Ramping Crop Yield in Districts of India

- Name: Akshara Shukla
- Class: AI43
- Genuine Challenge_02

Introduction

Agriculture and farming and one of the oldest and most valuable professionals in the world, they are termed as the back bone of a country. While in recent years, the number of suicide rates and protests amongs farmers in India has been increasing greatly. Therefore, with the aim of creating something meaningful which would help streamline the process of crop yield in different states and their corresponding districts, Harvestron.Inc model has been built. Through this, the farmers will be able to enter their district, state, crop, season, area and year to which as a result, the approximate crop yield would be predicted. This notebook consists of the Data Storage, Exploratory Data Analysis, Model Building Training and Testing phases.

The regression algorithms **RandomForest, AdaBoost and XGBoost regressors** were compared to their value of r2_score and the one with the high perfomance rate was chosen to make prediction. The reason to r2_score evaluation metric is used for compare the performance of our different regression models is as because it measures how nicely the independent variables are able to predict the variability or unseen chance in the dependent variables (Prediction of Crop Yield). Additionally, by using the web scraping tool **BeautifulSoup**, the necessary weather data has been scraped and shown as a notification which updates daily.

1 Exploratory Data Analysis

The first step is to import all the necessary libraries we will be using in this notebook to carry out our exploratory data analysis for finding key patterns and insights.

In [93]:

```
import pandas as pd #for data manipulation
import numpy as np
import matplotlib #for data visualization
import matplotlib.pyplot as plt
import seaborn as sns
import warnings #to ignore any unncessary warning messages
warnings.filterwarnings('ignore')

from sklearn.model_selection import GridSearchCV #to perform hyperparameter tuning
from sklearn.metrics import classification_report
from sklearn import metrics
from sklearn.model_selection import train_test_split #splitting the dataset
from sklearn import preprocessing

#Importing the terms for evaluating regression models.
from sklearn.metrics import mean_squared_error as MSE
from sklearn.metrics import mean_absolute_error as mae
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import r2_score, roc_auc_score

#Printing the versions of the libaries used.
print('pandas version:', pd.__version__)
print('matplotlib version:', matplotlib.__version__)
print('seaborn version:', sns.__version__)

#The below piece of code is known as the magic Jupyter command that allows us to display our graphs without the plt.show() command.
%matplotlib inline
```

```
pandas version: 1.0.5
matplotlib version: 3.2.2
seaborn version: 0.10.1
```

UTF-8 is an encoding system for Unicode. It can translate any Unicode character to a matching unique binary string, and can also translate the binary string back to a Unicode character. This is the meaning of “UTF”, or “Unicode Transformation Format.”

1.1 Importing the Data

Data Storage: Next we will be importing the dataset called *crop_production.csv* which was stored as a csv file on my computers root directory local drive folder *logs*, and was downloaded from the official Indian agriculture database website: https://data.gov.in/catalog/district-wise-season-wise-crop-production-statistics?filters%5Bfield_catalog_reference%5D=87631&format=json&offset=0&limit=6&sort%5Bcreated%5D=desc website.

In [94]:

```
df = pd.read_csv("crop_production.csv",encoding='utf-8')
#df = df.fillna(0)
df.columns = ['State','District', 'Year','Season', 'Crop','Area', 'Production']
df.head()
```

Out[94]:

	State	District	Year	Season	Crop	Area	Production
0	Andaman and Nicobar Islands	NICOBARS	2000	Kharif	Arecanut	1254.0	2000.0
1	Andaman and Nicobar Islands	NICOBARS	2000	Kharif	Other Kharif pulses	2.0	1.0
2	Andaman and Nicobar Islands	NICOBARS	2000	Kharif	Rice	102.0	321.0
3	Andaman and Nicobar Islands	NICOBARS	2000	Whole Year	Banana	176.0	641.0
4	Andaman and Nicobar Islands	NICOBARS	2000	Whole Year	Cashewnut	720.0	165.0

Our dataframe 'df' consists of the following variables:

- State: The different states of India where farming is practiced regularly.
- District: The districts of the states.
- Year: The particular year in which the crop was grown.
- Season: The different crop season practiced in India.
- Crop: The various crops grown in different districts of states.
- Area: The area measured in acres.
- Production: The production of the crop measured in kilograms per hectare (kg/ha)

In [95]:

```
print("The Crop Yield Production Dataset")
print("---------------------")
print()
print("As described on the original website:")
print()
print("\tThe data refers to district wise, crop wise, season wise and year wise data on crop covered area (Hectare) and production (Tonnes). The data is being used to study and analyse crop\n
production, production contribution to district/State/country, Agro-climatic zone wise performance, and high yield production order for crops, crop growing pattern and diversification. The\ns
ystem is also a vital source for formulating crop related schemes and assessing their impacts.")
print()
print("The target variable in our model are two. Firstly, the magnitude of Crop Yield which is being predicted by entering the respective fields. Additionally, after prediction, through websc
raping, to depict the notification of the current weather update in India, a notification stating 'WeatherUpdate', including values of Temperature, Humidity, Visibility and Dew Point are show
n in\nreal time.")
print()
print("We can use the dataset under the license,  National Data Sharing and Accessibility Policy (NDSAP). More info at: https://data.gov.in/catalog/district-wise-season-wise-crop-production-s
tatistics?filters%5Bfield_catalog_reference%5D=87631&format=json&offset=0&limit=6&sort%5Bcreated%5D=desc")
```

```
The Crop Yield Production Dataset
---------------------

As described on the original website:

        The data refers to district wise, crop wise, season wise and year wise data on crop covered area (Hectare) and production (Tonnes). The data is being used to study and an
alyse crop
production, production contribution to district/State/country, Agro-climatic zone wise performance, and high yield production order for crops, crop growing pattern and diversific
ation. The
system is also a vital source for formulating crop related schemes and assessing their impacts.

The target variable in our model are two. Firstly, the magnitude of Crop Yield which is being predicted by entering the respective fields. Additionally, after prediction, through
webscraping, to depict the notification of the current weather update in India, a notification stating 'WeatherUpdate', including values of Temperature, Humidity, Visibility and
Dew Point are shown in
real time.

We can use the dataset under the license,  National Data Sharing and Accessibility Policy (NDSAP). More info at: https://data.gov.in/catalog/district-wise-season-wise-crop-produc
tion-statistics?filters%5Bfield_catalog_reference%5D=87631&format=json&offset=0&limit=6&sort%5Bcreated%5D=desc
```

In [96]:

```
#Looking at the dataset and it's column/row values by using tail()
df.tail()
```

Out[96]:

	State	District	Year	Season	Crop	Area	Production
246086	West Bengal	PURULIA	2014	Summer	Rice	306.0	801.0
246087	West Bengal	PURULIA	2014	Summer	Sesamum	627.0	463.0
246088	West Bengal	PURULIA	2014	Whole Year	Sugarcane	324.0	16250.0
246089	West Bengal	PURULIA	2014	Winter	Rice	279151.0	597899.0
246090	West Bengal	PURULIA	2014	Winter	Sesamum	175.0	88.0

1.2 Exploring and Cleaning the dataset

After having imported the dataset, the aim is to gain a much better insight of the dataset, more specifically, try to catch key insights and patterns which will better help in predicting the value of the crop yield. In addition, by exploring and cleaning the dataset, it helps us to learn about which features are more important in coming to a final decision. We start by first understand the row and column values by using shape.

In [97]:

```
df.shape
print("Our dataset has the shape:", df.shape,"\n" "Meaning it consists of 246091 rows and 7 unique columns.")
```

```
Our dataset has the shape: (246091, 7)
Meaning it consists of 246091 rows and 7 unique columns.
```

In [98]:

```
#Understanding the data types of the different columns in our dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 246091 entries, 0 to 246090
Data columns (total 7 columns):
 #   Column      Non-Null Count   Dtype  
---  ------      --------------   -----  
 0   State       246091 non-null  object 
 1   District    246091 non-null  object 
 2   Year        246091 non-null  int64  
 3   Season      246091 non-null  object 
 4   Crop        246091 non-null  object 
 5   Area        246091 non-null  float64
 6   Production  242361 non-null  float64
dtypes: float64(2), int64(1), object(4)
memory usage: 13.1+ MB
```

It can be seen that our dataset consists of four datatypes mainly **State, District, Season,Crop** meaning they are our categorical columns. In addition, we also have two floats or decimal columns mainly **Area and Production** and we have one int column of our **year**.

In [99]:

```
df.describe()
```

Out[99]:

	Year	Area	Production
count	246091.000000	2.460910e+05	2.423610e+05
mean	2005.643018	1.200282e+04	5.825034e+05
std	4.952164	5.052340e+04	1.706581e+07
min	1997.000000	4.000000e-02	0.000000e+00
25%	2002.000000	8.000000e+01	8.800000e+01
50%	2006.000000	5.820000e+02	7.290000e+02
75%	2010.000000	4.392000e+03	7.023000e+03
max	2015.000000	8.580100e+06	1.250800e+09

Through using describe() function of the df, it can be seen that the time interval this dataset has been collected and which our model will train on. So, Harvestron model will be trained on data collected between 1997 and 2015 years. Moreover, the production state of each crop in different distrits also seem to have produced somewhere next to none to 1.25e+09 meaning adding 10 digits decimal places or 1250800000 to be exact. That is alot of production yield. It will be interesting to see which states and explicit crops are there which resulted into no production yield.

In [100]:

```
#Investigating if our dataset consists of any null values
df.isnull().sum()
```

Out[100]:

```
State            0
District         0
Year             0
Season           0
Crop             0
Area             0
Production    3730
dtype: int64
```

While having a closer look at the data values of our dataframe, it can be seen that the column **Production** consists of 3730 null values. Lets try to investigate what proportion of null values that is from the entire df. For this, I divide the null values with the total number of instances in our dataset. As a result, approximately 15% of null values are included in the production column. To tackle this problem, we will drop the NA values from the dataframe, this makes our df clean.

In [101]:

```
3730/246091
```

Out[101]:

```
0.015156994770227274
```

In [102]:

```
#Dropping the NA values in our df
df.dropna(subset=["Production"],axis=0,inplace=True)
```

In [103]:

```
print(df.isnull().sum())
print("Our dataset consists of no null values and is clean.")
```

```
State         0
District      0
Year          0
Season        0
Crop          0
Area          0
Production    0
dtype: int64
Our dataset consists of no null values and is clean.
```

In [104]:

```
#Investigating if our dataset consists of any duplicate values.
duplicate = df.duplicated()
print("The number of duplicate values are:", duplicate.sum())
```

```
The number of duplicate values are: 0
```

So far in our exploratory data analysis, it can be stated that the dataset consists of unique categorical variables which we will have to deal with as the machine learns on the language of 1s and 0s. But, firt lets see how many unique values are there of the three categorical values namely State, District, Season and Crop.

In [105]:

```
print("Number of unique States present in our dataset are:", df.State.unique().shape)
print("Number of unique Districts present in our dataset are", df.District.unique().shape)
print("Number of unique States present in our dataset are:", df.Season.unique().shape)
print("Number of unique Crops present in our dataset are: ", df.Crop.unique().shape)
```

```
Number of unique States present in our dataset are: (33,)
Number of unique Districts present in our dataset are (646,)
Number of unique States present in our dataset are: (6,)
Number of unique Crops present in our dataset are:  (124,)
```

Now, lets understand what distinct Indian districts our model will learn from.

