Forest Cover Type Prediction using Machine Learning Algorithms

Project Report

By
Shikhar Srivastava(20BCE0968)
Sri Charan Gundla(20BKT0149)
Vishwajeet Prasad(20BCE0560)

Under the guidance of
Prof. Gunavathi C
School of Computer Science and Technology
VIT, Vellore

Table of Contents

Abstract

Mapping of forest cover and determining the various flora and fauna that is found there is the key to forest planning and management. This info is also very crucial for management of natural resources, development strategies and tracking change in the distribution of forest area over the years.

The idea of this project is to make the mapping and classification of forest areas easy using machine learning algorithms using the data that can easily be made available by Google Earth Engine using satellite images.

In this project, we will be using 3 different machine learning algorithms and training them using the same set of data and comparing the accuracy of each of them with one another to determine which machine learning algorithm works the best for this job. The 3 machine learning algorithms are kNN (k Nearest Neighbours), SVM (Support Vector Machines) and Random Forests.

Algorithms Used

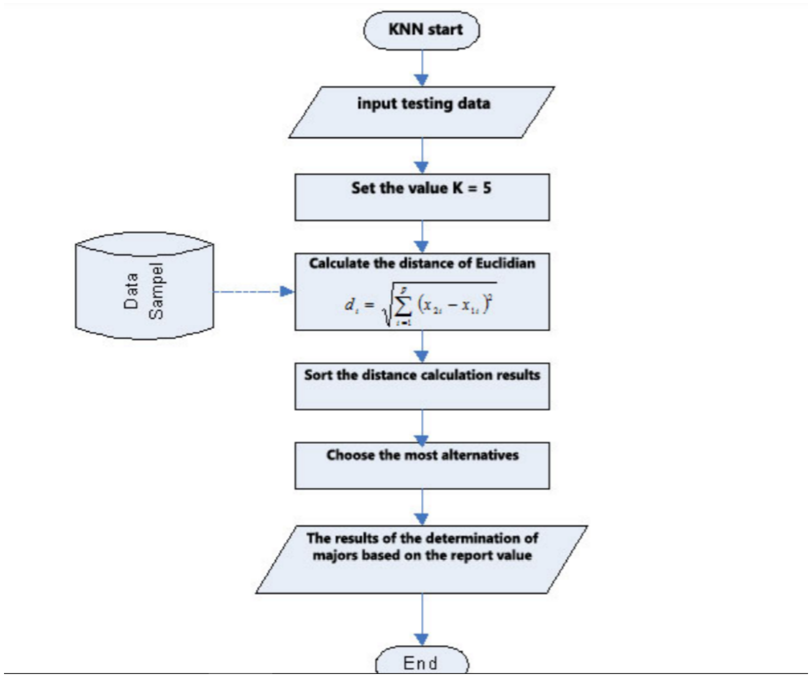
1. K Nearest Neighbours(KNN)
2. Support Vector Machines(SVM)
3. Random Forests
    a. Decision Tree

➔ Now we're gonna briefly explain each of the algorithms used.

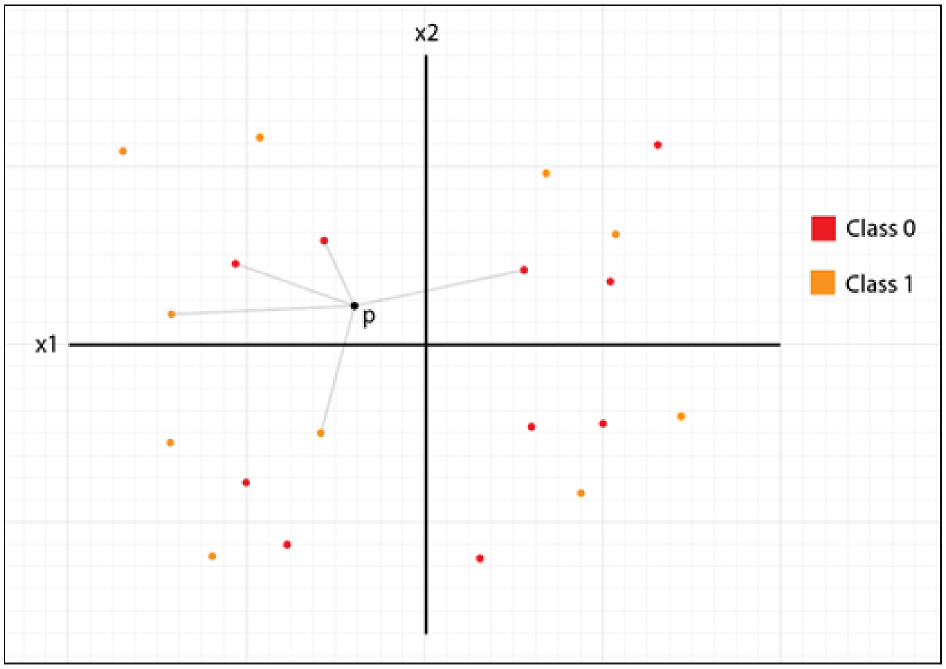
k Nearest Neighbours(kNN):

k Nearest Neighbours algorithm falls under the Supervised Learning category and is used for classification (most commonly) and regression. It is a versatile algorithm also used for imputing missing values and resampling datasets. As the name (k Nearest Neighbours) suggests it considers k Nearest Neighbours (Data points) to predict the class or continuous value for the new datapoint.

