Data Mining Project

Wilt Dataset Project Report

Group 11

Vamshi Pillamari-700705475
Guruvikas Reddy Busa-700703380
Medha Vijayvargia -700703734

Prof. Dr. Lianwen wang

Department of Computer science

University of Central Missouri

Individual contributions of the team:

Individual Contribution	Member Name
Selection of dataset, Decision tree holdout, Random Forest, Report	Guruvikas Reddy Busa -700703380
Boosting, Naïve Bayes Classifier, Support vector Machine (Linear and Polynomial), Report	Vamshi Pillamari - 700705475
Power Point Presentation, Report, Bagging, Support vector Machine (Radial)	Medha Vijayvargia - 700703734

Abstract:

The Objective of our project is to analyze **Wilt** Dataset to classify the data by using several classification models and then compare the misclassification rate between those models.

Classification models we are using are Decision tree using Hold-Out method, Boosting, Bagging, Random Forest and Naïve Bayes classifier, Support Vector machine (SVM). This dataset contains five deciding factors which help in predicting the class variable. The main purpose of the project is to predict the plant state (wilt and not wilt) based on the attributes in the dataset. Based on accuracies of each model we decide the best Wilt Classifying Algorithm.

Introduction:

You leave for work in the morning and your plant looks perfectly happy, but by the time you come home, they look pitiful and droopy. So why do plants wilt? Usually because they are thirsty!

Many nonwoody plants rely almost exclusively on water pressure within their cells to keep them erect. However, plants are constantly losing water through small openings in their leaves (called stomata) in a process known as transpiration. While transpiration is vital for photosynthesis and helps transport nutrients from the roots to the rest of the plant, the vast majority of the water absorbed by the roots is lost through this process. The imbalance in the water content of the plant cells results in the wilt of plant.

Based on above information a dataset is constructed with the details (color, shape) that help in predicting the state of plant.

Dataset:

This dataset contains six variables and 4839 instances. Training set contains 3387 instances (70% of data) and Testing set contains 1452 instances (30% of data). All the variables in this dataset are numerical except the class variable. There are **NO MISSING VALUES** in this dataset.

The data set consists of image segments, generated by segmenting the pansharpened image. The segments contain spectral information from the Quickbird multispectral image bands and texture information from the panchromatic (Pan) image band.

Atrribute Information:

1. class: 'w' (diseased trees), 'n' (all other land cover)
2. GLCM_Pan: GLCM mean texture (Pan band)
3. Mean_G: Mean green value
4. Mean_R: Mean red value
5. Mean_NIR: Mean NIR value
6. SD_Pan: Standard deviation (Pan band)

Decision Tree:

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is then repeated for the subtree rooted at the new node. It is a very popular machine learning algorithm. It solves the problem of machine learning by transforming the data into tree representation. Decision tree algorithm can be used to solve both regression and classification problems.

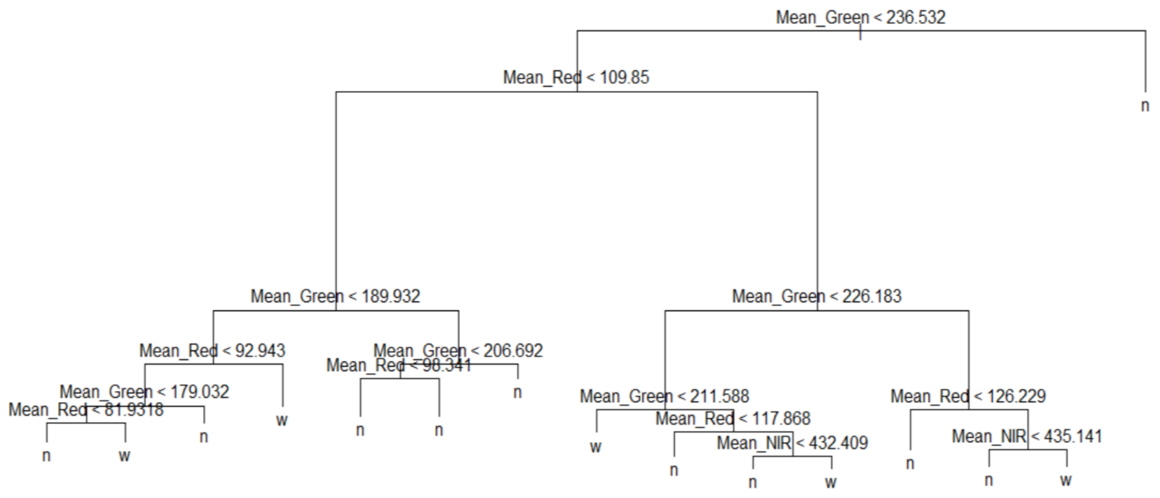
Advantages of Decision Tree:

1. Decision trees perform classification without requiring much computation.
2. Decision trees are able to handle both continuous and categorical variables.
3. A decision tree does not require normalization of data
4. Missing values in the data also does NOT affect the process of building decision tree to any considerable extent.
5. Decision trees are able to generate understandable rules.