In [106]:

```
df.District.unique()
```

Out[106]:

```
array(['NICOBARS', 'NORTH AND MIDDLE ANDAMAN', 'SOUTH ANDAMANS',
       'ANANTAPUR', 'CHITTOOR', 'EAST GODAVARI', 'GUNTUR', 'KADAPA',
       'KRISHNA', 'KURNOOL', 'PRAKASAM', 'SPSR NELLORE', 'SRIKAKULAM',
       'VISAKHAPATANAM', 'VIZIANAGARAM', 'WEST GODAVARI', 'ANJAW',
       'CHANGLANG', 'DIBANG VALLEY', 'EAST KAMENG', 'EAST SIANG',
       'KURUNG KUMEY', 'LOHIT', 'LONGDING', 'LOWER DIBANG VALLEY',
       'LOWER SUBANSIRI', 'NAMSAI', 'PAPUM PARE', 'TAWANG', 'TIRAP',
       'UPPER SIANG', 'UPPER SUBANSIRI', 'WEST KAMENG', 'WEST SIANG',
       'BAKSA', 'BARPETA', 'BONGAIGAON', 'CACHAR', 'CHIRANG', 'DARRANG',
       'DHEMAJI', 'DHUBRI', 'DIBRUGARH', 'DIMA HASAO', 'GOALPARA',
       'GOLAGHAT', 'HAILAKANDI', 'JORHAT', 'KAMRUP', 'KAMRUP METRO',
       'KARBI ANGLONG', 'KARIMGANJ', 'KOKRAJHAR', 'LAKHIMPUR', 'MARIGAON',
       'NAGAON', 'NALBARI', 'SIVASAGAR', 'SONITPUR', 'TINSUKIA',
       'UDALGURI', 'ARARIA', 'ARWAL', 'AURANGABAD', 'BANKA', 'BEGUSARAI',
       'BHAGALPUR', 'BHOJPUR', 'BUXAR', 'DARBHANGA', 'GAYA', 'GOPALGANJ',
       'JAMUI', 'JEHANABAD', 'KAIMUR (BHABUA)', 'KATIHAR', 'KHAGARIA',
       'KISHANGANJ', 'LAKHISARAI', 'MADHEPURA', 'MADHUBANI', 'MUNGER',
       'MUZAFFARPUR', 'NALANDA', 'NAWADA', 'PASHCHIM CHAMPARAN', 'PATNA',
       'PURBI CHAMPARAN', 'PURNIA', 'ROHTAS', 'SAHARSA', 'SAMASTIPUR',
       'SARAN', 'SHEIKHPURA', 'SHEOHAR', 'SITAMARHI', 'SIWAN', 'SUPAUL',
       'VAISHALI', 'CHANDIGARH', 'BALOD', 'BALODA BAZAR', 'BALRAMPUR',
       'BASTAR', 'BEMETARA', 'BIJAPUR', 'BILASPUR', 'DANTEWADA',
       'DHAMTARI', 'DURG', 'GARIYABAND', 'JANJGIR-CHAMPA', 'JASHPUR',
       'KABIRDHAM', 'KANKER', 'KONDAGAON', 'KORBA', 'KOREA', 'MAHASAMUND',
       'MUNGELI', 'NARAYANPUR', 'RAIGARH', 'RAIPUR', 'RAJNANDGAON',
       'SUKMA', 'SURAJPUR', 'SURGUJA', 'DADRA AND NAGAR HAVELI',
       'NORTH GOA', 'SOUTH GOA', 'AHMADABAD', 'AMRELI', 'ANAND',
       'BANAS KANTHA', 'BHARUCH', 'BHAVNAGAR', 'DANG', 'DOHAD',
       'GANDHINAGAR', 'JAMNAGAR', 'JUNAGADH', 'KACHCHH', 'KHEDA',
       'MAHESANA', 'NARMADA', 'NAVSARI', 'PANCH MAHALS', 'PATAN',
       'PORBANDAR', 'RAJKOT', 'SABAR KANTHA', 'SURAT', 'SURENDRANAGAR',
       'TAPI', 'VADODARA', 'VALSAD', 'AMBALA', 'BHIWANI', 'FARIDABAD',
       'FATEHABAD', 'GURGAON', 'HISAR', 'JHAJJAR', 'JIND', 'KAITHAL',
       'KARNAL', 'KURUKSHETRA', 'MAHENDRAGARH', 'MEWAT', 'PALWAL',
       'PANCHKULA', 'PANIPAT', 'REWARI', 'ROHTAK', 'SIRSA', 'SONIPAT',
       'YAMUNANAGAR', 'CHAMBA', 'HAMIRPUR', 'KANGRA', 'KINNAUR', 'KULLU',
       'LAHUL AND SPITI', 'MANDI', 'SHIMLA', 'SIRMAUR', 'SOLAN', 'UNA',
       'ANANTNAG', 'BADGAM', 'BANDIPORA', 'BARAMULLA', 'DODA',
       'GANDERBAL', 'JAMMU', 'KARGIL', 'KATHUA', 'KISHTWAR', 'KULGAM',
       'KUPWARA', 'LEH LADAKH', 'POONCH', 'PULWAMA', 'RAJAURI', 'RAMBAN',
       'REASI', 'SAMBA', 'SHOPIAN', 'SRINAGAR', 'UDHAMPUR', 'BOKARO',
       'CHATRA', 'DEOGHAR', 'DHANBAD', 'DUMKA', 'EAST SINGHBUM', 'GARHWA',
       'GIRIDIH', 'GODDA', 'GUMLA', 'HAZARIBAGH', 'JAMTARA', 'KHUNTI',
       'KODERMA', 'LATEHAR', 'LOHARDAGA', 'PAKUR', 'PALAMU', 'RAMGARH',
       'RANCHI', 'SAHEBGANJ', 'SARAIKELA KHARSAWAN', 'SIMDEGA',
       'WEST SINGHBHUM', 'BAGALKOT', 'BANGALORE RURAL', 'BELGAUM',
       'BELLARY', 'BENGALURU URBAN', 'BIDAR', 'CHAMARAJANAGAR',
       'CHIKBALLAPUR', 'CHIKMAGALUR', 'CHITRADURGA', 'DAKSHIN KANNAD',
       'DAVANGERE', 'DHARWAD', 'GADAG', 'GULBARGA', 'HASSAN', 'HAVERI',
       'KODAGU', 'KOLAR', 'KOPPAL', 'MANDYA', 'MYSORE', 'RAICHUR',
       'RAMANAGARA', 'SHIMOGA', 'TUMKUR', 'UDUPI', 'UTTAR KANNAD',
       'YADGIR', 'ALAPPUZHA', 'ERNAKULAM', 'IDUKKI', 'KANNUR',
       'KASARAGOD', 'KOLLAM', 'KOTTAYAM', 'KOZHIKODE', 'MALAPPURAM',
       'PALAKKAD', 'PATHANAMTHITTA', 'THIRUVANANTHAPURAM', 'THRISSUR',
       'WAYANAD', 'AGAR MALWA', 'ALIRAJPUR', 'ANUPPUR', 'ASHOKNAGAR',
       'BALAGHAT', 'BARWANI', 'BETUL', 'BHIND', 'BHOPAL', 'BURHANPUR',
       'CHHATARPUR', 'CHHINDWARA', 'DAMOH', 'DATIA', 'DEWAS', 'DHAR',
       'DINDORI', 'GUNA', 'GWALIOR', 'HARDA', 'HOSHANGABAD', 'INDORE',
       'JABALPUR', 'JHABUA', 'KATNI', 'KHANDWA', 'KHARGONE', 'MANDLA',
       'MANDSAUR', 'MORENA', 'NARSINGHPUR', 'NEEMUCH', 'PANNA', 'RAISEN',
       'RAJGARH', 'RATLAM', 'REWA', 'SAGAR', 'SATNA', 'SEHORE', 'SEONI',
       'SHAHDOL', 'SHAJAPUR', 'SHEOPUR', 'SHIVPURI', 'SIDHI', 'SINGRAULI',
       'TIKAMGARH', 'UJJAIN', 'UMARIA', 'VIDISHA', 'AHMEDNAGAR', 'AKOLA',
       'AMRAVATI', 'BEED', 'BHANDARA', 'BULDHANA', 'CHANDRAPUR', 'DHULE',
       'GADCHIROLI', 'GONDIA', 'HINGOLI', 'JALGAON', 'JALNA', 'KOLHAPUR',
       'LATUR', 'MUMBAI', 'NAGPUR', 'NANDED', 'NANDURBAR', 'NASHIK',
       'OSMANABAD', 'PALGHAR', 'PARBHANI', 'PUNE', 'RAIGAD', 'RATNAGIRI',
       'SANGLI', 'SATARA', 'SINDHUDURG', 'SOLAPUR', 'THANE', 'WARDHA',
       'WASHIM', 'YAVATMAL', 'BISHNUPUR', 'CHANDEL', 'CHURACHANDPUR',
       'IMPHAL EAST', 'IMPHAL WEST', 'SENAPATI', 'TAMENGLONG', 'THOUBAL',
       'UKHRUL', 'EAST GARO HILLS', 'EAST JAINTIA HILLS',
       'EAST KHASI HILLS', 'NORTH GARO HILLS', 'RI BHOI',
       'SOUTH GARO HILLS', 'SOUTH WEST GARO HILLS',
       'SOUTH WEST KHASI HILLS', 'WEST GARO HILLS', 'WEST JAINTIA HILLS',
       'WEST KHASI HILLS', 'AIZAWL', 'CHAMPHAI', 'KOLASIB', 'LAWNGTLAI',
       'LUNGLEI', 'MAMIT', 'SAIHA', 'SERCHHIP', 'DIMAPUR', 'KIPHIRE',
       'KOHIMA', 'LONGLENG', 'MOKOKCHUNG', 'MON', 'PEREN', 'PHEK',
       'TUENSANG', 'WOKHA', 'ZUNHEBOTO', 'ANUGUL', 'BALANGIR',
       'BALESHWAR', 'BARGARH', 'BHADRAK', 'BOUDH', 'CUTTACK', 'DEOGARH',
       'DHENKANAL', 'GAJAPATI', 'GANJAM', 'JAGATSINGHAPUR', 'JAJAPUR',
       'JHARSUGUDA', 'KALAHANDI', 'KANDHAMAL', 'KENDRAPARA', 'KENDUJHAR',
       'KHORDHA', 'KORAPUT', 'MALKANGIRI', 'MAYURBHANJ', 'NABARANGPUR',
       'NAYAGARH', 'NUAPADA', 'PURI', 'RAYAGADA', 'SAMBALPUR', 'SONEPUR',
       'SUNDARGARH', 'KARAIKAL', 'MAHE', 'PONDICHERRY', 'YANAM',
       'AMRITSAR', 'BARNALA', 'BATHINDA', 'FARIDKOT', 'FATEHGARH SAHIB',
       'FAZILKA', 'FIROZEPUR', 'GURDASPUR', 'HOSHIARPUR', 'JALANDHAR',
       'KAPURTHALA', 'LUDHIANA', 'MANSA', 'MOGA', 'MUKTSAR', 'NAWANSHAHR',
       'PATHANKOT', 'PATIALA', 'RUPNAGAR', 'S.A.S NAGAR', 'SANGRUR',
       'TARN TARAN', 'AJMER', 'ALWAR', 'BANSWARA', 'BARAN', 'BARMER',
       'BHARATPUR', 'BHILWARA', 'BIKANER', 'BUNDI', 'CHITTORGARH',
       'CHURU', 'DAUSA', 'DHOLPUR', 'DUNGARPUR', 'GANGANAGAR',
       'HANUMANGARH', 'JAIPUR', 'JAISALMER', 'JALORE', 'JHALAWAR',
       'JHUNJHUNU', 'JODHPUR', 'KARAULI', 'KOTA', 'NAGAUR', 'PALI',
       'PRATAPGARH', 'RAJSAMAND', 'SAWAI MADHOPUR', 'SIKAR', 'SIROHI',
       'TONK', 'UDAIPUR', 'EAST DISTRICT', 'NORTH DISTRICT',
       'SOUTH DISTRICT', 'WEST DISTRICT', 'ARIYALUR', 'COIMBATORE',
       'CUDDALORE', 'DHARMAPURI', 'DINDIGUL', 'ERODE', 'KANCHIPURAM',
       'KANNIYAKUMARI', 'KARUR', 'KRISHNAGIRI', 'MADURAI', 'NAGAPATTINAM',
       'NAMAKKAL', 'PERAMBALUR', 'PUDUKKOTTAI', 'RAMANATHAPURAM', 'SALEM',
       'SIVAGANGA', 'THANJAVUR', 'THE NILGIRIS', 'THENI', 'THIRUVALLUR',
       'THIRUVARUR', 'TIRUCHIRAPPALLI', 'TIRUNELVELI', 'TIRUPPUR',
       'TIRUVANNAMALAI', 'TUTICORIN', 'VELLORE', 'VILLUPURAM',
       'VIRUDHUNAGAR', 'ADILABAD', 'HYDERABAD', 'KARIMNAGAR', 'KHAMMAM',
       'MAHBUBNAGAR', 'MEDAK', 'NALGONDA', 'NIZAMABAD', 'RANGAREDDI',
       'WARANGAL', 'DHALAI', 'GOMATI', 'KHOWAI', 'NORTH TRIPURA',
       'SEPAHIJALA', 'SOUTH TRIPURA', 'UNAKOTI', 'WEST TRIPURA', 'AGRA',
       'ALIGARH', 'ALLAHABAD', 'AMBEDKAR NAGAR', 'AMETHI', 'AMROHA',
       'AURAIYA', 'AZAMGARH', 'BAGHPAT', 'BAHRAICH', 'BALLIA', 'BANDA',
       'BARABANKI', 'BAREILLY', 'BASTI', 'BIJNOR', 'BUDAUN',
       'BULANDSHAHR', 'CHANDAULI', 'CHITRAKOOT', 'DEORIA', 'ETAH',
       'ETAWAH', 'FAIZABAD', 'FARRUKHABAD', 'FATEHPUR', 'FIROZABAD',
       'GAUTAM BUDDHA NAGAR', 'GHAZIABAD', 'GHAZIPUR', 'GONDA',
       'GORAKHPUR', 'HAPUR', 'HARDOI', 'HATHRAS', 'JALAUN', 'JAUNPUR',
       'JHANSI', 'KANNAUJ', 'KANPUR DEHAT', 'KANPUR NAGAR', 'KASGANJ',
       'KAUSHAMBI', 'KHERI', 'KUSHI NAGAR', 'LALITPUR', 'LUCKNOW',
       'MAHARAJGANJ', 'MAHOBA', 'MAINPURI', 'MATHURA', 'MAU', 'MEERUT',
       'MIRZAPUR', 'MORADABAD', 'MUZAFFARNAGAR', 'PILIBHIT', 'RAE BARELI',
       'RAMPUR', 'SAHARANPUR', 'SAMBHAL', 'SANT KABEER NAGAR',
       'SANT RAVIDAS NAGAR', 'SHAHJAHANPUR', 'SHAMLI', 'SHRAVASTI',
       'SIDDHARTH NAGAR', 'SITAPUR', 'SONBHADRA', 'SULTANPUR', 'UNNAO',
       'VARANASI', 'ALMORA', 'BAGESHWAR', 'CHAMOLI', 'CHAMPAWAT',
       'DEHRADUN', 'HARIDWAR', 'NAINITAL', 'PAURI GARHWAL', 'PITHORAGARH',
       'RUDRA PRAYAG', 'TEHRI GARHWAL', 'UDAM SINGH NAGAR', 'UTTAR KASHI',
       '24 PARAGANAS NORTH', '24 PARAGANAS SOUTH', 'BANKURA', 'BARDHAMAN',
       'BIRBHUM', 'COOCHBEHAR', 'DARJEELING', 'DINAJPUR DAKSHIN',
       'DINAJPUR UTTAR', 'HOOGHLY', 'HOWRAH', 'JALPAIGURI', 'MALDAH',
       'MEDINIPUR EAST', 'MEDINIPUR WEST', 'MURSHIDABAD', 'NADIA',
       'PURULIA'], dtype=object)
```

Now, looking at what distinct seasons the crops are grown into. The seasons mainly include the season which are practiced in India: Kharif which is from in June and end in October, Rabi which is mostly in winter and harvested in the spring in India, Whole Year, Autumn, Summer and Winter.Hence, it can be stated that the season distribution in our dataset is fairly complete and isn't biased as it involves all the available seasons in which Indian farmers get their crops ready for production.

In [107]:

```
df.Season.unique()
```

Out[107]:

```
array(['Kharif     ', 'Whole Year ', 'Autumn     ', 'Rabi       ',
       'Summer     ', 'Winter     '], dtype=object)
```

Looking at what crops are available for training our model.

In [108]:

```
df.Crop.unique()
```

Out[108]:

```
array(['Arecanut', 'Other Kharif pulses', 'Rice', 'Banana', 'Cashewnut',
       'Coconut ', 'Dry ginger', 'Sugarcane', 'Sweet potato', 'Tapioca',
       'Black pepper', 'Dry chillies', 'other oilseeds', 'Turmeric',
       'Maize', 'Moong(Green Gram)', 'Urad', 'Arhar/Tur', 'Groundnut',
       'Sunflower', 'Bajra', 'Castor seed', 'Cotton(lint)', 'Horse-gram',
       'Jowar', 'Korra', 'Ragi', 'Tobacco', 'Gram', 'Wheat', 'Masoor',
       'Sesamum', 'Linseed', 'Safflower', 'Onion', 'other misc. pulses',
       'Samai', 'Small millets', 'Coriander', 'Potato',
       'Other  Rabi pulses', 'Soyabean', 'Beans & Mutter(Vegetable)',
       'Bhindi', 'Brinjal', 'Citrus Fruit', 'Cucumber', 'Grapes', 'Mango',
       'Orange', 'other fibres', 'Other Fresh Fruits', 'Other Vegetables',
       'Papaya', 'Pome Fruit', 'Tomato', 'Mesta', 'Cowpea(Lobia)',
       'Lemon', 'Pome Granet', 'Sapota', 'Cabbage', 'Rapeseed &Mustard',
       'Peas  (vegetable)', 'Niger seed', 'Bottle Gourd', 'Varagu',
       'Garlic', 'Ginger', 'Oilseeds total', 'Pulses total', 'Jute',
       'Peas & beans (Pulses)', 'Blackgram', 'Paddy', 'Pineapple',
       'Barley', 'Sannhamp', 'Khesari', 'Guar seed', 'Moth',
       'Other Cereals & Millets', 'Cond-spcs other', 'Turnip', 'Carrot',
       'Redish', 'Arcanut (Processed)', 'Atcanut (Raw)',
       'Cashewnut Processed', 'Cashewnut Raw', 'Cardamom', 'Rubber',
       'Bitter Gourd', 'Drum Stick', 'Jack Fruit', 'Snak Guard', 'Tea',
       'Coffee', 'Cauliflower', 'Other Citrus Fruit', 'Water Melon',
       'Total foodgrain', 'Kapas', 'Colocosia', 'Lentil', 'Bean',
       'Jobster', 'Perilla', 'Rajmash Kholar', 'Ricebean (nagadal)',
       'Ash Gourd', 'Beet Root', 'Lab-Lab', 'Ribed Guard', 'Yam',
       'Pump Kin', 'Apple', 'Peach', 'Pear', 'Plums', 'Litchi', 'Ber',
       'Other Dry Fruit', 'Jute & mesta'], dtype=object)
```

It can be seen that since the data is mainly for Indian agriculture system, the dataset includes all the crops which are regularly practiced and grown in different states and districts in different time periods in India. Nextly, understanding the distribution of Production values.