To classify the test data the algorithm plots the point for which we need to make the prediction on the plane and finds the k (value specified by the user) nearest neighbours to that point and check their class values. Then it uses a voting mechanism to decide what class should be assigned to that data point.


KNN start
input testing data
Set the value K = 5
Data Sampel
Calculate the distance of Euclidian
$d_i = \sqrt{\sum_{i=1}^{p} (x_{2i} - x_{1i})^2}$
Sort the distance calculation results
Choose the most alternatives
The results of the determination of majors based on the report value
End

Let's assume a sample dataset with 20 rows having two attributes x1 and x2 and class y which can hold the value 0 or 1. Assume k = 5. To predict the class of point 'p' the algorithm then finds the 5 nearest neighbours to ‘p’. The class of ‘p’ is whatever is the class of the majority of these neighbours. Here the class of ‘p’ would be 0 because 3 out of 5 neighbours are of class 0.

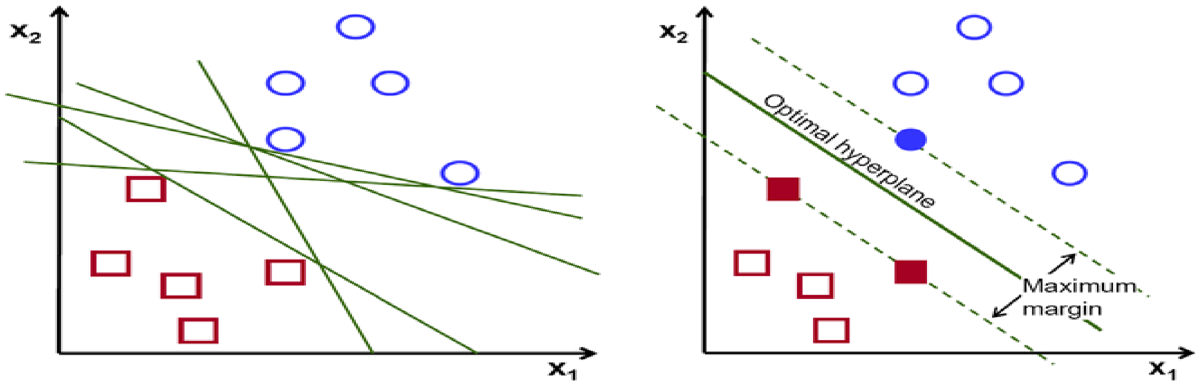


Search Vector Machine(SVM):

Support Vector Machine, abbreviated as SVM can be used for both regression and classification tasks. But, it is widely used in classification objectives.

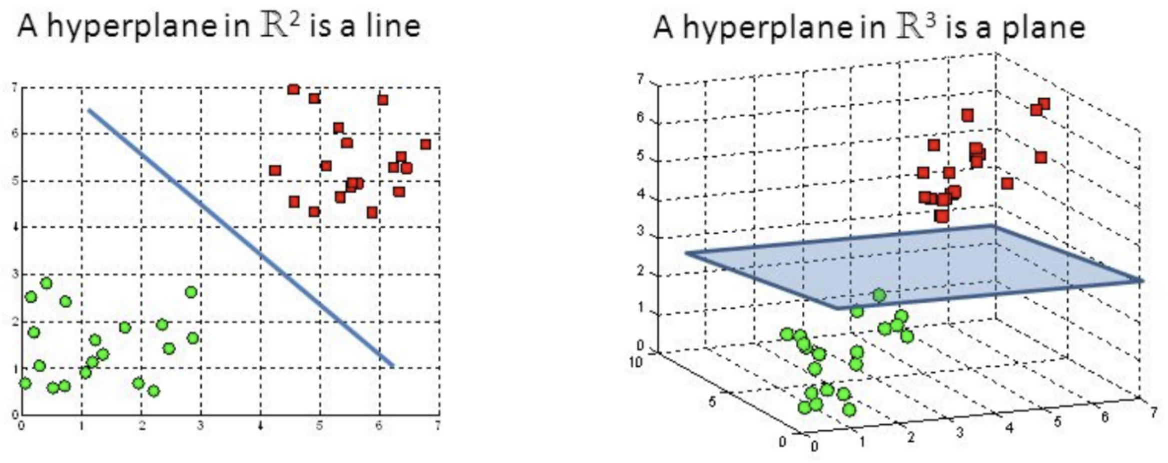
The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.