Disadvantages of Decision Tree:

1. Decision tree often involves higher time to train the model.
2. Decision tree training is relatively expensive as complexity and time taken is more.
3. For a Decision tree sometimes, calculation can go far more complex compared to other algorithms.
4. A small change in the data can cause a large change in the structure of the decision tree causing instability.

Decision Tree of Dataset



Hold-Out Method:

Hold-out is when you split up your dataset into a ‘train’ and ‘test’ set. The training set is what the model is trained on, and the test set is used to see how well that model performs on unseen data. A common split when using the hold-out method is using 70% of data for training and the remaining 30% of the data for testing.

Holdout Method Results:

```
data=read.csv("C:/Users/vamshi/Desktop/New folder (2)/prjctwilt.csv",header =TRUE)
> fix(data)
> set.seed(123)
> RNGkind(sample.kind = "Rounding")
>  Warning message:
>  In RNGkind(sample.kind = "Rounding") : non-uniform 'Rounding' sampler used
> library(e1071)
> library(ISLR)
> library(class)
> library(plotrix)
> attach(data)
> dim(data)
>  [1] 4839    6
> wiltdata=sample(1:nrow(data),3387)
> w_trainset=data[wiltdata,]
> w_testset=data[-wiltdata,]
> library(tree)
> wilt.test=class[-wiltdata]
> wilt.train=class[wiltdata]
> tree.dt=tree(class~.,data,subset =wiltdata)
> summary(tree.dt)

>  Classification tree:
>  tree(formula = class ~ ., data = data, subset = wiltdata)
>  Variables actually used in tree construction:
>  [1] "Mean_Green" "Mean_Red"   "Mean_NIR"
>  Number of terminal nodes:  15
>  Residual mean deviance:  0.05957 = 200.9 / 3372
>  Misclassification error rate: 0.01004 = 34 / 3387
>  plot(tree.dt)
>  text(tree.dt,pretty = 0)
>  tree.pred.train=predict(tree.dt,w_trainset,type = "class")
>  table(tree.pred.train,wilt.train)
>               wilt.train
>  tree.pred.train    n    w
>               n 3204   31
>               w    3  149
>  tree.pred.train.acc=mean(tree.pred.train==wilt.train)
>  tree.pred.train.err=1-tree.pred.train.acc
>  tree.pred.train.acc*100
>  [1] 98.99616
>  tree.pred.train.err*100
>  [1] 1.003838
>  tree.pred=predict(tree.dt,w_testset,type = "class")
>  table(tree.pred,wilt.test)
>          wilt.test
>  tree.pred    n    w
>         n 1365   27
>         w    6   54
>  tree.pred.test.acc=mean(tree.pred==wilt.test)
>  tree.pred.test.err=1-tree.pred.test.acc
>  tree.pred.test.acc*100
>  [1] 97.72727
>  tree.pred.test.err*100
>  [1] 2.272727
```

Result of Hold-out Method:

Training Accuracy: 98.99616

Training Error: 1.003838

Testing Accuracy: 97.72727

Testing error: 2.272727

Bagging:

Bagging is a bootstrap ensemble method that creates individuals for its ensemble by training each classifier on a random redistribution of the training set. An ensemble method is a technique that combines the predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model. Each classifier's training set is generated by randomly drawing, with replacement. It is designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It also reduces variance and helps to avoid overfitting

Code:

```
> library(randomForest)
>  randomForest 4.6-14
>  Type rfNews() to see new features/changes/bug fixes.
>  rf_bag=randomForest(class~.,data,subset =wiltdata,ntree=500,mtry=3)
>  rf_train_pred=predict(rf_bag,w_trainset,type = "class")
>  table(rf_train_pred,wilt.train)
>              wilt.train
>  rf_train_pred    n    w
>             n 3207    0
>             w    0  180
>  rf_bag_acc=mean(rf_train_pred==wilt.train)
>  rf_bag_err=1-rf_bag_acc
>  rf_bag_acc*100
>  [1] 100
>  rf_bag_err*100
> [1] 0
> rf_test_pred=predict(rf_bag,w_testset,type = "class")
> table(rf_test_pred,wilt.test)
>            wilt.test
> rf_test_pred    n    w
>           n 1367   21
>           w    4   60
> rf_bag_test_acc=mean(rf_test_pred==wilt.test)
> rf_bag_test_err=1-rf_bag_test_acc
> rf_bag_test_acc*100
> [1] 98.27824
> rf_bag_test_err*100
> [1] 1.721763
```

Result of Bagging:

Training Accuracy: 100

Training Error: 0

Testing Accuracy: 98.27824

Testing error: 1.721763

Random Forest:

Random forest is a supervised learning algorithm. The forest it builds, is an ensemble of decision trees, usually trained with the bagging method. The general idea of the bagging method is that a combination of learning models increases the overall result. It builds multiple decision trees and merges them together to get a more accurate and stable prediction.
Random forest has nearly the same hyperparameters as a decision tree or a bagging classifier.

Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model. Therefore, in random forest, only a random subset of the features is taken into consideration by the algorithm for splitting a node

Code:

```
> library(randomForest)
> library(ISLR)
> library(tree)
```

CASE1: ntree=500 mtry=sqrt(16)

```
> tree.random_1=randomForest(class~.,w_testset, ntree=500,mtry=sqrt(16))
> tree.pred=predict(tree.random_1,w_testset,type="class")
> table(tree.pred,w_testset$class)
tree.pred    n    w
        n 1371    0
        w    0   81
> ran_test_err=mean(tree.pred!=w_testset$class)
> ran_test_acc=1-ran_test_err
> ran_test_err*100
[1] 0
> ran_test_acc*100
[1] 100
> tree.random_1t=randomForest(class~.,w_trainset, ntree=500,mtry=sqrt(16))
> tree.pred=predict(tree.random_1t,w_trainset,type="class")
> table(tree.pred,w_trainset$class)
tree.pred    n    w
        n 3207    0
        w    0  180
> ran_train_err=mean(tree.pred!=w_trainset$class)
> ran_train_acc=1-ran_train_err
> ran_train_err*100
[1] 0
> ran_train_acc*100
[1] 100
```

CASE2: ntree=100 mtry=sqrt(4)

```
>  tree.random_2=randomForest(class~.,w_testset, ntree=100,mtry=sqrt(4))
>  tree.pred=predict(tree.random_2,w_testset,type="class")
>  table(tree.pred,w_testset$class)
tree.pred    n    w
        n 1371    0
        w    0   81
>  ran_test_err_1=mean(tree.pred!=w_testset$class)
>  ran_test_acc_1=1-ran_test_err
>  ran_test_err_1*100
[1] 0
>  ran_test_acc_1*100
[1] 100
> tree.random_2t=randomForest(class~.,w_trainset, ntree=100,mtry=sqrt(4))
> tree.pred=predict(tree.random_2t,w_trainset,type="class")
> table(tree.pred,w_trainset$class)
```

```
tree.pred    n    w
        n 3207    0
        w    0  180
> ran_train_err_1=mean(tree.pred!=w_trainset$class)
> ran_train_acc_1=1-ran_train_err
> ran_train_acc_1*100
[1] 100
> ran_train_err_1*100
[1] 0
```

CASE3: ntree=1000 mtry=sqrt(9)

```
tree.random_3=randomForest(class~.,w_testset, ntree=1000,mtry=sqrt(9))
> tree.pred=predict(tree.random_3,w_testset,type="class")
> table(tree.pred,w_testset$class)
tree.pred    n    w
        n 1371    0
        w    0   81
> ran_test_err_2=mean(tree.pred!=w_testset$class)
> ran_test_acc_2=1-ran_test_err
> ran_test_acc_2*100
[1] 100
> ran_test_err_2*100
[1] 0
> tree.random_3t=randomForest(class~.,w_trainset, ntree=1000,mtry=sqrt(9))
> tree.pred=predict(tree.random_3t,w_trainset,type="class")
> table(tree.pred,w_trainset$class)
tree.pred    n    w
        n 3207    0
        w    0  180
> ran_train_err_2=mean(tree.pred!=w_trainset$class)
> ran_train_acc_2=1-ran_train_err
> ran_train_acc_2*100
[1] 100
> ran_train_err_2*100
[1] 0
```

Result of Random Forest for CASE1:

Training Accuracy: 100

Training Error: 0

Testing Accuracy: 100

Testing error: 0

Result of Random Forest for CASE2:

Training Accuracy: 100

Training Error: 0

Testing Accuracy: 100

Testing error: 0

Result of Random Forest for CASE3:

Training Accuracy: 100

Training Error: 0

Testing Accuracy: 100

Testing error: 0

Boosting:

Boosting is an ensemble learning technique that uses a set of Machine Learning algorithms to convert weak learner to strong learners in order to increase the accuracy of the model. Ensemble learning is a method that is used to enhance the performance of Machine Learning model by combining several learners. When compared to a single model, this type of learning builds models with improved efficiency and accuracy.

```
> library(gbm)
Loaded gbm 2.1.5
> data=read.csv("C:/Users/vamshi/Desktop/New folder (2)/prjctwilt.csv",header =TRUE)
> data$class=ifelse(data$class=="w",1,0)
> fix(data)
> set.seed(123)
> wiltdata=sample(1:nrow(data),3387)
> w_trainset=data[wiltdata,]
> w_testset=data[-wiltdata,]
> tree.wilt=gbm(class~., w_trainset, distribution="gaussian",n.trees=500,interaction.depth=4)
> tree.pred.prob=predict(tree.wilt,w_testset, n.trees=500, type="response")
> tree.pred=ifelse(tree.pred.prob<0.04, "Yes", "No")
> table(w_testset$class,tree.pred)
  tree.pred
    No  Yes
 0 113  1261
 1  78   0
> err_test=mean(tree.pred!=w_testset$class)
> acc_test=1-err_test
> err_test*100
```

```
[1] 100
>  acc_test*100
[1] 0
```

Result of Boosting:

Testing Accuracy: 0

Testing Error: 100

Naïve Bayes Classifier:

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.It is the most straightforward and fast classification algorithm, which is suitable for a large chunk of data.