In [109]:

```
print(df.Production.min())
print(df.Production.max())
```

```
0.0
1250800000.0
```

By using min() and max() on the production column of our dataset, we can see the production of any crop ranges from yielding an absolute no production to a production of 1,250,800,000 kg/hectares which stands true to the details predicted by using the function describe() on the dataframe as well. We will be using these values for comparison later on in prediction state, so as to compare if the predicted production value exceeds the one in our current dataset.

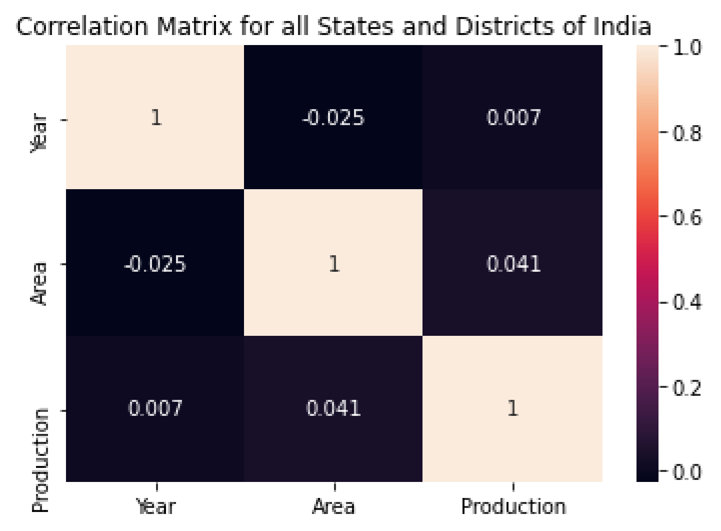
1.3 Visualizing the dataset

After having explored, what States, Districts and Crops the Indian farmers are mostly using, lets try to understand which features or independent variables play a higher factor or more specifically, lets explore the relationship between the numerical variables in our dataset with each other and use visualization tools such as graphing plots, histograms, distribution plots for better visualization of data points.

Finding Correlations between the numerical values present in our dataset

In [110]:

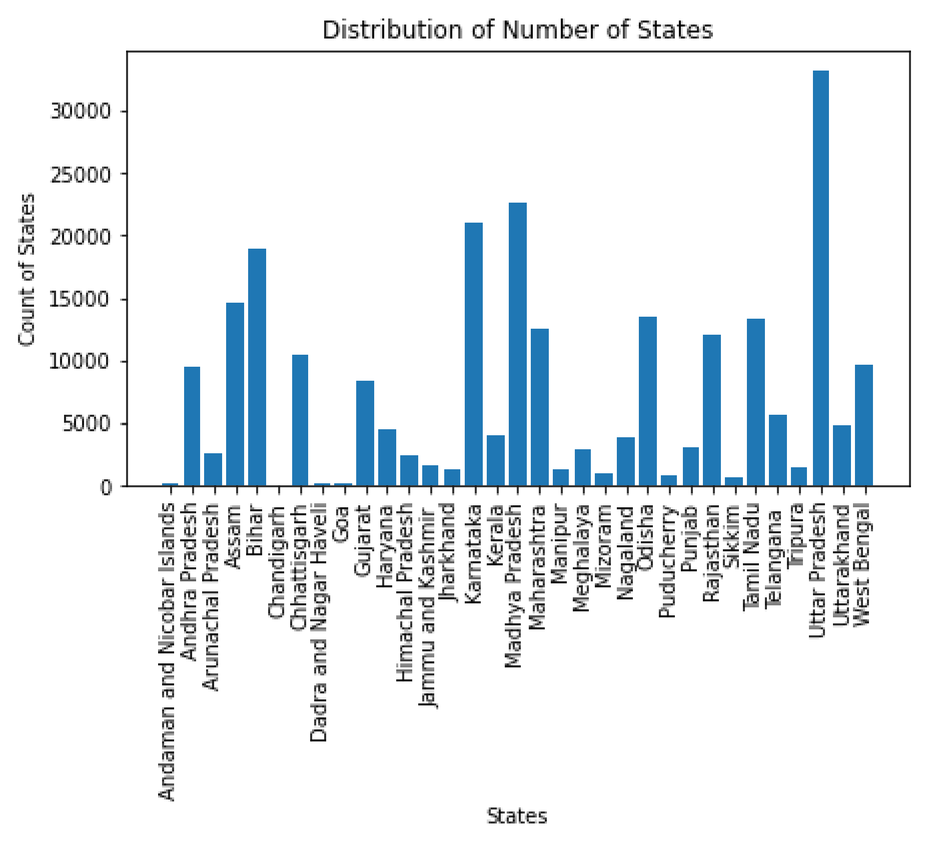
```
sns.heatmap(df.corr(), annot = True)
plt.title("Correlation Matrix for all States and Districts of India");
```



From the above correlation matrix of the numerical values present in our dataset, our target value is the Production. Therefore, comparing the correlation coefficient of production, it can be seen that the variable Area has a higher positive correlation with a magnitude of 0.041. The positive magnitude indicates that as the value of Area increases the value of Production yield would increase as well which is commonly true. Although, the value is quite low, it can be stated that the relationship between **Area and Production is weakly positively strong.**

In [111]:

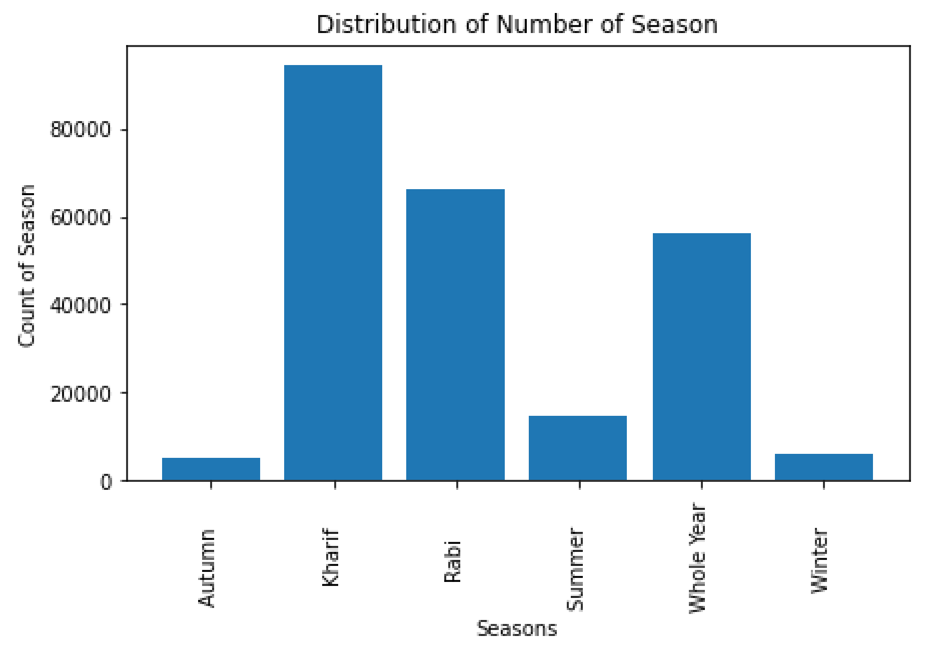
```
state_count = df.groupby(['State']).size().reset_index(name="State_Count")
fig, ax = plt.subplots(figsize=(7, 4))
plt.bar(state_count.State, state_count.State_Count)
plt.xticks(rotation=90)
plt.title("Distribution of Number of States")
plt.ylabel("Count of States")
plt.xlabel("States")
plt.show()
```



From the bar plot above, the relationship between different states and their value counts is depicted. The state present in high amount is Uttar Pradesh with an occurance of above 30,000 instances which is interestingly agreeable as Uttar Pradesh is known for it highest cultivable land. It is also ranked as the number one state for farming under state wise crop production in India for crops bajra, rice, sugarcane, food grains, and many more.

In [112]:

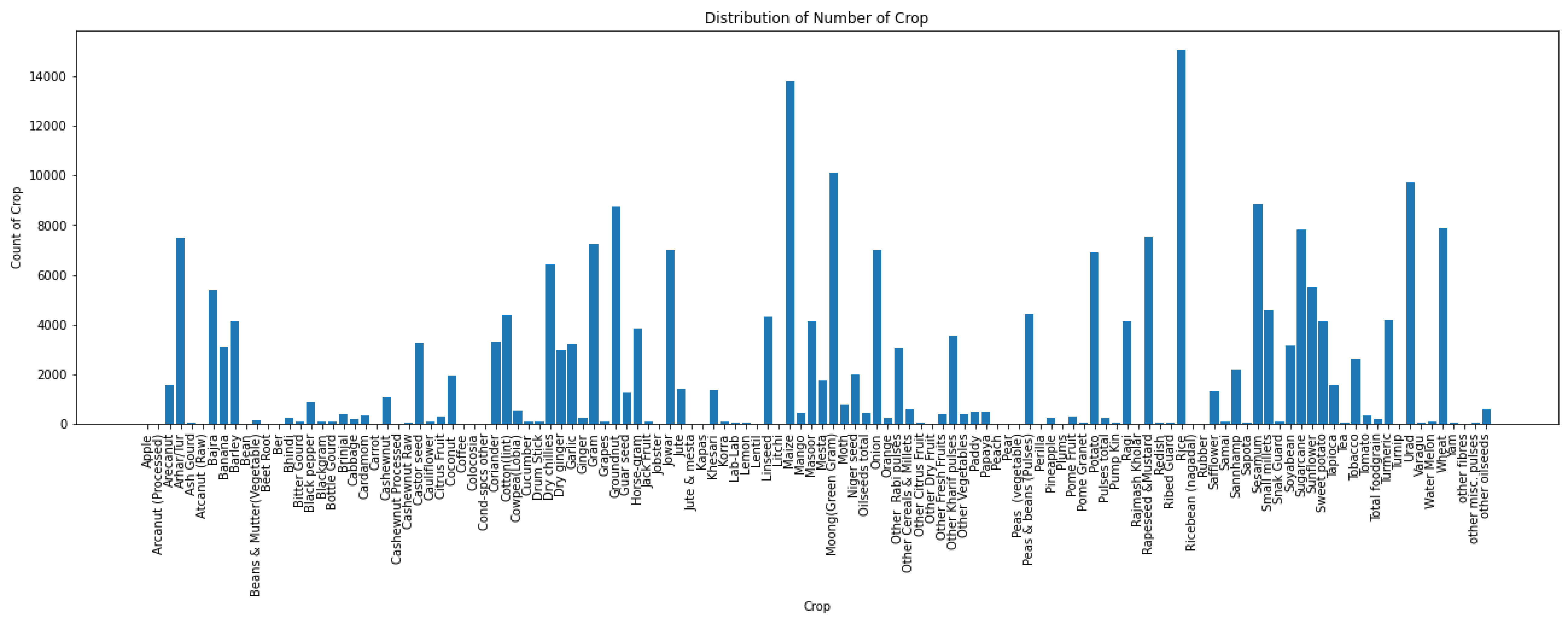
```
season_count = df.groupby(['Season']).size().reset_index(name="Season_Count")
fig, ax = plt.subplots(figsize=(7, 4))
plt.bar(season_count.Season, season_count.Season_Count)
plt.xticks(rotation=90)
plt.title("Distribution of Number of Season")
plt.ylabel("Count of Season")
plt.xlabel("Seasons")
plt.show()
```



The above plot shows the relationship between the different seasons and their value counts. It can be infered that, the season Kharif holds the maximum crop production value. Kharif is the season, which starts in June and ends in October. In addition, Kharif crops are usually sown at the beginning of the first rains during the advent of the south-west monsoon season, and they are harvested at the end of monsoon season (October-November). Some of the crops which are grown in Kharif season include Rice, soybean, groundnut and cotton. Rabi which comes on the second place with having more than 6000 instances is the season in which the crops are mainly harvested in the winter season from November to April. Some of the important rabi crops are wheat, barley, peas, gram and mustard. Furthermore, the distribution of crops is done in accordance with the Indian season system, so, the season Winter, Autumn and Summer received fairly less crop distribution as these season include explicilty just one season and the crops are explained in the Kharif, Rabi and Whole Year seasons.

In [113]:

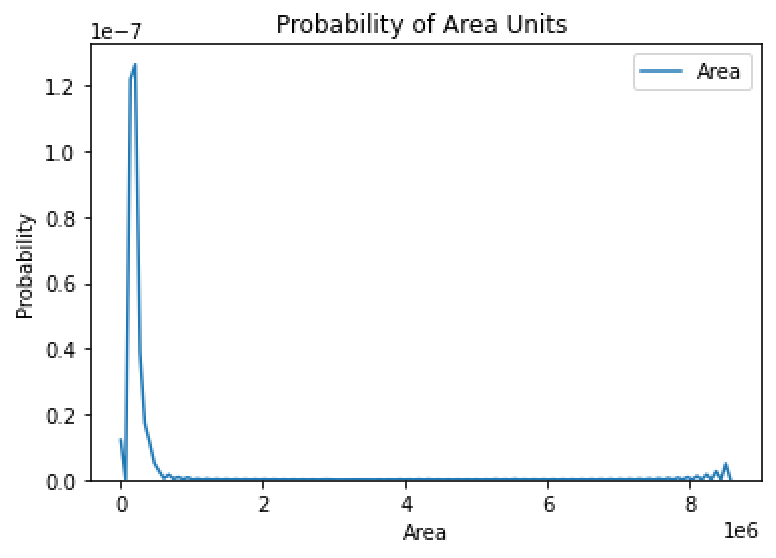
```
crop_count = df.groupby(['Crop']).size().reset_index(name="Crop_Count")
fig, ax = plt.subplots(figsize=(22, 6))
plt.bar(crop_count.Crop, crop_count.Crop_Count)
plt.xticks(rotation=90)
plt.title("Distribution of Number of Crop")
plt.ylabel("Count of Crop")
plt.xlabel("Crop")
plt.show()
```



The plot above shows the relationship between the different crops present in the dataframe with their value counts. From all the 124 unique crops present in our dataset, the crop Rice has been cultivated and yielded probably the most result with a count of over 14,000. Which is factually correct as rice is the highest produced crop in India with a production rate of 9.82 tonnes per hectare in the year 2017. With Maize crop yielding production above 12,000. Maize is the third most important food crops after rice and wheat in India. According to advance estimate its production is likely to be 22.23 M Tonnes (2012-13) mainly during Kharif season which covers 80% area. Its also worthwhile to notice that some of the crops do no yield any production such as Apple, Bean and Peach which are mainly fruits, this can be because of the reason that the districts/states that we are dealing with do no specialize in growing these fruit crops.

