Abstract Summary and Problem Statement

Problem Statement :Predict the age of abalone from physical measurements by applying K -Nearest Neighbour Algorithm from scratch. Evaluate the performance of the algorithm which has been implemented.

The aim is to predict the age of abalone from physical measurements. The age of abalone is traditionally determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope — a boring and time-consuming task. Other measurements, which are easier to obtain, are used to predict the age.

We are using KNN as the classifier here and would be implementing it from scratch. We did Data sanitization, Exploratory Data Analysis, Data Scaling, Categorical feature encoding and finally implemented the KNN algorithm with train and test and K Fold cross validation.

We have selected Euclidian distance as a method to gauge the similarity between any 2 data points in the Abalone Data set.

After doing a trial and error whereby we tried out different values of K and different folds and training data sets, we were able to recieve the best accuracy and least error rate at K = 20 with Training cut taken at 60% and Test cut taken at 40% or Folder value at 10 with 100% Training data

Extending values of K beyond K gives better results especially at k around 80 - 90. Please find the below the charts which ellicit this finding

From a run time perspective, there is a pretty big difference between K-fold and Train and Test, This is expected as there would be recurring distance calculations happening on K fold cross validation process

We have discussed the classification report and have given some possible improvements which I could do the KNN implementation to make it more effective

Data Set Information:

Predicting the age of abalone from physical measurements. The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope -- a boring and time-consuming task.

Other measurements, which are easier to obtain, are used to predict the age. Further information, such as weather patterns and location (hence food availability) may be required to solve the problem.

Attribute Information:

Given is the attribute name, attribute type, the measurement unit and a brief description. The number of rings is the value to predict: either as a continuous value or as a classification problem.

Name / Data Type / Measurement Unit / Description

Sex / nominal / -- / M, F, and I (infant)
Length / continuous / mm / Longest shell measurement
Diameter / continuous / mm / perpendicular to length
Height / continuous / mm / with meat in shell
Whole weight / continuous / grams / whole abalone
Shucked weight / continuous / grams / weight of meat
Viscera weight / continuous / grams / gut weight (after bleeding)
Shell weight / continuous / grams / after being dried
Rings / integer / -- / +1.5 gives the age in years

Do note that the Rings as an output has to be added with 1.5 to get the age of Abalone

The dataset contains 4177 observations, 8 descriptives features and 1 target feature.

Target Feature
The target feature is the rings of abalone. We need to add add 1.5 to the number of rings to give the age of the Abalone.

Introduction and Approach to solve the problem

We would be using K-Nearest Neighbour as a classifer to predict the target feature. In this report, target is the 'Rings' of abalone which ranges from 1 to 29.

This problem can be solved by both using KNN as a classifier as well as a regressor but in the scope of this report we would proceed with using it as a Classifier.

We would be using the below steps during the entire process

```
1. Data Preparation (Sanitization)
2. Exploratory Data Analysis (Using Pandas, Matplotlib and Seaborn)
3. Correlation Analysis to understand the intricasies of the data set
4. Encoding the categorical features to numerical outputs
5. Scaling all the features to bring the features to the same grain.
6. Distance Selection to be used in our KNN
7. Apply Train and Test at different training data set sizes
8. Apply K fold Cross validation on the overall Data Set
9. Apply K Fold Cross validation on 80% Data set and use the remaining 20% data se
t to measure the performance.
10. Performance comparison
```

Data Preparation

In [7]:

```python
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

In [233]:

```python
# Importing the dataset
data = pd.read_csv('abalone.data',header=None,names=["Sex", "Length", "Diameter", "Hei
data.head()
```

Out[233]:

	Sex	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

In [250]:
```
# Check the count of all the Rings record to check if the output is biased. Doesn't lo
values = (data['Rings'].value_counts())
values.head(100)
```

Out[250]:
```
9     689
10    634
8     568
11    487
7     391
12    267
6     259
13    203
14    126
5     115
15    103
16     67
17     58
4      57
18     42
19     32
20     26
3      15
21     14
23      9
22      6
24      2
27      2
1       1
25      1
2       1
26      1
29      1
Name: Rings, dtype: int64
```

The data looks to be not an unbalanced data set atleast from the output variable perspective so, we can be sure that we could take accuracy as an important metric when trying to gauge the performance of the data set.

Rings to be replaced with Age by adding 1.5 to Rings

From the website (https://archive.ics.uci.edu/ml/datasets/Abalone (https://archive.ics.uci.edu/ml/datasets/Abalone)) Age has been defined as Age = 1.5+Rings so we need to add a new field as Age and calculate it accordingly

In [22]:
```
data['Age'] = data['Rings']+1.5
data.drop('Rings', axis = 1, inplace = True)
data.head()
```

Out[22]:

	Sex	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Age
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	16.5
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	8.5
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	10.5
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	11.5
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	8.5

Display the shape of the dataset and the data types

In [23]:
```python
# Display the shape of the dataset
print(data.shape)
# Display the Datatype of each attribute
data.dtypes
```

```
(4177, 9)
```

Out[23]:
```
Sex                object
Length            float64
Diameter          float64
Height            float64
Whole_weight      float64
Shucked_weight    float64
Viscera_weight    float64
Shell_weight      float64
Age               float64
dtype: object
```

Data Sanitization

Checking whether we have null or empty data in our dataset

In [25]:
```python
print(data.isna().sum())
```

```
Sex               0
Length            0
Diameter          0
Height            0
Whole_weight      0
Shucked_weight    0
Viscera_weight    0
Shell_weight      0
Age               0
dtype: int64
```

We dont have NAN or empty cells in our dataset so no need to apply handle Empty Cell or empty values

Describe the Dataset

In [26]:

```
data.describe()
```

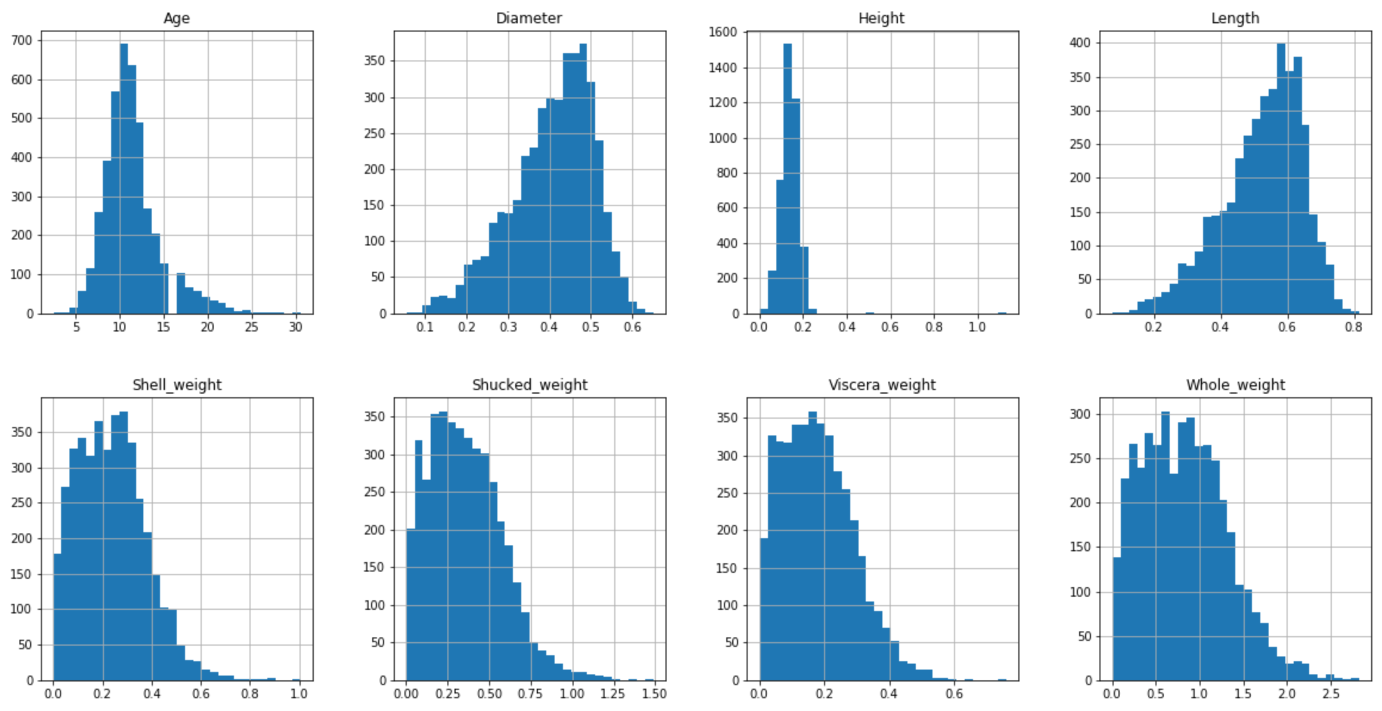
Out[26]:

	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	41
mean	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.238831	
std	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.139203	
min	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.001500	
25%	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.130000	
50%	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.234000	
75%	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.329000	
max	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.005000	

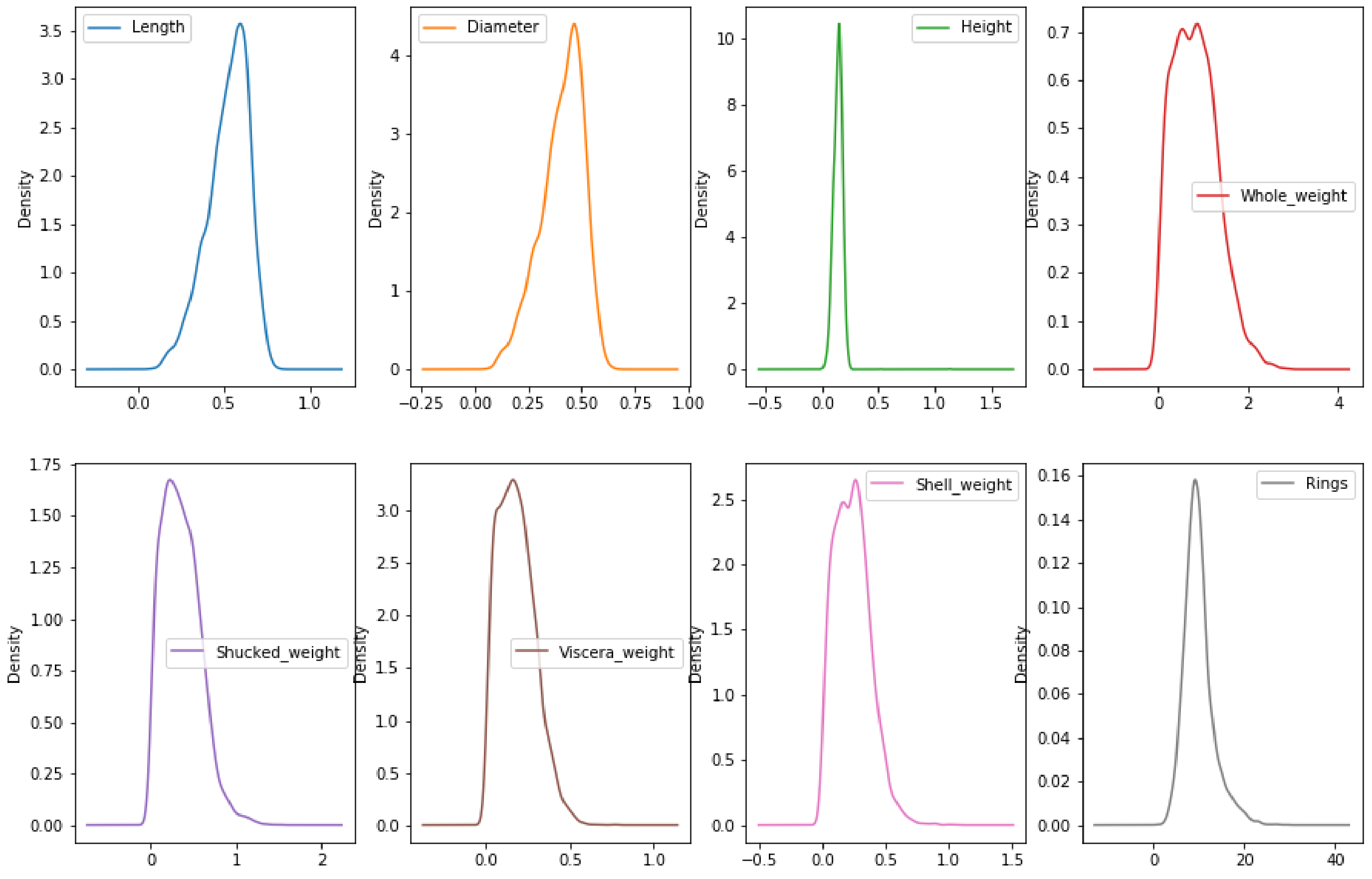
Histogram and Density Distribution of all features to check out the distribution which the data set is following

In [31]:

