A CLASSIFICATION MODEL UTILIZING FACIAL LANDMARK TRACKING TO DETERMINE SENTENCE TYPES FOR AMERICAN SIGN LANGUAGE RECOGNITION

A Thesis Presented to
The Faculty of Department of Electrical and Computer Engineering
California State University, Los Angeles

In Partial Fulfillment
Of the Requirements for the
Master of Science
In
Electrical and Computer Engineering

By

Janice Trinh Nguyen

August 2023

The thesis of Janice Nguyen is approved.

Y. Curtis Wang, Committee Director
Deborah Won, Committee Member
Charles Liu, Committee Member
Charles Liu, Department Chair

California State University, Los Angeles

ABSTRACT
A Classification Model Utilizing Facial Landmark Tracking to Determine Sentence Types for American Sign Language Recognition
By
Janice Nguyen

Real time ASL language interpreting is lacking and requires additional work in order to be achieved. To implement an effective real time interpreting model, the linguistic structure and rules of ASL must be taken into account. In general, ASL has 5 parameters (hand shape, palm orientation, facial expression, movement, and location relative to the body) that dictate sign meaning and sentence structure. A majority of ASL interpreting models are hand shape (gesture) based and lack the integration of facial cues, which are crucial in ASL to convey tone and distinguish sentence types.

Thus, the integration of facial cues in computer vision based ASL interpreting models has the potential to improve performance and reliability. With this in mind, we propose a new sentence classification model based on facial expressions that will supplement current ASL interpreting models that are hand gesture based. We implement the random forest tree classifier and support vector machine algorithms that are trained on facial angles to classify videos of ASL signers signing complete sentences. These facial angles were calculated using simple trigonometric calculations on facial points derived from the dlib library. This model is created to work in conjunction with current hand gesture based ASL models to ensure real time ASL interpreting.

Future directions for this project include designing the model so that it can interpret more sentence types and potentially interpret multiple people in one frame. With the improvement of these models, advance technology such as smart devices can be more accessible to hearing-impaired individuals and help improve building sustainability by improving accessibility of control of the building's HVAC systems for hearing impaired individuals.

ACKNOWLEDGMENTS

I would like to give my sincerest gratitude to the multitude of people who have made my master's degree an incredible journey and helped me become the engineer I am today.

First, I'd like to thank the CREST Center for Energy and Sustainability and the CREST Center for Advancement Towards Sustainable Urban Systems for funding this project through National Science Foundation awards HRD-1547723 and HRD-2112554, respectively.

Next, I would like to give immense gratitude to the very first professor I met at CSULA and the reason I was able to meet so many people here who have made such an impact on my education. Thank you so much Dr. Won for welcoming me into BE WINNORS and the Biomedical Engineering community at CSULA. From showing me constant support since our very first meeting to pushing me to be a better engineer, I would never have been able to find a supportive community at CSULA without your help.

To the computer vision team from BE WINNORS (Abby, Maria, and Chris), thank you for the constant support. From the late night codings to the random memes, I cherished every moment working with you all in BE WINNORS and all the wonderful moments we shared after the program was done. Thank you for always constantly cheering me on even when I was getting too distracted with Sesame Street games.

I would also like to give my thanks to Dr. Liu, Dr. Zhao and the DAD (Dat, Anthony, and Daniel) TAs for welcoming me into the computer engineering crew as an honorary MOM TA and giving me a chance to inspire the next generation of future engineers. From leading a group of students to creating educational videos, I greatly enjoyed having the opportunity to teach and learn more about the importance of computer architectures and other engineering topics. Thank you for giving me the opportunity to teach others the joys of engineering and sharing in the struggles of how to use and spell panopto.

Furthermore, I'd like to thank Dr. Shen and my mechanical engineering boys (Carlito, Luis, and Tim) who accompanied me to UW-Madison. Thank you for showing me true teamwork. From fighting off bugs (both in our code and the scary cockroaches) to exploring

Wisconsin together, I am immensely grateful for everyone's constant offer of help even outside of lab. Thank you for showing me what it's like to be part of a team and for always offering a helping hand.

And lastly, I'd like to give my eternal gratitude to my advisor, Dr. Wang. Thank you for doing whatever you could to help me achieve my goals and for being a huge part of my journey. From helping me get my first publication to sharing in the joys of my late-night messages about my Phd acceptances, I am eternally grateful for your constant support and nagging as I reach each of these milestones. Everything you have done for me has pushed me to become a better engineer including you nitpicking my code and your extremely difficult assignments. Yearncheesy your homework ain't easy, but those assignments and having you as an advisor sure did make me a better engineer and I can't wait to make you proud.

A small acknowledgement section can never convey the immense gratitude I have for all the people I have thanked and for the many others who had impacted my time here at CSULA. I hope I can make you all proud one day. Thank you again for all the wonderful memories.

DEDICATION

To SG:

Thank you for being there even when you weren't.

To Lucky:

Thank you for being the light of my life for 15 years.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The task of creating real time ASL interpreting models have been proven to be quite difficult due to the dynamic nature of ASL. A majority of past ASL interpreting models have been primarily based on hand gestures and work by tracking the joints of the user's hands [28], [27], [32]. However, ASL syntax also considers the user's facial expressions as a way of determining meaning. While these models are robust enough to interpret individual signs, there is lacking performance when it comes to more complex structures such as sentences. We propose a new way to quantify facial expressions and a new model to preclassify ASL sentences to be used in conjunction with these current ASL vocabulary interpreting models. We aim to ensure that the classification model is computational efficient so it can be used on embedded devices and smart devices that have less computational power than a typical desktop.

For our project, we propose utilizing the random forest tree classifier and the support vector machine (SVM) classifier models to build the preclassification model. These models would be trained on facial angles derived using simple trigonometric calculations. Furthermore, to minimize the amount of computational power needed to run this model, we implement principal component analysis (PCA) to reduce dimensionality in the number of features used to train the models.

1.1 Background

American Sign Language (ASL) is utilized by the deaf community as its primary mode of communication. However, deaf and hard of hearing individuals are often at risk for receiving inadequate treatment especially in healthcare due to communication barriers in a hearing centric society. Even though federal laws mandate hospitals to provide interpreters, a majority of hospitals opt to use web-based video remote interpreting (VRI), which causes problems with communication between hearing physicians and hard of hearing patients due to issues such as poor video call quality, and lack of space in more crowded hospitals for the patient to sign and see the interpreter [12]. The poor quality of VRI services can be problematic, especially in emergencies where the patient needs immediate care. Thus, ongoing research in accurate, real time ASL interpreting models is vital, however progress towards it is lacking.

1.2 ASL Syntax

ASL has its own unique linguistic structure and rules that dictate the meaning of individual signs and how to construct certain sentences. For instances, individual sign meaning and sentence formation is determined by five parameters: hand shape, palm orientation, facial expression, movement, and location relative to the body [3]. Letters and individual words tend to relay more on the first 4 parameters to distinguish its meaning. For example, the sign for "key" and the sign for "apple" utilizes the same hand shape, but the differentiating factor is the placement of the hand relative to the face and body of the signer. On the other hand, facial expressions play a crucial role in distinguishing the different types of sentences, especially in the case of asking a question where raising and lowering the eyebrows indicate different types of questions. There are 8 sentence types present in the ASL language: assertion, negation, rhetorical/statement, topic, What/Where/When/Why/How question, and Yes/No question [19].

1.3 Current ASL Models based on Hand Gesture

A majority of current ASL interpreting models are essentially based on a signer's hand shape. These models perform ideally at isolating sign meanings, but this often leads to a limited set of signs that the model can interpret ([34], [30]). Furthermore, these models perform poorly on more complex structures such as sentences and questions because the model is aimed towards interpreting individual signs instead of interpreting the meaning of entire sentences or questions. In cases where sentences have the same hand gestures but different meanings, the only way to differentiate these sentences is by facial cues. For example, "Are you hungry?" and "You are hungry" have the same hand gestures done in the same sequence and the only way to differentiate the two sentences is by raising eyebrows to indicate a yes or no question [19]. Thus the lack of integrating facial expression recognition can cause poor performance in more complex structures.

1.4 Other Works in Facial Expression in ASL Interpreting Models

Other ASL interpreting models based on facial expressions suggest there is potential in improving real time ASL interpreting integrating facial expressions, but the work in this field is limited due to the difficult task of tracking the dynamics of the subject's constantly changing facial expressions and the potential loss of information from obstruction of the signer's hands. Volger et al. addresses this problem by developing a 3D deformable model tracking system for the purpose of tracking faces of signers while they are signing in a video [33]. However, this model requires time-consuming preprocessing of data before a model can be created. Another approach is to determine points on the subjects' face and classify types of ASL signs based on that. Nguyen et al. developed a complex algorithm where 21 facial points of a subject's face was determined and tracked by probabilistic principal

component analysis and the results of the tracking were classified using a system based on Hidden Markov Models and another one based on a support vector machine [19]. While this model performs well, it is computationally expensive and time consuming to train and run due to its complex calculations.

1.5 Random Forest Classification Tree Model

The random forest classification tree model is a popular machine learning model that has been used for various classification applications. The model works by creating a forest of "decision trees" that act like an ensemble. Each tree will vote on what to classify the sample and the majority vote will be the final classification of the sample [4]. Each tree will have a set number of nodes and each node will have a set number of branches or choices that is determined by the hyperparameters of the model. At every node, the features that are taken into account are randomly chosen from a subset of features, a technique known as bagging. This is done because each tree is sensitive to the training data, and thus bagging is done to lower the variance from the sensitivity of each tree [4]. Furthermore, this ensures that each tree has low correlation with each other which will reduce the chance of overfitting the model to the training data set. The tree will continue to create new nodes until a decision is reached or until a max depth is achieved if the tree is limited to a max depth.

1.6 Support Vector Machine

The support vector machine (SVM) model is another popular machine learning model commonly used for classification purposes. The SVM model works by separating the patterns and observations between classes. It does this by finding the hyperplane that will separate the features of an N dimensional space where N is the number of features and maximize the distance between the hyperplane and the classes. This model works really well in cases where the number of features is significantly larger than the number of sample cases [23]. SVM

maximizes the hyperplane by minimizing the cost or in this case, the hinge loss function, which indicates how well the hyperplane differentiates the features present in the classes in the dataset.

1.7 Principal Component Analysis (PCA)

A key goal in training and testing these models is to keep run time low and utilize the least amount of computational power. One way to achieve this is to implement principal component analysis (PCA) to reduce redundant data. PCA is an mathematical technique used to reduce the dimensionality in data sets to a smaller number of dimensions called principal components (PC). In other words, PCA will transform the current data set to a new coordinate system that will have less variation while maintaining the most amount of information that can be extract from the data set. This new coordinate system's axes are the calculated PC. The number of PCs calculate is determined based on how well the model performs [31].

1.8 General Proposed Pipeline

The proposed pipeline is shown in the block diagram in Fig. 1.1. The pipeline is essentially split into 3 parts: facial feature extraction, classification model pipeline, and testing. During the facial feature extraction, frames from the training videos are extracted and undergo data augmentation. Then, we utilized a Python library to detect and extract key facial points that will be converted into angles and used for the classification model pipeline. The classification model pipeline includes reducing the dimensionality into PCs using PCA before using the newly calculated PCs to train the models. In this project, we used the random forest tree classifier and the SVM to classify the videos based on facial angles. Lastly, the model is tested using the remaining videos and its performance is evaluated according to its accuracy, confusion matrix, and execution time.

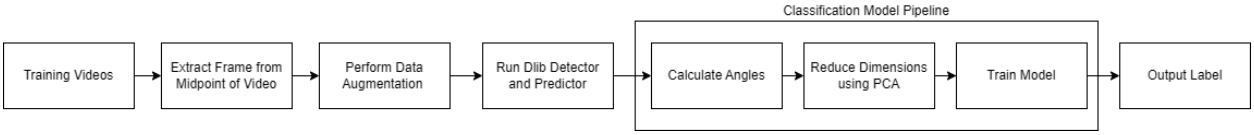

Fig. 1.1 – Training workflow for the proposed classification model pipeline.

CHAPTER 2

Facial Feature Extraction

2.1 Introduction

The goal of this project is to determine a computationally efficient way to preclassify sentences based on facial expressions. To do so, we need to figure out a way to quantify such facial cues so that it can be used to train a classification model. Other ASL interpreting models based on facial expressions have proposed different ways to quantify the facial clues present in ASL, but the proposed algorithm is often computationally expensive due to the difficult task of tracking the dynamics of the subject's constantly changing facial expressions and the potential loss of information from obstruction of the signer's hands. Volger et al. addresses this problem by developing a 3D deformable model tracking system for the purpose of tracking faces of signers while they are signing in a video [33]. However, this model requires time-consuming preprocessing of data before a model can be created. Another approach is to determine points on the subjects' face and classify types of ASL signs based on that. Nguyen et al. developed a complex algorithm where 21 facial points of a subject's face was determined and tracked by probabilistic principal component analysis and the results of the tracking were classified using a system based on Hidden Markov Models and another one based on a support vector machine [19]. While this model performs well, it is computationally

expensive and time consuming to train and run due to its complex calculations. We propose a computationally efficient facial landmark tracking approach using the Dlib library.

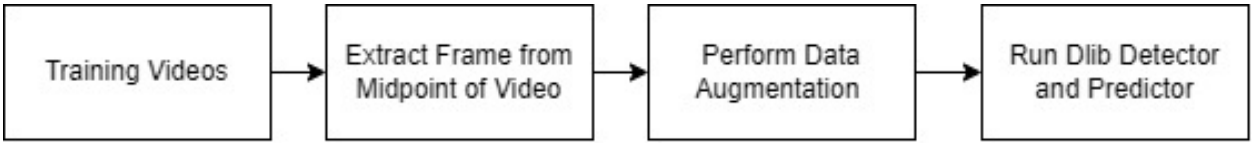

Fig. 2.1 – Training workflow for pre-processing

2.2 Data Set

Fig. 2.2 – Example of 6 Signers from How2Sign Dataset [7].

For this project, we need to find an appropriate data set that has videos of signers that are signing complete sentences. One of the main reasons for the delay in the progression towards integrating facial clues into ASL interpreting models is that a majority of publicly available data sets for ASL consist of mostly the alphabet and individual words. The dataset we ended up using for this project was the How2Sign data set [7], since it is one of the few publicly available data sets where the user is signing complete sentences. We used a total of 173 videos from this dataset of subjects signing sentences in ASL, where 121 videos were used for training and 52 videos were used for testing. There was a total of 6 signers that

Fig. 2.3 – Example of Data Augmented Images (Rotated) on Image from How2Sign Dataset [7].

were native or professional ASL users and were a mixture of deaf, hard of hearing, and professional ASL interpreters. These signers were predominantly right handed.

The videos were classified into assertions (AS) and statements (ST). Assertions are considered sentences where the signer declared an action will occur while statements are sentences stating a fact. For every video, we extracted the frame at 0.3, 0.5 and 0.75 time stamp of the videos to be used for training and testing. Upon further inspection, we chose to solely use the midpoint frame for both training and testing because the grammatical markers that differentiate AS and ST sentences are usually found towards the latter part of a signed sentence [3]. Furthermore, frames taken too early and too late often did not offer any sort of facial clues that can be used to distinguish the different types of sentences as the meaning of the sentence is usually conveyed towards the middle of the sentence.

To increase the amount of training data used for this project, we applied data augmentation methods to the training data set. Such augmentation methods included rotation,

Fig. 2.4 – Example of Data Augmented Images (Resized) on Image from How2Sign Dataset [7].

vertical shifting, horizontal shifting, and resizing. The degree of rotation, shifting and resizing were randomly chosen by the "ImageGenerator" function from the sklearn library. The data augmentation methods resulted in 3660 images and were all used for training. Testing was done on the remaining 53 videos and these frames were not augmented in any way.

2.3 Dlib Library

To calculate the facial points used for classification, we utilized the facial landmark detector and predictor from the Dlib library [15] to detect the faces of the subjects and track key facial points. First, we need to detect and localize the subjects' faces before extracting facial points. To do so, we used the *get_frontal_face_detector* function from the Dlib library which creates a facial detector that was developed using Histogram of Oriented Gradients (HOG) and Linear SVM classifier in conjunction with a image pyramid and sliding window

Fig. 2.5 – Example of Data Augmented Images (Horizontal Shifting) on Image from How2Sign Dataset [7].

method [37] to detect the subjects' faces before extracting key facial points.

After detecting the face, we used the *shape_predictor* function based on the landmark estimator algorithm proposed by Vahid et al. [14] and trained on the iBUG 300-W facial landmark dataset [26] to determine the location of certain facial points on the subject's face. This function outputs 68 coordinates with the origin being at the top left of the image.

In Listing 2.1, we implement the 2 functions that we use to extract the facial points of each image. We begin by creating a detector object using the Dlib library to detect the face, then within the for loop we run the predictor function so that it can extract the 68 facial points. At every iteration of the for loop, the predictor determines one of the facial points, thus there is a total of 68 iterations done per image.

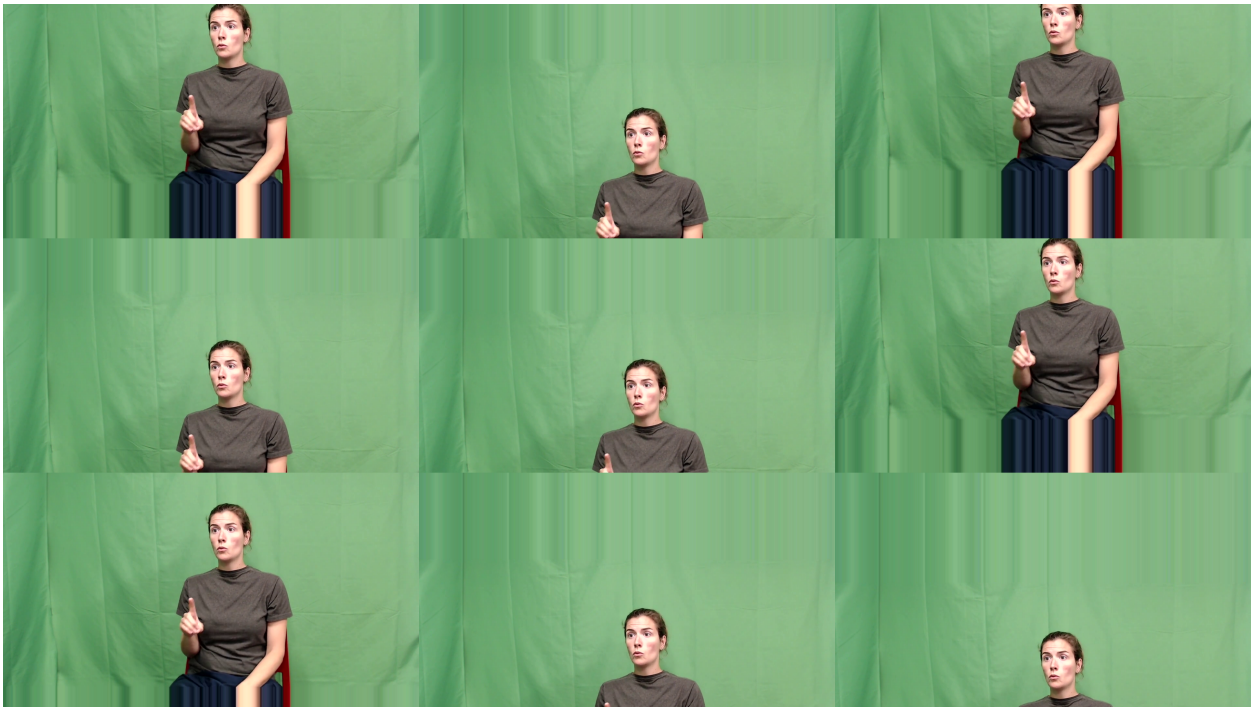

Fig. 2.6 – Example of Data Augmented Images (Vertical Shifting) on Image from How2Sign Dataset [7].

```
def run_dlib_model(self):
    '''
    Runs the model on gray scale image.
    '''
    rects = self.detector(self.gray)

    for (i, rect) in enumerate(rects):
        coor = self.predictor(self.gray, rect)
        self.coor = face_utils.shape_to_np(coor)
```

Listing 2.1 – Implementing dlib landmark tracking.

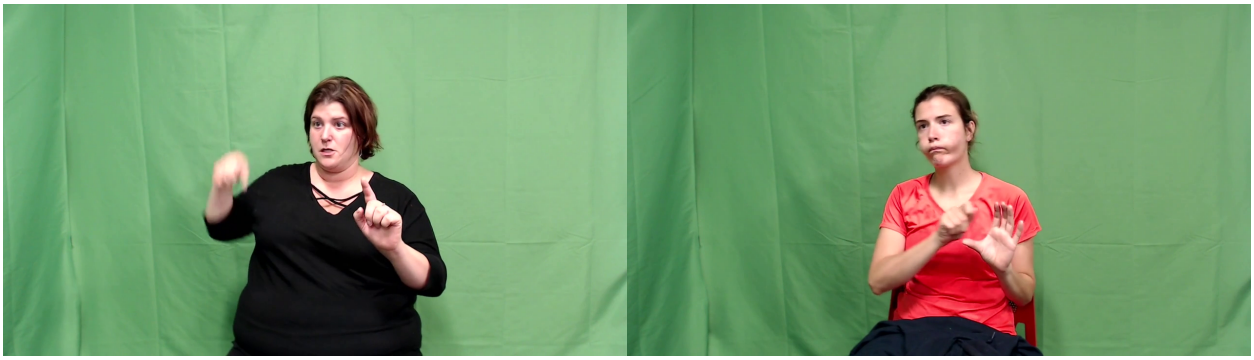

Fig. 2.7 – AS (Left) vs ST (Right) from How2Sign Dataset [7].

2.4 Changing Origin

After determining facial points, we calculate the feature vectors and angles using the output of *shape_predictor*. Before starting angle calculations, the coordinate system of the facial points needs to be changed to account for any possible movement of the user. The current origin is at the top left corner of the image and the frame of reference is the entire frame of the video. If the origin is not changed, then the vectors will be affected by the location of the user within the frame and will not be solely based on the user's face. Thus we change the coordinate system to have the origin to be at one of the facial points, so it would be a global coordinate system where the frame of reference is the user's face.

