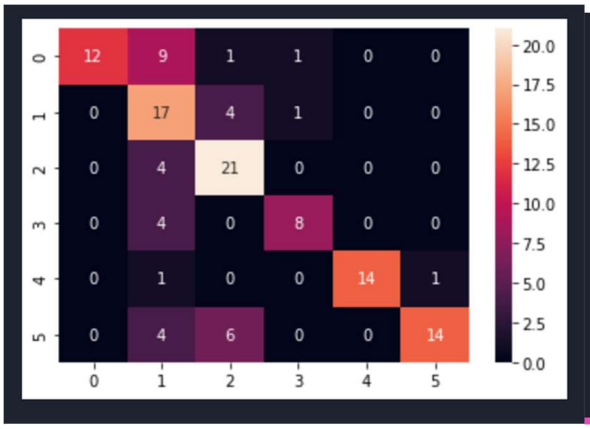
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

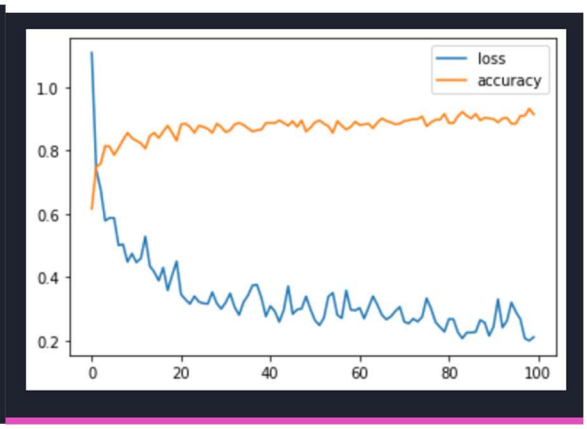
```
ust = NearMiss ()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

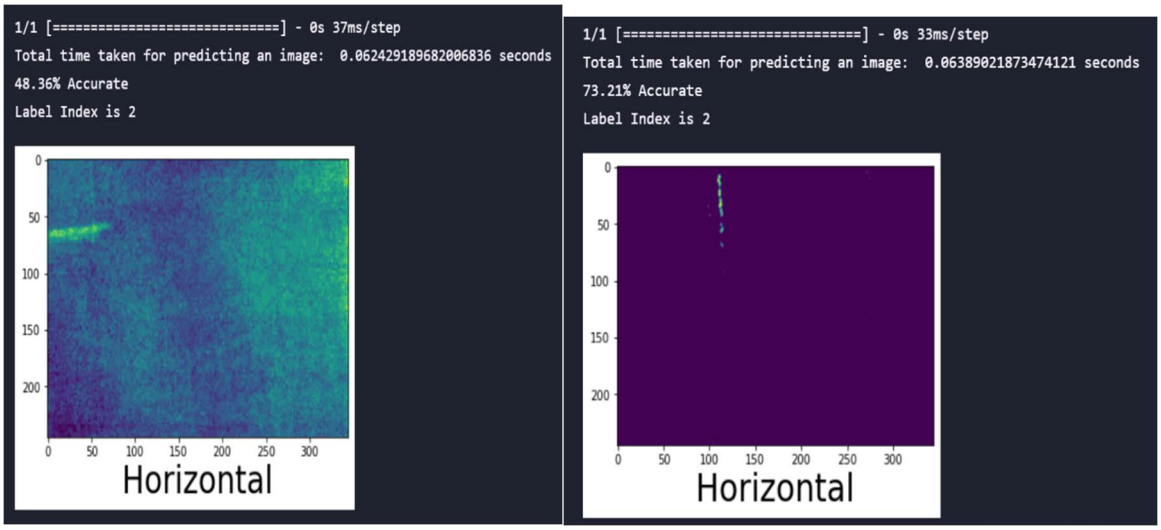
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

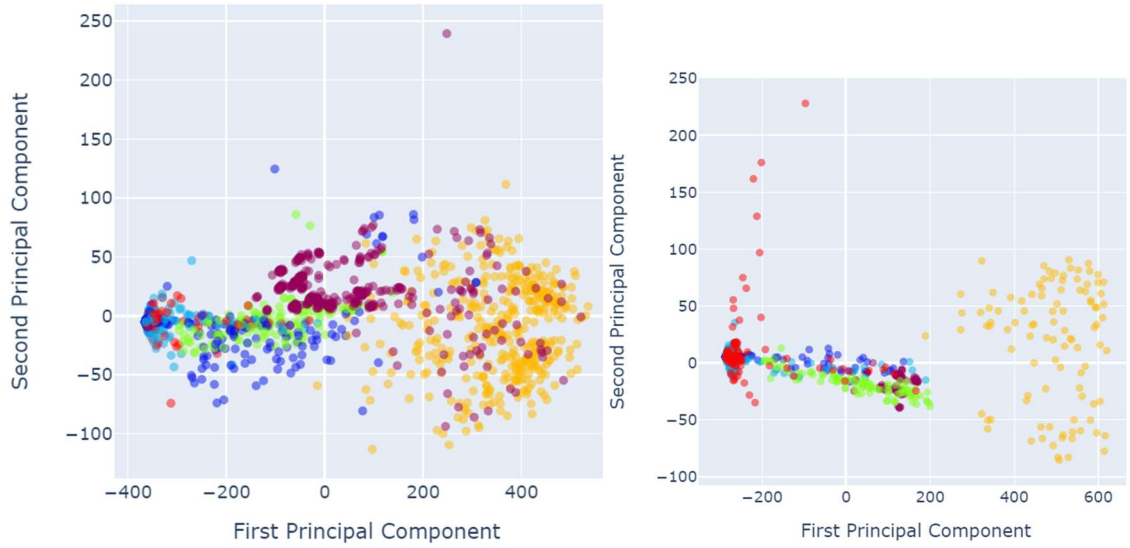
0	1.00	0.52	0.69	23
1	0.44	0.77	0.56	22
2	0.66	0.84	0.74	25
3	0.80	0.67	0.73	12
4	1.00	0.88	0.93	16
5	0.93	0.58	0.72	24
accuracy			0.70	122
macro avg	0.80	0.71	0.73	122
weighted avg	0.80	0.70	0.72	122



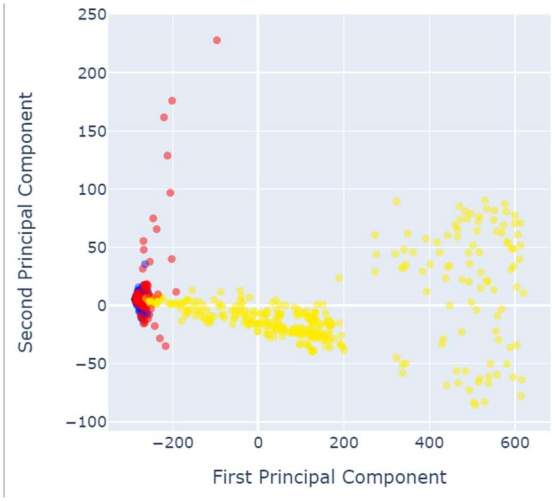




PCA Before and After Sampling



PCA After CNN



Appendix 2-8.1: Neighbourhood Cleaning Rule Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = NeighbourhoodCleaningRule()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

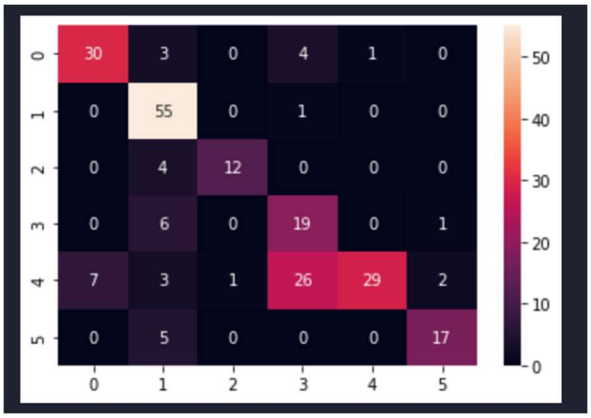
✓ 30.6s

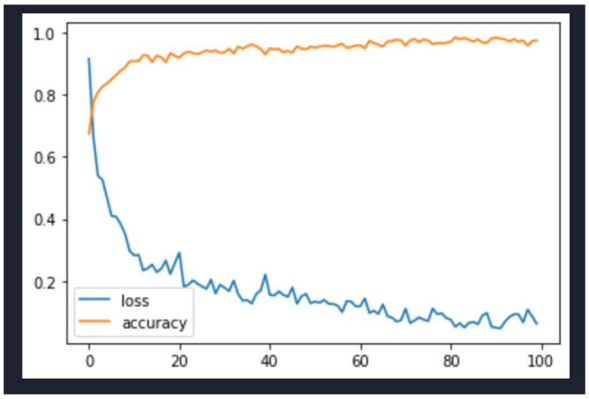
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

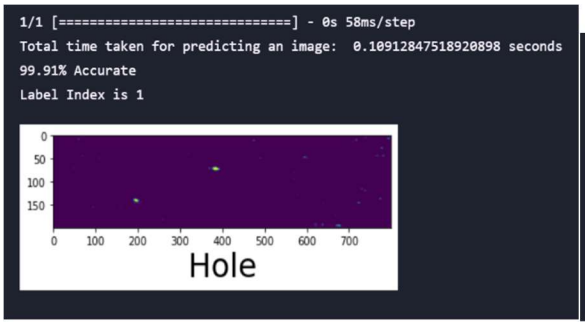
✓ 0.1s

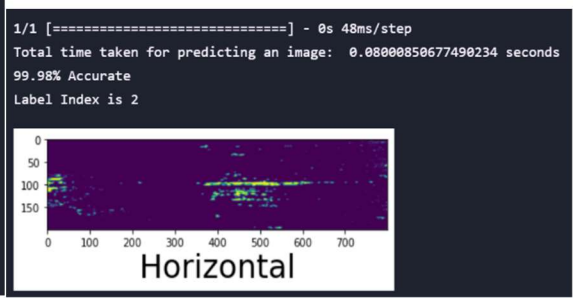
```
Resampled dataset shape Counter({4: 362, 1: 237, 0: 190, 3: 138, 5: 101, 2: 99})
```

```
8/8 [==============================] - 7s 806ms/step
              precision    recall  f1-score   support

           0       0.81      0.79      0.80        38
           1       0.72      0.98      0.83        56
           2       0.92      0.75      0.83        16
           3       0.38      0.73      0.50        26
           4       0.97      0.43      0.59        68
           5       0.85      0.77      0.81        22

    accuracy                           0.72       226
   macro avg       0.78      0.74      0.73       226
weighted avg       0.80      0.72      0.71       226
```

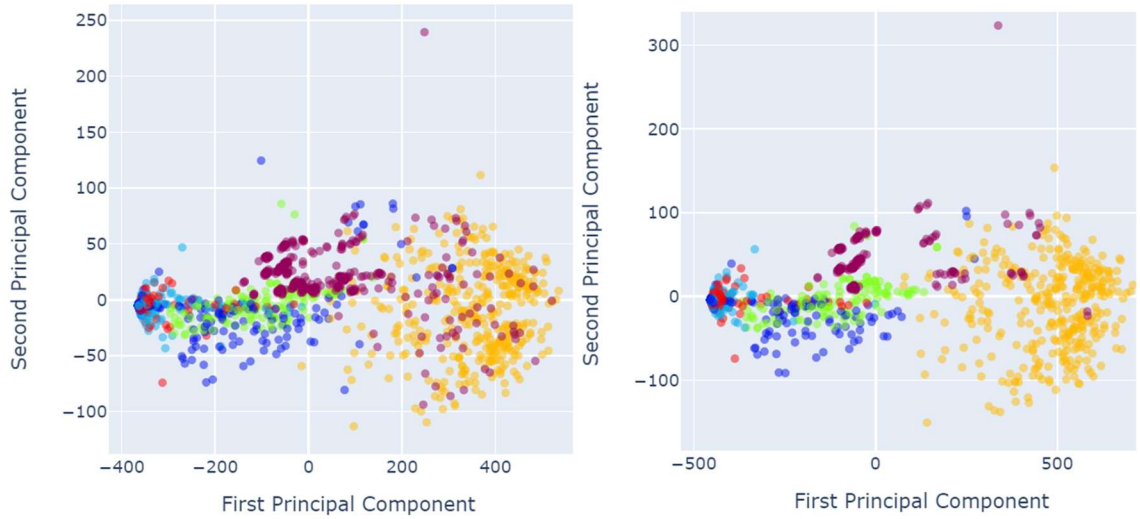




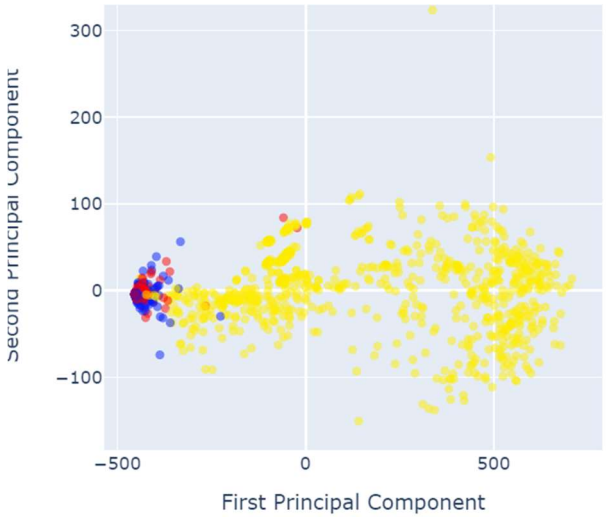




PCA Before and After Sampling



PCA After CNN



Appendix 2-8.2: Neighbourhood Cleaning Rule Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = NeighbourhoodCleaningRule()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

✓ 12.7s

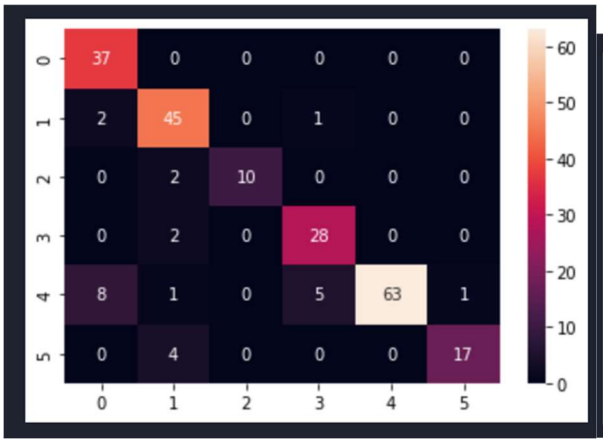
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

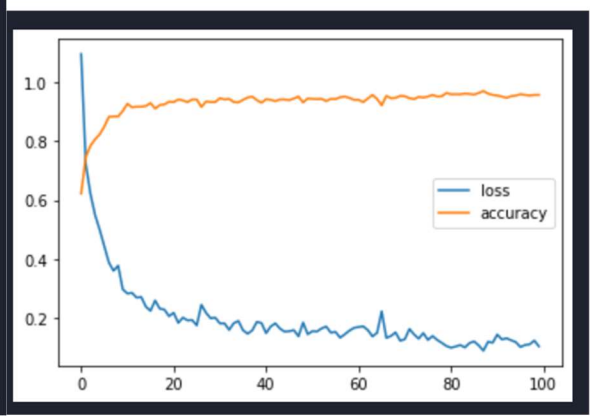
✓ 0.3s

```
Resampled dataset shape Counter({4: 362, 1: 237, 0: 192, 3: 138, 5: 101, 2: 100})
```

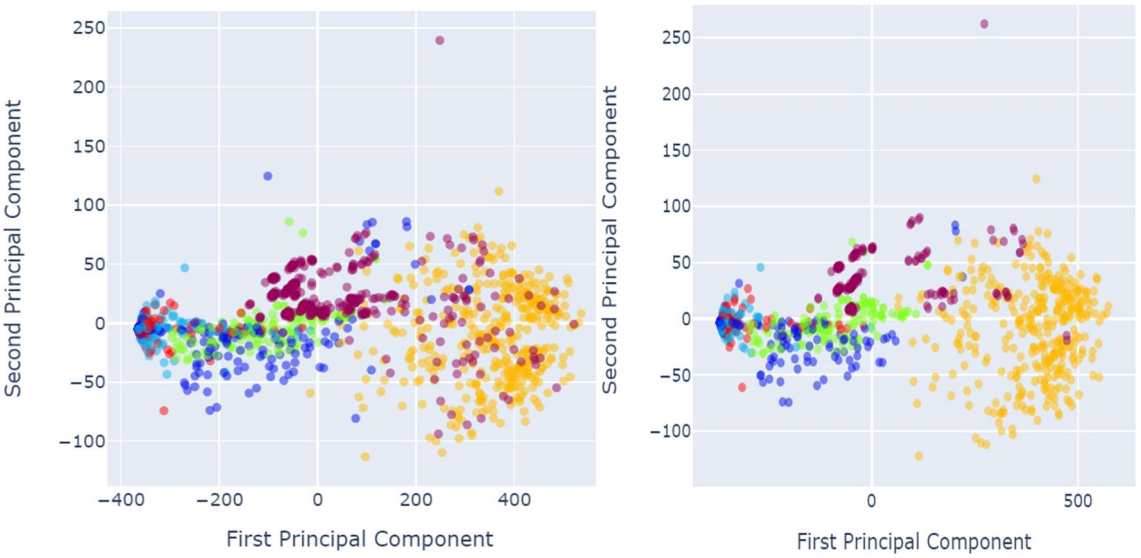
```
8/8 [==============================] - 4s 454ms/step
```

	precision	recall	f1-score	support
0	0.79	1.00	0.88	37
1	0.83	0.94	0.88	48
2	1.00	0.83	0.91	12
3	0.82	0.93	0.87	30
4	1.00	0.81	0.89	78
5	0.94	0.81	0.87	21
accuracy			0.88	226
macro avg	0.90	0.89	0.89	226
weighted avg	0.90	0.88	0.89	226

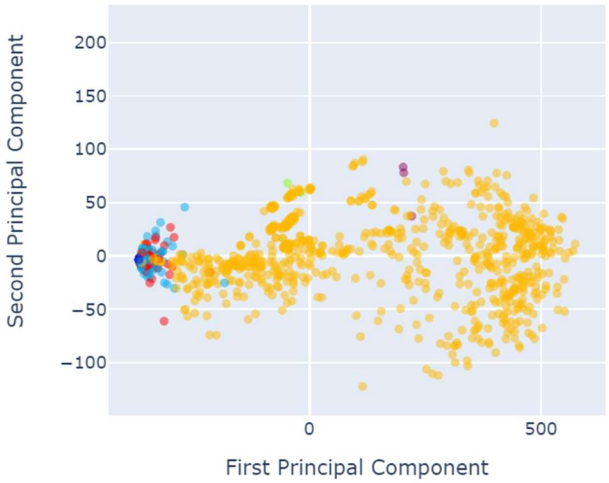




PCA Before and After Sampling



PCA After CNN



Appendix 2-8.3: Neighbourhood Cleaning Rule Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = NeighbourhoodCleaningRule()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

✓ 10.2s

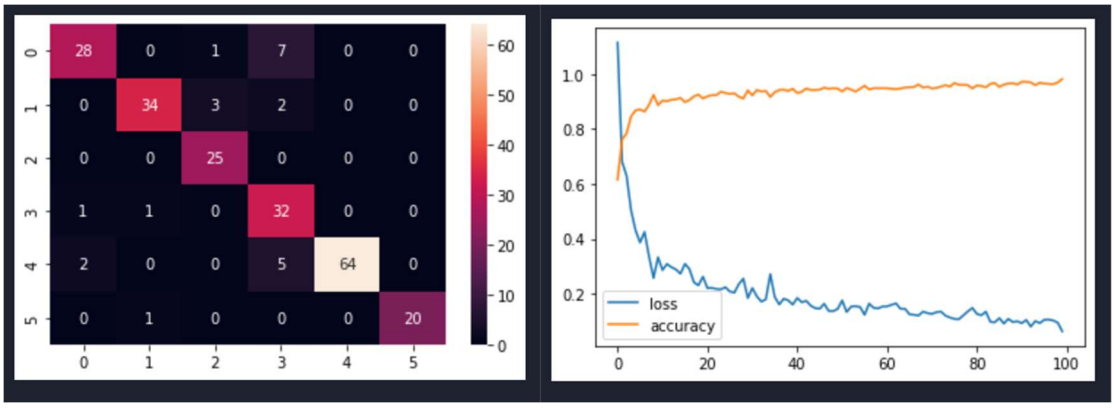
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

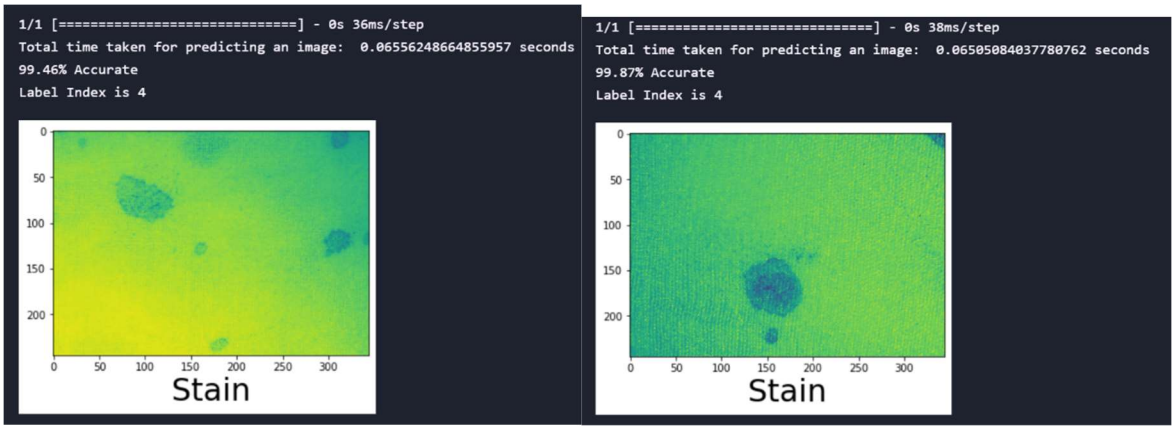
✓ 0.5s

```
Resampled dataset shape Counter({4: 362, 1: 237, 0: 192, 3: 138, 5: 101, 2: 99})
```

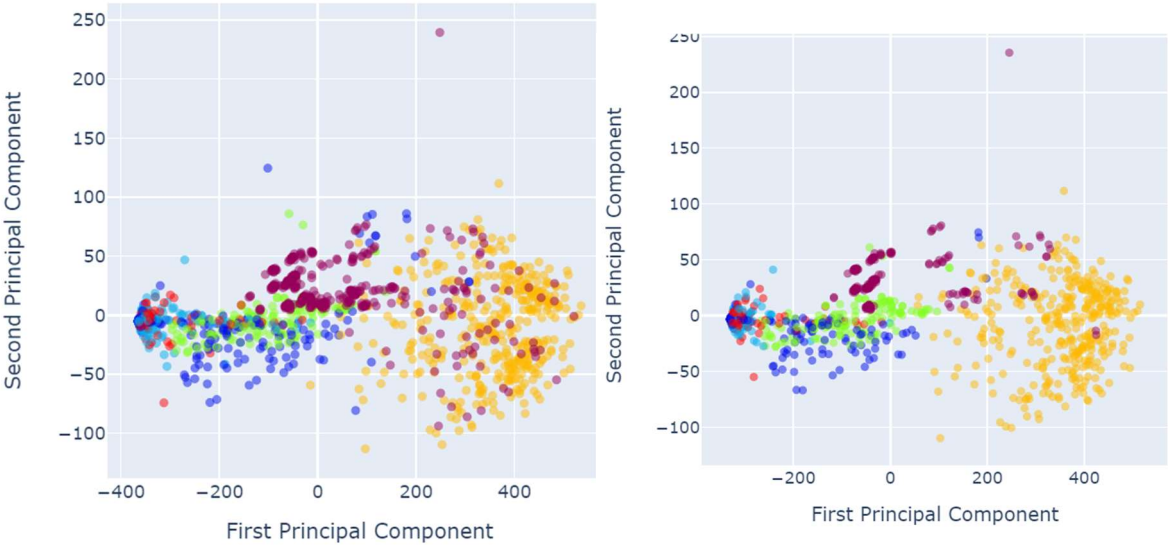
```
8/8 [==============================] - 3s 357ms/step
```

	precision	recall	f1-score	support
0	0.90	0.78	0.84	36
1	0.94	0.87	0.91	39
2	0.86	1.00	0.93	25
3	0.70	0.94	0.80	34
4	1.00	0.90	0.95	71
5	1.00	0.95	0.98	21
accuracy			0.90	226
macro avg	0.90	0.91	0.90	226
weighted avg	0.91	0.90	0.90	226

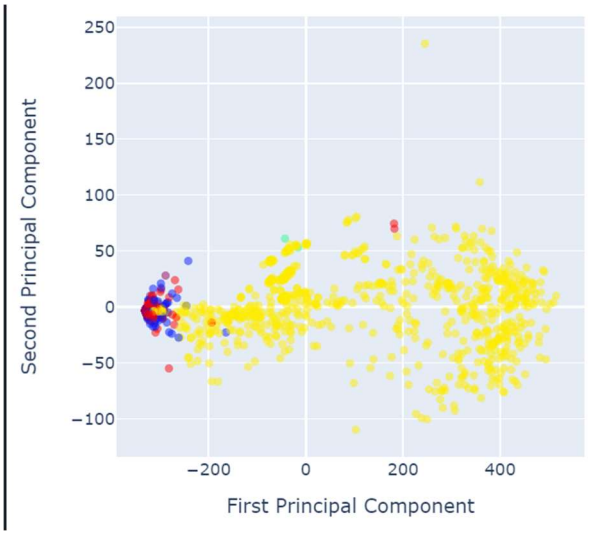




PCA Before and After Sampling



PCA After CNN



Appendix 2-9.1: TomLinks Undersampling 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

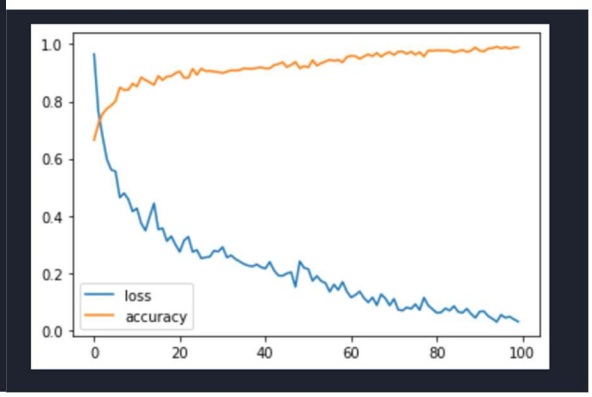
```
ust = TomekLinks()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```
✓ 4.6s

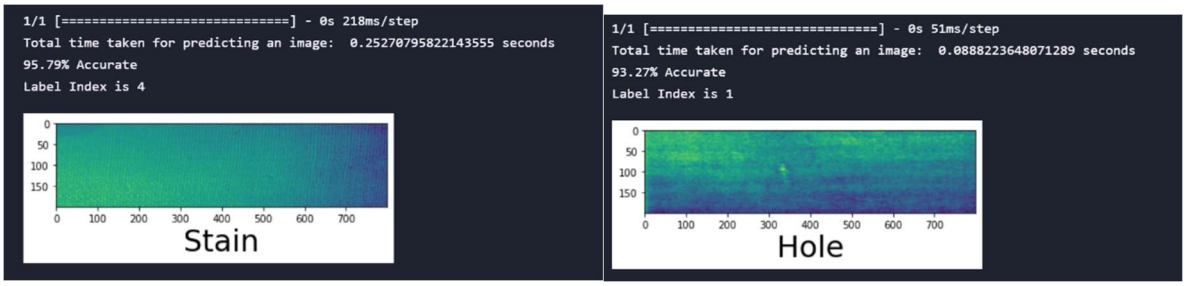
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```
✓ 0.4s

```
Resampled dataset shape Counter({0: 406, 4: 387, 1: 267, 3: 152, 2: 124, 5: 101})
```

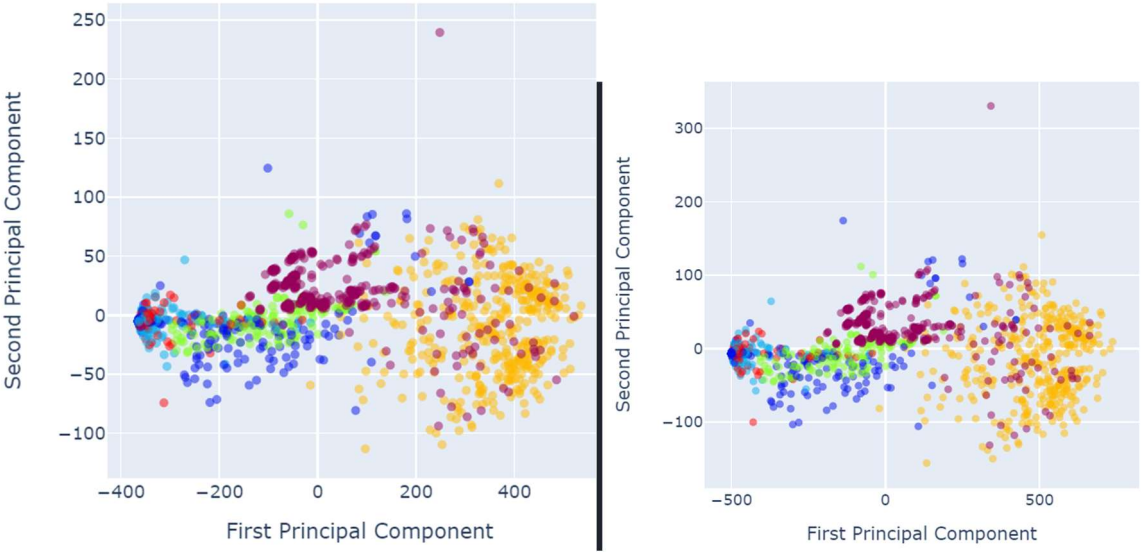
```
9/9 [==============================] - 8s 833ms/step
```

	precision	recall	f1-score	support
0	0.90	0.88	0.89	86
1	0.66	0.96	0.78	49
2	0.87	0.74	0.80	27
3	1.00	0.88	0.93	24
4	0.85	0.68	0.75	74
5	0.90	0.96	0.93	28
accuracy			0.84	288
macro avg	0.86	0.85	0.85	288
weighted avg	0.85	0.84	0.84	288

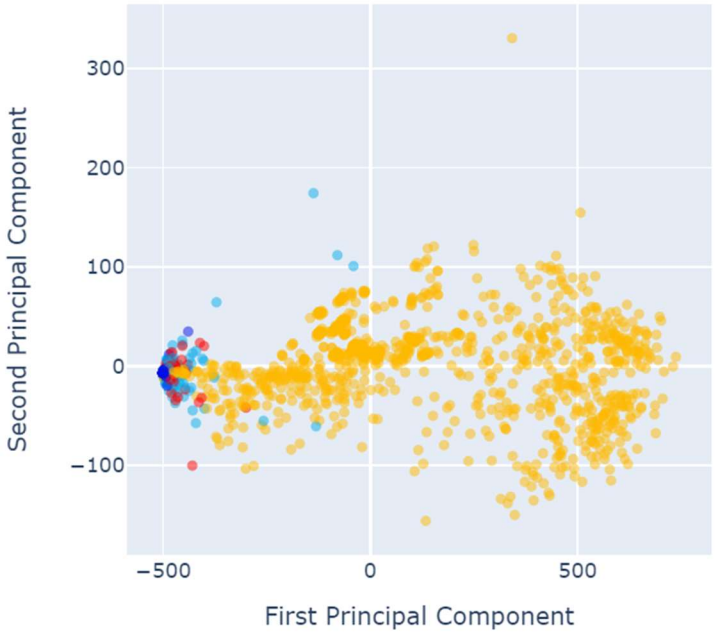




PCA Before and After Sampling



PCA After CNN



Appendix 2-9.2: TomLinks Undersampling Size 150x700

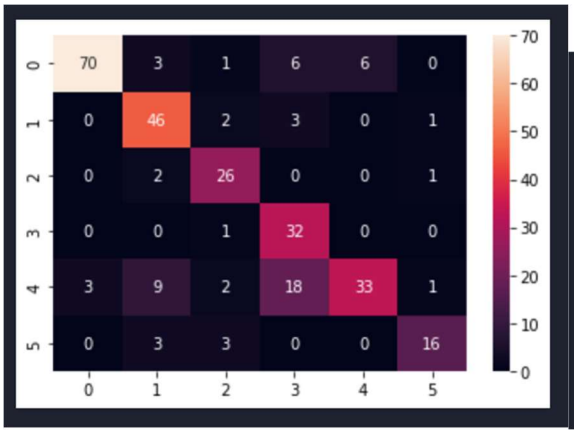
```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

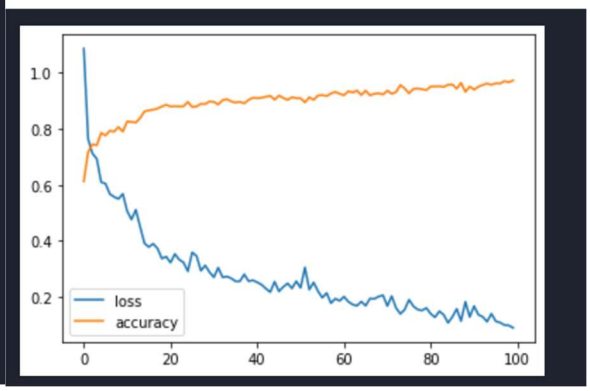
```
ust = TomekLinks()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

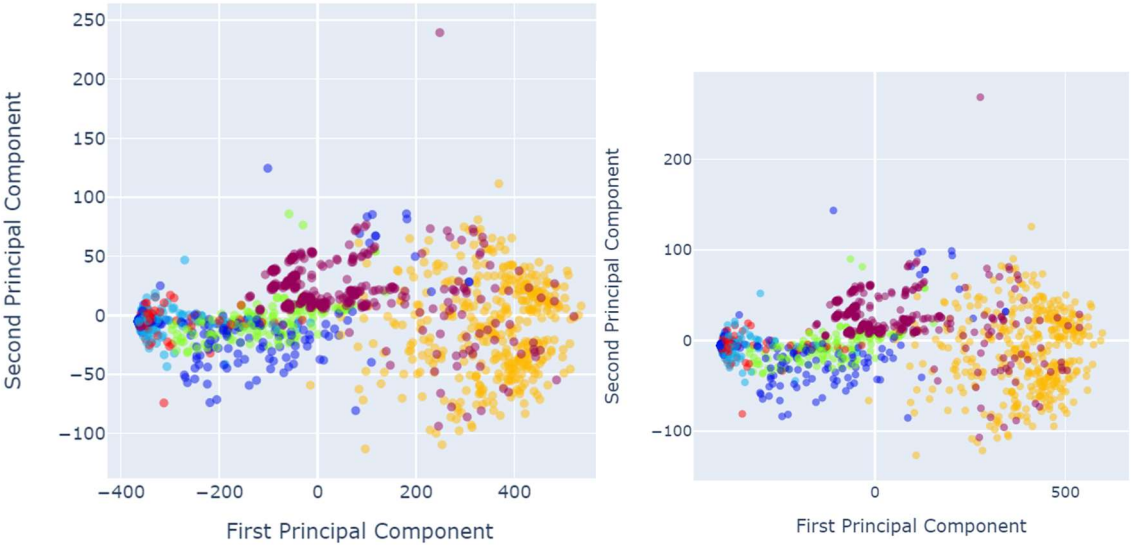
```
Resampled dataset shape Counter({0: 406, 4: 387, 1: 267, 3: 152, 2: 124, 5: 101})
```

```
9/9 [==============================] - 5s 531ms/step
              precision    recall  f1-score   support

           0       0.96      0.81      0.88        86
           1       0.73      0.88      0.80        52
           2       0.74      0.90      0.81        29
           3       0.54      0.97      0.70        33
           4       0.85      0.50      0.63        66
           5       0.84      0.73      0.78        22

    accuracy                           0.77       288
   macro avg       0.78      0.80      0.77       288
weighted avg       0.81      0.77      0.77       288
```

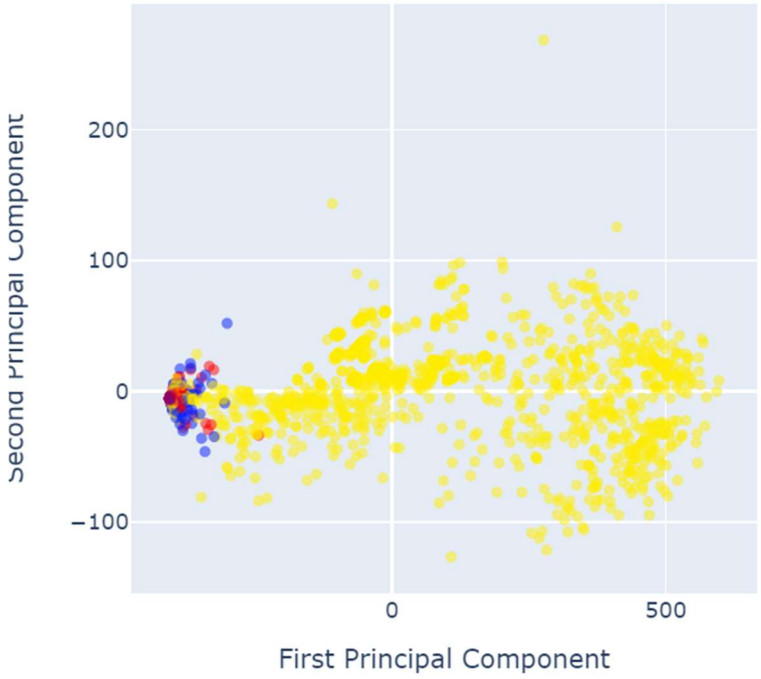




PCA Before and After Sampling



PCA After CNN



Appendix 2-9.3: TomLinks Undersampling 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ust = TomekLinks()
X_ust, y_ust = ust.fit_resample(X_res, y_res)
```

✓ 2.9s

+ Code + Markdown

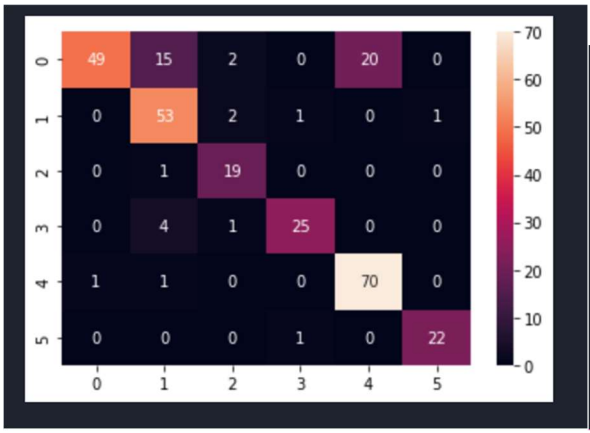
```
print(f'Resampled dataset shape {Counter(y_ust)}')
```

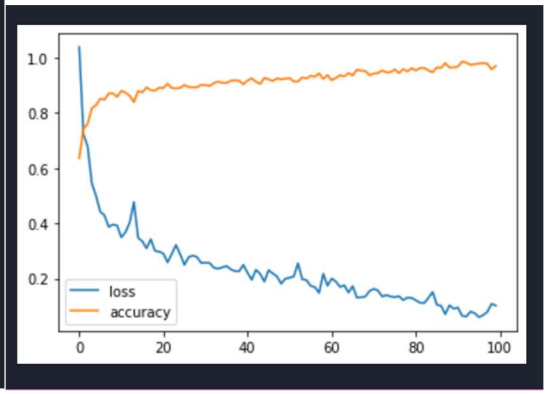
✓ 0.4s

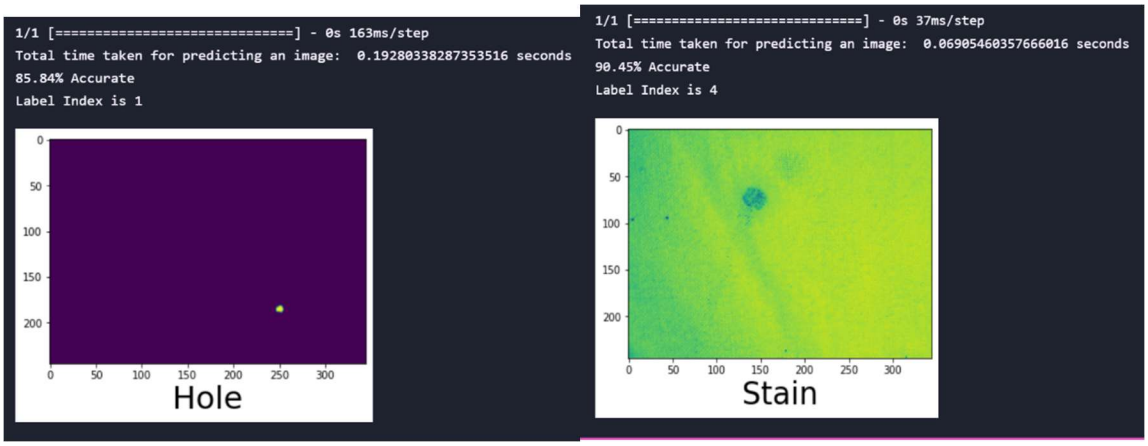
```
Resampled dataset shape Counter({0: 406, 4: 387, 1: 267, 3: 152, 2: 124, 5: 101})
```