```
data.hist(figsize=(20,10), grid=True, layout=(2, 4), bins = 30);
```



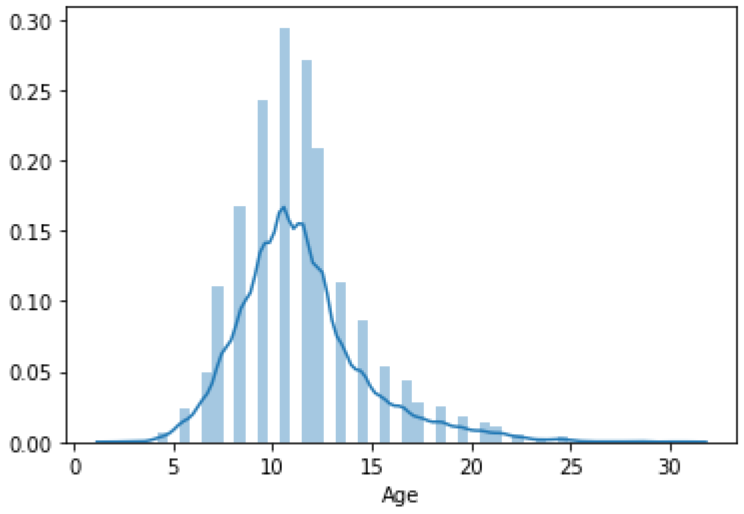
In [249]:
```
#Density Distribution