Possible Hyperplanes



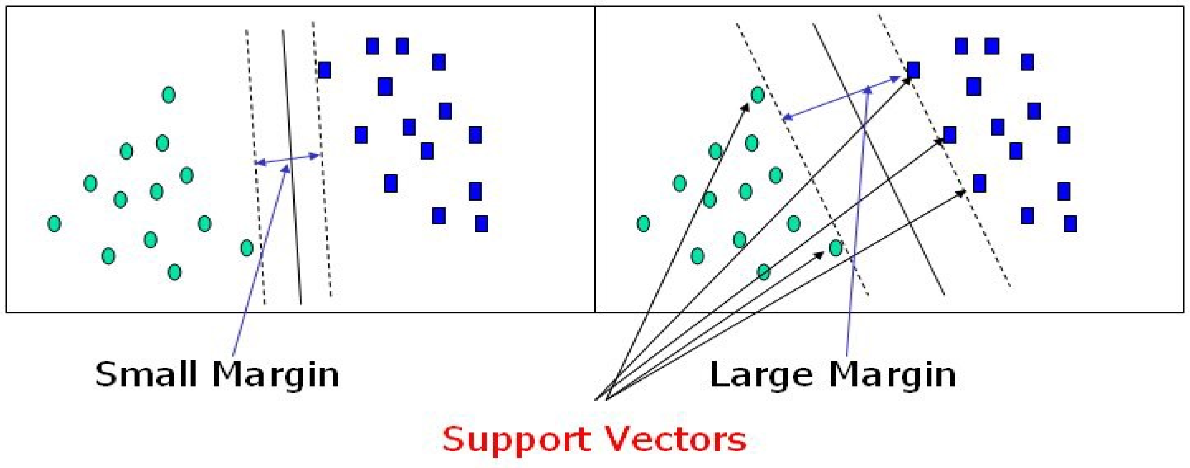
To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

Hyperplanes and Support Vectors:



Hyperplanes in 2D and 3D feature space.

Hyperplanes are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.



Support Vectors

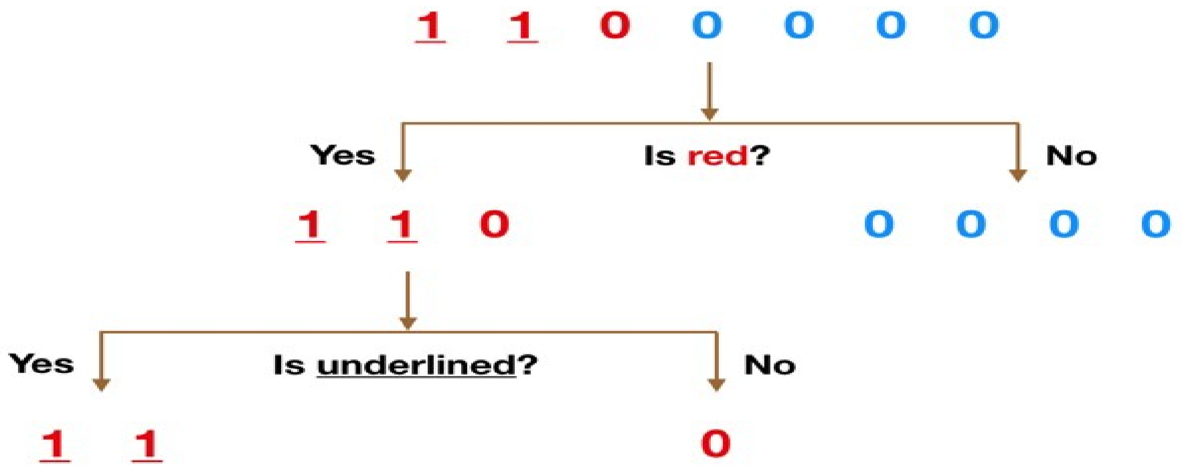
Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

Random Forest:

We already have gone through the two classification algorithms namely, k Nearest Neighbours(kNN) and Support Vector Machine(SVM). But near the top of the classifier hierarchy is the random forest classifier. We will first start by explaining about decision trees.

Decision Tree:

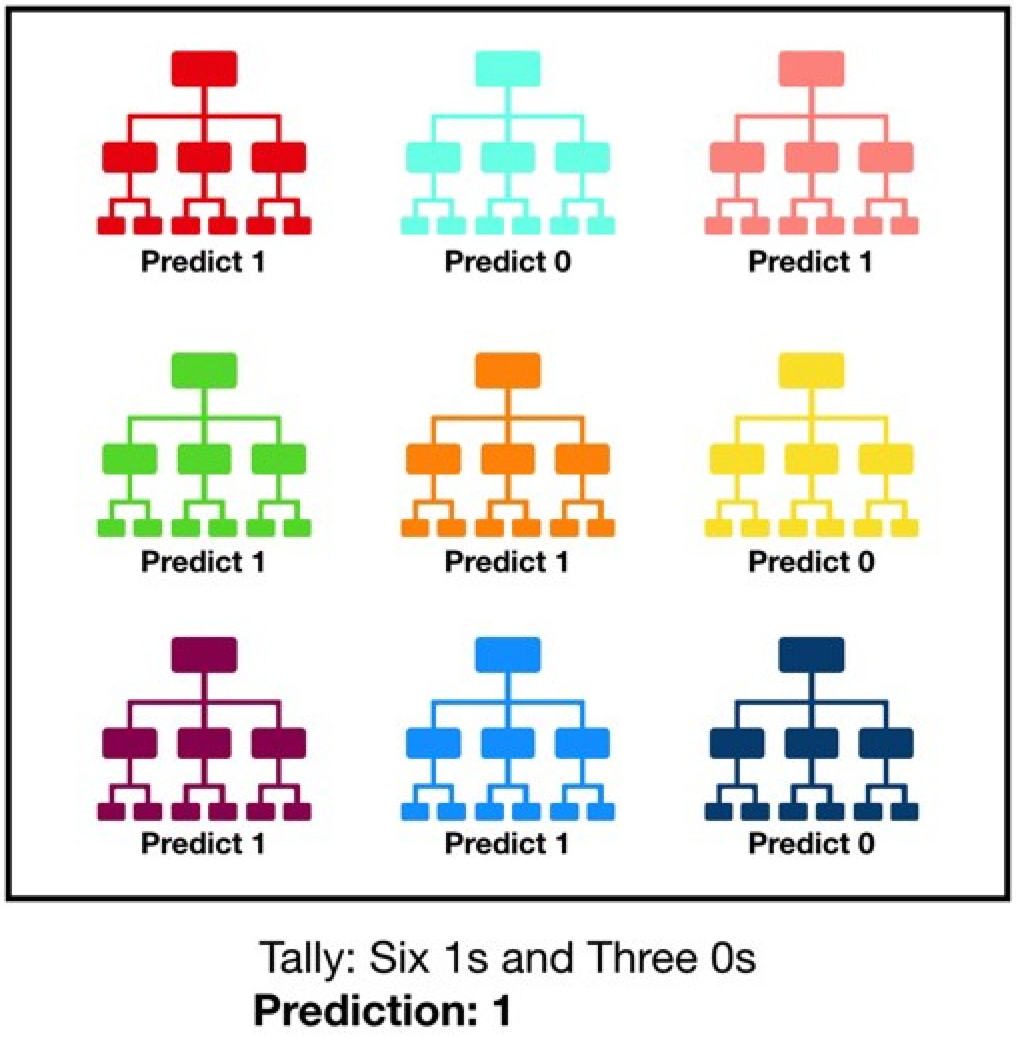
It's probably much easier to understand how a decision tree works through an example. Imagine that our dataset consists of the numbers at the top of the figure to the left. We have two 1s and five 0s (1s and 0s are our classes) and desire to separate the classes using their features. The features are color (red vs. blue) and whether the observation is underlined or not. So how can we do this?
Color seems like a pretty obvious feature to split by as all but one of the 0s are blue. So we can use the question, "Is it red?" to split our first node. You can think of a node in a tree as the point where the path splits into two — observations that meet the criteria go down the Yes branch and ones that don't go down the No branch.
The No branch (the blues) is all 0s now so we are done there, but our Yes branch can still be split further. Now we can use the second feature and ask, "Is it underlined?" to make a second split. The two 1s that are underlined go down the Yes sub branch and the 0 that is not underlined goes down the right sub branch and we are all done. Our decision tree was able to use the two features to split up the data perfectly. Victory!



Simple Decision Tree Example

Obviously in real life our data will not be this clean but the logic that a decision tree employs remains the same. At each node, it will ask —
What feature will allow me to split the observations at hand in a way that the resulting groups are as different from each other as possible (and the members of each resulting subgroup are as similar to each other as possible)?

The Random Forest Classifier:
Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction (see figure below).



Visualization of a Random Forest Model Making a Prediction

The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science speak, the reason that the random forest model works so well is:

A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.
The low correlation between models is the key. Just like how investments with low correlations (like stocks and bonds) come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason for this wonderful effect is that the trees protect each other from their individual errors (as long as they don’t constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction. So the prerequisites for random forest to perform well are:

1. There needs to be some actual signal in our features so that models built using those features do better than random guessing.
2. The predictions (and therefore the errors) made by the individual trees need to have low correlations with each other.