```
9/9 [==============================] - 4s 418ms/step
```

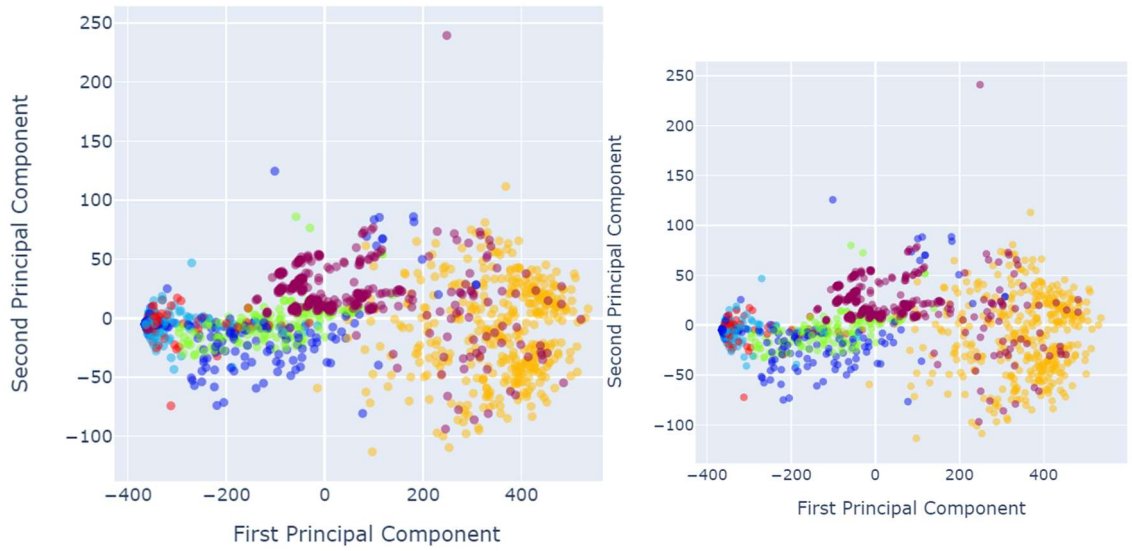
	precision	recall	f1-score	support
0	0.98	0.57	0.72	86
1	0.72	0.93	0.81	57
2	0.79	0.95	0.86	20
3	0.93	0.83	0.88	30
4	0.78	0.97	0.86	72
5	0.96	0.96	0.96	23
accuracy			0.83	288
macro avg	0.86	0.87	0.85	288
weighted avg	0.86	0.83	0.82	288



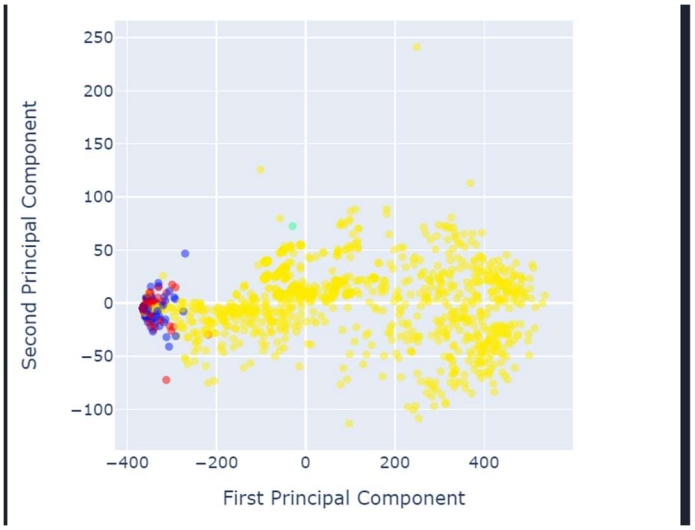




PCA Before and After Sampling



PCA After CNN



Appendix 2-10.1: Random UnderSampler 200x800

```
# Reshape the given X train for the Random Under sampler technique
X_res = X.reshape(X.shape[0], -1)
y_res = y.ravel()
print(f'Original dataset shape {Counter(y_res)}')
```

✓ 0.8s

```
riginal dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
random_under_sample = RandomUnderSampler(random_state=42)
X_rus, y_rus = random_under_sample.fit_resample(X_res, y_res)
```

✓ 0.2s

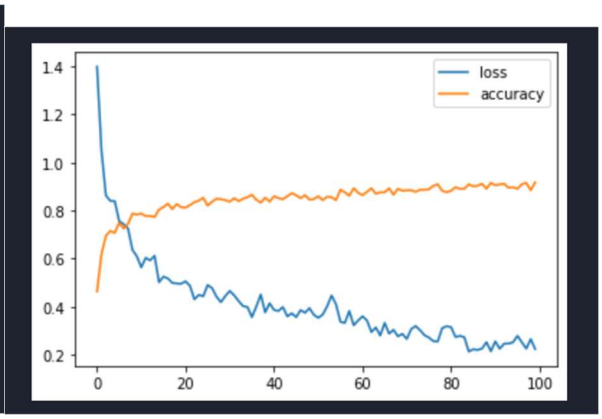
```
print(f'Resampled dataset shape {Counter(y_rus)}')
```

✓ 0.3s

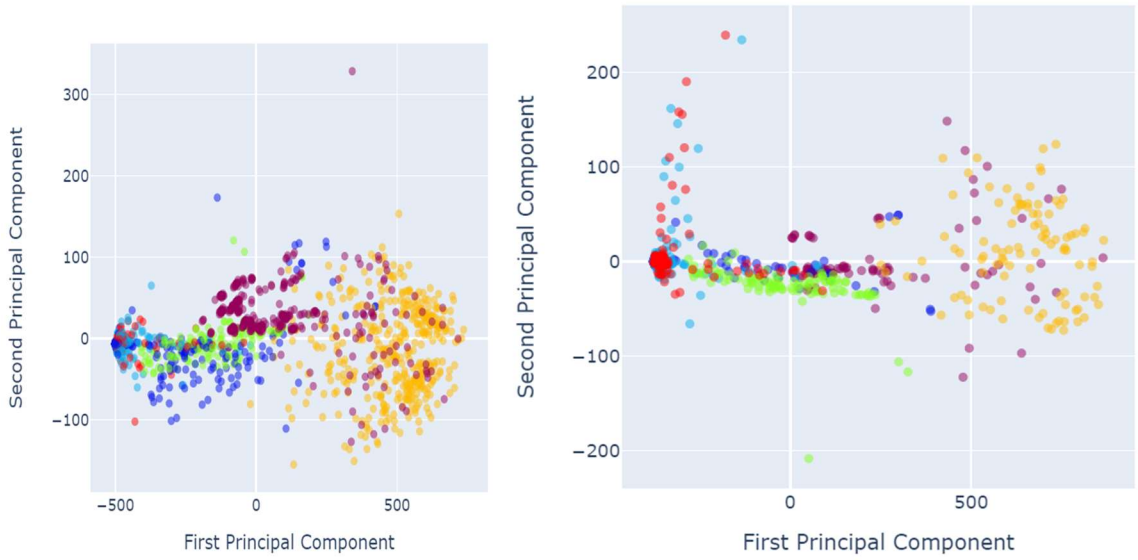
```
esampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

```
4/4 [==============================] - 3s 754ms/step
```

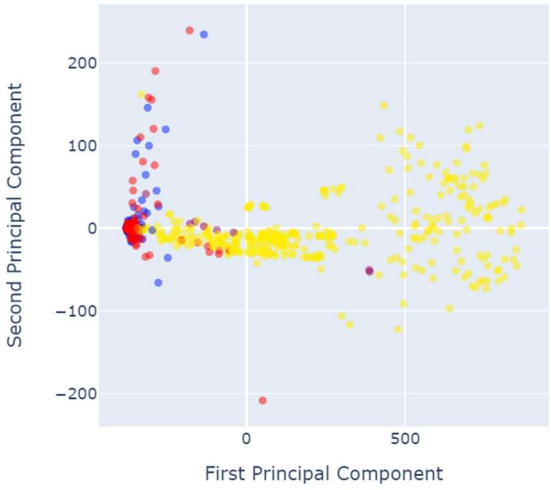
	precision	recall	f1-score	support
0	0.78	0.91	0.84	23
1	0.92	0.55	0.69	22
2	0.62	0.96	0.75	25
3	0.85	0.92	0.88	12
4	0.83	0.62	0.71	16
5	0.89	0.67	0.76	24
accuracy			0.77	122
macro avg	0.81	0.77	0.77	122
weighted avg	0.81	0.77	0.77	122



PCA Before and After Sampling



PCA after CNN



Appendix 2-10.2: Random UnderSampler Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
random_under_sample = RandomUnderSampler(random_state=42)
X_rus, y_rus = random_under_sample.fit_resample(X_res, y_res)
```

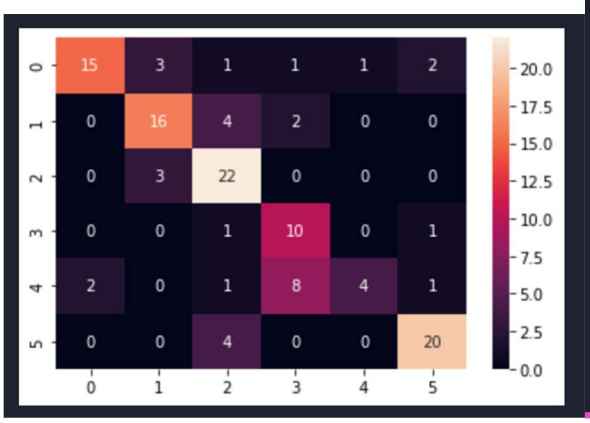
✓ 0.1s

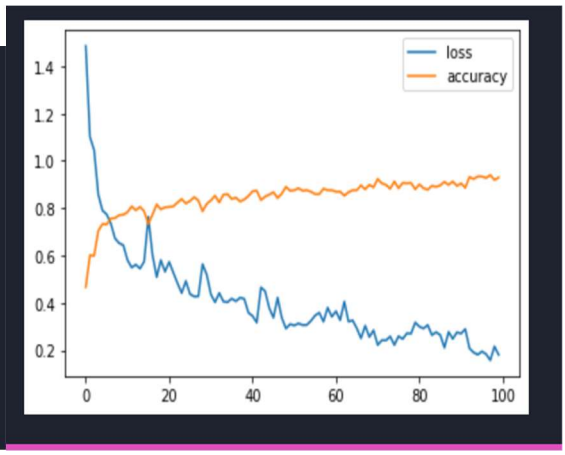
```
print(f'Resampled dataset shape {Counter(y_rus)}')
```

✓ 0.3s

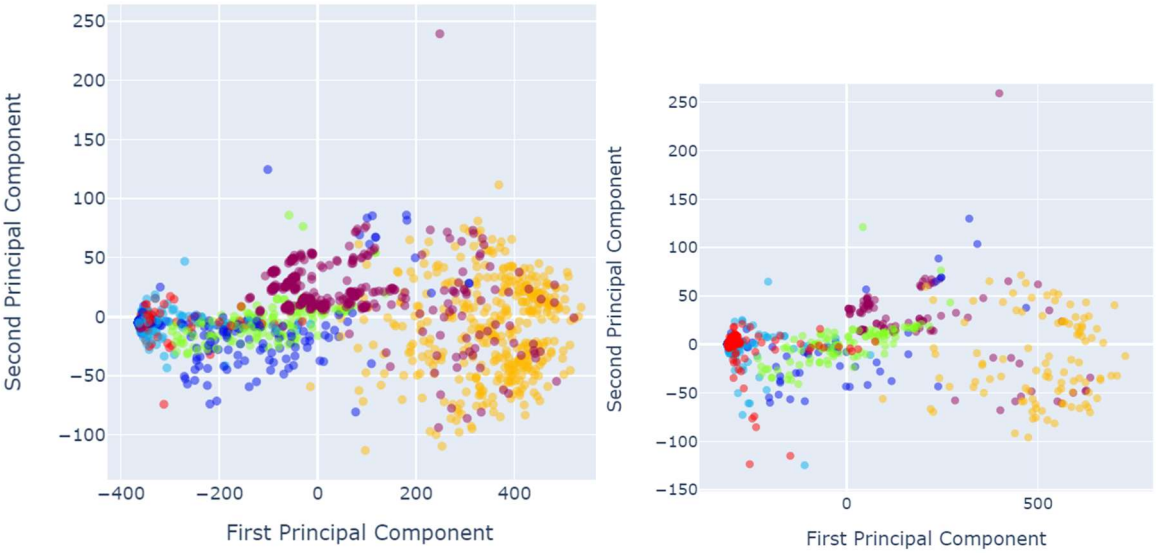
```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

```
4/4 [==============================] - 2s 476ms/step
              precision    recall  f1-score   support

           0       0.88      0.65      0.75        23
           1       0.73      0.73      0.73        22
           2       0.67      0.88      0.76        25
           3       0.48      0.83      0.61        12
           4       0.80      0.25      0.38        16
           5       0.83      0.83      0.83        24

    accuracy                           0.71       122
   macro avg       0.73      0.70      0.68       122
weighted avg       0.75      0.71      0.70       122
```

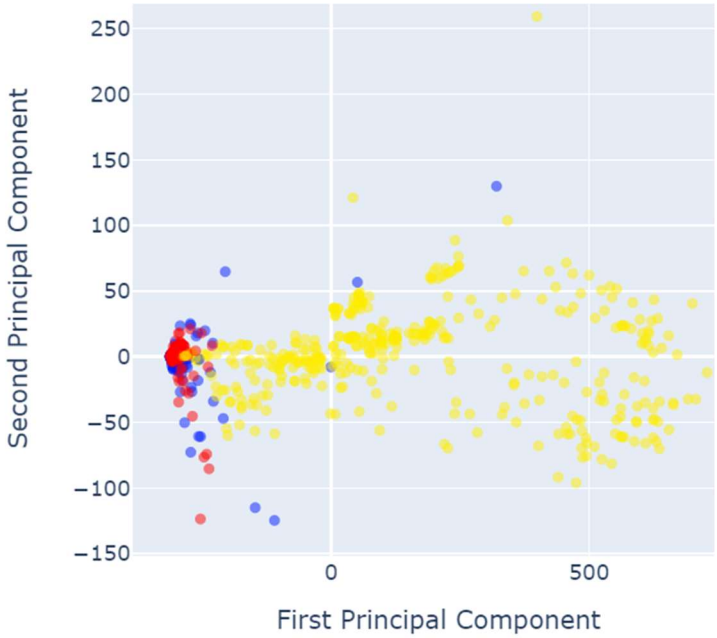




PCA Before and After Sampling



PCA After CNN



Appendix 2-10.3: Random UnderSampling Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
random_under_sample = RandomUnderSampler(random_state=42)
X_rus, y_rus = random_under_sample.fit_resample(X_res, y_res)
```

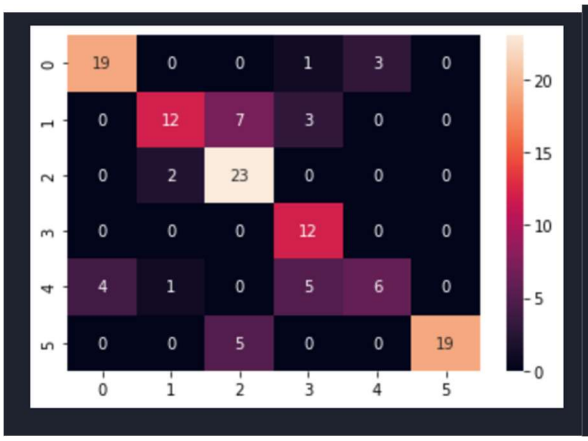
✓ 0.1s

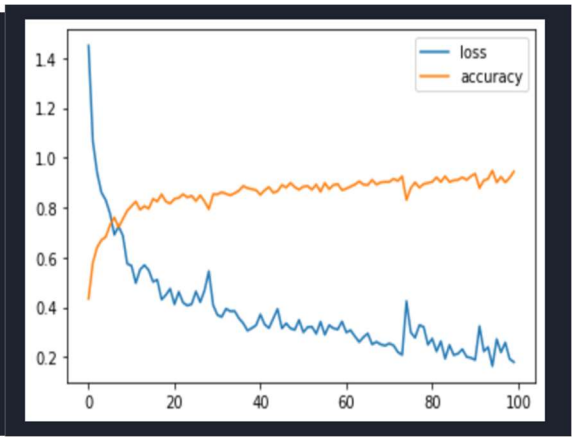
```
print(f'Resampled dataset shape {Counter(y_rus)}')
```

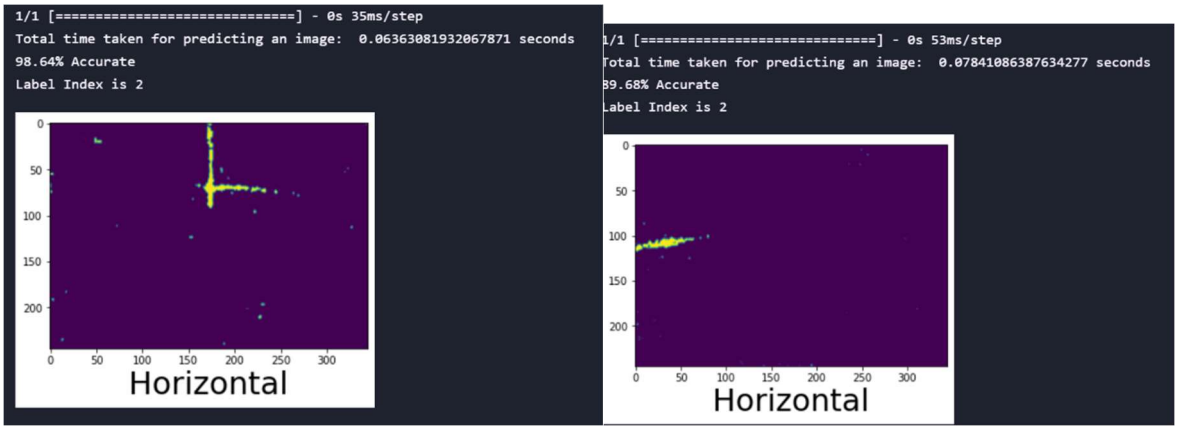
✓ 0.1s

```
Resampled dataset shape Counter({0: 101, 1: 101, 2: 101, 3: 101, 4: 101, 5: 101})
```

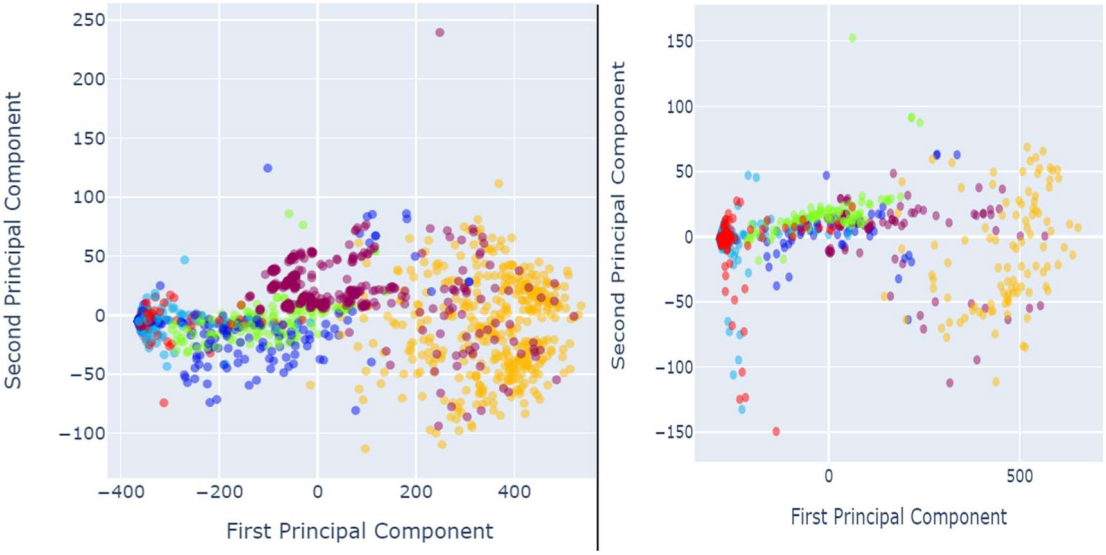
```
4/4 [==============================] - 2s 389ms/step
              precision    recall  f1-score   support

           0       0.83      0.83      0.83        23
           1       0.80      0.55      0.65        22
           2       0.66      0.92      0.77        25
           3       0.57      1.00      0.73        12
           4       0.67      0.38      0.48        16
           5       1.00      0.79      0.88        24

    accuracy                           0.75       122
   macro avg       0.75      0.74      0.72       122
weighted avg       0.78      0.75      0.74       122
```



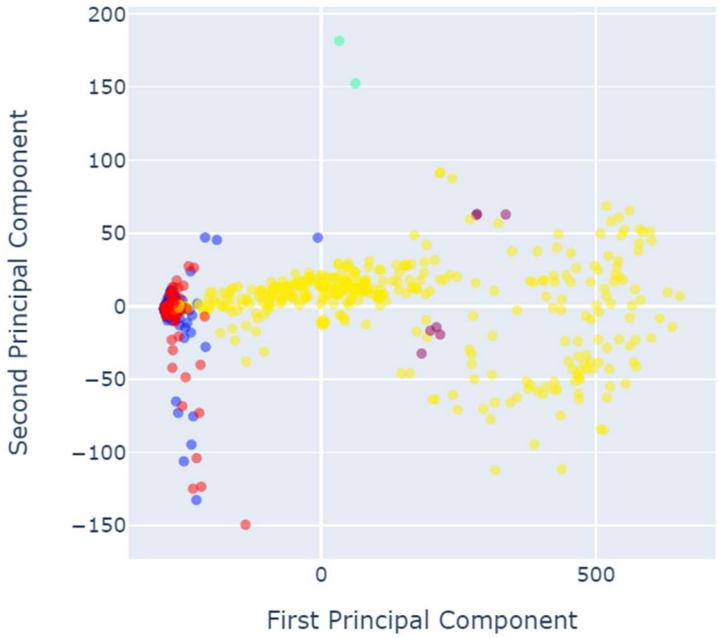




PCA Before and After Sampling



PCA After CNN



Appendix 3: Oversampling

Appendix 3-1.1: ADASYN Oversampling 200x800

```
# Reshape the given X train for the ADASYN Oversampling
X_res = X.reshape(X.shape[0], -1)
y_res = y.ravel()
print(f'Original dataset shape {Counter(y_res)}')
```

✓ 0.3s

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = ADASYN(sampling_strategy='minority')
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

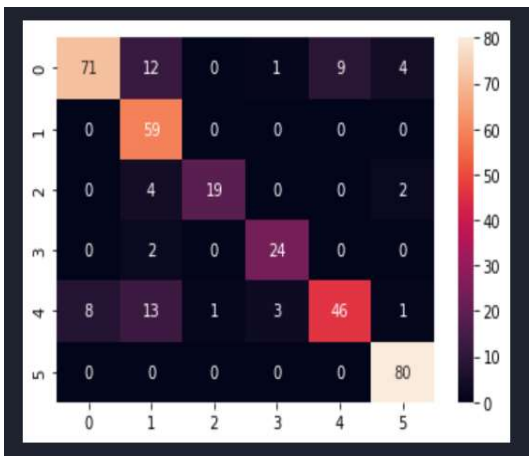
✓ 22.8s

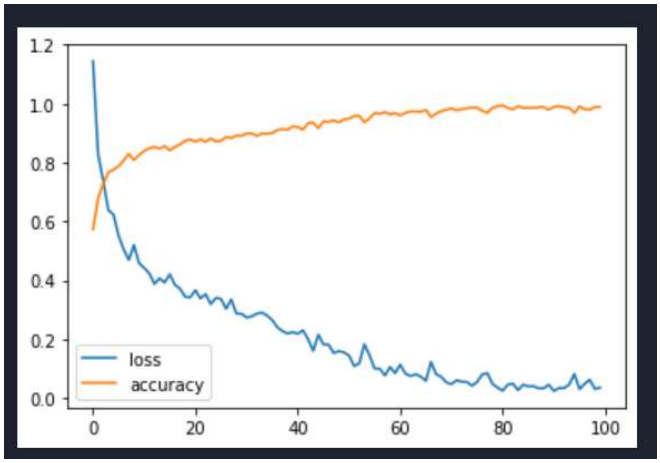
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

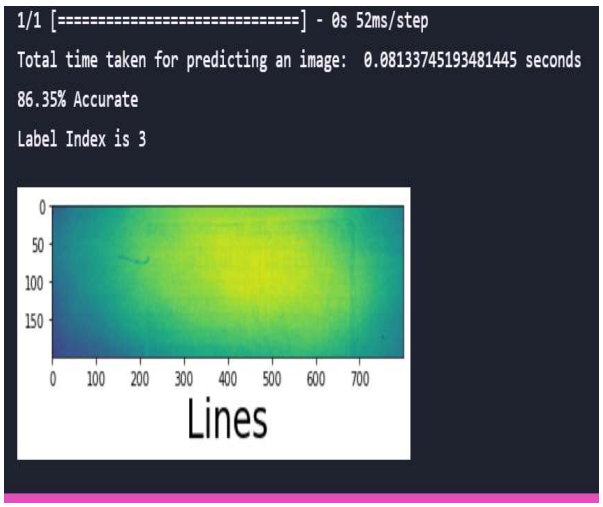
✓ 0.1s

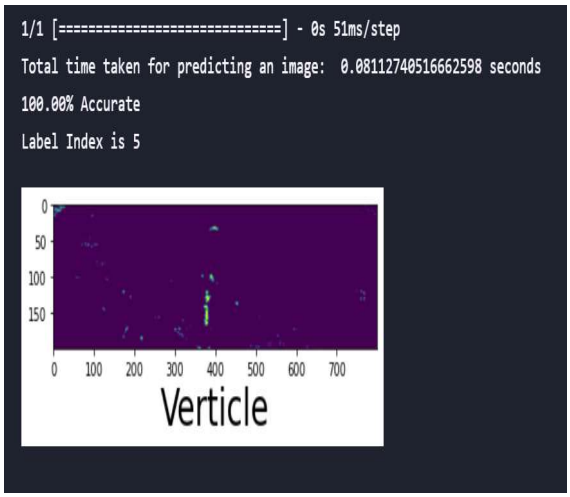
```
Resampled dataset shape Counter({0: 417, 5: 406, 4: 398, 1: 281, 3: 157, 2: 136})
```

```
12/12 [==============================] - 13s 1s/step
              precision    recall  f1-score   support

           0       0.90      0.73      0.81        97
           1       0.66      1.00      0.79        59
           2       0.95      0.76      0.84        25
           3       0.86      0.92      0.89        26
           4       0.84      0.64      0.72        72
           5       0.92      1.00      0.96        80

    accuracy                           0.83       359
   macro avg       0.85      0.84      0.84       359
weighted avg       0.85      0.83      0.83       359
```

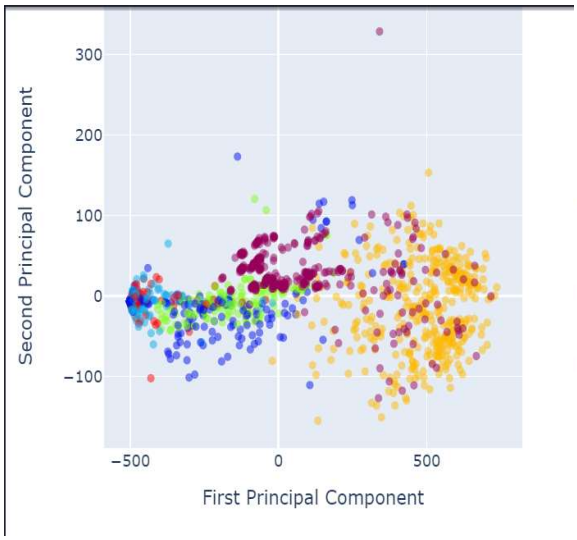


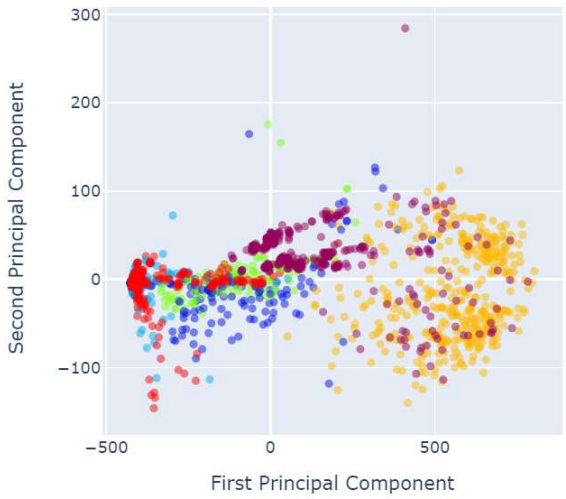




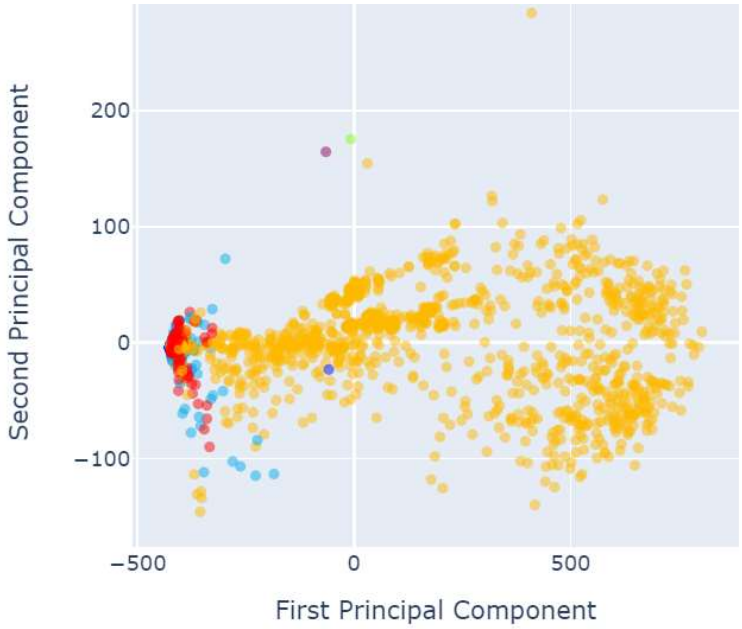


PCA before and after Sampling





PCA After CNN



Appendix 3-1.2: ADASYN Oversampling Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = ADASYN(sampling_strategy='minority')
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 3.5s

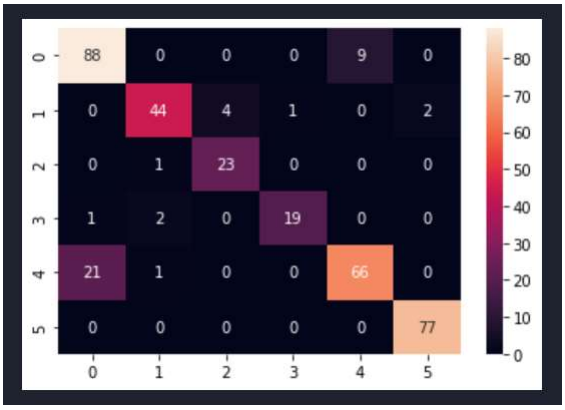
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

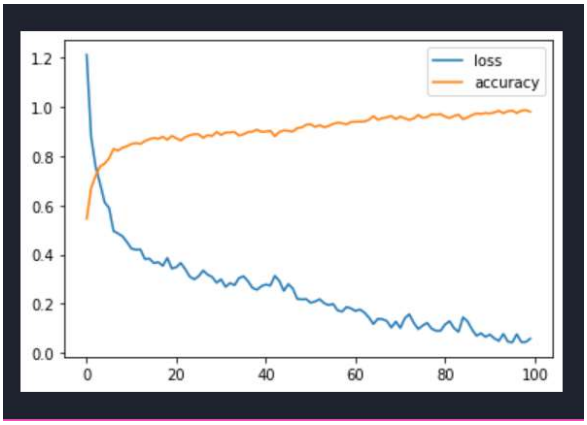
✓ 0.1s

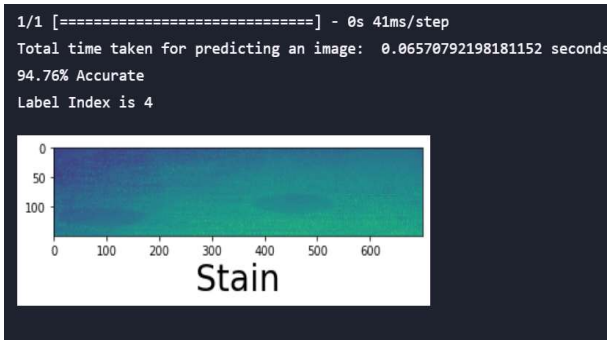
```
Resampled dataset shape Counter({0: 417, 5: 405, 4: 398, 1: 281, 3: 157, 2: 136})
```

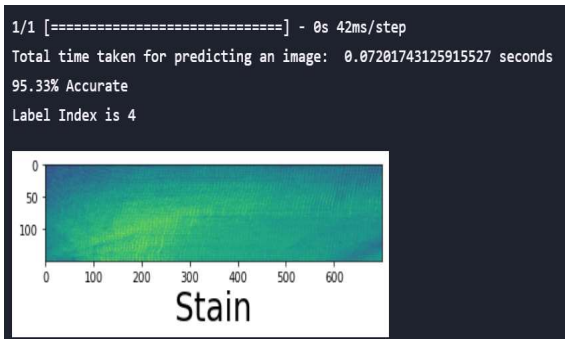
```
12/12 [==============================] - 11s 959ms/step
```

	precision	recall	f1-score	support
0	0.80	0.91	0.85	97
1	0.92	0.86	0.89	51
2	0.85	0.96	0.90	24
3	0.95	0.86	0.90	22
4	0.88	0.75	0.81	88
5	0.97	1.00	0.99	77
accuracy			0.88	359
macro avg	0.90	0.89	0.89	359
weighted avg	0.89	0.88	0.88	359

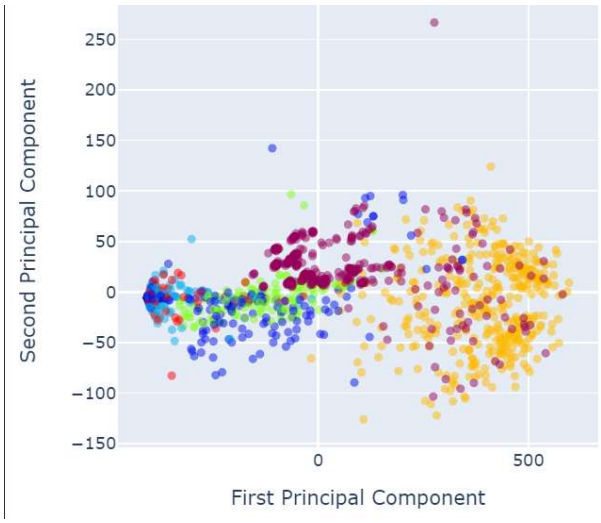


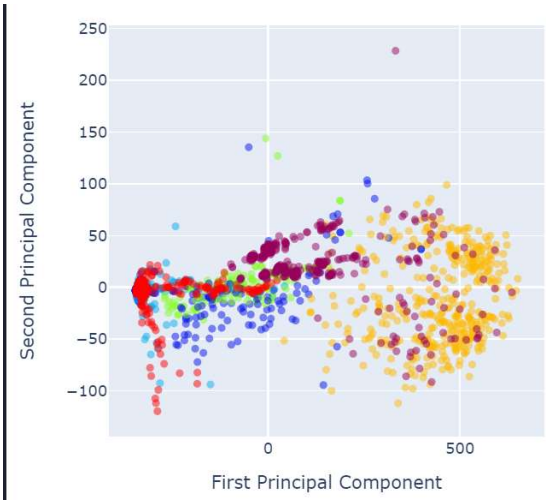




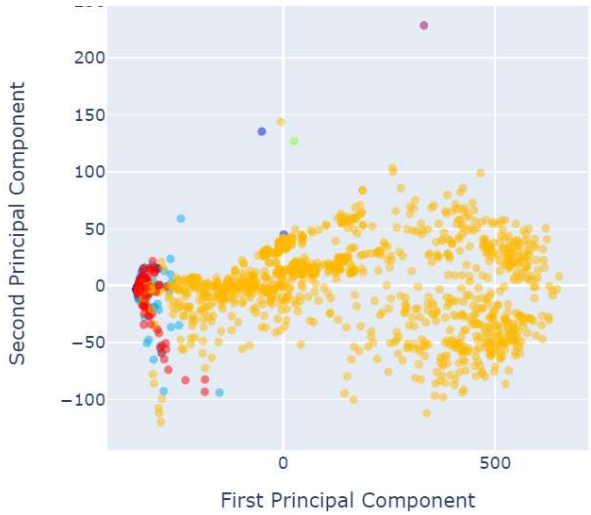


PCA before and After Sampling





PCA after CNN



Appendix 3-1.3: ADASYN Oversampling Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = ADASYN(sampling_strategy='minority')
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

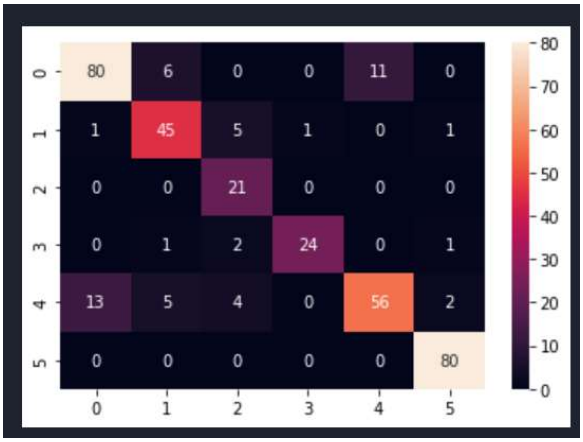
✓ 18.5s

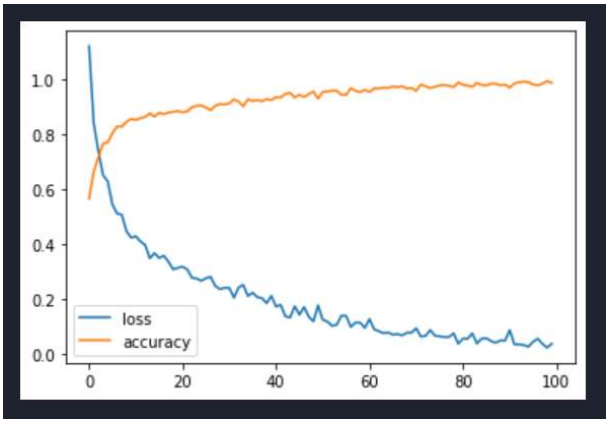
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

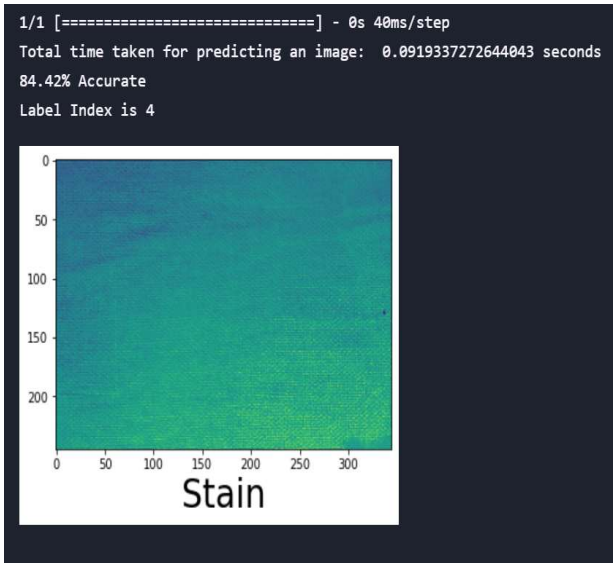
✓ 0.9s

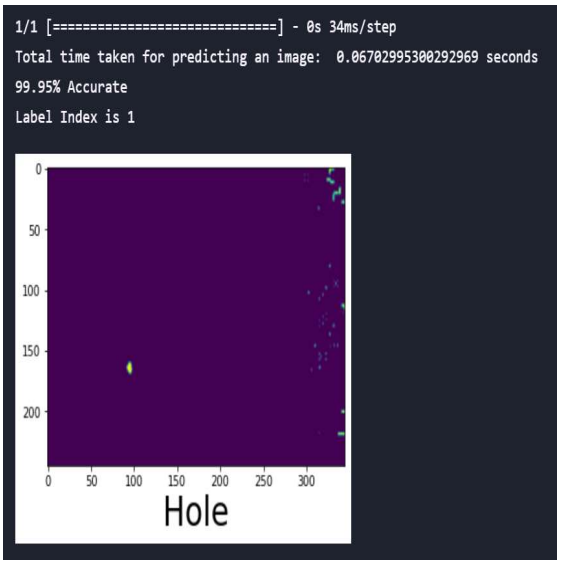
```
Resampled dataset shape Counter({0: 417, 5: 406, 4: 398, 1: 281, 3: 157, 2: 136})
```