data.plot(kind='density',layout=(2,4),sharex=False,sharey=False,subplots=True,grid=Fal
          figsize=(15,10));
plt.show()
```



Conclusion : Most of the features are not normally distributed but are skewed to the right. For the height attribute we can see that there might be some outliers. Though the features are not normally distributed but they are tending towards being normally distributed.

Though the other features are not normally distributed, the length and the output variable Age look to be more normally distributed.

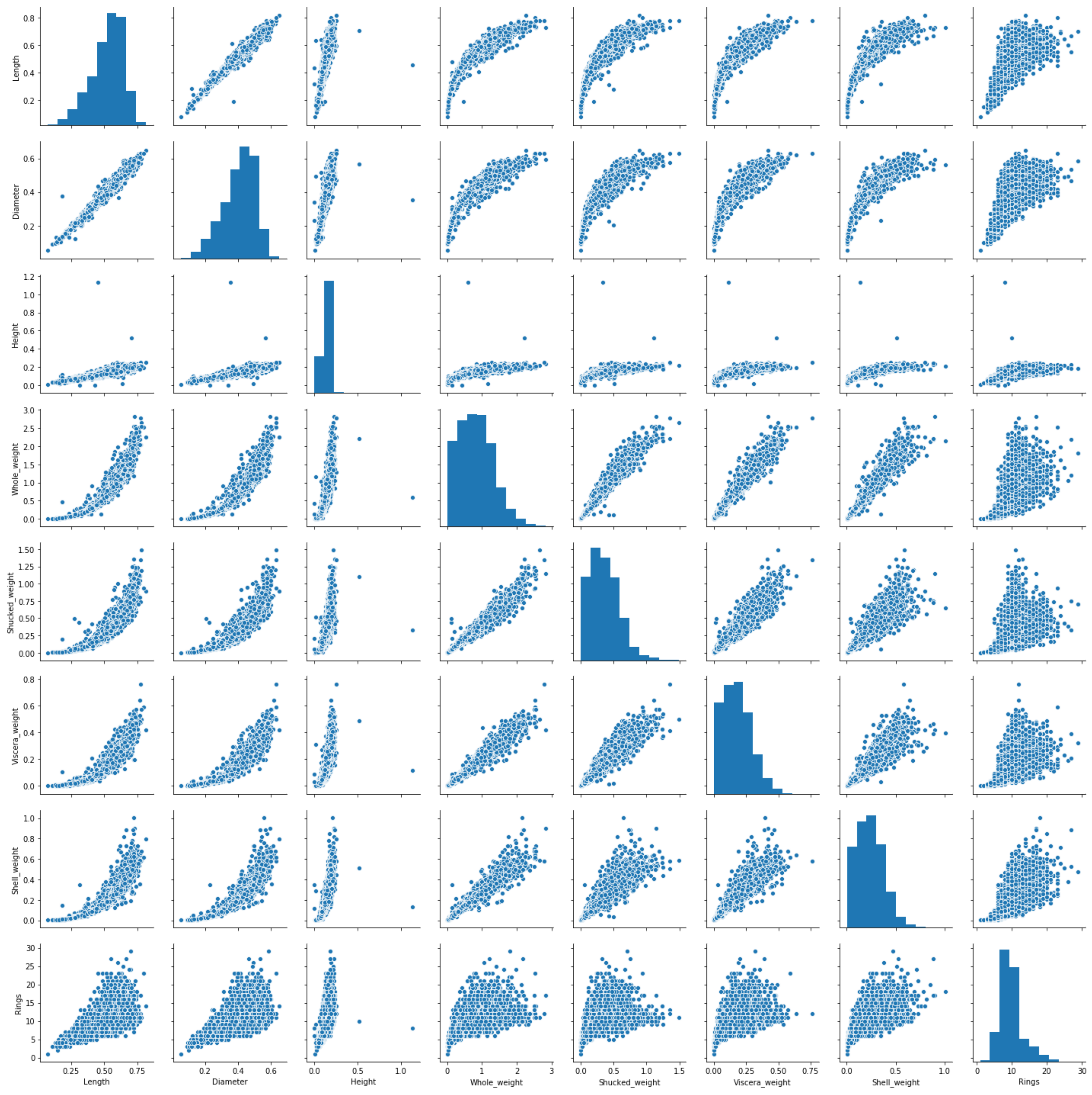
In [40]:
```
sb.distplot(data['Age']);
```



The Age attribute looks to be normally distributed

Correlation between the attributes and check how the different attributes are growing with respect to Age as an output

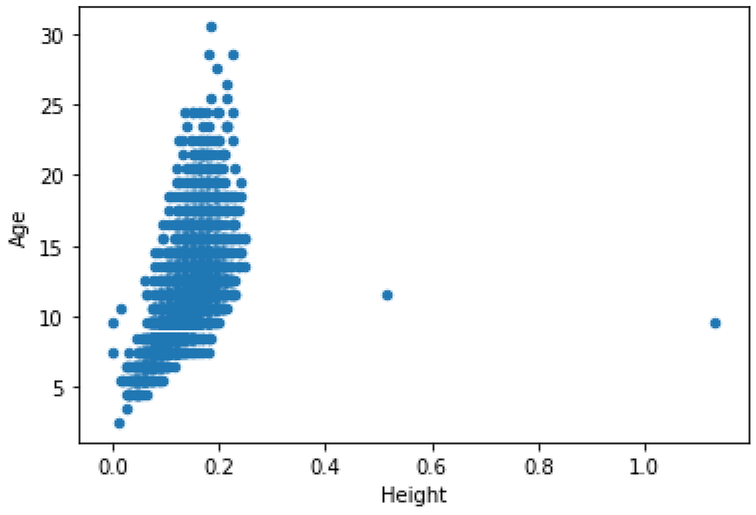
In [242]:
```
axis = sb.pairplot(data);
plt.show()
# Focus on the last row whereby we can see a correlation between the attribtes and the
```



Looking at the above pair plot, let's focus on the last row where we are comparing the different attributes with respect to Age

Based on the above plot, we can see that for certain attributes like Height etc there are some outliers.

In [50]:
```python
plotty=pd.concat([data['Height'],data['Age']],axis=1)
plotty.plot.scatter(x='Height',y='Age');
```

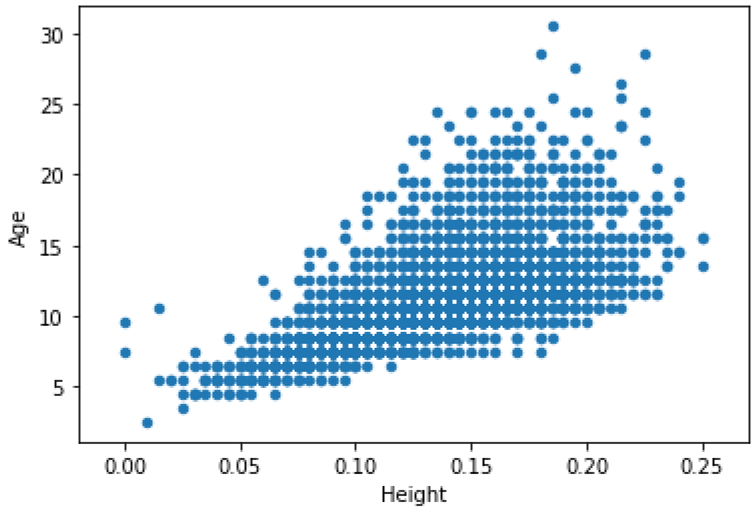


Now we can see that for height values greater than .4 are outliers, So these are possible values which can be removed. These attributes might improve the efficacy of the algorithm

In [53]:
```python
# Removing the outliers values from our dataset
condition = (data['Height']>0.4)
data[condition]
number_of_records = len(data)
data.drop(data[condition].index,inplace=True)
number_of_records = len(data)
print(f'Count after removal : {number_of_records}')
```

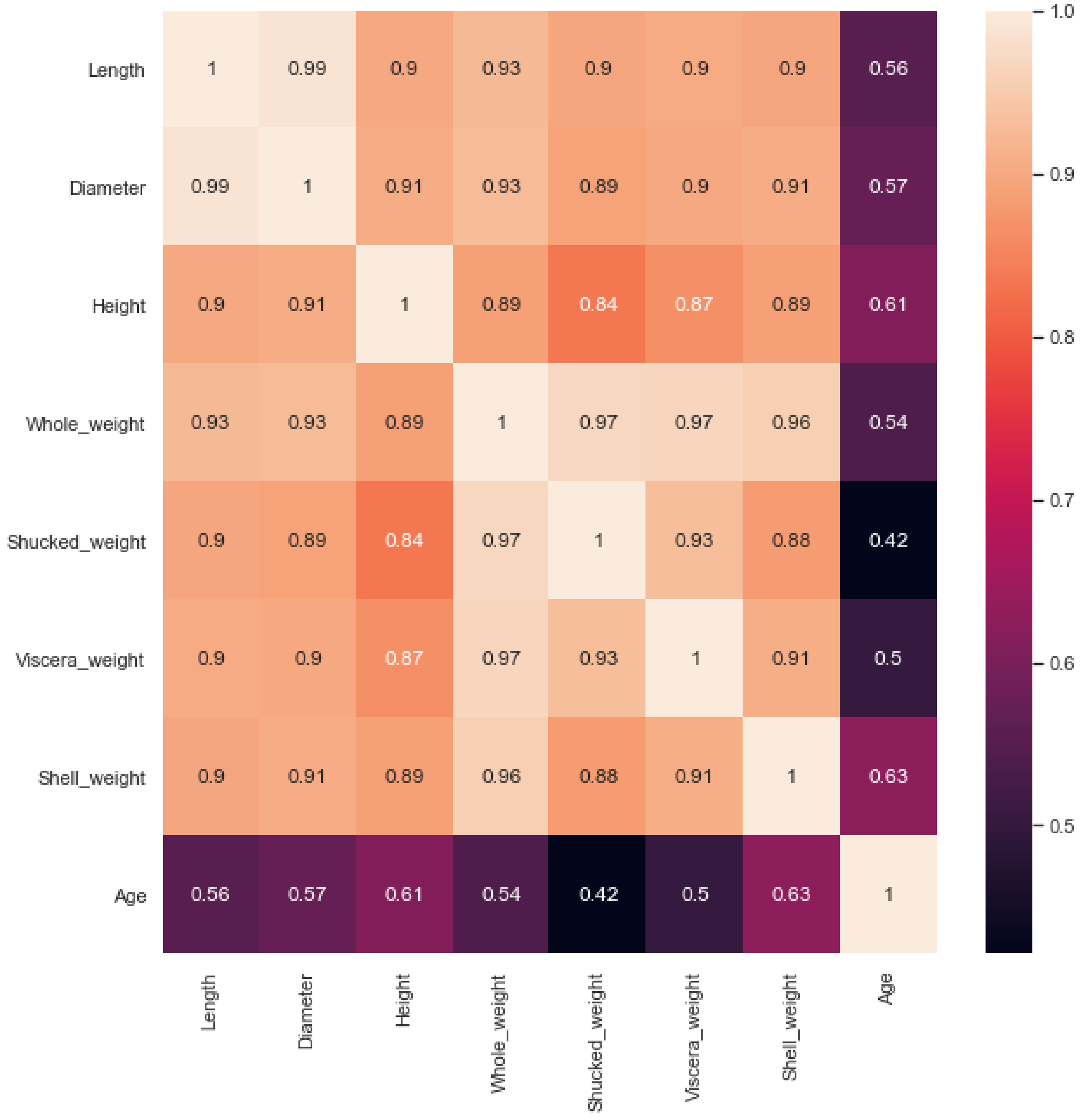
```
Count after removal : 4175
```

In [52]:
```python
# Plotting again to check if the records have been removed and to see if the outlier i
plotty=pd.concat([data['Height'],data['Age']],axis=1)
plotty.plot.scatter(x='Height',y='Age');
```



Correlation analysis

In [84]:
```python
plt.figure(figsize=(10, 10))
corr = data.corr()
ax = sb.heatmap(corr, annot=True)
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5);
```

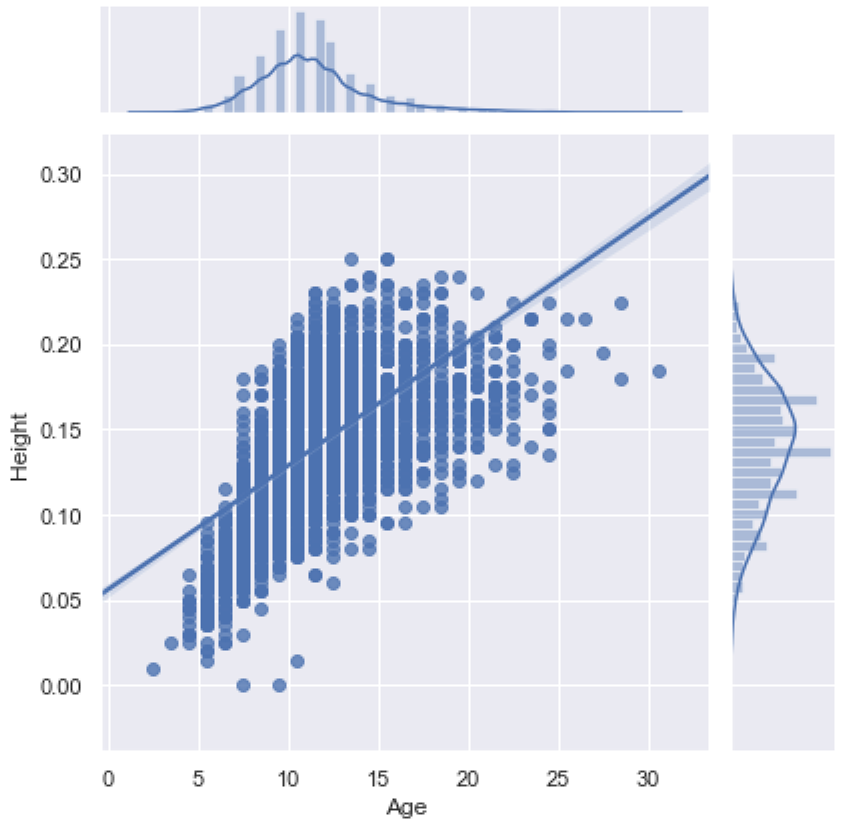


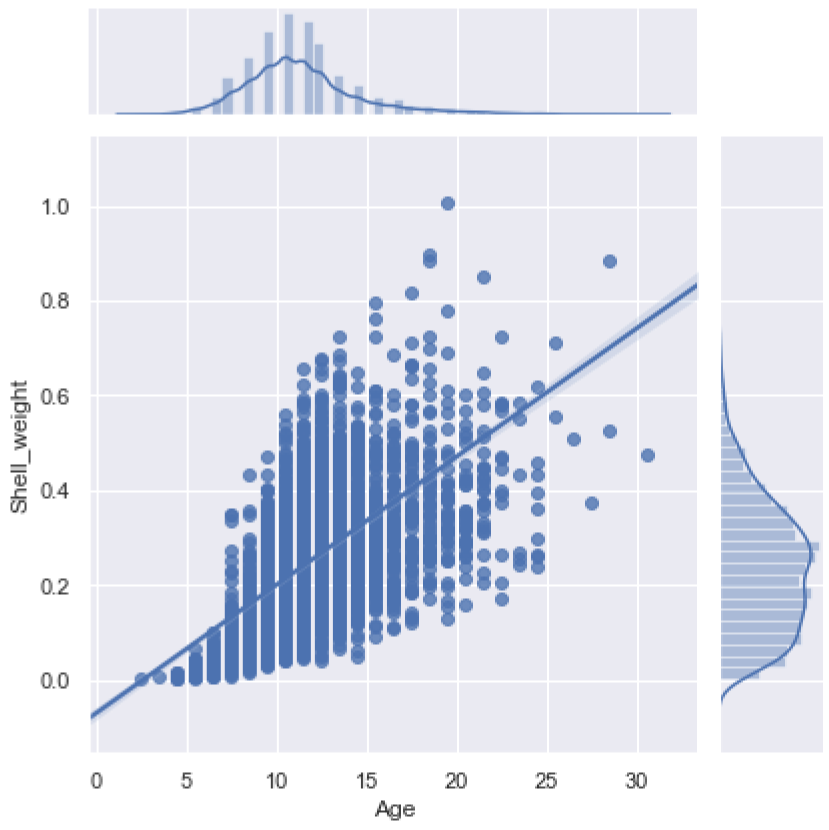
	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Age
Length	1	0.99	0.9	0.93	0.9	0.9	0.9	0.56
Diameter	0.99	1	0.91	0.93	0.89	0.9	0.91	0.57
Height	0.9	0.91	1	0.89	0.84	0.87	0.89	0.61
Whole_weight	0.93	0.93	0.89	1	0.97	0.97	0.96	0.54
Shucked_weight	0.9	0.89	0.84	0.97	1	0.93	0.88	0.42
Viscera_weight	0.9	0.9	0.87	0.97	0.93	1	0.91	0.5
Shell_weight	0.9	0.91	0.89	0.96	0.88	0.91	1	0.63
Age	0.56	0.57	0.61	0.54	0.42	0.5	0.63	1

The correlation matrix, indicates that Height and Shell weight are the attributes that most correlates with respect to Age, so let's focus into those 2 attributes

In [91]:
```
# Let's focus on the Shell Weights and Height as we can see that from the Heat map the
plt.figure(figsize=(20, 5))
sb.jointplot(data=data, x='Age', y='Height', kind='reg');
sb.jointplot(data=data, x='Age', y='Shell_weight', kind='reg');
```