Code Implementation and Output

1. k-Nearest Neighbours

Code:

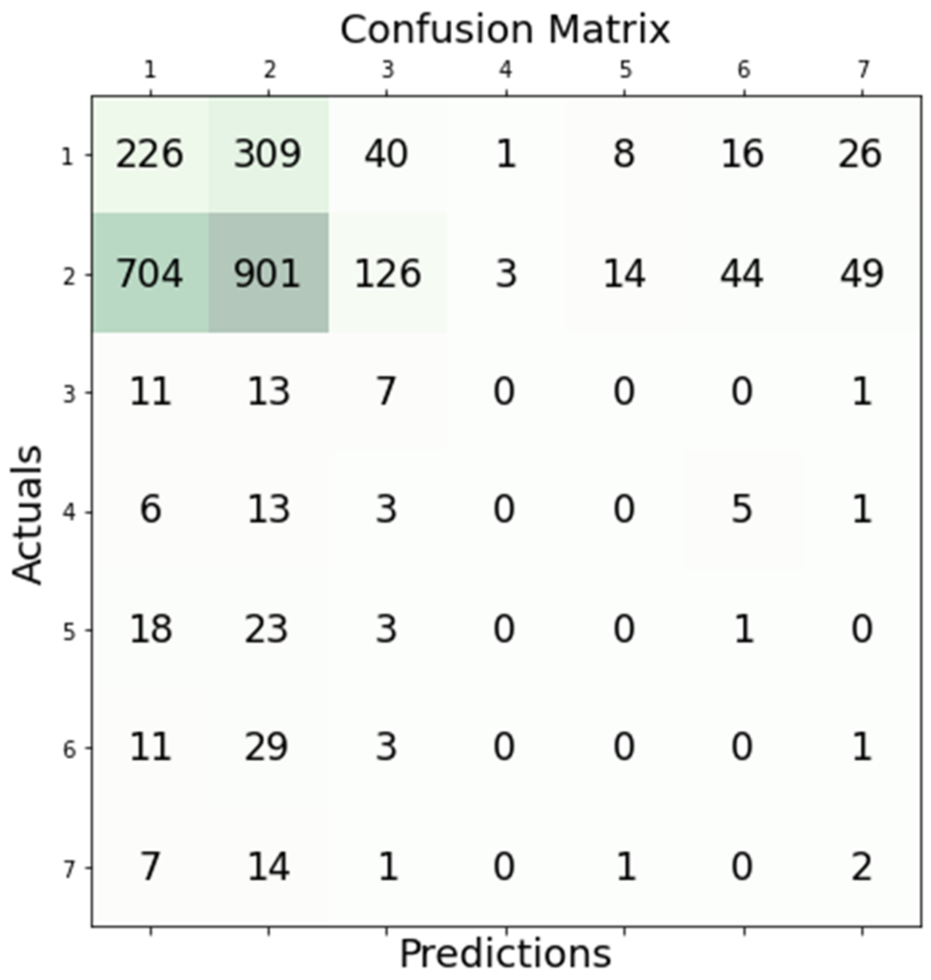
```
# Libraries for easy visualization and manipulation of data
import numpy as np
import pandas as pd

# Function for the k nearest neighbors classification
from sklearn.neighbors import KNeighborsClassifier

# Functions for splitting the data training & testing datasets & for
checking the accuracy of our predictions
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Library to plot the confusion matrix for our predictions
import matplotlib.pyplot as plt
# Function to convert the list consisting of the actual and predicted
labels into a proper matrix format
def matrix(y_test, pred):

     # Combining the lists consisting the actual & predicted class
labels
     val = pd.concat([y_test, pred], axis = 1)
     val.columns = ['Actual', 'Predicted']

     # Counting the values of each element of the matrix
     grp = val.groupby(['Actual', 'Predicted'])
     count = grp.apply(lambda x: len(x))

     indexes = count.index.to_frame(index = False)

     count.index = np.arange(count.shape[0])

     count = pd.concat([indexes, count], axis = 1)
     count.columns = ['Actual', 'Predicted', 'Count']
```

```
      # Coverting the dataframe into a matrix
      mat = count.pivot(index = 'Actual', columns = 'Predicted',
values = 'Count')
      mat.fillna(0, inplace = True)

      return mat
# Function to plot the confusion matrix
def show_plot(mat):

      # Designing the visual of the confusion matrix
      fig, ax = plt.subplots(figsize = (7, 7))
      ax.matshow(mat, cmap = plt.cm.Greens, alpha = 0.3)

      for i in range(mat.shape[0]):
      for j in range(mat.shape[1]):
             ax.text(x = j,
                     y = i,
                     s = int(mat.values[i, j]),
                     va = 'center',
                     ha = 'center',
                     size = 'xx-large')

    ax.xaxis.set_ticks(np.arange(mat.shape[1]))
      ax.yaxis.set_ticks(np.arange(mat.shape[0]))

      # Assigning the labels & the title for the matrix
    ax.xaxis.set_ticklabels([str(int(x)) for x in mat.columns])
    ax.yaxis.set_ticklabels([str(int(x)) for x in mat.index])

      plt.xlabel('Predictions', fontsize = 18)
      plt.ylabel('Actuals', fontsize = 18)

      plt.title('Confusion Matrix', fontsize = 18)

      # Putting the matrix on the output screen
      plt.show()
# Reading the data from a csv file & converting it to a dataframe
df = pd.read_csv('covtype.csv', skiprows = lambda x: x % 20 != 0)
```

```
cols_lst = df.columns

# Seperating the features from the class labels
x = df[cols_lst[0:14]]
y = df[cols_lst[-1]]

# Normalizing the data so that no one feature has more impact on the
calculations than the others
minmaxscale = lambda x: (x - x.min()) / (x.max() - x.min())
x_normalized = x.apply(minmaxscale)

# Spliting the data into traning & testing datasets
x_train, x_test, y_train, y_test = train_test_split(x_normalized, y,
test_size = 0.3, random_state = 5)
# Calling the function for the knn classifier
knn = KNeighborsClassifier(n_neighbors = 5, weights = 'distance')
knn.fit(x_train, y_train)

# Predicting the class labels for the testing data using the knn
classifier
pred = knn.predict(x_test)
pred = pd.Series(pred)

# Checking the accuracy of our predictions
print('Accuracy:', accuracy_score(y_test, pred))

# Plotting the confusion matrix for the predicted class labels
mat = matrix(y_test, pred)
show_plot(mat)
```