```
12/12 [==============================] - 5s 396ms/step
              precision    recall  f1-score   support

           0       0.85      0.82      0.84        97
           1       0.79      0.85      0.82        53
           2       0.66      1.00      0.79        21
           3       0.96      0.86      0.91        28
           4       0.84      0.70      0.76        80
           5       0.95      1.00      0.98        80

    accuracy                           0.85       359
   macro avg       0.84      0.87      0.85       359
weighted avg       0.86      0.85      0.85       359
```

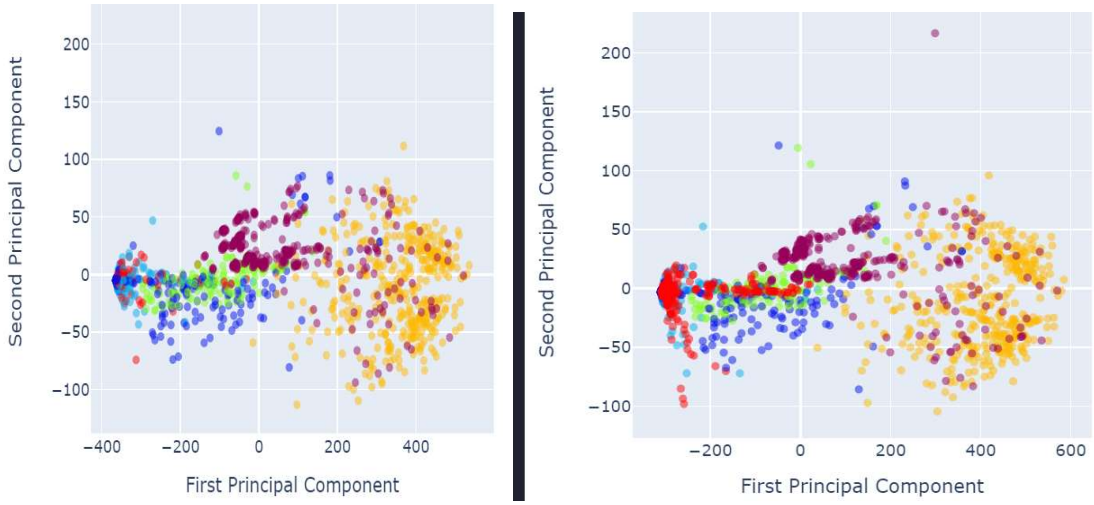




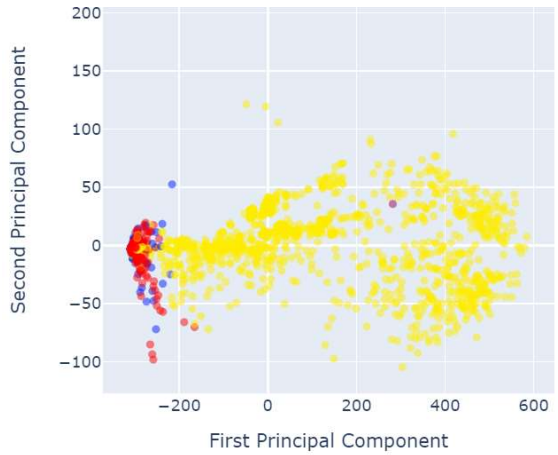




PCA Before and After Sampling



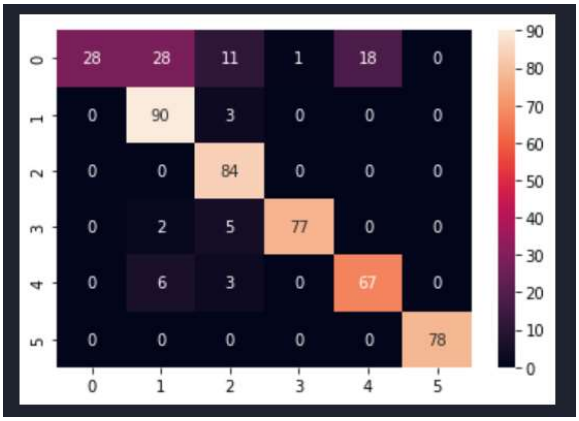
PCA After CNN

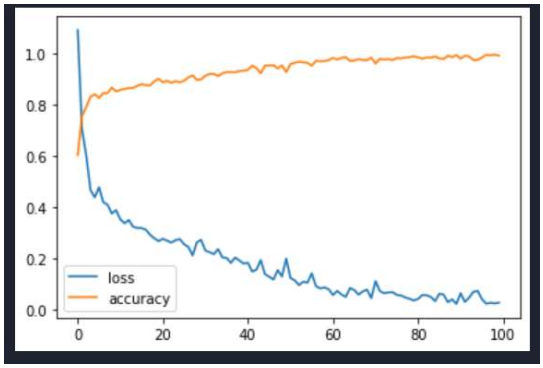


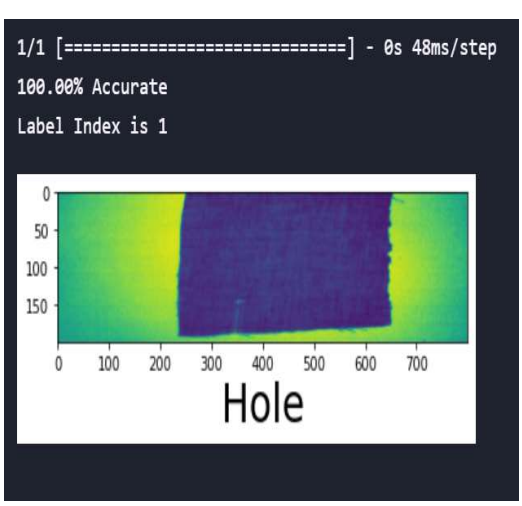
Appendix 3-2.1: Random Oversampling Size 200x800

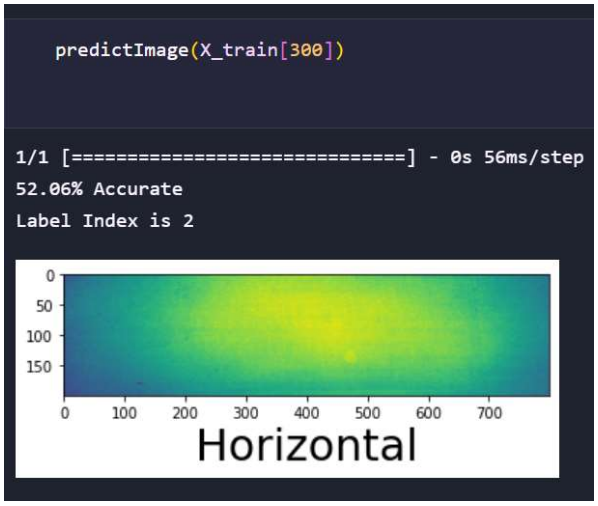
```
    # Reshape the given X train for the Random Under sampler technique
    X_res = X.reshape(X.shape[0], -1)
    y_res = y.ravel()
    print(f'Original dataset shape {Counter(y_res)}')
23] ✓ 0.9s

··  Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})

    ost = RandomOverSampler(random_state=42)
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
24] ✓ 8.4s

    print(f'Resampled dataset shape {Counter(y_ost)}')
25] ✓ 0.7s

··  Resampled dataset shape Counter({2: 417, 1: 417, 3: 417, 4: 417, 5: 417, 0: 417})
```

```
16/16 [==============================] - 12s 745ms/step
              precision    recall  f1-score   support

           0       1.00      0.33      0.49        86
           1       0.71      0.97      0.82        93
           2       0.79      1.00      0.88        84
           3       0.99      0.92      0.95        84
           4       0.79      0.88      0.83        76
           5       1.00      1.00      1.00        78

    accuracy                           0.85       501
   macro avg       0.88      0.85      0.83       501
weighted avg       0.88      0.85      0.83       501
```

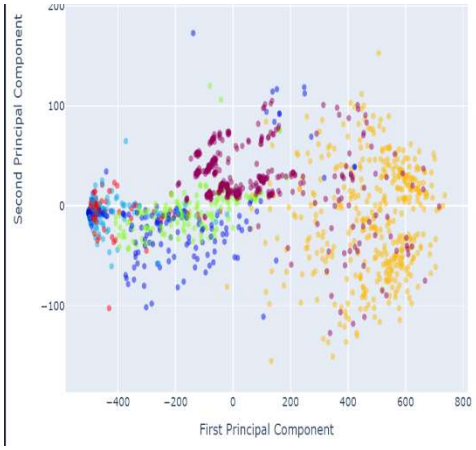


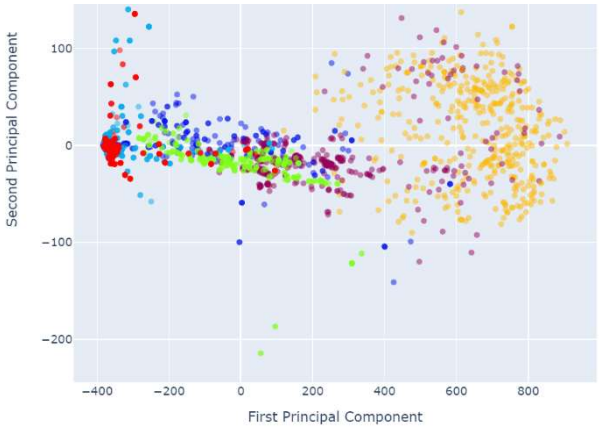




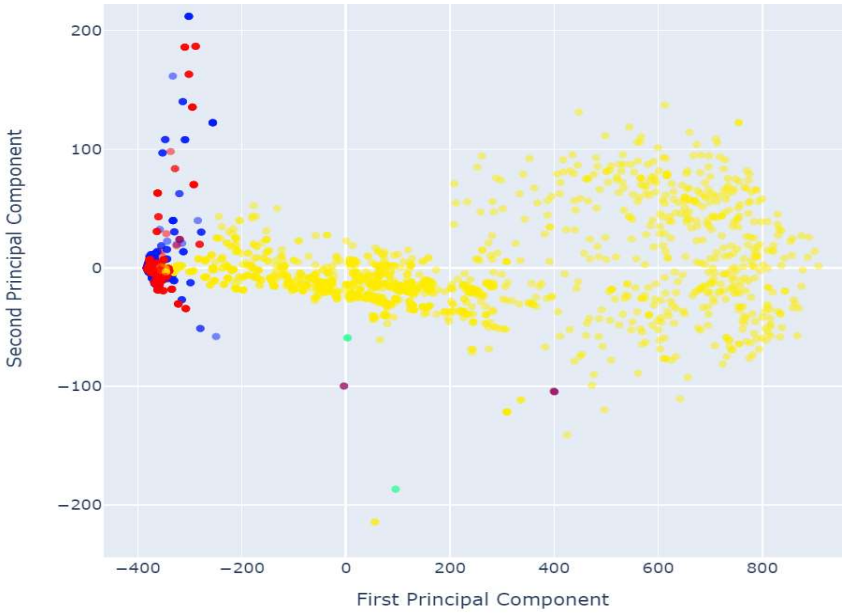


PCA Before and after Sampling





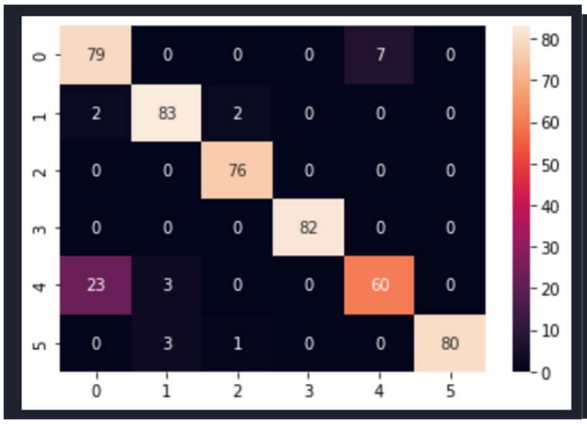
PCA After CNN

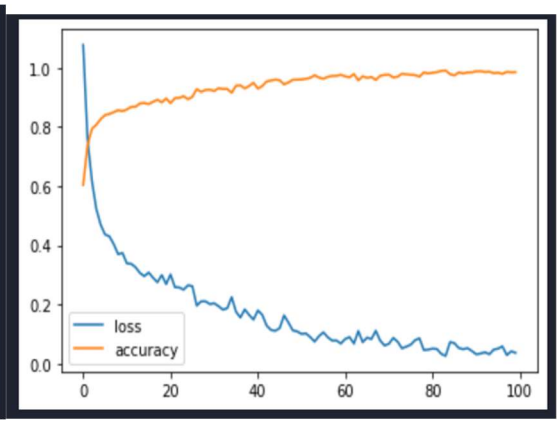


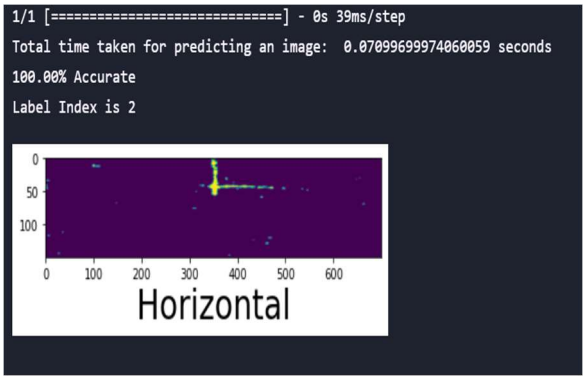
Appendix 3-2.2: Random Oversampling Size 150x700

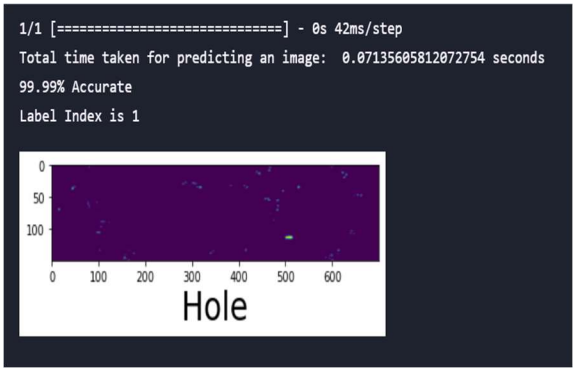
```
    # Reshape the given X train for the Random Under sampler technique
    X_res = X.reshape(X.shape[0], -1)
    y_res = y.ravel()
    print(f'Original dataset shape {Counter(y_res)}')
23]  ✓ 0.9s

··  Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})

    ost = RandomOverSampler(random_state=42)
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
24]  ✓ 8.4s

    print(f'Resampled dataset shape {Counter(y_ost)}')
25]  ✓ 0.7s

··  Resampled dataset shape Counter({2: 417, 1: 417, 3: 417, 4: 417, 5: 417, 0: 417})
```

```
16/16 [==============================] - 8s 496ms/step
              precision    recall  f1-score   support

           0       0.76      0.92      0.83        86
           1       0.93      0.95      0.94        87
           2       0.96      1.00      0.98        76
           3       1.00      1.00      1.00        82
           4       0.90      0.70      0.78        86
           5       1.00      0.95      0.98        84

    accuracy                           0.92       501
   macro avg       0.92      0.92      0.92       501
weighted avg       0.92      0.92      0.92       501
```

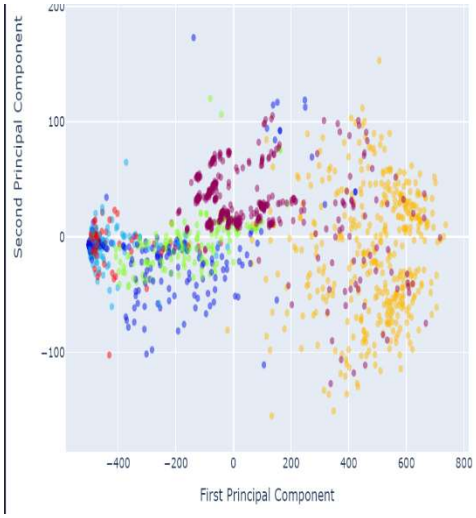


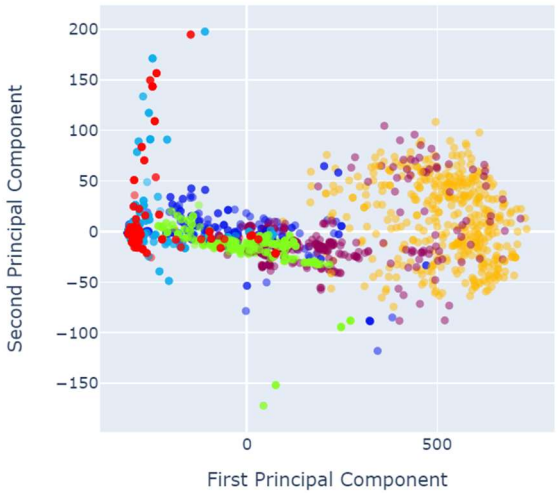




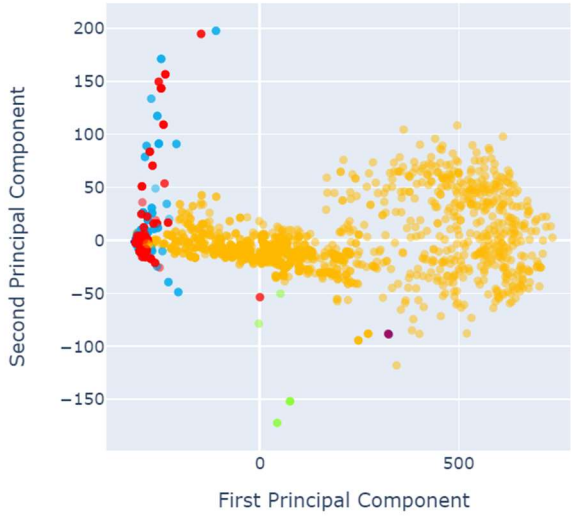


PCA Before and after Sampling





PCA After CNN



Appendix 3-2.3: Random Oversampling Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = RandomOverSampler(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

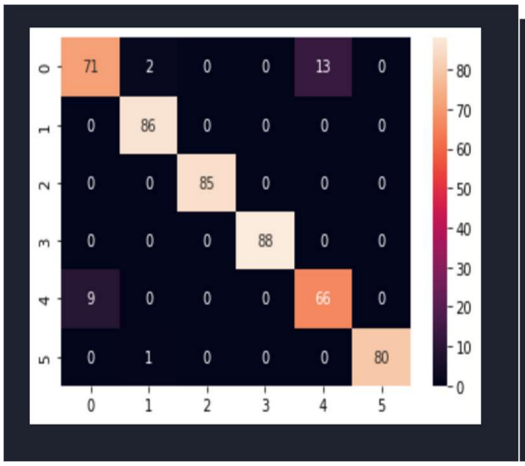
✓ 0.7s

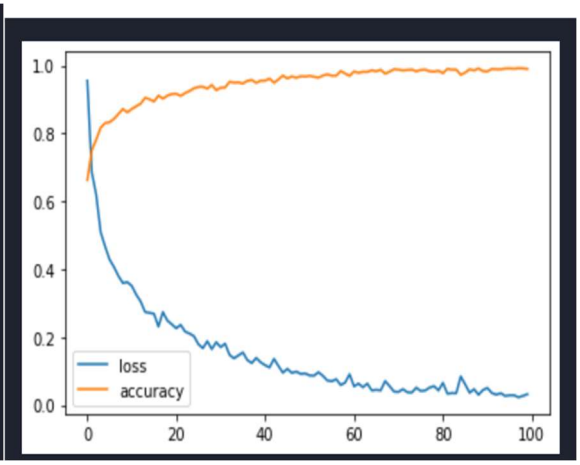
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

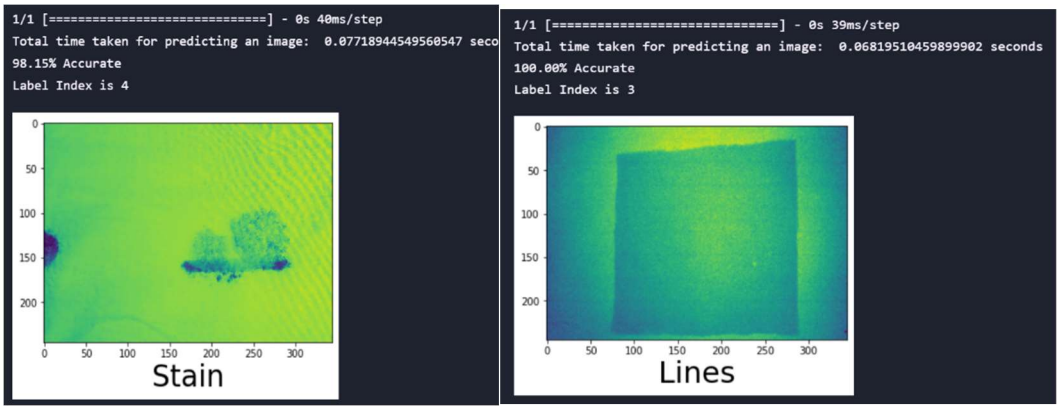
✓ 0.9s

```
Resampled dataset shape Counter({4: 417, 1: 417, 3: 417, 5: 417, 2: 417, 0: 417})
```

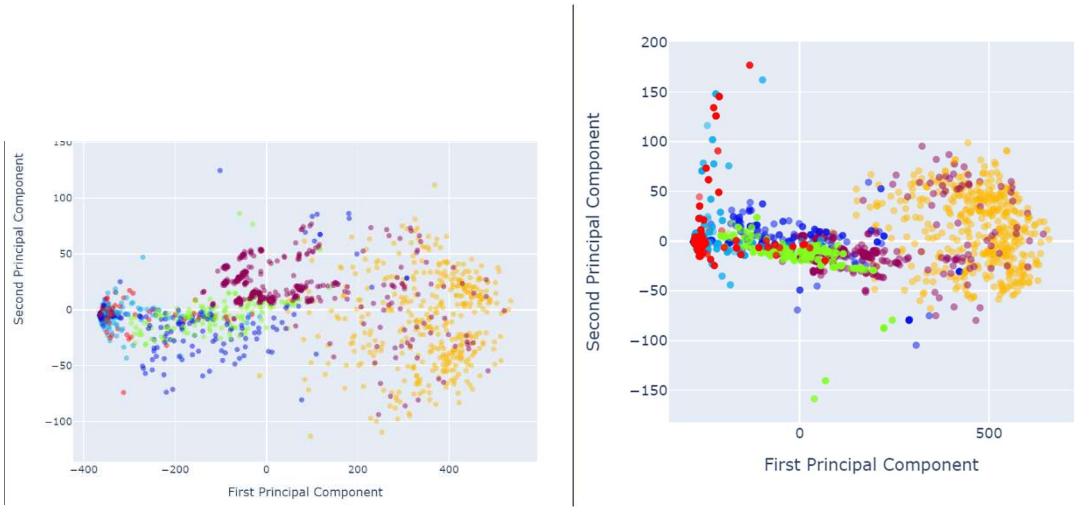
```
16/16 [==============================] - 8s 474ms/step
              precision    recall  f1-score   support

           0       0.89      0.83      0.86        86
           1       0.97      1.00      0.98        86
           2       1.00      1.00      1.00        85
           3       1.00      1.00      1.00        88
           4       0.84      0.88      0.86        75
           5       1.00      0.99      0.99        81

    accuracy                           0.95       501
   macro avg       0.95      0.95      0.95       501
weighted avg       0.95      0.95      0.95       501
```



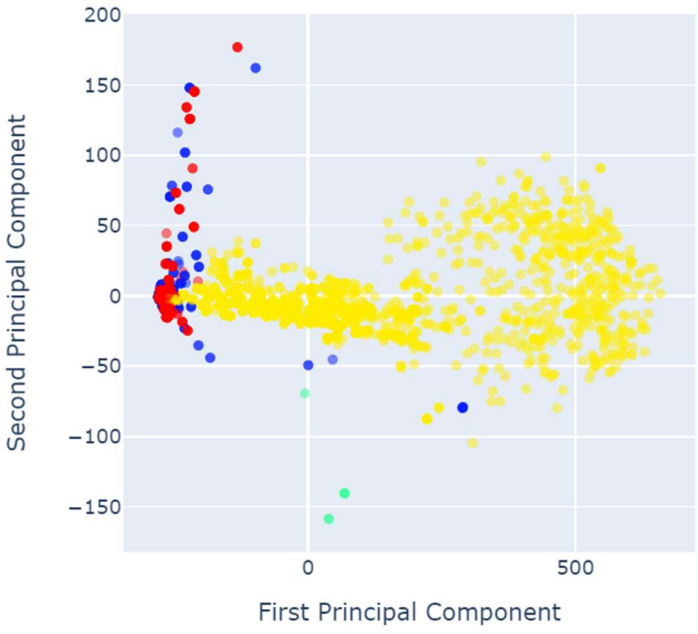




PCA before and after sampling



PCA After CNN



Appendix 3-3.1: Borderline SMOTE Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = BorderlineSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

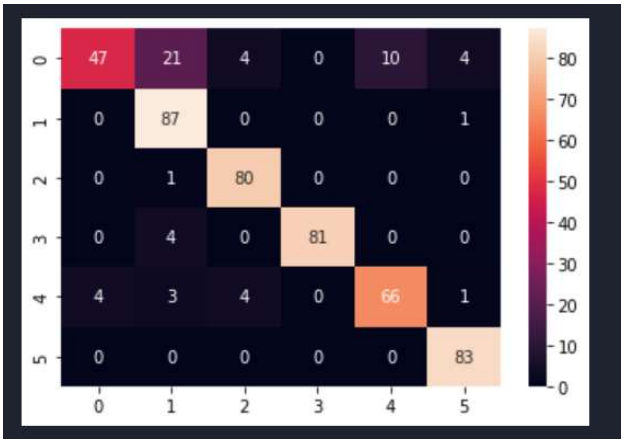
✓ 31.3s

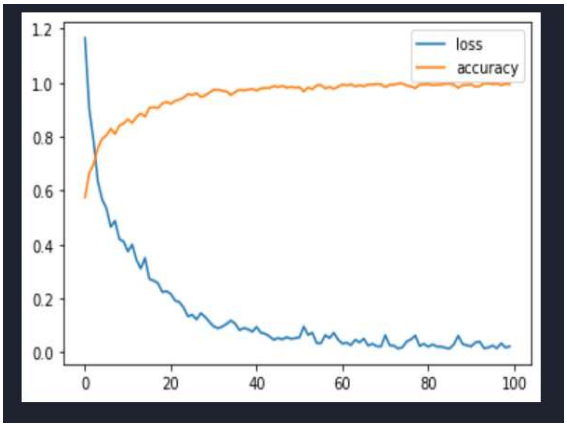
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

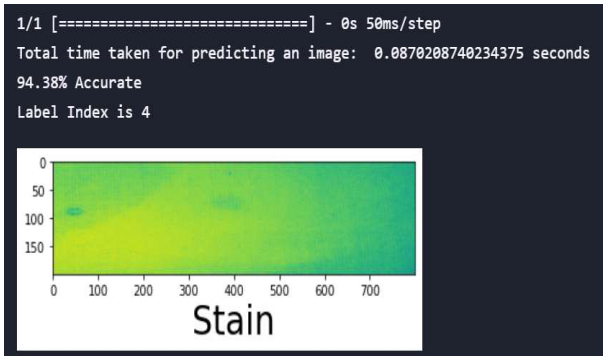
✓ 0.1s

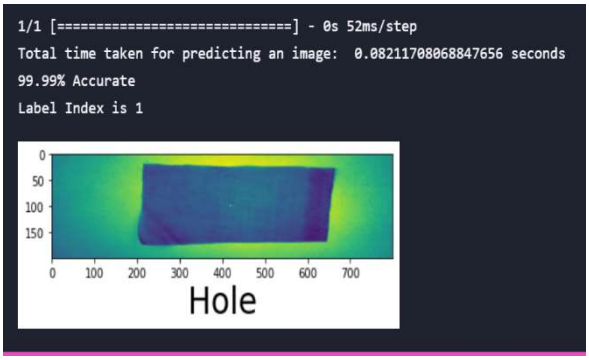
```
Resampled dataset shape Counter({1: 417, 4: 417, 5: 417, 3: 417, 2: 417, 0: 417})
```

```
16/16 [==============================] - 13s 766ms/step
              precision    recall  f1-score   support

           0       0.92      0.55      0.69        86
           1       0.75      0.99      0.85        88
           2       0.91      0.99      0.95        81
           3       1.00      0.95      0.98        85
           4       0.87      0.85      0.86        78
           5       0.93      1.00      0.97        83

    accuracy                           0.89       501
   macro avg       0.90      0.89      0.88       501
weighted avg       0.90      0.89      0.88       501
```

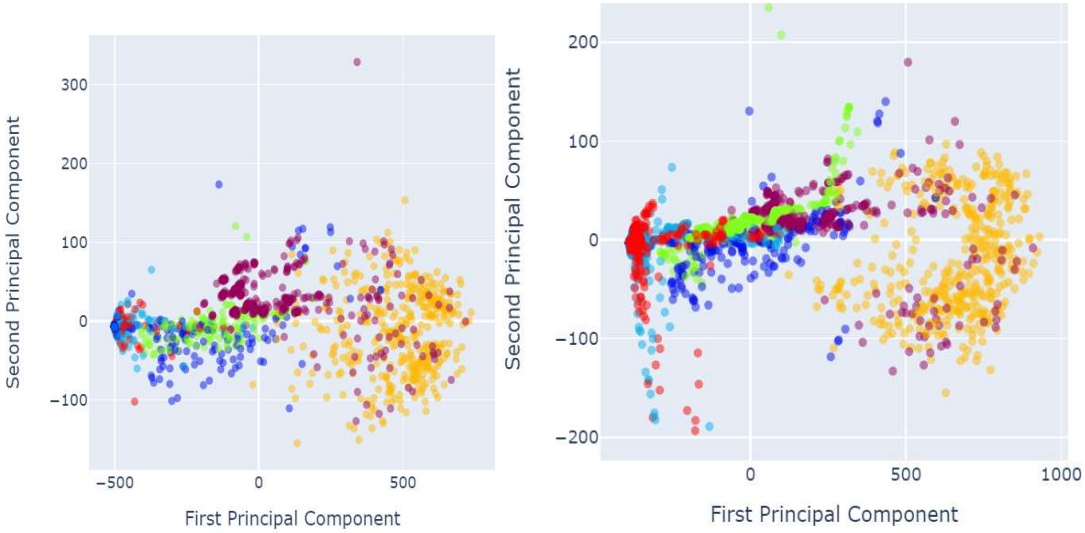




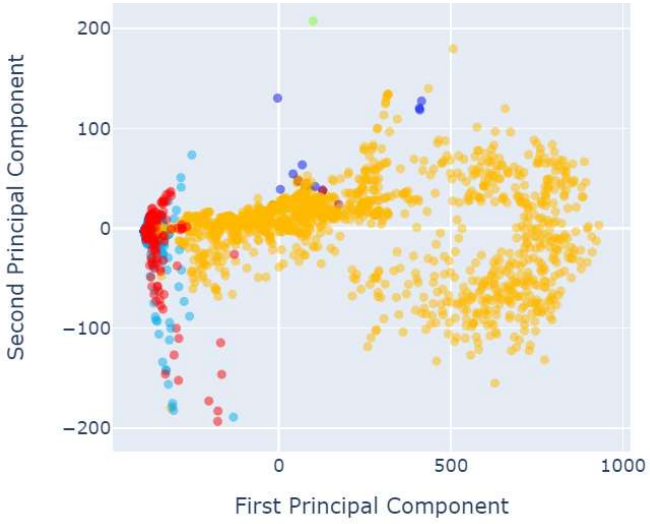




PCA Before and after sampling



PCA after CNN



Appendix 3-3.2: Borderline SMOTE Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

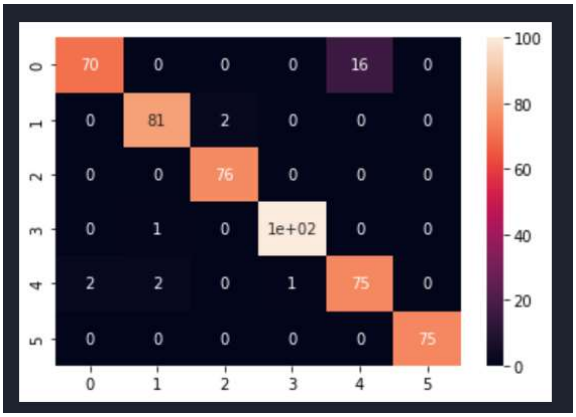
```
ost = BorderlineSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

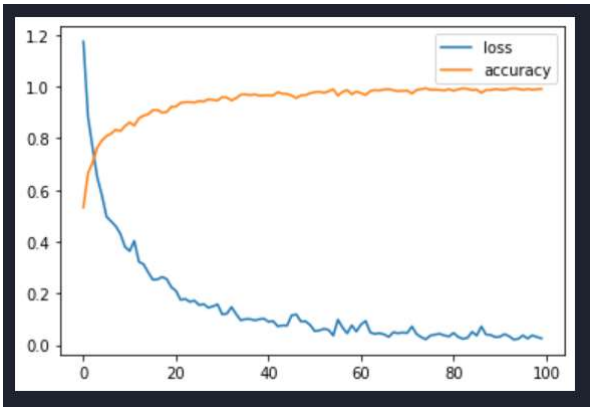
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

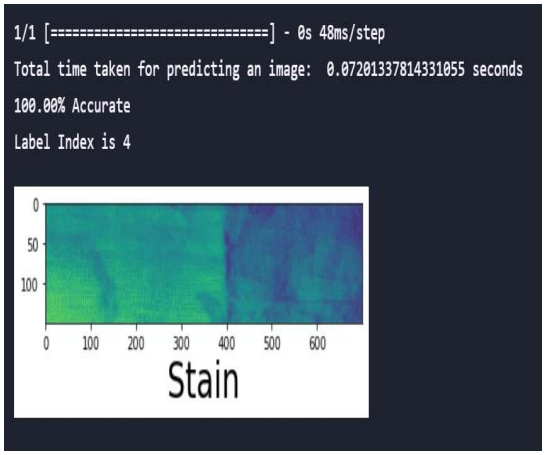
```
Resampled dataset shape Counter({2: 417, 1: 417, 4: 417, 3: 417, 5: 417, 0: 417})
```

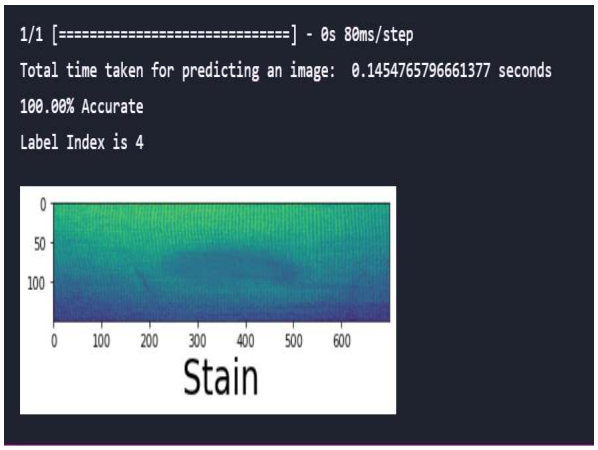
```
··  16/16 [==============================] - 10s 571ms/step
```

	precision	recall	f1-score	support
0	0.97	0.81	0.89	86
1	0.96	0.98	0.97	83
2	0.97	1.00	0.99	76
3	0.99	0.99	0.99	101
4	0.82	0.94	0.88	80
5	1.00	1.00	1.00	75
accuracy			0.95	501
macro avg	0.95	0.95	0.95	501
weighted avg	0.96	0.95	0.95	501

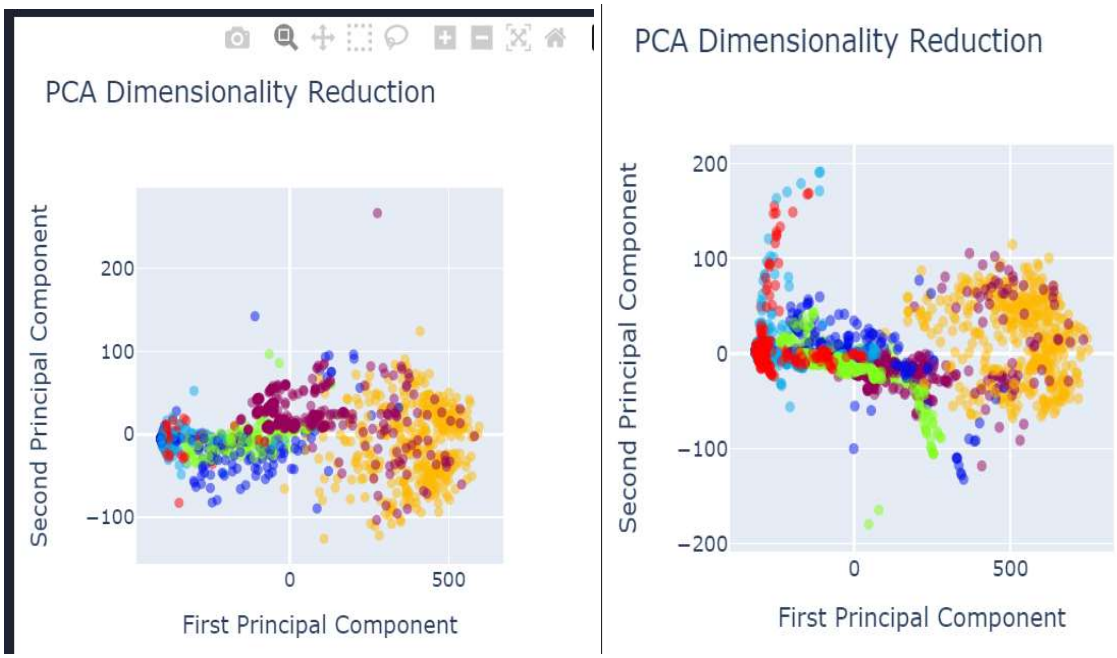




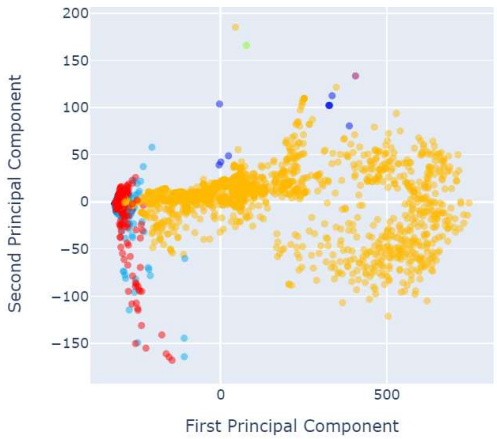




PCA before and after sampling



PCA after CNN



Appendix 3-3.3: Borderline SMOTE Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = BorderlineSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 20.6s