Code:

```
> Naive_Bayes_Model=naiveBayes(class~.,w_trainset)
> NB_train= predict(Naive_Bayes_Model, w_trainset)
> table(NB_train,w_trainset$class)
NB_train    n    w
       n 2947   86
       w 260   94
> NB_error=mean(NB_train!=w_trainset$class)
> NB_acc=1-NB_error
> NB_acc*100
[1] 89.78447
> NB_error*100
[1] 10.21553
> NB_test = predict(Naive_Bayes_Model,w_testset)
> table(NB_test,w_testset$class)
NB_test    n    w
      n 1271   36
      w 100   45
> NB_testerr= mean(NB_test!=w_testset$class)
> NB_testacc=1- NB_testerr
> NB_testacc*100
[1] 90.63361
> NB_testerr*100
[1] 9.366391
```

Result of Naive Bayes:

Training Accuracy: 89.78447

Training Error: 10.21553

Testing Accuracy: 90.63361

Testing error: 9.366391

Support Vector Machine classifier (SVM):

SVM is a supervised learning algorithm that can be used for both classification and regression challenges. In SVM algorithm we plot each data item as a point in N- Dimensional space (where N is number of features you have) with the value of each feature being the value of a particular coordinate. It can solve linear and non-linear problems and work well for many practical problems. The idea of SVM is simple: The algorithm creates a line or a hyperplane which separates the data into classes.

Support vector machine using linear kernel with cost (0.01):

Code:

```
> library(e1071)
> svmfit=svm(class~., data=w_trainset, kernel="linear", cost=0.01)
> summary(svmfit)
Call:
svm(formula = class ~ ., data = w_trainset, kernel = "linear",
    cost = 0.01)
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01
Number of Support Vectors:  363
 ( 183 180 )
Number of Classes:  2
Levels:
 n w
> svm.pred = predict(svmfit,w_trainset)
> table(svm.pred,w_trainset$class)
svm.pred    n    w
       n 3207  180
       w    0    0
> svm_err=mean(svm.pred!=w_trainset$class)
> svm_acc=1-svm_err
> svm_acc*100
[1] 94.68556
> svm_err*100
[1] 5.314438
> svm.testpred = predict(svmfit,w_testset)
> table(svm.testpred ,w_testset$class)
svm.testpred    n    w
           n 1371   81
           w    0    0
> svm_testerr=mean(svm.testpred!=w_testset$class)
> svm_testacc=1-svm_testerr
> svm_testacc*100
[1] 94.42149
> svm_testerr*100
[1] 5.578512
```

Support vector machine using linear kernel with different costs (0.001,0.01,0.1,1,10,100):

```
> tune.out=tune(svm,class~., data=w_trainset,kernel="linear",
 ranges=list(cost=c(0.001,0.01,0.1,1,10,100)))
> summary(tune.out)
Parameter tuning of ‘svm’:
- sampling method: 10-fold cross validation
- best parameters:
 cost
   10
- best performance: 0.03248241
- Detailed performance results:
   cost      error   dispersion
1 1e-03 0.05314273 0.009226813
2 1e-02 0.05314273 0.009226813
3 1e-01 0.05314273 0.009226813
4 1e+00 0.04339861 0.007090489
5 1e+01 0.03248241 0.008718505
6 1e+02 0.03277827 0.008318958
> bestmod=tune.out$best.model
> summary(bestmod)
Call:
best.tune(method = svm, train.x = class ~ ., data = w_trainset,
    ranges = list (cost = c(0.001, 0.01, 0.1, 1, 10, 100)),
    kernel = "linear")
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  10
Number of Support Vectors:  290
 ( 146 144 )
Number of Classes:  2
Levels:
 n w
> train.pred = predict(bestmod,w_trainset)
> table(train.pred ,w_trainset$class)
train.pred    n    w
         n 3169   72
         w   38  108
> best_error=mean(train.pred!=w_trainset$class)
> best_acc=1-best_error
> best_acc*100
[1] 96.75229
> best_error*100
[1] 3.247712
> test.pred = predict(bestmod,w_testset)
> table(test.pred ,w_testset$class)
test.pred    n    w
        n 1357   29
        w   14   52
> best_testerr=mean(test.pred!=w_testset$class)
> best_testacc=1-best_testerr
> best_testacc*100
[1] 97.03857
> best_testerr*100
[1] 2.961433
```

Result of Support vector machine using Linear kernel for cost 0.01 :

Training Accuracy: 94.68556

Training Error: 5.314438

Testing Accuracy: 94.42149

Testing error: 5.578512

Result of Support vector machine using Linear kernel for best case:

Training Accuracy: 96.75229

Training Error: 3.247712

Testing Accuracy: 97.03857

Testing error: 2.961433

Support Vector Machine using Radial Kernel with cost =0.01

```
> svmfit_radial=svm(class~., data=w_trainset, kernel="radial",gamma=1,cost=0.01)
> summary(svmfit_radial)
Call:
svm(formula = class ~ ., data = w_trainset, kernel = "radial",
    gamma = 1, cost = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  0.01
Number of Support Vectors:  391
 ( 211 180 )
Number of Classes:  2
Levels:
 n w
> svm.pred = predict(svmfit_radial,w_trainset)
> table(svm.pred,w_trainset$class)
svm.pred    n    w
       n 3207  180
       w    0    0
> svm_err=mean(svm.pred!=w_trainset$class)
> svm_acc=1-svm_err
> svm_acc*100
[1] 94.68556
> svm_err*100
[1] 5.314438
> svm.testpred = predict(svmfit_radial,w_testset)
> table(svm.testpred ,w_testset$class)
svm.testpred    n    w
           n 1371   81
           w    0    0
> svm_testerr=mean(svm.testpred!=w_testset$class)
> svm_testacc=1-svm_testerr
> svm_testacc*100
[1] 94.42149
> svm_testerr*100
[1] 5.578512
```

Support vector machine using radial kernel with different cost (0.1,1,10,100,1000):

Code:

```
>svmtune_radial=tune(svm,class~.,data=w_trainset,kernel="radial",gamma=c(0.5,1,
 2,3,4),ranges=list(cost=c(0.1 ,1 ,10 ,100 ,1000)))
> summary(svmtune_radial)
Parameter tuning of ‘svm’:
- sampling method: 10-fold cross validation
- best parameters:
 cost
   10
- best performance: 0.01358154
- Detailed performance results:
   cost      error  dispersion
1 1e-01 0.05314535 0.011736730
2 1e+00 0.01918888 0.007387365
3 1e+01 0.01358154 0.007260899
4 1e+02 0.01564993 0.005920823
5 1e+03 0.01889738 0.007904867

> bestmodel_rad=svmtune_radial$best.model
> summary(bestmodel_rad)
Call:
best.tune(method = svm, train.x = class ~ ., data = w_trainset,
    ranges = list(cost = c(0.1, 1, 10, 100, 1000)), kernel = "radial",
    gamma = c(0.5, 1, 2, 3, 4))
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
Number of Support Vectors:  173
 ( 94 79 )
Number of Classes:  2
Levels:
 n w
> pred=predict(bestmodel_rad,w_trainset)
> table(pred,w_trainset$class)
pred    n    w
  n 3196   17
  w   11   163
> svmtrain_pred_err=mean(pred!=w_trainset$class)
> svmtrain_pred_acc=1-svmtrain_pred_err
> svmtrain_pred_acc *100
[1] 99.17331
> svmtrain_pred_err*100
[1] 0.8266903
> pred_test=predict(bestmodel_rad,w_testset)
> table(pred_test,w_testset$class)
pred_test    n    w
       n 1364   13
       w    7   68
> svmtest_pred_testerr=mean(pred_test!=w_testset$class)
> svmtest_pred_testacc=1-svmtest_pred_testerr
> svmtest_pred_testacc*100
[1] 98.62259
> svmtest_pred_testerr*100
[1] 1.37741
```

Result of Support vector machine using Radial kernel for cost 0.01:

Training Accuracy: 94.68556

Training Error: 5.314438

Testing Accuracy: 94.42149

Testing error: 5.578512

Result of Support vector machine using Radial kernel for best case:

Training Accuracy: 99.17331

Training Error: 0.8266903

Testing Accuracy: 98.62259

Testing error: 1.37741

Support vector machine using polynomial kernel with cost (0.01):

Code:

```
> svm.poly_1 = svm(class ~ ., data = w_trainset,gamma=1, kernel = "poly", degree =
2,cost=0.01)
```

```
> summary(svm.poly_1)
Call:
svm(formula = class ~ ., data = w_trainset, gamma = 1, kernel = "poly", degree = 2, cost = 0.01)
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  0.01
     degree:  2
     coef.0:  0
Number of Support Vectors:  372
( 192 180 )
Number of Classes:  2
Levels:
 n w
> tr_pred = predict(svm.poly_1, w_trainset)
> table(tr_pred,w_trainset$class)
tr_pred    n    w
      n 3207   180
      w    0    0
> err=mean(tr_pred! =w_trainset$class)
> acc=1-err
> acc*100
[1] 94.68556
> err*100
[1] 5.314438
> ts_pred = predict(svm.poly_1,w_testset)
> table(ts_pred,w_testset$class)
ts_pred    n    w
      n 1371   81
      w    0    0
> err_test=mean(ts_pred!=w_testset$class)
> acc_test=1-err_test
> acc_test*100
[1] 94.42149
> err_test*100
[1] 5.578512
```

Result of Support vector machine using Polynomial kernel for cost=0.01 gamma=1 degree=2:

Training Accuracy: 94.59699

Training Error: 5.403012

Testing Accuracy: 94.6281

Testing error:5.371901

Support vector machine using Polynomial kernel with different cost (0.1,1,10,100,1000):

```
>svmtune_polynomial=tune(svm,class~.,data=w_trainset,kernel="poly",gamma=c(0.5,1,2,3,4),ranges
=list(cost=c(0.1 ,1 ,10 ,100 ,1000)))
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
WARNING: reaching max number of iterations
> summary(svmtune_polynomial)
Parameter tuning of ‘svm’:
- sampling method: 10-fold cross validation
- best parameters:
 cost
 1000
- best performance: 0.02450472
- Detailed performance results:
   cost      error   dispersion
1 1e-01 0.05314709 0.007766045
2 1e+00 0.05166867 0.009044577
3 1e+01 0.04163132 0.008966250
4 1e+02 0.03070028 0.009125132
5 1e+03 0.02450472 0.008909126
> bestmodel_pol=svmtune_polynomial$best.model
> summary(bestmodel_pol)
```

```
Call:
best.tune(method = svm, train.x = class ~ ., data = w_trainset,
    ranges = list(cost = c(0.1, 1, 10, 100, 1000)), kernel = "poly", gamma = c(0.5, 1, 2, 3, 4))
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  1000
     degree:  3
     coef.0:  0
Number of Support Vectors:  220
( 119 101 )
Number of Classes:  2
Levels:
 n w
> pred=predict(bestmodel_pol, w_trainset)
> table(pred,w_trainset$class)
pred    n    w
   n 3196   48
   w   11  132
> svmtrain_pred_err=mean(pred!=w_trainset$class)
> svmtrain_pred_acc=1-svmtrain_pred_err
> svmtrain_pred_acc *100
[1] 98.25805
> svmtrain_pred_err*100
[1] 1.741955
> pred_test=predict(bestmodel_rad,w_testset)
> table(pred_test,w_testset$class)
pred_test    n    w
        n 1364   13
        w    7   68
> svmtest_pred_testerr=mean(pred_test!=w_testset$class)
> svmtest_pred_testacc=1-svmtest_pred_testerr
> svmtest_pred_testacc*100
[1] 98.62259
> svmtest_pred_testerr*100
[1] 1.37741
```

Result of Support vector machine using Polynomial kernel for different cost (0.1,1,10,100,1000) and gammas=(0.5,1,2,3,4) for bestcase:

Training Accuracy: 98.25805

Training Error: 1.741955

Testing Accuracy: 98.62259

Testing error: 1.37741

Comparison of multiple classification techniques:

Classification Method	Training Accuracy	Training Error	Testing Accuracy	Testing Error
Holdout Method	98.34662	1.653381	97.79614	2.203817
Bagging	100	0	98.62259	1.37741
Random Forest	100	0	100	0
Naïve Base	90.37496	9.625037	89.73829	10.26171
SVM linear	96.87039	3.129613	96.76309	3.236915
SVM Polynomial	98.13995	1.860053	98.69146	1.30854
SVM Radial	99.20283	0.7971656	98.69146	1.30854

Potential Performance Issues and Possible Future Study:

We didn’t face many potential performance issues. Decision tree code using Hold Out Method, Random Forest and Naive Bayes classifier were very fast, but while running SVM Polynomial it took 5 minutes 32 seconds. This dataset is giving good fit for all classification techniques.

Possibly we would like to implement our study in the Gardening/Agriculture based on cloud integrated with IOT’s that would fetch real-time data, a continuous evaluation is conducted on plant status based on the analysis of input data and water is passed to the plant based on the status of the plant thereby saving water resources and protecting the crop health.

Conclusion:

In this project, we have performed many classification techniques such as Decision tree hold out, Random forest, Bagging, Naïve Bayes and Support vector machine. We have analyzed the accuracy and error rates obtained from the techniques. Among all the techniques Random Forest, Holdout and

Bagging methods are the best classification for our data set which provide error rate less than 2.3% and accuracy greater than 97.7% when compared to other classifiers.

REFERENCES:

1. https://archive.ics.uci.edu/ml/datasets/Wilt
2. https://towardsdatascience.com/ensemble-methods-bagging-boosting-and-stacking-c9214a10a205
3. https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb
4. https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
5. https://towardsdatascience.com/naive-bayes-in-machine-learning-f49cc8f831b4
6. https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f