Output:

Accuracy: **0.7967871485943775**



	1	2	3	4	5	6	7
1	226	309	40	1	8	16	26
2	704	901	126	3	14	44	49
3	11	13	7	0	0	0	1
4	6	13	3	0	0	5	1
5	18	23	3	0	0	1	0
6	11	29	3	0	0	0	1
7	7	14	1	0	1	0	2

2. **Support Vector Machines**

Code:

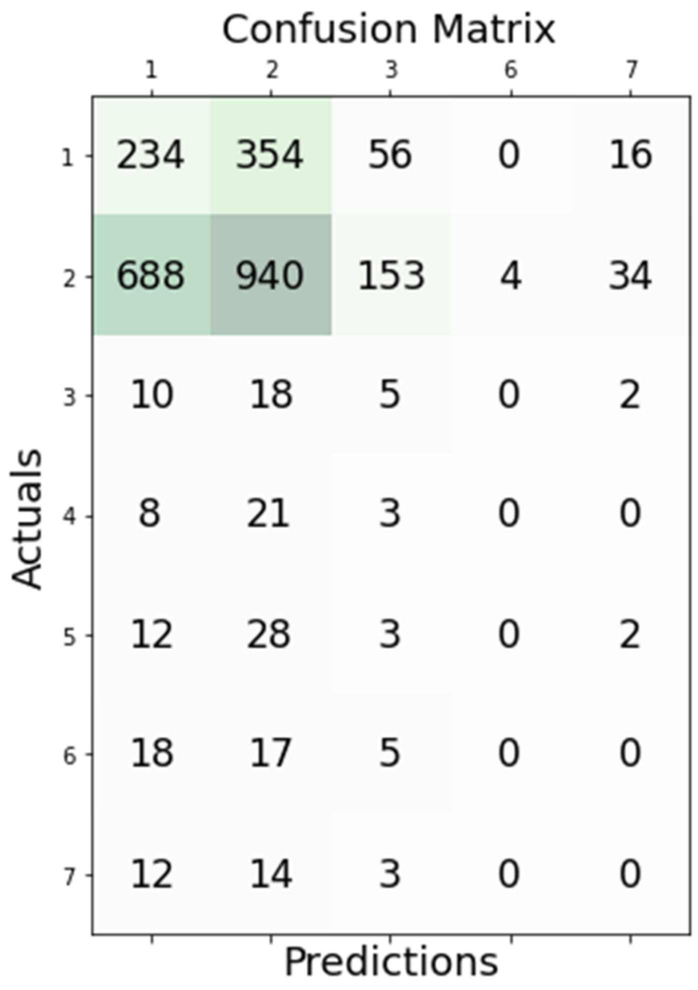
```
# Libraries for easy visualization and manipulation of data
import numpy as np
import pandas as pd

# Function for the support vector classification
from sklearn.svm import SVC

# Functions for splitting the data training & testing datasets & for
checking the accuracy of our predictions
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Library to plot the confusion matrix for our predictions
import matplotlib.pyplot as plt
# Function to convert the list consisting of the actual and predicted
labels into a proper matrix format
def matrix(y_test, pred):

      # Combining the lists consisting the actual & predicted class
labels
      val = pd.concat([y_test, pred], axis = 1)
      val.columns = ['Actual', 'Predicted']

      # Counting the values of each element of the matrix
      grp = val.groupby(['Actual', 'Predicted'])
      count = grp.apply(lambda x: len(x))

      indexes = count.index.to_frame(index = False)

      count.index = np.arange(count.shape[0])

      count = pd.concat([indexes, count], axis = 1)
      count.columns = ['Actual', 'Predicted', 'Count']

      # Coverting the dataframe into a matrix
      mat = count.pivot(index = 'Actual', columns = 'Predicted',
values = 'Count')
      mat.fillna(0, inplace = True)

      return mat
# Function to plot the confusion matrix
def show_plot(mat):

      # Designing the visual of the confusion matrix
      fig, ax = plt.subplots(figsize = (7, 7))
      ax.matshow(mat, cmap = plt.cm.Greens, alpha = 0.3)

      for i in range(mat.shape[0]):
      for j in range(mat.shape[1]):
            ax.text(x = j,
                    y = i,
                    s = int(mat.values[i, j]),
                    va = 'center',
                    ha = 'center',
                    size = 'xx-large')

    ax.xaxis.set_ticks(np.arange(mat.shape[1]))
     ax.yaxis.set_ticks(np.arange(mat.shape[0]))

     # Assigning the labels & the title for the matrix
    ax.xaxis.set_ticklabels([str(int(x)) for x in mat.columns])
    ax.yaxis.set_ticklabels([str(int(x)) for x in mat.index])

     plt.xlabel('Predictions', fontsize = 18)
     plt.ylabel('Actuals', fontsize = 18)

     plt.title('Confusion Matrix', fontsize = 18)

     # Putting the matrix on the output screen
     plt.show()
# Reading the data from a csv file & converting it to a dataframe
df = pd.read_csv('covtype.csv', skiprows = lambda x: x % 20 != 0)
cols_lst = df.columns

# Seperating the features from the class labels
x = df[cols_lst[0:14]]
y = df[cols_lst[-1]]

# Normalizing the data so that no one feature has more impact on the
calculations than the others
minmaxscale = lambda x: (x - x.min()) / (x.max() - x.min())
x_normalized = x.apply(minmaxscale)

# Spliting the data into traning & testing datasets
x_train, x_test, y_train, y_test = train_test_split(x_normalized, y,
test_size = 0.3, random_state = 30)
# Calling the function for the sv classifier
sv = SVC(kernel = 'poly', decision_function_shape = 'ovr')
sv.fit(x_train, y_train)
```

```
# Predicting the class labels for the testing data using the sv
classifier
pred = sv.predict(x_test)
pred = pd.Series(pred)

# Checking the accuracy of our predictions
print('Accuracy:', accuracy_score(y_test, pred))

# Plotting the confusion matrix for the predicted class labels
mat = matrix(y_test, pred)
show_plot(mat)
```