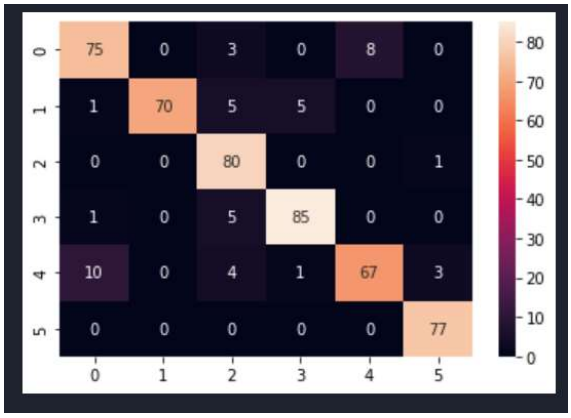
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

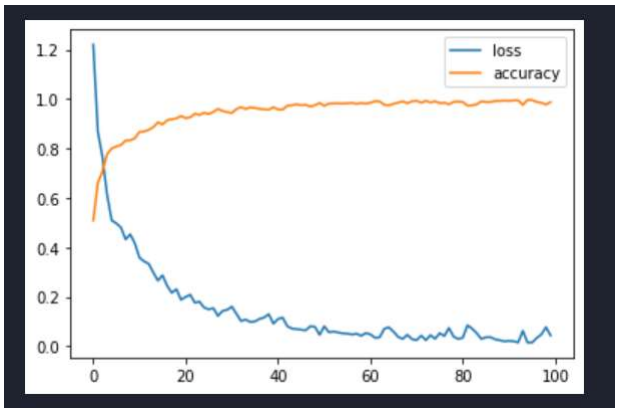
✓ 0.7s

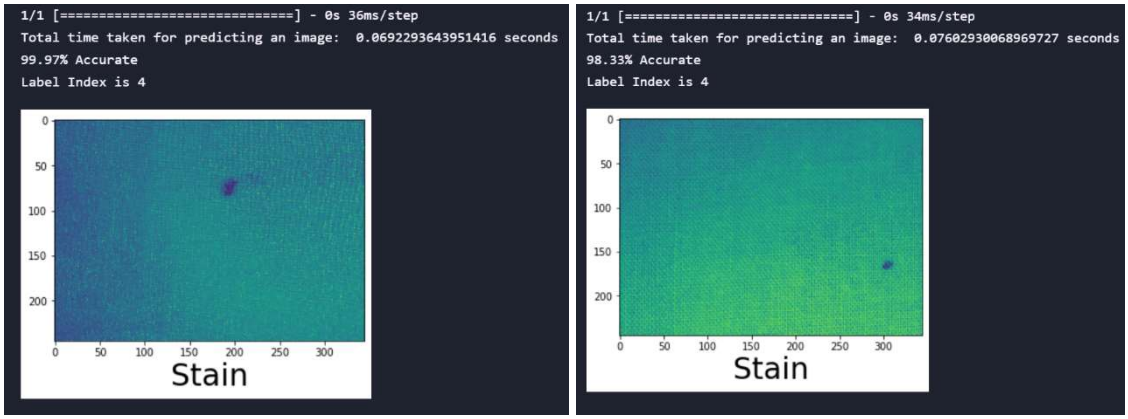
```
Resampled dataset shape Counter({1: 417, 2: 417, 5: 417, 4: 417, 3: 417, 0: 417})
```

```
16/16 [==============================] - 7s 420ms/step
```

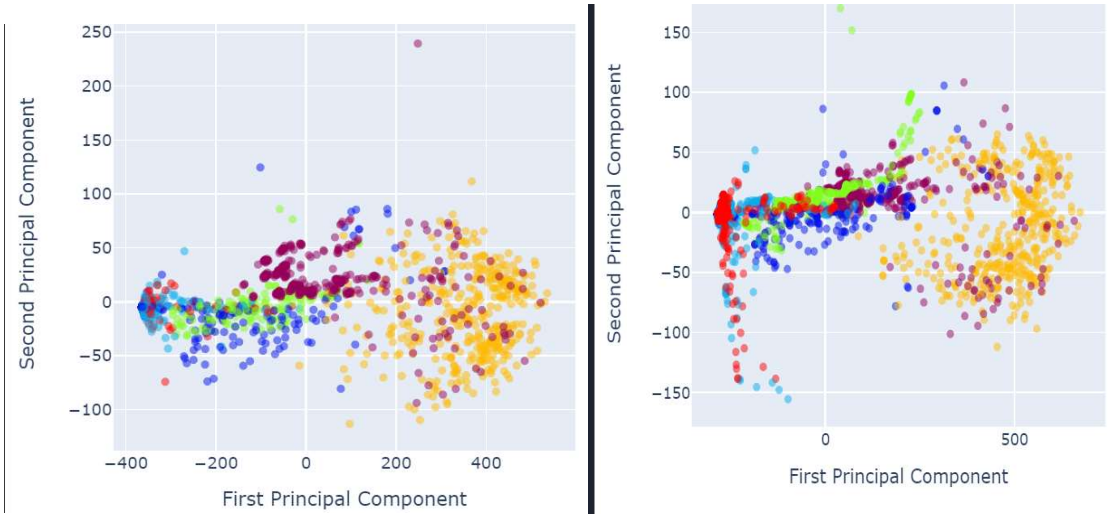
	precision	recall	f1-score	support
0	0.86	0.87	0.87	86
1	1.00	0.86	0.93	81
2	0.82	0.99	0.90	81
3	0.93	0.93	0.93	91
4	0.89	0.79	0.84	85
5	0.95	1.00	0.97	77
accuracy			0.91	501
macro avg	0.91	0.91	0.91	501
weighted avg	0.91	0.91	0.91	501



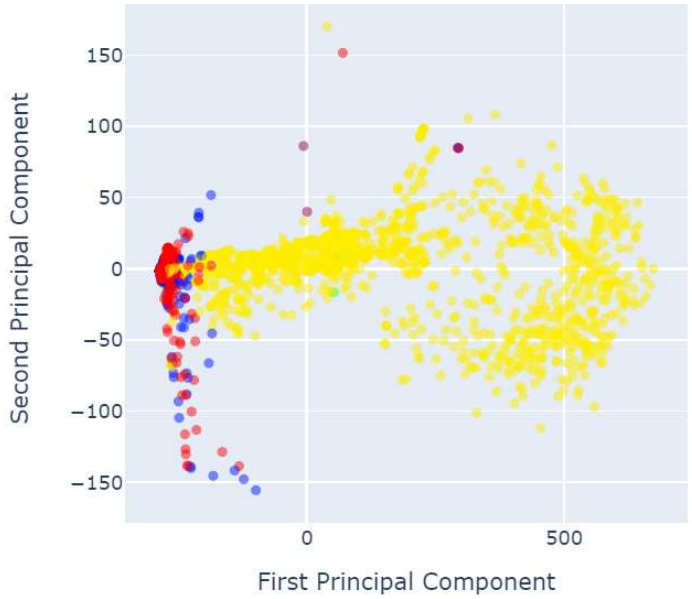




PCA before and after sampling



PCA after CNN



Appendix 3-4.1: K Means SMOTE Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = KMeansSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

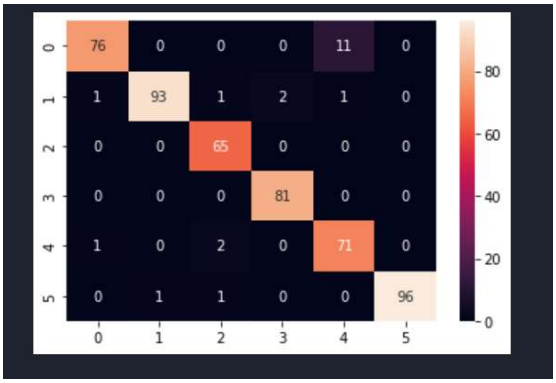
✓ 4m 55.2s

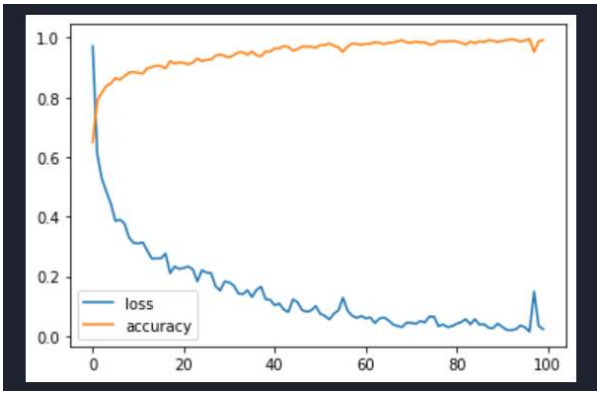
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

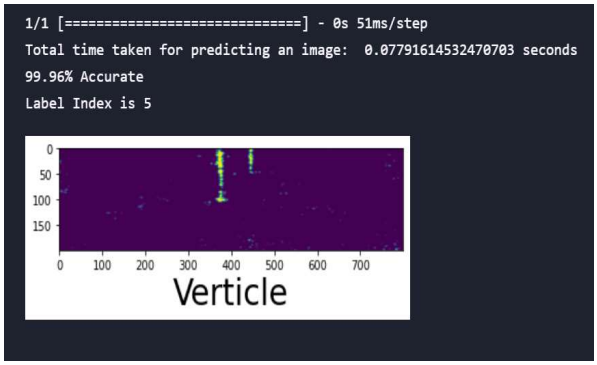
✓ 0.2s

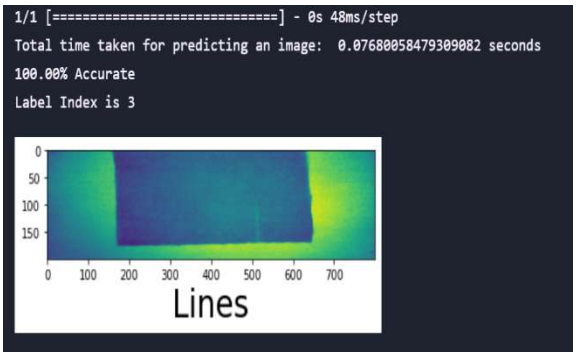
```
Resampled dataset shape Counter({4: 421, 1: 420, 3: 418, 2: 417, 5: 417, 0: 417})
```

0	0.97	0.87	0.92	87
1	0.99	0.95	0.97	98
2	0.94	1.00	0.97	65
3	0.98	1.00	0.99	81
4	0.86	0.96	0.90	74
5	1.00	0.98	0.99	98
accuracy			0.96	503
macro avg	0.96	0.96	0.96	503
weighted avg	0.96	0.96	0.96	503

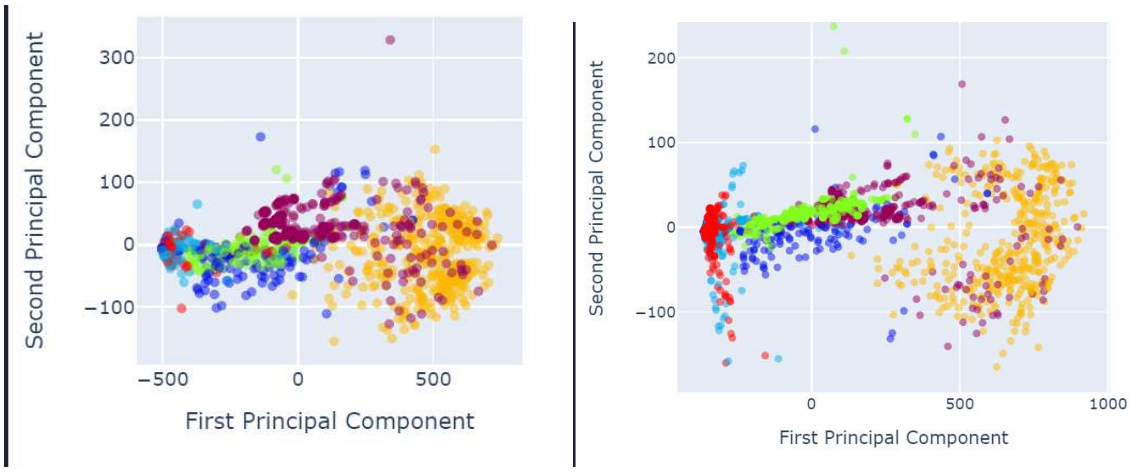




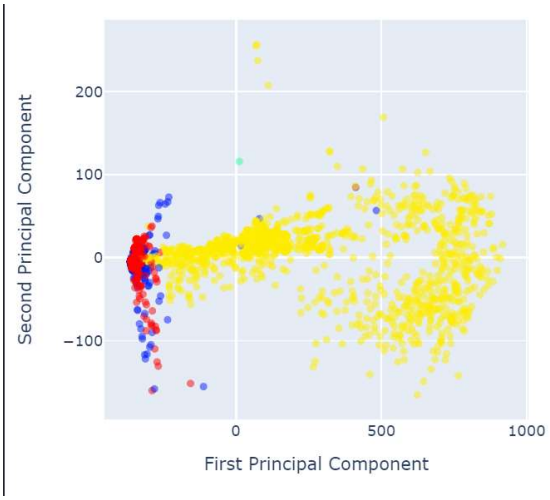




PCA Before and After Sampling



PCA After Sampling



Appendix 3-4.2: K Means SMOTE Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = KMeansSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

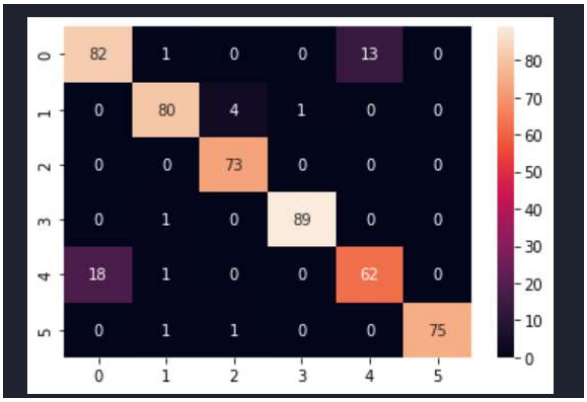
✓ 1m 45.4s

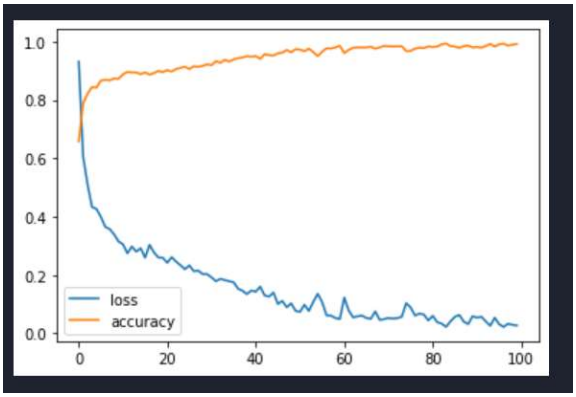
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

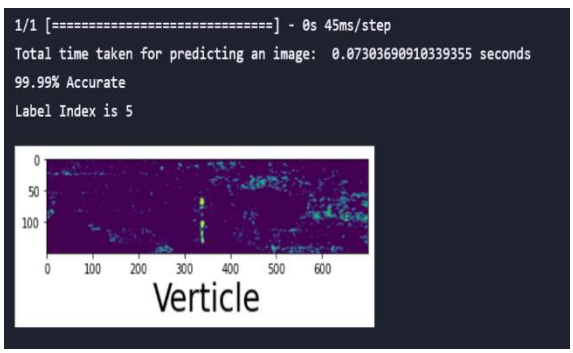
✓ 0.1s

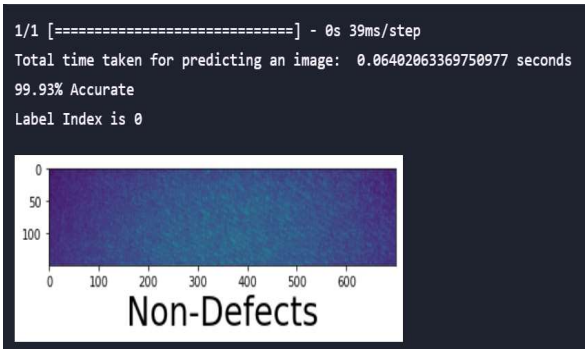
```
Resampled dataset shape Counter({1: 421, 4: 420, 3: 419, 5: 417, 2: 417, 0: 417})
```

```
16/16 [==============================] - 8s 501ms/step
              precision    recall  f1-score   support

           0       0.82      0.85      0.84        96
           1       0.95      0.94      0.95        85
           2       0.94      1.00      0.97        73
           3       0.99      0.99      0.99        90
           4       0.83      0.77      0.79        81
           5       1.00      0.97      0.99        77

    accuracy                           0.92       502
   macro avg       0.92      0.92      0.92       502
weighted avg       0.92      0.92      0.92       502
```

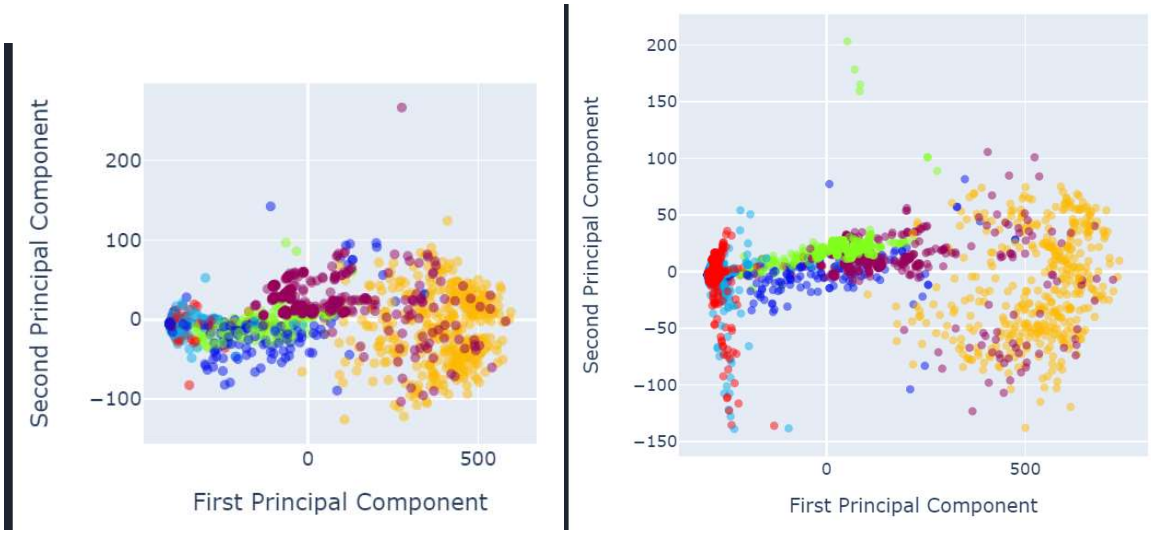




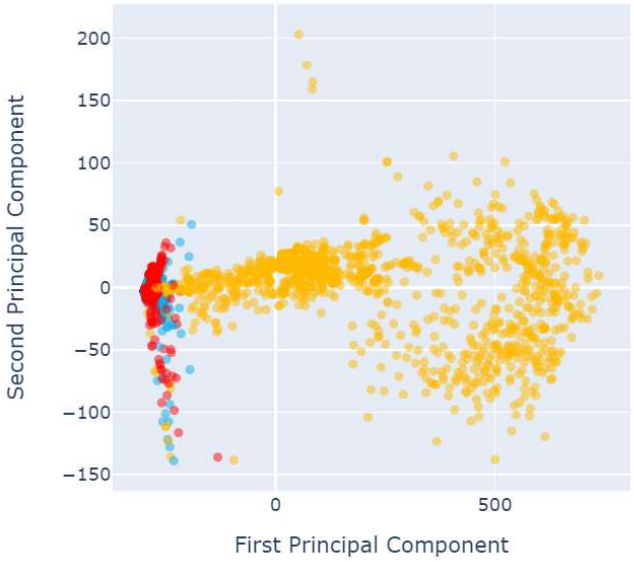




PCA Before and After Sampling



PCA after CNN



Appendix 3-4.3: K Means SMOTE Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = KMeansSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

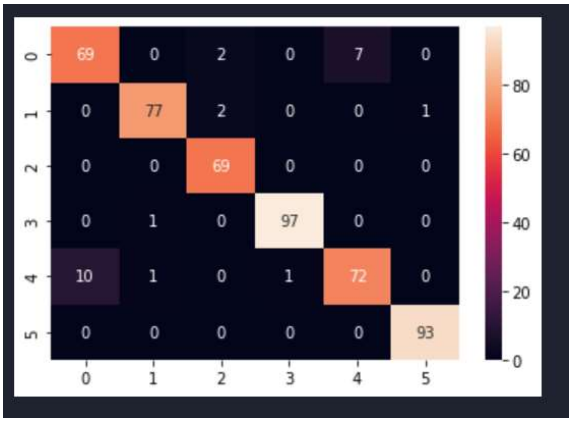
✓ 1m 7.4s

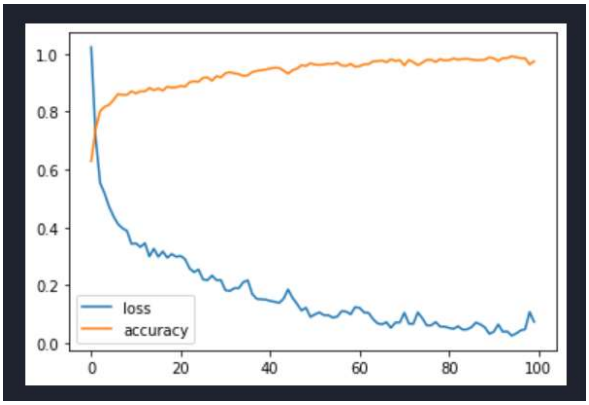
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

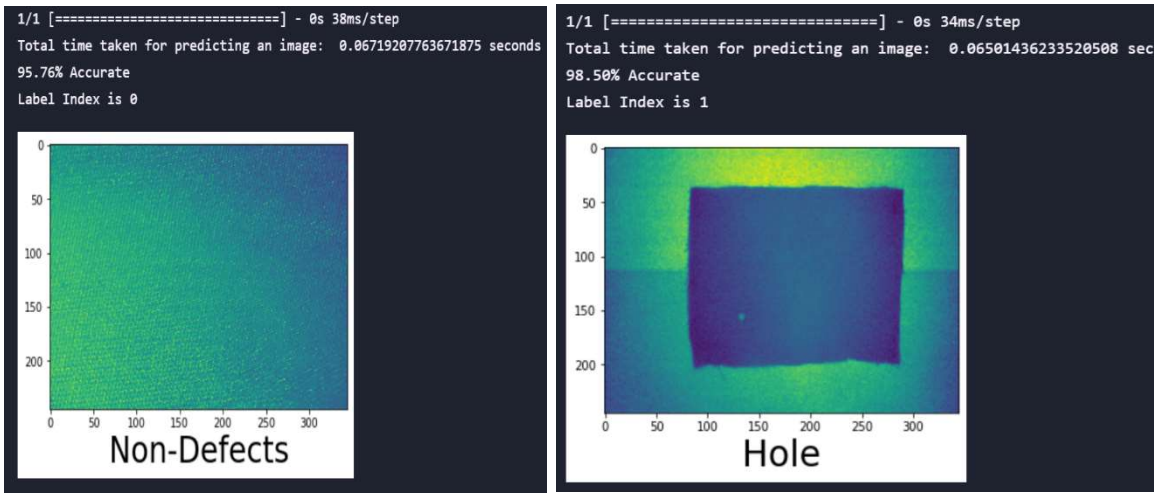
✓ 0.1s

```
Resampled dataset shape Counter({4: 421, 1: 420, 3: 418, 2: 417, 5: 417, 0: 417})
```

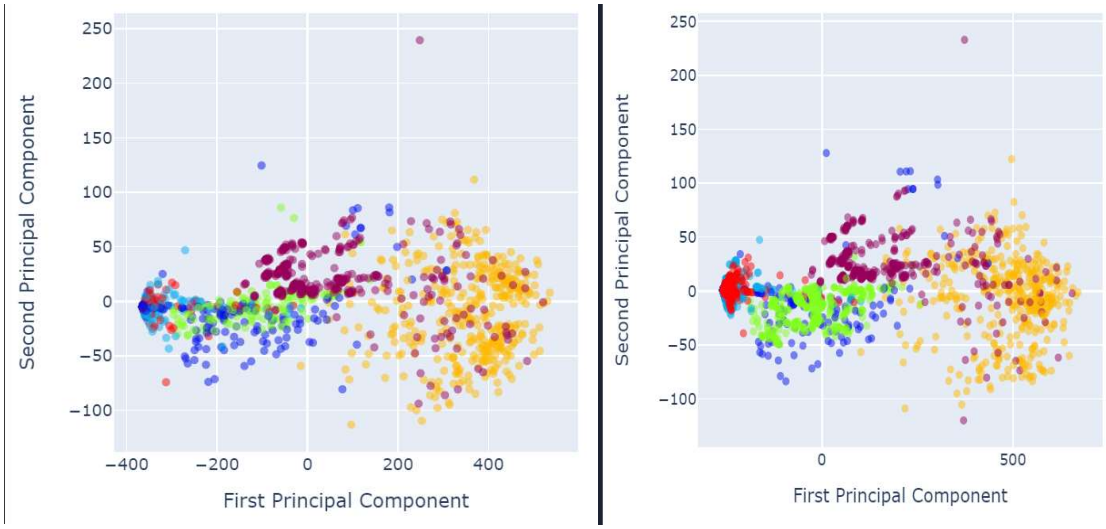
```
16/16 [==============================] - 7s 406ms/step
              precision    recall  f1-score   support

           0       0.87      0.88      0.88        78
           1       0.97      0.96      0.97        80
           2       0.95      1.00      0.97        69
           3       0.99      0.99      0.99        98
           4       0.91      0.86      0.88        84
           5       0.99      1.00      0.99        93

    accuracy                           0.95       502
   macro avg       0.95      0.95      0.95       502
weighted avg       0.95      0.95      0.95       502
```



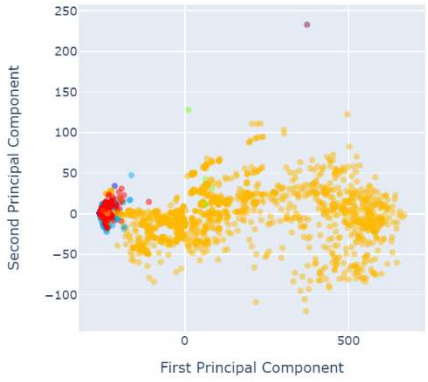




PCA before and after Sampling



PCA after CNN



Appendix 3-5.1: SMOTEN Oversampling 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTEN(random_state=0)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 95m 15.6s

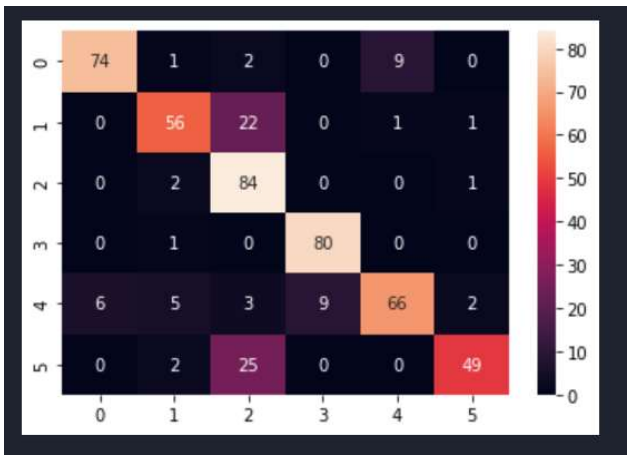
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

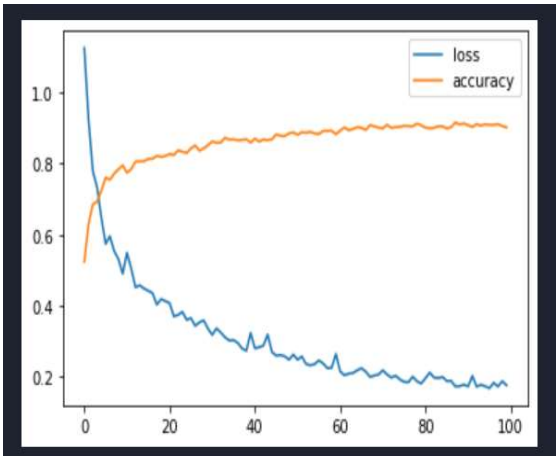
✓ 0.2s

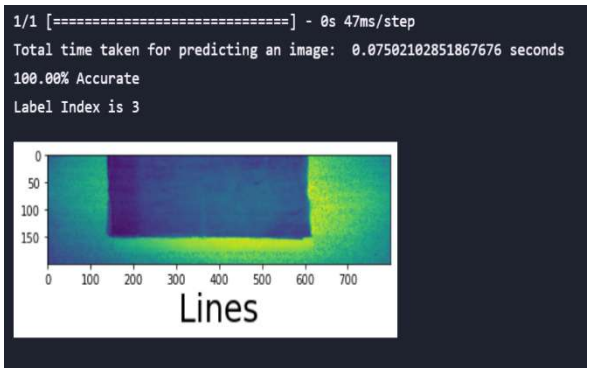
```
Resampled dataset shape Counter({4: 417, 1: 417, 5: 417, 3: 417, 2: 417, 0: 417})
```

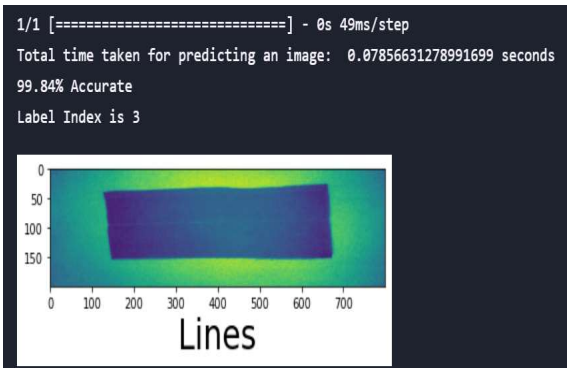
```
16/16 [==============================] - 12s 760ms/step
```

	precision	recall	f1-score	support
0	0.93	0.86	0.89	86
1	0.84	0.70	0.76	80
2	0.62	0.97	0.75	87
3	0.90	0.99	0.94	81
4	0.87	0.73	0.79	91
5	0.92	0.64	0.76	76
accuracy			0.82	501
macro avg	0.85	0.81	0.82	501
weighted avg	0.84	0.82	0.82	501

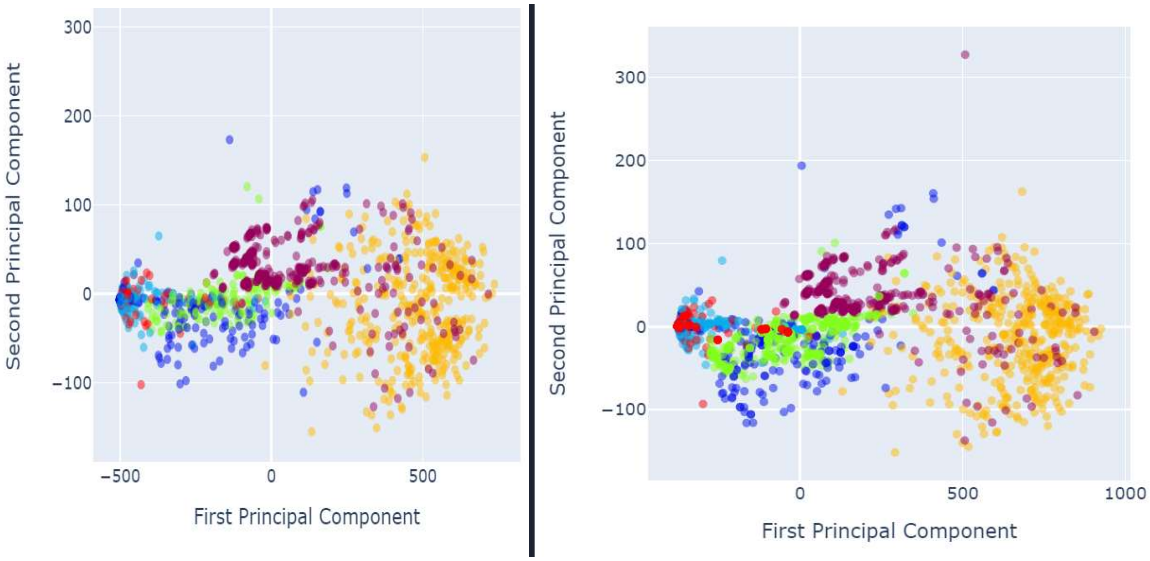




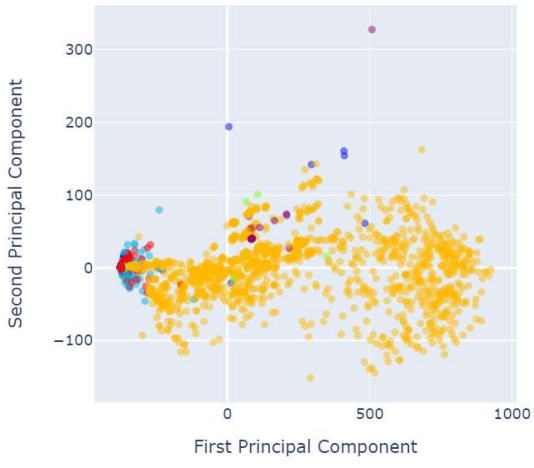




PCA before and after sampling



PCA After CNN



Appendix 3-5.2: SMOTEN Oversampling 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTEN(random_state=0)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

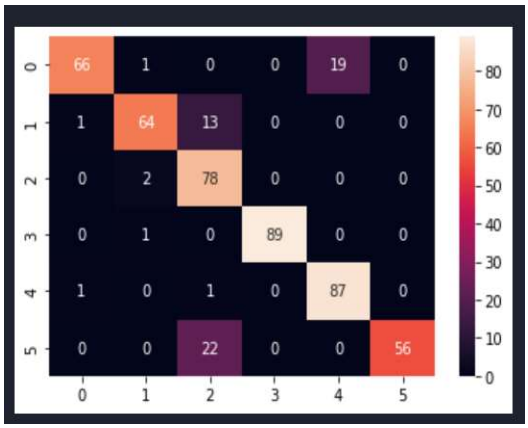
✓ 78m 36.3s

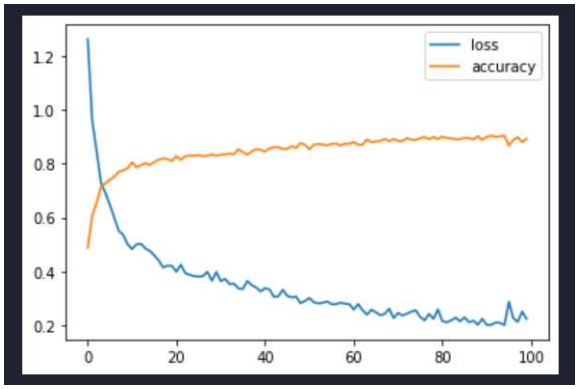
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

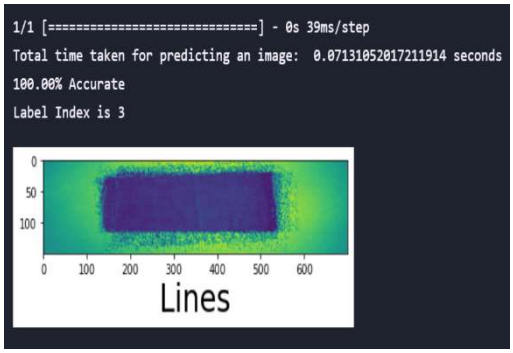
✓ 0.4s

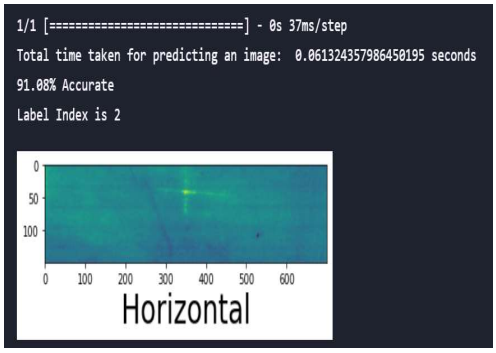
```
Resampled dataset shape Counter({3: 417, 4: 417, 1: 417, 2: 417, 5: 417, 0: 417})
```

```
16/16 [==============================] - 14s 857ms/step
              precision    recall  f1-score   support

           0       0.97      0.77      0.86        86
           1       0.94      0.82      0.88        78
           2       0.68      0.97      0.80        80
           3       1.00      0.99      0.99        90
           4       0.82      0.98      0.89        89
           5       1.00      0.72      0.84        78

    accuracy                           0.88       501
   macro avg       0.90      0.87      0.88       501
weighted avg       0.90      0.88      0.88       501
```

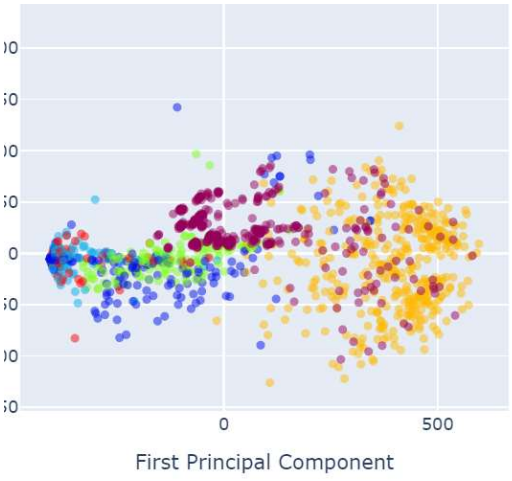


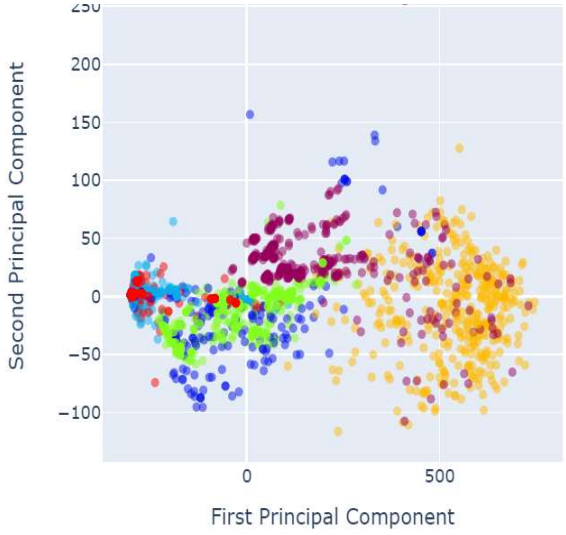




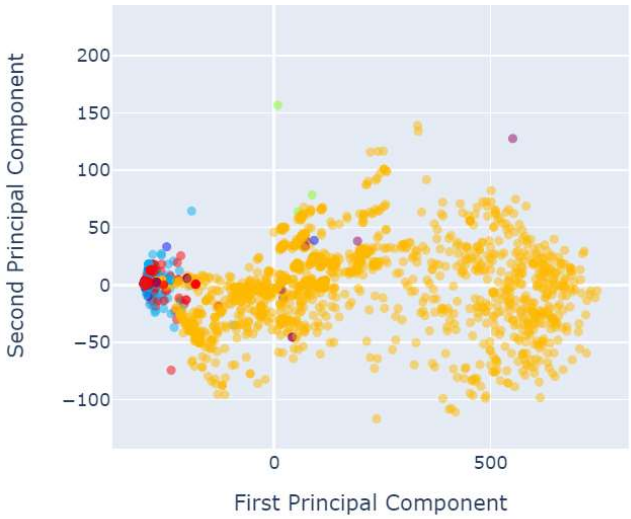


PCA before and after sampling





PCA After CNN



Appendix 3-5.3: SMOTEN Oversampling 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTEN(random_state=0)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 53m 4.5s

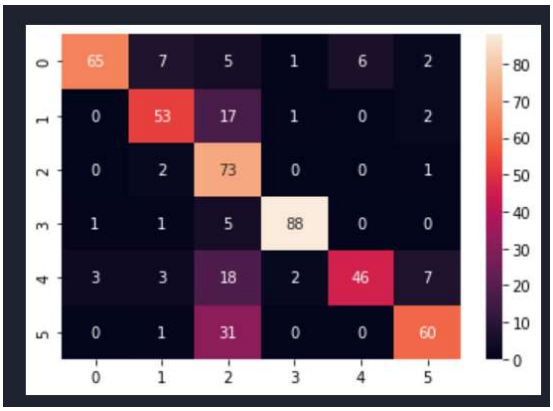
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

