*APPENDIX*

## IMPORT LIBRARY

```
import csv
import random
import math
import operator
import time
import matplotlib.pyplot as plt
```

## RUN TIME PROCESS

```
start_time=time.time()
print("--- %s seconds ---" % (time.time() - start_time))
```

## LOAD DATA USING KKN ALGORITHM

```
def load_csv(filename):
        dataset = list()
        with open(filename, 'r') as file:
                csv_reader = reader(file)
                for row in csv_reader:
                        if not row:
                                continue
                        dataset.append(row)
        return dataset
```

## SPLIT DATA USING KKN ALGORITHM

```
def loaddatasets(datasets, split, trainingSet=[] , testSet=[]):
        for a in range(len(datasets)):
           for b in range(8):
              datasets[a][b] = float(datasets[a][b])
           if random.random() < split:
              trainingSet.append(datasets[a])
           else:
```

```
            testSet.append(datasets[a])
```

## EUCLIDEAN DISTANCE

```
def euclideanDistance(instance1, instance2, length):
        distancing = 0
        for a in range(length):
                distancing += pow((instance1[a] - instance2[a]), 2)
        return math.sqrt(distancing)
```

## SELECT SUBSETWITH  THE SMALLEST DISTANCE

```
def getNeighbors(trainingSet, testInstance, k):
        distances = []
        length = len(testInstance)-1
        for a in range(len(trainingSet)):
                dist = euclideanDistance(testInstance, trainingSet[a], length)
                distances.append((trainingSet[a], dist))
        distances.sort(key=operator.itemgetter(1))
```

## SELECT SUBSETWITH  THE SMALLEST DISTANCE WITH K

```
        neighbors = []
        for a in range(k):
                neighbors.append(distances[a][0])
                # print ("neighbors: ", neighbors)
        return neighbors
```

## ASSEMBLY ACCORDING TO CLASS

```
def VoteSort(neighbors):
        classVotes = {}
        for a in range(len(neighbors)):
                response = neighbors[a][-1]


                if response in classVotes:
                        classVotes[response] += 1
                else:
                        classVotes[response] = 1
```

```
        sortedVotes=sorted(classVotes.items(),key=operator.itemgetter(1), reverse=True)
        return sortedVotes[0][0]
```

## MAKE CONFUSION MATRIX

```
def compute_tp_tn_fn_fp(testSet, predictions):
        '''
        True positive - actual = 1, predicted = 1
        True negative - actual = 0, predicted = 0
        False positive - actual = 0, predicted = 1
        False negative - actual = 1, predicted = 0
        '''
        tp=0
        tn=0
        fn=0
        fp=0

        for a in range(len(testSet)):
                if(testSet[a][-1]=='TEPAT' and predictions[a]=='TEPAT'):
                        tp += 1
        for a in range(len(testSet)):
                if(testSet[a][-1]=='TERLAMBAT'                                  and
predictions[a]=='TERLAMBAT'):
                        tn += 1
        for a in range(len(testSet)):
                if(testSet[a][-1]=='TERLAMBAT' and predictions[a]=='TEPAT'):
                        fp += 1
        for a in range(len(testSet)):
```

```
            if(testSet[a][-1]=='TEPAT' and predictions[a]=='TERLAMBAT'):
                fn += 1
    return tp, tn, fp, fn
```

## COUNT ACCURACY

```
def compute_accuracy(tp, tn, fn, fp):
    '''Accuracy = TP + TN / FP + FN + TP + TN'''
    return ((tp + tn) * 100)/ float( tp + tn + fn + fp)
    print('Accuracy:', compute_accuracy(tp, tn, fn, fp))

    return (correct/float(len(testSet))*100)
```

## COUNT PRECISION

```
def compute_precision(tp, fp):
    '''Precision = TP  / FP + TP '''
    return (tp  * 100)/ float( tp + fp)

    print('Precision:', compute_precision(tp, fp))
```

## COUNT RECALL

```
def compute_recall(tp, fn):
    '''Recall = TP /FN + TP '''
    return (tp  * 100)/ float( tp + fn)
    print('Recall:', compute_recall(tp, fn))
```