Output:

Accuracy: **0.725530694205393**



	1	2	3	6	7
1	234	354	56	0	16
2	688	940	153	4	34
3	10	18	5	0	2
4	8	21	3	0	0
5	12	28	3	0	2
6	18	17	5	0	0
7	12	14	3	0	0

3. Random Forests

Code:

```
# Libraries for easy visualization and manipulation of data
import numpy as np
import pandas as pd

# Function for the random forest classification
from sklearn.ensemble import RandomForestClassifier

# Functions for splitting the data training & testing datasets & for
checking the accuracy of our predictions
```

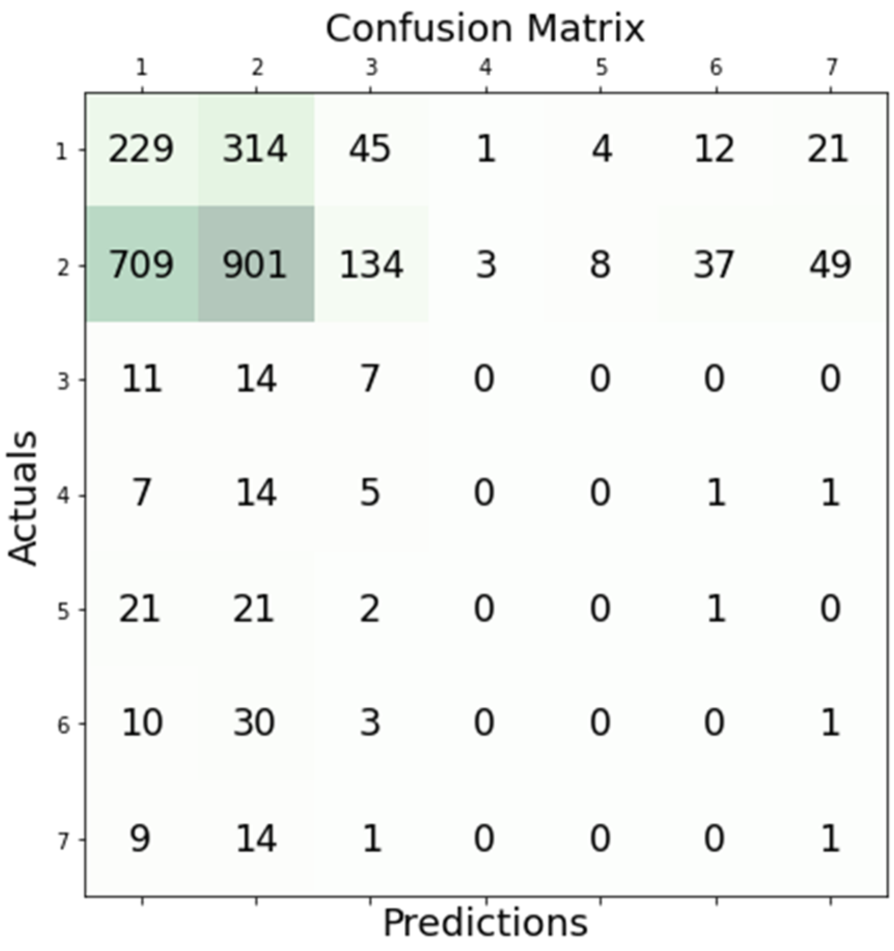
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Library to plot the confusion matrix for our predictions
import matplotlib.pyplot as plt
# Function to convert the list consisting of the actual and predicted
labels into a proper matrix format
def matrix(y_test, pred):