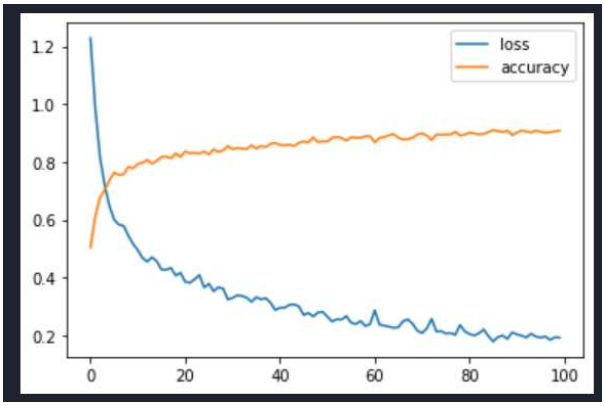
✓ 0.1s

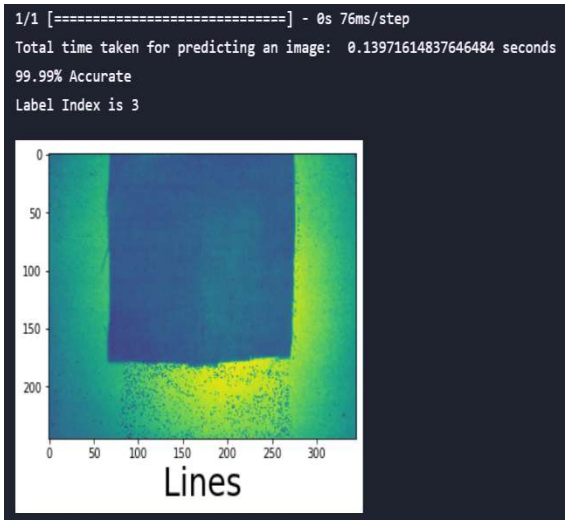
```
Resampled dataset shape Counter({1: 417, 4: 417, 5: 417, 2: 417, 3: 417, 0: 417})
```

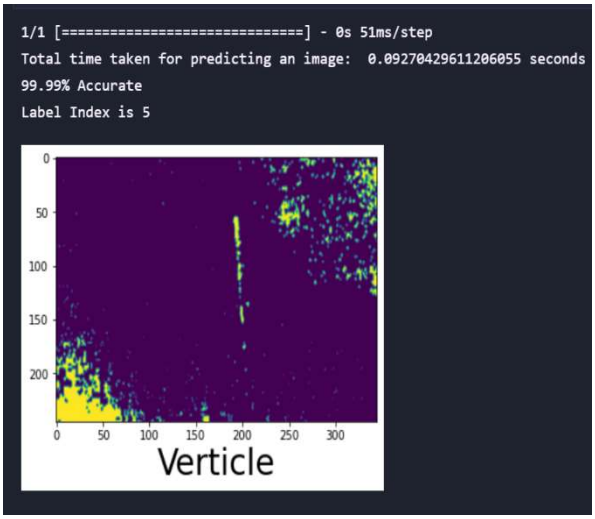
```
16/16 [==============================] - 9s 528ms/step
```

	precision	recall	f1-score	support
0	0.94	0.76	0.84	86
1	0.79	0.73	0.76	73
2	0.49	0.96	0.65	76
3	0.96	0.93	0.94	95
4	0.88	0.58	0.70	79
5	0.83	0.65	0.73	92
accuracy			0.77	501
macro avg	0.82	0.77	0.77	501
weighted avg	0.83	0.77	0.78	501

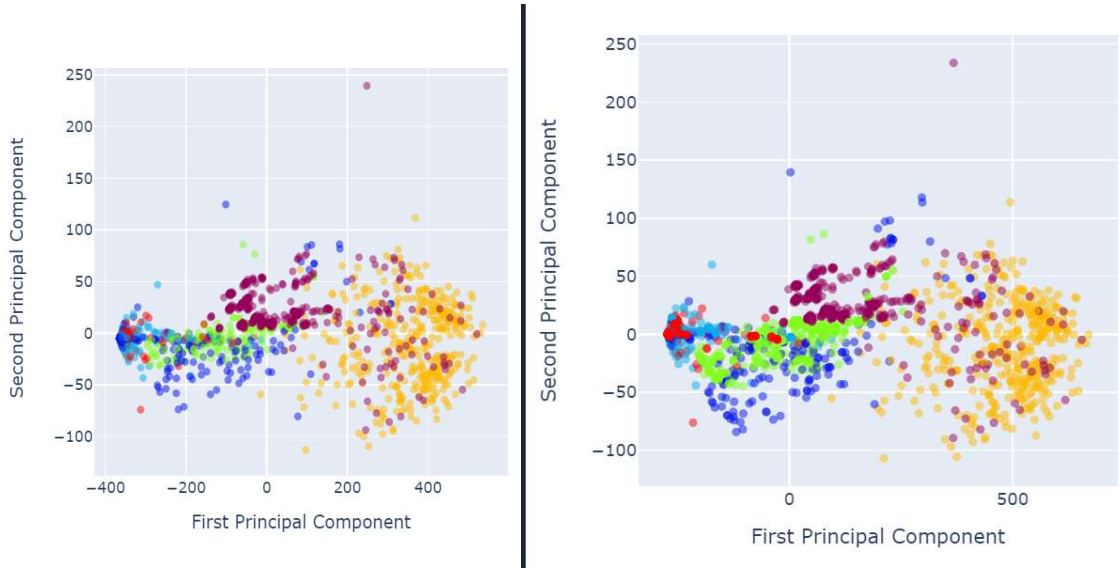




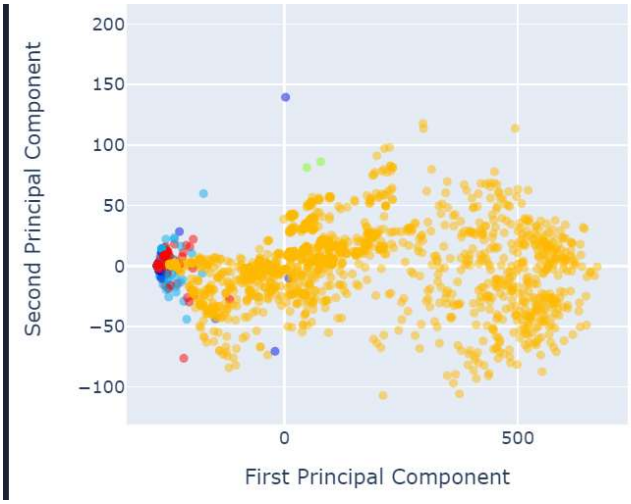




PCA before and after sampling



PCA After CNN



Appendix 3-6.1: SVM SMOTE Size 200x800

```
# Reshape the given X train for the resampling technique
X_res = X.reshape(X.shape[0], -1)
y_res = y.ravel()
print(f'Original dataset shape {Counter(y_res)}')
```

✓ 0.6s

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SVMSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 7m 4.7s

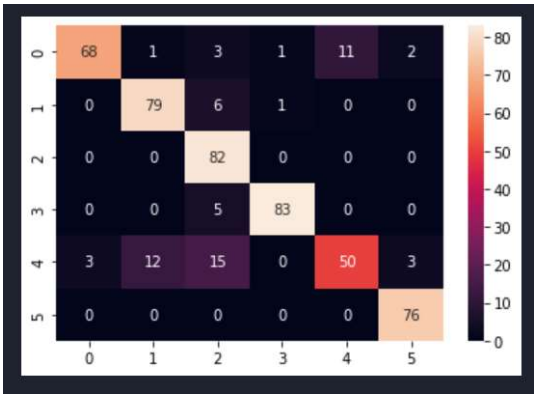
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

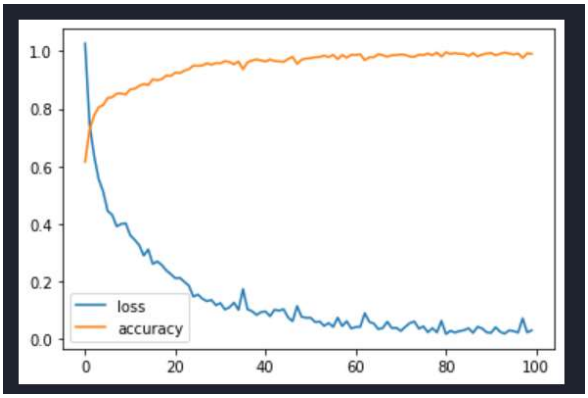
✓ 0.3s

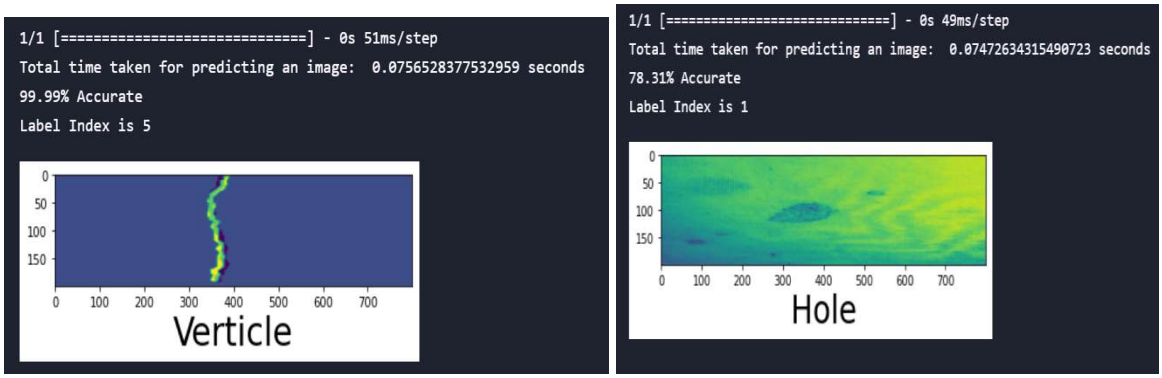
```
Resampled dataset shape Counter({1: 417, 4: 417, 3: 417, 2: 417, 5: 417, 0: 417})
```

```
16/16 [==============================] - 12s 760ms/step
```

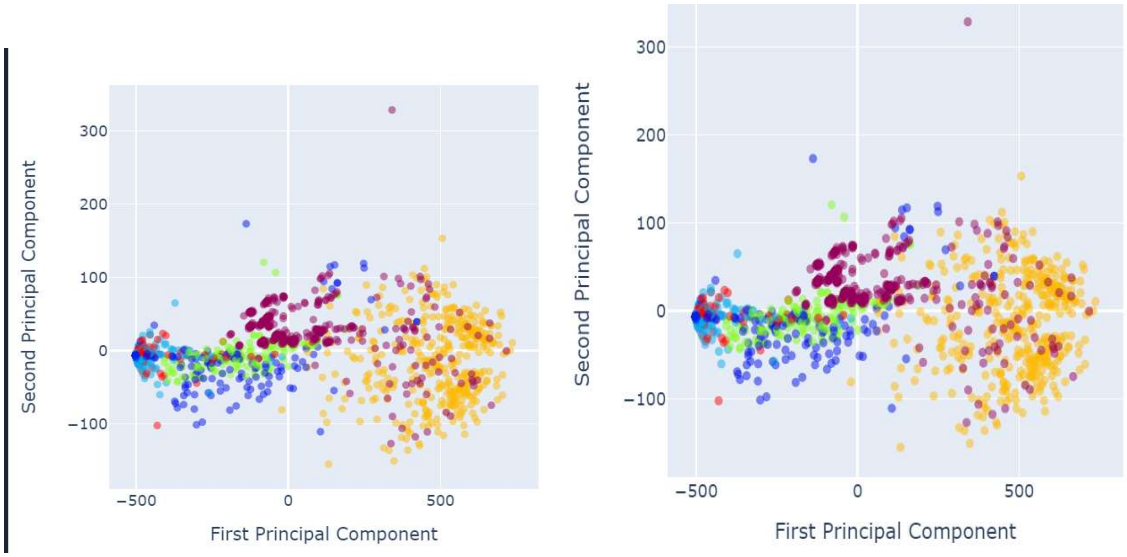
	precision	recall	f1-score	support
0	0.96	0.79	0.87	86
1	0.86	0.92	0.89	86
2	0.74	1.00	0.85	82
3	0.98	0.94	0.96	88
4	0.82	0.60	0.69	83
5	0.94	1.00	0.97	76
accuracy			0.87	501
macro avg	0.88	0.88	0.87	501
weighted avg	0.88	0.87	0.87	501



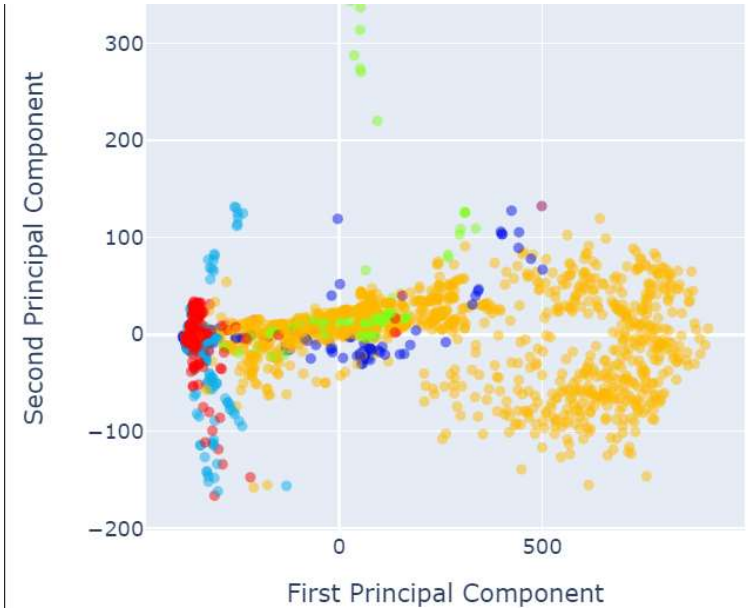




PCA Before and After Sampling



PCA after CNN



Appendix 3-6.2: SVM SMOTE Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SVMSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

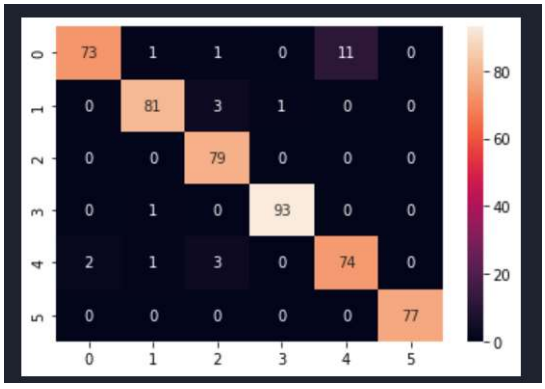
✓ 4m 52.2s

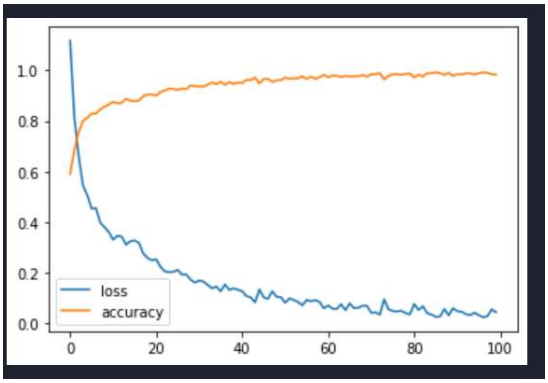
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

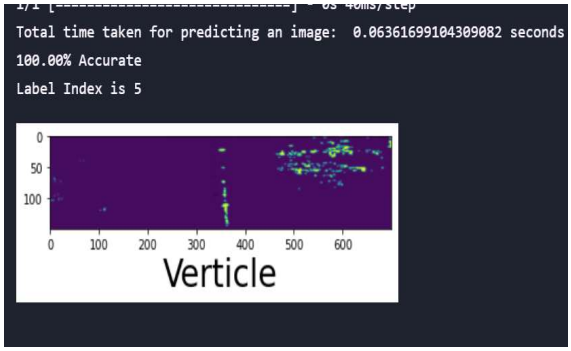
✓ 0.3s

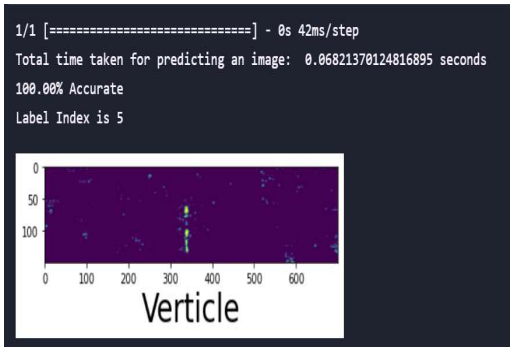
```
Resampled dataset shape Counter({5: 417, 4: 417, 2: 417, 3: 417, 1: 417, 0: 417})
```

```
16/16 [==============================] - 8s 498ms/step
              precision    recall  f1-score   support

           0       0.97      0.85      0.91        86
           1       0.96      0.95      0.96        85
           2       0.92      1.00      0.96        79
           3       0.99      0.99      0.99        94
           4       0.87      0.93      0.90        80
           5       1.00      1.00      1.00        77

    accuracy                           0.95       501
   macro avg       0.95      0.95      0.95       501
weighted avg       0.95      0.95      0.95       501
```

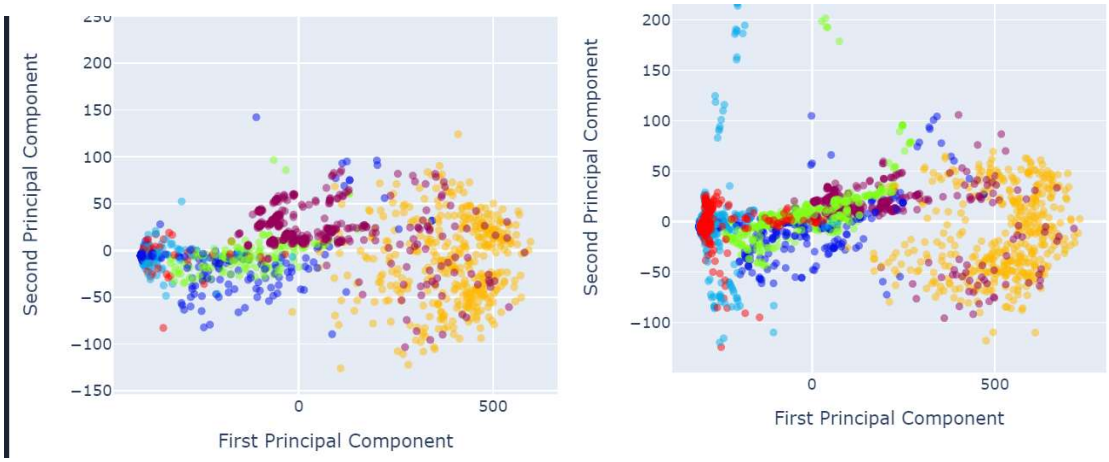








PCA before and after sampling



PCA after CNN

Appendix 3-6.3: SVM SMOTE Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

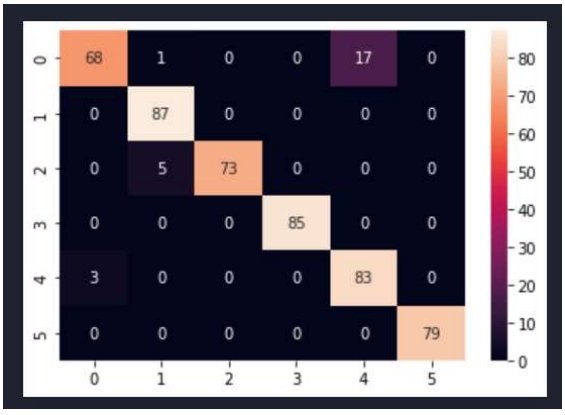
```
ost = SVMSMOTE(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

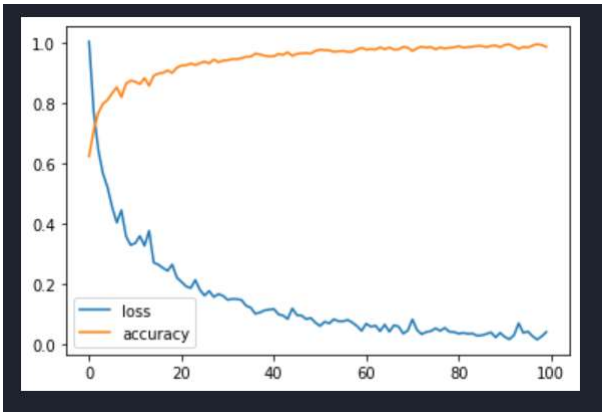
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

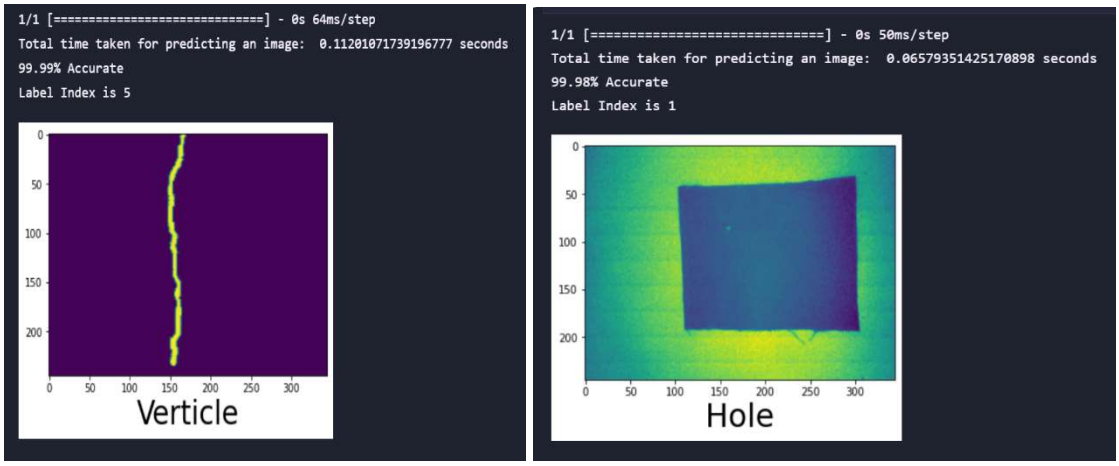
```
Resampled dataset shape Counter({4: 417, 5: 417, 1: 417, 3: 417, 2: 417, 0: 417})
```

```
16/16 [==============================] - 7s 431ms/step
```

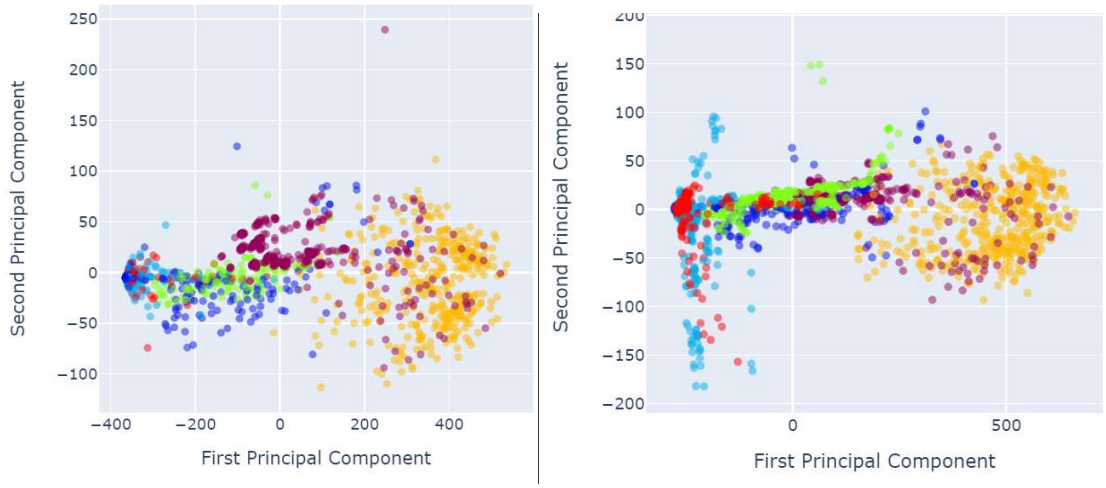
	precision	recall	f1-score	support
0	0.96	0.79	0.87	86
1	0.94	1.00	0.97	87
2	1.00	0.94	0.97	78
3	1.00	1.00	1.00	85
4	0.83	0.97	0.89	86
5	1.00	1.00	1.00	79
accuracy			0.95	501
macro avg	0.95	0.95	0.95	501
weighted avg	0.95	0.95	0.95	501



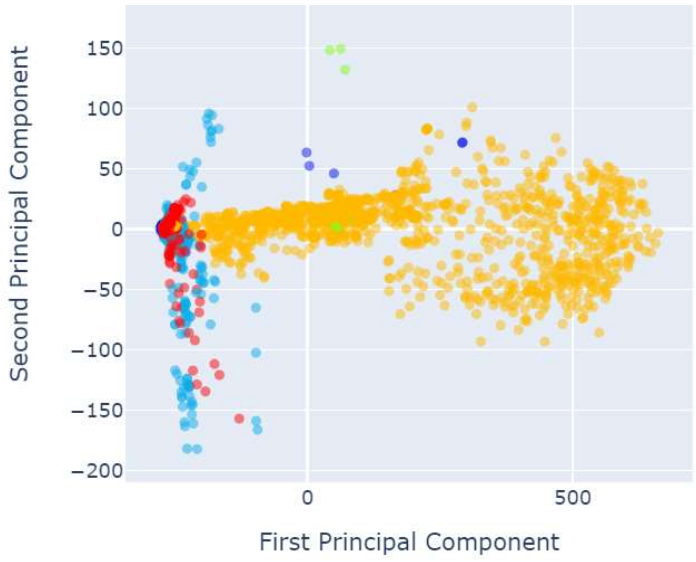




PCA Before and after sampling



PCA After CNN



Appendix 3-7.1: SMOTENC Oversampling Size 200x800

```
    ost = SMOTENC(random_state=42, categorical_features=[18, 19])
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

⊗ 9m 19.6s

```
Output exceeds the size limit. Open the full output data in a text editor
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
e:\DBS Studies\ML\CNN\Fabric_Defects\SMOTENC_Oversampling_200_800.ipynb Cell 30 in <cell line: 2>()
      1 ost = SMOTENC(random_state=42, categorical_features=[18, 19])
----> 2 X_ost, y_ost = ost.fit_resample(X_res, y_res)

File ~\AppData\Roaming\Python\Python310\site-packages\imblearn\base.py:83, in SamplerMixin.fit_resample(self, X, y)
     77 X, y, binarize_y = self._check_X_y(X, y)
     79 self.sampling_strategy_ = check_sampling_strategy(
     80     self.sampling_strategy, y, self._sampling_type
     81 )
---> 83 output = self._fit_resample(X, y)
     85 y_ = (
     86     label_binarize(output[1], classes=np.unique(y)) if binarize_y else output[1]
     87 )
     89 X_, y_ = arrays_transformer.transform(output[0], y_)
```

```
File ~\AppData\Roaming\Python\Python310\site-packages\imblearn\over_sampling\_smote\base.py:332, in SMOTE._fit_resample(self, X, y)
...
--> 673 data = np.empty(nnz, dtype=dtype)
    674 idx_dtype = get_index_dtype(maxval=max(shape))
    675 row = np.empty(nnz, dtype=idx_dtype)

MemoryError: Unable to allocate 2.39 GiB for an array with shape (320673346,) and data type float64
```

Appendix 3-7.2: SMOTENC Oversampling Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
    ost = SMOTENC(random_state=42, categorical_features=[18, 19])
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 4m 29.8s

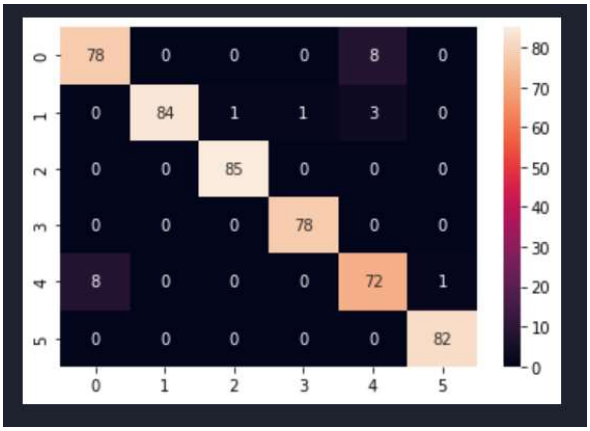
+ Code + Markdown

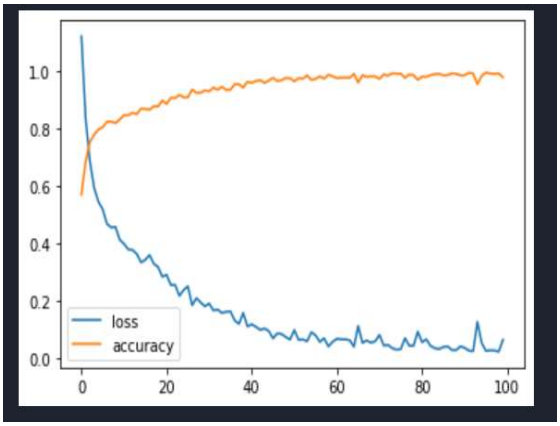
```
    print(f'Resampled dataset shape {Counter(y_ost)}')
```

✓ 0.4s

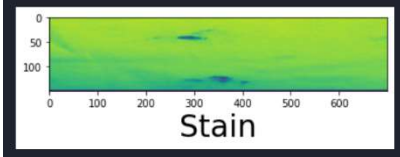
```
Resampled dataset shape Counter({1: 417, 4: 417, 5: 417, 3: 417, 2: 417, 0: 417})
```

```
16/16 [==============================] - 8s 514ms/step
              precision    recall  f1-score   support

           0       0.91      0.91      0.91        86
           1       1.00      0.94      0.97        89
           2       0.99      1.00      0.99        85
           3       0.99      1.00      0.99        78
           4       0.87      0.89      0.88        81
           5       0.99      1.00      0.99        82

    accuracy                           0.96       501
   macro avg       0.96      0.96      0.96       501
weighted avg       0.96      0.96      0.96       501
```

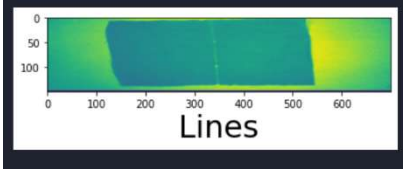




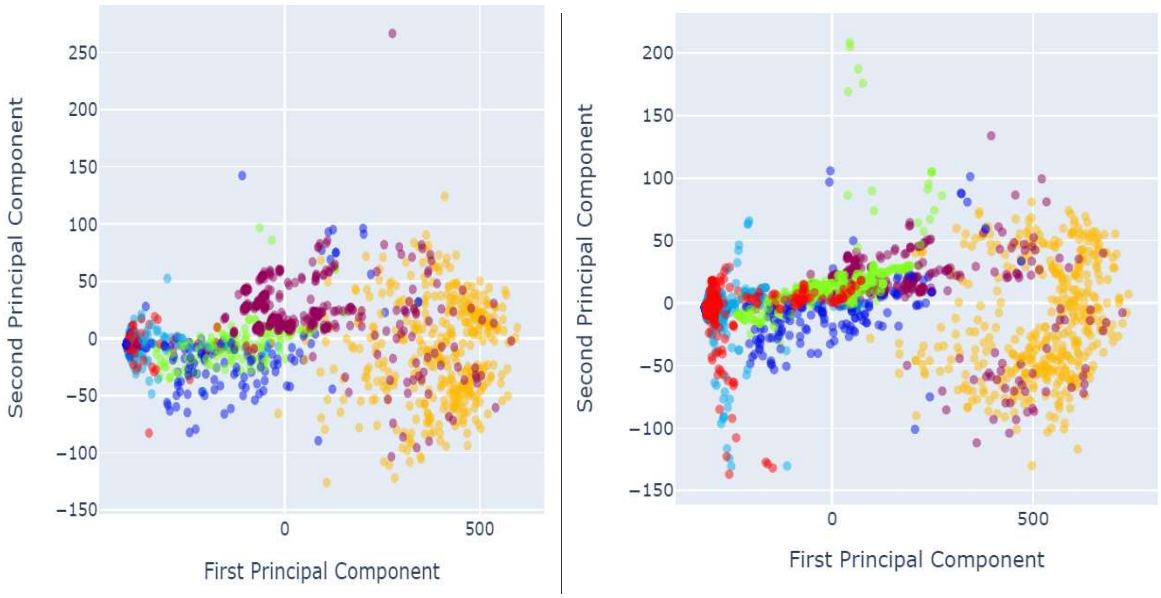

```
1/1 [==============================] - 0s 41ms/step
Total time taken for predicting an image:  0.06807374954223633 seconds
99.92% Accurate
Label Index is 4
```



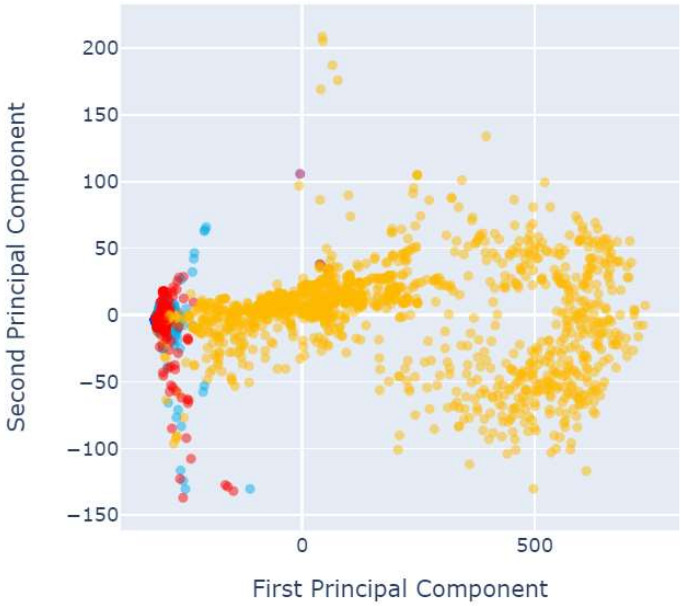

```
1/1 [==============================] - 0s 43ms/step
Total time taken for predicting an image:  0.07146382331848145 seconds
100.00% Accurate
Label Index is 3
```



PCA before and after sampling



PCA After CNN



Appendix 3-7.3: SMOTENC Oversampling Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTENC(random_state=42, categorical_features=[18, 19])
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

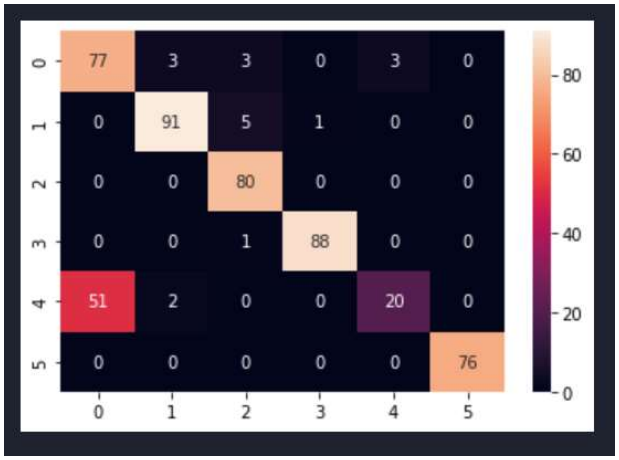
✓ 2m 52.8s

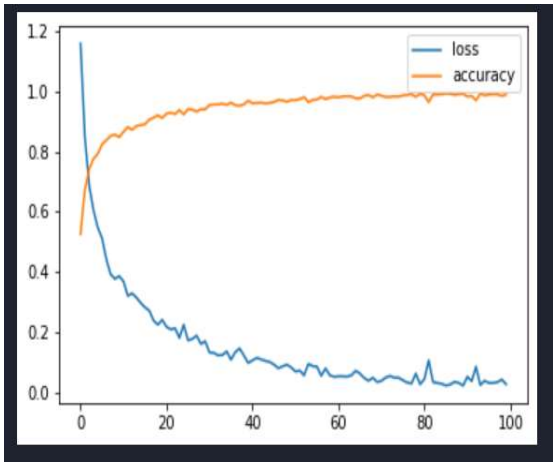
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

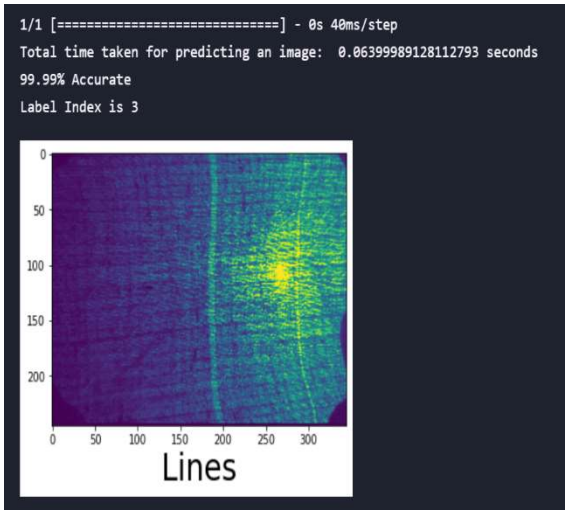
✓ 0.7s

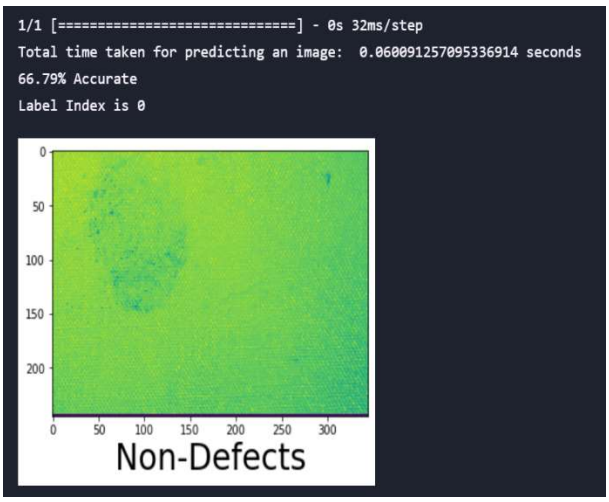
```
Resampled dataset shape Counter({1: 417, 5: 417, 2: 417, 4: 417, 3: 417, 0: 417})
```

```
16/16 [==============================] - 7s 439ms/step
              precision    recall  f1-score   support

           0       0.60      0.90      0.72        86
           1       0.95      0.94      0.94        97
           2       0.90      1.00      0.95        80
           3       0.99      0.99      0.99        89
           4       0.87      0.27      0.42        73
           5       1.00      1.00      1.00        76

    accuracy                           0.86       501
   macro avg       0.88      0.85      0.84       501
weighted avg       0.88      0.86      0.85       501
```