The point at the center of the subject's chin (the red point in Figure 2.8) is chosen to be the origin and all the other points is converted to the new origin's coordinate system before further calculations is done. The point at the chin is chosen because it would result in the largest angles when undergoing angle calculations. After changing the origin, the 68 original points is reduced to 67 points.

```
def get_relative_coordinate(self) -> np.ndarray:
    '''
    Sets the selected coordinate to be the origin and
    calculates the other coordinates wrt to the new origin.
    '''
    origin = self.coor[self.origin]
    dim = np.shape(self.coor)
    ones = np.ones([dim[0], 1])
    x_origin = origin[0]
    y_origin = origin[1]
    x_origin_list = ones*x_origin
    y_origin_list = ones*y_origin
    self.x_relative_coor = self.x_coor - x_origin_list
    self.y_relative_coor = y_origin_list-self.y_coor
    self.relative_coor = np.array([[x[0], y[0]] for x, y in
    zip(self.x_relative_coor, self.y_relative_coor)])
```

Listing 2.2 – Changing the origin to one of the 68 facial points Dlib outputted.

2.5 Angle Calculation

Due to the inconsistencies in distances between the points from factors such as recording camera angle and differing subject's face sizes, we cannot consider the vectors alone and convert the vectors to angles using the following equation:

$$\theta_i = \arccos \frac{x_0 - x_i}{\sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2}} \tag{2.1}$$

where i is the i^{th} point and (x_0, y_0) is the red point on Figure 2.8. Using angles ensures that factors such as inconsistencies in the subjects faces or rotation done by the subject

during the video do not affect the model. Every image resulted in 67 angles that is later on reduced to principal components.

```
def calculate_angle(self):
    '''
    Calculates the magnitude and angle of each point
    '''
    x = self.relative_coor[:, 0]
    y = self.relative_coor[:, 1]
    squared_x = np.square(x)
    squared_y = np.square(y)
    self.mag = np.sqrt(squared_x+squared_y)
    inside = y/self.mag
    self.angle = np.arccos(inside)
```

Listing 2.3 – Implementing angle calculation.

Fig. 2.8 – Raw Data from Dlib Library [7].

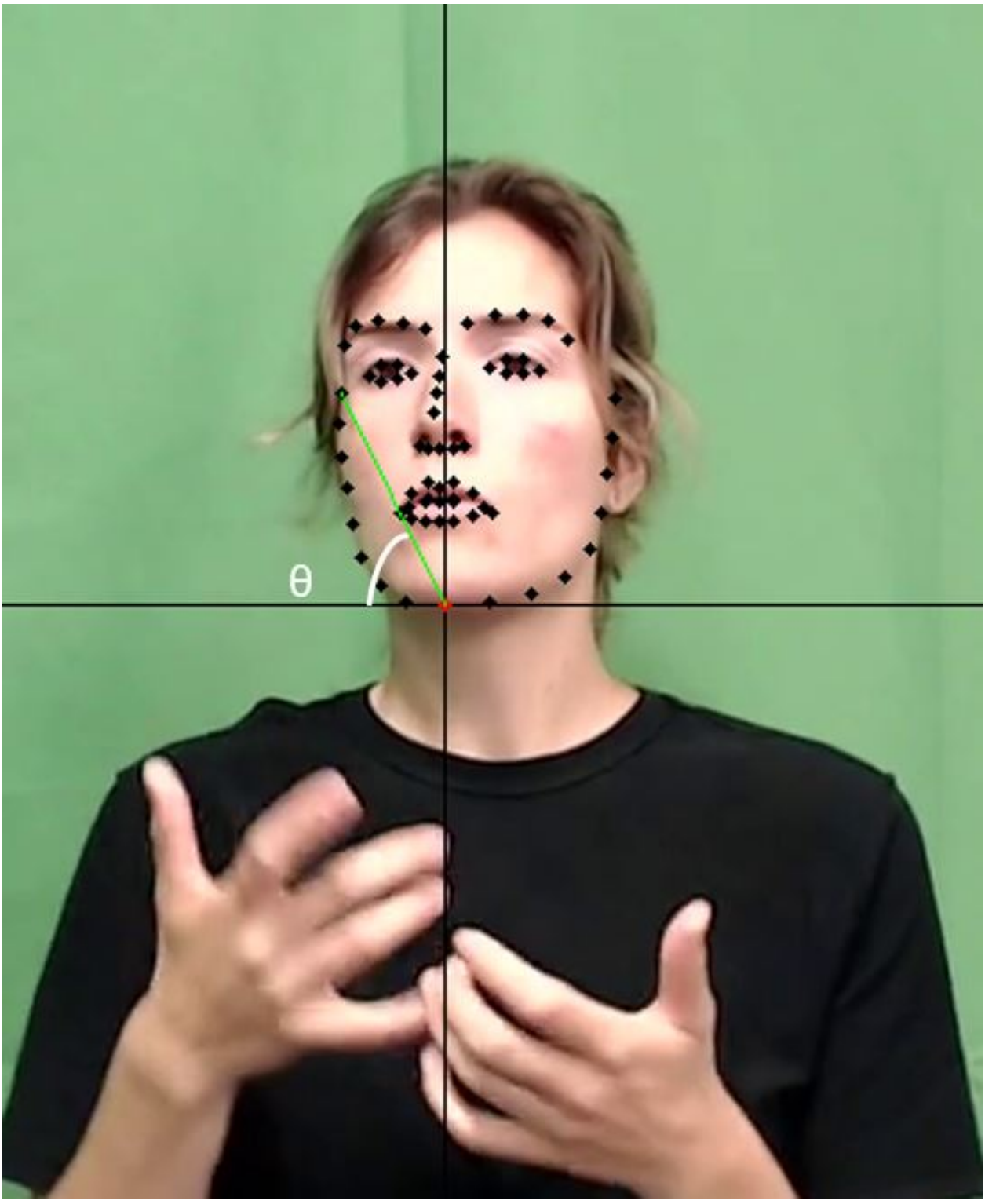

Fig. 2.9 – Example of Angle Calculation using the Red Point as the Origin on Image from How2Sign Dataset [7].

CHAPTER 3

Classification Model Pipeline

3.1 Introduction

After quantifying the facial expressions of each video, we can begin training a classification model aimed towards classifying the sentence type of the sentence being signed. To begin training a model, the data set is split into a training and testing subset, where the training set undergoes data augmentation to increase its size and the testing set is left untouched until after the model is trained and is ready to be tested. Before starting the training phase, PCA is performed to reduce dimensionality while maintaining performance. During the training phase (see Figure 3.1), the models undergo the typical training and tuning of its hyperparameters. To do so, both models are implemented using functions available in the sklearn library and are further tuned according to a grid search. Each model has different hyperparameters that need to be adjusted and optimized while also considering the time it takes to train and test each model.

3.2 Reducing Redundant Data

A key goal in this project is to keep computational power consumption low and one way to do so is to reduce the redundant data present. The balance between minimizing

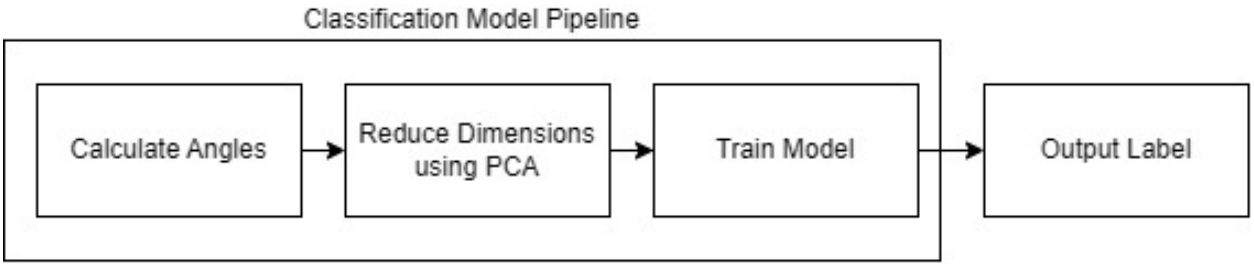

Fig. 3.1 – Training workflow for the proposed classification model pipeline.

computational power consumption and maximizing performance can be effected by how much data is used to train and test the model. Ideally, we would remove most of the redundant data while maintaining the information the data set provides. For this project, we remove redundant data by implementing PCA.

3.2.1 Reducing Dimensions using Averaging

We also try implementing averaging as a way to reduce dimensionalty. This consist of selecting key facial areas and calculating the average distance of a group of points that defined that key facial area and using that for angle calculations. We chose the 2 eyebrows, the nose and the mouth as key facial areas. Each eyebrow has 5 key points. The nose has 4 and the mouth had 25 key points. Each set of points was averaged out, resulting in 4 averaged distances. These were used in angle calculation and then the newly calculated angles were used to train the Random Forest classifier. This method resulted in terrible accuracies. The accuracies calculated ended up being under 0.5, which is worse than letting chance classify the 2 classes. Various combinations of different points for each key facial areas also resulted in poor performance. Thus, we choose to explore PCA as another option to reduce dimensionality.

3.2.2 Principal Component Analysis (PCA)

As discussed above, we utilize PCA to reduce the number of features used to train and test the model. We implement PCA using the sklearn library [31] and perform PCA on both the training and testing data sets. This is because both data sets need to have the same number of features. One parameter of the PCA function within sklearn is the number of PCs the algorithm is suppose to reduce the current data set. The possible choices for the number of PCs is 1 until the total number of features minus 1 [22]. The number of PCs to be calculate was determined through a grid search process, where a variety of numbers for PCs was calculated and the one that produced the best results was kept.

3.3 Preliminary Classifiers Results

To begin deciding which classifying algorithm is the most appropriate for this, we examine the accuracy of various models that are already built into sklearn [22] to see which ones performed the best before dimensionality reduction. As seen in Table 3.1, the top 3 performing classifiers were the KNeighbors Classifier, Linear SVC, and NuSVC. The Decision Tree classifier, Random Forest Classifier, and AdaBoost Classifier also look promising since their accuracies are roughly close to 0.8. At first glance, the best choice seems to be KNeighbors Classifiers, most facial classifiers and facial classification in ASL uses the SVM ([20], [16], [1], and [19]) and Random Forest classifiers ([17], [35], and[14]). Furthermore, amongst the top performing classifiers, the Linear SVC and NuSVC are types of SVM algorithms and the Decision Tree Classifier and Random Forest Tree Classifier exhibit the potential of utilizing decision trees. Thus, we choose the Random Forest and SVM classifiers as our algorithms for this project.

Classifier	Accuracy
KNeighbors Classifier	0.865
Linear SVC	0.846
NuSVC	0.808
Decision Tree Classifier	0.788
Random Forest Classifier	0.788
AdaBoost Classifier	0.788
Gradient Boosting Classifier	0.769
Gaussian NB	0.519
Linear Discriminant Analysis	0.615
Quadratic Discriminant Analysis	0.731

Table 3.1 – Grid Search for Optimal PC for Random Forest Model (Number of Trees = 200, Min. Sample Leaf Size = 5)

3.4 Random Forest Classification

The random forest classifier is a model that essentially creates a forest of decision trees that vote on a final classification. The number of trees and the number of nodes and branches each tree are factors that can affect the performance and execution time of the model. The more trees the forest has the longer it would take for the model to make a decision on a subject, because there is more trees to cast a vote on the final decision. On the flip side, the more trees there are, the more likely the final classification would be correct. Furthermore, each tree will have a set number of branches and nodes that it uses to make a decision and the more nodes and branches each tree has the more features the tree has to classify each subject. However, this can also slow down execution time, since the more nodes and branches each tree has, the longer each tree will take to make a decision. All 3 of these factors is dependent on the hyperparameters of the model. Thus, we need to take into account the

these factors when deciding on the hyperparameters that will result in the fastest execution time while maintaining performance.

3.4.1 Code Implementation

We implement the random forest classifier algorithm using the "RandomForestClassifier" function from the sklearn library [22]. The function we created to run the algorithm is shown in List 3.1. The "RandomForestClassifier" function has 3 hyperparameters that we needed to determine. Our initial parameter to consider is the *n_estimator* parameter which indicates how many trees will be in our forest. As discussed above the number of trees can affect how long it takes to execute the whole process, but the more trees there are the better the prediction. The next parameter we need to set is the *min_samples_leaf* and this determines the minimum samples at each node of a tree [22]. The last parameter we need to determine is the *max_depth* parameter that controls the size of each tree and how many levels of decisions a tree can create [22]. The rest of the hyperparameters were set to the defaulted values, since the default values often result in decent results [8].

```
def random_forest_model(self):
    '''
    Runs random forest model on inputted data and calculates
    accuracy and confusion matrix using sklearn algorithms.
    '''
    clf = RandomForestClassifier(n_estimators=self.n_estimators,
    max_features="sqrt", random_state=0,
    min_samples_leaf = self.samples_leaf, max_depth = self.depth)
    clf.fit(self.x_train_aug, self.y_train_aug)
    clf.score(self.x_train_aug, self.y_train_aug)
    self.y_pred = clf.predict(self.x_test)
    self.accuracy = metrics.accuracy_score(self.y_test,
    self.y_pred)
    self.confusion = confusion_matrix(self.y_test, self.y_pred)
    self.confusion_norm = confusion_matrix(self.y_test,
    self.y_pred, normalize = "all")
    self.report = classification_report(self.y_test, self.y_pred)
```

Listing 3.1 – Implementing random forest classifier using sklearn.

3.4.2 Hyperparameters

We focus on 3 hyperparameters of the random forest tree classifier: the number of trees, the minimum sample leaf size and the max depth. A key property of the random forest tree classifier is that the ensemble of the trees in the model are diverse. The more diverse the trees are the better, because the diversity in the trees the lowers the likelihood of having duplicate trees [25] which leads to overfitting [29]. The diversity of the forest is affected by these 3 parameters, however as discussed before, if these parameters are too large, then it

would slow down the algorithm, Thus, we need to fully understand the importance of each parameter to determine its optimal value.

The first parameter, *n_estimator*, determines the number of trees the ensemble will consist of. Choosing an appropriate value for this parameter is important because the number of trees can affect different aspects of the model including execution time, possibility of over fitting, accuracy and more [4]. An advantage to having multiple trees is that the mistake of one tree can be compensated by other trees [25], thus having multiple trees is beneficial. However, having multiple trees can slow down the model. Thus, we need to determine the optimal number of trees that can ensure no one tree dominates the overall decision and keep execution time low.

The parameter, *min_samples_leaf*, determines the minimum samples in each leaf node. In other words, it sets a minimum size for each leaf so that when a node splits, if the child node is smaller than the minimum, the node does not continue to split and stops here. Since each tree uses a different training data subset to increase diversity [24], the leaf size can also contribute to the diversity of each tree by creating leaves per tree. The more splits there are, the more diverse the tree but the longer it takes to produce one tree decision. Thus, we need to ensure that the value we chose for this sample allows for enough nodes to be split to ensure diversity amongst the trees while maintaining a low execution time.

The last parameter we focus on is the *max_depth* which determines how many levels each tree can have. Traditional random forest tress usually allows the tree to keep constructing until the maximum level is reached, however there has been research that suggest bounding the maximum level and decreasing the size of each tree can result in a higher accuracy and prevent overfitting [18]. Furthermore, having smaller trees can decrease execution times by requiring less computational power. Thereby, we need to find an appropriate max depth that will allow for optimal performance while maintaining low computational cost.

3.5 Support Vector Machine

The SVM model works by finding the hyper plane that will separate classes and maximize the distance between the hyper plane and the classes; in this project, it works to find the hyperplane separating AS and ST using the angles calculated. To do so, it must determine the relationship between the 2 classes (i.e. linear VS. nonlinear relationship) and find the appropriate kernel to represent this relationship. This feature selection method is considered an embedded method and is known for its computational efficiency and can help prevent overfitting [23]. One way to determine the relationship between classes is to plot a pair plot using the seaborn library [36]. This function will plot each feature pairwise, allowing one to visualize the relationship between features, and help in finding out the optimal hyperparameters.

3.5.1 Code Implementation

We use the *SVC* (support vector classifier) function from the sklearn library [22]. We created the function in Listing 3.2 to implement the pipeline to run the SVM algorithm from sklearn. The two parameters we focus on here is the *kernel* and *gamma*. The *kernel* parameter refers to the kernel type used to determine the hyper plane that would maximize the distance between the 2 classes. The *gamma* parameter sets the coefficient of the kernel to $\frac{1}{n_{features}}$, where $n_{features}$ is the number of features used to train the model [22]. These 2 parameters are determined by analyzing the type of relationship between the 2 classes.

```
def SVM(self):
    '''
    Runs SVM on inputted data and calculates accuracy and
    confusion matrix using sklearn algorithms.
    '''
    clf = make_pipeline(StandardScaler(), SVC(gamma='auto',
    kernel = "poly"))
    clf.fit(self.x_train_aug, self.y_train_aug)
    clf.score(self.x_train_aug, self.y_train_aug)
    self.y_pred = clf.predict(self.x_test)
    self.accuracy = metrics.accuracy_score(self.y_test,
    self.y_pred)
    self.confusion = confusion_matrix(self.y_test, self.y_pred)
    self.confusion_norm = confusion_matrix(self.y_test,
    self.y_pred, normalize = "all")
    self.report = classification_report(self.y_test, self.y_pred)
```

Listing 3.2 – Implementing SVM using sklearn.

3.5.2 Hyperparameters

A key hyperparameter of the *SVC* function is the *kernel* parameter. Within the sklearn library, this parameter can hold 5 different values: linear, polynomial, radial basis function (rbf), sigmoid, and precomputed [22]. Each kernel corresponds with a different type of separation boundary between the classes and has been proven to drastically change the results for various applications ([5], [21], [10]). One way to visualize the type of separation boundary the 2 classes have is by plotting the pairwise relationship of the 67 angles. As shown in Fig. 3.2, the 2 classes tend to overlap quite a bit even after performing PCA. This

is problematic as the SVM algorithm works to find a hyperplane to separate the classes and if there is no separation, the model would be expected to perform poorly. However, the type of kernel used in the SVM algorithm can alleviate this problem and can handle more complex relationships between features such as non linearity [13]. We consider the linear, polynomial, sigmoid and rbf as possible choices for the kernel parameter since the precomputed input requires us to create our own kernel.

The linear kernel is usually used for data that is linearly separable. This kernel is the simplest one out of the 4 possible kernel sources and performs the best when there is many features present with a linear relationship [6]. In other works where the authors tested and compared different kernels to see which one performed the best on the same data sets, the linear kernel has been proven to work the best with lung cancer datasets [10] and lower back pain symptoms [13].

The nonlinear kernel handles data that have a nonlinear relationship or in other words can be modeled a polynomial. Due to its nature, the polynomial kernel tends to have more parameters to determine than the other kernels we are considering [11]. One of those parameters is the degree of the polynomial of a nonlinear kernel. We used the sklearn's default value which is a polynomial of 3 degrees when performing the grid search for the optimal parameters.

The rbf kernel is the most commonly used in various applications and is considered translational invariant [6]. This kernel is a popular choice amongst various applications because it can handle both the linear and nonlinear cases and in some cases, its parameters can be altered to behave like other kernels such as the linear, nonlinear and sigmoid kernel [11]. It has been shown to work the best in such works like termite detection using acoustic signals[2]

The sigmoid kernel, also known as the the hiperbolic tangent kernel, is more commonly used in neural networks [6]. Usually sigmoid kernels require more tuning than the typical rbf kernal, but in some cases where the amount of feature vectors are high or there is a nonlinear

boundary in 2 dimensions, the sigmoid kernel can perform as good or better than the rbf kernels [9].

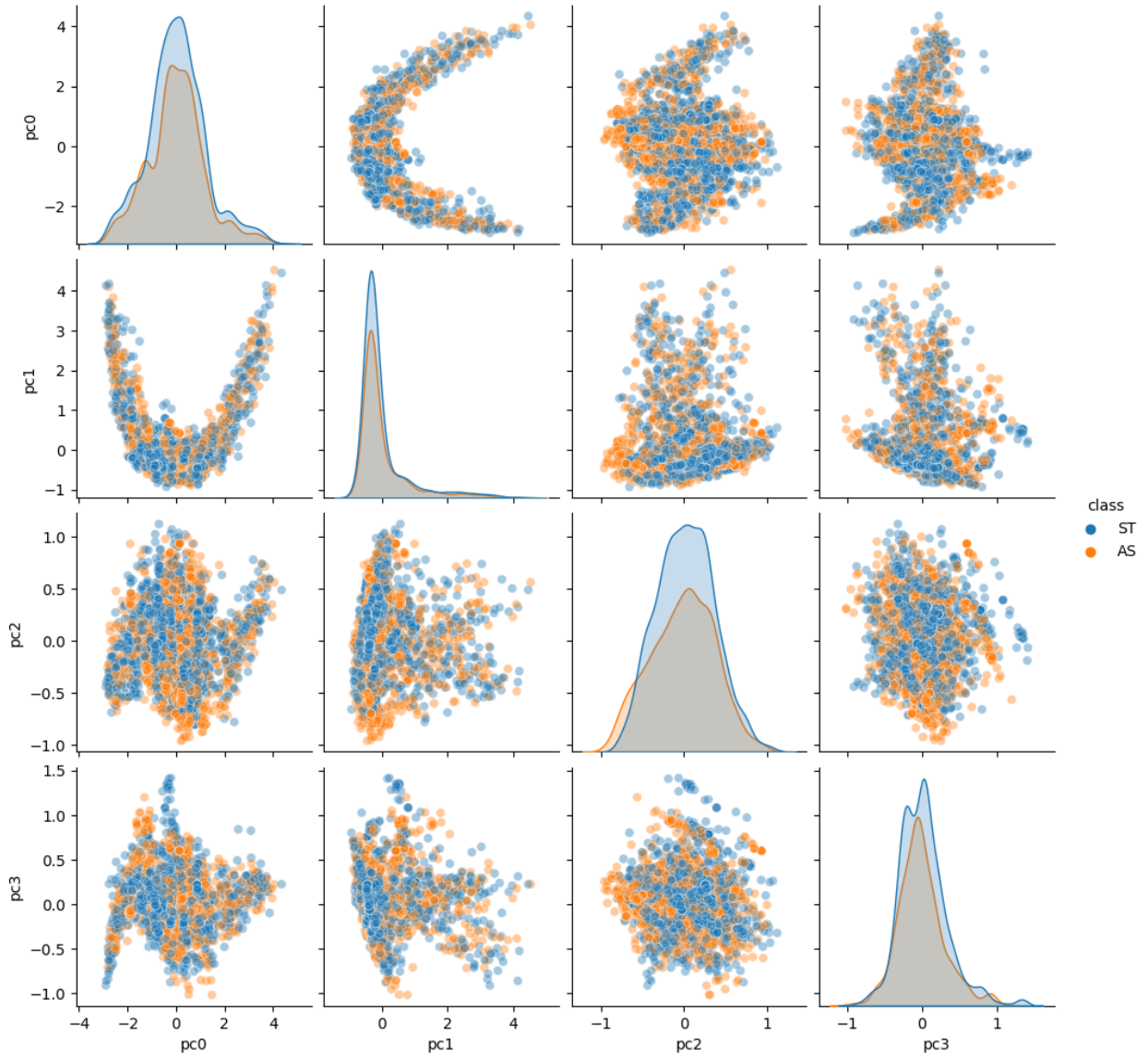

Fig. 3.2 – Pairwise Relationship for 4 PC

CHAPTER 4

Results

We evaluate the performance of the models using 3 metrics: its accuracy, confusion matrix and the execution time. The accuracy (Eq 4.1) refers to how many videos the model was able to correctly classify. Since there is only 2 classes, the accuracy if the video is classified by chance is 0.5. We are aiming for an accuracy of 0.8 or higher. The confusion matrix offers more details on how the model performed with respect to specific classes. The confusion matrix lists the true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), and false negative rate (FNR). In general, we want a model that will maximize the TPR and TNR while minimizing the FPR and FNR. We aim to have a TPR and TNR of 0.8 or higher. We find the most optimal combination of PCs and hyperparameters by analyzing the evaluation metrics we just described. A grid search tends to be more time consuming, however other methods that optimize parameters through approximations and more complex mathematical analysis tends to take roughly the same amount of time and the time it takes to conduct a grid search can be reduced by such methods as parallel processing [11]. The execution times we examined were the time it took to train the model and test it. We also look at the time it took to run PCA, since that was done right before training and testing the model. This is so we can see how fast the model and pca would perform during the training and testing phase. The entire process was run on Google Colab Pro+ using the

Intel(R) Xeon(R) CPU @ 2.20GHz.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$TPR = \frac{TP}{TP + FN} \tag{4.2}$$

$$FPR = \frac{FP}{FP + TN} \tag{4.3}$$

$$TNR = \frac{TN}{TN + FP} \tag{4.4}$$

$$FNR = \frac{FN}{TP + FN} \tag{4.5}$$

4.1 Random Forest Classification

To start the gird search process, we set the number of trees to 200, the minimum sample leaf size to 5 and leave the max depth at the default value until the grid search for that parameter is done. As shown in Table 4.1, we can improve the performance of the model further using PCA. Table 4.1 shows the confusion matrix of the model before running PCA. As seen in the TPR and TNR, there is a bias towards the ST class, since the TPR and TNR for ST is significantly higher than the AS class. This indicates the model tends to only correctly classify the ST class. We want a model to correctly classify both classes, thus we implement PCA to see if we can have a more balanced model. The first step to implementing PCA is to determine the optimal PC value. After determining the optimal PC value, we determined the optimal number of trees next, followed by the optimal minimum sample leaf size and lastly the max depth through the means of a grid search. At each stage of determining the optimal parameter, we run 10 trials of each parameter set to see the

average execution times.