Appendix

```
data=read.csv("C:/Users/vamshi/Desktop/New folder (2)/prjctwilt.csv",header =TRUE)
fix(data)
RNGkind(sample.kind = "Rounding")
set.seed(123)
library(e1071)
library(ISLR)
library(class)
library(plotrix)
attach(data)
dim(data)
wiltdata=sample(1:nrow(data),3387)
w_trainset=data[wiltdata,]
w_testset=data[-wiltdata,]
#Decision Tree Holdout Method  Implementation
library(tree)
wilt.test=class[-wiltdata]
wilt.train=class[wiltdata]
tree.dt=tree(class~.,data,subset =wiltdata)
summary(tree.dt)
plot(tree.dt)
text(tree.dt,pretty = 0)
tree.pred.train=predict(tree.dt,w_trainset,type = "class")
table(tree.pred.train,wilt.train)
tree.pred.train.acc=mean(tree.pred.train==wilt.train)
tree.pred.train.err=1-tree.pred.train.acc
tree.pred.train.acc*100
tree.pred.train.err*100
tree.pred=predict(tree.dt,w_testset,type = "class")
table(tree.pred,wilt.test)
tree.pred.test.acc=mean(tree.pred==wilt.test)
tree.pred.test.err=1-tree.pred.test.acc
tree.pred.test.acc*100
tree.pred.test.err*100
#Bagging  Algorithm implementation
library(randomForest)
rf_bag=randomForest(class~.,data,subset =wiltdata,ntree=500,mtry=3)
rf_train_pred=predict(rf_bag,w_trainset,type = "class")
table(rf_train_pred,wilt.train)
rf_bag_acc=mean(rf_train_pred==wilt.train)
rf_bag_err=1-rf_bag_acc
rf_bag_acc*100
rf_bag_err*100
rf_test_pred=predict(rf_bag,w_testset,type = "class")
table(rf_test_pred,wilt.test)
rf_bag_test_acc=mean(rf_test_pred==wilt.test)
rf_bag_test_err=1-rf_bag_test_acc
rf_bag_test_acc*100
rf_bag_test_err*100
#Random Forest Implementation with change in M and N values
library(randomForest)
library(ISLR)
library(tree)
tree.random_1=randomForest(class~.,w_testset, ntree=500,mtry=sqrt(16))
tree.pred=predict(tree.random_1,w_testset,type="class")
table(tree.pred,w_testset$class)
ran_test_err=mean(tree.pred!=w_testset$class)
ran_test_acc=1-ran_test_err
ran_test_err
ran_test_acc
tree.random_1t=randomForest(class~.,w_trainset, ntree=500,mtry=sqrt(16))
tree.pred=predict(tree.random_1t,w_trainset,type="class")
table(tree.pred,w_trainset$class)
ran_train_err=mean(tree.pred!=w_trainset$class)
ran_train_acc=1-ran_train_err
ran_train_err
ran_train_acc
```

```
tree.random_2=randomForest(class~.,w_testset, ntree=100,mtry=sqrt(4))
tree.pred=predict(tree.random_2,w_testset,type="class")
table(tree.pred,w_testset$class)
ran_test_err_1=mean(tree.pred!=w_testset$class)
ran_test_acc_1=1-ran_test_err
ran_test_err_1
ran_test_acc_1
tree.random_2t=randomForest(class~.,w_trainset, ntree=100,mtry=sqrt(4))
tree.pred=predict(tree.random_2t,w_trainset,type="class")
table(tree.pred,w_trainset$class)
ran_train_err_1=mean(tree.pred!=w_trainset$class)
ran_train_acc_1=1-ran_train_err
ran_train_acc_1
ran_train_err_1
tree.random_3=randomForest(class~.,w_testset, ntree=1000,mtry=sqrt(9))
tree.pred=predict(tree.random_3,w_testset,type="class")
table(tree.pred,w_testset$class)
ran_test_err_2=mean(tree.pred!=w_testset$class)
ran_test_acc_2=1-ran_test_err
ran_test_acc_2
ran_test_err_2
tree.random_3t=randomForest(class~.,w_trainset, ntree=1000,mtry=sqrt(9))
tree.pred=predict(tree.random_3t,w_trainset,type="class")
table(tree.pred,w_trainset$class)
ran_train_err_2=mean(tree.pred!=w_trainset$class)
ran_train_acc_2=1-ran_train_err
ran_train_acc_2
ran_train_err_2
#Decision Tree Using Boosting
library(gbm)
data=read.csv("C:/Users/vamshi/Desktop/New folder (2)/prjctwilt.csv",header =TRUE)
data$class=ifelse(data$class=="w",1,0)
fix(data)
RNGkind(sample.kind = "Rounding")
set.seed(123)
wiltdata=sample(1:nrow(data),3387)
w_trainset=data[wiltdata,]
w_testset=data[-wiltdata,]
tree.wilt=gbm(class~., w_trainset, distribution="gaussian",n.trees=500,interaction.depth=4)
tree.pred.prob=predict(tree.wilt,w_testset, n.trees=500, type="response")
tree.pred=ifelse(tree.pred.prob<0.04, "Yes", "No")
table(w_testset$class,tree.pred)