```
<Figure size 1440x360 with 0 Axes>
```





Looking at the above charts, for lower Ages the Shell_weights and Heights are more focussed. For higher values of Age, we can see that the Shell Weights and Shell Weights are dispersed

Encoding the categorical features to numerical outputs (Label Encoding)

Based on the initial data analysiss since Sex is a categorical variable, let's do a label encoding as below

```
- M: 0.333
- F: 0.666
- IL 1.000
```

In [54]:

```python
def loadData(filename):
    # Load data from filename into X
    X = []
    count = 0

    text_file = open(filename, "r")
    lines = text_file.readlines()

    for line in lines:
        X.append([])
        words = line.split(",")
        # convert value of first attribute into float
        for word in words:
            if (word == 'M'):
                word = 0.333
            if (word == 'F'):
                word = 0.666
            if (word == 'I'):
                word = 1
            X[count].append(float(word))
        count += 1

    return np.asarray(X)
```

Scaling all the features to bring the features to the same grain.

We would be normalizing the Data set using a Min - Max Scalar. The normalization equation is as below.

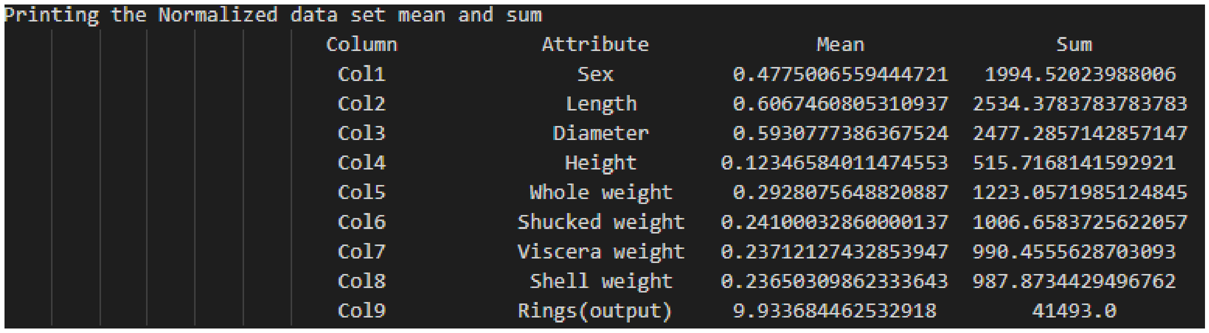
$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

In [58]:

```python
# We loop through the columns and apply the min max scalar formulae here
def dataNorm(X):
    number_of_columns = X.shape[1]
    for i in range(number_of_columns - 1):
        v = X[:, i]
        maximum_value = v.max()
        minimum_value = v.min()
        denominator = maximum_value - minimum_value
        normalized_column = (v - minimum_value) / (denominator)
        X[:, i] = normalized_column

    return X
```

Validate the data below after doing the data normalization

In [252]:
```python
def printMeanAndSum(X):
    print('Printing the Normalized data set mean and sum')
    column_names = ["Column", "Attribute", "Mean", "Sum"]
    attribute_names = ['Sex', 'Length', 'Diameter', 'Height',
                       'Whole weight',
                       'Shucked weight', 'Viscera weight', 'Shell weight'
                       , 'Rings(output)']
    format_row = "{:^20}" * (len(column_names)+1)
    print(format_row.format("", *column_names))

    number_of_columns = X.shape[1]
    for i in range(number_of_columns):
        mean_value = np.mean(X[:, i], axis=0)
        sum_value = np.sum(X[:, i], axis=0)
        column_number = 'Col' + str(i+1)
        row = [column_number, attribute_names[i],  mean_value, sum_value]
        print(format_row.format('', *row))
```



```
Printing the Normalized data set mean and sum
                           Column           Attribute             Mean                 Sum
                            Col1               Sex          0.4775006559444721   1994.52023988006
                            Col2              Length        0.6067460805310937  2534.3783783783783
                            Col3             Diameter       0.5930777386367524  2477.2857142857147
                            Col4              Height       0.12346584011474553   515.7168141592921
                            Col5           Whole weight     0.2928075648820887  1223.0571985124845
                            Col6          Shucked weight   0.24100032860000137  1006.6583725622057
                            Col7          Viscera weight   0.23712127432853947   990.4555628703093
                            Col8           Shell weight    0.23650309862333643   987.8734429496762
                            Col9          Rings(output)     9.933684462532918        41493.0
```

Split the dataset into training and test set

1. Using train-and-test split

2. Using K-Fold cross validation

Train and Test Split

The below function would be provided a dataset and the percentage of data which has to be spliited as train data set The remaining dataset would be would be returned as Test data set

In [60]:
```python
def splitTT(X, percentTrain):
    np.random.shuffle(X)
    N = len(X)
    sample = int(percentTrain*N)
    x_train, x_test = X[:sample, :], X[sample:, :]
    return [x_train, x_test]
```

Using K-Fold cross validation

The below function returns a list X_split=[X1,X2,…,XK] given the number of folds needed.

In [ ]:
```python
def splitCV(X, folds):
    np.random.shuffle(X)
    split_array = np.array_split(X, folds)
    return split_array
```

In [243]:
```python

def k_fold_cross_validation(X, k, folds):
    accuracy_listing = []
    actual_predicted_labels = []
    k_fold_partitions = splitCV(X, folds)
    for index, item in enumerate(k_fold_partitions):
        cross_validation_dataset = item
        list_of_items_from_zero_to_index = k_fold_partitions[0:index]
        list_of_items_from_index_to_end = k_fold_partitions[index+1:]
        total_train_list = list_of_items_from_zero_to_index + \
            list_of_items_from_index_to_end
        train_data_set = np.vstack(total_train_list)
        accuracy_for_cross_validation, actual_predicted_labels_from_partition = knn(
            train_data_set, cross_validation_dataset, k)
        accuracy_listing.append(accuracy_for_cross_validation)
        actual_predicted_labels.append(actual_predicted_labels_from_partition)

    accuracy_average = np.average(accuracy_listing)

    print(
        f'k-value :{k}, Folds : {folds}, Accuracy Average : {accuracy_average} ')
    return accuracy_average, actual_predicted_labels
```

Distance Selection to be used in our KNN

We have a few distance options which we could choose from for figuring out the similarity between the data points.
We would be selecting the Euclidian Distance for the above use case.

Euclidean Distance = np.sqrt((a1-b1)^2 + (a2-b2)^2)
Manhattan distance = np.abs(a1-b1) + np.abs(a2-b2)
Cosine Distance = 1 - [((a1xb1)+(a2xb2)) / (np.sqrt(a1^2 + a2^2) x np.sqrt(b1^2 + b2^2))]
Cosine distance = 1 - Cosine_Similarity
Minkowski distance = ((a1-b1)^q + (a2-b2)^q)^q

Distance functions

Euclidean $$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan $$\sum_{i=1}^{k}|x_i - y_i|$$

Minkowski $$\left(\sum_{i=1}^{k}(|x_i - y_i|)^q\right)^{1/q}$$

We would be going for Eucludian distance for the Abalone data to figure out the similarity between any two data points

Euclidean Distance = np.sqrt((a1-b1)^2 + (a2-b2)^2)

Calculation Euclidian Distance which is L2-distance between the a set of test vectors, held in a matrix X (MxD), and a set of training vectors, held in a matrix Train Matrix (NxD). can be done in 2 ways :

```
1. Looping through each test vector and subtract it from the entire training matri
x. We use numpy addition broadcasting. As long as the dimensions match up, numpy k
nows to do a row-wise subtraction if the element on the right is one-dimensional.W
e can do an element-wise square and sum along the column dimension to get a single
row of the distance matrix for test vector i.
2. For every 2 vectors A and B the L2 distance can be given as in the below formul
ae. To vectorize we need to express this operation for ALL the vectors at once in
 numpy. The first vector is (x1,y1) and the second vector is (x2,y2). The distance
calculation can happen based on the below equations.
```

$$(x-y)^2 = x^2 + y^2 - 2\,x\,y$$
$$(y2 - y1)^2 + (x2 - x1)^2$$
$$x1^2 - 2\,x1\,x2 + x2^2 + v1^2 - 2\,v1\,v2 + v2^2$$
$$x1^2 + x2^2 + y1^2 + y2^2 - 2\,(x1\,x2 + y1\,y2)$$

In [1]:
```python
def distance_matrix(X_Train, X_Test, k):
    test_excluding_output_column = X_Test[:, :8]
    train_excluding_output_column = X_Train[:, :8]

    dists = (-2 * np.dot(test_excluding_output_column, train_excluding_output_column.T
    + np.sum(train_excluding_output_column**2, axis=1)
    + np.sum(test_excluding_output_column**2, axis=1)[:, np.newaxis])**0.5
```

In the above formulae

x1^2 + x2^2 can be mapped to np.sum(train_excluding_output_column^2, axis=1)
y1^2 _ y2^2 can be mapped to np.sum(test_excluding_output_column^2, axis=1)[:, np.newaxis]
-2(x1*x2 + y1*y2) can be mapped to -2 * np.dot(test_excluding_output_column, train_excluding_output_column.T)

Calculating Euclidian Distance this way gives us a record timing of (.02 - .04) second instead of around 50 seconds which it takes to calculate it using the first approach.

KNN Implementation

The implementation is passed a Train data set, Test data set and the hyper parameter K.
Since KNN is a Lazy prediction algorithm, do note that the test data set initially contributes a lot in figuring out the Hyper parameter k.
The most similar neighbors collected from the training dataset can be used to make predictions.

The Euclidian Distance approach mentioned in the above part would be used to calculate the distances with respect to every data point in the Training Data set.

Voting Strategy : Distance Weighted Voting

Traditionally we use a Majority voting strategy whereby for a given list of k nearest neighbours we would pick the classes which have a higher frequency.

But there are often scenarios whereby we might have a point which is is surrounded by a few points which are at a much shorter distance as compared to the other points which win in the majority vote but might not truely clasiffy the point.

We use a weighted Voting strategy whereby we keep the weights as (1/distance) and this weight factor multiplied by the frequency of occcurence becomes the voting criteria for the data point.

By using this approach we tend to penalize long distances and reward short distances.

Accuracy Metric Calculation :

Accuracy metric is being calculated as simply the ratio of right to wrong answers. Given n data points in the test data set, if we are able to predict X percentage of data points correctly, that becomes the accuracy

Accuracy = Correctly predicted data points / Total test data points

In [2]:
```python
# KNN Implementation : returns an accuracy and list of predicted outputs from KNN

def knn(X_Train, X_Test, k):