Class	TPR	FPR	TNR	FNR
AS	0.608	0.103	0.897	0.391
ST	0.897	0.391	0.609	0.103

Table 4.1 – Confusion Matrix Random Forest Tree before PCA (Accuracy = 0.769, PC = 0, Number of Trees = 200, Min. Sample Leaf Size = 5)

4.1.1 PC Grid Search

We begin by finding the optimal PCs that will result in the highest accuracy. To find the optimal PC, we perform a grid search where we ran the model with different PCs and we look at the accuracy produced by each PC. We kept the other parameters of the model constant. Tables 4.2 shows the accuracy that resulted from the PC grid search. The random forest model *n_estimators* parameter was kept at 200 and the *min_sample_leaf* was at 5. We chose to chosen to look at PC values at intervals of 4 because the accuracy tends to stay similar to PCs of closer values. Preliminary results show that the accuracy is the highest when the PC is set to 20. However, there is a possibility that a PC of 20 could have a poor confusion matrix, so we decided to also analyze the confusion matrix when the PC is 4. Since it is the smallest number and its accuracy was over 0.8, it potentially offers a better balance of performance and computational cost.

When further examining the confusion matrix for PC = 20 (Table 4.4), the TPR for the AS class is below 0.8. We are aiming for the TPR and TNR for both classes to be at least 0.8. Furthermore, usually a larger number PC causes the model to take longer to run as seen in Table 4.5 where the average execution time to train the model for PC = 4 is faster than PC = 20. Thus, we try examining the confusion matrix for PC of 4, since it is the smallest PC that had an accuracy of over 0.8. When looking at the confusion matrix for a PC of 4, both the TPR and TNR for both classes is over 0.8. Furthermore, there is a better balance between the TPR and TNR of both classes, indicating there is little bias in the model. Thus,

we choose to use a PC of 4 when continuing onto the grid search for the hyperparameters.

When looking at Table 4.5, 4.7, 4.8, and 4.6, PC = 4 has the best performance while also being computational efficient. The training execution times for both the model and PCA indicate that a PC = 4 is faster than PC = 20, however the testing times show that PC = 20 is faster than PC = 4. This could be due to the fact that the testing data set is significantly smaller than the training set, and thus when comparing values there is a slight variation due to the nature of running the program on Google Colab Pro+. When using Google Colab Pro+, resources are shared amongst users and there is a possibilty that at that moment there were multiple users utilizing Google Colab resources at the time. The difference in execution times between PC = 4 and PC = 20 when ran on the testing data set (Table 4.6 and 4.8) is minuscule, so we used the training execution times to compare the 2. We chose PC = 4 for the rest of the grid searches.

PC	**Accuracy**
4	0.827
8	0.827
12	0.788
16	0.769
20	0.827
24	0.808
28	0.788

Table 4.2 – Grid Search for Optimal PC for Random Forest Model (Number of Trees = 200, Min. Sample Leaf Size = 5)

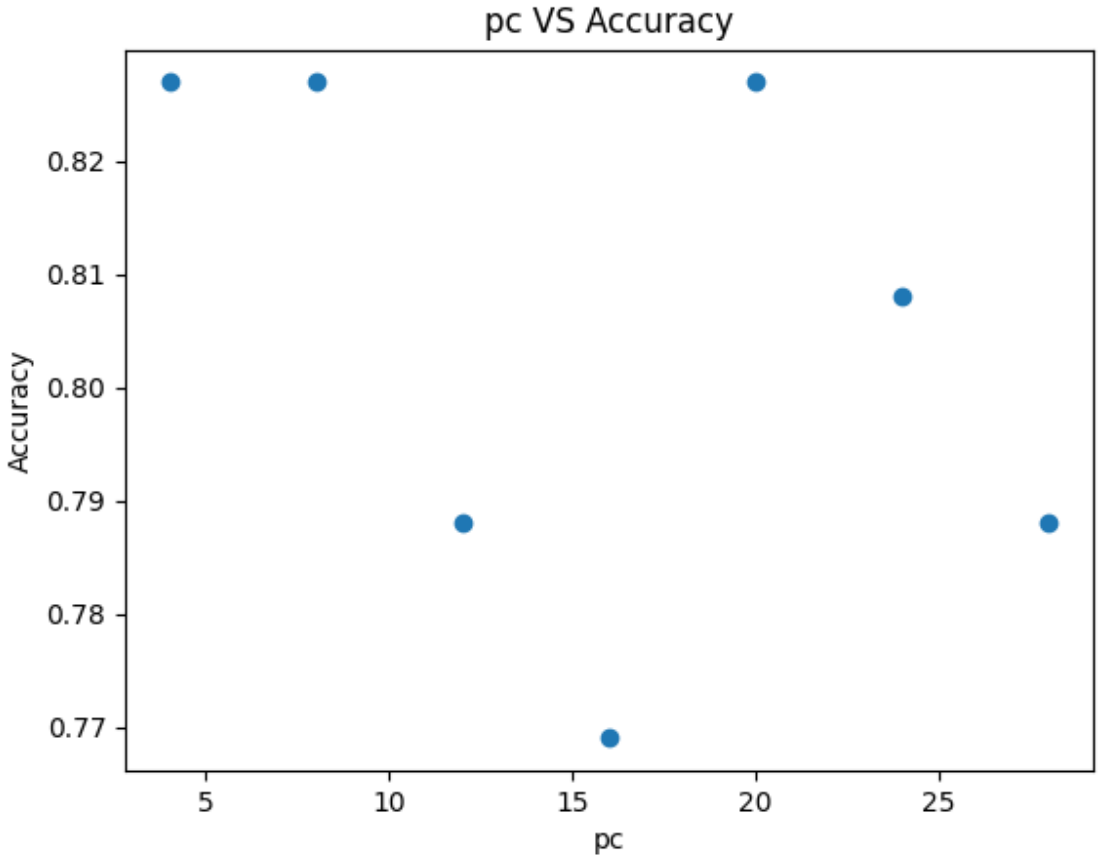

Fig. 4.1 – PC VS. Accuracy for Random Forest Tree

Class	TPR	FPR	TNR	FNR
AS	0.826	0.172	0.827	0.174
ST	0.828	0.174	0.826	0.172

Table 4.3 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.827, PC = 4, Number of Trees = 200, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.739	0.034	0.966	0.261
ST	0.966	0.261	0.739	0.034

Table 4.4 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 20 Number of Trees = 200, Min. Sample Leaf Size = 5)

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	1448.13	1426.846	1905.353	1401.402	1417.185	1459.921	1474.254	1415.991	1427.595	1687.23	1506.391
8	1543.208	1543.099	2761.738	1521.554	1504.551	2088.304	1469.94	1482.328	2102.186	1496.72	1751.363
12	1979.312	2494.04	2005.445	1965.914	2057.433	2292.332	1932.043	1959.907	2286.48	1899.733	2087.264
16	2441.773	3088.231	2396.165	2419.143	3348.205	2377.223	2343.828	3275.478	2337.851	2376.554	2640.445
20	3537.033	2439.434	2467.811	3175.573	2441.134	2741.382	3738.071	2472.908	2447.493	2432.638	2789.348
24	2767.104	2497.678	2756.523	2836.428	2470.891	2447.669	3020.397	2452.924	2455.318	3357.313	2706.224
28	3072.987	2977.449	3789.341	2891.961	2897.539	3872.685	3485.908	2897.762	3788.97	2994.46	3266.906

Table 4.5 – Grid Search Model Training Execution Time (ms) for Optimal PC for Random Forest Model (Number of Trees = 200; Min. Leaf Sample = 5)

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	21.737	21.308	29.69	20.78	21.024	30.991	21.801	23.913	35.878	20.532	24.765
8	21.28	23.903	21.102	20.6	26.865	33.468	22.376	21.051	29.17	21.754	24.157
12	21.215	30.286	21.386	20.867	29.117	20.859	21.053	27.903	21.426	20.641	23.475
16	20.754	21.191	21.368	22.131	25.635	21.235	20.057	38.546	21.218	21.916	23.405
20	29.413	22.594	26.361	31.291	21.275	21.224	29.509	20.628	19.847	21.593	24.373
24	22.317	21.717	30.52	21.317	20.563	21.942	20.369	20.154	21.903	36.096	23.69
28	21.011	21.26	23.729	20.28	21.497	33.835	19.159	20.664	30.627	23.534	23.56

Table 4.6 – Grid Search Model Execution Time (ms) on Testing Data Set for Optimal PC for Random Forest Model (Number of Trees = 200; Min. Leaf Sample = 5)

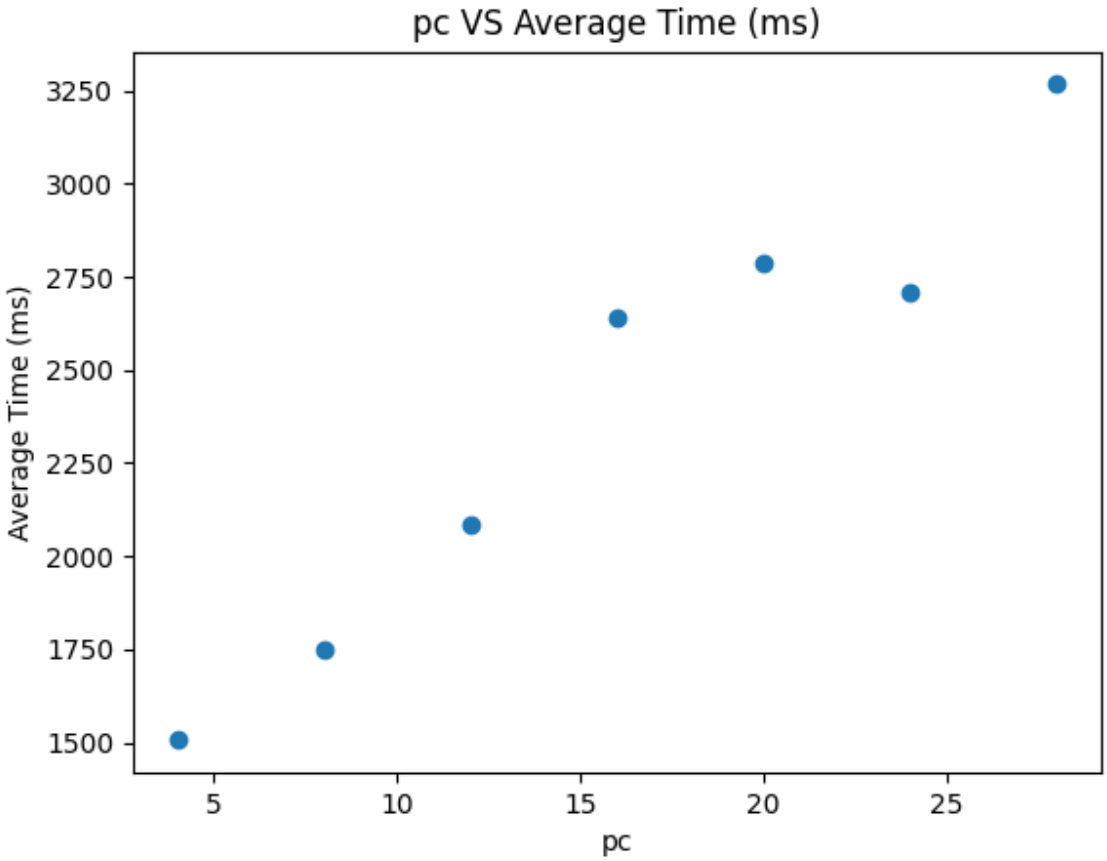

Fig. 4.2 – PC VS Average Time (ms) for Random Forest Tree Model on Training Data

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	1.588	1.793	1.678	1.719	1.931	1.877	6.416	1.862	3.022	2.188	2.407
8	2.362	2.476	7.815	3.146	2.592	7.768	2.464	3.069	4.116	2.598	3.841
12	3.081	3.245	3.259	3.293	3.083	5.152	3.169	2.824	8.982	3.125	3.921
16	3.813	17.967	3.682	3.692	9.675	4.004	3.633	7.137	3.624	3.465	6.069
20	4.35	4.423	4.553	4.018	4.233	4.219	4.091	3.669	4.579	8.483	4.662
24	10.087	9.336	5.178	10.166	4.75	5.22	9.711	8.969	5.475	10.168	7.906
28	6.307	5.905	14.495	6.081	6.294	5.332	13.232	6.046	6.016	13.717	8.342

Table 4.7 – Grid Search PCA Training Execution Time (ms) for Optimal PC for Random Forest Model (Number of Trees = 200; Min. Leaf Sample = 5)

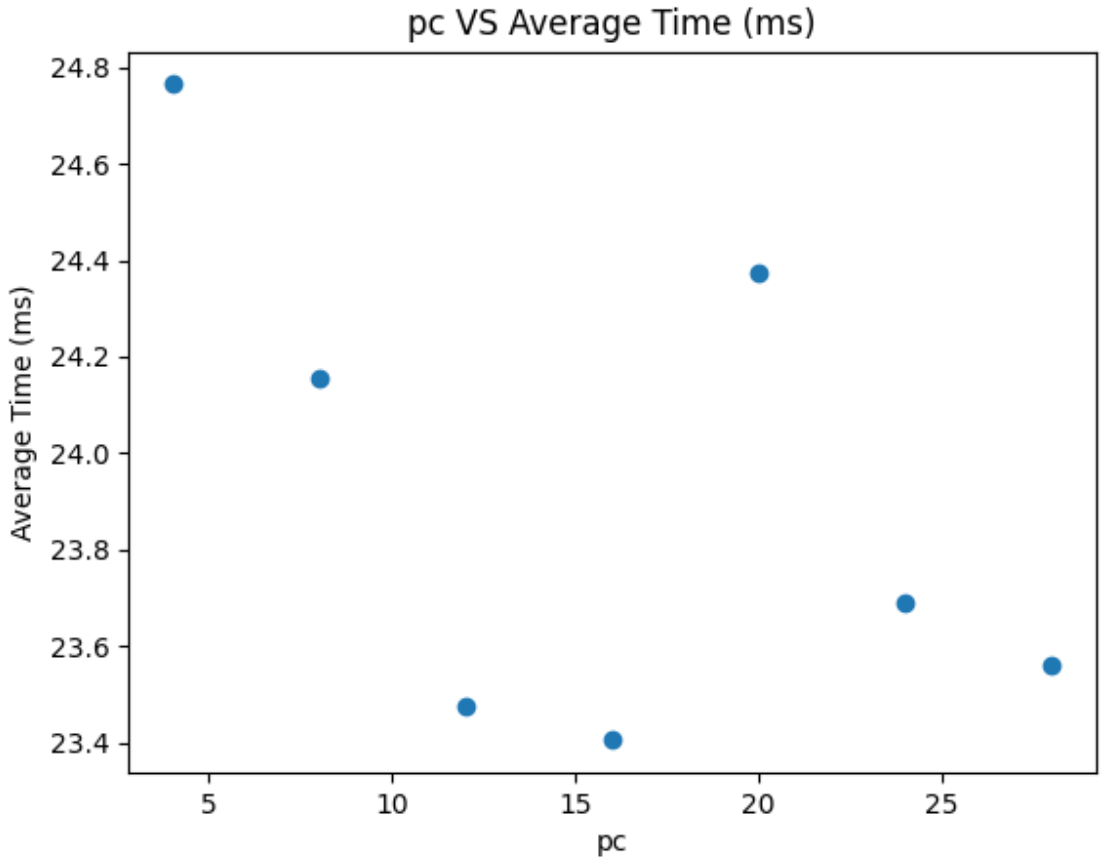

Fig. 4.3 – PC VS Average Time (ms) for Random Forest Tree Model on Testing Data

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	0.177	0.255	0.268	0.166	0.179	0.24	0.165	0.208	0.242	0.164	0.206
8	0.179	0.185	0.163	0.184	0.243	0.237	0.186	0.206	0.235	0.173	0.199
12	0.193	0.231	0.195	0.165	0.274	0.183	0.165	0.218	0.193	0.18	0.2
16	0.177	0.231	0.191	0.203	0.186	0.237	0.188	0.24	0.164	0.194	0.201
20	0.344	0.18	0.179	0.263	0.187	0.166	0.211	0.185	0.196	0.199	0.211
24	0.208	0.164	0.254	0.194	0.195	0.181	0.18	0.176	0.211	0.225	0.199
28	0.205	0.203	0.424	0.184	0.184	0.231	0.196	0.22	0.228	0.19	0.226

Table 4.8 – Grid Search PCA Execution Time (ms) on Testing Data Set for Optimal PC for Random Forest Model (Number of Trees = 200; Min. Leaf Sample = 5)

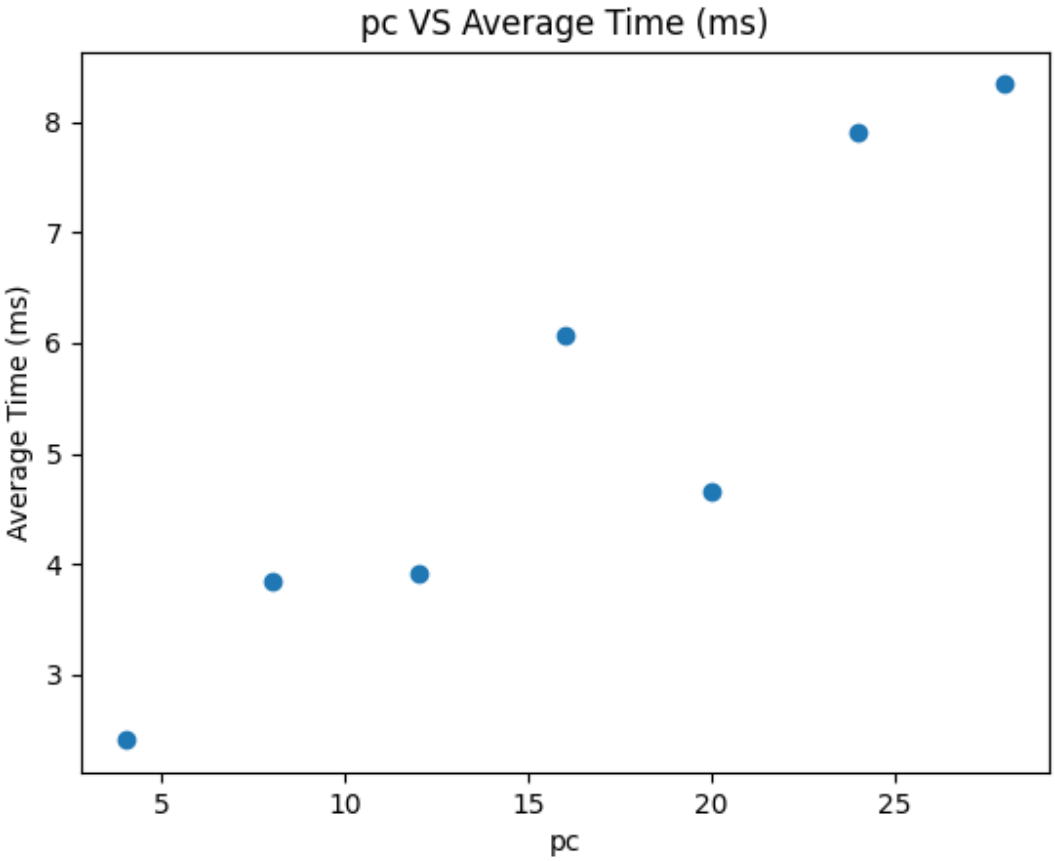

Fig. 4.4 – PC VS Average Time (ms) for PCA on Training Data

4.1.2 Number of Trees Grid Search

After conducting the grid search for the optimal PC, we repeat the same process for determining the optimal number of trees, but we set the PC value to 4 and continue to keep the minimum sample leaf value to 5. We start by analyzing the accuracy and execution time for the following values for the trees: 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300. Initially, we look at the accuracy of each value to see which ones look the most promising. Amongst the most promising, we look at the confusion matrix and average execution time over 10 trials to see which one performs the bests overall.