In [114]:

```
sns.kdeplot(df['Area'])
plt.xlabel("Area")
plt.ylabel("Probability")
plt.title("Probability of Area Units ");
```



The above plot shows the probability distribution of the different area values. It is interesting to notice here that most of the area values lie between 0.0 with the highest trend experienced in between 0 and 1 and then a gradual probability distribution of the rest remaining instances. This can be merely due to the reason that the value of area depends from farmers living situation and other economical factors. It will be interesting to see which crops yield highest production rate and how does the value of area play a role.

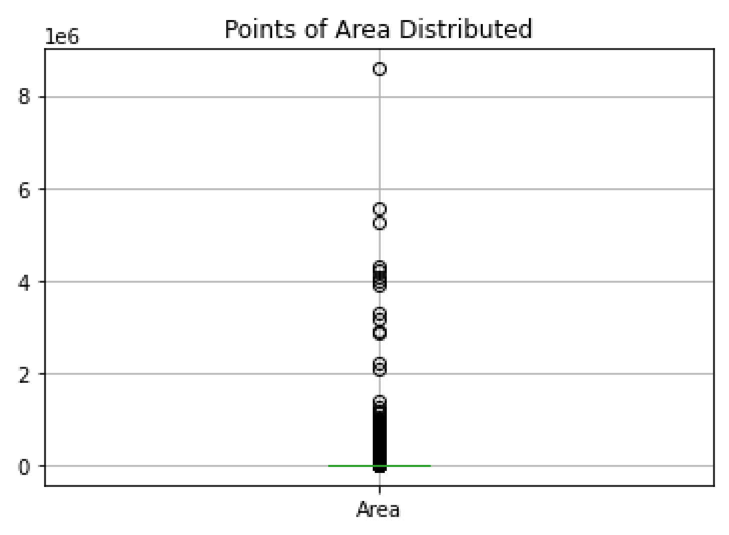
In [115]:

```
print("Maximum area occupied:",df.Area.max())
print("Minimum area occupied:",df.Area.min())
```

```
Maximum area occupied: 8580100.0
Minimum area occupied: 0.1
```

In [116]:

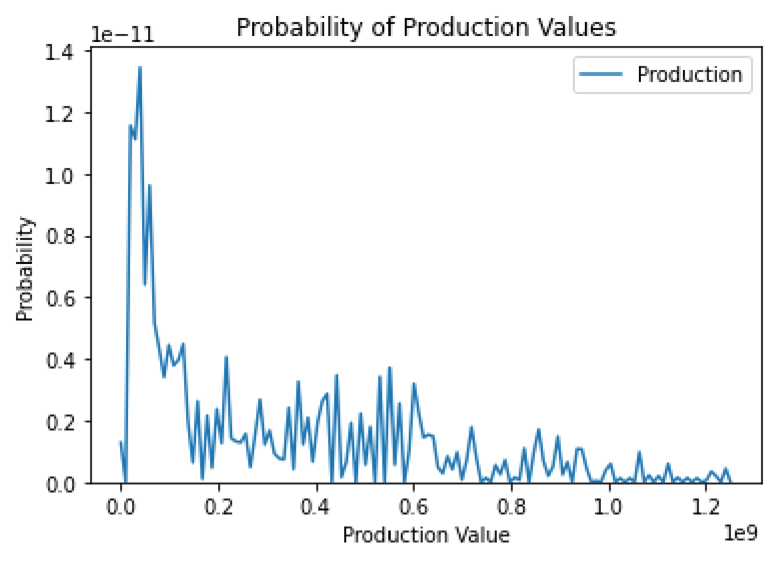
```
df.boxplot(column = ['Area'])
plt.title("Points of Area Distributed");
```



The above plot shows the boxplot for the area column and it helps in forming the distribution of data points. It can be seen that the plot contains many outliers which are depicted by the small black circles. When essentially seen the minimum and the maximum values of area it ranged from 0.1 to 8580100. The boxplot understood or categorized the instances with maximum of them lying in the range of 0 to 1 for example as we saw in the distribution plot plotted above and categorized the rest or the new area values with respect to their crop and selected season as outliers. One of the outliers that can be termed as a good outlier would be the one which is above 8 magnitude as the difference between the accumulated instances and that is fairly very high.

In [117]:

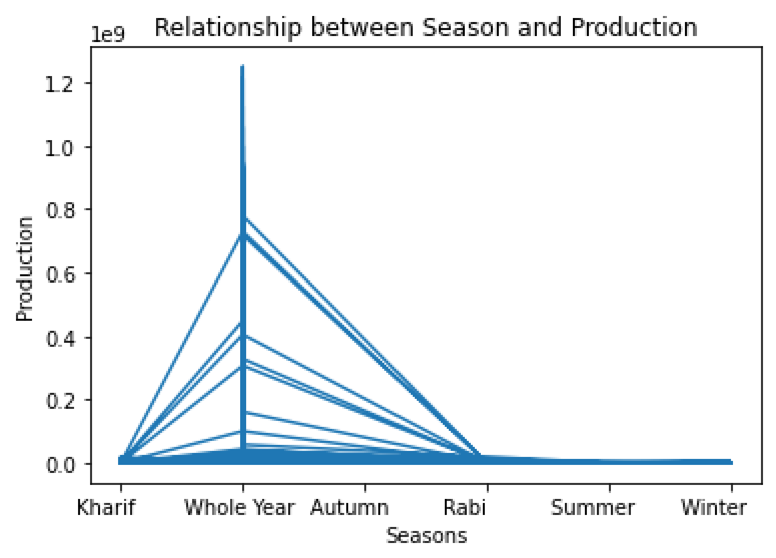
```
sns.kdeplot(df.Production)
plt.xlabel("Production Value")
plt.ylabel("Probability")
plt.title("Probability of Production Values");
```



The above graph shows how the different values of production are distributed. It's necessary to notice that most of the values rely between 0.0 and 0.2 and then seems to increase with a similar pattern until there the very end. This could mean that the production yield in different states of different crops with this specific area are deeply involved in predicting the production value directly. There seems to be a similar pattern shown in the data point distribution as that to that of area units.

In [118]:

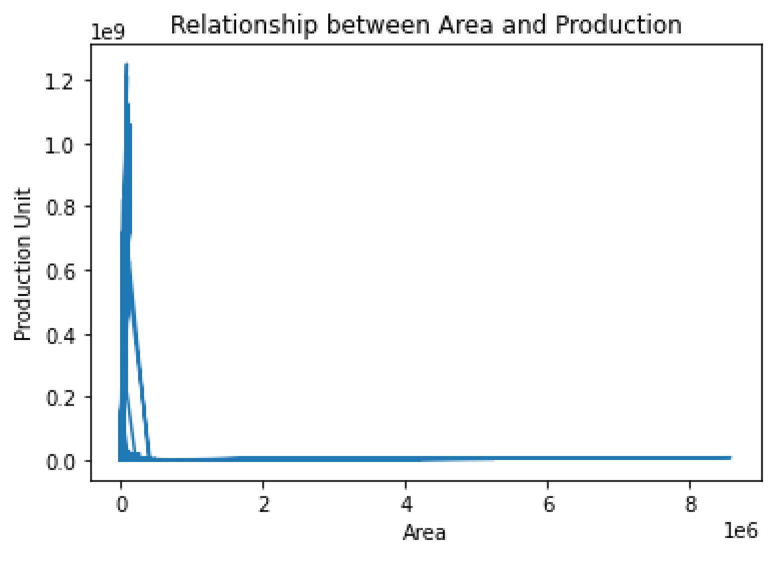
```
plt.plot(df.Season, df.Production)
plt.title("Relationship between Season and Production")
plt.xlabel("Seasons")
plt.ylabel("Production");
```



The above graph shows the relationship of how the production value varies from season to season. It's worthwhile to notice that whole year season was the season in which the production of different crops was the highest. There can also be seen a repeating pattern as the production starts to increase from a change from Kharif to whole year and then seems to decrease until the Rabi season. This trend could be because Kharif is the season which involves the crop production from the end of June whereas Rabi is from November to April which is termed as the harvesting period of crops.

In [119]:

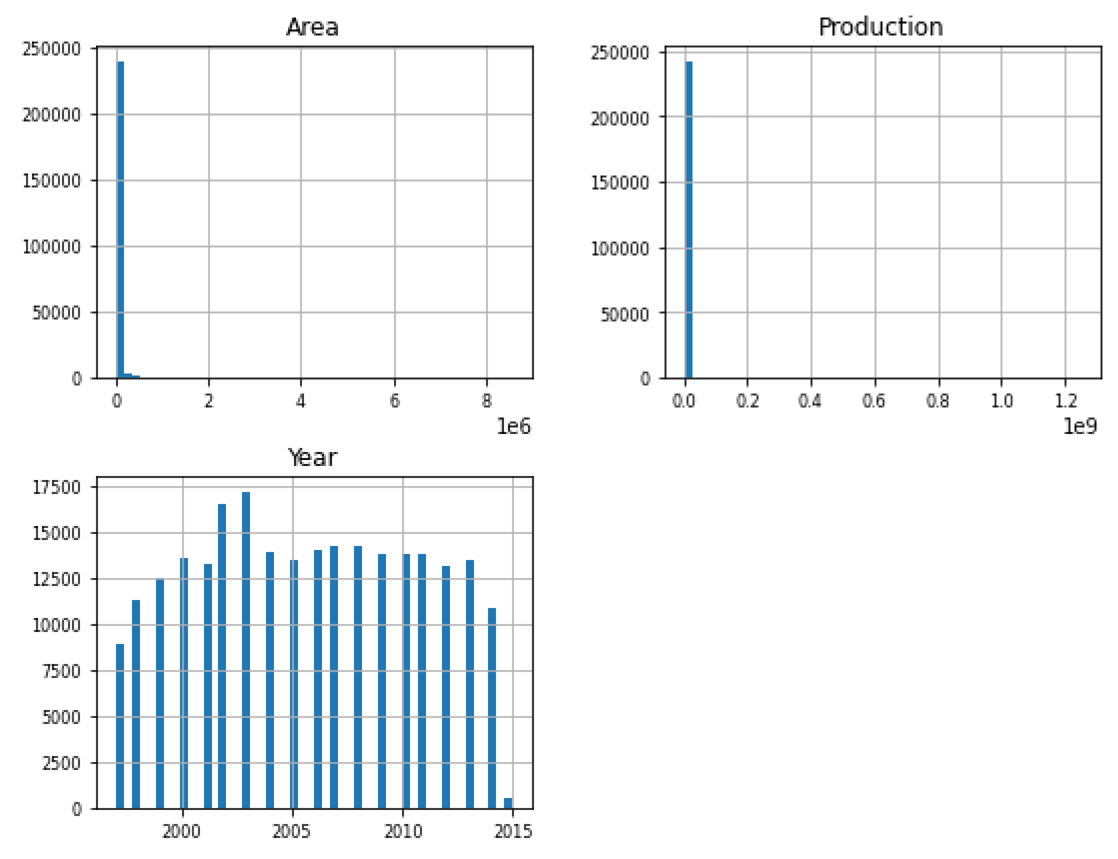
```
plt.plot('Area','Production', data = df)
plt.title("Relationship between Area and Production")
plt.xlabel("Area")
plt.ylabel("Production Unit")
plt.show();
```



In order to understand for explicilitly how indepedent variables Area and Production vary as they seem to have similar data distribution values with maximum of the values lying between 0 to 0.2/ 2, the above plot shows the relationship of how production varies by the area occupied by farmers. According to a data report it is stated that above 80% of the farmers have or own land area which is very marginal and small owning around 0-2 hectares traits. Keeping this fact in mind it can be stated true with this data that as the area ranges or experiences its highest trend in 0 and 1, the production rates are very high between them but as the area increases the production rate starts to stay gradual or linear very low. It states the fact true, that Indian farmers are forced to produce in excess when they own less amount of land.

In [120]:

```
df.hist(figsize=(9, 7), bins=50, xlabelsize=8, ylabelsize=8);
plt.show()
```

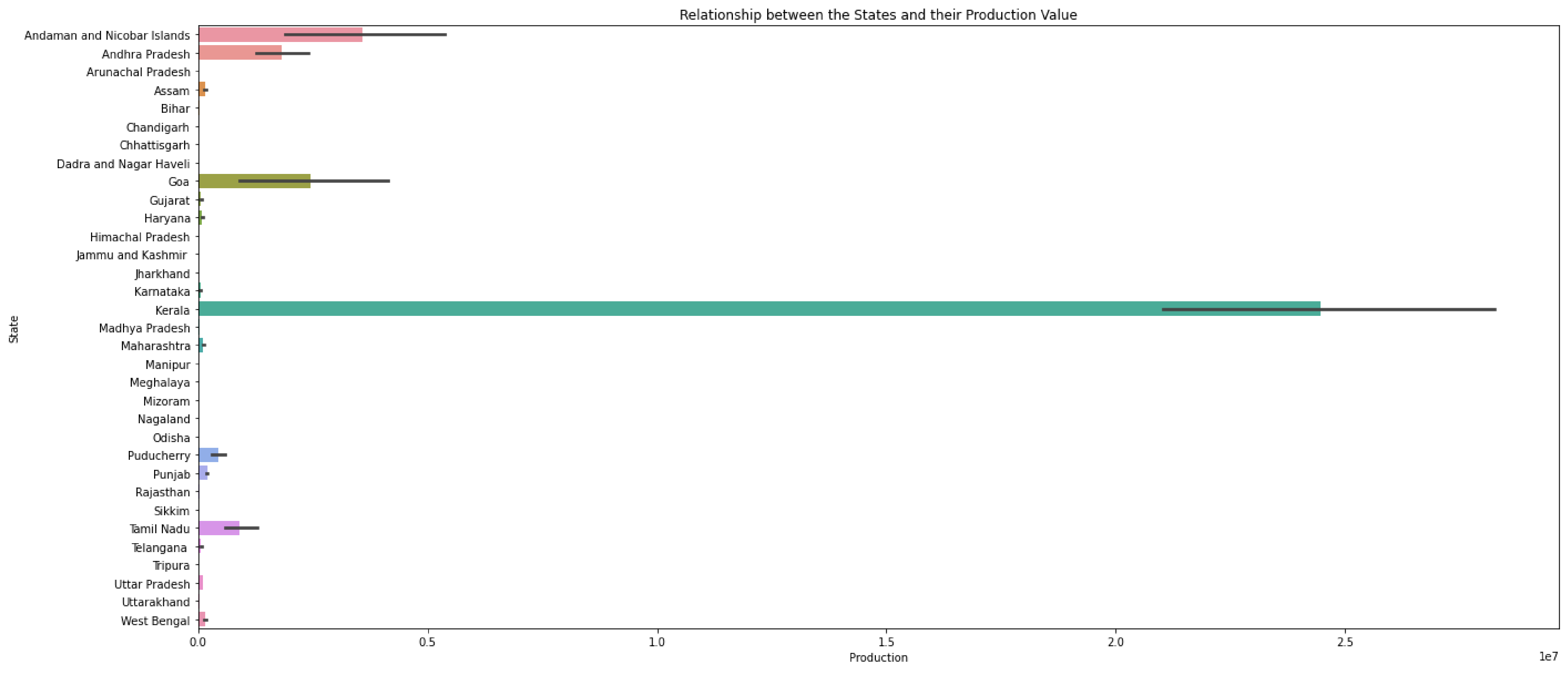


The above three graphs illustrate the distribution of numerical variables of our dataset. From our area graph of histogram it can be stated that majority of the magnitude of area lies between 0 and 2 this could be because most of the area values are in the format of 0.1 for example. Nextly, for our production histogram we can also see that majority of the production values lie between 0.0 and 0.2 as well with values reaching to more than 200000. It can also be said that most of the production was active in the year 2002 and 2003 which could be because the number of farmers protests in India started to rise around this period.