    test_excluding_output_column = X_Test[:, :8]
    train_excluding_output_column = X_Train[:, :8]


    dists = (-2 * np.dot(test_excluding_output_column,
                         train_excluding_output_column.T)
    + np.sum(train_excluding_output_column**2, axis=1) + \
        np.sum(test_excluding_output_column**2, axis=1)[:, np.newaxis])**0.5

    correct_count = 0
    actual_predicted_labels = []
    sorted_distances = np.argsort(dists)
    k_nearest_distances_indexes = sorted_distances[:, :k]

    for item_index, neighbour_index_listing in enumerate(k_nearest_distances_indexes):
        nearest_neighbours = []
        actual_label = X_Test[item_index][8]
        counter = {}
        for neighbour_index in neighbour_index_listing:
            class_label_in_train_data_set = X_Train[neighbour_index][-1]
            distance_from_this_point = dists[item_index][neighbour_index]
            weight = (1/distance_from_this_point)
            nearest_neighbours.append(
                (class_label_in_train_data_set, distance_from_this_point, weight))
            counter[class_label_in_train_data_set] = counter.get(
                class_label_in_train_data_set, 0)+weight
        predicted_label = max(counter, key=counter.get)
        actual_predicted_labels.append((actual_label, predicted_label))
        if predicted_label == actual_label:
            correct_count += 1

    accuracy = (correct_count / len(X_Test)) * 100

    return accuracy, actual_predicted_labels
```

Performance Classification

The 'k' in KNN algorithm is measured based on feature similarity. Choosing the right value of K is a process called Hyper - parameter tuning and is important for better accuracy.Ideally the distace methodology which we are using might also be considered as a Hyper parameter but we are fixing this to be Euclidian Distance. Finding the value of k is not easy and needs a few approaches.

1. Trial and Error : Try out different combinations of k till we settle down on a good k value for a given distribution.
2. Choosing a lower K value might make it over fitted with the data.
3. Choosing a higer K value might make it more over fittted but it would have smoother decision boundaries.

Whatever might be the value of k which we might get as a result of our trial and error would depend to a large extent on distribution of the underlying data set. After running the above knn on our data set we come accross the below output for different values of k.

In [1]:

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
stats = pd.read_csv('statistics.csv')
stats['Test_Percentage'] = 100 - stats['Train_Percentage']
stats['Error_Rate'] = 100 - stats['Accuracy']
train_test_stats = stats[stats['Method'] == 'Train and Test']
cross_validation_stats = stats[stats['Method'] == 'K-Fold-Cross-Validation']

train_test_stats = train_test_stats[['Method','Train_Percentage','K_value','Accuracy',
                                     'Error_Rate']]
cross_validation_stats = cross_validation_stats[['Method','Cross_Validation_Fold','K_v

train_test_stats['Train_Percentage'] = train_test_stats['Train_Percentage'].astype(str
train_test_stats['Test_Percentage'] = train_test_stats['Test_Percentage'].astype(str)
train_test_stats['Key'] = "( "+train_test_stats['Method']+ " " +train_test_stats['Trai
train_test_stats_final = train_test_stats[['Method','Key','K_value','Accuracy','Run_Ti

cross_validation_stats['Cross_Validation_Fold'] = cross_validation_stats['Cross_Valida
cross_validation_stats['Key'] = "( "+cross_validation_stats['Method']+ ", " +cross_val
cross_validation_stats_final = cross_validation_stats[['Method','Key','K_value','Accur
concatenated = pd.concat([train_test_stats_final, cross_validation_stats_final])

```

Plotting Train and Test Split and K-fold Cross Validation with Accuracy

Train and Test

In [5]:

```python
pd.pivot_table(train_test_stats,index=["K_value"],values=["Accuracy"],
               columns=["Train_Percentage"],margins=False)
```

Out[5]:

```
                                       Accuracy
Train_Percentage       50.0       60.0       70.0
         K_value
---------------------------------------------------
               1  19.722355  21.304608  19.856459
               5  22.020105  23.399162  22.727273
              10  22.833892  24.476361  23.604466
              15  24.605074  25.134650  24.720893
              20  24.557204  26.152005  25.358852
```

K-fold Cross Validation

In [14]:
```
pd.pivot_table(cross_validation_stats,index=["K_value"],values=["Accuracy"],
               columns=["Cross_Validation_Fold"],margins=False)

```

Out[14]:

	Accuracy		
Cross_Validation_Fold	10	15	5
K_value			
1	19.775338	19.990545	19.559207
5	22.048983	21.858212	22.599891
10	23.628791	23.699836	23.390425
15	24.324120	23.941019	25.065209
20	24.826340	24.226726	23.893104

Plotting K-Fold Cross Validation Split and Train and Test with Run Time

Train and Test

In [8]:
```
pd.pivot_table(train_test_stats,index=["K_value"],values=["Run_Time"],
               columns=["Train_Percentage"],margins=False)
```

Out[8]:

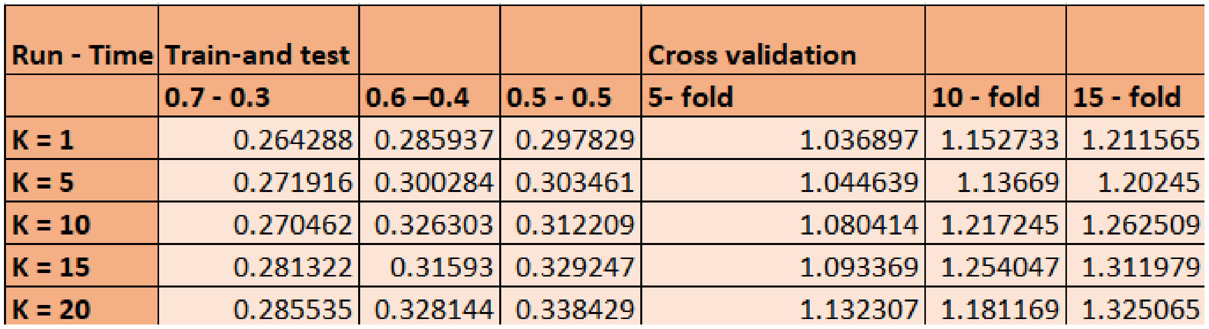
	Run_Time		
Train_Percentage	50.0	60.0	70.0
K_value			
1	0.297829	0.285937	0.264288
5	0.303461	0.300284	0.271916
10	0.312209	0.326303	0.270462
15	0.329247	0.315930	0.281322
20	0.338429	0.328144	0.285535

K-fold Cross Validation

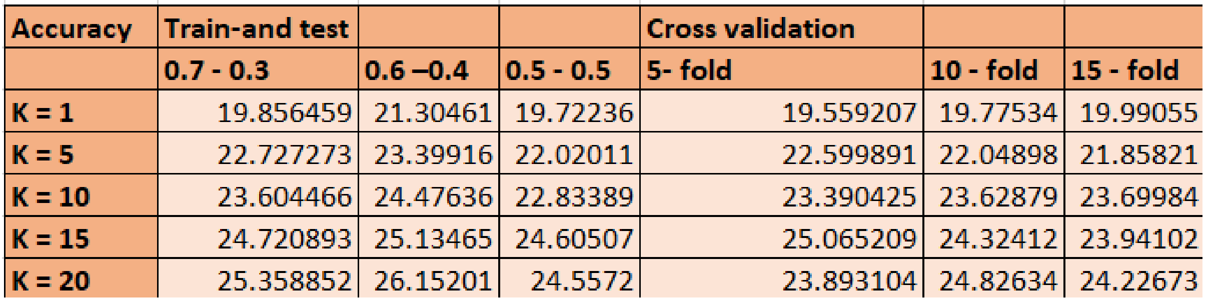
In [16]:
```
pd.pivot_table(cross_validation_stats,index=["K_value"],values=["Run_Time"],
               columns=["Cross_Validation_Fold"],margins=False)
```

Out[16]:
```
                                               Run_Time
Cross_Validation_Fold        10         15            5
K_value
                    1  1.152733   1.211565   1.036897
                    5  1.136690   1.202450   1.044639
                   10  1.217245   1.262509   1.080414
                   15  1.254047   1.311979   1.093369
                   20  1.181169   1.325065   1.132307
```

Consolidated Report for Accuracy and Run Time for Splits



Run - Time	Train-and test			Cross validation		
	0.7 - 0.3	0.6 –0.4	0.5 - 0.5	5- fold	10 - fold	15 - fold
K = 1	0.264288	0.285937	0.297829	1.036897	1.152733	1.211565
K = 5	0.271916	0.300284	0.303461	1.044639	1.13669	1.20245
K = 10	0.270462	0.326303	0.312209	1.080414	1.217245	1.262509
K = 15	0.281322	0.31593	0.329247	1.093369	1.254047	1.311979
K = 20	0.285535	0.328144	0.338429	1.132307	1.181169	1.325065