According to Table 4.9, the number of trees that produced the highest accuracy, 0.865, is 240, 260, 280, and 300 trees. However, those are really large amounts of trees and are

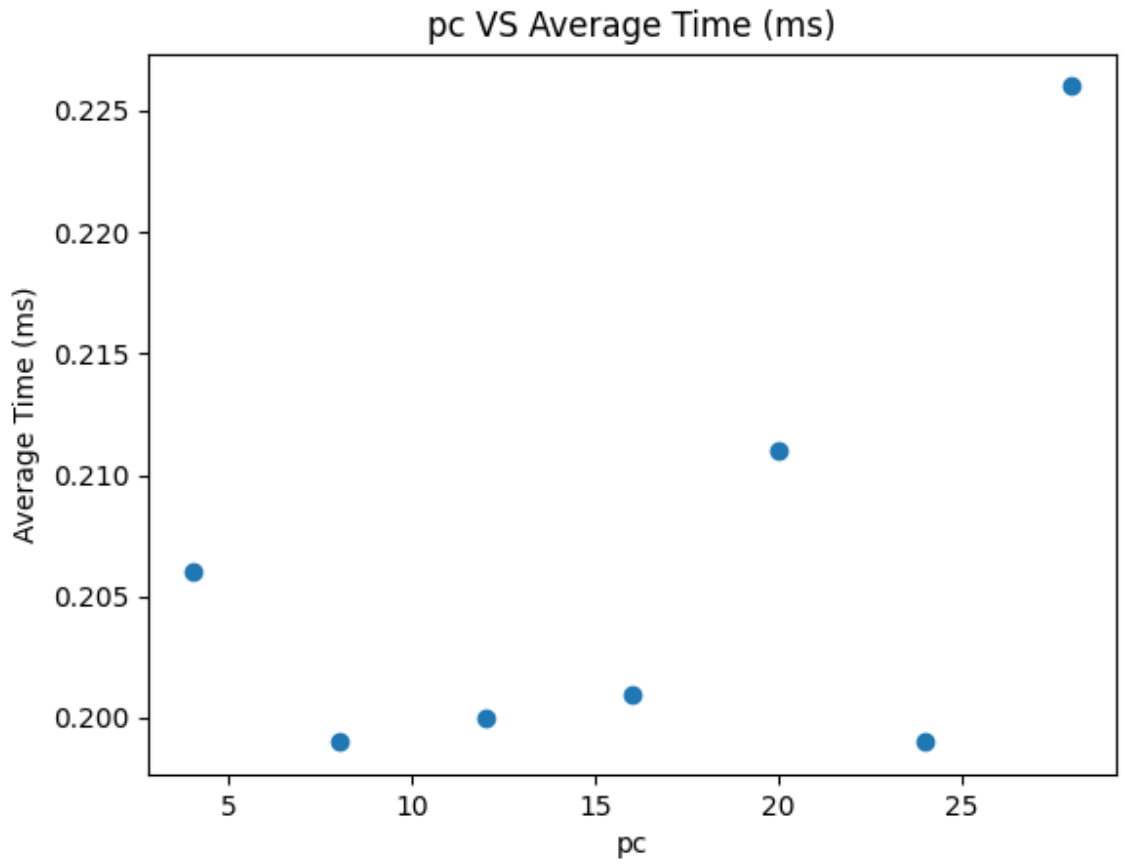

Fig. 4.5 – PC VS Average Time (ms) for PCA on Testing Data

expected to take longer to execute. Thus, we look at other possible values that produced decent accuracy results. We also look at the confusion matrices for 20, 60, 80, 100, and 220 trees to see if the execution times would be faster given that performance deteriorates by an insignificant amount.

Looking at Tables 4.15, 4.16, 4.17, 4.18, we can see that for 240, 260, 280, and 300 trees the confusion matrix and accuracy is identical. However, as the number of trees increase, the execution time increases as well. This indicates that at a certain point, the performance of the model cannot be improved by increasing the number of trees, but the execution time will continue to increase. Thus, if we want to have the best overall performance while being computational efficient, choosing Number of Trees = 240 would be the better choice.

We can also try to decrease the execution times by looking at lower tree values that perform almost as good as 240 trees. As shown in tables 4.10 and 4.11, 20 and 60 trees have the same accuracy and confusion matrix. Upon further inspection, 100 trees has the same accuracy as 20 and 60 trees, but has a better confusion matrix (Table 4.13. This confusion matrix shows no bias, while 20 and 60 trees show that there is some bias towards the ST class. If we compare the confusion matrix for 100 trees (Table 4.13) and 200 trees (Table 4.3), their results is also the same. Similarly for 80 trees (Table 4.12) and 220 trees (Table 4.14), the accuracy and confusion matrix for the 2 are identical and show bias towards the ST class. This again shows that at a certain point, increasing the number of trees would not improve performance and could potentially slow down the algorithm instead.

The general trend for execution times regardless of performance is that the more trees there are included in the model, the longer it takes to execute as shown in Table 4.19 and 4.19. However, looking at the trend in performance, we know that there is a certain point where the performance will not improve by increasing the trees. Thus, we can look at the lower number of trees and see if we can improve its performance in the next grid search. We will continue the grid search using 100 and 240 trees since these 2 values had the best accuracy and confusion matrix while having a low execution time.

Number of Trees	Accuracy
20	0.827
40	0.827
60	0.827
80	0.846
100	0.827
120	0.808
140	0.827
160	0.827
180	0.827
200	0.827
220	0.846
240	0.865
260	0.865
280	0.865
300	0.865

Table 4.9 – Grid Search for Optimal Number of Trees (PC = 4; Min. Sample Leaf = 5)

Class	TPR	FPR	TNR	FNR
AS	0.783	0.138	0.862	0.217
ST	0.862	0.217	0.783	0.138

Table 4.10 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.827, PC = 4, Number of Trees = 20, Min. Sample Leaf Size = 5,)

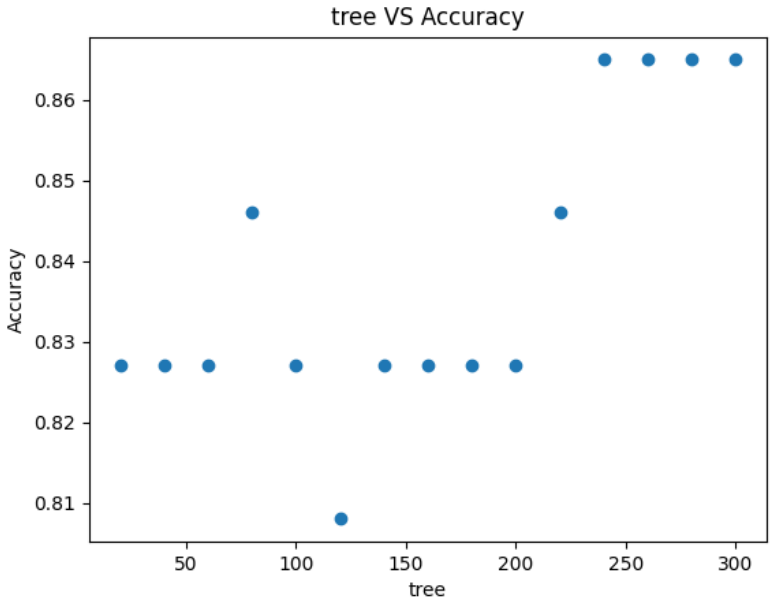

Fig. 4.6 – Trees vs Accuracy

Class	TPR	FPR	TNR	FNR
AS	0.783	0.138	0.862	0.217
ST	0.862	0.217	0.783	0.138

Table 4.11 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.827, PC = 4, Number of Trees = 60, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.172	0.828	0.130
ST	0.828	0.130	0.870	0.172

Table 4.12 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.846, PC = 4, Number of Trees = 80, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.826	0.172	0.827	0.174
ST	0.828	0.174	0.826	0.172

Table 4.13 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.827, PC = 4, Number of Trees = 100, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.172	0.828	0.130
ST	0.828	0.130	0.870	0.172

Table 4.14 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.846, PC = 4, Number of Trees = 220, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.138	0.862	0.130
ST	0.862	0.130	0.870	0.138

Table 4.15 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 4, Number of Trees = 240, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.138	0.862	0.130
ST	0.862	0.130	0.870	0.138

Table 4.16 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 4, Number of Trees = 260, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.138	0.862	0.130
ST	0.862	0.130	0.870	0.138

Table 4.17 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 4, Number of Trees = 280, Min. Sample Leaf Size = 5)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.138	0.862	0.130
ST	0.862	0.130	0.870	0.138

Table 4.18 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 4, Number of Trees = 300, Min. Sample Leaf Size = 5)

tree	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
20	175.885	169.522	174.866	179.188	267.942	167.862	206.931	176.435	177.845	179.25	187.573
40	305.77	315.973	309.976	315.386	477.76	303.66	445.687	305.37	318.028	332.994	343.06
60	448.542	454.085	455.318	449.193	656.746	438.366	499.949	463.439	444.827	454.744	476.521
80	585.941	596.906	631.119	594.549	829.339	589.761	591.591	592.409	595.014	836.838	644.347
100	718.231	720.703	1039.351	716.109	927.753	716.515	718.344	729.638	733.238	1048.84	806.872
120	957.292	871.43	1236.256	870.422	850.066	854.853	859.557	1106.824	882.428	1257.756	974.688
140	1475.124	1025.245	1278.842	1026.349	985.898	1006.224	1009	1439.029	1022.805	1110.317	1137.883
160	1609.789	1136.355	1134.245	1159.081	1144.175	1570.611	1135.664	1582.038	1152.826	1161.025	1278.581
180	1306.02	1259.144	1286.215	1621.283	1269.703	1927.096	1291.095	1300.425	1294.822	1309.837	1386.564
200	1395.909	1594.282	1428.807	2010.418	1392.792	1514.487	1437.443	1431.49	1756.709	1418.664	1538.1
220	1569.657	2209.524	1590.991	1630.199	1544.767	1575.431	1803.937	1579.927	2239.64	1559.75	1730.382
240	1695.541	1902.501	1695.96	1670.111	1923.16	1696.959	2389.31	1727.343	1711.262	1672.987	1808.513
260	1788.841	1802.483	2183.423	1837.431	2578.433	1829.595	1981.286	1859.263	1848.839	2387.209	2009.68
280	2352.373	1971.832	2795.654	2000.529	1985.934	1952.545	1998.385	2596.332	1945.519	2581.314	2218.042
300	2771.659	2104.448	2121.961	2145.937	2090.94	2834.96	2134.016	2643.288	2081.095	2135.119	2306.342

Table 4.19 – Grid Search Model Training Execution Time (ms) for Optimal Number of Trees for Random Forest Model (PC = 4; Min Leaf Sample = 5)

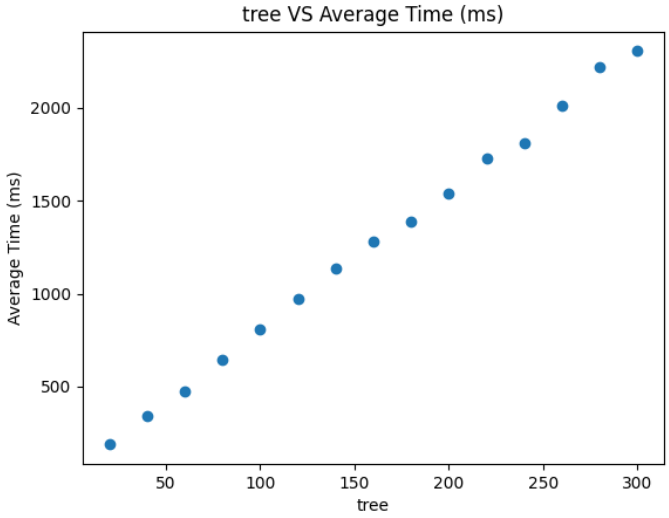

Fig. 4.7 – Tree VS Average Time (ms) for Random Forest Tree Model on Training Data

tree	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
20	1.853	1.808	1.862	1.781	1.912	1.82	1.857	2.015	2.013	1.828	1.875
40	1.984	1.634	1.865	1.782	1.945	1.919	2.062	1.903	1.983	2.709	1.979
60	1.937	2.411	1.732	1.841	1.893	1.785	2.649	1.955	1.901	1.86	1.996
80	2.044	2.066	1.94	1.911	1.88	2.007	1.926	1.803	1.847	6.295	2.372
100	2.021	1.948	2.226	1.898	6.506	1.734	1.822	1.812	1.751	6.522	2.824
120	2.045	2.003	6.396	1.735	1.677	1.884	1.909	1.61	1.846	2.048	2.315
140	6.362	2.066	6.113	1.85	1.769	1.771	1.882	5.937	1.782	2.051	3.158
160	1.996	2.074	1.834	1.829	1.68	1.853	1.837	6.508	2.022	1.892	2.352
180	2.035	2.029	1.869	1.794	2.024	6.494	2.059	1.921	1.974	1.808	2.401
200	2.082	2.168	1.866	6.435	1.739	6.318	1.86	1.873	1.705	1.883	2.793
220	2.112	2.142	1.877	1.866	1.782	2.022	1.828	1.92	6.315	1.792	2.366
240	2.004	6.378	2.019	2.073	1.693	2.105	6.79	2.05	1.998	2.112	2.922
260	2.111	1.943	2.183	1.93	6.881	1.995	2.182	2.129	2.146	2.02	2.552
280	1.792	1.893	2.008	1.831	1.837	1.899	1.85	1.897	1.942	7.013	2.396
300	6.623	1.778	1.733	1.912	1.796	1.866	1.962	6.544	1.843	1.829	2.789

Table 4.21 – Grid Search PCA Training Execution Time (ms) for Optimal Number of Trees for Random Forest Model (PC = 4; Min Leaf Sample = 5)

tree	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
20	2.636	2.845	2.647	3.735	3.613	2.408	3.773	2.807	2.525	2.642	2.963
40	4.545	5.131	4.884	4.479	6.584	6.418	10.799	5.626	4.794	5.486	5.875
60	6.621	7.036	7.128	6.553	10.752	6.724	6.694	7.405	6.48	7.034	7.243
80	8.817	9.521	14.464	10.972	13.165	8.817	9.353	9.321	9.362	11.631	10.542
100	10.789	15.379	14.305	10.392	10.331	11.505	16.222	12.275	11.215	14.843	12.726
120	19.938	14.223	20.627	15.792	11.954	14.112	12.989	17.963	14.327	20.655	16.258
140	19.88	14.376	14.682	15.196	14.695	15.932	15.296	20.675	16.826	16.018	16.358
160	26.658	17.321	18.581	16.001	17.933	25.769	17.551	18.275	18.479	18.675	19.524
180	20.029	26.033	19.939	25.905	18.643	32.384	20.003	18.614	19.314	22.708	22.357
200	20.48	30.789	21.025	29.788	20.921	22.335	23.5	20.726	31.752	21.533	24.285
220	24.701	37.967	25.152	22.003	22.822	24.441	33.665	23.339	37.676	23.945	27.571
240	24.047	26.05	28.363	26.15	39.438	26.596	42.569	25.839	28.62	24.94	29.261
260	28.522	26.493	43.518	27.802	44.308	32.409	26.918	27.608	27.397	39.37	32.434
280	42.566	30.006	29.319	31.152	29.396	31.725	31.341	40.946	29.968	30.976	32.74
300	32.263	31.907	32.3	53.026	31.194	50.345	33.468	38.56	33.265	32.472	36.88

Table 4.20 – Grid Search Model Execution Time (ms) on Testing Data Set for Optimal Number of Trees for Random Forest Model (PC = 4; Min Leaf Sample = 5)

tree	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
20	0.151	0.186	0.191	0.214	0.218	0.191	0.239	0.24	0.159	0.178	0.197
40	0.163	0.2	0.18	0.173	0.234	0.221	0.222	0.183	0.209	0.235	0.202
60	0.19	0.164	0.208	0.167	0.258	0.159	0.153	0.199	0.165	0.168	0.183
80	0.193	0.19	0.255	0.208	0.257	0.164	0.19	0.15	0.169	0.243	0.202
100	0.165	0.254	0.224	0.155	0.175	0.162	0.188	0.16	0.155	0.228	0.187
120	0.25	0.168	0.256	0.163	0.166	0.167	0.172	0.276	0.171	0.294	0.208
140	0.234	0.157	0.19	0.162	0.174	0.167	0.168	0.214	0.192	0.166	0.182
160	0.25	0.157	0.175	0.163	0.163	0.467	0.165	0.173	0.169	0.177	0.206
180	0.18	0.178	0.178	0.213	0.157	0.237	0.195	0.168	0.157	0.167	0.183
200	0.18	0.259	0.168	0.235	0.16	0.17	0.168	0.168	0.257	0.172	0.194
220	0.176	0.269	0.18	0.178	0.153	0.174	0.243	0.192	0.249	0.166	0.198
240	0.17	0.199	0.172	0.151	0.217	0.196	0.267	0.179	0.158	0.157	0.187
260	0.173	0.17	0.27	0.168	0.258	0.169	0.157	0.205	0.167	0.221	0.196
280	0.254	0.173	0.169	0.22	0.173	0.22	0.179	0.227	0.189	0.174	0.198
300	0.176	0.23	0.166	0.252	0.165	0.217	0.177	0.184	0.177	0.181	0.193

Table 4.22 – Grid Search PCA Execution Time (ms) on Testing Data Set for Optimal Number of Trees for Random Forest Model (PC = 4; Min Leaf Sample = 5)

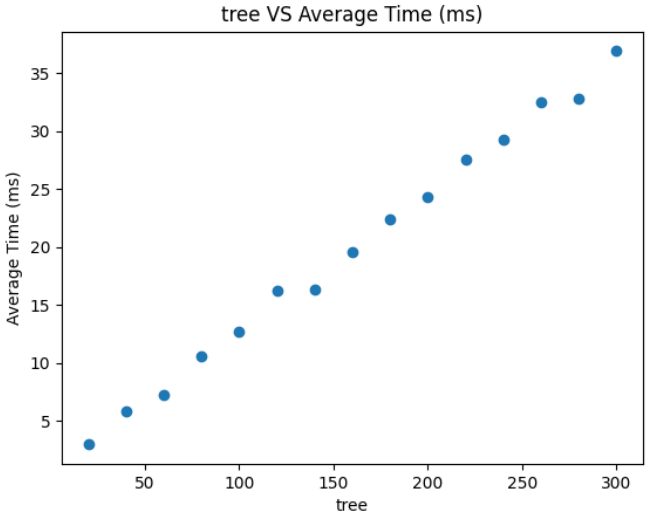

Fig. 4.8 – Tree VS Average Time (ms) for Random Forest Tree Model on Testing Data

4.1.3 Minimum Sample Leaf

We start another grid search for finding the minimum sample leaf size. For this grid search, we looked at the following possible leaf sizes: 2, 3, 4, 5, 6,, 7, 8, 9, and 10. We kept PC = 4, and we looked at performance at 100 and 240 trees to see if we can optimize both performance and execution times.

According to Table 4.27 and 4.28, both values for the tree can produce the same accuracy (0.865), but using different minimum leaf sample sizes. For 100 trees, the highest accuracy was produced when the minimum leaf sample size was set to 4 and for 240 trees, the highest accuracy was produced when the minimum leaf sample size was set to 5. If we look closer at both the confusion matrices at these parameter sets (Table 4.33 and 4.15), we can see they are identical. However, the execution times differ. According to Table 4.23, 4.24, 4.29 and 4.30, 100 trees set at 4 leaves executed faster than 240 trees set at 5 leaves.This could be due to the fact that more branches can be created with a smaller minimum sample leaf size and that accounts for the extra trees that 240 trees has in the decision process. Thus, the optimal performance can be achieved while maintaining a faster execution time through fine tuning the hyperparameters.