In [121]:

```
fig, ax = plt.subplots(figsize=(22, 10))
sns.barplot(df.Production, df.State, orient='h');

plt.title("Relationship between the States and their Production Value")
plt.show()
```



The above plot shows the relationship between the unique states and their production magnitude. It can be seen that Kerala is the state with the highesh production rate which can be stated because Kerala is known to be the greatest producer of small cardamom and black pepper. Additionally, as we saw above the most grown is rice, Kerala is known as the 'rice bowl of the state' because about 600 varieties of rice are grown in the sprawling paddy fields. Hence, this fact stands correct to the state and the crop they are famously known for. Its worthwhile to notice, that by conducting deep exploratory data analysis, many facts are coming in line to be proven correct by using the dataset.

2. Label Encoding

Now for the formation of our model we need to first investigate what types of variables or data or values we are working with. We see that our data set consists of *categorical variables* such as state , district , season , crop and *numerical variables* such as year , area and our target variable production. These are the categorical variables because the user would be entering these values separately to get the crop yield for each, thus these will be our categories. To make it easier for our machine learning model to read and learn from our training set, we will have to transform the categorical variables into numbers. To do this, feature engineering was done by label encoding our categorical variables into unique numbers.

In [122]:

```
#Importing LabelEncoding to preprocess and prepare the data for model training.
from sklearn.preprocessing import LabelEncoder

#Instantiating labelencoder and assigining it to variable le.
le = LabelEncoder()
```

Fit label encoder and return encoded labels.

The reason I chose to do label encoder was because initially, I decided to further transform my independent variables into one hot encoding but because that resulted into the Indian farmer having to fill many fields, 62 to be exact, it would make their jobs more or less hectec and difficult which isn't the goal of the Harvestron model. Hence, to make the process easier I did label encoding for each of the categorical variable present in our data set which ranges from 0 to the number of classes present.

In [123]:

```
#Fitting and transforming label encoded categorical variables into their number of classes
df.State = le.fit_transform(df.State)
df.Season = le.fit_transform(df.Season)
#df.Year = le.fit_transform(df.Year)
df.District = le.fit_transform(df.District)
df.Crop = le.fit_transform(df.Crop)

df
```

Out[123]:

	State	District	Year	Season	Crop	Area	Production
0	0	427	2000	1	2	1254.0	2000.0
1	0	427	2000	1	74	2.0	1.0
2	0	427	2000	1	95	102.0	321.0
3	0	427	2000	4	7	176.0	641.0
4	0	427	2000	4	22	720.0	165.0
...	...	...	...	...	...	...	...
246086	32	471	2014	3	95	306.0	801.0
246087	32	471	2014	3	102	627.0	463.0
246088	32	471	2014	4	106	324.0	16250.0
246089	32	471	2014	5	95	279151.0	597899.0
246090	32	471	2014	5	102	175.0	88.0

242361 rows × 7 columns

Finding the range of the values of our label encoded

Now that all the categorical variables have been efficiently label encoded and transformed, lets understand the range in which they vary in.

In [124]:

```
print("The Label Encoded values of States ranges from: ",range(df.State.min(), df.State.max()))
print("The Label Encoded values of District ranges from: ",range(df.District.min(), df.District.max()))
print("The Label Encoded values of Season ranges from: ",range(df.Season.min(), df.Season.max()))
print("The Label Encoded values of Crop ranges from: ",range(df.Crop.min(), df.Crop.max()))
```

```
The Label Encoded values of States ranges from:  range(0, 32)
The Label Encoded values of District ranges from:  range(0, 645)
The Label Encoded values of Season ranges from:  range(0, 5)
The Label Encoded values of Crop ranges from:  range(0, 123)
```

Printing the dictionary containing the classes and their transformed forms with their corresponding label encoded values

In [125]:

```
cropdict = dict(zip(le.classes_, le.transform(le.classes_)))
print(cropdict)
```

```
{'Apple': 0, 'Arcanut (Processed)': 1, 'Arecanut': 2, 'Arhar/Tur': 3, 'Ash Gourd': 4, 'Atcanut (Raw)': 5, 'Bajra': 6, 'Banana': 7, 'Barley': 8, 'Bean': 9, 'Beans & Mutter(Vegetab
le)': 10, 'Beet Root': 11, 'Ber': 12, 'Bhindi': 13, 'Bitter Gourd': 14, 'Black pepper': 15, 'Blackgram': 16, 'Bottle Gourd': 17, 'Brinjal': 18, 'Cabbage': 19, 'Cardamom': 20, 'Ca
rrot': 21, 'Cashewnut': 22, 'Cashewnut Processed': 23, 'Cashewnut Raw': 24, 'Castor seed': 25, 'Cauliflower': 26, 'Citrus Fruit': 27, 'Coconut ': 28, 'Coffee': 29, 'Colocosia': 3
0, 'Cond-spcs other': 31, 'Coriander': 32, 'Cotton(lint)': 33, 'Cowpea(Lobia)': 34, 'Cucumber': 35, 'Drum Stick': 36, 'Dry chillies': 37, 'Dry ginger': 38, 'Garlic': 39, 'Ginge
r': 40, 'Gram': 41, 'Grapes': 42, 'Groundnut': 43, 'Guar seed': 44, 'Horse-gram': 45, 'Jack Fruit': 46, 'Jobster': 47, 'Jowar': 48, 'Jute': 49, 'Jute & mesta': 50, 'Kapas': 51,
'Khesari': 52, 'Korra': 53, 'Lab-Lab': 54, 'Lemon': 55, 'Lentil': 56, 'Linseed': 57, 'Litchi': 58, 'Maize': 59, 'Mango': 60, 'Masoor': 61, 'Mesta': 62, 'Moong(Green Gram)': 63,
'Moth': 64, 'Niger seed': 65, 'Oilseeds total': 66, 'Onion': 67, 'Orange': 68, 'Other  Rabi pulses': 69, 'Other Cereals & Millets': 70, 'Other Citrus Fruit': 71, 'Other Dry Frui
t': 72, 'Other Fresh Fruits': 73, 'Other Kharif pulses': 74, 'Other Vegetables': 75, 'Paddy': 76, 'Papaya': 77, 'Peach': 78, 'Pear': 79, 'Peas  (vegetable)': 80, 'Peas & beans (P
ulses)': 81, 'Perilla': 82, 'Pineapple': 83, 'Plums': 84, 'Pome Fruit': 85, 'Pome Granet': 86, 'Potato': 87, 'Pulses total': 88, 'Pump Kin': 89, 'Ragi': 90, 'Rajmash Kholar': 91,
'Rapeseed &Mustard': 92, 'Redish': 93, 'Ribed Guard': 94, 'Rice': 95, 'Ricebean (nagadal)': 96, 'Rubber': 97, 'Safflower': 98, 'Samai': 99, 'Sannhamp': 100, 'Sapota': 101, 'Sesam
um': 102, 'Small millets': 103, 'Snak Guard': 104, 'Soyabean': 105, 'Sugarcane': 106, 'Sunflower': 107, 'Sweet potato': 108, 'Tapioca': 109, 'Tea': 110, 'Tobacco': 111, 'Tomato':
112, 'Total foodgrain': 113, 'Turmeric': 114, 'Turnip': 115, 'Urad': 116, 'Varagu': 117, 'Water Melon': 118, 'Wheat': 119, 'Yam': 120, 'other fibres': 121, 'other misc. pulses':
122, 'other oilseeds': 123}
```

3. Data Preprocessing

In this section, the preprocessing of the data has been done which involves normalizing the values of area column by using the min and max scaler. By normalizing our numerical values, the model learns more easily and provides a better reduces the chances of having redundant data values. Additionally, it's helpful and a good practise to normalize numerical values of the training set as it yields in more flexible understanding.

In [126]:

```
#Printing the first five values of Area.
df['Area'].head()
```

Out[126]:

```
0    1254.0
1       2.0
2     102.0
3     176.0
4     720.0
Name: Area, dtype: float64
```

In [127]:

```
#Extracting the Area columns as a float from the df
dfarea = df[['Area']].values.astype(float)

#Instantiating the minmaxscaler
min_max_scaler = preprocessing.MinMaxScaler(feature_range = (0,1))

#Fitting the area column variable to fit and transform
area_scaled = min_max_scaler.fit_transform(dfarea)

#Assigning it back to the area column of the dataset containing now the normalized values
df['Area'] = area_scaled
df['Area'].head()
```

Out[127]:

```
0    1.461405e-04
1    2.214426e-07
2    1.187632e-05
3    2.050093e-05
4    8.390345e-05
Name: Area, dtype: float64
```

Justifying the Normalization Step

In [129]:

```
print(df.Area.min())
print(df.Area.max())
```

```
0.0
1.0
```

By using the min() and max() function on the Area column of our dataset, we can confirm that the values are normalised as they all lie in the range of 0 and 1 as seen in the above chunk.

4. Model Building

For building the model which will efficiently predict the value of crop yield, the use of supervised machine learning regression algorithms will be used mainly **RandomForest Regression, AdaBoost Regressor and XGBoost Regressor**. The model which will result in higher score value of r2_score, will be chosen to ultimately predict the value of yield.

In [74]:

```
#Creating a list of accuracy and modelname to store the values of accuracy of different models.
accuracy = []
modelname = []
```

Assigning our feature (X) to all the columns except the Production column and target value (y) to Production Column by using .iloc[] to index and get the column values. It works simply by using the following terms: iloc[row_position, column_position].

In [75]:

```
X = df.iloc[:,:-1] #feature values
y = df.iloc[:,-1]  #target value (Crop Production Yield)
```

Further splitting our dataset into 70% training and 30% testing which is the ideal or fair splitting proportion. Moreover, using random state set to 42 which is the universal value, to get the same output of your test and train data points after running it again and again.

In [76]:

```
#Splitting our features and target variables into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 42, test_size = 0.3)

print("X_train shape:",X_train.shape)
print("y_train shape:{}".format(y_train.shape))
print("X_test shape:",X_test.shape)
print("y_test shape:{}".format(y_test.shape))
```

```
X_train shape: (169652, 6)
y_train shape:(169652,)
X_test shape: (72709, 6)
y_test shape:(72709,)
```

Looking at the distribution of rows which were taken randomly in our training set from the actual dataset.

In [77]:

```
#x_train shape has 70% of the data from x -30% of x_test
print(f'Shape of x_train {X_train.shape}')
X_train.head()
```

```
Shape of x_train (169652, 6)
```

Out[77]:

	State	District	Year	Season	Crop	Area
227784	30	528	2002	2	41	0.000079
73466	11	545	2007	1	43	0.000004
148845	22	76	2004	4	32	0.000042
209150	30	182	1999	1	43	0.000003
30541	4	102	2000	4	106	0.000038

Selecting Features and Scaling them

By using MinMaxScaler() specifically on the numerical variable Area, we made the values of df['Area'] in the range of (0,1). Now, for making sure that the distribution of all the training values used are distributed normally i.e forming a bell shape curve, I used StandardScaler() on the training set. The X_train and the X_test values are the training values which will be fed into each regressor, for training.

In [82]:

```
#StandardScaler is used to make the mean 0 and standard deviation of 1 for the data. Making the sd = 1, it forces the dataset to form a normal distribution (the bell shape curve)
from sklearn.preprocessing import StandardScaler
X_train = StandardScaler().fit_transform(X_train)
X_test = StandardScaler().fit_transform(X_test)
```

In [517]:

```
X_train[:3]
```

Out[517]:

```
array([[ 1.27414235,  1.15785573, -0.73355364, -0.13044347, -0.80638983,
        -0.21542333],
       [-0.6400609 ,  1.24864251,  0.27522874, -0.9082445 , -0.74766258,
        -0.22733515],
       [ 0.46816203, -1.25600452, -0.33004069,  1.4251586 , -1.07066243,
        -0.22126722]])
```

Random Forest Regressor

The first regression model, we will be using for comparison is a random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

In [518]:

```
#Importing RandomForestRegressor in our notebook
from sklearn.ensemble import RandomForestRegressor
```

In [519]:

```
#Instantiating a random forest with random state 42.
rf = RandomForestRegressor(random_state = 42)
```

Performing GridSearchCV on RandomForestRegressor to get the optimal values of hyperparameters

In [520]:

```
#Defined a grid of hyperparameters
params_rf = {'n_estimators': [50,100,120],  #The number of trees in the forest
            'min_samples_split': [1,2,3,4], #The minimum number of samples required to split an internal node
            'max_features':["auto", "sqrt", "log2"]}
```

In [521]:

```
#Then instantiating a default 5 fold CV grid search
grid_rf = GridSearchCV(estimator = rf,
                      param_grid = params_rf,
                      scoring = 'r2', #setting r2 to scoring since we're doing a regression problem
                      cv = 5,
                      n_jobs = -1) #for using the best available nodes in cpu
```

In [522]:

```
#Fit grid_dt to training set
grid_rf.fit(X_train,y_train)
```

Out[522]:

```
GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42), n_jobs=-1,
             param_grid={'max_features': ['auto', 'sqrt', 'log2'],
                         'min_samples_split': [1, 2, 3, 4],
                         'n_estimators': [50, 100, 120]},
             scoring='r2')
```

After having fit our training set on the available hyperparameter values in the gridsearchcv while setting the scoring to r2 because I want to evaluate the model's r2_score value, the best parameters are resulted.