      # Combining the lists consisting the actual & predicted class
labels
      val = pd.concat([y_test, pred], axis = 1)
      val.columns = ['Actual', 'Predicted']

      # Counting the values of each element of the matrix
      grp = val.groupby(['Actual', 'Predicted'])
      count = grp.apply(lambda x: len(x))

      indexes = count.index.to_frame(index = False)

      count.index = np.arange(count.shape[0])

      count = pd.concat([indexes, count], axis = 1)
      count.columns = ['Actual', 'Predicted', 'Count']

      # Coverting the dataframe into a matrix
      mat = count.pivot(index = 'Actual', columns = 'Predicted',
values = 'Count')
      mat.fillna(0, inplace = True)

      return mat
# Function to plot the confusion matrix
def show_plot(mat):

      # Designing the visual of the confusion matrix
      fig, ax = plt.subplots(figsize = (7, 7))
      ax.matshow(mat, cmap = plt.cm.Greens, alpha = 0.3)

      for i in range(mat.shape[0]):
```

```
      for j in range(mat.shape[1]):
            ax.text(x = j,
                    y = i,
                    s = int(mat.values[i, j]),
                    va = 'center',
                    ha = 'center',
                    size = 'xx-large')

     ax.xaxis.set_ticks(np.arange(mat.shape[1]))
      ax.yaxis.set_ticks(np.arange(mat.shape[0]))

      # Assigning the labels & the title for the matrix
     ax.xaxis.set_ticklabels([str(int(x)) for x in mat.columns])
     ax.yaxis.set_ticklabels([str(int(x)) for x in mat.index])

      plt.xlabel('Predictions', fontsize = 18)
      plt.ylabel('Actuals', fontsize = 18)

      plt.title('Confusion Matrix', fontsize = 18)

      # Putting the matrix on the output screen
      plt.show()
# Reading the data from a csv file & converting it to a dataframe
df = pd.read_csv('covtype.csv', skiprows = lambda x: x % 20 != 0)
cols_lst = df.columns

# Seperating the features from the class labels
x = df[cols_lst[0:14]]
y = df[cols_lst[-1]]

# Normalizing the data so that no one feature has more impact on the
calculations than the others
minmaxscale = lambda x: (x - x.min()) / (x.max() - x.min())
x_normalized = x.apply(minmaxscale)

# Spliting the data into traning & testing datasets
x_train, x_test, y_train, y_test = train_test_split(x_normalized, y,
test_size = 0.3, random_state = 5)
# Calling the function for the rf classifier
```

```
rf = RandomForestClassifier(n_estimators = 50, criterion = 'entropy',
max_depth = 20)
rf.fit(x_train, y_train)

# Predicting the class labels for the testing data using the rf
classifier
pred = rf.predict(x_test)
pred = pd.Series(pred)

# Checking the accuracy of our predictions
print('Accuracy:', accuracy_score(y_test, pred))

# Plotting the confusion matrix for the predicted class labels
mat = matrix(y_test, pred)
show_plot(mat)
```

Output:

Accuracy: **0.8201950659781985**



Confusion Matrix

Actuals \ Predictions	1	2	3	4	5	6	7
1	229	314	45	1	4	12	21
2	709	901	134	3	8	37	49
3	11	14	7	0	0	0	0
4	7	14	5	0	0	1	1
5	21	21	2	0	0	1	0
6	10	30	3	0	0	0	1
7	9	14	1	0	0	0	1

Conclusion

After using the three machine learning algorithms, we have obtained the following accuracies for each of the algorithms.

K-Nearest Neighbors: **0.7967871485943775**

Support Vector Machines: **0.725530694205393**

Random Forests: **0.8201950659781985**

Hence, we can conclude that Random Forests algorithm is most efficient to predict forest cover type.

References

1. https://www.mdpi.com/1424-8220/18/1/18/htm
2. https://www.hindawi.com/journals/ijfr/2020/8896310/
3. https://www.hindawi.com/journals/tswj/2020/7402846/
4. https://www.hindawi.com/journals/js/2020/4817234/
5. https://ieeexplore.ieee.org/abstract/document/8861326/citations#citations
6. https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/
7. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/