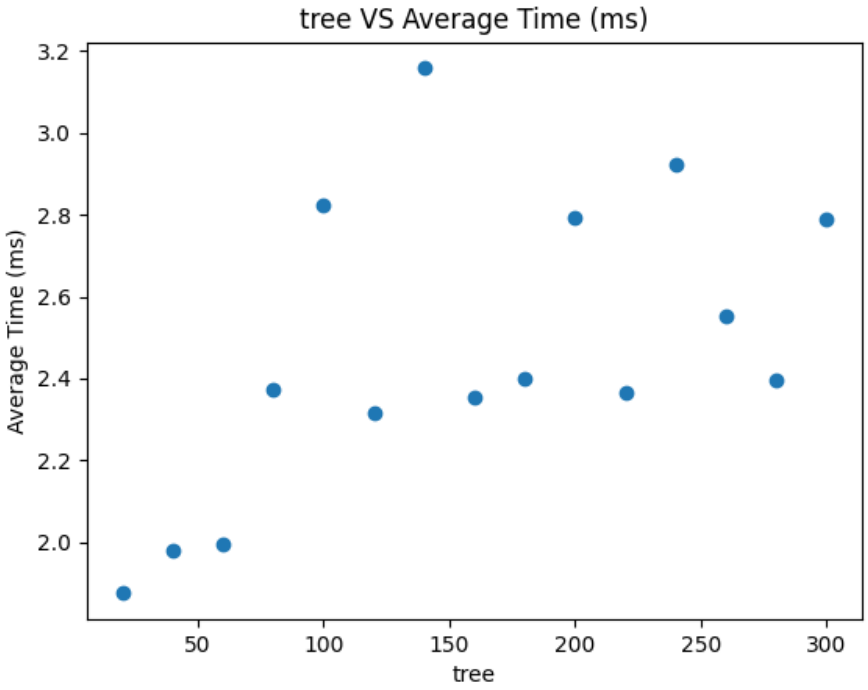

Fig. 4.9 – Tree VS Average Time (ms) for PCA on Training Data

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	820.12	803.58	792.79	770.66	1107.15	779.14	769.24	1122.91	796.5	796.64	855.87
3	765.77	782.11	839.35	746.59	796.23	777.18	757.75	1091.78	767.51	781.31	810.56
4	896.49	751.52	1079.94	734.98	758.41	748.42	770.11	1059.03	759.81	769.04	832.77
5	869.87	735.81	1091.75	728.02	727.42	974.94	731.53	743.89	743.68	754.19	810.11
6	1110.13	750.95	1030.23	708.03	717.59	1015.29	727.67	739.93	780.39	729.96	831.02
7	1263.5	712.04	720.16	705.48	726.87	986.02	710.24	715.62	1007.02	718.83	826.58
8	1249.73	710.43	740.65	718.93	705.18	903.78	703.10	702.28	1013.70	722.69	817.05
9	1079.13	694.2	715.12	1034.12	698.76	723.76	696.73	726.19	985.95	716.63	807.06
10	957.84	692.54	697.34	994.77	686.29	678.67	747.68	696.78	687.48	690.11	752.95

Table 4.23 – Grid Search Model Training Execution Time (ms) for Minimum Number of Leaves for Random Forest Model (Number of Trees = 100; PC = 4)

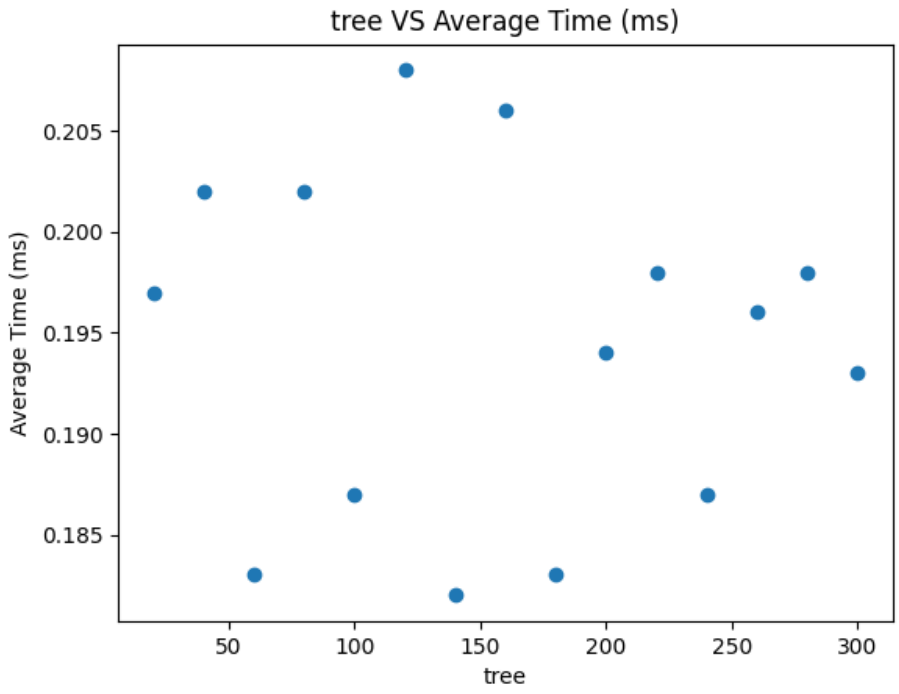

Fig. 4.10 – Tree VS Average Time (ms) for PCA on Testing Data

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	11.49	11.4	11.4	13.22	16.2	11.47	11.08	15.4	11.15	10.96	12.38
3	11.38	11.58	15.27	10.44	11.19	11.99	10.81	15.57	11.04	11.45	12.071
4	15.879	11.833	17.272	11.049	11.22	10.953	11.632	12.131	11.23	11.479	12.468
5	14.908	11.206	15.411	10.153	11.322	14.501	12.063	11.163	12.169	11.419	12.432
6	17.405	10.995	17.897	11.231	10.967	14.375	10.862	11.275	18.68	12.711	13.64
7	25.776	11.809	11.377	11.277	12.586	15.939	10.782	10.846	19.43	10.65	14.047
8	27.928	10.264	11.033	11.072	12.149	12.416	10.81	10.882	20.904	11.579	13.904
9	16.822	12.374	11.946	15.148	10.394	10.756	11.059	11.272	15.518	10.212	12.55
10	17.704	10.75	10.861	17.303	10.493	11.104	14.775	15.304	10.817	10.621	12.973

Table 4.24 – Grid Search Model Execution Time (ms) on Testing Data Set for Minimum Number of Leaves for Random Forest Model (Number of Trees = 100; PC = 4)

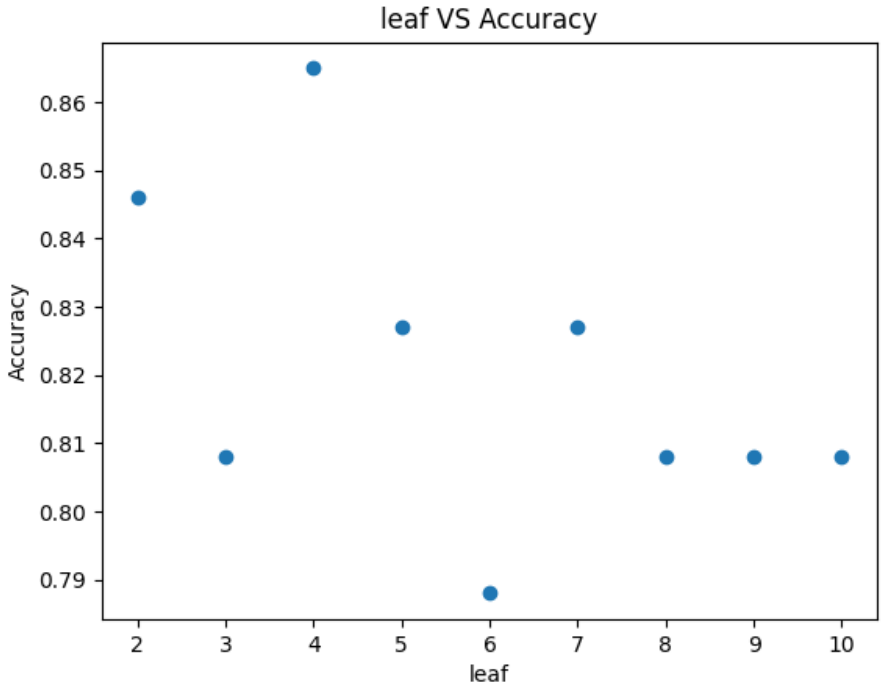

Fig. 4.11 – Minimum Sample Leaf Size VS. Accuracy for Number of Trees = 100

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	1.864	2.032	2.081	1.961	2.21	1.419	2.034	4.789	2.172	2.118	2.268
3	1.725	2.235	2.102	2.042	2.161	2.005	2.171	6.5	2.288	2.041	2.527
4	1.64	2.055	2.021	1.97	2.06	2.154	2.802	6.469	2.112	2.203	2.549
5	5.339	2.105	9.907	2.127	2.097	2.246	2.151	2.098	2.067	2.241	3.238
6	4.81	2.093	6.821	2.028	2.116	2.246	2.097	2.105	1.949	2.251	2.852
7	3.187	2.026	2.093	2.061	2.201	2.109	1.889	2.088	7.012	2.114	2.678
8	2.385	2.145	2.172	2.025	2.086	6.294	2.086	2.042	9.033	2.157	3.242
9	2.261	2.219	2.068	3.22	2.069	2.259	2.084	2.156	2.244	2.06	2.264
10	7.039	2.048	2.146	6.956	2.134	2.139	2.116	2.092	2.274	2.154	3.11

Table 4.25 – Grid Search PCA Training Execution Time (ms) for Minimum Number of Leaves for Random Forest Model (Number of Trees = 100; PC = 4)

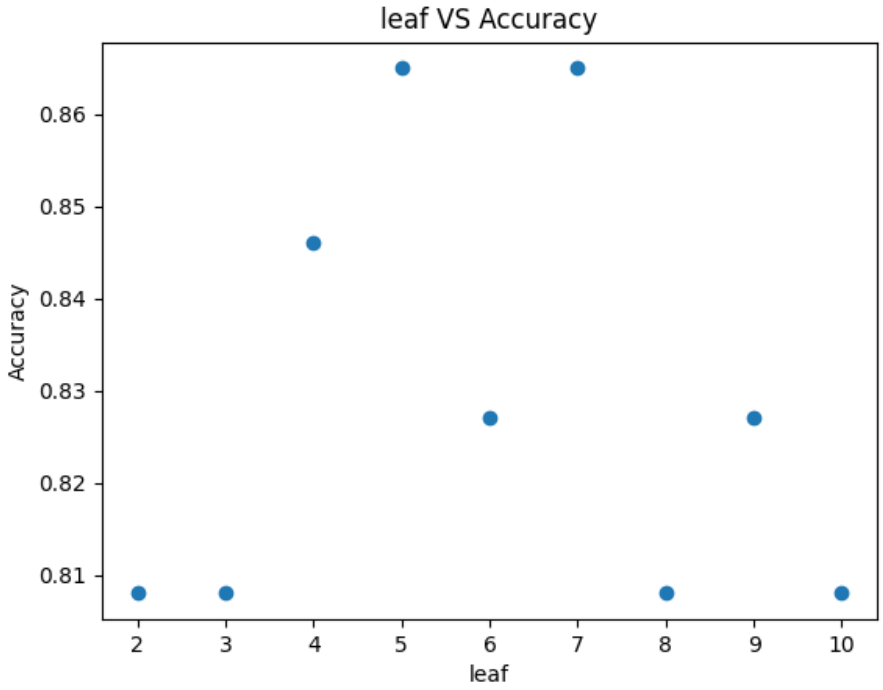

Fig. 4.12 – Minimum Sample Leaf VS. Accuracy Number of Trees = 240

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	0.17	0.186	0.164	0.23	0.233	0.163	0.171	0.233	0.172	0.186	0.191
3	0.202	0.187	0.232	0.169	0.159	0.196	0.188	0.265	0.148	0.157	0.19
4	0.598	0.207	0.241	0.169	0.156	0.163	0.171	0.239	0.197	0.212	0.235
5	0.21	0.175	0.223	0.161	0.181	0.21	0.176	0.176	0.191	0.165	0.187
6	0.228	0.162	0.218	0.172	0.176	0.212	0.157	0.193	0.262	0.176	0.196
7	0.345	0.189	0.174	0.173	0.184	0.219	0.162	0.157	0.227	0.159	0.199
8	0.281	0.169	0.226	0.167	0.214	0.175	0.151	0.165	0.231	0.174	0.195
9	0.272	0.205	0.199	0.253	0.185	0.154	0.197	0.165	0.242	0.202	0.207
10	0.331	0.179	0.207	0.243	0.168	0.155	0.215	0.23	0.157	0.18	0.206

Table 4.26 – Grid Search PCA Execution Time (ms) on Testing Data Set for Minimum Number of Leaves for Random Forest Model (Number of Trees = 100)

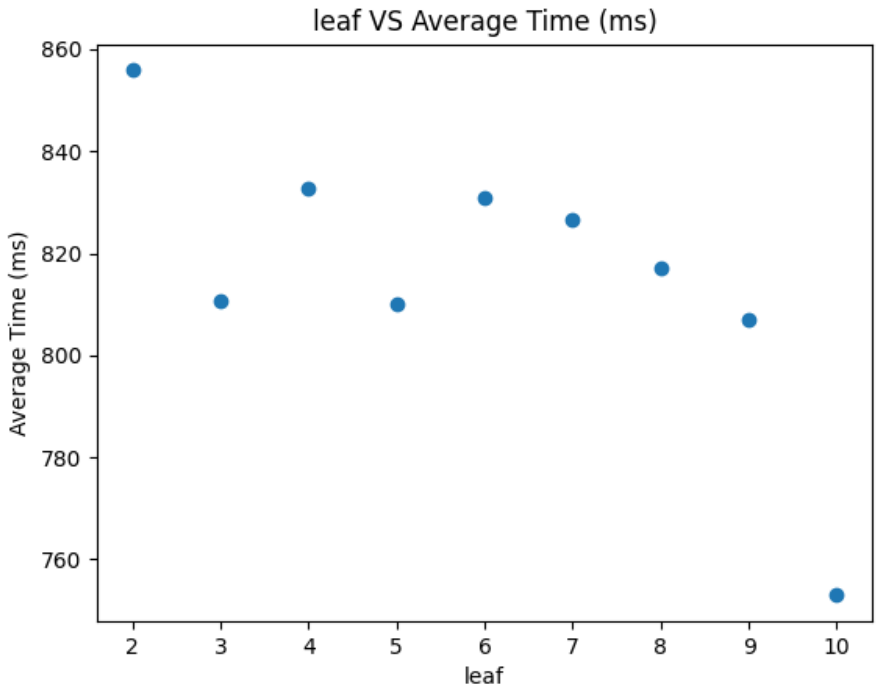

Fig. 4.13 – Leaf VS Average Time (ms) for Random Forest Tree Model on Training Data (for Tree = 100; PC =4)

Min. Leaf Size	Accuracy
2	0.846
3	0.808
4	0.865
5	0.827
6	0.788
7	0.827
8	0.808
9	0.808
10	0.808

Table 4.27 – Grid Search for Optimal Minimum Sample Leaf Size for Number of Trees = 100 (PC = 4)

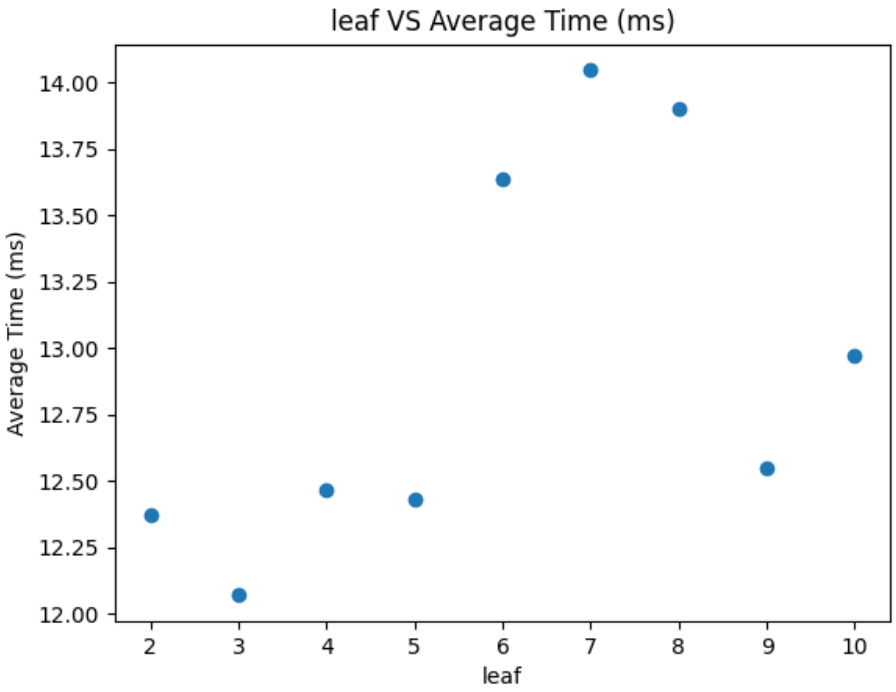

Fig. 4.14 – Leaf VS Average Time (ms) for Random Forest Tree Model on Testing Data (for Tree = 100)

Min. Leaf Size	Accuracy
2	0.808
3	0.808
4	0.846
5	0.865
6	0.827
7	0.865
8	0.808
9	0.827
10	0.808

Table 4.28 – Grid Search for Optimal Minimum Sample Leaf Size for Number of Trees = 240 (PC = 4)

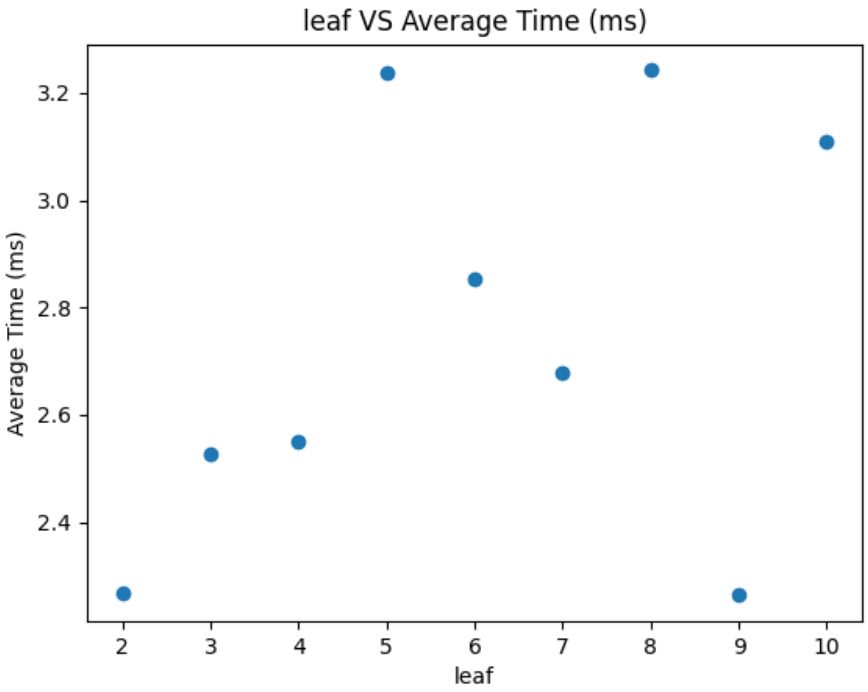

Fig. 4.15 – Leaf VS Average Time (ms) for PCA on Training Data (for Tree = 100)

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	2590.307	1859.567	1832.99	2027.184	1823.246	2221.376	1837.944	1773.652	2541.457	1769.339	2027.706
3	1818.666	1773.531	2452.889	1793.56	1770.699	2526.279	1801.92	1750.951	2210.309	1749.433	1964.824
4	1777.52	1714.426	2139.81	1730.671	1764.101	1721.064	1734.329	2304.586	1702.273	1724.235	1831.302
5	1752.83	2101.131	1700.99	1697.423	2502.036	1691.819	1701.668	2284.312	1754.601	1659.782	1884.659
6	1673.419	2388.607	1690.436	1657.775	1899.708	1704.827	2001.897	1673.035	1704.228	2251.877	1864.581
7	1684.882	1641.892	1629.123	2216.982	1663.684	1669.939	2389.227	1596.02	1673.122	2173.253	1833.812
8	2363.167	1584.835	1636.11	2255.96	1647.066	1625.216	1710.829	1618.1	1959.062	1664.988	1806.533
9	1880.157	1622.017	1770.947	1613.214	1620.022	2321.453	1634.007	1599.685	2292.008	1648.163	1800.167
10	1606.194	1612.474	2293.572	1600.005	1615.525	2051.74	1560.119	1587.157	1593.636	1608.186	1712.861

Table 4.29 – Grid Search Model Training Execution Time (ms) for Minimum Number of Leaves for Random Forest Model (Number of Trees = 240; PC = 4)

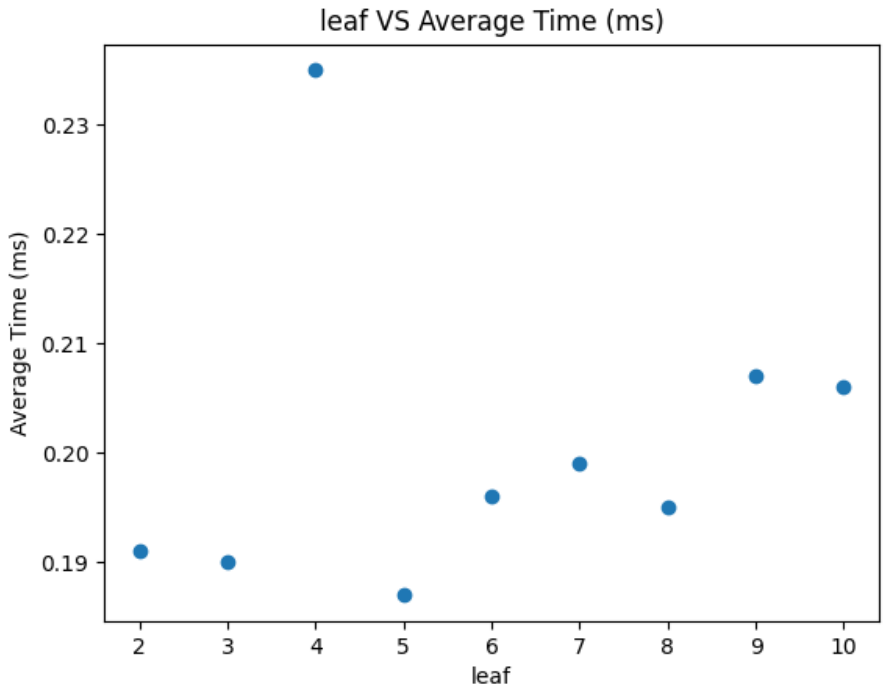

Fig. 4.16 – Leaf VS Average Time (ms) for PCA on Testing Data (for Tree = 100)

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	26.575	36.24	28.438	27.575	29.094	39.82	26.555	25.642	35.513	27.841	30.329
3	26.447	29.321	37.564	27.532	26.284	43.764	28.582	25.64	26.095	26.195	29.742
4	28.924	25.392	24.985	24.325	44.391	25.68	25.736	38.225	24.435	26.241	28.833
5	26.447	35.345	26.372	25.978	40.701	28.928	25.19	24.901	25.149	27.769	28.678
6	24.68	27.474	24.864	26.35	25.965	26.307	39.712	28.595	33.348	34.783	29.208
7	40.014	24.84	25.587	33.499	30.58	27.784	44.989	22.744	37.492	27.328	31.486
8	35.21	26.585	25.422	26.847	24.468	25.87	25.238	25.531	37.959	25.658	27.879
9	25.961	25.214	37.948	24.869	29.035	47.154	26.847	25.141	34.122	35.443	31.173
10	24.711	24.316	33.965	24.813	24.872	24.356	24.384	25.996	23.997	24.327	25.574

Table 4.30 – Grid Search Model Execution Time (ms) on Testing Data Set for Minimum Number of Leaves for Random Forest Model (Number of Trees = 240; PC = 4)