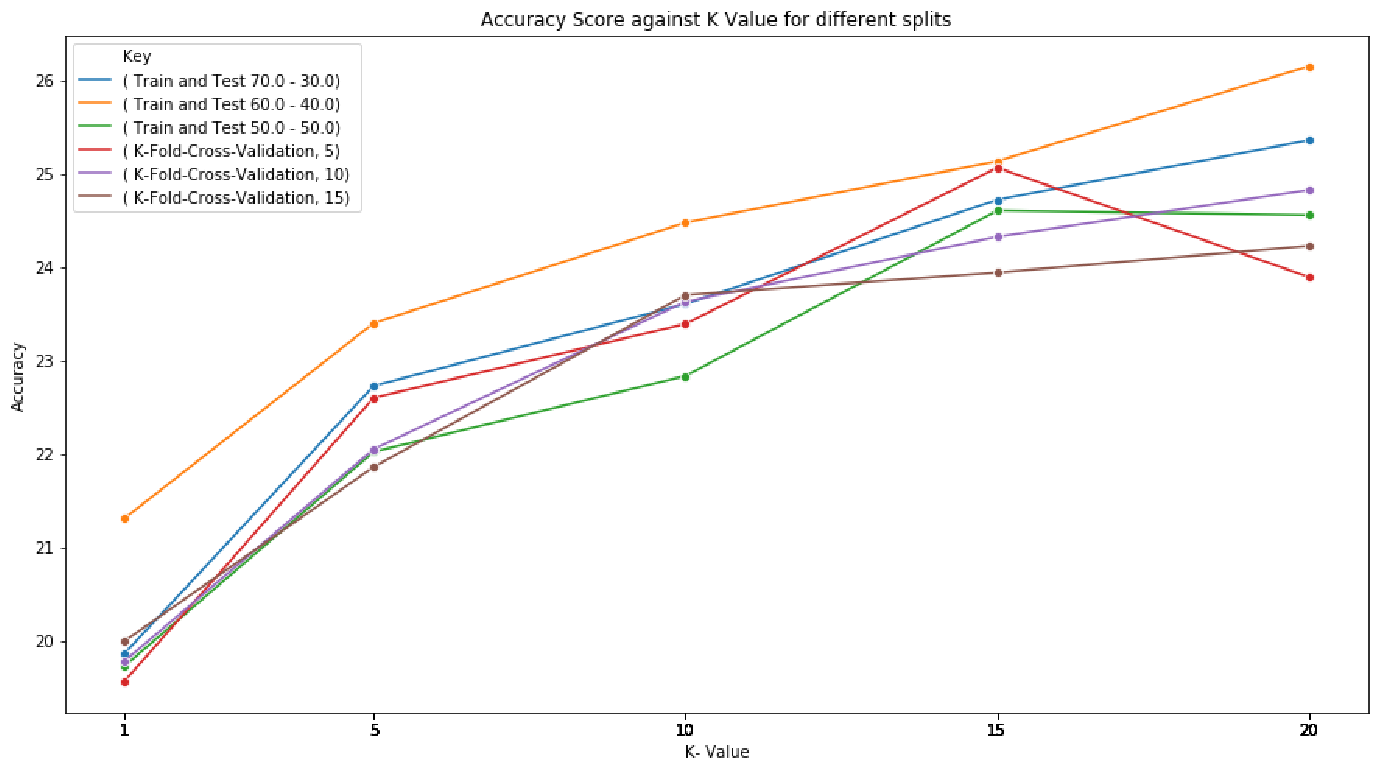


Accuracy	Train-and test			Cross validation		
	0.7 - 0.3	0.6 –0.4	0.5 - 0.5	5- fold	10 - fold	15 - fold
K = 1	19.856459	21.30461	19.72236	19.559207	19.77534	19.99055
K = 5	22.727273	23.39916	22.02011	22.599891	22.04898	21.85821
K = 10	23.604466	24.47636	22.83389	23.390425	23.62879	23.69984
K = 15	24.720893	25.13465	24.60507	25.065209	24.32412	23.94102
K = 20	25.358852	26.15201	24.5572	23.893104	24.82634	24.22673

Plots eliciting relation between K- Values, Training splits / categories and accuracy and run time values

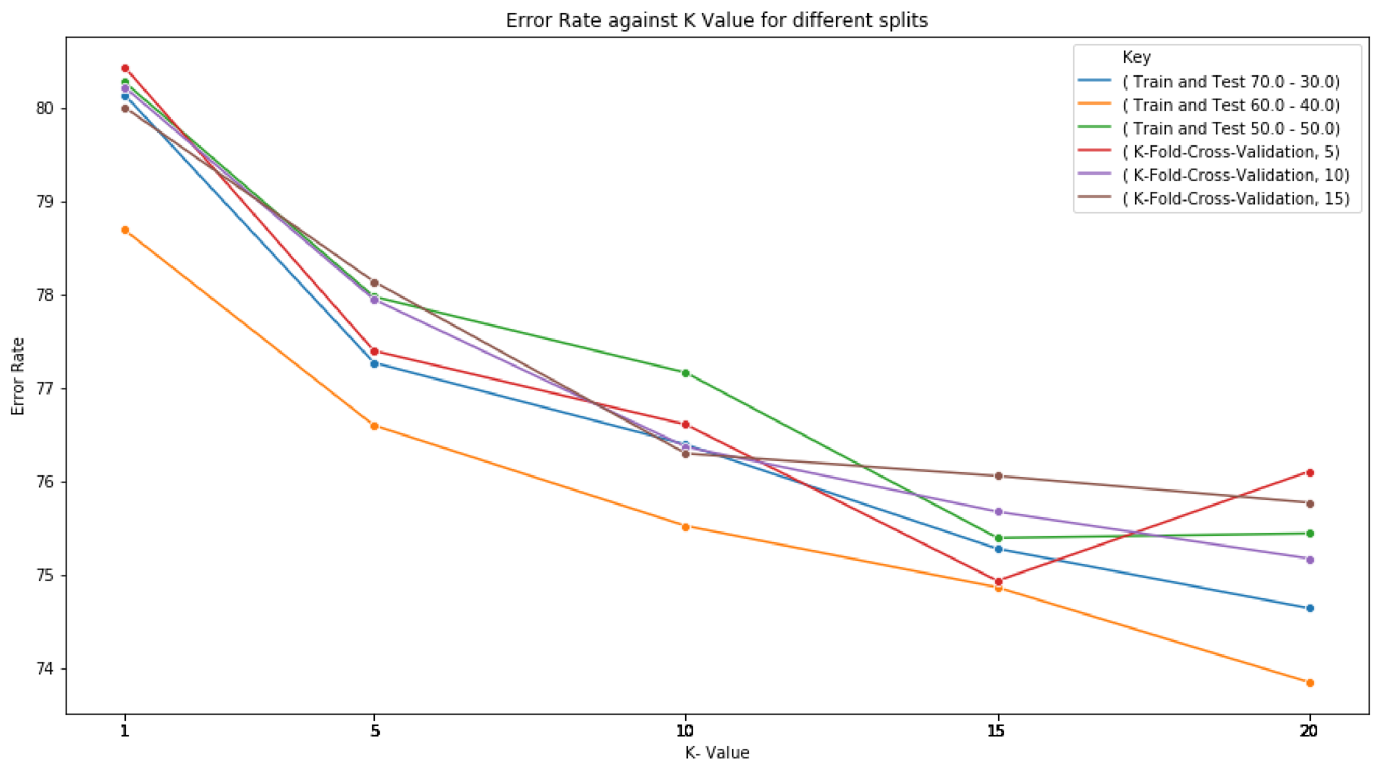
In [10]:

```python
import seaborn as sns
plt.figure(figsize=(15,8))
ax = sns.lineplot(data=concatenated, x='K_value', y='Accuracy', hue='Key',marker="o");
ax.set_title('Accuracy Score against K Value for different splits')
ax.set_ylabel('Accuracy')
ax.set_xlabel('K- Value')
ax.set(xticks=concatenated.K_value.values);

```



As the K- Value increases, the Accuracy figure also goes up. At higher values of K this increment would taper off. We can see that K = 20 has the maximum Accuracy value. Earlier we had seen that the data set is not an unbalanced data set, thus accuracy is a good enough metric to be considered here and we need not look at alternate metrics like Log loss, and we can safely rely on accuracy as a metric to gauge the model performance.

In [11]:
```python
plt.figure(figsize=(15,8))
ax = sns.lineplot(data=concatenated, x='K_value', y='Error_Rate', hue='Key',marker="o"
ax.set_title('Error Rate against K Value for different splits')
ax.set_ylabel('Error Rate')
ax.set_xlabel('K- Value')
ax.set(xticks=concatenated.K_value.values);
```



As the K- Value increases, the Error Rate figure Decreases. For K = 1 we would be having execessive overfitting and as we see the curve starts smoothing up as and when the k-value increases. At very high values of K, the error rates would start increasing and would be the maximum as the k-value apporaches N.

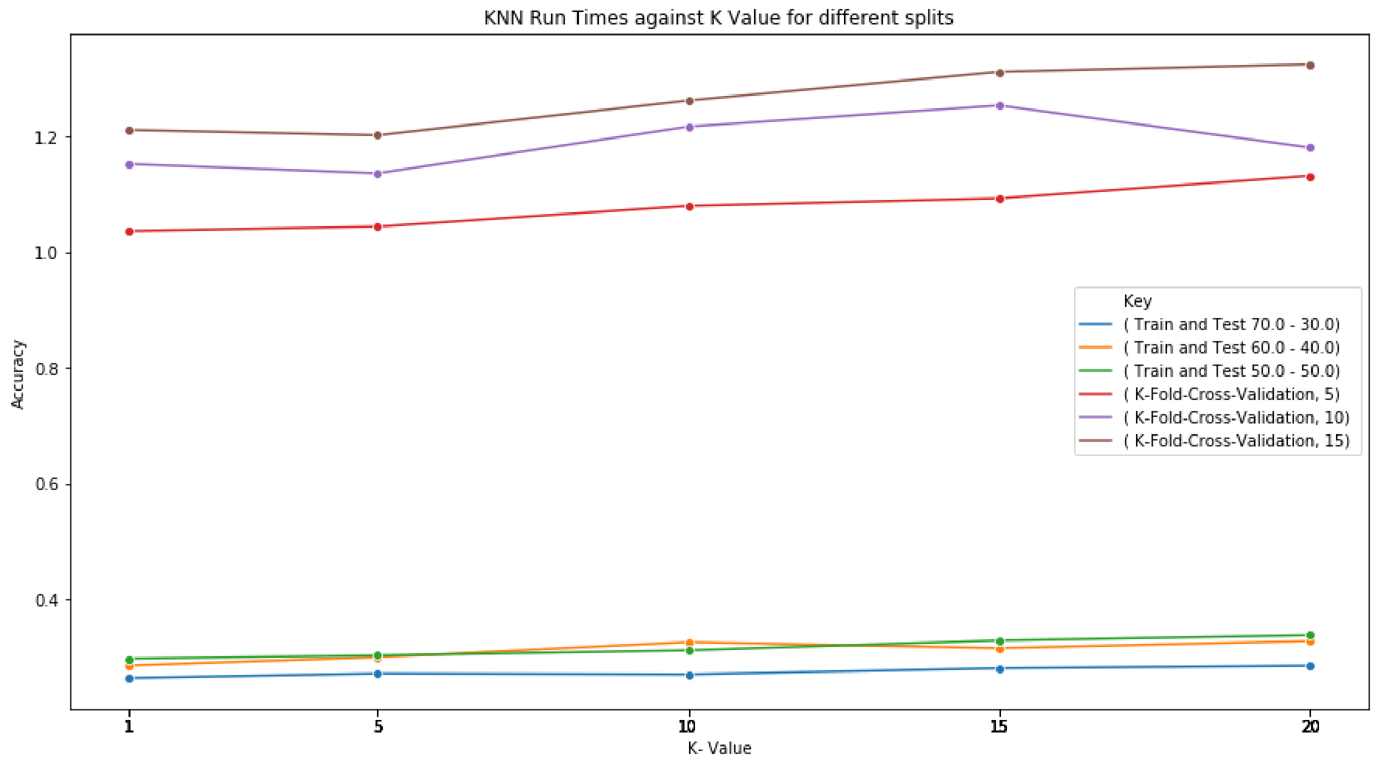
Findings

1.) The Accuracy vs K- Value curve clearly shows that as the value of K increases the accuracy tends to go up.
2.) For K = 1, we can see that there is huge Error Rate as the Decision Boundary tries to overfit the distribution and starts catering to outliers or sporadic points in the alternate class clusters.
3.) As K increases to 5, 10, 15, 20 we can see that the error rate starts decreasing as the Decision Boundary starts smoothening up and it is less prone to noise
4.) As K increases and approaches much higher values K ~ N, we can see that the Accuracy starts dropping as the model starts becoming underfit and it starts predicting the majority class value for most of the inputs.
5.) One common observation is that the decision surfaces start becoming smooth as and when the K-Value increases.
6.) We can see that the mean accuracy of 24% is better than the baseline of 19 - 20 %, but is quite poor in general.This is because of the large number of classes which makes accuracy a poor judegement factor.
7.) We can also see from the below attached classification report that many of the classes have few or one

data point, so the possibility of predicting those data points is very difficult as for any value of K, the possibility of them overpowering other classes with even lower weights is very less, because of their frequency of occurence.