## COUNT F1 SCORE

```
def compute_f1_score(testSet, predictions):
  # calculates the F1 score
  tp, tn, fp, fn = compute_tp_tn_fn_fp(testSet, predictions)
  precision = compute_precision(tp, fp)/100
  recall = compute_recall(tp, fn)/100
  f1_score = (2*precision*recall)/ (precision + recall)
  return f1_score* 100
```

## PREDICTIONS AND MAIN PROGRAM

```
def main():
	filename='data_mahasiswa4.csv'
	dataset = load_csv(filename)
	print(len(dataset))
	while input("new dataset??") != 'n':
		print("input file name")
		k = input()
		dataset2 = load_csv(k)
		dataset.extend(dataset2)
	print(len(dataset))
	# prepare data
	random.seed(1)
	trainingSet=[]
	testSet=[]
	split = 0.5
	loaddatasets(dataset, split, trainingSet, testSet)
	print ('Train set: ' + repr(len(trainingSet)))
	print ('Test set: ' + repr(len(testSet)))
	# generate predictions
	predictions=[]
	accuracy=[]
	precision2=[]
	recall2=[]
	f1_score2=[]
	k=list(range(1, 100))
	for i in range(1, len(k)+1):
		predictions.clear()
		for a in range(len(testSet)):
			neighbors = getNeighbors(trainingSet, testSet[a], i)
			result = VoteSort(neighbors)
			predictions.append(result)
			#print('> predicted=' + repr(result) + ', actual=' +
repr(testSet[a]))
		tp, tn, fp, fn =compute_tp_tn_fn_fp(testSet, predictions)
```

```
                accuracy.append(compute_accuracy(tp, tn, fn, fp))
                precision2.append(compute_precision(tp, fp))
                recall2.append(compute_recall(tp, fn))
                f1_score2.append(compute_f1_score(testSet, predictions))

                print('========================')
                print('k= ', i)
                print('========================')
                print('tp= ', tp)
                print('tn= ', tn)
                print('fp= ', fp)
                print('fn= ', fn)
                print('Accuracy :', accuracy[i-1])
                print('Precision :', compute_precision(tp, fp))
                print('recall :', compute_recall(tp, fn))
                print('f1 Score :', compute_f1_score(testSet, predictions))
        print("Maximum Accuracy = ",max(accuracy))
        print("Maximum Precision = ",max(precision2))
        print("Maximum Recall = ",max(recall2))
        print("Maximum f1_score = ",max(f1_score2))

        plt.plot(k,accuracy, color='green', linewidth = 1, marker='o',
markerfacecolor='blue', markersize=5)
        plt.ylabel('Accuracy')
        plt.xlabel('K-Value')
        plt.xlim(1,100)
        plt.title('KNN')
        plt.show()
main()
print("--- %s seconds ---" % (time.time() - start_time))
```

## IMPORT LIBRARY

```
from random import seed
from random import randrange
```

```
from csv import reader

from math import sqrt

import time
```

## RUN TIME PROCESS

```
start_time=time.time()
print("--- %s seconds ---" % (time.time() - start_time))
```

## LOAD DATASET

```
filename = 'data_mahasiswa4.csv'

dataset = load_csv(filename)

def load_csv(filename):
        dataset = list()
        with open(filename, 'r') as file:
                csv_reader = reader(file)
                for row in csv_reader:
                        if not row:
                                continue
                        dataset.append(row)
        return dataset
```

## K FOLD CROSS VALIDATION SPLIT

```
def cross_validation_split(dataset, n_folds):
        dataset_split = list()
        dataset_copy = list(dataset)
        fold_size = int(len(dataset) / n_folds)

        for i in range(n_folds):
```

```
            fold = list()
            while len(fold) < fold_size:
                index = randrange(len(dataset_copy))
                fold.append(dataset_copy.pop(index))
            dataset_split.append(fold)
    return dataset_split
```