err_test=mean(tree.pred!=w_testset$class)
acc_test=1-err_test
err_test*100
acc_test*100
#Implementation by using Navies Bayes Algorithm
Naive_Bayes_Model=naiveBayes(class~.,w_trainset)
NB_train= predict(Naive_Bayes_Model, w_trainset)
table(NB_train,w_trainset$class)
NB_error=mean(NB_train!=w_trainset$class)
NB_acc=1-NB_error
NB_acc*100
NB_error*100
NB_test = predict(Naive_Bayes_Model,w_testset)
table(NB_test,w_testset$class)
NB_testerr=mean(NB_test!=w_testset$class)
NB_testacc=1- NB_testerr
NB_testacc*100
NB_testerr*100
#SVM
library(e1071)
set.seed(123)
svmfit=svm(class~., data=w_trainset, kernel="linear", cost=0.01)
summary(svmfit)
svm.pred = predict(svmfit,w_trainset)
table(svm.pred,w_trainset$class)
svm_err=mean(svm.pred!=w_trainset$class)
svm_acc=1-svm_err
svm_acc*100
svm_err*100
svm.testpred = predict(svmfit,w_testset)
table(svm.testpred ,w_testset$class)
svm_testerr=mean(svm.testpred!=w_testset$class)
svm_testacc=1-svm_testerr
svm_testacc*100
svm_testerr*100
#BEST MODEL LINEAR KERNEL
```

```
tune.out=tune(svm,class~., data=w_trainset,kernel="linear",
ranges=list(cost=c(0.001,0.01,0.1,1,10,100)))
summary(tune.out)
bestmod=tune.out$best.model
summary(bestmod)
train.pred = predict(bestmod,w_trainset)
table(train.pred ,w_trainset$class)
best_error=mean(train.pred!=w_trainset$class)
best_acc=1-best_error
best_acc*100
best_error*100
test.pred = predict(bestmod,w_testset)
table(test.pred ,w_testset$class)
best_testerr=mean(test.pred!=w_testset$class)
best_testacc=1-best_testerr
best_testacc*100
best_testerr*100
#RADIAL KERNEL
svmfit_radial=svm(class~., data=w_trainset, kernel="radial",gamma=1,cost=0.01)
summary(svmfit_radial)
svm.pred = predict(svmfit_radial,w_trainset)
table(svm.pred,w_trainset$class)
svm_err=mean(svm.pred!=w_trainset$class)
svm_acc=1-svm_err
svm_acc*100
svm_err*100
svm.testpred = predict(svmfit_radial,w_testset)
table(svm.testpred ,w_testset$class)
svm_testerr=mean(svm.testpred!=w_testset$class)
svm_testacc=1-svm_testerr
svm_testacc*100
svm_testerr*100
#RADIAL KERNEL BEST MODEL
svmtune_radial=tune(svm,class~.,data=w_trainset,,
kernel="radial",gamma=c(0.5,1,2,3,4),ranges=list(cost=c(0.1 ,1 ,10 ,100 ,1000)))
summary(svmtune_radial)
bestmodel_rad=svmtune_radial$best.model
summary(bestmodel_rad)
pred=predict(bestmodel_rad,w_trainset)
```

```
table(pred,w_trainset$class)
svmtrain_pred_err=mean(pred!=w_trainset$class)
svmtrain_pred_acc=1-svmtrain_pred_err
svmtrain_pred_acc *100
svmtrain_pred_err*100
pred_test=predict(bestmodel_rad,w_testset)
table(pred_test,w_testset$class)
svmtest_pred_testerr=mean(pred_test!=w_testset$class)
svmtest_pred_testacc=1-svmtest_pred_testerr
svmtest_pred_testacc*100
svmtest_pred_testerr*100
#POLYNOMIAL KERNEL
svm.poly_1 = svm(class ~ ., data = w_trainset,gamma=1, kernel = "poly", degree = 2,cost=0.01)
summary(svm.poly_1)
tr_pred = predict(svm.poly_1, w_trainset)
table(tr_pred,w_trainset$class)
err=mean(tr_pred!=w_trainset$class)
acc=1-err
acc*100
err*100
ts_pred = predict(svm.poly_1,w_testset)
table(ts_pred,w_testset$class)
err_test=mean(ts_pred!=w_testset$class)
acc_test=1-err_test
acc_test*100
err_test*100
#POLYNOMIAL KERNEL BEST MODEL
svmtune_polynomial=tune(svm,class~.,data=w_trainset,kernel="poly",gamma=c(0.5,1,2,3,4),ranges=
list(cost=c(0.1 ,1 ,10 ,100 ,1000)))
summary(svmtune_polynomial)
bestmodel_pol=svmtune_polynomial$best.model
summary(bestmodel_pol)
pred=predict(bestmodel_pol,w_trainset)
table(pred,w_trainset$class)
svmtrain_pred_err=mean(pred!=w_trainset$class)
svmtrain_pred_acc=1-svmtrain_pred_err
svmtrain_pred_acc *100
svmtrain_pred_err*100
pred_test=predict(bestmodel_rad,w_testset)
```

```
table(pred_test,w_testset$class)
svmtest_pred_testerr=mean(pred_test!=w_testset$class)
svmtest_pred_testacc=1-svmtest_pred_testerr
svmtest_pred_testacc*100
svmtest_pred_testerr*100
```