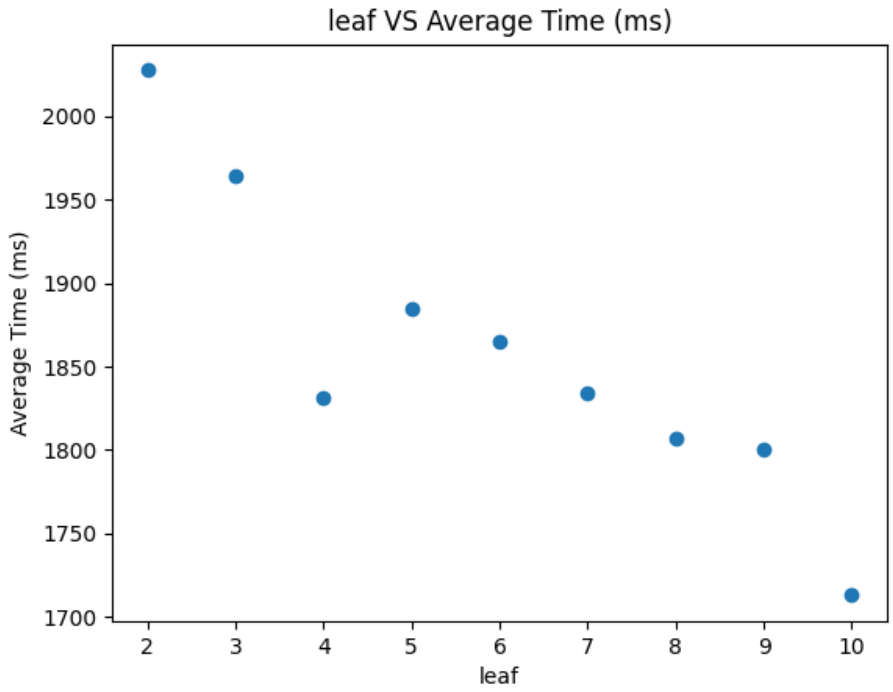

Fig. 4.17 – Leaf VS Average Time (ms) for Random Forest Tree Model on Training Data (for Tree = 240)

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	6.746	1.948	2.177	6.776	2.071	2.183	2.002	1.905	2.162	2.049	3.002
3	2.146	1.942	2.128	2.167	2.226	2.071	2.113	1.97	4.153	1.817	2.273
4	2.052	2.03	2.183	2.213	2.028	2.039	2.112	2.086	2.085	1.966	2.079
5	2.04	2.026	2.264	2.123	2.18	2.089	1.968	2.15	2.114	1.992	2.095
6	2.223	7.072	2.263	2.063	7.653	2.081	2.056	2.17	2.017	2.177	3.178
7	2.086	2.025	2.048	2.126	2.124	1.962	3.835	2.019	2.053	2.162	2.244
8	7.716	2.122	1.993	2.282	2.05	2.11	6.597	1.868	2.045	2.091	3.087
9	4.096	2.258	2.066	2.081	2.182	6.856	2.012	2.068	10.059	2.166	3.584
10	2.026	1.966	2.307	2.079	2.115	2.168	1.902	2.951	2.125	2.044	2.168

Table 4.31 – Grid Search PCA Training Execution Time (ms) for Minimum Number of Leaves for Random Forest Model (Number of Trees = 240; PC = 4)

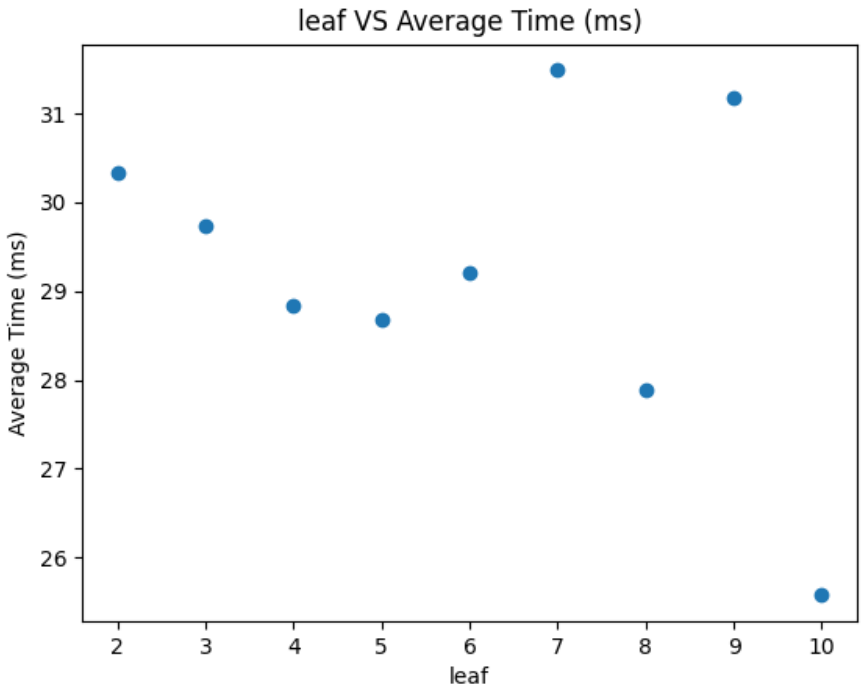

Fig. 4.18 – Leaf VS Average Time (ms) for Random Forest Tree Model on Testing Data (for Tree = 240)

leaf	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
2	0.172	0.164	0.202	0.171	0.186	0.222	0.174	0.169	0.218	0.178	0.186
3	0.168	0.206	0.222	0.164	0.192	0.299	0.155	0.173	0.182	0.206	0.197
4	0.166	0.169	0.192	0.185	0.293	0.194	0.173	0.266	0.185	0.181	0.2
5	0.175	0.236	0.202	0.175	0.258	0.185	0.169	0.175	0.159	0.166	0.19
6	0.155	0.176	0.169	0.163	0.162	0.193	0.233	0.192	0.199	0.218	0.186
7	0.211	0.196	0.205	0.229	0.177	0.183	0.283	0.155	0.164	0.19	0.199
8	0.232	0.18	0.168	0.182	0.173	0.157	0.166	0.169	0.257	0.184	0.187
9	0.168	0.163	0.229	0.164	0.205	0.289	0.175	0.153	0.219	0.182	0.195
10	0.18	0.157	0.2	0.182	0.189	0.196	0.174	0.167	0.158	0.174	0.178

Table 4.32 – Grid Search PCA Execution Time (ms) on Testing Data Set for Minimum Number of Leaves for Random Forest Model (Number of Trees = 240; PC = 4)

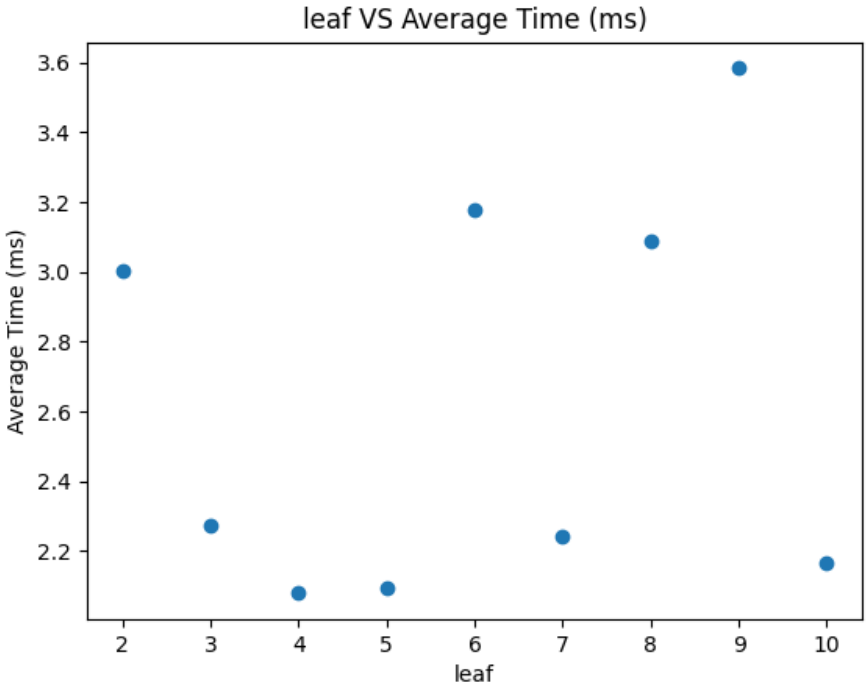

Fig. 4.19 – Leaf VS Average Time (ms) for PCA on Training Data (for Tree = 240)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.138	0.862	0.130
ST	0.862	0.130	0.870	0.138

Table 4.33 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.865, PC = 4, Number of Trees = 100, Min. Sample Leaf Size = 4)

Class	TPR	FPR	TNR	FNR
AS	0.870	0.172	0.828	0.130
ST	0.828	0.130	0.870	0.172

Table 4.34 – Confusion Matrix Random Forest Tree after PCA (Accuracy = 0.846, PC = 4, Number of Trees = 240, Min. Sample Leaf Size = 4)

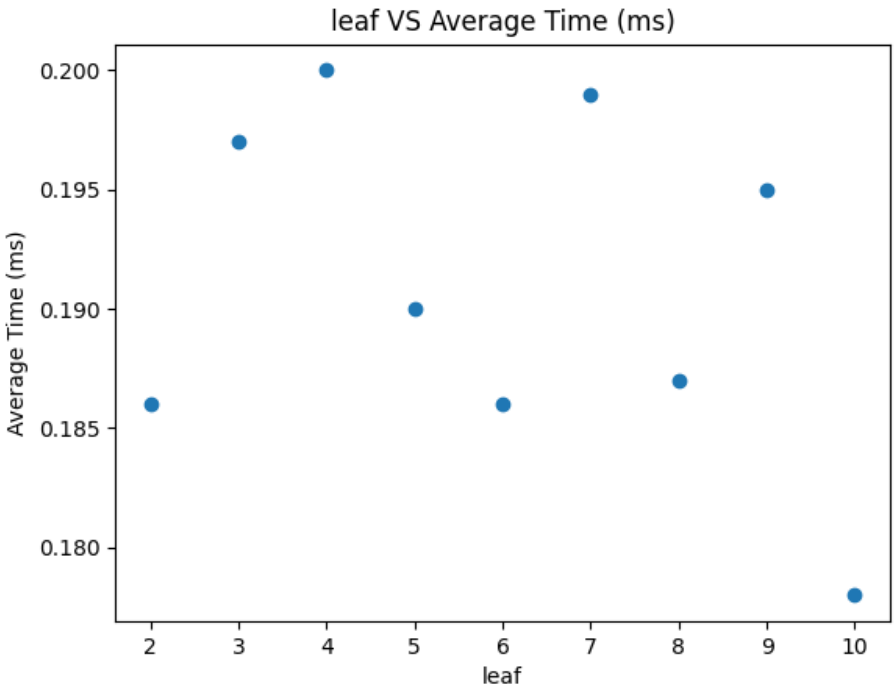

Fig. 4.20 – Leaf VS Average Time (ms) for PCA on Testing Data (for Tree = 240)

4.1.4 Max Depth

We finish up the grid search by finding the optimal max depth value. We had set the max depth value to its default value, which is to continue running until the leaves are pure. This can be computational costly if the trees end up having too many levels [18], thus limiting the sizes of the trees by fine tuning the mx depth parameter can decrease execution times. To start off, we find the accuracy of the following values for the max depth: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. According to Table 4.35, the max depth that gave us the highest accuracy was 20. Looking at the execution times, a max depth of 20 tends to take longer than the other max depth values including the default value, however the difference between the times is small and we chose to stick with a max depth of 20 since it offers the best performance and combats overfitting with little cost in terms execution times.

Max Depth	Accuracy
10	0.827
11	0.788
12	0.827
13	0.846
14	0.827
15	0.808
16	0.808
17	0.846
18	0.846
19	0.846
20	0.865

Table 4.35 – Grid Search for Optimal Max Depth (Number of Trees = 100; PC = 4; Min. Sample Leaf Size = 4))

depth	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Trial 6**	**Trial 7**	**Trial 8**	**Trial 9**	**Trial 10**	**Average**
10	799.41	805.308	754.77	1137.594	979.342	782.834	748.345	774.022	794.495	755.503	833.162
11	831.774	841.092	788.939	1177.753	1171.694	800.02	782.921	827.363	803.92	789.909	881.539
12	832.673	819.224	800.861	849.508	1218.522	1185.249	812.735	863.449	808.912	803.03	899.416
13	838.492	870.748	829.602	844.288	937.538	1231.574	973.17	824.197	830.995	822.32	900.292
14	842.967	829.764	842.167	848.22	829.983	1202.896	1189.993	844.138	830.211	830.203	909.054
15	858.877	852.631	847.567	832.941	848.946	902.117	1193.488	1167.823	838.591	870.073	921.305
16	954.787	884.525	840.546	860.185	831.354	864.235	1113.057	1239.265	861.739	881.33	933.102
17	1299.627	881.253	855.996	842.671	865.155	861.611	843.458	1192.163	1248.616	840.048	973.06
18	1274.936	1245.802	875.989	836.992	861.835	863.149	867.188	889.99	1253.195	1034.745	1000.382
19	1126.781	1235.126	877.949	874.885	854.519	850.86	872.573	862.021	1198.836	1209.669	996.322
20	869.907	1272.559	1285.752	870.843	846.494	851.468	842.382	851.978	858.32	1199.903	974.961

Table 4.36 – Grid Search Model Training Execution Time (ms) for Max Depth for Random Forest Model (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

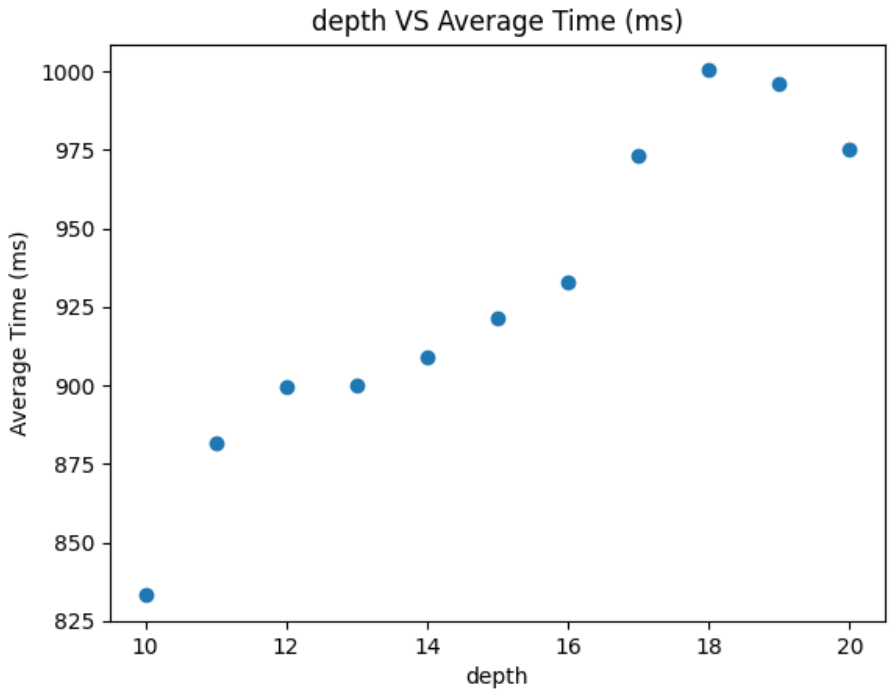

Fig. 4.21 – Depth VS Average Time (ms) for Random Forest Tree Model on Training Data (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

depth	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
10	6.977	9.415	6.265	9.911	11.131	9.809	5.983	6.92	6.283	7.207	7.99
11	7.856	6.563	9.803	6.892	10.048	6.789	7.036	7.761	7.302	7.442	7.749
12	6.715	7.07	6.435	12.512	11.074	10.624	7.167	8.939	6.212	7.283	8.403
13	7.269	10.6	10.072	9.022	8.3	12.183	12.906	6.803	8.949	6.46	9.256
14	9.812	6.95	6.752	7.161	6.9	10.253	9.809	7.023	7.551	7.136	7.935
15	7.814	7.73	8.229	7.286	7.384	7.675	12.046	13.029	7.117	7.009	8.532
16	12.78	7.477	6.736	7.042	12.47	7.877	6.964	11.429	13.4	7.902	9.408
17	12.13	6.887	7.828	6.33	7.262	6.963	8.17	11.329	11.796	6.807	8.55
18	12.641	11.143	7.602	8.28	7.178	7.747	7.376	7.502	10.319	11.367	9.116
19	7.489	10.483	14.429	6.507	8.694	6.685	8.117	6.878	7.882	10.393	8.756
20	7.421	16.207	11.139	7.266	6.393	6.995	6.869	7.593	6.704	10.605	8.719

Table 4.37 – Grid Search Model Execution Time (ms) on Testing Data Set for Max Depth for Random Forest Model (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

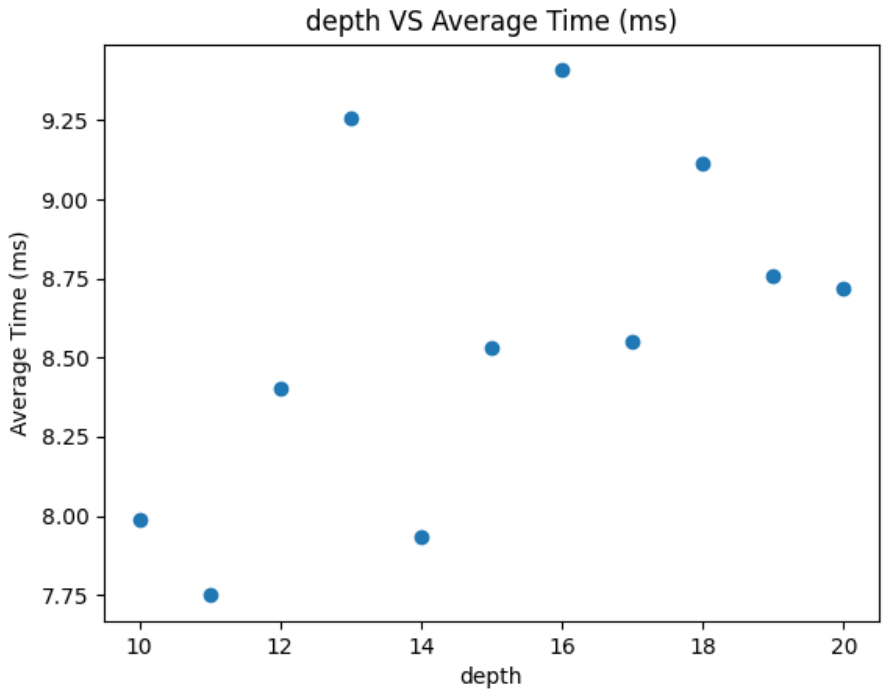

Fig. 4.22 – Depth VS Average Time (ms) for Random Forest Tree Model on Testing Data (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

depth	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
10	0.279	0.514	0.267	0.432	0.583	0.355	0.241	0.287	0.333	0.293	0.358
11	0.412	0.303	0.315	0.312	0.448	0.243	0.241	0.336	0.294	0.229	0.313
12	0.258	0.3	0.288	0.408	0.494	0.432	0.423	0.358	0.298	0.297	0.356
13	0.248	0.4	0.28	0.55	0.351	0.408	0.537	0.252	0.371	0.244	0.364
14	0.34	0.293	0.422	0.276	0.33	0.505	0.417	0.311	0.285	0.303	0.348
15	0.435	0.364	0.307	0.326	0.345	0.354	0.486	0.528	0.259	0.279	0.368
16	0.46	0.375	0.308	0.272	0.298	0.291	0.297	0.641	0.564	0.496	0.4
17	0.42	0.317	0.251	0.245	0.332	0.369	0.348	0.494	0.453	0.26	0.349
18	0.535	0.5	0.318	0.307	0.282	0.319	0.538	0.273	0.502	0.492	0.407
19	0.303	0.458	0.393	0.294	0.327	0.381	0.266	0.338	0.391	0.411	0.356
20	0.319	0.654	0.405	0.262	0.26	0.268	0.587	0.338	0.314	0.455	0.386

Table 4.38 – Grid Search PCA Training Execution Time (ms) for Max Depth for Random Forest Model (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

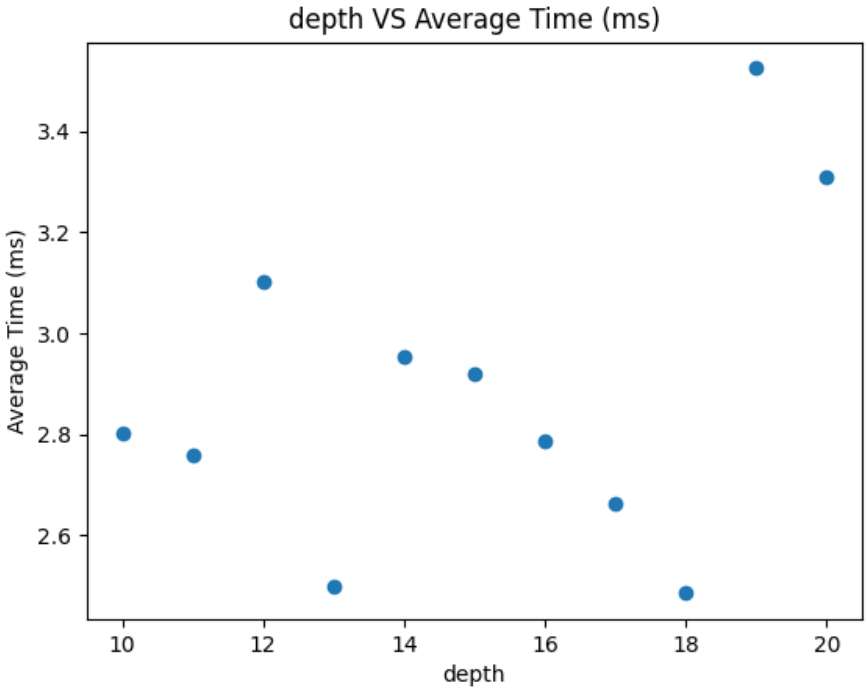

Fig. 4.23 – Depth VS Average Time (ms) for PCA on Training Data ((Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

depth	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
10	0.279	0.514	0.267	0.432	0.583	0.355	0.241	0.287	0.333	0.293	0.358
11	0.412	0.303	0.315	0.312	0.448	0.243	0.241	0.336	0.294	0.229	0.313
12	0.258	0.3	0.288	0.408	0.494	0.432	0.423	0.358	0.298	0.297	0.356
13	0.248	0.4	0.28	0.55	0.351	0.408	0.537	0.252	0.371	0.244	0.364
14	0.34	0.293	0.422	0.276	0.33	0.505	0.417	0.311	0.285	0.303	0.348
15	0.435	0.364	0.307	0.326	0.345	0.354	0.486	0.528	0.259	0.279	0.368
16	0.46	0.375	0.308	0.272	0.298	0.291	0.297	0.641	0.564	0.496	0.4
17	0.42	0.317	0.251	0.245	0.332	0.369	0.348	0.494	0.453	0.26	0.349
18	0.535	0.5	0.318	0.307	0.282	0.319	0.538	0.273	0.502	0.492	0.407
19	0.303	0.458	0.393	0.294	0.327	0.381	0.266	0.338	0.391	0.411	0.356
20	0.319	0.654	0.405	0.262	0.26	0.268	0.587	0.338	0.314	0.455	0.386

Table 4.39 – Grid Search PCA Execution Time (ms) on Testing Data Set for Max Depth for Random Forest Model (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

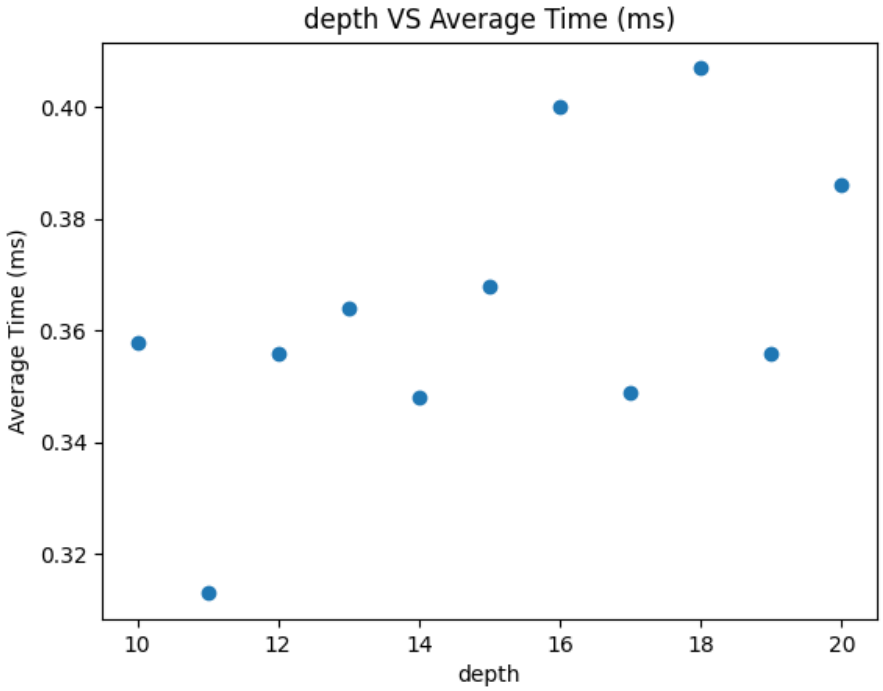

Fig. 4.24 – Depth VS Average Time (ms) for PCA on Testing Data (Number of Trees = 100; PC = 4; Min. Sample Leaf = 4)

After conducting the grid search we decided that the optimal parameters are: 4 PC, 100 decision trees, a minimum sample leaf of 4 and a max depth of 20. We have seen that increasing these parameters can result in a better performance, but the extend to which it has to be increased to see a great improvement in performance causes the model to run significantly longer. Thus we choose the set of parameters that had the fastest execution time with the best performance.