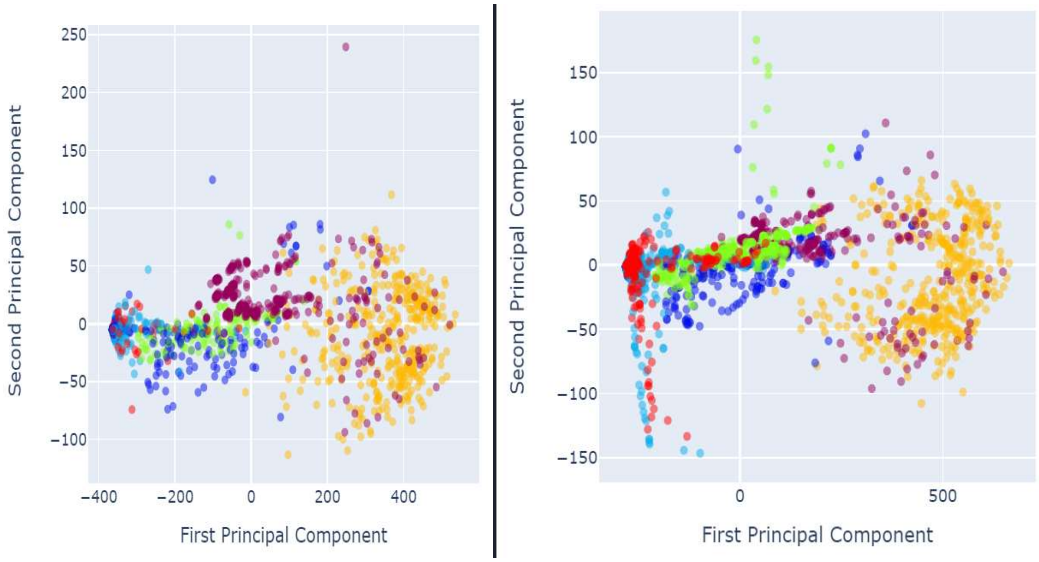




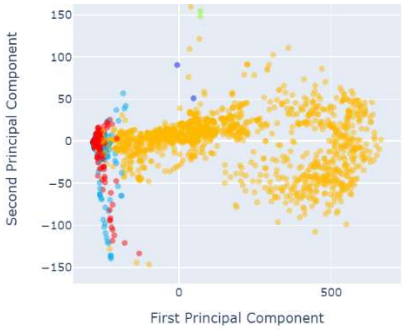




PCA Before and After Sampling



PCA After CNN



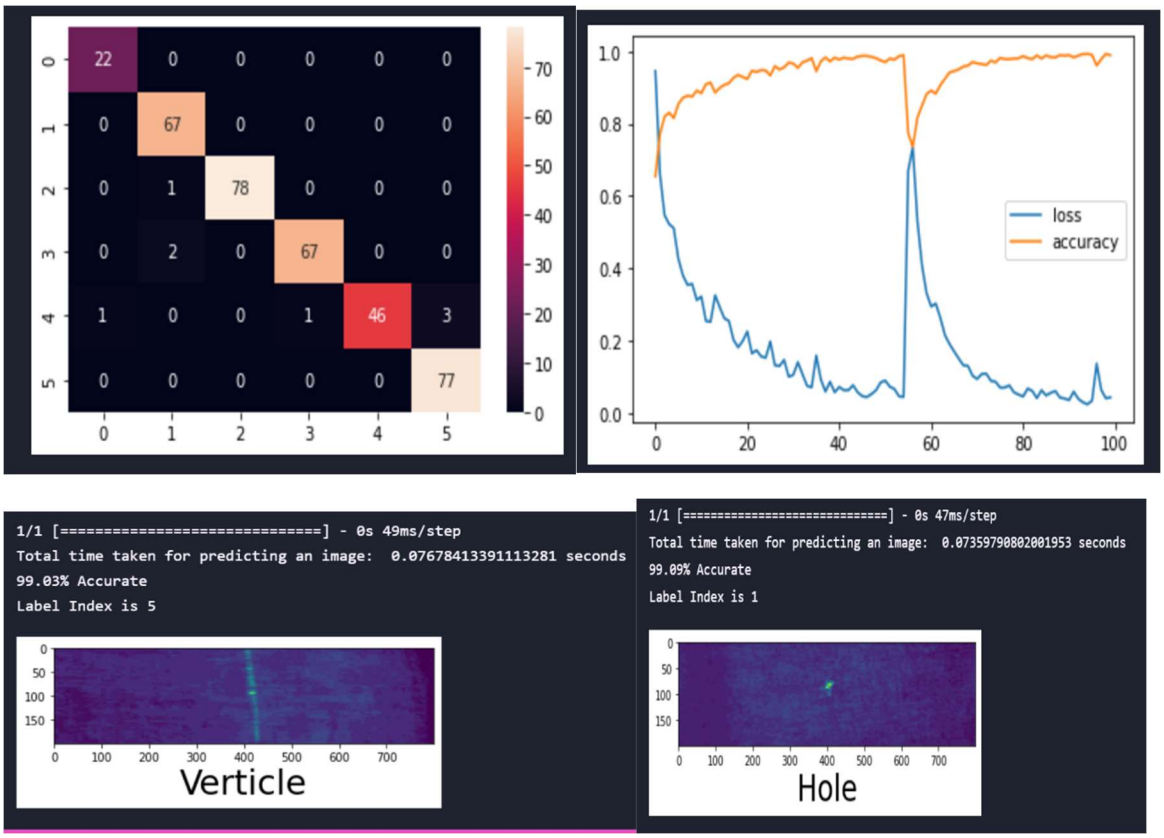
Appendix 4: Oversampling and Undersampling Combined

Appendix 4-1.1: SMOTEENN Oversampling and Undersampling 200x800

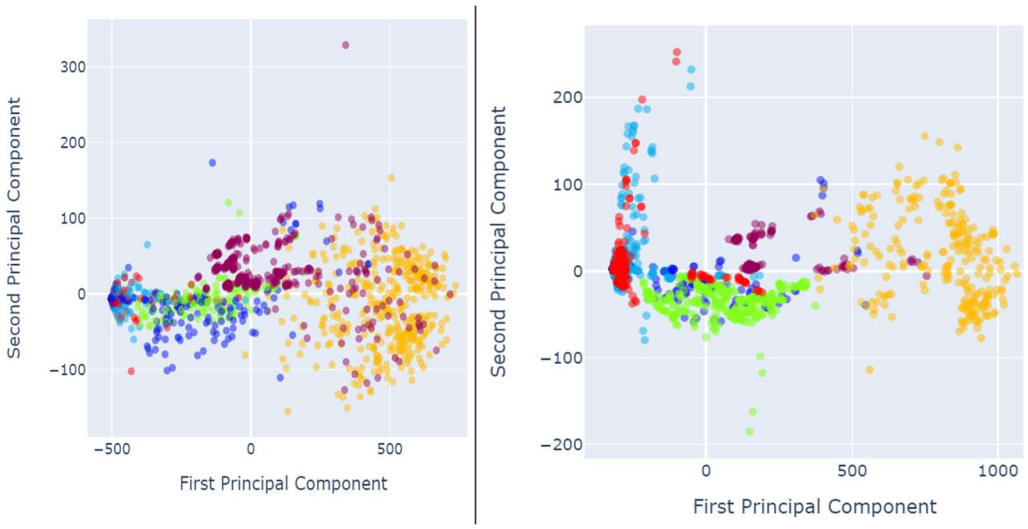
```
    # Reshape the given X train for the resampling technique
    X_res = X.reshape(X.shape[0], -1)
    y_res = y.ravel()
    print(f'Original dataset shape {Counter(y_res)}')
✓ 0.5s

Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
    ost = SMOTEENN(random_state=42)
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
✓ 1m 53.5s

    print(f'Resampled dataset shape {Counter(y_ost)}')
✓ 0.3s

Resampled dataset shape Counter({3: 399, 5: 374, 2: 351, 1: 322, 4: 245, 0: 132})
```

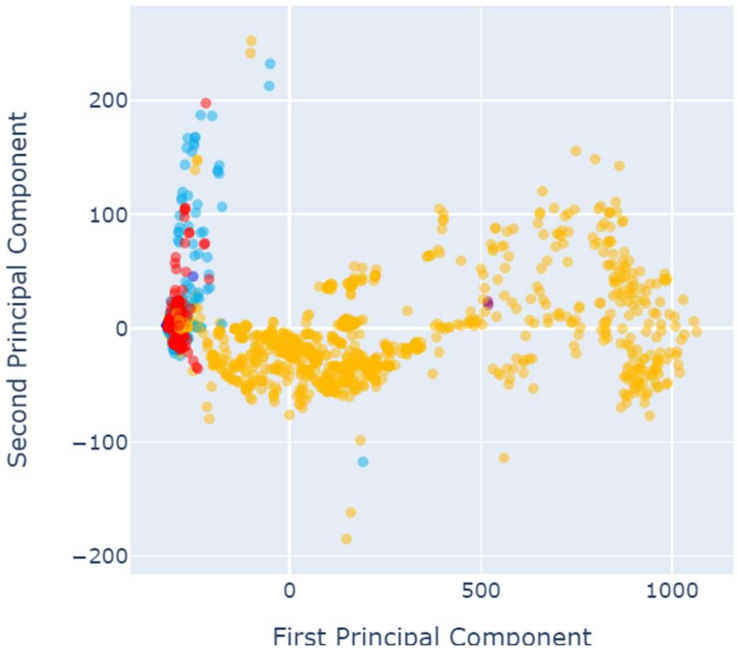
```
12/12 [==============================] - 9s 732ms/step
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       0.96      1.00      0.98        67
           2       1.00      0.99      0.99        79
           3       0.99      0.97      0.98        69
           4       1.00      0.90      0.95        51
           5       0.96      1.00      0.98        77

    accuracy                           0.98       365
   macro avg       0.98      0.98      0.98       365
weighted avg       0.98      0.98      0.98       365
```



PCA Before and After Sampling



PCA After CNN



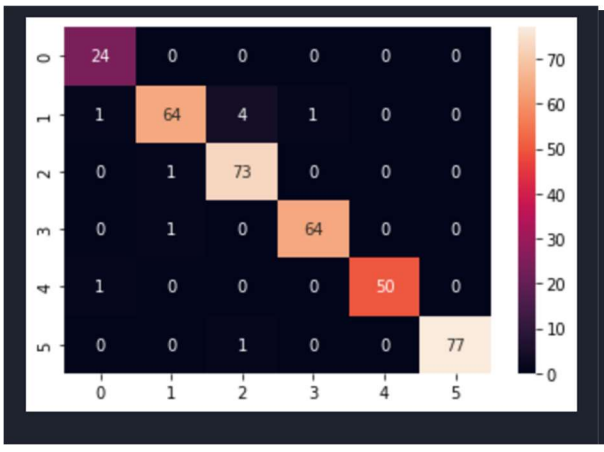
Appendix 4-1.2: SMOTEENN Oversampling and Undersampling 150x700

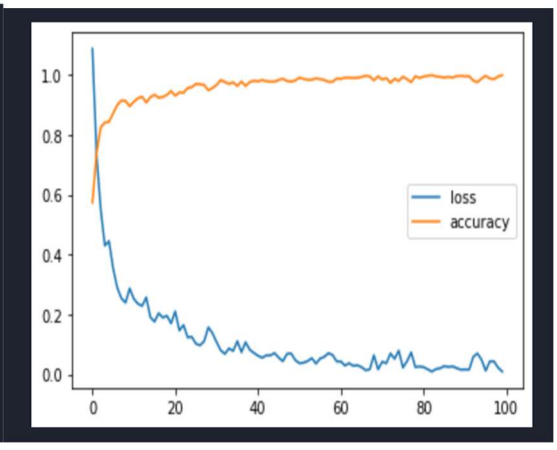
```
    # Reshape the given X train for the resampling technique
    X_res = X.reshape(X.shape[0], -1)
    y_res = y.ravel()
    print(f'Original dataset shape {Counter(y_res)}')
✓ 0.3s

Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

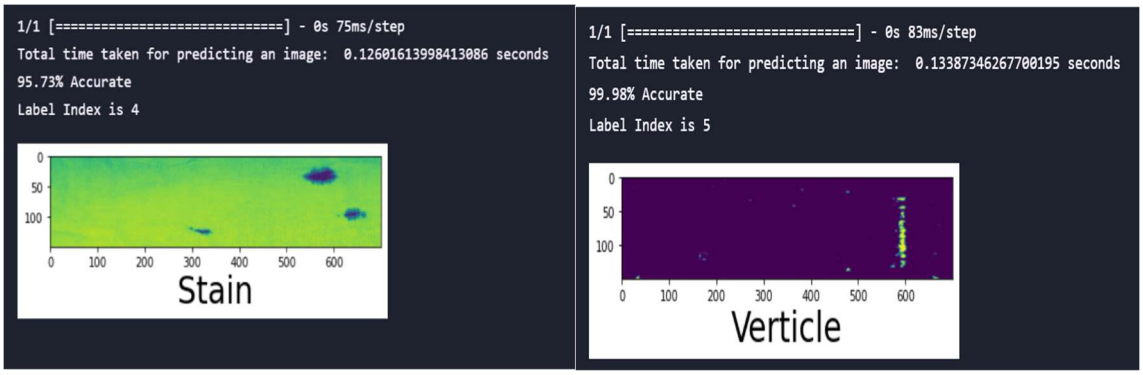
```
    ost = SMOTEENN(random_state=42)
    X_ost, y_ost = ost.fit_resample(X_res, y_res)
✓ 43.7s
```

```
    print(f'Resampled dataset shape {Counter(y_ost)}')
✓ 0.1s

Resampled dataset shape Counter({3: 394, 5: 365, 2: 355, 1: 316, 4: 234, 0: 143})
```

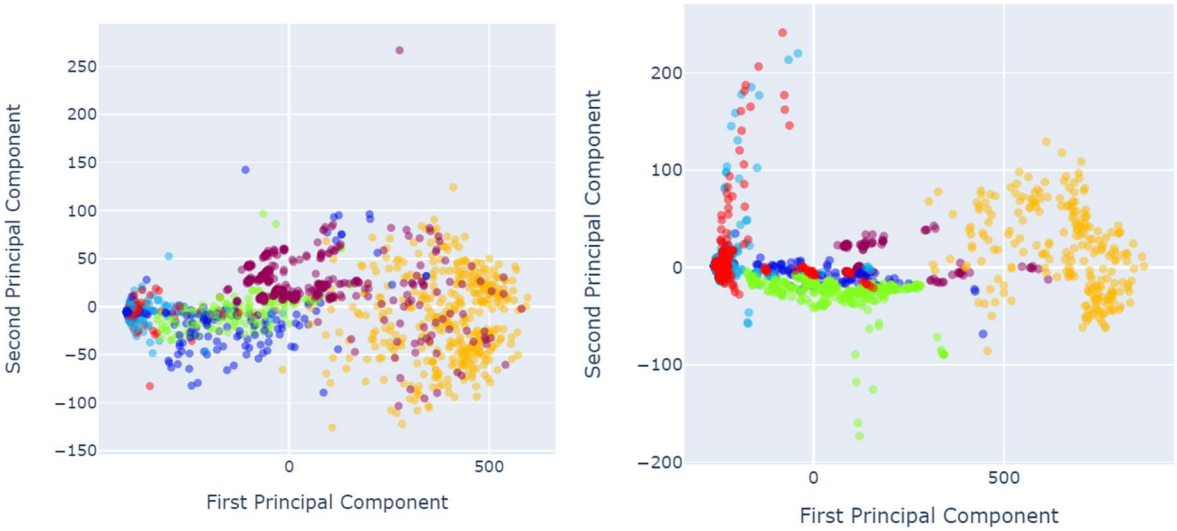
```
12/12 [==============================] - 6s 507ms/step
              precision    recall  f1-score   support

           0       0.92      1.00      0.96        24
           1       0.97      0.91      0.94        70
           2       0.94      0.99      0.96        74
           3       0.98      0.98      0.98        65
           4       1.00      0.98      0.99        51
           5       1.00      0.99      0.99        78

    accuracy                           0.97       362
   macro avg       0.97      0.98      0.97       362
weighted avg       0.97      0.97      0.97       362
```



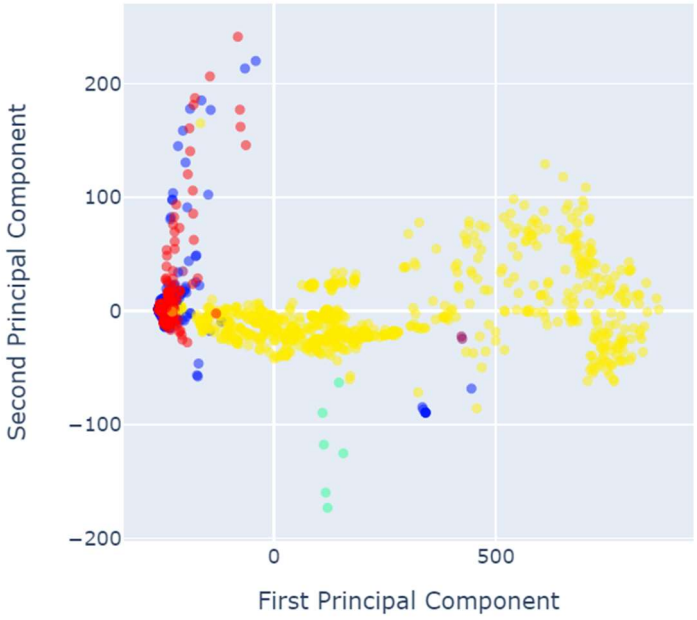




PCA Before and after Sampling



PCA after CNN



Appendix 4-1.3: SMOTEENN Oversampling and Undersampling 245x345

```
# Reshape the given X train for the resampling technique
X_res = X.reshape(X.shape[0], -1)
y_res = y.ravel()
print(f'Original dataset shape {Counter(y_res)}')
```

✓ 0.4s

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTEENN(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 25.4s

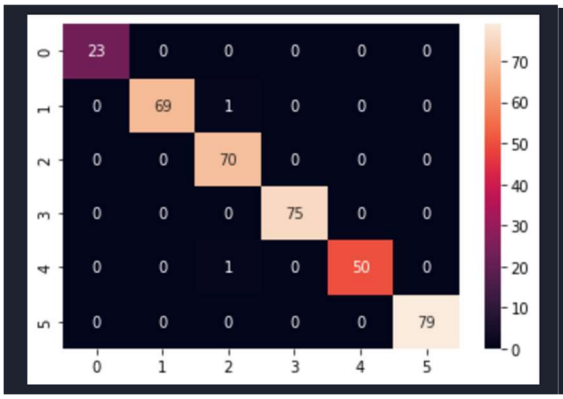
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

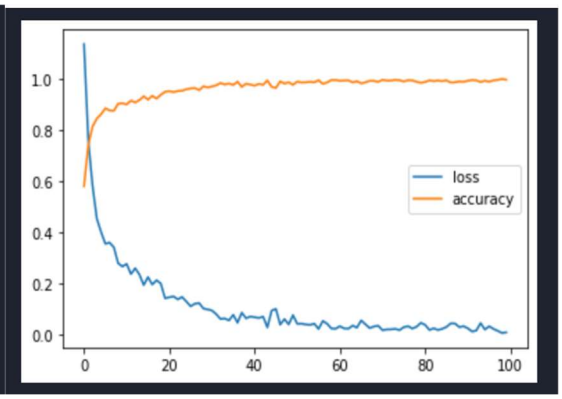
✓ 0.5s

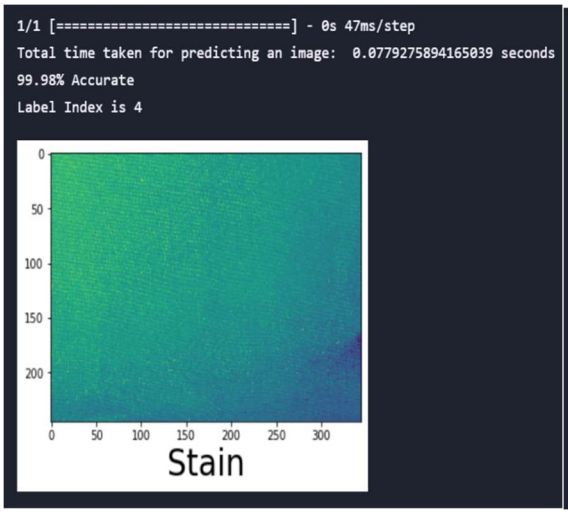
```
Resampled dataset shape Counter({3: 401, 5: 377, 2: 364, 1: 318, 4: 238, 0: 138})
```

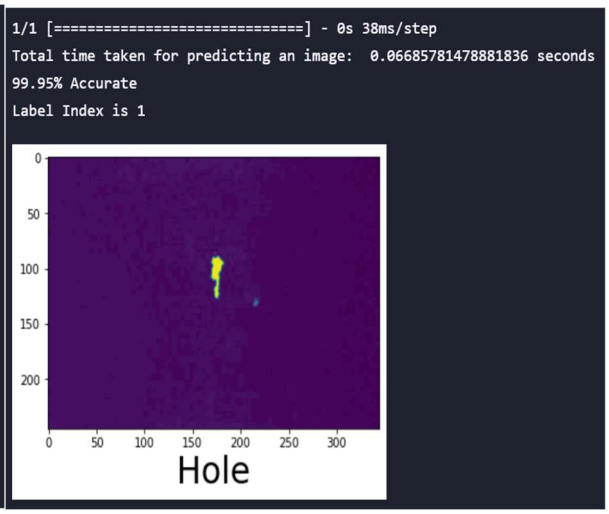
```
12/12 [==============================] - 5s 405ms/step
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	23
1	1.00	0.99	0.99	70
2	0.97	1.00	0.99	70
3	1.00	1.00	1.00	75
4	1.00	0.98	0.99	51
5	1.00	1.00	1.00	79
accuracy			0.99	368
macro avg	1.00	0.99	0.99	368
weighted avg	0.99	0.99	0.99	368

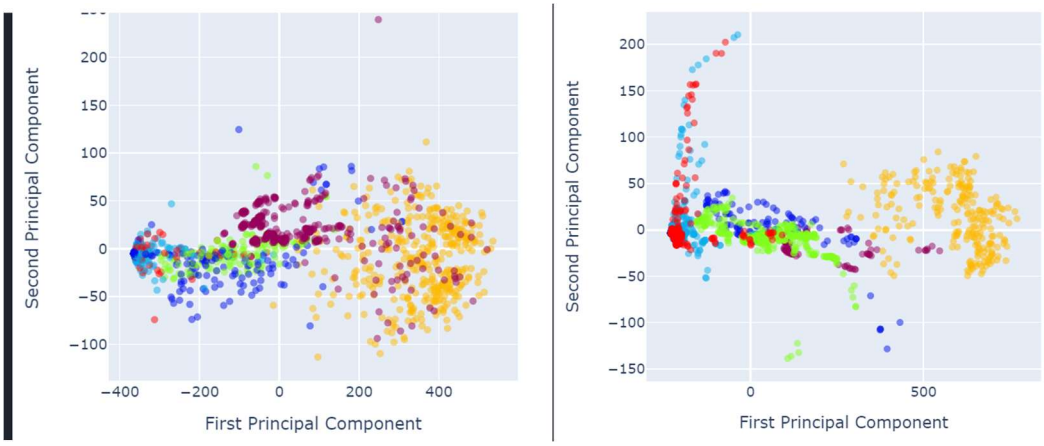




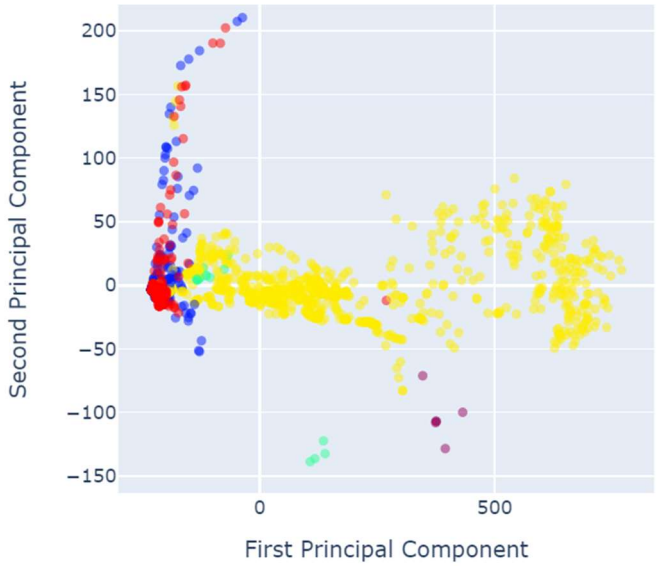




PCA Before and After Sampling



PCA After CNN



Appendix 4-2.1: Tomek Oversampling Undersampling 200x800

```
# Reshape the given X train for the resampling technique
X_res = X.reshape(X.shape[0], -1)
y_res = y.ravel()
print(f'Original dataset shape {Counter(y_res)}')
```

✓ 0.3s

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTETomek(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 44.3s

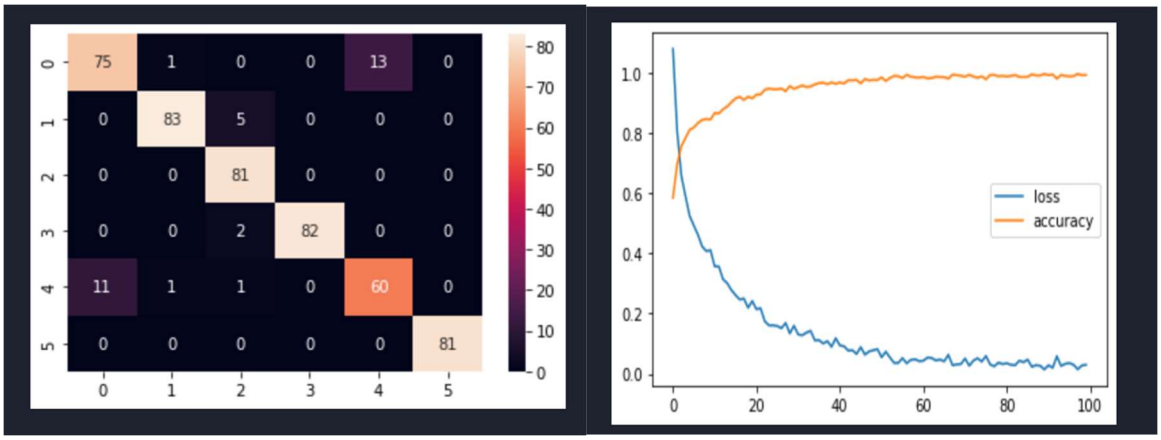
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

✓ 0.3s

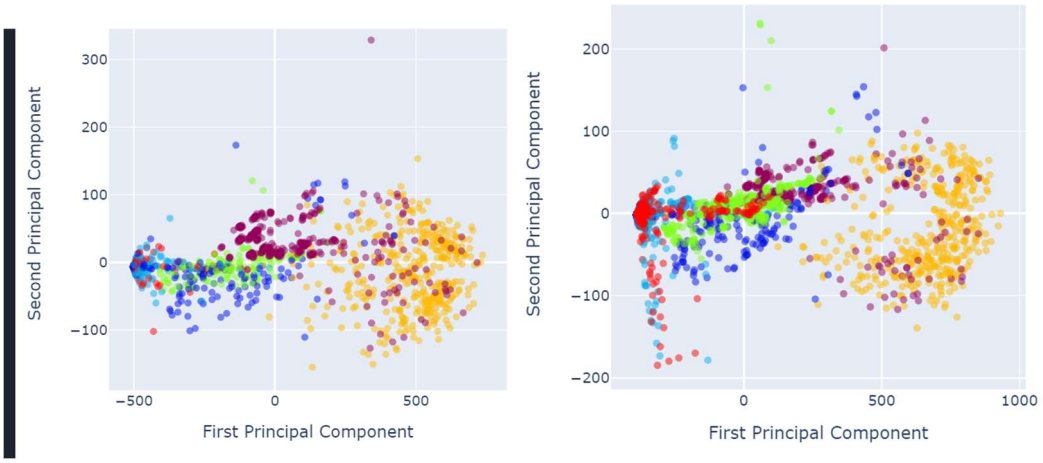
```
Resampled dataset shape Counter({3: 417, 5: 417, 1: 415, 2: 415, 4: 408, 0: 408})
```

```
16/16 [==============================] - 12s 747ms/step
```

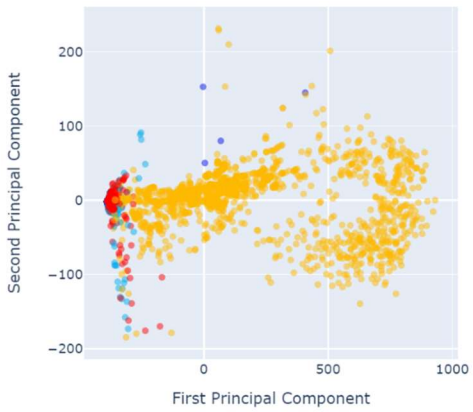
	precision	recall	f1-score	support
0	0.87	0.84	0.86	89
1	0.98	0.94	0.96	88
2	0.91	1.00	0.95	81
3	1.00	0.98	0.99	84
4	0.82	0.82	0.82	73
5	1.00	1.00	1.00	81
accuracy			0.93	496
macro avg	0.93	0.93	0.93	496
weighted avg	0.93	0.93	0.93	496



PCA Before and after Sampling



PCA After CNN



Appendix 4-2.2: SMOTE Tomek Oversampling Undersampling 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTETomek(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

✓ 14.2s

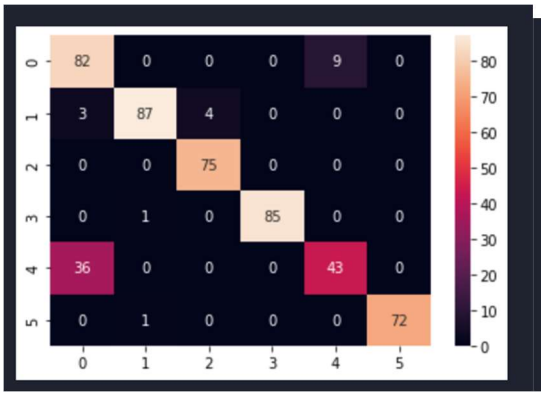
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

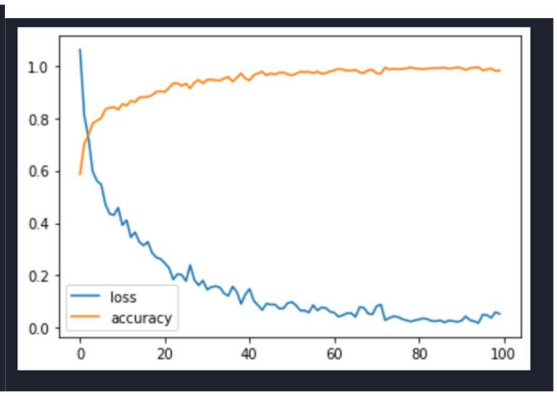
✓ 0.7s

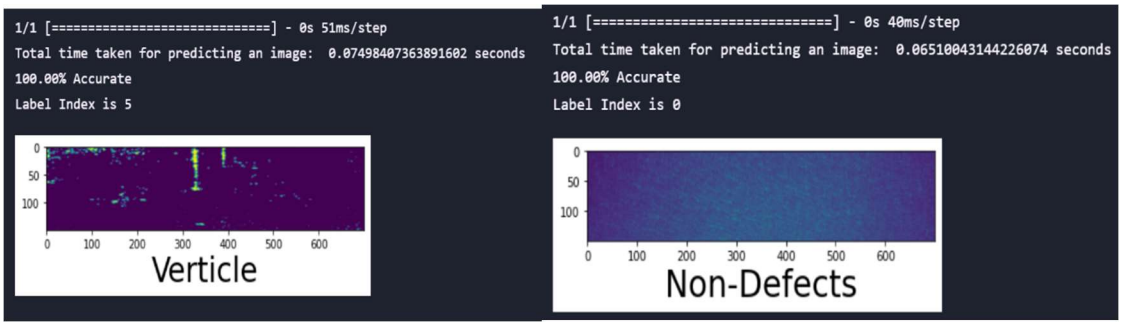
```
Resampled dataset shape Counter({1: 417, 2: 417, 5: 417, 3: 417, 4: 410, 0: 410})
```

```
16/16 [==============================] - 8s 506ms/step
```

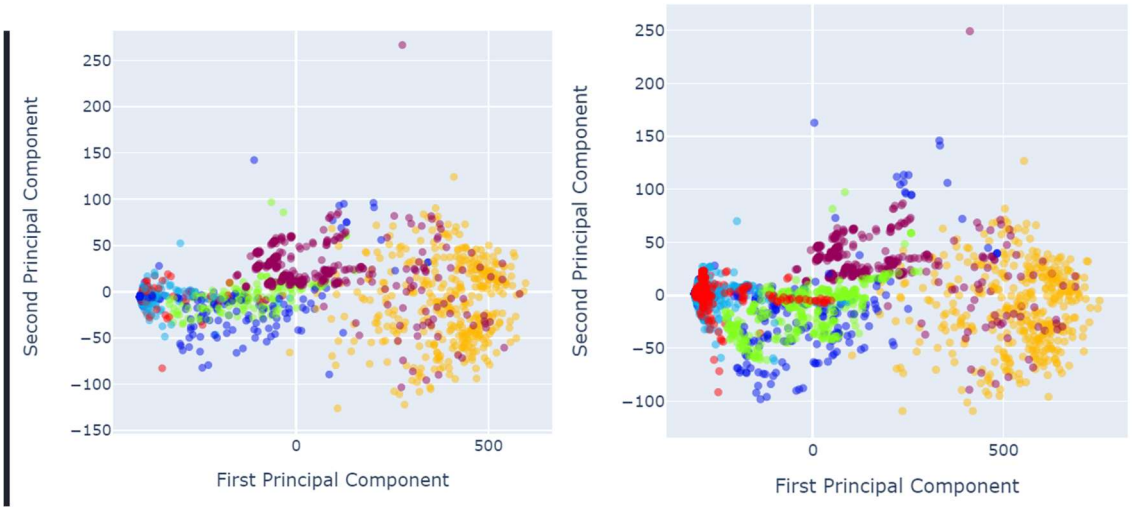
	precision	recall	f1-score	support
0	0.68	0.90	0.77	91
1	0.98	0.93	0.95	94
2	0.95	1.00	0.97	75
3	1.00	0.99	0.99	86
4	0.83	0.54	0.66	79
5	1.00	0.99	0.99	73
accuracy			0.89	498
macro avg	0.91	0.89	0.89	498
weighted avg	0.90	0.89	0.89	498



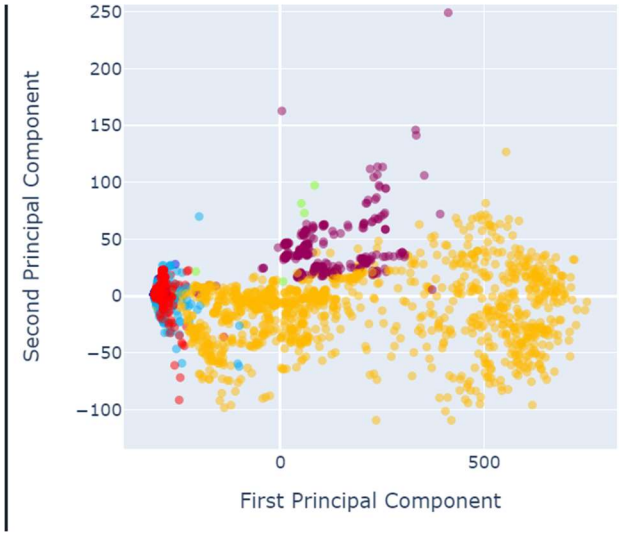




PCA Before and After Sampling



PCA After CNN



Appendix 4-2.3: SMOTE Tomek Oversampling Undersampling 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
ost = SMOTETomek(random_state=42)
X_ost, y_ost = ost.fit_resample(X_res, y_res)
```

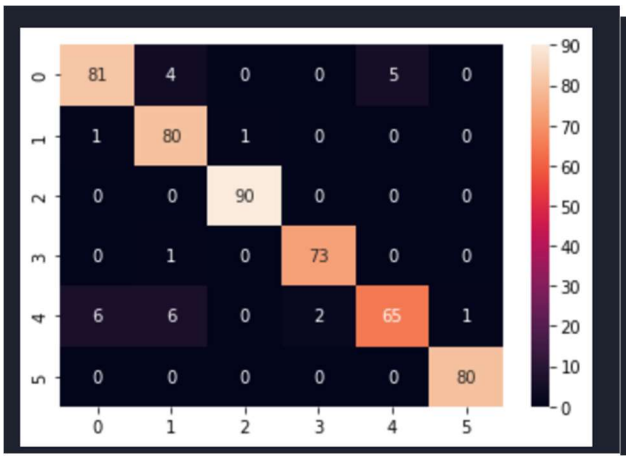
✓ 24.3s

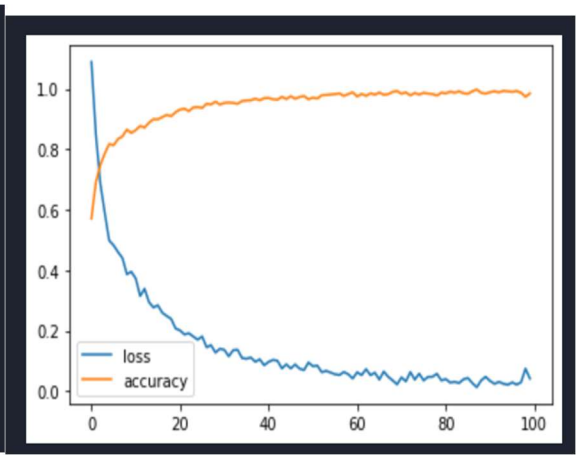
```
print(f'Resampled dataset shape {Counter(y_ost)}')
```

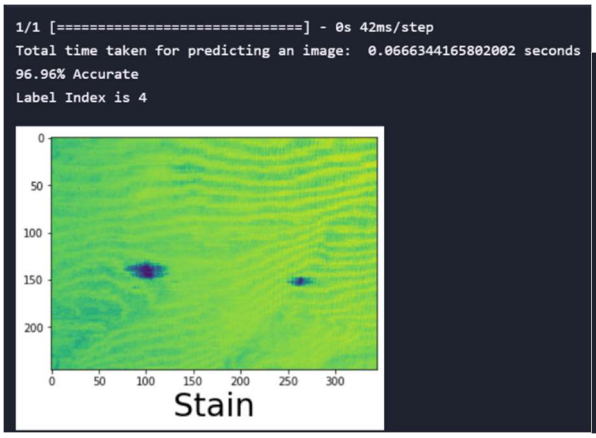
✓ 0.1s

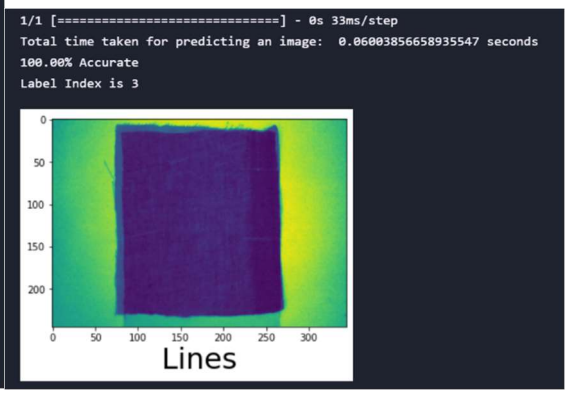
```
Resampled dataset shape Counter({5: 417, 2: 416, 3: 415, 1: 414, 4: 409, 0: 409})
```

```
16/16 [==============================] - 7s 408ms/step
              precision    recall  f1-score   support

           0       0.92      0.90      0.91        90
           1       0.88      0.98      0.92        82
           2       0.99      1.00      0.99        90
           3       0.97      0.99      0.98        74
           4       0.93      0.81      0.87        80
           5       0.99      1.00      0.99        80

    accuracy                           0.95       496
   macro avg       0.95      0.95      0.94       496
weighted avg       0.95      0.95      0.94       496
```

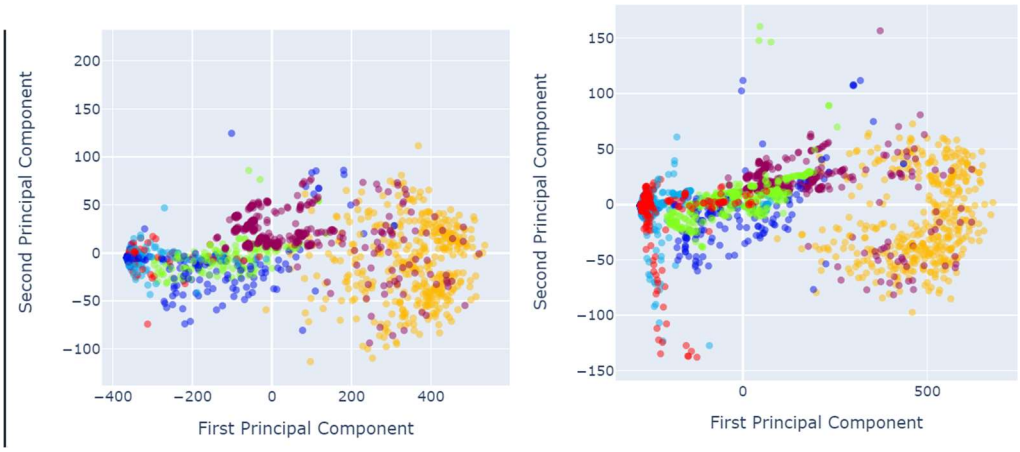




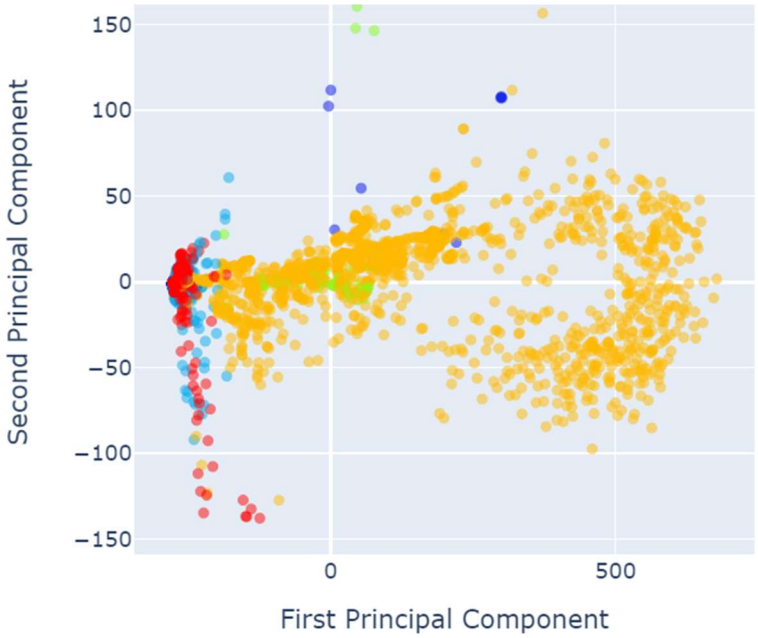




PCA Before and After Sampling



PCA After CNN



Appendix 5: Ensemble

Appendix 5-1.1: Pipeline Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
# Make pipeline
renn = RepeatedEditedNearestNeighbours()
smt = SMOTE(random_state=42)
enn = EditedNearestNeighbours()
pipeline = Pipeline([('smt', smt), ('enn', enn), ('renn', renn)])
```