## SPLIT DATA BASED ON ATTRIBUTE AND VALUE

```
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right
```

## GINI INDEX

```
def gini_index(groups, classes):
    # count all samples at split point
    n_point = float(sum([len(group) for group in groups]))
    # n_point2 = ([len(group) for group in groups])

    # sum weighted Gini index for each group
    gini = 0.0
    for group in groups:
        size = float(len(group))
```

```
            if size == 0:
                continue
            score = 0.0
            # score the group based on the score for each class
            for class_val in classes:
                p = [row[-1] for row in group].count(class_val) / size
                score += p * p

            # weight the group score by its relative size
            gini += (1.0 - score) * (size / n_point)
```

## GET BEST SPLIT

```
def get_best_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None

    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
```

```
                b_index, b_value, b_score, b_groups = index,
row[index], gini, groups
        return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

## CREATE TERMINAL NODE

```
def terminal(group):
        outcomes = [row[-1] for row in group]
        # print('outcomes:',outcomes)
        return max(set(outcomes), key=outcomes.count)
```

## CREATE CHILD SPLIT

```
def split(node, max_depth, min_size, n_features, depth):
        left, right = node['groups']
        #Proses pemangkasan dengan cara menghapus dari node.
        del(node['groups'])
        # check for a no split
        if not left or not right:
                node['left'] = node['right'] = terminal(left + right)
                return
        # check for max depth
        if depth >= max_depth:
                node['left'], node['right'] = terminal(left), terminal(right)
                return
        # process left child
        if len(left) <= min_size:
                node['left'] = terminal(left)
        else:
                node['left'] = get_best_split(left, n_features)
                split(node['left'], max_depth, min_size, n_features, depth+1)
        # process right child
        if len(right) <= min_size:
                node['right'] = terminal(right)
        else:
                node['right'] = get_best_split(right, n_features)
                split(node['right'], max_depth, min_size, n_features, depth+1)
```

## CREATE DECISION TREE

```
def create_tree(train, max_depth, min_size, n_features):
        root = get_best_split(train, n_features)
        split(root, max_depth, min_size, n_features, 1)
        return root
```

## MAKE PREDICTION WITH A DECISION TREE

```
def predict(node, row):
        if row[node['index']] < node['value']:
                if isinstance(node['left'], dict):
                        return predict(node['left'], row)
                else:
                        return node['left']
        else:
                if isinstance(node['right'], dict):
                        return predict(node['right'], row)
                else:
                        return node['right']
```

## CREATE SUBSAMPLE RANDOM

```
def subsample(dataset, ratio):
        sample = list()
        n_sample = round(len(dataset) * ratio)
        while len(sample) < n_sample:
                index = randrange(len(dataset))
                sample.append(dataset[index])

        return sample
```

## MAKE PREDICTIONS WITH A LIST OF BAGGED TREES

```
def bagging_predict(trees, row):
        predictions = [predict(tree, row) for tree in trees]
        return max(set(predictions), key=predictions.count)
```

## RANDOM FOREST ALGORITHM

```
seed(3)
def random_forest(train, test, max_depth, min_size, sample_size, n_trees, n_features):
```

```
    trees = list()
    for i in range(n_trees):
        sample = subsample(train, sample_size)
        tree = create_tree(sample, max_depth, min_size, n_features)
        trees.append(tree)
    predictions = [bagging_predict(trees, row) for row in test]
    # print('predictions:',predictions)
    return(predictions)
```

## CALCULATE ACCURACY PERCENTAGE WITHOUT CONFUSION MATRIX

```
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    # print('correct:',correct)
    return correct / float(len(actual)) * 100.0
```

## MAKE CONFUSION MATRIX

```
def compute_tp_tn_fn_fp(testSet, predictions):
    '''
    True positive - actual = 1, predicted = 1
    True negative - actual = 0, predicted = 0
    False positive - actual = 0, predicted = 1
    False negative - actual = 1, predicted = 0
    '''
    tp=0
    tn=0
    fn=0
    fp=0

    for a in range(len(testSet)):
        if(testSet[a][-1]=='TEPAT' and predictions[a]=='TEPAT'):
            tp += 1
```

```
        for a in range(len(testSet)):
                if(testSet[a][-1]=='TERLAMBAT' and predictions[a]=='TERLAMBAT'):
                        tn += 1
        for a in range(len(testSet)):
                if(testSet[a][-1]=='TERLAMBAT' and predictions[a]=='TEPAT'):
                        fp += 1
        for a in range(len(testSet)):
                if(testSet[a][-1]=='TEPAT' and predictions[a]=='TERLAMBAT'):
                        fn += 1
        return tp, tn, fp, fn
```

## COUNT ACCURACY