In [523]:

```
#Extracting the best hyperparamter values
best_hyp = grid_rf.best_params_
print('Best hyperparameters:\n', best_hyp)
```

```
Best hyperparameters:
 {'max_features': 'sqrt', 'min_samples_split': 3, 'n_estimators': 120}
```

After having fit the gridsearchCV on random forest regressor, the best values of the hyperparameters are yielded.

- The max_feature is the number of features to consider when looking for the best split. Which by default is auto, but by tuning, sqrt is the best fitted criterion meaning max_features=sqrt(n_features). The min_samples_split is resulted to be 3, meaning the minimum number of samples required to split an internal node of the forest/tree is 3.
- Additionally, the n_estimators hyperparameter is the one which takes in the number of trees to be built in the forest which is 120. Moreover, the best accuracy score from the 5 folds formed in the cross validation, the highest score is of 94% by using the best*score* function mentioned in the next chunk.
- Here, because we have used cv=5 which is the default number of folds required to conduct cross validation, i.e. assigning an integer value to the cv in the GridSearchCV, therefore the *StratifiedKFold* cross validation method has been used to calculate the best cvscore. StratifiedKfold is a variation of k-fold which returns stratified folds meaning, each set contains approximately the same percentage of samples of each target class as the complete set.

In [524]:

```
#Extracting the best CV score
best_cv_score = grid_rf.best_score_
print('Best CV accuracy',best_cv_score)
```

```
Best CV accuracy 0.9418632578997613
```

In [525]:

```
#Extracting the best estimators test set accuracy
best_model = grid_rf.best_estimator_
best_model.score(X_test,y_test)
```

Out[525]:

```
0.79080705124374
```

The 79% score by using .score indicates the approximate value of the r2_score or the coefficient of randomness in the actual Random Forest Regressor model which can be achieved.

Instantiating RandomForest Regressor *rfr* with the best hyperparameter values taken from the GridSearchCV results

After having gotten the optimal values of hyperparameters from the GridSearchCV, forming and instantiating a new randomforest regressor with the optimal values. In addition, seeting the criterion as mse, because we are conducting a regression analysis and use mse for evaluation which is also the default value.

In [526]:

```
#Instantiating a new random forest regressor and assigning it to rfr
rfr = RandomForestRegressor(max_features = 'sqrt',min_samples_split = 3,n_estimators = 120, random_state = 42,criterion='mse')
```

In [527]:

```
#Appending the model name to modelname list
modelname.append("RandomForestRegressor")
```

In [528]:

```
#Fitting the rfr to the training set
rfr.fit(X_train,y_train)
```

Out[528]:

```
RandomForestRegressor(max_features='sqrt', min_samples_split=3,
                      n_estimators=120, random_state=42)
```

In [529]:

```
#Creating y_pred for carrying out predictions
y_pred = rfr.predict(X_test)
```

Again calculating the MSE for rfr. Interstingly, the use of MAE which stands for *Mean Absolute Error* is used to evaluate as it calculates the absolute average distance between the real data and the predicted data, but it fails to punish large errors in prediction.

When an integer is passed to the cv parameter of cross_val_score():

StratifiedKFold is used if the estimator is a classifier and y is either binary or multiclass. In all other cases, KFold is used.

Cross Validation

When training the model en tuning its hyperparameters, overfitting on the training data is possible. Therefore, to overcome this, we will be performing KFold cross validation to compute the the accuracy scores and the average value.

In [530]:

```
from sklearn.model_selection import cross_val_score, KFold
```

In [531]:

```
print("The shape of our feature (X) is", X.shape)
```

```
The shape of our feature (X) is (242361, 6)
```

In [532]:

```
#Importing the KFold cross validation, where K is the number of splits the training data will be splitted into.
from sklearn.model_selection import KFold
kfold=KFold(n_splits=5, shuffle=True,random_state=42)

i = 1
for train_index, test_index in kfold.split(X):
    print("Fold: ", i)
    print(f'Train Index: has {len(train_index)} elements')
    print(f'Test Index: has {len(test_index)} elements')
    print("---------------------------------")
    i +=1
```

```
Fold:  1
Train Index: has 193888 elements
Test Index: has 48473 elements
---------------------------------
Fold:  2
Train Index: has 193889 elements
Test Index: has 48472 elements
---------------------------------
Fold:  3
Train Index: has 193889 elements
Test Index: has 48472 elements
---------------------------------
Fold:  4
Train Index: has 193889 elements
Test Index: has 48472 elements
---------------------------------
Fold:  5
Train Index: has 193889 elements
Test Index: has 48472 elements
---------------------------------
```

- We can see that the KFold tried to split the data into 5 folds equally. Initially, our dataset, the X feature consits of a total of 242,361 elements which is an odd number. Therefore, we can see the first split contains 193,888 elements for train index and 48,473 elements for test index. But in the following folds 2 and up until 5, all the training and test elements are split equally.
- For evaluation, 242361/5 = 48.472,2 (testing elements) and the remaining 48.472,2 * 4 = 193,888 (testing elements).

In [533]:

```
cvs = cross_val_score(rfr,X_train, y_train,cv=kfold)
print(f'The accuracy score of the 5 folds are: {cvs}')
print("The mean cross validations score", format(cvs.mean()))
```

```
The accuracy score of the 5 folds are: [0.97126577 0.95396997 0.83785117 0.92248506 0.95853081]
The mean cross validations score 0.9288205566926715
```

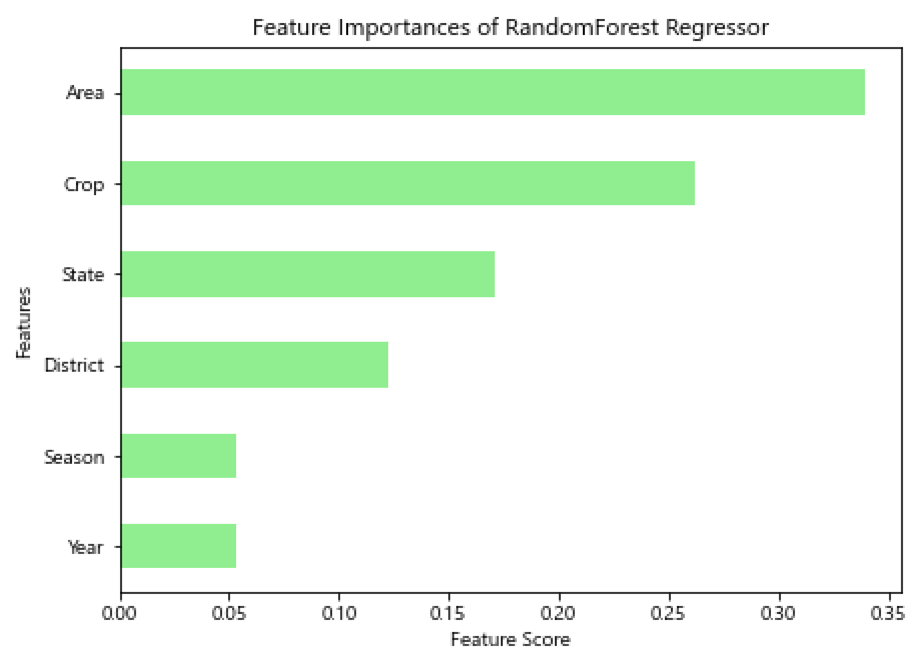
- Using KFold and cross_validation we have got a mean accuracy of ~ 93.6%
- From the accuracy score of the 5 folds, it can seen that the third fold with an accuracy of 86% was the fold which performed slight lower i.e., the chances of our data being missclassified in this fold is high when compared to the rest.

In [534]:

```
#create a pd.series of feature importance
importances_rfr = pd.Series(rfr.feature_importances_, index = X.columns)
#sorting values in descending order.
sorted_importances_rfr = importances_dt.sort_values()
```

In [535]:

```
#Plotting graph of feature importance
fig, ax = plt.subplots(figsize=(7, 5))
sorted_importances_rfr.plot(kind = 'barh', color = 'lightgreen')
plt.title("Feature Importances of RandomForest Regressor")
plt.ylabel("Features")
plt.xlabel("Feature Score")
plt.show()
```



The above plot shows which feature played a higher role in determining or predicting the value of crop yield by the random forest regressor model. As seen, the feature area received the highest score with about 35%, meaning this model defined the crop yield with highly focusing on the area, which was owned, or which was used by the farmer to crop his or her crops. This is quite interestingly true because in reality also the area for the land on which the crop has to be harvested is an important key feature. Hence, we could say our random forest regressor learned from the training set quite efficiently.

The Mean Absolute Error (MAE) is the measurement of the absolute average distance between the real data and the predicted data, but it fails to punish large errors in prediction. While, the Mean Square Error (MSE) is the measurement of the average distance between the real data and the predicted data. To initially evaluate the model, these two are evaluted. With getting a MSE value of 71002168683683, that high magnitude could mean the model has learnt to predict the training data and unseen data points perfectly.

In [536]:

```
print('The MSE of rfr is:', MSE(y_pred, y_test))
print('The MAE', mae(y_pred, y_test))
```

```
The MSE of rfr is: 71963986325069.48
The MAE 268343.42121975875
```

Since, MSE and MAE aren't the appropriate evaluation metrics, therefore r2_score which is also known as the Coefficient of determination is used to evaluate the performance of our random forest regression model. It is the amount of the variation in the output dependent attribute which is predictable from the input independent variable(s). Achieving an accuracy score of 75% shows the model has learnt from the training set provided quite well and the score value of 79% achieved from the GridSearchCV resulted out to be somewhere nearby correct.

In [537]:

```
#For higher dependency, calculating the r2_score of rfr
rfscore = r2_score(y_pred,y_test)
print("The r2_score of the rfr model is", rfscore)
```

```
The r2_score of the rfr model is 0.7528597988540843
```

In [538]:

```
#Storing the score of rfr in accuracy list
accuracy.append(rfscore)
```

AdaBoost Regressor

AdaBoost Regressor or Adaptive Boosting uses an ensemble technique of boosting, which is a method involving combining various weak learners to generate high results. A weak learner is a model that is essentially very simple and learns the dataset easily. It involves using one level decision trees (also known as stumps) as weak learners that are sequentially added to the ensemble. Each subsequent model learns and corrects the predictions/results made by the previous model which is handled by assigning the learning rate and weights.

In [539]:

```
#Importing AdaBoost Regressor into the notebook
from sklearn.ensemble import AdaBoostRegressor
```

In [540]:

```
#Importing RandomizedSearchCV into the work environment for hyperparameter tuning.
from sklearn.model_selection import RandomizedSearchCV
```

In [541]:

```
#Instantiating an adaboost regressor with a random state of 0 and assigning it to abr
abr = AdaBoostRegressor(random_state=0)
```

Performing RandomizedSearchCV on AdaBoostRegressor to get the optimal values of hyperparameters

Here, to get the best hyperparamter values, we try a different method i.e implementing RandomizedSearchCV which uses a fit and a score method. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings. In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter which will be assigned as 10 to the default value. One interesting thing about this method is if all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used.

In [542]:

```
params = {'n_estimators': [50, 100],
          'learning_rate' : [0.01,0.05,0.1,0.3,1],
          'loss' : ['linear', 'square', 'exponential']}
```

In [543]:

```
#Listing and assigning the parameters with the specified one to the abr estimator with the default 5 cross validation folds.
adacv = RandomizedSearchCV(estimator = abr,
                          param_distributions=params,
                          cv = 5,
                          n_jobs = -1)
```

In [544]:

```
#Fitting each parameter of gridsearch cv on the training set
adacv.fit(X_train,y_train)
```

Out[544]:

```
RandomizedSearchCV(cv=5, estimator=AdaBoostRegressor(random_state=0), n_jobs=-1,
                   param_distributions={'learning_rate': [0.01, 0.05, 0.1, 0.3,
                                                          1],
                                        'loss': ['linear', 'square',
                                                 'exponential'],
                                        'n_estimators': [50, 100]})
```

In [545]:

```
#Extracting the best hyperparamter values
best_hyp = adacv.best_params_
print('Best hyperparameters:\n', best_hyp)
```

```
Best hyperparameters:
 {'n_estimators': 50, 'loss': 'square', 'learning_rate': 0.3}
```

After performing RandomisedGridSearchCV, we get the correct values of the hyperparameters. Explicitly stating, to use 50 estimators i.e. the maximum number of estimators at which boosting technique is terminated. In case of perfect fit, the learning procedure is stopped early. The loss assigned as exponential, meaning the loss function to use when updating the weights after each boosting iteration. And the learning rate to 0.3 which is how the weights are assigned and changed in every boosting iterations. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learning_rate and n_estimators parameters.

Cross Validation Results of AdaBoost

From the cross validation technique of k-fold used in 5 folds, the best cv fold has an accuracy score of 74% meaning the rest of the folds contain the chances of missclassification highly.

In [546]:

```
#Extracting the best CV score
best_cv_score = adacv.best_score_
print('Best CV accuracy',best_cv_score)
```

```
Best CV accuracy 0.7421241669463928
```

In [547]:

```
#Extracting the best estimators test set accuracy
best_model = adacv.best_estimator_
best_model.score(X_test,y_test)
```

Out[547]:

```
0.6335041809710229
```

With having 63% as the best score or the best magnitude of r2_score value for the AdaBoost Regressor indicates that the probability of being able to perform on unseen data if used specifically this model would be quite low.

Instantiating AdaBoost Regressor *adr* with the best hyperparameter values taken from the RandomizedSearchCV results

After having received the optimal values from RandomizedSearchCV, putting the values and assigning it to adr regressor.