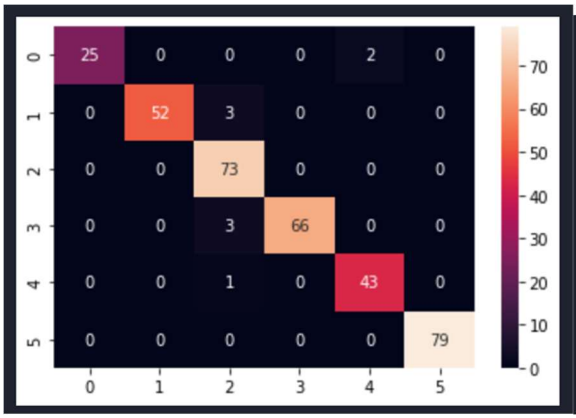
```
# apply function sampler
X_pdt, y_pdt = pipeline.fit_resample(X_res, y_res)
```

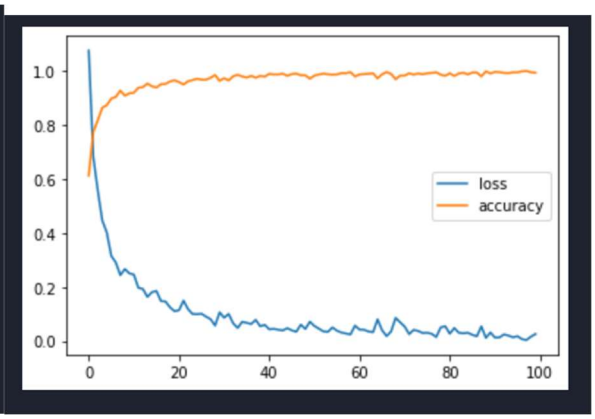
+ Code + Markdown

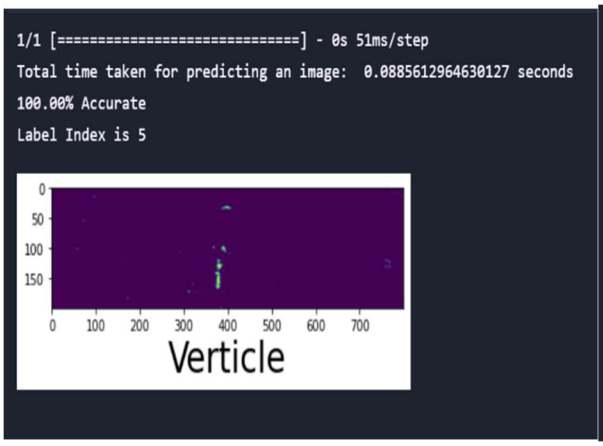
```
print(f'Resampled dataset shape {Counter(y_pdt)}')
```

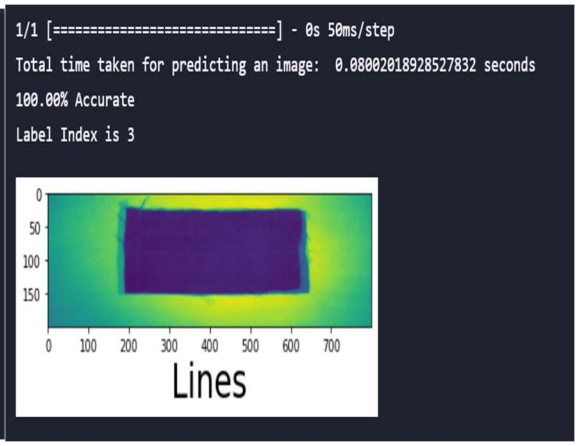
```
Resampled dataset shape Counter({3: 393, 5: 376, 2: 324, 1: 250, 4: 238, 0: 154})
```

```
11/11 [==============================] - 9s 773ms/step
              precision    recall  f1-score   support

           0       1.00      0.93      0.96        27
           1       1.00      0.95      0.97        55
           2       0.91      1.00      0.95        73
           3       1.00      0.96      0.98        69
           4       0.96      0.98      0.97        44
           5       1.00      1.00      1.00        79

    accuracy                           0.97       347
   macro avg       0.98      0.97      0.97       347
weighted avg       0.98      0.97      0.97       347
```

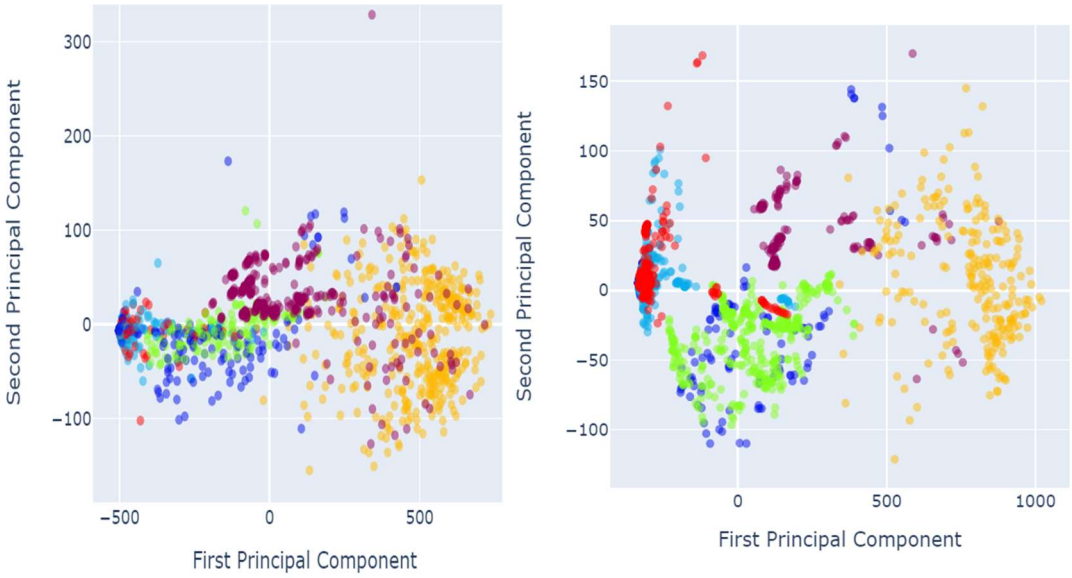




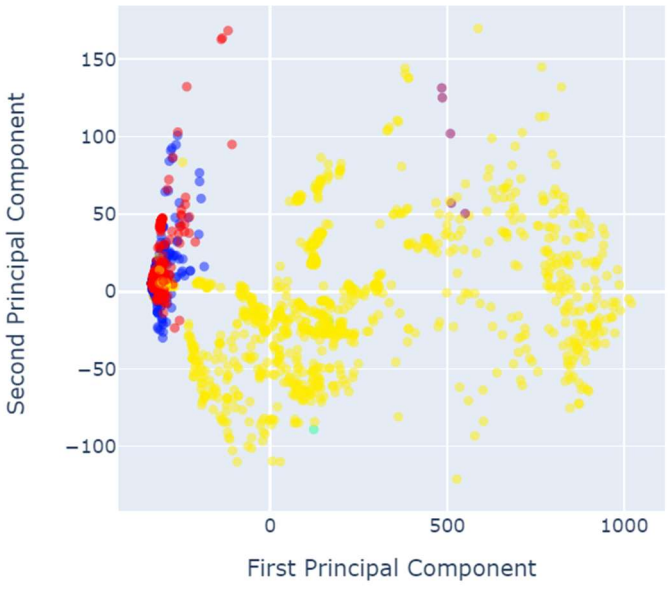




PCA Before and After Sampling



PCA After CNN



Appendix 5-1.2: Pipeline Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

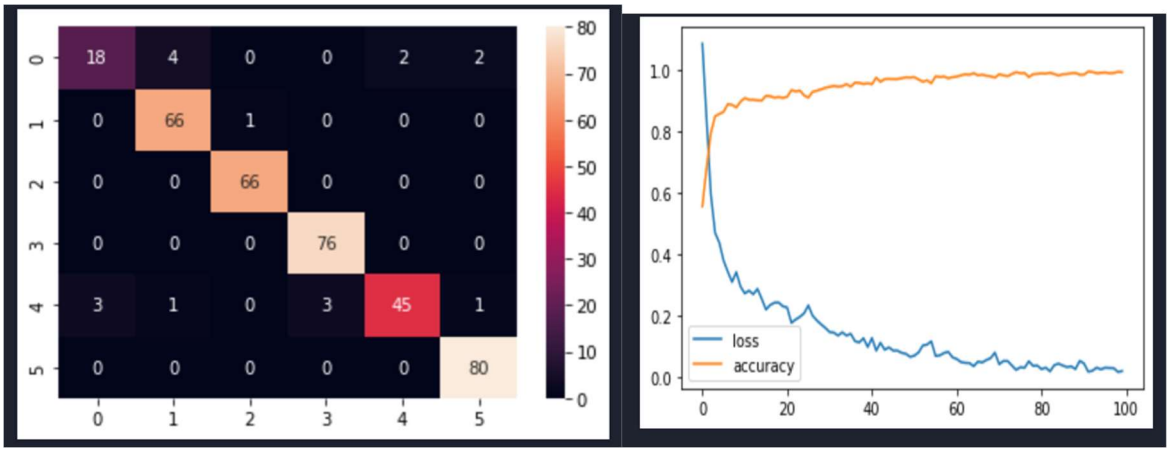
```
# Make pipeline
renn = RepeatedEditedNearestNeighbours()
smt = SMOTE(random_state=42)
enn = EditedNearestNeighbours()
pipeline = Pipeline([('smt', smt), ('enn', enn), ('renn', renn)])
```

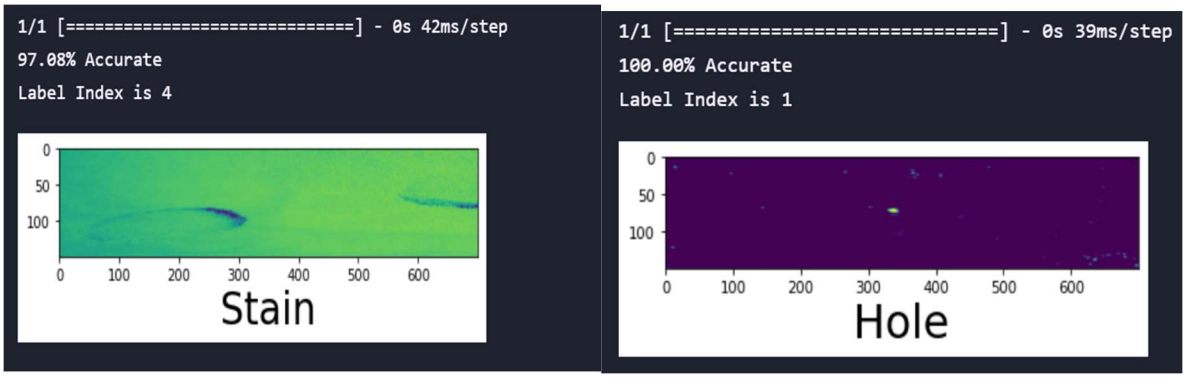
```
# apply function sampler
X_pdt, y_pdt = pipeline.fit_resample(X_res, y_res)
```

```
print(f'Resampled dataset shape {Counter(y_pdt)}')
```

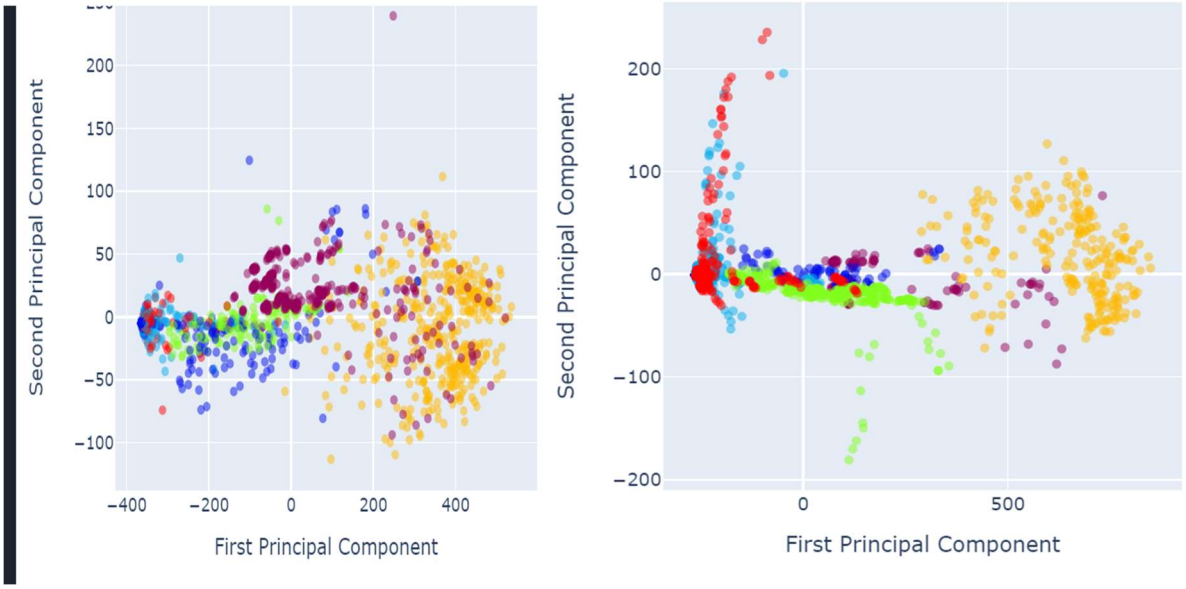
```
Resampled dataset shape Counter({3: 400, 5: 385, 2: 350, 1: 288, 4: 241, 0: 172})
```

```
12/12 [==============================] - 6s 490ms/step
              precision    recall  f1-score   support

           0       0.86      0.69      0.77        26
           1       0.93      0.99      0.96        67
           2       0.99      1.00      0.99        66
           3       0.96      1.00      0.98        76
           4       0.96      0.85      0.90        53
           5       0.96      1.00      0.98        80

    accuracy                           0.95       368
   macro avg       0.94      0.92      0.93       368
weighted avg       0.95      0.95      0.95       368
```

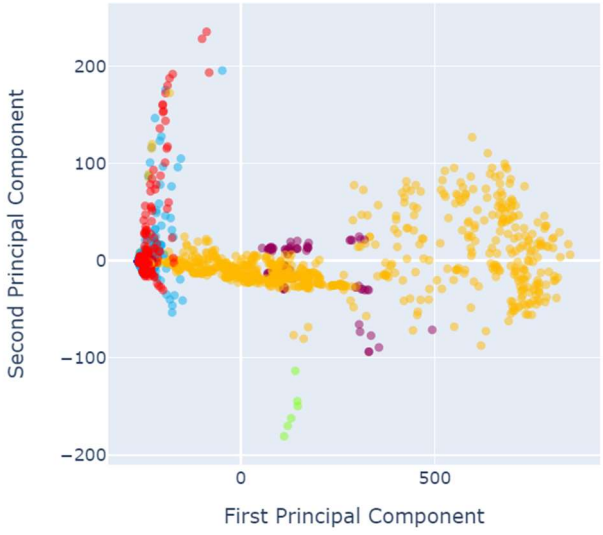




PCA Before and After CNN



PCA After CNN



Appendix 5-1.3: Pipeline Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

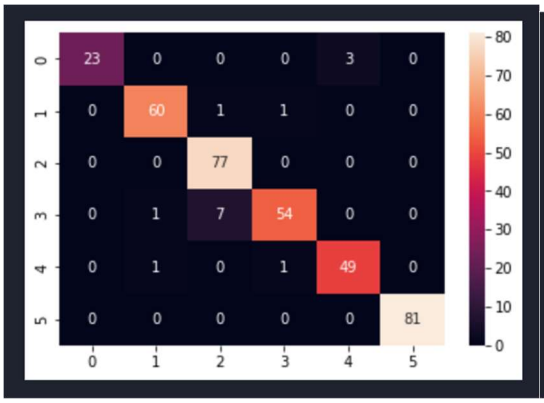
```
# Make pipeline
renn = RepeatedEditedNearestNeighbours()
smt = SMOTE(random_state=42)
enn = EditedNearestNeighbours()
pipeline = Pipeline([('smt', smt), ('enn', enn), ('renn', renn)])
```

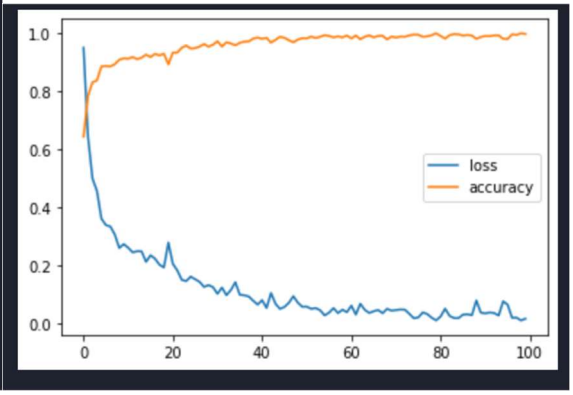
```
# apply function sampler
X_pdt, y_pdt = pipeline.fit_resample(X_res, y_res)
```

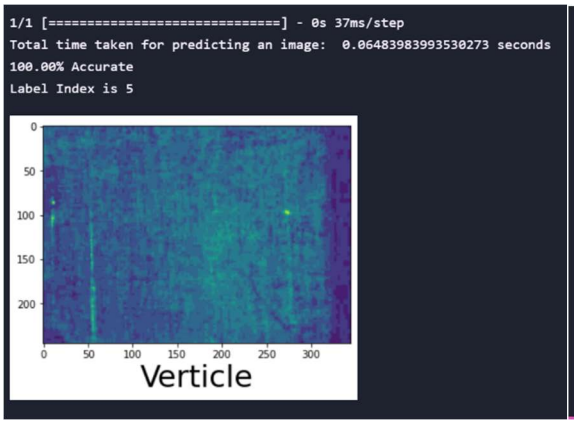
```
print(f'Resampled dataset shape {Counter(y_pdt)}')
```

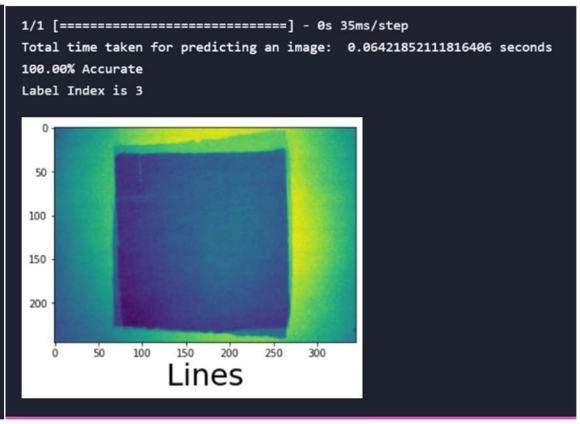
```
Resampled dataset shape Counter({3: 398, 5: 371, 2: 353, 1: 263, 4: 237, 0: 170})
```

```
12/12 [==============================] - 5s 428ms/step
              precision    recall  f1-score   support

           0       1.00      0.88      0.94        26
           1       0.97      0.97      0.97        62
           2       0.91      1.00      0.95        77
           3       0.96      0.87      0.92        62
           4       0.94      0.96      0.95        51
           5       1.00      1.00      1.00        81

    accuracy                           0.96       359
   macro avg       0.96      0.95      0.95       359
weighted avg       0.96      0.96      0.96       359
```


23 0 0 0 3 0
0 60 1 1 0 0
0 0 77 0 0 0
0 1 7 54 0 0
0 1 0 1 49 0
0 0 0 0 0 81
0 1 2 3 4 5
80 70 60 50 40 30 20 10 0


1.0 0.8 0.6 0.4 0.2 0.0
0 20 40 60 80 100
loss
accuracy


1/1 [==============================] - 0s 37ms/step
Total time taken for predicting an image: 0.06483983993530273 seconds
100.00% Accurate
Label Index is 5
0 50 100 150 200
0 50 100 150 200 250 300
Verticle


1/1 [==============================] - 0s 35ms/step
Total time taken for predicting an image: 0.06421852111816406 seconds
100.00% Accurate
Label Index is 3
0 50 100 150 200
0 50 100 150 200 250 300
Lines

Appendix 6: Miscellaneous

Appendix 6-1.1: Function Sampling Size 200x800

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

```
# define func for sampler
def func(X, y, sampling_strategy='majority', random_state=0):
  return RandomUnderSampler(
      sampling_strategy=sampling_strategy,
      random_state=random_state).fit_resample(X, y)
```

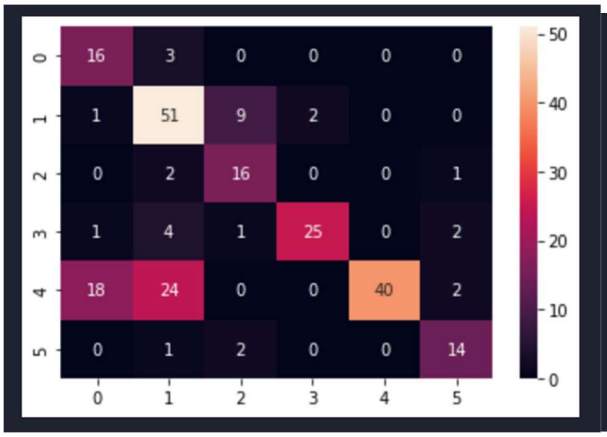
```
# apply function sampler
sampler = FunctionSampler(func=func)
X_fst, y_fst = sampler.fit_resample(X_res, y_res)
```

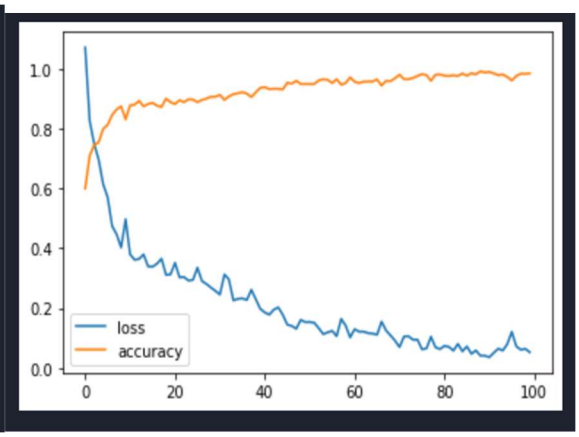
```
print(f'Resampled dataset shape {Counter(y_fst)}')
```

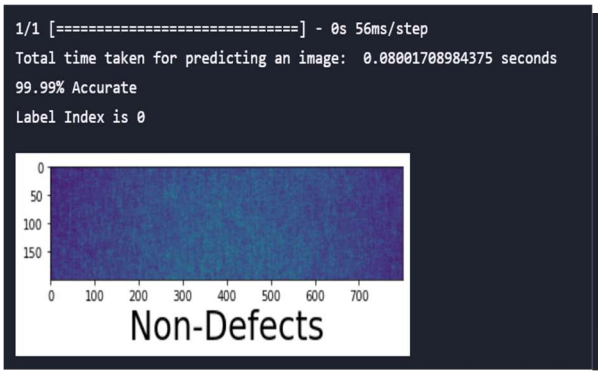
```
Resampled dataset shape Counter({4: 398, 1: 281, 3: 157, 2: 136, 0: 101, 5: 101})
```

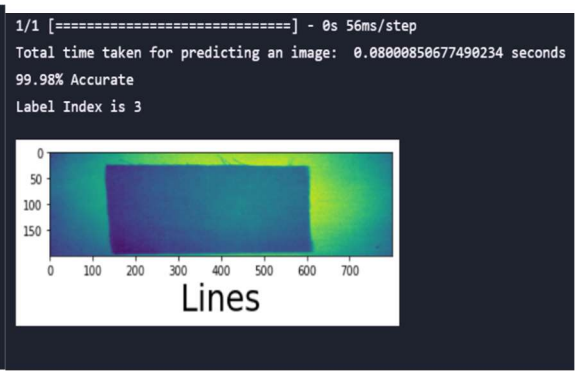
```
8/8 [==============================] - 6s 756ms/step
```

	precision	recall	f1-score	support
0	0.44	0.84	0.58	19
1	0.60	0.81	0.69	63
2	0.57	0.84	0.68	19
3	0.93	0.76	0.83	33
4	1.00	0.48	0.65	84
5	0.74	0.82	0.78	17
accuracy			0.69	235
macro avg	0.71	0.76	0.70	235
weighted avg	0.78	0.69	0.69	235

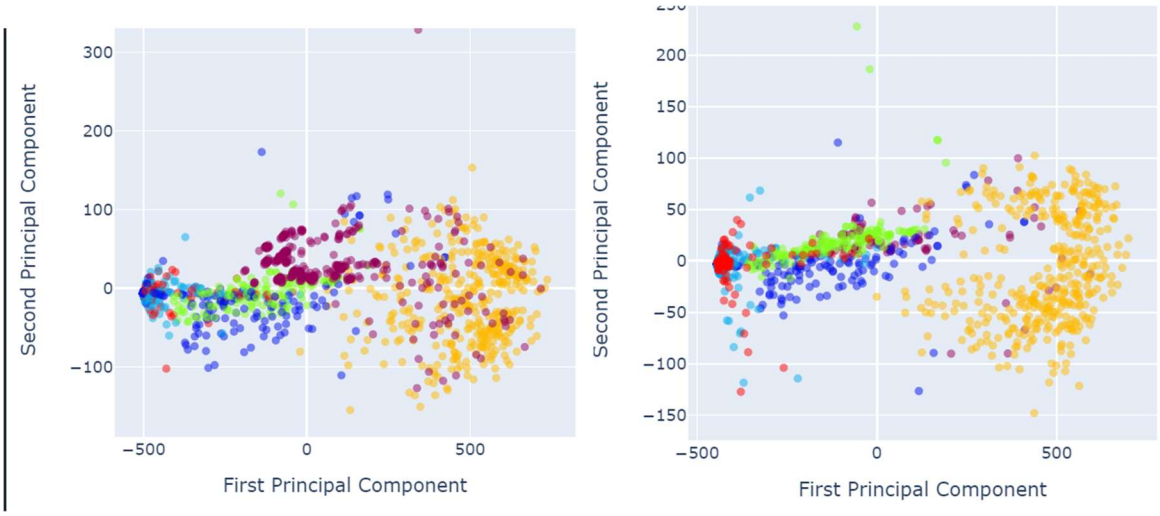




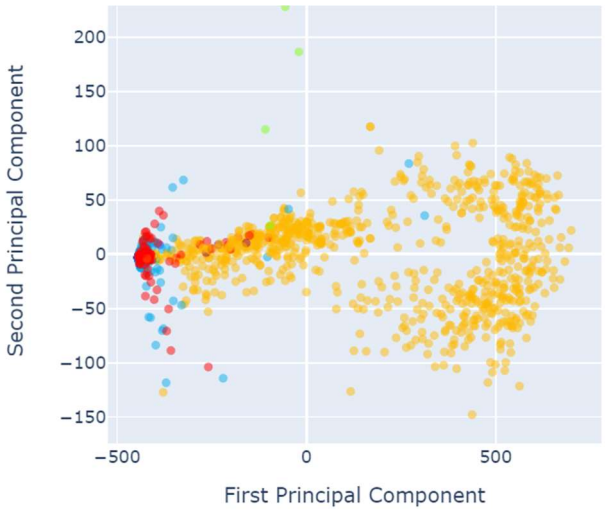




PCA Before and After Sampling



PCA After CNN



Appendix 6-1.2: Function Sampling Size 150x700

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

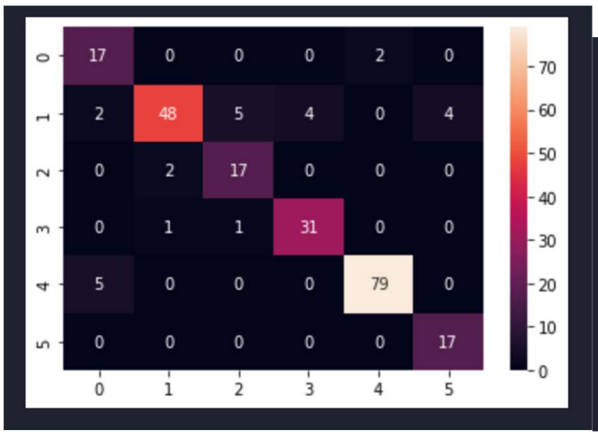
```
# define func for sampler
def func(X, y, sampling_strategy='majority', random_state=0):
    return RandomUnderSampler(
        sampling_strategy=sampling_strategy,
        random_state=random_state).fit_resample(X, y)
```

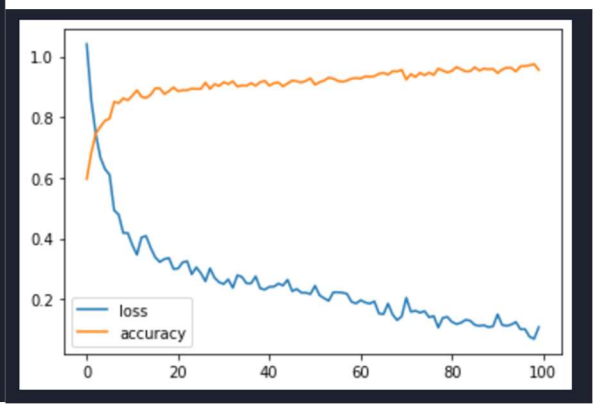
```
# apply function sampler
sampler = FunctionSampler(func=func)
X_fst, y_fst = sampler.fit_resample(X_res, y_res)
```

```
print(f'Resampled dataset shape {Counter(y_fst)}')
```

```
Resampled dataset shape Counter({4: 398, 1: 281, 3: 157, 2: 136, 0: 101, 5: 101})
```

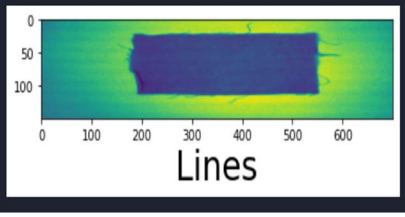
```
8/8 [==============================] - 4s 461ms/step
              precision    recall  f1-score   support

           0       0.71      0.89      0.79        19
           1       0.94      0.76      0.84        63
           2       0.74      0.89      0.81        19
           3       0.89      0.94      0.91        33
           4       0.98      0.94      0.96        84
           5       0.81      1.00      0.89        17

    accuracy                           0.89       235
   macro avg       0.84      0.91      0.87       235
weighted avg       0.90      0.89      0.89       235
```



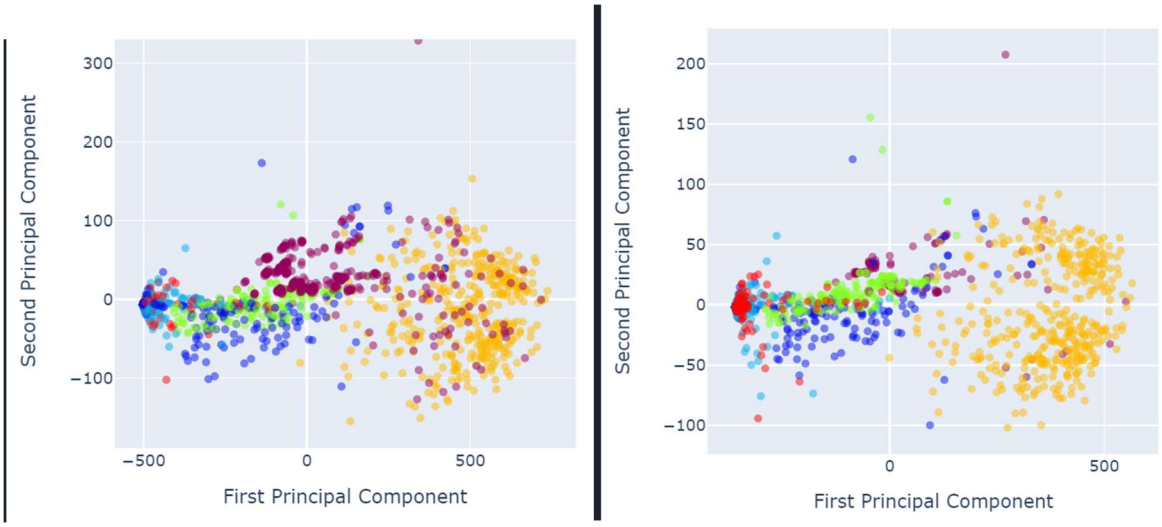



```
1/1 [==============================] - 0s 40ms/step
Total time taken for predicting an image:  0.06890058517456055 seconds
86.84% Accurate
Label Index is 0
```

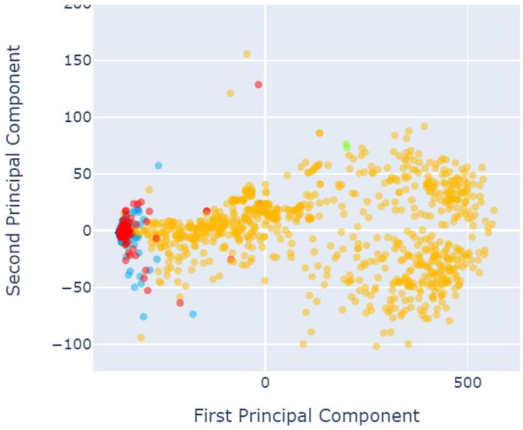

```
1/1 [==============================] - 0s 39ms/step
Total time taken for predicting an image:  0.0702207088470459 seconds
99.93% Accurate
Label Index is 3
```



PCA Before and After Sampling



PCA After CNN



Appendix 6-1.3: Function Sampling Size 245x345

```
Original dataset shape Counter({0: 417, 4: 398, 1: 281, 3: 157, 2: 136, 5: 101})
```

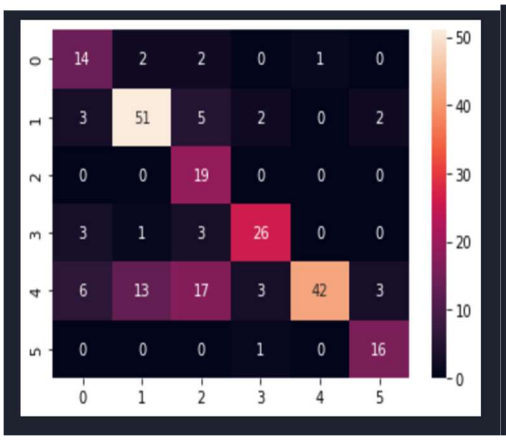
```
# apply function sampler
sampler = FunctionSampler(func=func)
X_fst, y_fst = sampler.fit_resample(X_res, y_res)
```
✓ 0.1s

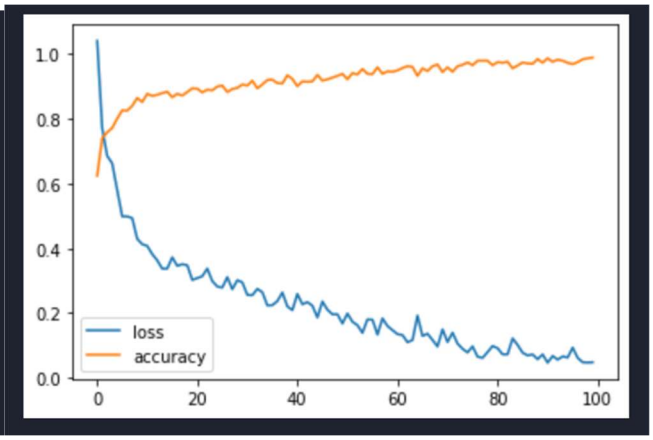
```
print(f'Resampled dataset shape {Counter(y_fst)}')
```
✓ 0.5s

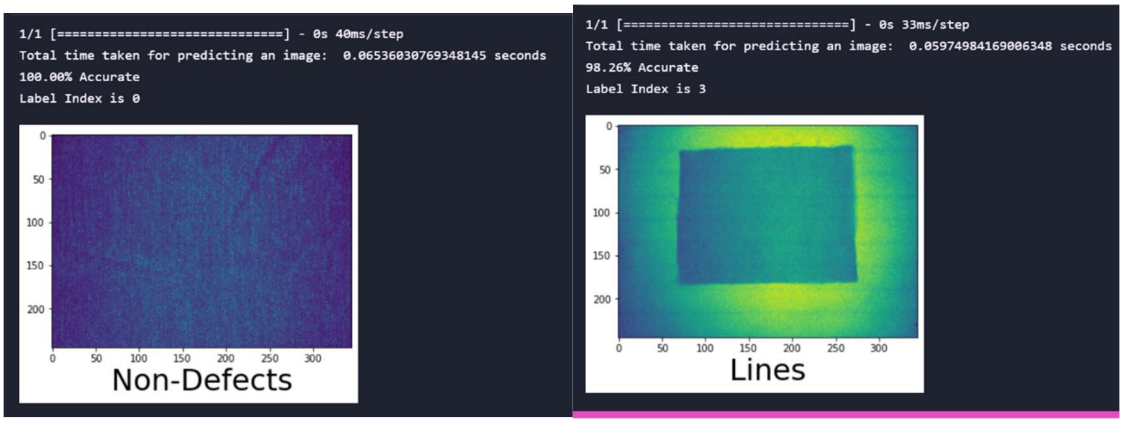
```
Resampled dataset shape Counter({4: 398, 1: 281, 3: 157, 2: 136, 0: 101, 5: 101})
```

```
8/8 [==============================] - 3s 384ms/step
```

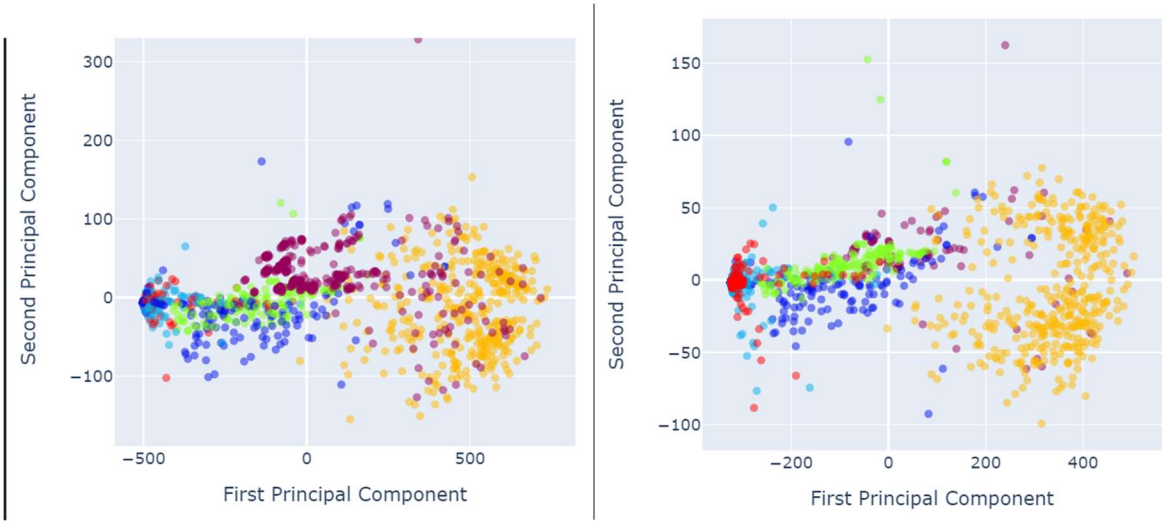
	precision	recall	f1-score	support
0	0.54	0.74	0.62	19
1	0.76	0.81	0.78	63
2	0.41	1.00	0.58	19
3	0.81	0.79	0.80	33
4	0.98	0.50	0.66	84
5	0.76	0.94	0.84	17
accuracy			0.71	235
macro avg	0.71	0.80	0.72	235
weighted avg	0.80	0.71	0.72	235



