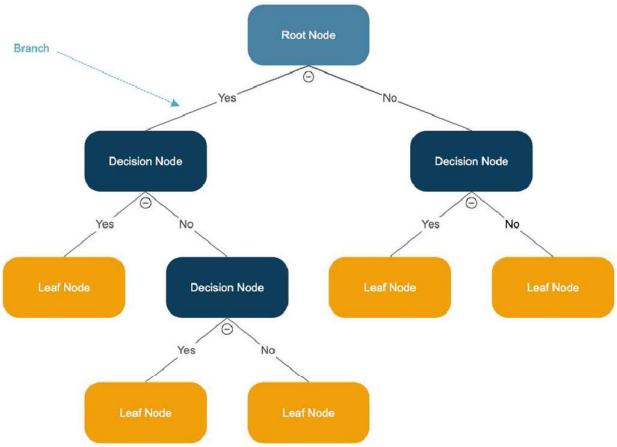
Practical 7

Aim: Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Decision Tree

- A supervised learning method called a decision tree can be used to solve classification and regression problems, but it is typically favoured for doing so.
- It is a tree-structured classifier, where each leaf node represents the classification outcome and inside nodes represent the features of a dataset.
- The Decision Node and Leaf Node are the two nodes of a decision tree. While Leaf nodes are the results of decisions and do not have any more branches, Decision nodes are used to create decisions and have numerous branches.
- The given dataset's features are used to execute the test or make the decisions.
- It is a graphical depiction for obtaining all feasible answers to a choice or problem based on predetermined conditions.
- It is known as a decision tree because, like a tree, it begins with the root node and grows on subsequent branches to form a structure resembling a tree.
- The CART algorithm, which stands for Classification and Regression Tree algorithm, is used to construct a tree.
- A decision tree only poses a question and divides the tree into subtrees according to the response (Yes/No).



Decision Tree Terminologies

- Root Node: The decision tree begins at the root node. It is the complete dataset, which is then separated into two or more homogeneous sets.
- Leaf Node: After receiving a leaf node, the tree cannot be further divided; leaf nodes are the ultimate output nodes.
- Splitting: In splitting, the decision node or root node is divided into sub-nodes in accordance with the specified conditions.
- Branch/Sub Tree: A tree formed by splitting the tree.
- Pruning: Pruning is the procedure of removing the tree's undesirable branches.
- Parent/Child node: The parent node of the tree and the remaining nodes are referred to as the child nodes.

Working of decision tree

1. The decision tree starts with a root node that represents the entire dataset.
2. It selects the best attribute to split the data into subsets of more homogeneous nodes based on the criterion of maximum information gain, Gini index or entropy.
3. This process of dividing the dataset into smaller subsets continues recursively for each child node until a stopping criterion is met, such as reaching a pre-specified maximum depth or minimum number of samples per node.
4. At each node, the decision tree chooses the attribute that can best distinguish between the target classes, resulting in a binary decision.

5. Once the tree is constructed, it can be used to predict the target variable of new or unseen data points by traversing the tree from the root to a leaf node, where the prediction is made.
6. The decision tree can also be pruned to improve its generalization performance by removing branches that do not improve the accuracy of the model.
7. The interpretability of decision trees makes it easier to understand the relationship between the input features and the output target, which can be useful in gaining insights into the underlying problem domain.

Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
Step-3: Divide the S into subsets that contains possible values for the best attributes.
Step-4: Generate the decision tree node, which contains the best attribute.
Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

ID3 ALGORITHM

- It is a classification algorithm that follows a greedy approach by selecting a best attribute that yields maximum Information Gain(IG) or minimum Entropy(H).

Steps in ID3 algorithm

1. Calculate entropy for dataset.
2. For each attribute/feature.
   2.1. Calculate entropy for all its categorical values.
   2.2. Calculate information gain for the feature.
3. Find the feature with maximum information gain.
4. Repeat it until we get the desired tree.

Entropy Gain and Information Gain

- Entropy is a measure of the amount of uncertainty in the dataset S. Mathematical Representation of Entropy is shown here -

$$H(S) = \sum_{c \in C} -p(c) log_2 p(c)$$

Where,

- S - The current dataset for which entropy is being calculated(changes every iteration of the ID3 algorithm).
- C - Set of classes in S {example - C ={yes, no}}
- p(c) - The proportion of the number of elements in class c to the number of elements in set S.

In ID3, entropy is calculated for each remaining attribute. The attribute with the smallest entropy is used to split the set S on that particular iteration.

Entropy = 0 implies it is of pure class, that means all are of same category.

- Information Gain IG(A) tells us how much uncertainty in S was reduced after splitting set S on attribute A. Mathematical representation of Information gain is shown here -

$$IG(A,S) = H(S) - \sum_{t \in T} p(t) H(t)$$

Where,

- H(S) - Entropy of set S.
- T - The subsets created from splitting set S by attribute A such that

$$S = \bigcup_{t \in T} t$$

- p(t) - The proportion of the number of elements in t to the number of elements in set S.
- H(t) - Entropy of subset t.