KNN Run time vs K- Value for different split scenarios

In [12]:
```
plt.figure(figsize=(15,8))
ax = sns.lineplot(data=concatenated, x='K_value', y='Run_Time', hue='Key',marker="o");
ax.set_title('KNN Run Times against K Value for different splits')
ax.set_ylabel('Accuracy')
ax.set_xlabel('K- Value')
ax.set(xticks=concatenated.K_value.values);

```



Findings

1.) There is huge difference in the run times between K-Fold Cross Validation runs and Train and Test runs. This can be explained as in the K fold runs, we would be calculating the distances multiple times for every K-fold as the algo is Lazy learning algorithm and would not predicting from a stored heurestics or model.

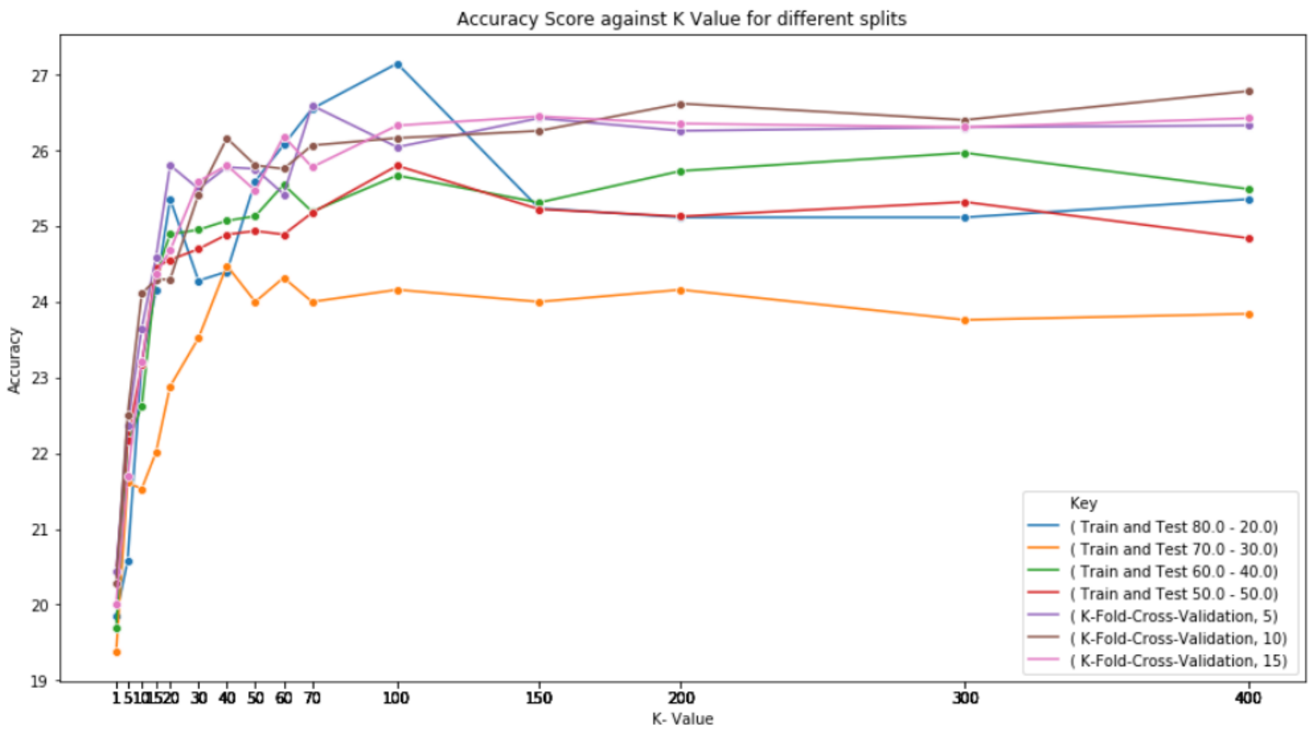
2.) Increasing the K Value does not contribute a lot of run time as we are still calculating the distances across all the data points from the test data points.

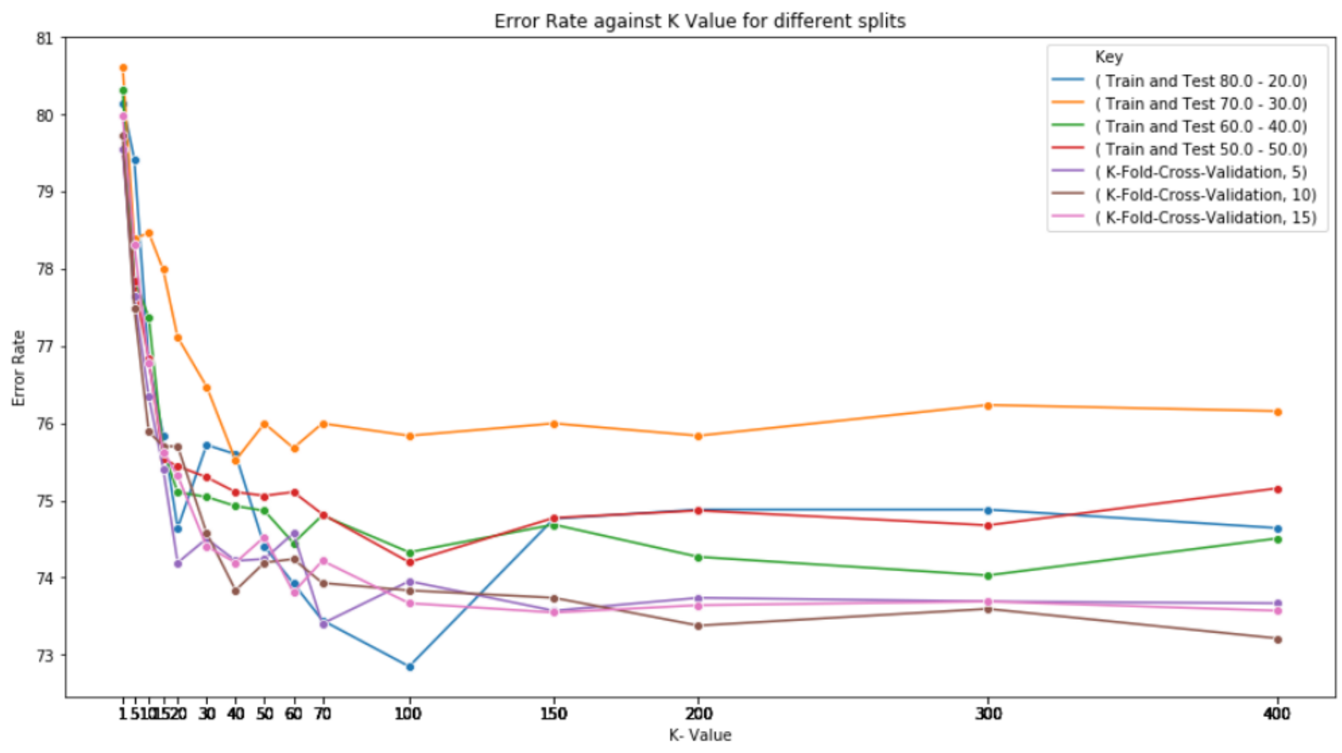
3.) In an alternate run, doing the same runs with the distance function implemented as a single looping as discussed in the Distance calculation section, showed a significant difference as the time taken to calculate the distance across any one single run took around 50 seconds.

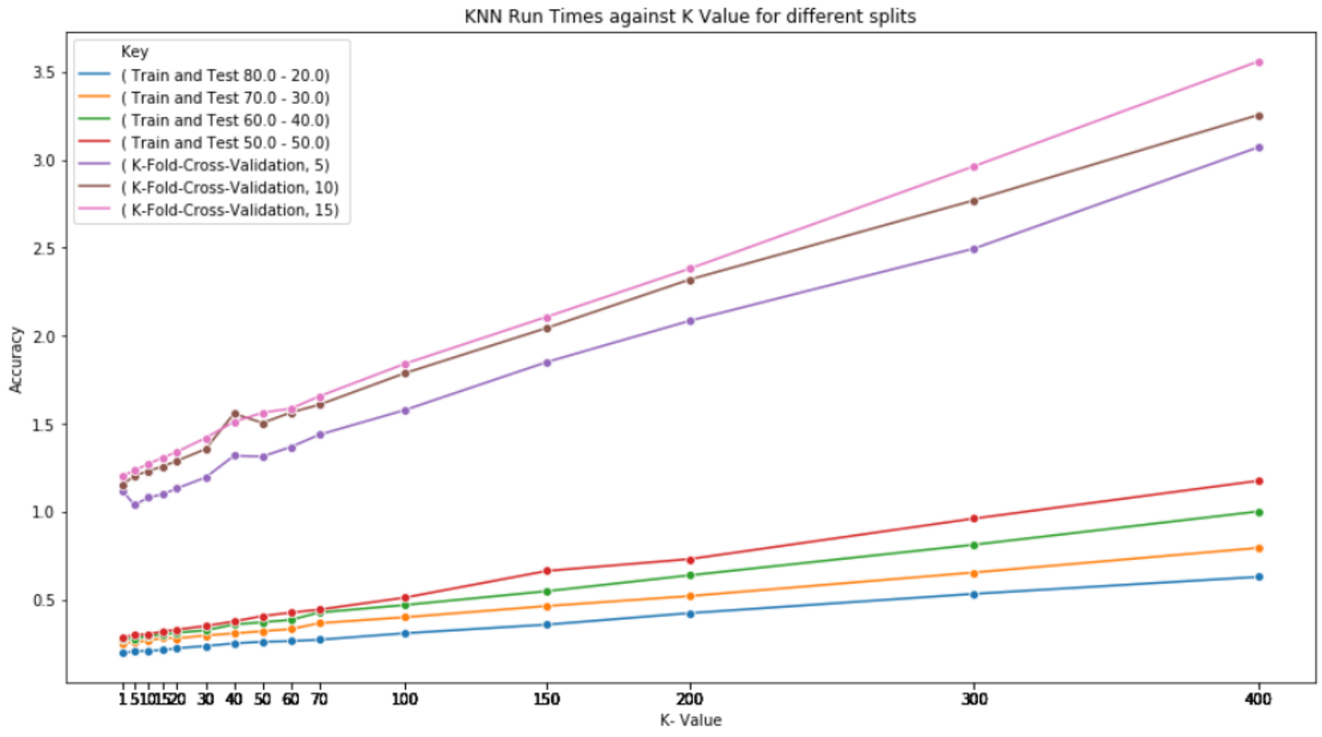
4.) We could implement Locality Sensitive Hashing to improve the performance of the run time as the time taken to get the k nearest neighbours would drastically reduce as we would be saving the conjugal points in a LSH data structure which would be hashed. Subsequent addition of new points to this data set would have to be placed in this data structure