```
def compute_accuracy(tp, tn, fn, fp):
        '''Accuracy = TP + TN / FP + FN + TP + TN'''
        return ((tp + tn) * 100)/ float( tp + tn + fn + fp)
        print('Accuracy:', compute_accuracy(tp, tn, fn, fp))

        return (correct/float(len(testSet))*100)
```

## COUNT PRECISION

```
def compute_precision(tp, fp):
        '''Precision = TP  / FP + TP '''
        return (tp  * 100)/ float( tp + fp)

        print('Precision:', compute_precision(tp, fp))
```

## COUNT RECALL

```
def compute_recall(tp, fn):
        '''Recall = TP /FN + TP '''
        return (tp  * 100)/ float( tp + fn)
        print('Recall:', compute_recall(tp, fn))
```

## COUNT F1 SCORE

```
def compute_f1_score(testSet, predictions):

   # calculates the F1 score
```

```
    tp, tn, fp, fn = compute_tp_tn_fn_fp(testSet, predictions)
    precision = compute_precision(tp, fp)/100
    recall = compute_recall(tp, fn)/100
    f1_score = (2*precision*recall)/ (precision + recall)
    return f1_score* 100
```

## EVALUATE AN ALGORITHM USING CROSS VALIDATION SPLIT

```
n_folds = 5
max_depth = 10
min_size = 1
sample_size = 0.5
n_features = int(sqrt(len(dataset[0])-1))
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
        folds = cross_validation_split(dataset, n_folds)
        scores = list()
        for fold in folds:
                train_set = list(folds)
                train_set.remove(fold)
                train_set = sum(train_set, [])
                test_set = list()
                for row in fold:
                        row_copy = list(row)
                        test_set.append(row_copy)
                        row_copy[-1] = None
                predicted = algorithm(train_set, test_set, *args)
                actual = [row[-1] for row in fold]
                accuracy = accuracy_metric(actual, predicted)
                scores.append(accuracy)
                print('> predicted=' + repr(predicted) + ', actual=' + repr(test_set))
        return scores
for n_trees in [1, 5, 10]:
```

```
        scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size, sample_size, n_trees, n_features)
        print('Trees: %d' % n_trees)
        print('Scores: %s' % scores)
        print('Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

        tp, tn, fp, fn =compute_tp_tn_fn_fp(test_set, predicted)

        print('tp= ', tp)
        print('tn= ', tn)
        print('fp= ', fp)
        print('fn= ', fn)
        print('Accuracy :', compute_accuracy(tp, tn, fn, fp))
        print('Precision :', compute_precision(tp, fp))
        print('recall :', compute_recall(tp, fn))
        print('f1 Score :', compute_f1_score(test_set, predicted))

        #Untuk akurasi menjumlahkan semua nilai scores ditotal lalu dibagi banyaknya scores
        print("--- %s seconds ---" % (time.time() - start_time))
```

PLAGIARISM CHECK.ORG

**2.03%** PLAGIARISM APPROXIMATELY

# Report #12340321

chapter 1 Introduction Background In classifying data, a complete and comprehensive data collection is required. Classification is the process of analyzing data and producing models that can predict the future. Clustering is a data analysis method that functions to group data with the same characteristics into the same area and data with different characteristics to other regions. Classification aims to group data and estimate the possibilities that occur based on the same class and different classes of an object. Classification is very important in predicting the possibilities that arise so that the data used can be developed and utilized in processing very large amounts of data. The large use of data can produce mixed and varied results. Classification can be based on a value, group or other things. [2] [3] Algorithms that can be used in data classification techniques are naive bayes, k-nearest neighbor (KNN), support vector machine (SVM), decision tree, random forest and so on. Each algorithm has advantages and disadvantages as well as a different level of accuracy. The problem is how to share the data we have which is the benchmark in data classification. The large amount of data used will affect the results of the data classification. Another problem is the method we use in classifying the data itself. The method used can have a positive or negative impact on the classification