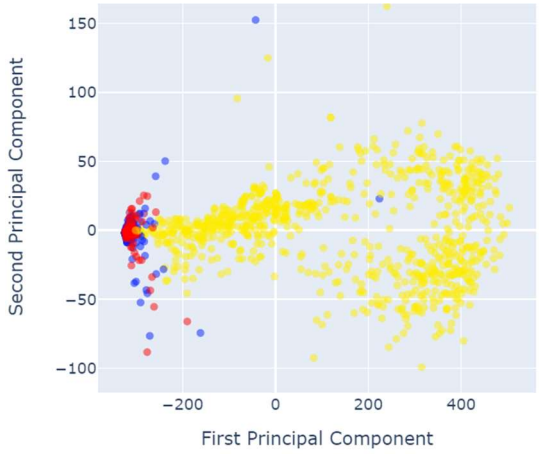




PCA Before and After Sampling



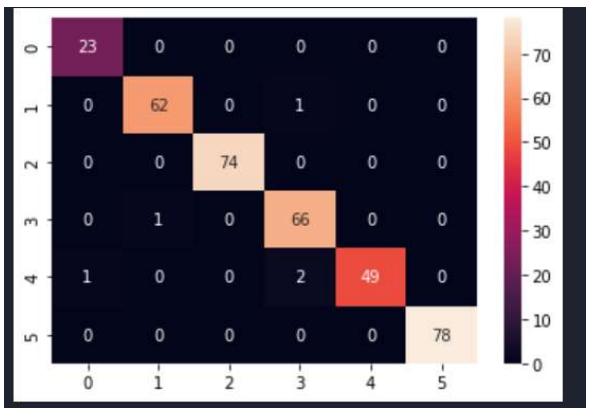
PCA After CNN

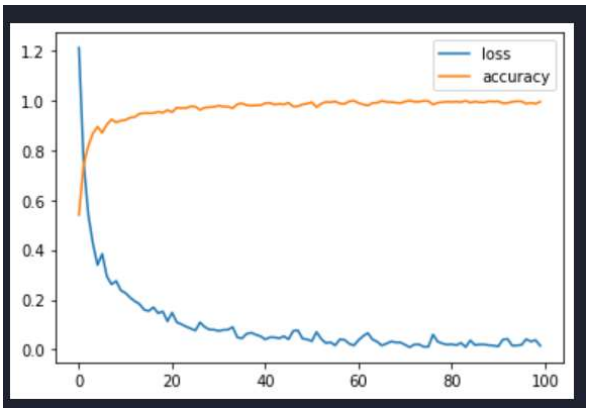


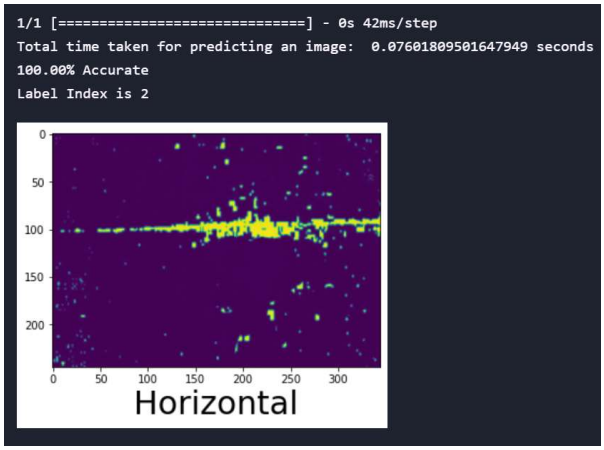
Appendix 7

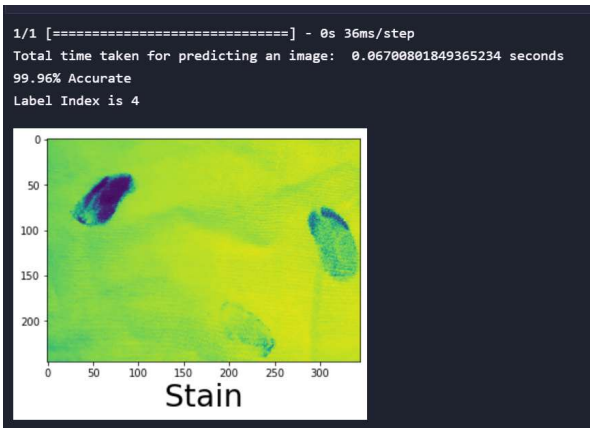
Appendix 7.1 CNN Model with SMOTEENN size 245x345

```
12/12 [==============================] - 6s 443ms/step
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        23
           1       0.98      0.98      0.98        63
           2       1.00      1.00      1.00        74
           3       0.96      0.99      0.97        67
           4       1.00      0.94      0.97        52
           5       1.00      1.00      1.00        78

    accuracy                           0.99       357
   macro avg       0.98      0.99      0.98       357
weighted avg       0.99      0.99      0.99       357
```

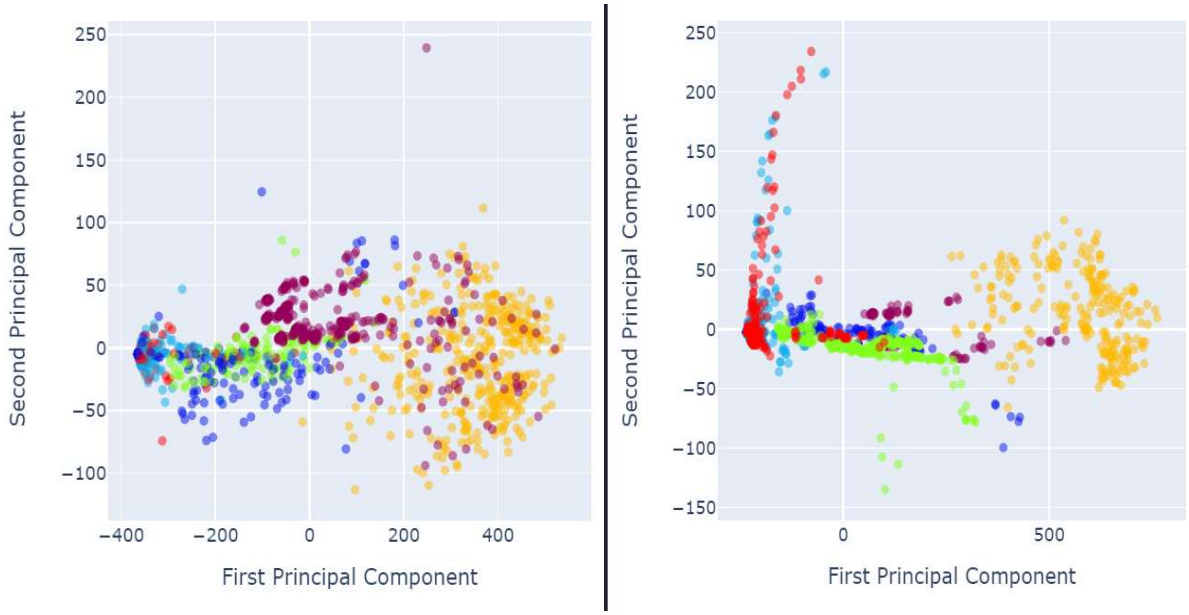






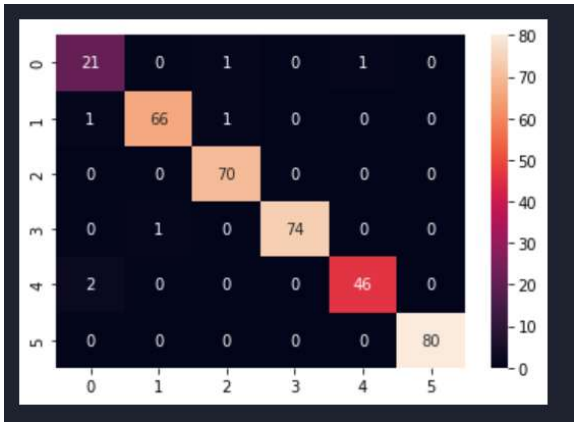


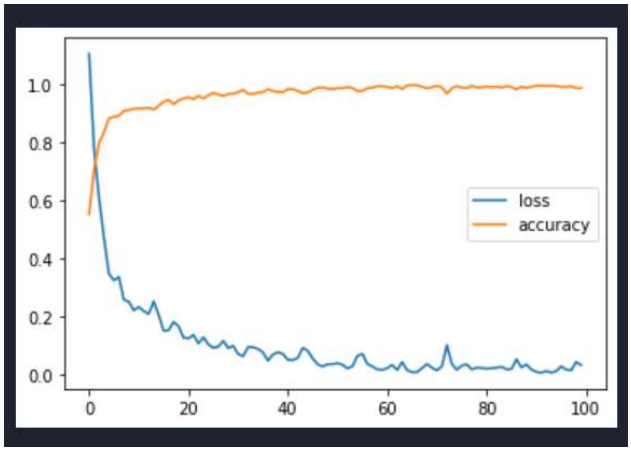
PCA Before and After Sampling

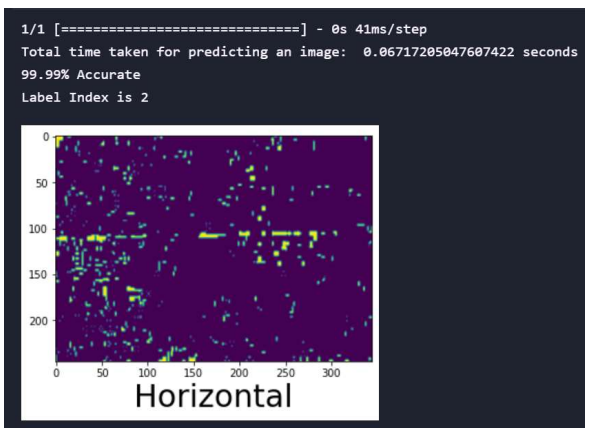


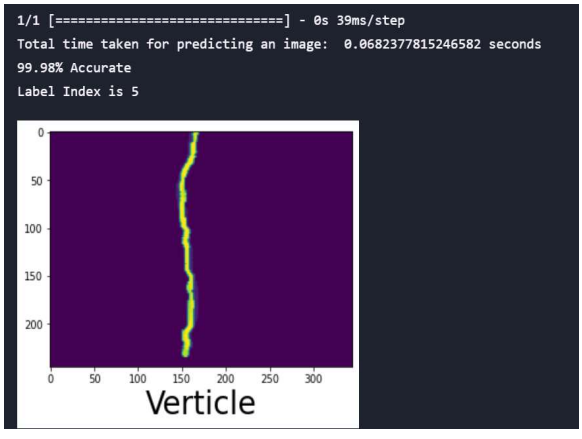
Appendix 7.2 CNN Model with SMOTEENN size 245x345

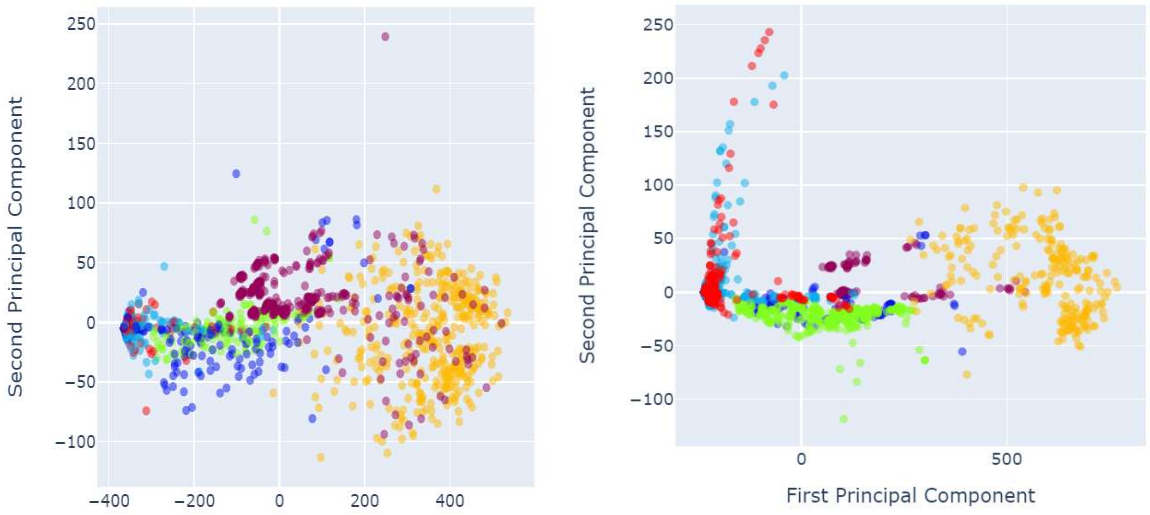
```
12/12 [==============================] - 6s 441ms/step
              precision    recall  f1-score   support

           0       0.88      0.91      0.89        23
           1       0.99      0.97      0.98        68
           2       0.97      1.00      0.99        70
           3       1.00      0.99      0.99        75
           4       0.98      0.96      0.97        48
           5       1.00      1.00      1.00        80

    accuracy                           0.98       364
   macro avg       0.97      0.97      0.97       364
weighted avg       0.98      0.98      0.98       364
```





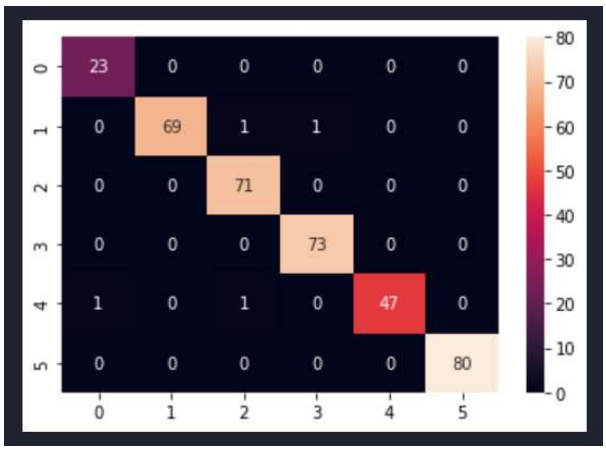


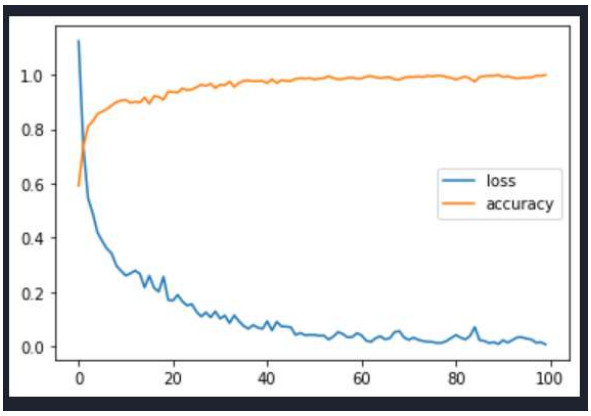


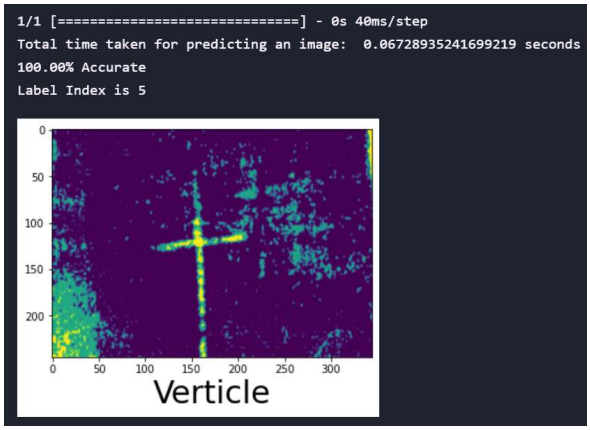


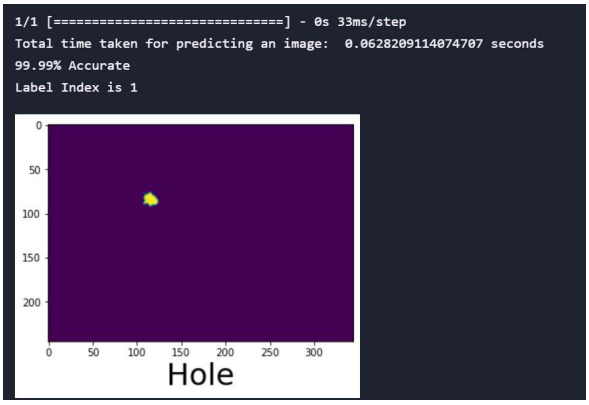
Appendix 7.3 CNN Model with SMOTEENN size 245x345

```
12/12 [==============================] - 6s 456ms/step
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        23
           1       1.00      0.97      0.99        71
           2       0.97      1.00      0.99        71
           3       0.99      1.00      0.99        73
           4       1.00      0.96      0.98        49
           5       1.00      1.00      1.00        80

    accuracy                           0.99       367
   macro avg       0.99      0.99      0.99       367
weighted avg       0.99      0.99      0.99       367
```

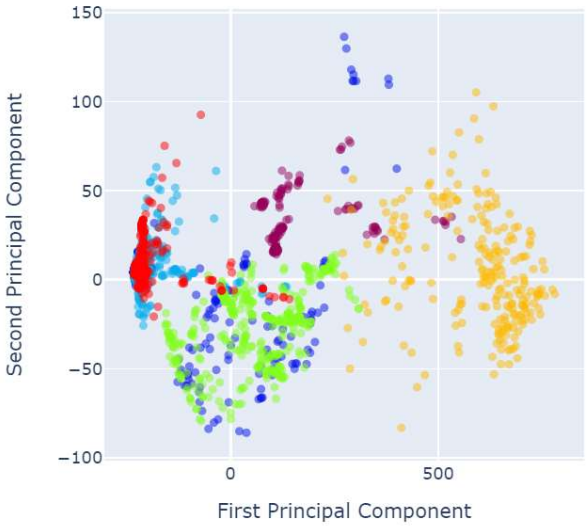






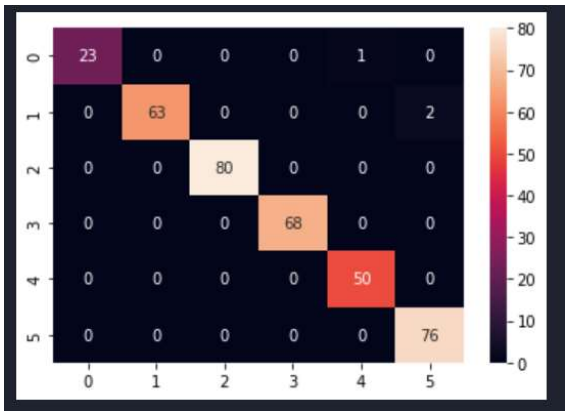


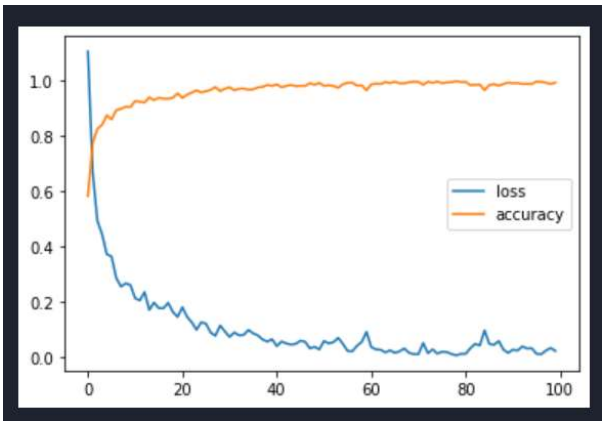
PCA Before and After Sampling

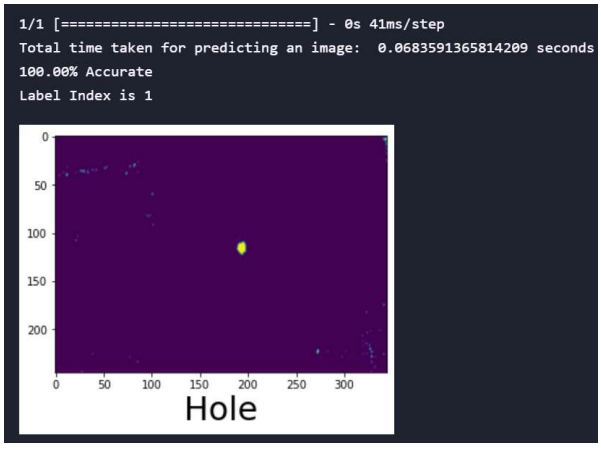


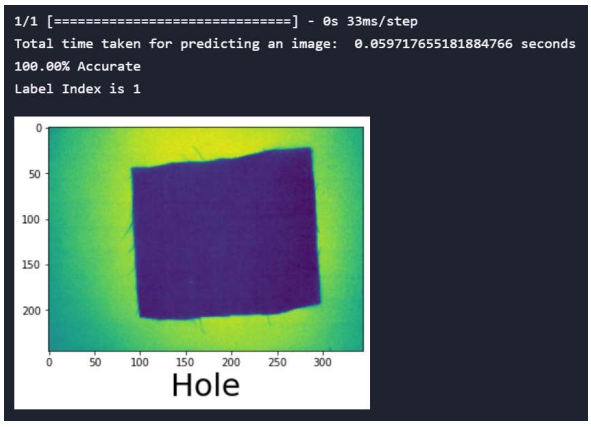
Appendix 7.4 CNN Model with SMOTEENN size 245x345

```
12/12 [==============================] - 6s 451ms/step
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        24
           1       1.00      0.97      0.98        65
           2       1.00      1.00      1.00        80
           3       1.00      1.00      1.00        68
           4       0.98      1.00      0.99        50
           5       0.97      1.00      0.99        76

    accuracy                           0.99       363
   macro avg       0.99      0.99      0.99       363
weighted avg       0.99      0.99      0.99       363
```

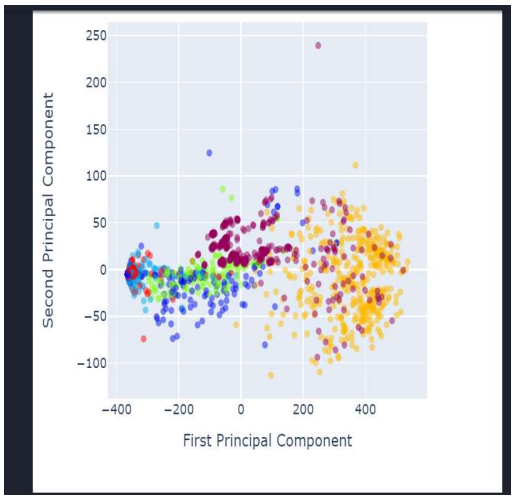


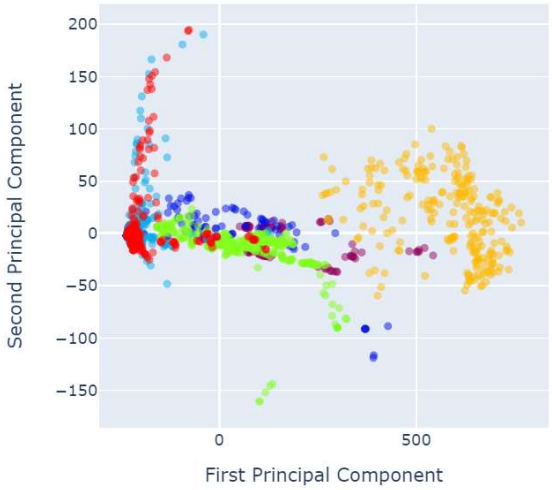






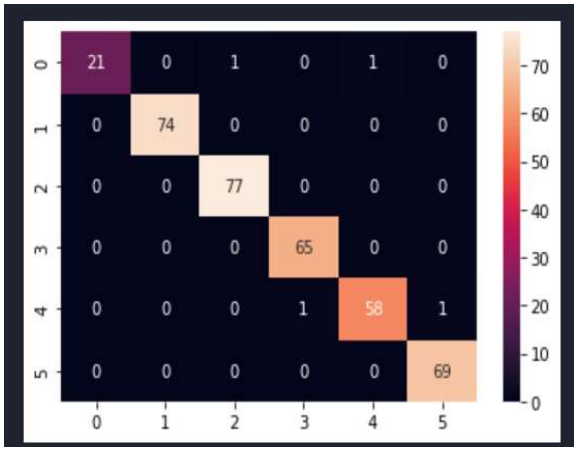
PCA Before and after Sampling

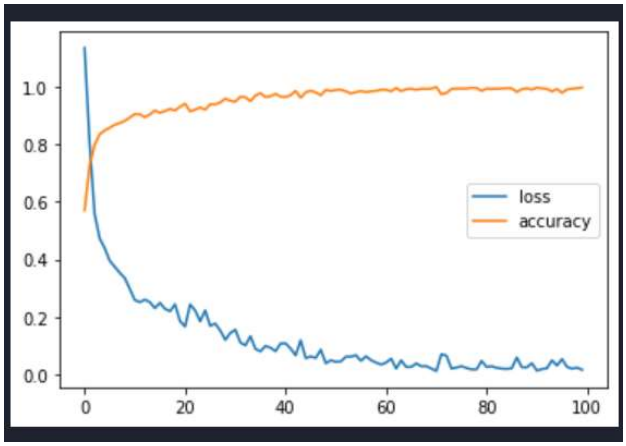


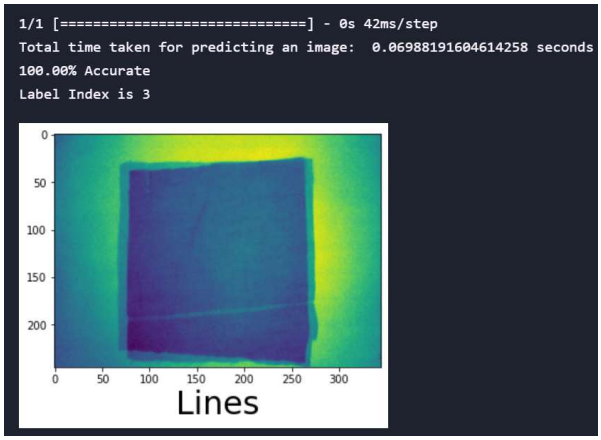


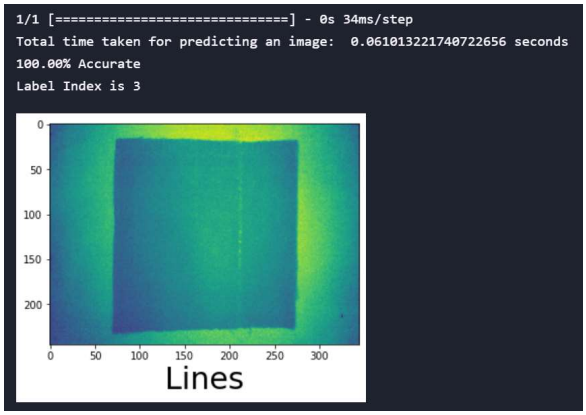
Appendix 7.5 CNN Model with SMOTEENN size 245x345

```
12/12 [==============================] - 6s 459ms/step
              precision    recall  f1-score   support

           0       1.00      0.91      0.95        23
           1       1.00      1.00      1.00        74
           2       0.99      1.00      0.99        77
           3       0.98      1.00      0.99        65
           4       0.98      0.97      0.97        60
           5       0.99      1.00      0.99        69

    accuracy                           0.99       368
   macro avg       0.99      0.98      0.98       368
weighted avg       0.99      0.99      0.99       368
```

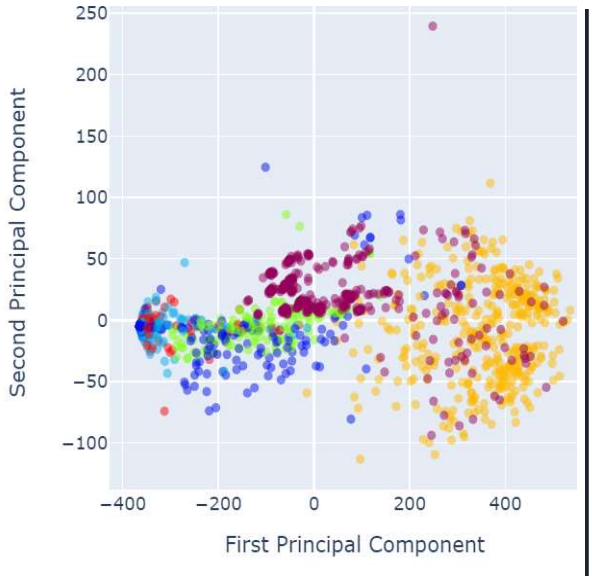


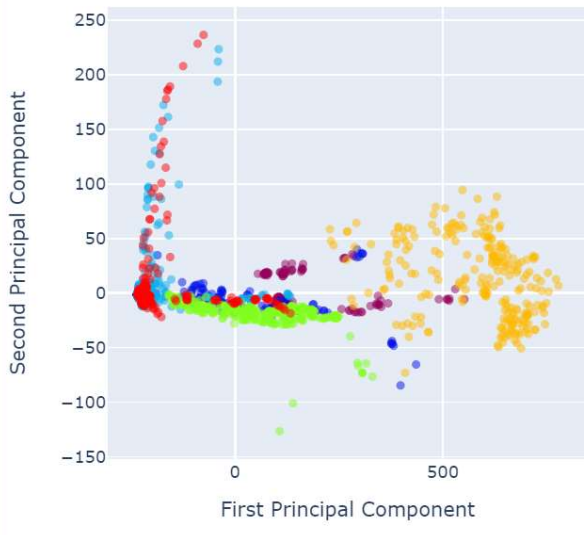






PCA Before and After Sampling





References

Agustianto, K. and Destarianto, P. (2019),'Imbalance Data Handling using Neighborhood Cleaning Rule (NCL) Sampling Method for Precision Student Modeling', *IEEE.* pp.86-89. doi:10.1109/ICOMITEE.2019.8921159

Almeida, T., Moutinho, F. and Matos-Carvalho, J., P. (2021) 'Fabric Defect Detection with Deep Learning and False Negative Reduction' *IEEE Access*, 2021, pp. 1-11. doi:10.1109/ACCESS.2021.3086028.

Brems, M. (2017), 'A One-Stop Shop for Principal Component Analysis', *Towards Data Science,* Available at: https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

Chawla, V., N., Bowyer, K., W., Hall, L., O. and Kegelmeyer, W., P. (2002), ' SMOTE: Synthetic Minority Oversampling Technique', *Journal of Artificial Intelligence Research,* 16(2), 321-357.

Chen, M. *et al.* (2022) 'Improved faster R-CNN for fabric defect detection based on Gabor filter with Genetic Algorithm optimization' *Computers in Industry*, 134, pp. 103551. doi: 10.1016/j.compind.2021.103551.

Duan, H., Wei, Y., Liu, P. and Yin, H. (2020), 'A Novel Ensemble Framewok Based on K-Means and Resampling for Imbalanced Data', *Applied Sciences,* 10, pp. 1-16. doi:10.3390/app10051684

Gowda. K., C. and Krishna, G. (1979), 'The Condensed Nearest Neighbor rule using the concept of mutual nearest neighbor', IEEE Transaction Theory, 25 (4), pp.488-490. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1056066

Han, H., Wang, W-Y. and Mao, B-H. (2005),'Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning', *ICIC* , 878-887, doi:10.1007/11538059_91

He, H., Bai, Y., Garcia E. A. and Li, S. (2008),' ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning', *International Joint Conference on Neural Networks,*pp. 1322-1328, Available at:https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4633969

He, Y., Zhang, H., D., Huang, X., Y. and Tay, F., E., H. (2021), 'Fabric Defect Detection based on Improved Faster RCNN' *International Journal of Artificial Intelligence& Applications*, 12(4), pp.23–32. doi: 10.5121/ijaia.2021.12402.

Howarth, J. P., Zakharov, D. N., Megret, R. and Stach, E. A. (2020), 'Understanding important features of deep learning models for segmentation of high-resolution transmission electron microscopy images', *npj Computational Materials,* (2020)6:108, 9. doi: https://doi.org/10.1038/s41524-020-00363-x

Imbalanced Learn (n.d.a), 'SVMSMOTE'. Available at: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SVMSMOTE.html

Imbalanced Learn (n.d.b),'ClusterCentroids'. Available at: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.ClusterCentroids.html

Imbalanced Learn (n.d.c),'EditedNearestNeighbor'. Available at: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.EditedNearestNeighbours.html

Imbalanced Learn (n.d. d), 'RepeatedEditedNearestNeighbors'. Available at: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RepeatedEditedNearestNeighbours.html

Imbalanced Learn (n.d.e),'AIIKNN'. Available at:

https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.AllKNN.html

Imbalanced Learn (n.d.,f), 'Under-sampling'. Available at: https://imbalanced-learn.org/stable/under_sampling.html#edited-nearest-neighbors

Imbalanced Learn (n.d.h)', Combination of Over- and Under-Sampling'. Available at: https://imbalanced-learn.org/stable/combine.html

Imbalanced Learn (n.d.i)', 'RandomOverSampler'. Available at: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.RandomOverSampler.html

Jing, J., Wang, Z., Rätsch, M. and Zhang, H. (2022), 'Mobile-Unet: An efficient convolutional neural network for fabric defect detection', *Textile Research Journal*, 92(1–2), pp. 30–42, doi: 10.1177/0040517520928604.

kmeans-smote.readthedocs.io (no date),'kmeans_smote module'. Available at: https://kmeans-smote.readthedocs.io/en/latest

Letteri, I., Cecco, A.D., Dyoub, A. and Penna, G. D. (2020), 'A Novel Resampling Technique for Imbalanced Dataset Optimization', *Researchagte,* pp. 1-23. Available at: https://www.researchgate.net/publication/348078650_A_Novel_Resampling_Technique_for_Imbalanced_Dataset_Optimization

Li, C., Li, J., J., Li, Y., He, L., Fu, X. and Chen, J. (2021), 'Fabric Defect Detection in Textile Manufacturing: A Survey of the State of the Art', *Security and Communication Networks*, 2021, pp. 1-13. doi: 10.1155/2021/9948808.

Liu, Z., Zhang, C., Li, C., Ding, S., Dong, Y. and Huang, Y. (2019), 'Fabric defect recognition using optimized neural networks', *Journal of Engineered Fibers and Fabrics*, 14, pp. 1-10. doi: 10.1177/1558925019897396.

Liu, Z., Liu, S., Li, C., Ding, S. and Dong, Y. (2018), 'Fabric defects detection based on SSD', *ACM International Conference Proceeding Series*, pp. 74–78. doi: 10.1145/3282286.3282300.

Luke, J. J., Joseph, R. and Balaji, M. (209),'Impact of image size on accuracy and generalization of convolutional neural networks', International Jouranl of Research and Analytical Reviews *Ijrar,* 6(1), pp.70-80, Available at: https://www.researchgate.net/publication/332241609.

Madhukar, B. (2020),'Using Near-Miss Algorithm For Imbalanced Datasets', *Developers Corner.* Available at: https://analyticsindiamag.com/using-near-miss-algorithm-for-imbalanced-datasets/

Minhas, M., S. and Zelek, J. (2020), 'Defect detection using deep learning from minimal annotations', *VISIGRAPP 2020 - Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 4, pp. 506–513. doi: 10.5220/0009168005060513

Pahren, L., Thomas, P., Jia, X. and Lee, J. (2022),'A Novel Method in Intelligent Synthetic Data Creation for Machine Learning-based Manufacturing Quality Control', *IFAC PapersOnline,* 55-19(2022), pp. 73-78. DOI:10.1016/j.ifacol.2022.09.186.

Peng, P., Wang, Y., Hao, C., Zhu, Z., Liu, T. and Zhou, W. (2020) 'Automatic fabric defect detection method using pran-net' *Applied Sciences (Switzerland)*, 10 (23), pp. 1–13. doi: 10.3390/app10238434.

Pathirana, P. (2020), 'Fabric stain dataset', Kaggle, Available at: https://www.kaggle.com/datasets/priemshpathirana/fabric-stain-dataset

Ranathunga, S. (2020), 'Fabric defect dataset', Kaggle, Available at: https://www.kaggle.com/datasets/rmshashi/fabric-defect-dataset

Ringner, M. (2008)'what is principal component analysis',*Nature Biotechnology,*26(3), pp.303-304, Available at: http://146.6.100.192/users/BCH339N_2018/NBT_primer_PCA.pdf

Rukundo, O. (2021), ' Effects of image size on Deep Learning', *Arxiv,* pp.19, doi:10.48550/arXiv.2101.11508

Silvestre-Blanes, J., Albero-Albero, T., Miralles, I., Pérez-Llorens, R. and Moreno, J. (2019), ' Aitex Fabric Image Database', Aitex, Available at: https://www.aitex.es/afid/

Sabottke, C., E. and Spieler, B. M. (2019) 'The Effect of Image Resolution on Deep Learning in Radiography', *Radiology: Artificial Intelligence,* 2(1), pp.1-7, Available at: https://doi.org/10.1148/ryai.2019190015

Sisters, Li. (2020),'Instance Hardness Threshold: An Undersampling Method to Tackle Imbalanced Classification Problems', *Towards Data Science.* Available at: https://towardsdatascience.com/instance-hardness-threshold-an-undersampling-method-to-tackle-imbalanced-classification-problems 6d80f91f0581#:~:text=Instance%20Hardness%20Threshold%20(IHT)%20is,be%20removed%20from%20the%20dataset.

Thambawita, V., Strumke, I., Hicks, S. A., Halvorsen, P., Parasa, S. and Riegler, M. A. (2021), 'Impact of Image Resolution on Deep Learning Performance in Endoscopy Image Classification: An Experimental Study Using a Large Dataset of Endoscopic Images' , *National Library of Medicine*,11(12):2183, doi: 10.3390/diagnostics11122183

Vladimir, P. (2020),'Handling imbalanced dataset in image classification', *Analytics Vidhya.* Available at: Handling imbalanced dataset in image classification | by Privalov Vladimir | Analytics Vidhya | Medium

Viadinugroho, R. A.A. (2021),'Imbalanced Classification in Python: SMOTE-Tomek Links Method', *TowardsData Science,* Available at: https://towardsdatascience.com/imbalanced-classification-in-python-smote-tomek-links-method-6e48dfe69bbc

Wang, Z. and Jing, J. (2020) 'Pixel-Wise Fabric Defect Detection by CNNs without Labeled Training Data', *IEEE Access*, 8, pp. 161317–161325. doi: 10.1109/ACCESS.2020.3021189.

Wei, B., Hao, K., Tang, X., S. and Ren, L. (2019) 'Fabric defect detection based on faster RCNN' in *Advances in Intelligent Systems and Computing*, 849, pp. 45–51. doi: 10.1007/978-3-319-99695-0_6.

Wilson, D., L. (1972),'Asymptotic Properties of Nearest Neighbor Rules Using Edited Data', *IEEE Transactions of Systems, Man and Cybernetics,*2 (3), 408-421. DOI: 10.1109/TSMC.1972.4309137

Yamashita, R., Nishio, M., Do, R., K., G. and Togasi, K. (2018), 'Convolutional neural networks: an overview and application in radiology', *SpringerOpen,* Available at: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

Zhao, Y., Hao, K., He, H., Tang, X. and Wei, B. (2019) 'A visual long-short-term memory based integrated CNN model for fabric defect image classification' *Neurocomputing*, 380, pp. 259–270. doi: 10.1016/j.neucom.2019.10.067.

Zhao,S., Yin, L., Zhang, J., Wang, J. and Zhong, R. (2020) 'Real-time fabric defect detection based on multi-scale convolutional neural network' *IET Collaborative Intelligent Manufacturing*, 2(4), pp. 189–196. doi:10.1049/IET-CIM.2020.0062.

Zhang, H., Zhang, L., Li, P. and Gu, D. (2018)'Yarn-dyed Fabric Defect Detection with YOLOV2 Based on Deep Convolution Neural Networks', *IEEE 7th Data Driven Control and Learning Systems Conference,* pp. 170-174. doi**:** 10.1109/DDCLS.2018.8516094