Accuracy, Error Rate and Run Time at Higher K Values

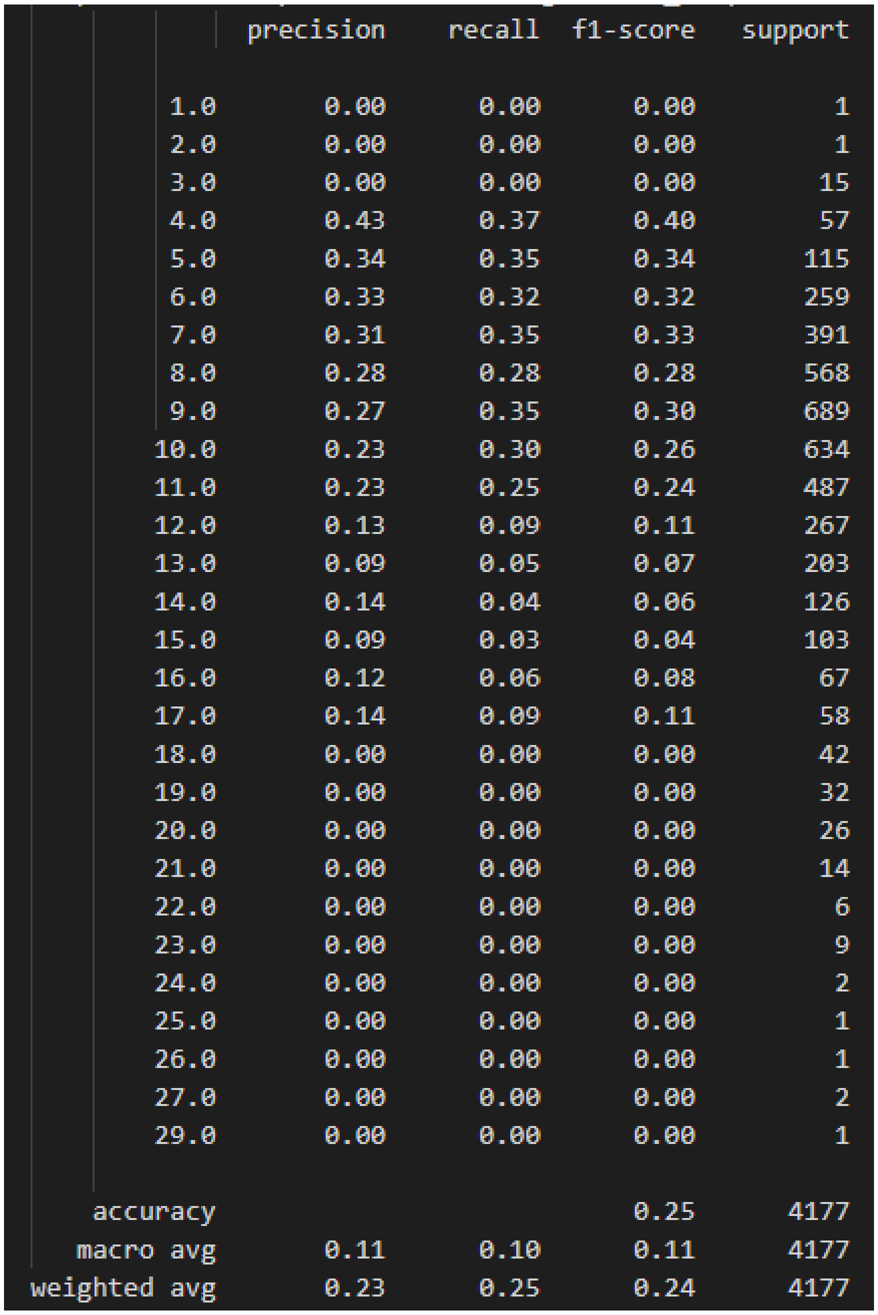
On running the above KNN algorathm on higher K Values like 40,50,60,70,100,150,300,400 we can see that the Accuracy starts dropping and the error rate starts stabilizing and eventually increasing.
for some of the keys. As we approach K = N theoretically the Error rate should increase drastically as for every prediction, it would end up giving the class label with the max frequency as the answer mostly.Do note that the run time increases drastically as we move to higher K Values as a lot many iterations are needed for higher K values






KNN Run Times against K Value for different splits
Key
(Train and Test 80.0 - 20.0)
(Train and Test 70.0 - 30.0)
(Train and Test 60.0 - 40.0)
(Train and Test 50.0 - 50.0)
(K-Fold-Cross-Validation, 5)
(K-Fold-Cross-Validation, 10)
(K-Fold-Cross-Validation, 15)
Accuracy
K- Value
3.5
3.0
2.5
2.0
1.5
1.0
0.5
1
5
10
15
20
30
40
50
60
70
100
150
200
300
400

Classification Report for 5 fold cross validation and K = 15



```
              precision    recall  f1-score   support

         1.0       0.00      0.00      0.00         1
         2.0       0.00      0.00      0.00         1
         3.0       0.00      0.00      0.00        15
         4.0       0.43      0.37      0.40        57
         5.0       0.34      0.35      0.34       115
         6.0       0.33      0.32      0.32       259
         7.0       0.31      0.35      0.33       391
         8.0       0.28      0.28      0.28       568
         9.0       0.27      0.35      0.30       689
        10.0       0.23      0.30      0.26       634
        11.0       0.23      0.25      0.24       487
        12.0       0.13      0.09      0.11       267
        13.0       0.09      0.05      0.07       203
        14.0       0.14      0.04      0.06       126
        15.0       0.09      0.03      0.04       103
        16.0       0.12      0.06      0.08        67
        17.0       0.14      0.09      0.11        58
        18.0       0.00      0.00      0.00        42
        19.0       0.00      0.00      0.00        32
        20.0       0.00      0.00      0.00        26
        21.0       0.00      0.00      0.00        14
        22.0       0.00      0.00      0.00         6
        23.0       0.00      0.00      0.00         9
        24.0       0.00      0.00      0.00         2
        25.0       0.00      0.00      0.00         1
        26.0       0.00      0.00      0.00         1
        27.0       0.00      0.00      0.00         2
        29.0       0.00      0.00      0.00         1

    accuracy                           0.25      4177
   macro avg       0.11      0.10      0.11      4177
weighted avg       0.23      0.25      0.24      4177
```

Possible Improvements on the way I have implemented KNN Algorithm

I could have tried out the below improvements on the way I have implemented the KNN Algorithm

1. The time taken to calculate the Nearest Neighbour for a single data point in the test set is O(nd) where d is the number of dimensions. If we have let's say k data points in our test data set this might go on to O(ndk). Now assuming we move this model to production, O(nd) is an unacceptable time taken for us to fetch the prediction results. The reason it takes this long a time is because we need to loop through every test point and figure out the distance between the test data point and all other data points. This contributes to the fact that we call KNN a Lazy loading algorithm. We could improve this by employing a data structure which partitions and stores the data points in such a way that the nearest neighbour fetch happens in a faster way. I could have used the below 2 techniques

   ```
   a. KD Tree
   b. Locality Sensitive Hashing
   ```

   KD Tree might be ok for lower dimensions like 2-4 but as the number of dimensions increases the time taken to access nearest neighbour keeps increasing. for lower dimensions we could get a log(n) but as the dimensions increase the time taken to access a data point might be even more than O(n), due to the way the algorithm is structured.

   Locality Sensitive Hashing (LSH) is a state of the art randomized alogarithm which used Hash maps to store neighbouring points together, so given a data point, I don't need to search the whole data set but just need to focus on the set of data points thrown out by the LSH algo. But we need to be mindfull that this is a randomized algo, so it might not give a correct result every single time but the probablity of it giving the correct prediction might be high. Basically the data structure is built in such a way that for any 2 data points which are similar(could use any similarity technique like cosine similarity) the hash function would generate the same has value and the value part in the hash table would have pointer these 2 similar data points and any other such point in its vicinity.

2. Distance Measures : I could have employed more distance measures like Cosine similarity(this one might not be of much help), Manhattan Distances or higher factor Minkowsky distance, to see if it improves the accuracy of the prediction

3. Could have tried Regression approach also on top of the classification : KNN can be used as both a regressor as well as a Classifier. So once we calculate the distances from the test data point and arrive at the list of neighbouring points instead of taking a majority vote, we might just take a mean of all these neighbouring values to arrive at the predicted value. We could club this predicted value with the classification output by taking a mean of (Classifier, Regressor) and see if this could improve the output

4. Try out higher values of KNN. We could try out a higher values of K-Value to see the accuracy level at higher values of K. I did try to run the KNN at higher values as can be seen from the above chart. We could observe that a K value of 60-90 at 80% training data gave us a high accuracy of around 26%. Though compared to the baseline this is a good accuracy but on a whole, other algorithms could give us better results.

5. Sanitizing the Data : Certain outliers which we initially found out in the data set could have been removed. I did not remove it but it might have contributed towards better accuracy for lower k Values. for higher K-Values the boundaries would be smoothened though.

6. Dimentionality Reduction : Dimentionality reduction techniques like PCA can be employed to make some of the distance metrics more meaning full

References

https://medium.com/@souravdey/l2-distance-matrix-vectorization-trick-26aa3247ac6c (https://medium.com/@souravdey/l2-distance-matrix-vectorization-trick-26aa3247ac6c)
https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn (https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn)
https://serokell.io/blog/ml-optimization (https://serokell.io/blog/ml-optimization)
https://medium.com/better-programming/machine-learning-optimization-methods-and-techniques-56f5a6fc5d0e (https://medium.com/better-programming/machine-learning-optimization-methods-and-techniques-56f5a6fc5d0e)