In [548]:

```
#Instantianting adr regressor with the hyperparameter values gotten.
adr = AdaBoostRegressor(n_estimators= 100, loss= 'exponential', learning_rate =0.01)
```

In [549]:

```
#Adding the name to the list
modelname.append("AdaBoostRegressor")
```

In [550]:

```
#Fitting the randomizedsearchcv parameters to the training set
adr.fit(X_train,y_train)
```

Out[550]:

```
AdaBoostRegressor(learning_rate=0.01, loss='exponential', n_estimators=100)
```

In [551]:

```
#Performing Kfold cross validation explicitly to understand the behaviour of the data instances in the model.
cvs = cross_val_score(adr,X_train, y_train,cv=KFold(n_splits=5,shuffle=True,random_state = 42))
print(f'The accuracy score of the 5 folds are: {cvs}')
print("The mean cross validations score", format(cvs.mean()))
```

```
The accuracy score of the 5 folds are: [0.58437561 0.66519078 0.48986213 0.69109991 0.5828609 ]
The mean cross validations score 0.6026778663395169
```

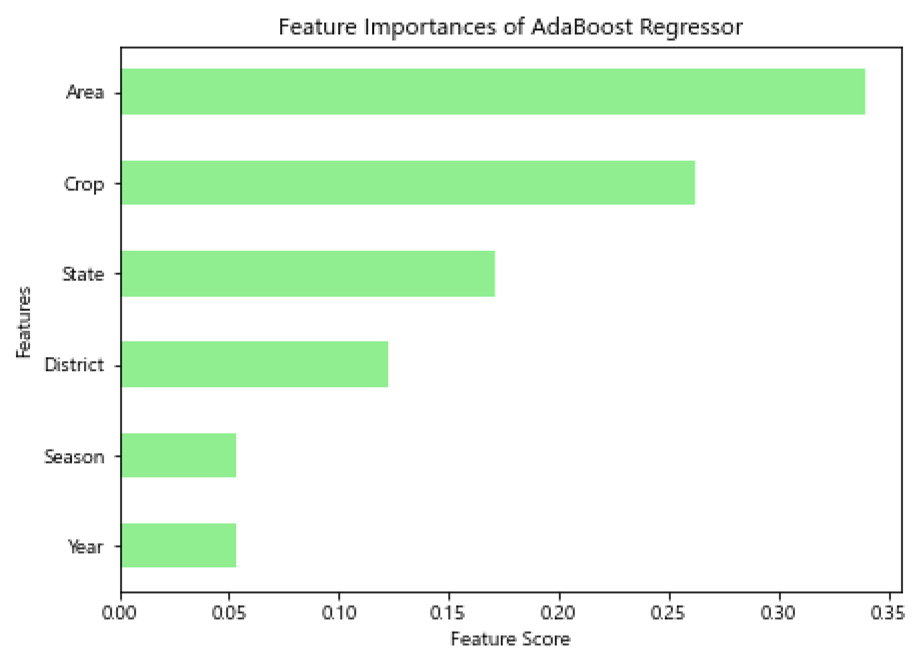
- The mean cross validation score is 60% with the highest accuracy received on 4th fold with 69%.
- The fold which holds missclassification error most likely is 3rd fold with the lowest accuracy of just 48%.

In [552]:

```
#create a pd.series of feature importance
importances_afr = pd.Series(adr.feature_importances_, index = X.columns)
#sorting values in descending order.
sorted_importances_adr = importances_dt.sort_values()
```

In [553]:

```
#Plotting graph of feature importance
fig, ax = plt.subplots(figsize=(7, 5))
sorted_importances_adr.plot(kind = 'barh', color = 'lightgreen')
plt.title("Feature Importances of AdaBoost Regressor")
plt.ylabel("Features")
plt.xlabel("Feature Score")
plt.show()
```



It is quite interesting to notice here, that 'Area' remains the most important feature for AdaBoost Regressor also with a feature score reaching to 35%. Which can be agreed upon since, the area or the land on which the crop is to be harvested does play a key factor. And it alligns correctly to the results from the correlation matrix as well.

In [554]:

```
#Storing the predictions made by AdaBoost into y_pred variable
y_pred = adr.predict(X_test)
```

In [555]:

```
#Getting the score of the model
adscore = r2_score(y_pred,y_test)
print("The r2_score of the adr model is:", adscore)
```

```
The r2_score of the adr model is: 0.27604735875943764
```

So, the AdaBoost regressor performed slightly lower with achieving an accuracy of 27%, thought the feature importances very quite similar to those of the RandomForestRegressor the adaboost regressor was able to learn and can be improved for future implementations if adaboost regressor was the one and only model which was being taken for prediction but for now we will be storing it's accuracy into accuracy list and compare it with XGBoost Regressor.

In [556]:

```
#Adding to the score list
accuracy.append(adscore)
```

XGBoost Regressor

The last model used for prediction of the crop yield is the Extreme Gradient Boosting or XGBoost. It is widely known for outperforming while solving regression predictive problems. It is an implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm, which attempts to accurately predict a target variable by combining the estimates of a set of simpler, weaker models.

When using gradient boosting for regression, the weak learners are regression trees, and each regression tree maps an input data point to one of its leafs that contains a continuous score. XGBoost minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). The training proceeds iteratively, adding new trees that predict the residuals or errors of prior trees that are then combined with previous trees to make the final prediction. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

In [557]:

```
#Importing xgboost
import xgboost as xgb
from xgboost import XGBRegressor as xgbr
print('xgboost version:', xgb.__version__)
```

```
xgboost version: 1.4.1
```

First, we will be converting the dataset into an optimized data structure called Dmatrix that XGBoost supports and gives it acclaimed performance and efficiency gains. We will be using the variable **data_dmatrix** for plotting and understanding more deeply how the xgboost regression model learns.

In [558]:

```
#Creating a matrix of our data with x and y
data_dmatrix = xgb.DMatrix(data=X,label=y)
```

Instantiating a xgbr model with the object 'reg:linear', as the aim of Harvestron is to solve a predictive regression model.

In [559]:

```
xgreg = xgbr(objective ='reg:linear')
```

Performing RandomizedSearchCV on XGBoost to get the optimal values of hyperparameters

For yielding the optimal values of the hyperparamter variables in xgboost, we will be using another hyperparameter technique which is RandomizedSearchCV.

In [560]:

```
#Importing RandomizedSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

The main hyperparameters tuned are the learning rate (eta) which is the rate by which the weak learners shall learn to combine. The available values for ets are 0.01, 0.1 and 0.2 because lower learning rate is often associated with higher performance. The alpha which is the L1 regularization on leaf weights, larger the value, more would be the regularization which causes many leaf weights in the base learner to go to 0. For this, the typical values range between 0.5-0.9, therefore for our xgboost the values 0 and 1 have been assigned.

n_estimators is the number of trees which will be formed to help the model learn, for this the available values chosen are 100, 200, 300, 400 whereas 100 is the lowest typical value with 500 being the maximum. In addition to the construction of trees, the max_depth has also been assigned values 3,4,5,7,9 which is the number which tells how many intermediate nodes the tree will have during boosting round, the maximum depth's default is 3, therfore to experiment if more depth can be a better fit, numbers higher than 3 are added.

The colsamples_by tree is the hyperparameter which takes in the total percentage of features which will be used for boosting and making the model learn. The default value is 1 while the value ranges between 0.5–1. In order to experiment if lower values could also possibily give high result, therefore values 0.3,0.4, and 0.6 are used.

In [561]:

```
parameters = {'eta': [0.01,0,1,0.2],   #learning rate
              'n_estimators' : [100,200,300,400], #number of trees to be used
              'max_depth'    : [3,4,5,7,9], #determines how deeply each tree is allowed to grow during any boosting round
              'gamma': [0,1], #controls whether a given node will split based on the expected reduction in loss after the split. A higher value leads to fewer splits. Supported only for tree-based learners.
             'alpha': [10,20], #L1 regularization on leaf weights. A large value leads to more regularization
             'colsample_bytree': [0.3,0.4,0.6]} #percentage of features used per tree. High value can lead to overfitting
```

In [562]:

```
randm_xgb = RandomizedSearchCV(estimator=xgreg,
                               param_distributions = parameters,
                               cv = 5,
                               n_jobs=-1)
```

In [563]:

```
import time
```

In [564]:

```
start = time.time()
randm_xgb.fit(X_train,y_train)

print("Time taken: ", time.time() - start, "seconds")
```

```
[12:39:20] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/objective/regression_obj.cu:171: reg:linear is now deprecated in favor of reg:squarederror.
Time taken:  95.27251172065735 seconds
```

In [565]:

```
#Extracting the best hyperparamter values
best_hyp = randm_xgb.best_params_
print('Best hyperparameters:\n', best_hyp)
```

```
Best hyperparameters:
 {'n_estimators': 100, 'max_depth': 7, 'gamma': 1, 'eta': 0.2, 'colsample_bytree': 0.6, 'alpha': 20}
```

In [566]:

```
#Extracting the best CV score
best_cv_score = randm_xgb.best_score_
print('Best CV accuracy',best_cv_score)
```

```
Best CV accuracy 0.9109108679772188
```

In [567]:

```
#Extracting the best estimators test set accuracy
best_model =  randm_xgb.best_estimator_
best_model.score(X_test,y_test)
```

Out[567]:

```
0.7174320800647171
```

With a score of 71%, the XGBoost regressor model has managed to learn better and will be able to predict values of crop yield with higher approximation.

Instantiating XGBoost Regressor *xgbr* with the best hyperparameter values taken from the GridSearchCV results

In [568]:

```
xg = xgbr(n_estimators = 100, max_depth = 5, gamma = 0, eta = 1, colsample_bytree = 0.6, alpha = 10)
```

In [569]:

```
modelname.append("XGBoostRegressor")
```

In [570]:

```
#Fitting the xgb model on the training set
xg.fit(X_train,y_train)
```

Out[570]:

```
XGBRegressor(alpha=10, base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.6, eta=1, gamma=0,
             gpu_id=-1, importance_type='gain', interaction_constraints='',
             learning_rate=1, max_delta_step=0, max_depth=5, min_child_weight=1,
             missing=nan, monotone_constraints='()', n_estimators=100,
             n_jobs=12, num_parallel_tree=1, random_state=0, reg_alpha=10,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
```

In [571]:

```
#Assigning the predicted values of crop yield in y_pred
y_pred = xg.predict(X_test)
```

Achieving an root mean squared error of ~11838987 for the regression loss, means that that data points are being read and learnt from the model.

In [572]:

```
rmse = np.sqrt(MSE(y_test, y_pred))
print("The rmse of xgbr regression model is", rmse)
```

```
The rmse of xgbr regression model is 11838987.089097256
```

With achieving a r2_score of 58%, it can indicate the XGboost regression model is 58% close to the datapoints to the actual fitted regression/decision stumps line. The reason for such lower accuracy rate could be because, the hyperparameter tuning which we have used is the RandomizedSearchCV which will only yield the perfect value for the specified hyperparameter, while the rest will be stayed at their default value.

In [573]:

```
xgscore = r2_score(y_pred,y_test)
xgscore
```

Out[573]:

```
0.5867392362033323
```

In [574]:

```
#appending r2_score
accuracy.append(xgscore)
```

k-fold Cross Validation using XGBoost

In order to further understand the robust model, we will be performing k-fold cross validation method by using the cv() method with using the original training dataset for both training as well as validation where each entry is used for testing just once.

For conducting the cross validation, we need to specify some paramters within the cv() method. Initially, here the data_dmatrix variable that was created in the beginning is used, this is because xgboost needs to be transformed into a matrix, then specifying the number of folds which is 3, so we will be having 3 cross validation sets because I wanted to investigate if lesser number of folds could improve in higer performance or not.

Nextly, the num_boost_round is set to 100 as it denotes the number of trees you want to build which corresponds to our n_estimator hyperparameter for the final xgb. The early_stop_rounds is the parameter which finishes training of the model early if the hold-out metric which in our case is rmse stays same for 10 rounds i.e. shows no improvement. Lastly, setting the as_pandas parameter as true to return the results in a pandas dataframe and seeting seed to 123 for reproducibility.

In [575]:

```
#Creating a dictionary of hyperparameters used for the final xgb regressor and storing them in params.
params = {"objective": "reg:linear", "colsample_bytree": 0.6, "alpha":10,"eta":1,"max_depth":5}
cv_results = xgb.cv(dtrain=data_dmatrix,
                    params = params,
                    nfold=3,
                    num_boost_round=100,
                    early_stopping_rounds=10,
                    metrics="rmse", as_pandas=True, seed=123)
cv_results.head()
```

```
[12:39:27] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/objective/regression_obj.cu:171: reg:linear is now deprecated in favor of reg:squarederror.
[12:39:27] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/objective/regression_obj.cu:171: reg:linear is now deprecated in favor of reg:squarederror.
[12:39:27] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/objective/regression_obj.cu:171: reg:linear is now deprecated in favor of reg:squarederror.
```

Out[575]:

	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	9.076848e+06	1.310175e+05	9.747262e+06	4.281366e+05
1	8.553676e+06	2.544965e+05	9.421005e+06	4.597711e+05
2	7.791084e+06	1.033599e+06	8.691166e+06	1.116636e+06
3	7.516286e+06	9.918987e+05	8.581019e+06	1.081032e+06
4	7.220408e+06	8.203377e+05	8.427501e+06	1.220449e+06

Hence, from the above chunk we can see that cv_results contains the train and test rmse values for each boosting round. One thing to notice here, is that the early_stopping_rounds parameter was active as we only see 4 rounds.

In [576]:

```
print((cv_results["test-rmse-mean"]).tail(1))
```

```
75    6735777.0
Name: test-rmse-mean, dtype: float64
```

It can be seen that the rmse value for test set is reduced to 6,735,777 from 11,838,987. when compared to the rmse value of the final xbg model. This implies that by using cross validation techniques, the performance of powerful machine learning models such as xgboost can be improved gradually.

Visualising Boosting Trees and Feature Importance

Nextly, after having evaluted the performance of our xgboost regressor, we would like to visualise and understand how the model has actually understood and came to giving results. For this, the use of data_dmatric will be followed with the same parameters as that specified above. xgb.train is an advanced interface for training an xgboost model. The num_boost_rounds is set to a lower value of 10 to fit all the 10 trees since contructing trees of 100 wouldn't be easy to visualize.

In [577]:

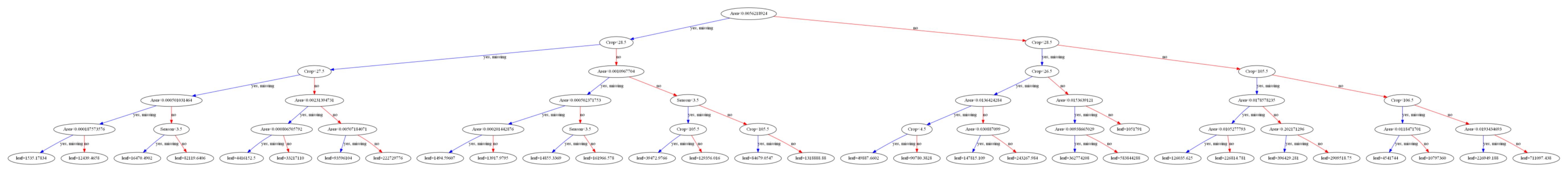
```
#Setting xg_reg as the variable which will be training our data_dmatrix
xg_reg = xgb.train(params=params, dtrain=data_dmatrix, num_boost_round=10)
```

```
[12:39:35] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/objective/regression_obj.cu:171: reg:linear is now deprecated in favor of reg:squarederror.
```

For plotting, xgb library provides a .plot_tree() function which gives us the plot. Here, the num_trees parameter is specifing the ordinal number of target tree is set to 0 which is the default value.

In [606]:

```
xgb.plot_tree(xg_reg,num_trees=0)
plt.rcParams['figure.figsize'] = [60, 40] #rcParams["figure.figsize"] plotting the plot
plt.show()
```



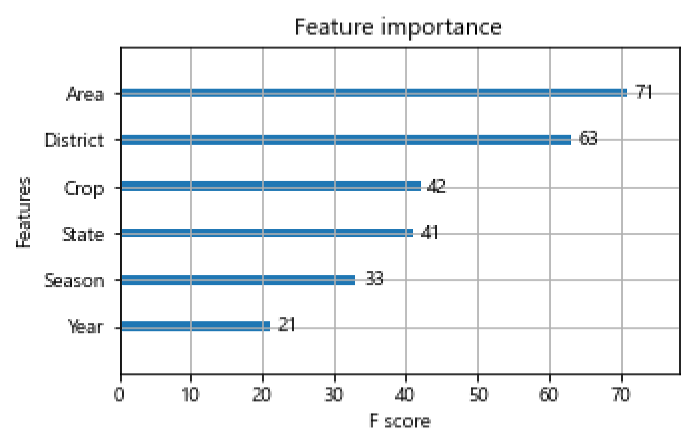
The above plot show us how the xgboost regressor reads, learns and ultimately comes to it's final decisions and how it's been splitted for different trees.

Understanding which feature got the most importance

For doing this, we use the .plot_importance() function on the xg_reg defined above used to train the matrixed version of data_dmatrix.

In [604]:

```
xgb.plot_importance(xg_reg)
plt.rcParams['figure.figsize'] = [3, 5]
plt.show()
```



From the above plot, it can be seen that the feature *Area* has been given the highest importance score amongst all the features with an feature score of above 70. In addition, the District feature received higher importance than Crop which was seen in the above two models. Accordingly, Area, District and Crop can be concluded as the top three important features to be taken into consideration when predicting crop yield magnitude.

Model Evaluation

Now having instantiated and evaluated the different models, we now compare the accuracies of the different models by comparing their r2_score by using the lists modelname and accuracy which were created above.

In [580]:

```
modelname
```

Out[580]:

```
['RandomForestRegressor', 'AdaBoostRegressor', 'XGBoostRegressor']
```

In [581]:

```
accuracy
```

Out[581]:

```
[0.7528597988540843, 0.27604735875943764, 0.5867392362033323]
```

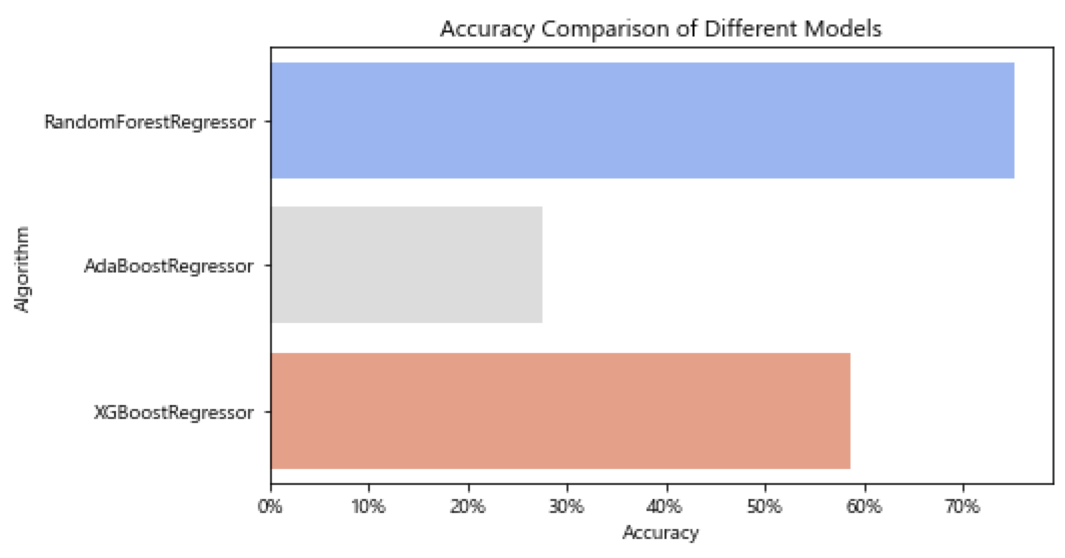
In [582]:

```
import matplotlib.ticker as mtick
```

In [583]:

```
#Plotting the graph of differerent models
fig, ax = plt.subplots(figsize=(7, 4))
sns.barplot(x = accuracy,y = modelname,palette='coolwarm')

ax.xaxis.set_major_formatter(mtick.PercentFormatter(1.0))
plt.title('Accuracy Comparison of Different Models')
plt.xlabel('Accuracy')
plt.ylabel('Algorithm')
matplotlib.rcParams['font.family'] = "segoe ui"
```



From the graph above, it can be seen that models RandomForest achieved 75% accuracy which indicated how powerful it is in learning from categorical and numerical values. While the ensemble method algorithms XGBoost Regressor performed upto 58% and AdaBoost Regressor with only 28%. By using hyperparameter tuning GridSearchCV, the performance of these models has resulted into The lower value for XGBoost can be due to the reason of implementing RandomizedSearchCV hyperparameter tuning since it only tunes and focuses on the ones explicity specified. This also shows and results into declaring GridSearchCV as the optimal solution for getting the best values for your hyperparameters.

Therefore, RandomForestRegresor model would be the correct choice to pick for predicting the final crop yield values.

Weather Prediction Notification

For building a weather prediction notification which changes from day to day and predicts the approximate values of *temperature*,*humidity*,*dewpoint*, and *visibility* overall for India, web scrapping has been used.

In [584]:

```
#Importing request module from urllib for fetching URLs.
from urllib.request import urlopen, Request
```

For conducting web scrapping efficiently, the *BeautifulSoup* tool has been used. It helps in grabbing and returning string stripped of textual data on any HTML tags and metadata. In addition, it helps in retriveing specific information from webpages easily.

In [585]:

```
from bs4 import BeautifulSoup as bs
```

Nextly, for enabling notifications, showing notification on the windows environment will be used to illustrate or replicate the notification prototype. For this, the use of *win10toast* library is used which provides ToastNotifier class which helps in displaying notification message.

In [586]:

```
from win10toast import ToastNotifier
```

Assigning the header as the application, or the user agent which will be used to open the url to Chrome. Inserting the url from which will be scraping in Request() function and setting it to open and assigning it to variable html.

In [587]:

```
header = {"User-Agent":"Chrome"}
req = Request("https://www.timeanddate.com/weather/india/new-delhi", headers = header)
html = urlopen(req)
```

Checking the status of the response. With a status equal to 200, corresponds to a success status response and that the request has been passed.

In [588]:

```
#Successful response
html.status
```

Out[588]:

```
200
```

Setting the url with the request to variable obj with using the web scraping tool.

In [589]:

```
obj = bs(html)
```

Now, for fetching the exact data values, by manually going to the data value that needs to be displayed in the notification on the website and fetching it's html elements. Finding the div tag with the class h2 which contains the value of temperature. In addition, splitting the text to extract just the temperature value.

In [590]:

```
#Extracting the temperature of India (New Delhi) as using it as the average temperature of India
temp = obj.find("div", {"class":"h2"}).text.split()[0]
degree = obj.find("div", {"class":"h2"}).text.split()[1]

temp
degree
```

Out[590]:

```
'°C'
```

In [591]:

```
obj.find("div", {"class":"bk-focus__info"})
```

Out[591]:

```
<div class="bk-focus__info"><table class="table table--left table--inner-borders-rows"><tbody><tr><th>Location: </th><td>New Delhi / Safdarjung</td></tr><tr><th>Current Time: </t
h><td id="wtct">8 mei 2021 16:09:40</td></tr><tr><th>Latest Report: </th><td>8 mei 2021 14:30</td></tr><tr><th>Visibility: </th><td>4 km</td></tr><tr><th>Pressure: </th><td>1005
mbar</td></tr><tr><th>Humidity: </th><td>24%</td></tr><tr><th>Dew Point: </th><td>14 °C</td></tr></tbody></table></div>
```

Next, since the tr has been used multiple times as seen in the chunk below, therefore, creating a list of all the tr tags and using the .next_siblings on it to fetch the value of visibility by again splitting it.

In [592]:

```
visibility = list(obj.find("div", {"class":"bk-focus__info"}).tr.next_siblings)[2].text.split()[1]
pressure = list(obj.find("div", {"class":"bk-focus__info"}).tr.next_siblings)[3].text.split()[1]
humidity = list(obj.find("div", {"class":"bk-focus__info"}).tr.next_siblings)[4].text.split()[1]
dewpoint = list(obj.find("div", {"class":"bk-focus__info"}).tr.next_siblings)[5].text.split()[2]

print("Visibility:",visibility)
print("Pressure:",pressure)
print("Humidity:",humidity)
print("DewPoint:",dewpoint)
```

```
Visibility: 4
Pressure: 1005
Humidity: 24%
DewPoint: 14
```

Instantiating the ToastNotifier() function and setting it to variable notifier.

In [593]:

```
notifier = ToastNotifier()
```

Forming the message that will be displayed as the notification on the screen

In [594]:

```
message = "Current Temp: " + temp + degree + "\nHumidity: " + humidity + "\nDewPoint: " + dewpoint+'°C'+ "\nVisibility: "+visibility + "km"
message
```

Out[594]:

```
'Current Temp: 38°C\nHumidity: 24%\nDewPoint: 14°C\nVisibility: 4km'
```

Lastly, using the show_toast() function and assiging the relevant title, the message with the icon to be showed in the message for a duration of 5 seconds on screen.

In [595]:

```
notifier.show_toast(title = "WeatherUpdate", msg = message, duration = 5, icon_path = r"wt-9.ico")
```

Out[595]:

```
True
```

Predictions

Now finally, after evaluated the performance of all regression models, we will be going forward with the RandomForestRegressor. For predictions, we will be doing it in two ways. First, is generating a random sample size from our X i.e., train set and predicting it's values and secondly, predicting on unseen data points by manually entering the values of each field.

Forming predictions from a random sample taken from the feature variable/set X by setting the random state to 42 and assigning it to X_new variable

In [596]:

```
X_new = X.sample(5, random_state = 42)
X_new
```

Out[596]:

	State	District	Year	Season	Crop	Area
51163	6	272	2007	2	119	0.000637
55141	6	476	2012	2	92	0.000097
104122	16	65	2012	4	87	0.000001
161230	23	643	2010	2	63	0.000002
229232	30	549	2002	2	59	0.000014

The sample above includes randomized 5 row sample taken from the feature variable X.

- **The number of unique states available are 6 = Chattisgarh, 16 = Madhya Pradesh, 23 = Puducherry and 30 = Uttar Pradesh.**
- **Their corresponding districts are 272 = PALAKKAD, 476 = JHALAWAR, 65 = BEGUSARAI, 643 = MEDINIPUR WEST, 549 = AURAIYA.**
- **The seasons in which crops are harvested are 2 = Autumn, 4 = Summer**
- **The crops are 119 = Plums , 92 = Bitter Gourd, 87= Atcanut (Raw), 63 = Peas (vegetable), 59 = Lemon.**

The interesting thing to notice here is that since the categorical values have been label encoded individually, so when applying reverse labeling on each of the categorical variable, it returns the reverse labeled form of the 'Crop' value without actually mapping them to the correct value as seen below. The understanding from this result, results out to when applying label encoding to multiple columns in python making sure to reduce the number of categorical values for better readability, this can be added towards the future implementation section. Since, currently, the values extracted have been written by manually slicing the indexed value from the initial dataframe df.

In [597]:

```python
#Applying reverse labeling to the columns in the X_new which is a random set of data points taken from X.
inverState = le.inverse_transform(X_new.State)  #Inversing State Name
inverSeason = le.inverse_transform(X_new.Season) #Inversing Season
inverCrop = le.inverse_transform(X_new.Crop)     #Inversing Crop harvested

print(inverState)
print(inverSeason)
print(inverCrop)
```

```
['Bajra' 'Bajra' 'Blackgram' 'Cashewnut Processed' 'Colocosia']
['Arecanut' 'Arecanut' 'Ash Gourd' 'Arecanut' 'Arecanut']
['Wheat' 'Rapeseed &Mustard' 'Potato' 'Moong(Green Gram)' 'Maize']
```

In [598]:

```python
#Predicting crop yield magnitude
rfr.predict(X_new)
```

Out[598]:

```
array([162797.09884325, 162797.09884325, 162797.09884325, 162797.09884325,
       162797.09884325])
```

Hence, by using the rfr regression model on random samples taken from the dataset, the predicted crop yield has been seen. From the results, it can be seen that the state 6 which is 'Chattisgrah' in district 272 = Palakkad in the year 2007, the crop 119 = Plumns harvested in season 2 = Autumn with area 0.000637 in the normalized form, produced the highest production yield of 4951.61 kg/hectares. One key insights or pattern to notice here is that, while the State Chattisgrah planted different crops 119 = Plumns and 92 = Bitter Gourd in the same season 2 = Autumn, when compared it's yield in district 476 = Jhalawar in the year 2012, the crop yield has reduced to 690 kg/hectares. This indicates that since the Mainpat district of Chattisgrah which is pretty close to the Palakkad district is famously known for producing rich fruits such as Plums, Peach, Pear etc.

Forming predictions by manually entering values of each input/independent variable and getting the magnitude of crop yield

In [599]:

```python
state = input("Enter State:")
district = input("Enter District:")
Year = input("Year:")
Season = input("Season:")
Crop = input("Crop: ")
Area = int(input("Area"))

yield_result = rfr.predict([[state,district,Year,Season,Crop,Area]])

print("The crop yield for the specified input values is: ", yield_result, "kg/hac.")
```

```
The crop yield for the specified input values is:  [856731.14123016] kg/hac.
```

Hence, for West Bengal (33) State, PURULIA (646) District, in year 2021, crop 3 (rice), Season 1 (Whole Year) with area 143, the crop yield is 856731.141 kg/hac.

Conclusions

By using RandomForestRegressor supervised machine learning algorithm with an accuracy of ~75%, it can be well stated that Harvestron was successfully able to predict the magnitude of crop yield for different states in their districts. With getting *Area* as the most important feature for all the three evaluated model, it can be agreed that the results from the confusion matrix showing Area having the most positive and strong relationship with out target variable "Crop Production" was true. Moreover, evaluating models which follow boosting techniques such as AdaBoost and XGBoost was a great practice for understanding their boosting and "combining weak learners together to produce high results" principles. For future implementations, the main area for improvement would definetely be to enter the actual categorical variable name instead of the label encoded number. The dataset used was agreeably good for predicted crop yield values as while exploring the dataset, the facts which were stated by wikipedia were coming out to be true so, as we were able to determine the accuracy of correctness of a fact online by using this dataset, I strongly believe this dataset was a perfect fit for predicting the crop yield in different districts of India and hopefully contributes someway in providing ease and comfort in thier day to day lifestyle activities.