4.2 Support Vector Machine

To find the optimal parameter set for SVM, we repeat a similar process as we did for the Random Forest Tree classifier. We start off by finding the optimal PC and keeping the other parameters constant. The kernel is set to "rbf" and gamma is set to "auto". As seen in Table 4.40, the confusion matrix performance is extremely poor. The TPR for AS and ST is

extremely bias and shows that the model tends to classify videos as ST, resulting in a higher TPR for ST and a low TPR for AS. Moreover, the FPR for ST is higher than we would like, further demonstrating that the model would classify more videos as ST. Thus, we need to implement PCA to try to improve the model's performance. After implementing PCA and determining the optimal PC, we found the optimal kernel and its parameters before finding the optimal coefficient. Again, we run 10 trials of each potential parameter set to see and compare the average execution times.

Class	TPR	FPR	TNR	FNR
AS	0.478	0.138	0.826	0.522
ST	0.862	0.521	0.478	0.138

Table 4.40 – Confusion Matrix for SVM before PCA (Accuracy = 0.69, kernel = rbf, gamma = auto)

4.2.1 PC Grid search

We started the PC grid search by examining the accuracy of each potential PC while the model's kernel parameter was set to "rbf" and the model's gamma parameter was set to "auto". Preliminary analysis show that PC = 12, 20 and 24 produced the highest accuracy (Table 4.41) of 0.808. However, we also examined the confusion matrices for PC = 8 and 16 to see whether its confusion matrix performs better than the higher valued PCs.

The confusion matrices for PC = 8 and 16 (Table 4.42 and 4.44) show poor performance in the TPR for the ST class, which indicates there is a significant bias towards the AS class and exhibits the model's tendency to classify a video as AS. The confusion matrix for PC = 12 (Table 4.43 is only slightly better, but there is a bias towards the ST class as indicated by the higher TPR for ST. As seen in Table 4.45 and 4.46, the confusion matrices for PC = 20 and 24 are identical, however out of all the potential options for the optimal PC we analyzed, these 2 show the least amount of bias while maintaining relatively fast execution times. We chose PC = 20 to continue with the other optimizing grid searches because it has the highest

accuracy, shows the least amount of bias in its confusion matrix, and has a relatively fast execution time (Table 4.53, 4.47, 4.54 and 4.48). Again, we see that the average time for the lower PCs is slightly slower than some of the higher PCs. This could be due to the fact that the testing data set is so small that there is small variations in execution times due to the resources available on Google Colab Pro+. The difference is very minuscule in some cases, but overall, PC = 20 offers the fastest execution time with the best performance in terms of accuracy and its confusions matrix.

PC	Accuracy
4	0.596
8	0.769
12	0.808
16	0.788
20	0.808
24	0.808
28	0.788

Table 4.41 – Grid Search for Optimal PC for SVM Model (kernel = rbf, gamma = auto)

Class	TPR	FPR	TNR	FNR
AS	0.609	0.103	0.897	0.391
ST	0.897	0.391	0.609	0.103

Table 4.42 – Confusion Matrix for SVM after PCA (Accuracy = 0.769, PC = 8, kernel = rbf, gamma = auto)

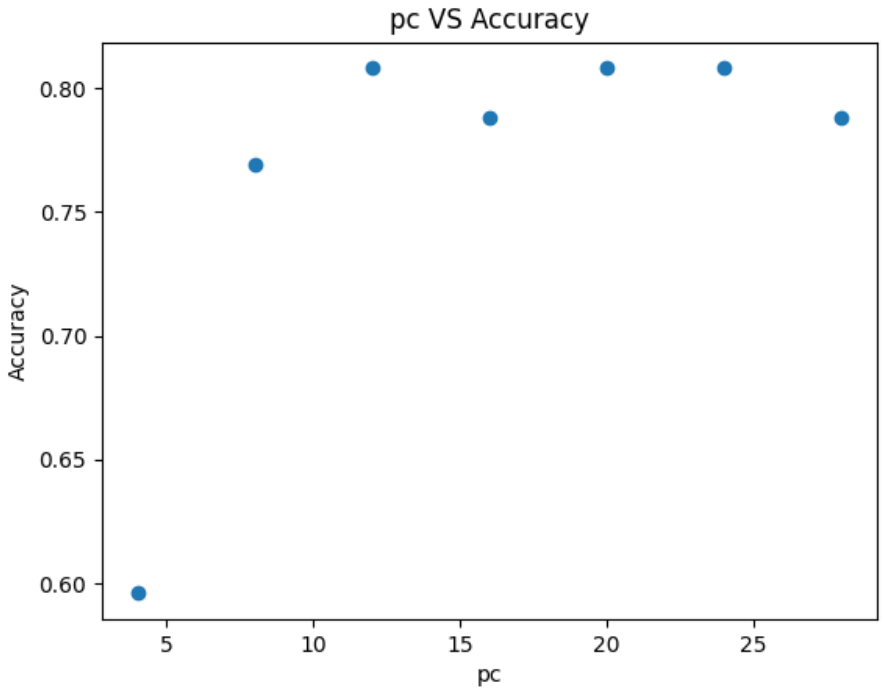

Fig. 4.25 – PC VS. Accuracy for SVM

Class	TPR	FPR	TNR	FNR
AS	0.700	0.103	0.897	0.304
ST	0.897	0.304	0.700	0.103

Table 4.43 – Confusion Matrix for SVM after PCA (Accuracy = 0.808, PC = 12, kernel = rbf, gamma = auto)

Class	TPR	FPR	TNR	FNR
AS	0.696	0.138	0.862	0.304
ST	0.862	0.304	0.696	0.138

Table 4.44 – Confusion Matrix for SVM after PCA (Accuracy = 0.788, PC = 16, kernel = rbf, gamma = auto)

Class	TPR	FPR	TNR	FNR
AS	0.739	0.138	0.862	0.261
ST	0.862	0.261	0.739	0.138

Table 4.45 – Confusion Matrix for SVM after PCA (Accuracy = 0.808, PC = 20, kernel = rbf, gamma = auto)

Class	TPR	FPR	TNR	FNR
AS	0.739	0.138	0.862	0.261
ST	0.862	0.261	0.739	0.138

Table 4.46 – Confusion Matrix for SVM after PCA (Accuracy = 0.808, PC = 24, kernel = rbf, gamma = auto)

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	588.334	570.108	569.499	566.198	926.769	595.809	553.196	965.701	556.671	550.601	644.289
8	547.007	553.351	633.063	536.553	554.558	542.253	553.588	701.63	540.555	541.412	570.397
12	551.804	557.868	932.014	547.318	546.564	884.045	557.257	558.111	563.442	536.72	623.514
16	506.325	515.507	802.58	491.359	490.408	711.686	511.18	504.546	760.883	502.805	579.728
20	532.371	548.698	521.889	521.399	518.283	852.292	538.556	532.551	839.261	535.783	594.108
24	934.011	563.337	567.735	713.126	540.756	540.176	551.112	554.802	630.044	544.594	613.969
28	950.662	593.515	586.539	914.151	564.562	564.704	894.41	567.075	586.051	575.864	679.753

Table 4.47 – Grid Search Model Training Execution Time (ms) for Optimal PC for SVM (kernel = rbf, gamma = auto)

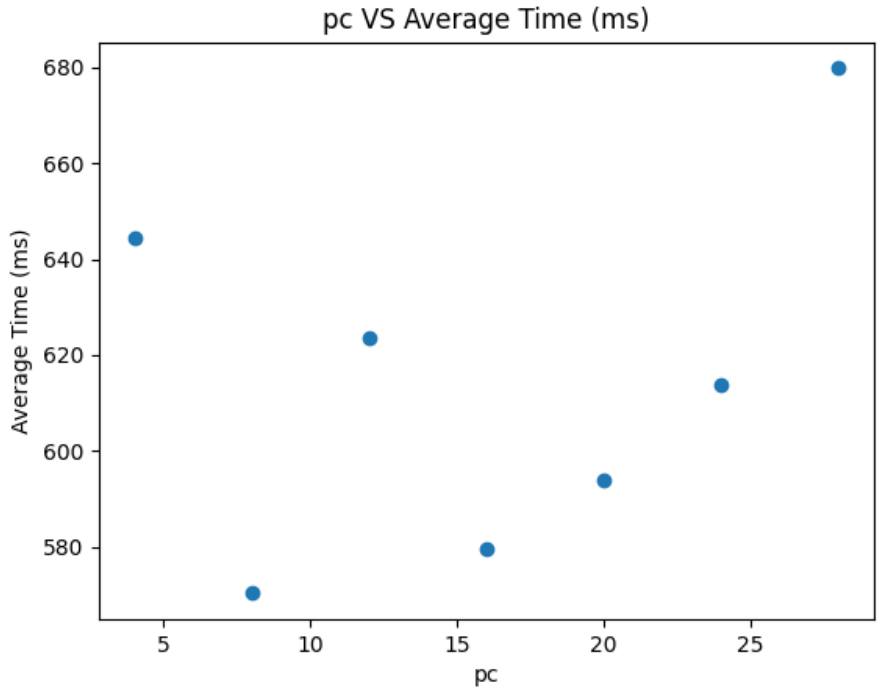

Fig. 4.26 – PC VS Average Time (ms) for SVM on Training Data

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	8.687	7.893	7.942	7.884	8.044	7.854	8.842	15.511	8.146	8.029	8.883
8	10.277	8.816	16.844	7.891	8.246	7.843	7.811	7.927	8.033	7.887	9.158
12	7.44	7.511	15.681	7.515	7.582	15.177	7.684	7.482	7.414	7.467	9.095
16	6.898	6.352	12.529	6.469	6.437	13.221	6.684	6.39	12.706	6.762	8.445
20	14.708	7.179	6.997	7.313	7.011	7.343	7.041	6.923	13.829	6.93	8.527
24	14.254	10.302	7.978	14.385	7.416	7.237	7.283	7.461	7.361	8.677	9.235
28	15.277	7.769	9.045	15.668	8.093	7.751	15.193	7.823	8.213	8.384	10.322

Table 4.48 – Grid Search Model Execution Time (ms) on Testing Data Set for Optimal PC for SVM (kernel = rbf, gamma = auto)

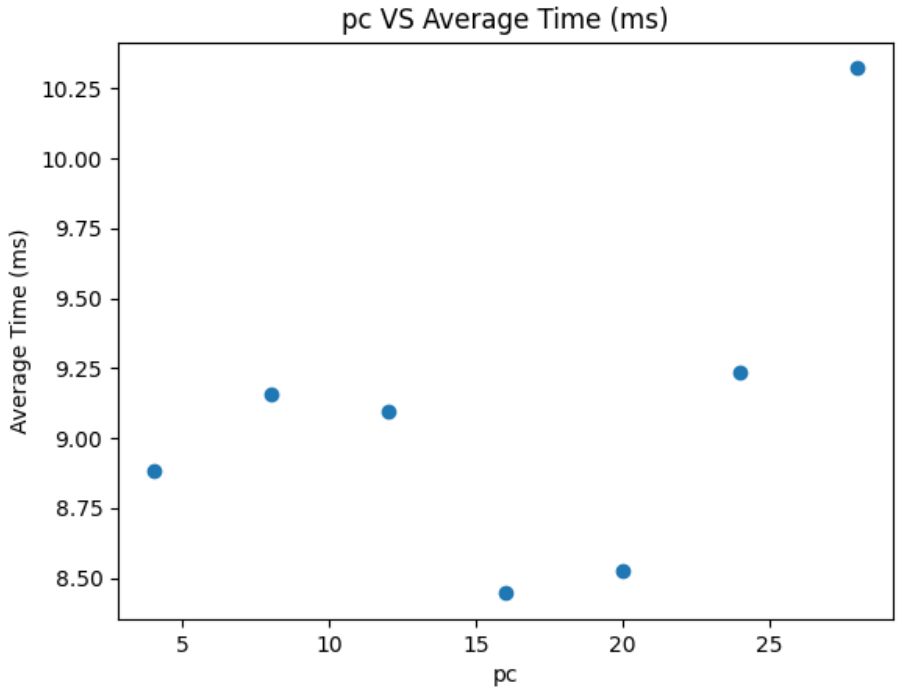

Fig. 4.27 – PC VS Average Time (ms) for SVM on Testing Data

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	49.2	22.217	20.041	20.082	36.229	24.075	20.699	40.509	20.46	38.022	29.153
8	22.973	28.24	20.512	23.038	22.565	19.749	20.013	34.243	19.971	20.411	23.172
12	24.113	23.023	35.058	22.958	41.348	40.617	23.438	19.635	20.435	19.671	27.03
16	40.264	19.952	34.674	20.042	20.901	47.282	21.244	23.782	35.854	20.113	28.411
20	19.749	20.81	21.035	19.944	19.913	36.742	23.249	20.703	35.394	19.848	23.739
24	36.35	21.609	21.86	19.956	20.477	19.822	20.65	20.464	45.254	19.901	24.634
28	35.689	23.917	20.064	50.357	20.469	20.063	22.164	21.815	20.254	20.707	25.55

Table 4.49 – Grid Search PCA Training Execution Time (ms) for Optimal PC for SVM (kernel = rbf, gamma = auto)

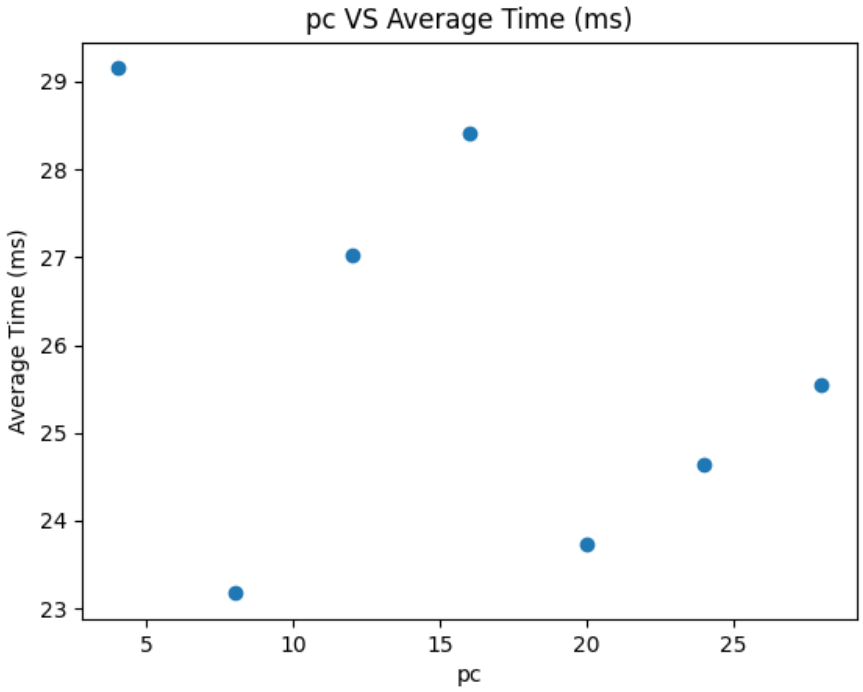

Fig. 4.28 – PC VS Average Time (ms) for PCA on Training Data

pc	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
4	6.244	2.094	1.906	1.965	1.939	1.921	1.87	3.152	2.035	1.889	2.502
8	2.666	1.898	2.991	1.861	1.895	1.941	1.857	1.896	2.253	1.871	2.113
12	1.903	1.875	2.821	1.978	1.933	2.755	1.85	1.906	1.864	1.887	2.077
16	1.985	1.903	2.818	1.91	1.941	2.914	2.293	2.034	2.769	2.825	2.339
20	2.534	1.866	2.066	2.01	1.869	2.789	1.923	1.864	2.768	1.857	2.155
24	3.005	1.915	2.105	3.285	1.903	1.817	1.889	2.499	1.847	2.02	2.228
28	3.34	1.964	2.417	2.792	1.975	1.853	2.886	1.854	1.93	2.795	2.381

Table 4.50 – Grid Search PCA Execution Time (ms) on Testing Data Set for Optimal PC for SVM (kernel = rbf, gamma = auto)

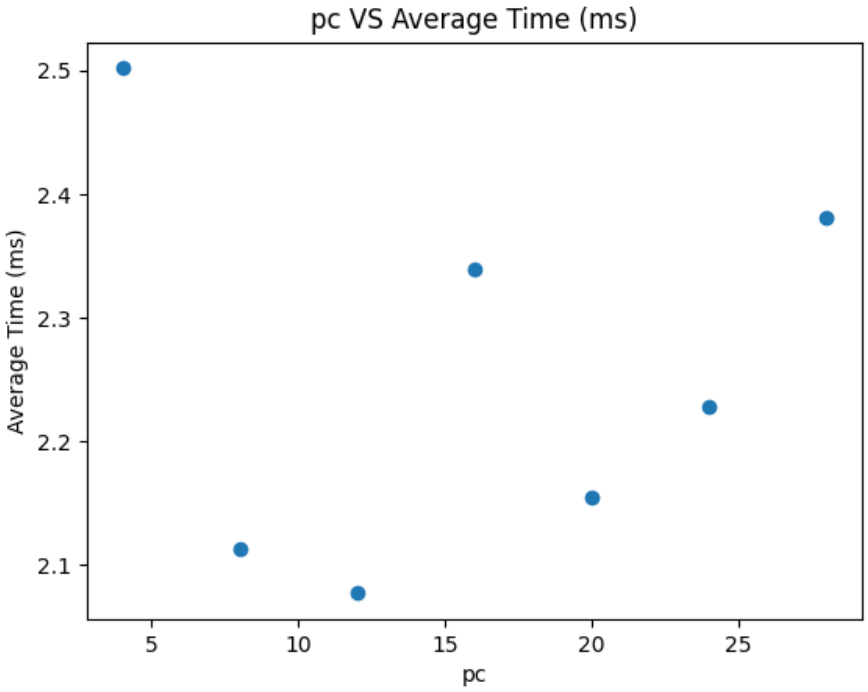

Fig. 4.29 – PC VS Average Time (ms) for PCA on Testing Data

4.2.2 Kernel Grid Search

For the kernel grid search, we look at the performance of the following kernels: linear, polynomial, rbf and sigmoid. According to Table 4.57, the best performing kernel was the polynomial kernel, followed by the rbf kernel. The polynomial kernel had the faster execution time when compared to the rbf kernel (Table 4.53 and 4.54). Furthermore, when examining the confusion matrix for the polynomial kernel (Table 4.52), there is less bias compared to the confusion matrix for the rbf kernel (Table 4.45). Thus, we chose to use the polynomial kernel as our optimal kernel.