In ID3, information gain can be calculated (instead of entropy) for each remaining attribute. The attribute with the largest information gain is used to split the set S on that particular iteration.

```
import pandas as pd
import numpy as np
import plotly
import plotly.express as px
import plotly.offline as pyo
import cufflinks as cf
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.graph_objs as go
import  pylab as pl
```

```
iris = pd.read_csv("/content/drive/MyDrive/AIML/Iris.csv")
iris.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

There are 150 rows and 6 columns in iris dataset

```
iris.shape
```

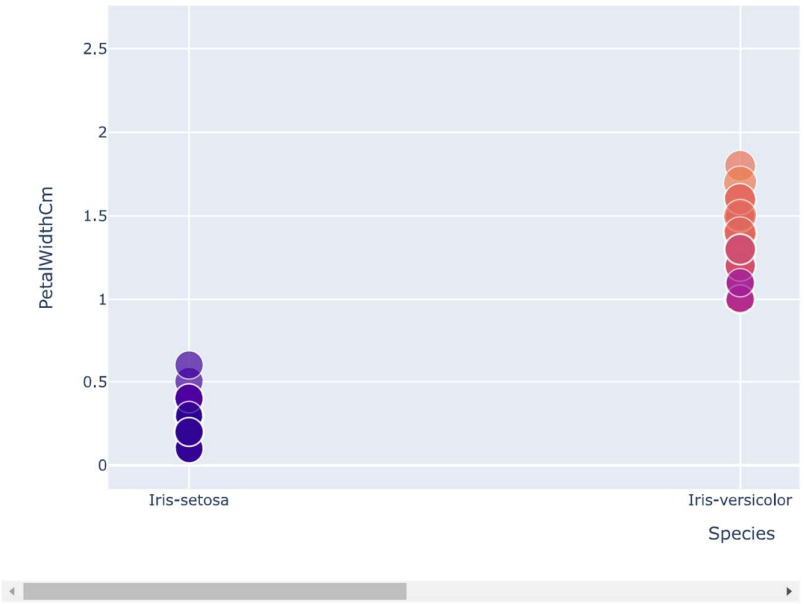
```
(150, 6)
```

```
iris
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species

Creating scatter graph for different labels in species and assigning color based on PetalWidthCm and PetalLengthCm

```
px.scatter(iris,x='Species', y='PetalWidthCm',size= 'SepalLengthCm', color='PetalLengthCm')
```



X doesnot have species columns and only Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm

```
X = iris.drop(['Species'],axis=1)
X
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
0	1	5.1	3.5	1.4	0.2
1	2	4.9	3.0	1.4	0.2
2	3	4.7	3.2	1.3	0.2
3	4	4.6	3.1	1.5	0.2
4	5	5.0	3.6	1.4	0.2
...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3
146	147	6.3	2.5	5.0	1.9
147	148	6.5	3.0	5.2	2.0
148	149	6.2	3.4	5.4	2.3
149	150	5.9	3.0	5.1	1.8

150 rows × 5 columns

Assigning Species column of iris dataset to y

```
y = iris['Species']
y
```

```
0         Iris-setosa
1         Iris-setosa
2         Iris-setosa
3         Iris-setosa
4         Iris-setosa
             ...
145    Iris-virginica
146    Iris-virginica
147    Iris-virginica
148    Iris-virginica
```

```
149    Iris-virginica
Name: Species, Length: 150, dtype: object
```

The value of array according to labels:

1. 0 - setosa
2. 1 - versicolor
3. 2 - virginica

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
y
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Spliting data set in 70% and 30% testing

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.3, random_state=0)
```

DecisionTreeClassifier - The decision tree classifier creates the classification model by building a decision tree.

```
from sklearn.tree import DecisionTreeClassifier
DT = DecisionTreeClassifier(criterion='entropy')
DT.fit(X_train,y_train)
```

```
DecisionTreeClassifier(criterion='entropy')
```

DT.predict - DTs predict the value of a target variable by learning simple decision rules inferred from the data features.

accuracy is multiplied by 100 to get in percentage

```
from sklearn.metrics import accuracy_score, confusion_matrix
pred = DT.predict(X_test)
accuracy = accuracy_score(y_test,pred)*100
```

```
accuracy
```

```
97.77777777777777
```

Data is split in 80% training and 20% testing to find accuracy on this data

```
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.2, random_state=0)
```

```
DT1 = DecisionTreeClassifier(criterion='entropy')
DT1.fit(X_train,y_train)
```

```
DecisionTreeClassifier(criterion='entropy')
```

the accuracy decreased as compare to previous splitting of data

```
pred = DT1.predict(X_test)
accuracy = accuracy_score(y_test,pred)*100
```

```
accuracy
```

```
96.66666666666667
```