Kernel	Accuracy
linear	0.670
polynomial	0.846
rbf	0.807
sigmoid	0.423

Table 4.51 – Grid Search for Optimal Kernel

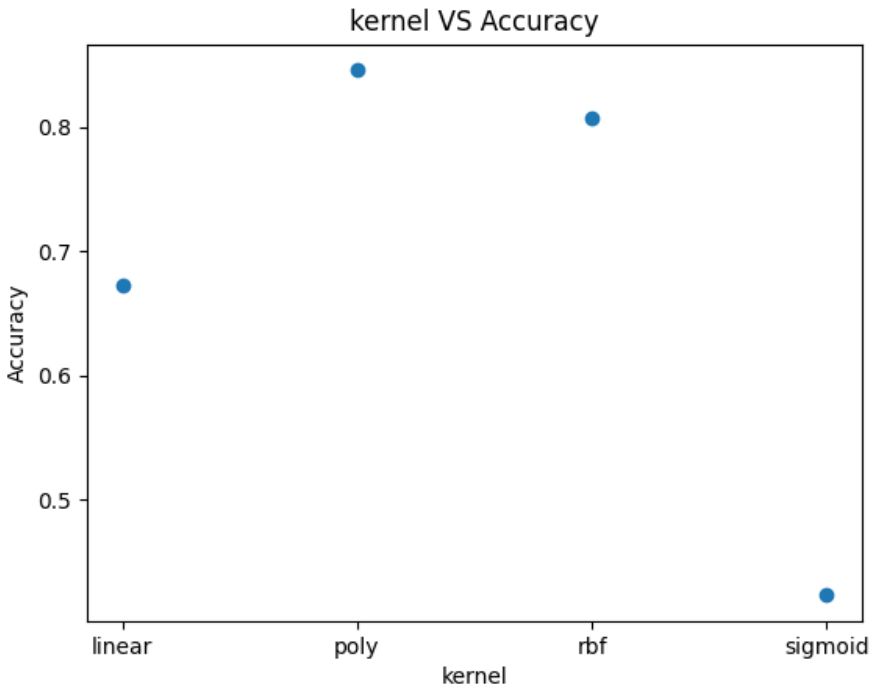

Fig. 4.30 – Kernel VS. Accuracy for SVM

Class	TPR	FPR	TNR	FNR
AS	0.783	0.103	0.896	0.217
ST	0.897	0.217	0.783	0.103

Table 4.52 – Confusion Matrix for SVM after PCA (Accuracy = 0.846, PC = 20, kernel = poly, gamma = auto, degree = 3)

kernel	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
linear	1179.95	1204.19	1833.11	1198.13	1490.50	1181.80	1181.84	1250.11	1170.25	1823.33	1351.32
poly	560.36	544.10	558.73	555.55	840.60	543.85	549.92	540.80	566.54	860.48	612.1
rbf	555.16	530.43	537.30	574.7	834.53	531.09	775.54	538.06	535.87	546.92	596.0
sigmoid	970.88	1387.82	975.52	956.86	975.72	966.16	1398.38	957.68	954.82	973.70	1051.76

Table 4.53 – Grid Search Model Training Execution Time (ms) for Optimal Kernel for SVM (PC = 20, gamma = auto)

kernel	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
linear	4.704	4.829	9.305	5.214	9.712	5.676	4.781	4.816	4.882	10.422	6.434
poly	3.96	3.914	3.891	3.905	7.23	3.956	4.071	3.992	4.147	6.683	4.575
rbf	7.855	6.944	7.097	6.937	14.117	6.956	13.679	7.114	7.005	7.281	8.498
sigmoid	8.545	13.978	8.376	8.853	8.327	8.808	12.644	8.291	8.366	8.412	9.46

Table 4.54 – Grid Search Model Execution Time (ms) on Testing Data Set for Optimal Kernel for SVM (PC = 20, gamma = auto)

kernel	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
linear	21.009	19.651	45.825	20.36	20.75	26.752	20.478	42.907	22.204	39.258	27.919
poly	20.473	21.252	20.712	20.468	40.328	21.838	19.601	44.985	20.985	33.482	26.412
rbf	19.663	27.112	23.12	20.211	53.327	20.528	20.304	19.725	19.971	22.746	24.671
sigmoid	22.178	20.593	26.112	33.39	20.581	19.661	51.026	19.848	21.43	22.028	25.685

Table 4.55 – Grid Search PCA Training Execution Time (ms) for Optimal Kernel for SVM (PC = 20, gamma = auto)

kernel	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
linear	1.933	1.92	2.856	2.654	2.848	1.864	1.889	1.895	1.863	3.234	2.296
poly	1.867	1.863	1.866	1.866	2.762	1.927	1.884	1.949	1.88	3.141	2.101
rbf	1.886	1.842	1.889	1.868	2.816	1.887	2.839	2.269	2.114	1.899	2.131
sigmoid	1.946	2.83	1.906	1.855	1.982	1.974	2.823	1.865	1.884	1.858	2.092

Table 4.56 – Grid Search PCA Execution Time (ms) on Testing Data Set for Optimal Kernel for SVM (PC = 20, gamma = auto)

4.2.3 Optimal Degree for Polynomial Kernel

Since we chose the polynomial kernel for the SVM algorithm, there is an additional parameter that we need to optimize, the degree of the polynomial. The sklearn library has this parameter default value set to 3 [22], but we perform another grid search to see if changing the degree would improve the performance and execution times. We test degrees between 2 and 10. As shown in Fig. 4.31, the highest accuracy produced was from a degree of 3. The other degrees did not produce an accuracy over 0.8, so we choose a degree of 3 to continue with the grid search without looking at the execution times.

Degree	Accuracy
2	0.750
3	0.846
4	0.769
5	0.769
6	0.711
7	0.692
8	0.654
9	0.654
10	0.654

Table 4.57 – Grid Search for Optimal Degree for Polynomial Kernel (PC = 20)

4.2.4 Kernel Coefficient

The SVM function in the sklearn library has one more parameter, *gamma*, that we can try to alter to improve the performance of the optimal parameter set we found. The *gamma* parameter has 2 inputs: "auto" and "scale". The default value for this parameter was set to "auto" which sets the the kernel's coefficient to Eq. 4.6. We test to see if "scale," which sets the coefficient to Eq. 4.7, can improve the confusion matrix of the parameter set we chose.

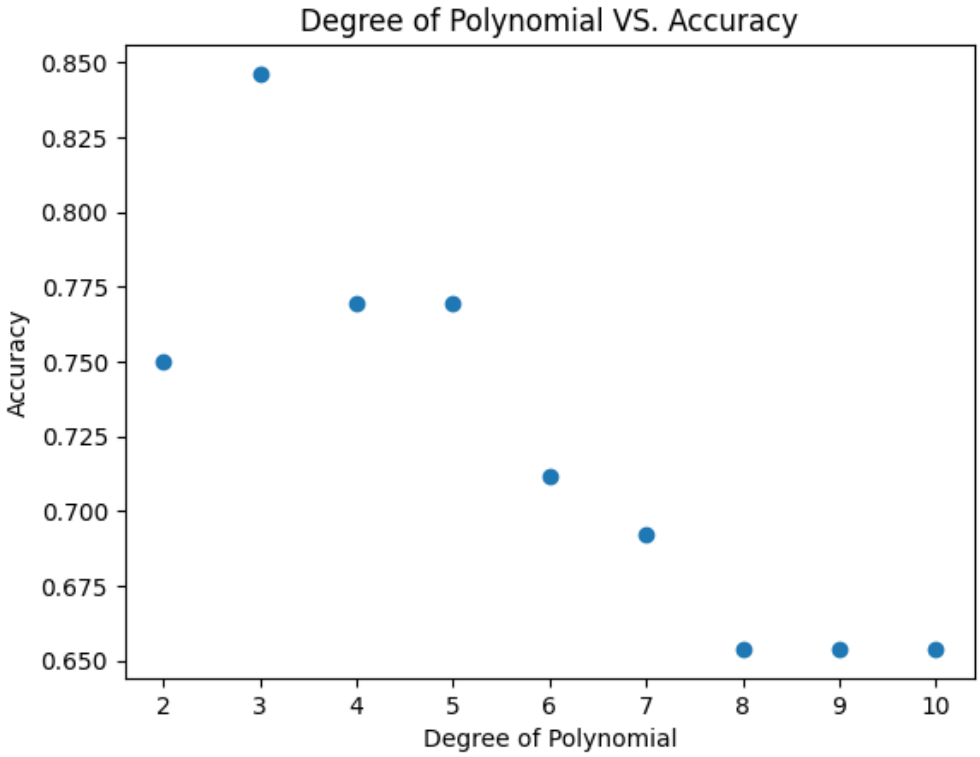

Fig. 4.31 – Degree of Polynomial VS. Accuracy

As seen in Table 4.58, the accuracy did not change between "auto" and "scale." Furthermore, the confusion matrix for "auto" (Table 4.45) and "scale" (Table 4.59 are the identical. Thereby, we conclude that the coefficient does not really impact the performance of our model.

$$\gamma = \frac{1}{n_{features}} \tag{4.6}$$

$$\gamma = \frac{1}{n_{features} * X.var()} \tag{4.7}$$

Gamma	Accuracy	Execution Time (s)
auto	0.846	0.898
scale	0.846	0.845

Table 4.58 – Grid Search for Optimal Gamma Value for Polynomial Kernel (PC = 20, kernel = poly, degree = 3)

Class	TPR	FPR	TNR	FNR
AS	0.783	0.103	0.896	0.217
ST	0.897	0.217	0.783	0.103

Table 4.59 – Confusion Matrix for SVM after PCA (Accuracy = 0.846, PC = 20, kernel = poly, degree = 3, gamma = scale)

gamma	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
auto	688.946	852.508	554.391	587.695	543.445	539.245	546.091	661.223	860.746	552.923	638.721
scale	846.344	832.143	554.013	546.912	562.571	558.343	543.365	817.445	834.553	544.719	664.041

Table 4.60 – Grid Search Model Training Execution Time (ms) for Optimal Kernel Coefficient (gamma) for SVM (PC = 20, kernel = polynomial, degree = 3)

gamma	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
auto	7.205	7.322	3.969	4.15	3.865	4.025	3.923	7.092	7.455	3.928	5.293
scale	7.231	4.494	3.919	3.883	3.965	4.112	3.951	7.207	3.881	3.852	4.65

Table 4.61 – Grid Search Model Execution Time (ms) on Testing Data Set for Optimal Kernel Coefficient (gamma) for SVM (PC = 20, kernel = polynomial, degree = 3)

gamma	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
auto	27.466	34.454	20.926	20.991	19.559	21.863	19.92	22.294	35.108	20.087	24.267
scale	35.521	35.15	22.508	19.956	21.245	21.116	20.139	52.406	35.896	29.88	29.382

Table 4.62 – Grid Search PCA Training Execution Time (ms) for Optimal Kernel Coefficient (gamma) for SVM (PC = 20, kernel = polynomial, degree = 3)

gamma	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
auto	3.091	2.868	1.864	1.894	1.85	1.921	1.821	2.824	2.959	1.912	2.3
scale	2.824	2.031	1.869	1.913	1.891	2.061	1.904	2.769	1.845	1.834	2.094

Table 4.63 – Grid Search PCA Execution Time (ms) on Testing Data Set for Optimal Kernel Coefficient (gamma) for SVM (PC = 20, kernel = polynomial, degree = 3)

After conducting the grid search we decided that the optimal parameters are: 20 PC, polynomial kernel of degree 3 and a gamma set to auto. The SVM can be further improved since the optimal confusion matrix we determined still showed some bias towards the ST class. Improvements can be done by increasing the data set or creating a kernel that was not built into sklearn.

4.3 Comparison

Both the random forest classifier and SVM algorithms each have their own strengths and weaknesses, especially when it comes to our purposes. As seen in Table 4.64, the random forest classifier algorithm was able to produce a better accuracy then SVM, but SVM has faster model execution time. The PCA times for SVM were longer than the random forest ones because there were more PCs calculated (SVM used 20 PCs and the Random Forest model used 4 PCs). However, the random forest model times were higher than the SVM model times. This could be because the random forest algorithm requires to execute multiple decision trees, whereas SVM finds one singular hyperplane. Furthermore, the SVM optimal parameter set (Table 4.52) produced a more biased confusion matrix while the optimized random forest model(Table 4.33) shows a more balanced confusion matrix. In general, while both models offer some benefits, they can be improved with further work such as examining their errors to see what affected the models ability to classify images.

Evaluation Parameters	Random Forest	SVM
Accuracy	0.865	0.846
Avg. Model Test Time (ms)	12.468	5.293
Avg. Model Train Time (ms)	832.77	638.721
Avg. PCA Test Time (ms)	0.235	2.300
Avg. PCA Train Time (ms)	2.549	24.267

Table 4.64 – Random Forest and SVM Evaluation Parameters

Hyperparameter	Optimal Value
PC	4
Number of Trees	100
Min. Sample Leaf	4
Max Depth	20

Table 4.65 – Optimal Random Forest Hyperparameters

Hyperparameter	Optimal Value
PC	20
Kernel	poly
Optimal Degree for Polynomial Kernel	3
Kernel Coefficient	auto

Table 4.66 – Optimal SVM Hyperparameters

4.4 Classification Errors

We can get more insights on why the models failed to correctly classify some of the videos by looking further into those specific cases. For each model, we look closely to at an example where each class was incorrectly classified.

For the random forest classifier, the model incorrectly classified Test Images ID 1 and ID 12. Test Image ID 1 (Fig. 4.32) is classified as AS but the model classified it as ST. Test Image ID 12 (Fig.4.33) on the other hand was incorrectly classified as AS by the random forest classifier. A possible reason these 2 models were incorrectly classified could be because of the signer's eyebrow placement. In these 2 images, the eyebrow placement of the signers look very similar, whereas in typical ASL AS and ST sentences, the eyebrow placement is slightly different. AS tend to have a more furrowed eyebrow placement while ST has a more neutral looking eyebrow. In these 2 example cases, the eyebrows appear raised which could have confused the model.

For SVM, the model incorrectly classified Test Images ID 25 and 44. Test Image ID 25 (Fig. 4.34) is classified as AS, but was incorrectly classified as ST. Test Image ID 44 (Fig. 4.35) was incorrectly classified as AS when it was actually ST. Similar to the random forest example cases, the eyebrows in these 2 images could have interfered with the model's ability to correctly classify it. In this case, the 2 example cases for SVM show a more neutral but slightly raised eye brow placement, which is different from the typical eyebrow placement of AS and ST sentences. This placement again could explain why SVM could not classify these cases correctly.

Interestingly, the 2 example error cases from each model was correctly classified by the other model. This could indicate that some models are better at classifying specific types of sentences, but further research is required to know for sure. However, there was 1 test case that both models incorrectly classified. Test Image ID 33 (Fig. 4.36) was classified as AS, but was actually ST. This could be due to the misalignment in the coordinates derived from Dlib. As seen in Fig. 4.36, the signer had his eyes closed and his head tilted back, causing a slight error in the coordinates derived from the Dlib model. This could have caused both models to incorrectly classify this image.

Fig. 4.32 – Test Image ID 1

Fig. 4.33 – Test Image ID 12

Fig. 4.34 – Test Image ID 25

Fig. 4.35 – Test Image ID 44

Fig. 4.36 – Test Image ID 33

CHAPTER 5

Conclusions and Future Directions

Real time language interpreting models have been proven very useful in today's society, as it works to reduce the language barrier among people and is more accessible due to the advancement of mobile technology. However, current real time ASL interpreting models are limited due to them being based solely on hand gestures and not taking into account facial expressions, which play a fundamental role in ASL syntax. Furthermore, other attempts at integrating facial expressions into real time ASL interpreting models such as generating a facial mesh [33] or creating normalized facial distances between key points [19] are quite computational expensive and would not work very well on mobile and embedded devices. In our project, we have proposed creating a computational efficient sentence classification model that will supplement these current hand gesture based ASL interpreting models.

For our project, we propose a new, computational efficient way to quantify facial expressions and we use this new way to train 2 different classification models to see how well each one would perform. Initially, the models are trained to classify 2 types of sentences: assertions and statements. We quantify facial expressions by calculating the angles between key facial points and reduce dimensionality with PCA. From there, we implement 2 types of classification models (random forest classifier and SVM) to see how well each one would perform.

Further directions for this project include expanding the data set and its capabilities to classify all 8 sentences present in ASL syntax. This project is currently able to classify 2 sentences types and to achieve real time ASL interpreting, all 8 sentences has to be classified. To do so, we need to expand and diversify the data set. The dataset needs to have enough data so that all 8 sentence types are represented and be diverse in the type of signers (i.e. native, beginner, left handed, different ethnicities) so that the model trained will be able to handle a more diverse user set.

Bibliography

[1] Muzammil Abdulrahman and Alaa Eleyan. Facial expression recognition using Support Vector Machines. In *2015 23nd Signal Processing and Communications Applications Conference (SIU)*, pages 276–279, Malatya, Turkey, May 2015. IEEE.

[2] Muhammad Achirul Nanda, Kudang Boro Seminar, Dodi Nandika, and Akhiruddin Maddu. A Comparison Study of Kernel Functions in the Support Vector Machine and Its Application for Termite Detection. *Information*, 9(1):5, Jan. 2018.

[3] Charlotte Baker-Shenk and Dennis Cokely. *American Sign Language - A Teacher's Resource Text on Grammar and Culture*. Cleric Books, Gallaudet University Press, Washington, DC, USA, 1980.

[4] Gérard Biau and Erwan Scornet. A random forest guided tour. *TEST*, 25(2):197–227, June 2016.

[5] Hajar Bouirouga, Sanaa El Fkihi, Abdeilah Jilbab, and Driss Aboutajdine. Comparison of Performance between Different SVM Kernels for the Identification of Adult Video. 5(5), 2011.

[6] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, Sept. 2020.

[7] Amanda Duarte, Shruti Palaskar, Lucas Ventura, Deepti Ghadiyaram, Kenneth DeHaan, Florian Metze, Jordi Torres, and Xavier Giro-i Nieto. How2Sign: A Large-scale Multimodal Dataset for Continuous American Sign Language. In *2021 IEEE/CVF*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 2734–2743, Nashville, TN, USA, June 2021. IEEE.

[8] Kai-Yin Fok, Nuwan Ganganath, Chi-Tsun Cheng, and Chi K. Tse. A Real-Time ASL Recognition System Using Leap Motion Sensors. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 411–414, Xi'an, China, Sept. 2015. IEEE.

[9] Sourish Ghosh, Anasuya Dasgupta, and Aleena Swetapadma. A Study on Support Vector Machine based Linear and Non-Linear Pattern Classification. In *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, pages 24–28, Palladam, Tamilnadu, India, Feb. 2019. IEEE.

[10] Aditi Goel and Saurabh Kr. Srivastava. Role of Kernel Parameters in Performance Evaluation of SVM. In *2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)*, pages 166–169, Ghaziabad, India, Feb. 2016. IEEE.

[11] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification.

[12] Tyler G. James, Kyle A. Coady, Jeanne-Marie R. Stacciarini, Michael M. McKee, David G. Phillips, David Maruca, and JeeWon Cheong. "They're Not Willing To Accommodate Deaf patients": Communication Experiences of Deaf American Sign Language Users in the Emergency Department. *Qualitative Health Research*, 32(1):48–63, Jan. 2022.

[13] Omer Karal. Performance comparison of different kernel functions in SVM for different k value in k-fold cross-validation. In *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pages 1–5, Istanbul, Turkey, Oct. 2020. IEEE.

[14] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. In *2014 IEEE Conference on Computer Vision and Pattern*

Recognition, pages 1867–1874, Columbus, OH, June 2014. IEEE.

[15] Davis E. King. Dlib-ml: A Machine Learning Toolkit. 10:1755–1758, 2009.

[16] Yuan Luo, Cai-ming Wu, and Yi Zhang. Facial expression recognition based on fusion feature of PCA and LBP with SVM. *Optik - International Journal for Light and Electron Optics*, 124(17):2767–2770, Sept. 2013.

[17] M. I. N. P. Munasinghe. Facial Expression Recognition Using Facial Landmarks and Random Forest Classifier. In *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, pages 423–427, Singapore, June 2018. IEEE.

[18] Abolfazl Nadi and Hadi Moradi. Increasing the views and reducing the depth in random forest. *Expert Systems with Applications*, 138:112801, Dec. 2019.

[19] Tan Dat Nguyen and Surendra Ranganath. Facial expressions in American sign language: Tracking and recognition. *Pattern Recognition*, 45(5):1877–1891, May 2012.

[20] E. Osuna, R. Freund, and F. Girosit. Training support vector machines: an application to face detection. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 130–136, San Juan, Puerto Rico, 1997. IEEE Comput. Soc.

[21] A. Patle and D. S. Chouhan. SVM kernel functions for classification. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–9, Mumbai, Jan. 2013. IEEE.

[22] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, and David Cournapeau. Scikit-learn: Machine Learning in Python. *MACHINE LEARNING IN PYTHON*.

[23] Derek A. Pisner and David M. Schnyer. Support vector machine. In *Machine Learning*, pages 101–121. Elsevier, 2020.

[24] V. Rodriguez-Galiano, M. Sanchez-Castillo, M. Chica-Olmo, and M. Chica-Rivas. Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geology Reviews*, 71:804–818, Dec. 2015.

[25] Lior Rokach. Decision forest: Twenty years of research. *Information Fusion*, 27:111–125, Jan. 2016.

[26] Christos Sagonas, Epameinondas Antonakos, Georgios Tzimiropoulos, Stefanos Zafeiriou, and Maja Pantic. 300 Faces In-The-Wild Challenge: database and results. *Image and Vision Computing*, 47:3–18, Mar. 2016.

[27] Shikhar Sharma and Krishan Kumar. ASL-3DCNN: American sign language recognition technique using 3-D convolutional neural networks. *Multimedia Tools and Applications*, 80(17):26319–26331, July 2021.

[28] Jungpil Shin, Akitaka Matsuoka, Md. Al Mehedi Hasan, and Azmain Yakin Srizon. American Sign Language Alphabet Recognition by Extracting Feature from Hand Pose Estimation. *Sensors*, 21(17):5856, Aug. 2021.

[29] Cheng Tang and Damien Garreau. When do random forests fail?

[30] Murat Taskiran, Mehmet Killioglu, and Nihan Kahraman. A Real-Time System for Recognition of American Sign Language by using Deep Learning. In *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, pages 1–5, Athens, July 2018. IEEE.

[31] Michael E Tipping and Christopher M Bishop. Mixtures of Probabilistic Principal Component Analysers. page 30.

[32] Omkar Vedak, Prasad Zavre, Abhijeet Todkar, and Manoj Patil. Sign Language Interpreter using Image Processing and Machine Learning. 06(04):4, 2019.

[33] Christian Vogler and Siome Goldenstein. Facial movement analysis in ASL. *Universal Access in the Information Society*, 6(4):363–374, Feb. 2008.

[34] C. Vogler and D. Metaxas. ASL recognition based on a coupling between HMMs and 3D motion analysis. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 363–369, Bombay, India, 1998. Narosa Publishing House.

[35] Yingying Wang, Yibin Li, Yong Song, and Xuewen Rong. Facial Expression Recognition Based on Random Forest and Convolutional Neural Network. *Information*, 10(12):375, Nov. 2019.

[36] Michael Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, Apr. 2021.

[37] Dujuan Zhang, Jie Li, and Zhenfang Shan. Implementation of Dlib Deep Learning Face Recognition Technology. In *2020 International Conference on Robots & Intelligent System (ICRIS)*, pages 88–91, Sanya, China, Nov. 2020. IEEE.