Now data is split in 90% training and 10% testing to find another accuracy

```
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.1, random_state=0)
```

```
DT2 = DecisionTreeClassifier(criterion='entropy')
DT2.fit(X_train,y_train)
```

```
DecisionTreeClassifier(criterion='entropy')
```

```
pred = DT2.predict(X_test)
accuracy = accuracy_score(y_test,pred)*100
accuracy
```

```
93.33333333333333
```

Data is split in 60% training and 40% testing to measure accuracy

```
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.4, random_state=0)
```

```
DT3 = DecisionTreeClassifier(criterion='entropy')
DT3.fit(X_train,y_train)
```

```
DecisionTreeClassifier(criterion='entropy')
```

```
pred = DT3.predict(X_test)
accuracy = accuracy_score(y_test,pred)*100
accuracy
```

```
96.66666666666667
```

```
y_test
```

```
array([2, 1, 0, 2, 0, 2, 0, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 0, 0, 2, 1,
       0, 0, 2, 0, 0, 1, 1, 0, 2, 1, 0, 2, 2, 1, 0, 1, 1, 1, 2, 0, 2, 0,
       0, 1, 2, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 2])
```

```
pred
```

```
array([2, 1, 0, 2, 0, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 0, 0, 2, 1,
       0, 0, 2, 0, 0, 1, 1, 0, 2, 1, 0, 2, 2, 1, 0, 1, 1, 1, 2, 0, 2, 0,
       0, 1, 2, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 2, 0, 2])
```

```
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.2, random_state=0)
```

```
DT = DecisionTreeClassifier(criterion='entropy')
DT.fit(X_train,y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy')
```

```
pred = DT1.predict(X_test)
accuracy = accuracy_score(y_test,pred)*100
accuracy
```

```
96.66666666666667
```

```
y_test
```

```
array([2, 1, 0, 2, 0, 2, 0, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 0, 0, 2, 1,
       0, 0, 2, 0, 0, 1, 1, 0])
```

```
pred
```

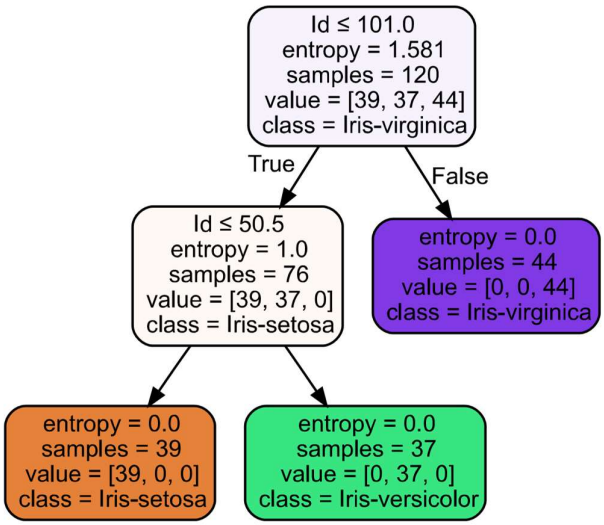
```
array([2, 1, 0, 2, 0, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 0, 0, 2, 1,
       0, 0, 2, 0, 0, 1, 1, 0])
```

```
conf = confusion_matrix(y_test,pred)
conf
```

```
array([[11,  0,  0],
       [ 0, 13,  0],
       [ 0,  1,  5]])
```

This is the visualization of Decision Tree, which shows entropy, samples, value, class at each node

```
import graphviz
from sklearn import tree
vis_dt = tree.export_graphviz(DT, feature_names = iris.drop(['Species'],axis=1).keys(), class_names=iris['Species'].unique(),filled=True,
graphviz.Source(vis_dt)
```



12:28 PM P a a D3 o b a

ID3 Algorithm

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split

# Load the iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create a decision tree classifier with ID3 algorithm
clf = DecisionTreeClassifier(criterion="entropy")

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Use the classifier to predict the classes of the testing data
y_pred = clf.predict(X_test)

# Evaluate the performance of the classifier on the testing data
accuracy = clf.score(X_test, y_test)
print("Accuracy: ",accuracy)
```

```
Accuracy:  0.9666666666666667
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

# Load the iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Create a decision tree classifier with ID3 algorithm
clf = DecisionTreeClassifier(criterion="entropy")

# Train the classifier on the full dataset
clf.fit(X, y)

# Define a new sample to classify
new_sample = [[5.0, 3.6, 1.4, 0.2]]  # This is an example new sample, with sepal length 5.0, sepal width 3.6, petal length 1.4, and petal

# Use the classifier to predict the class of the new sample
predicted_class = clf.predict(new_sample)

# Print the predicted class label
print("Predicted class:", predicted_class)
```

```
Predicted class: [0]
```

Practical 8

Aim: Write a program to classify IRIS data using Random forest classifier.

Random Forest

- Random forest is a widely used ensemble learning algorithm in supervised machine learning, where there is a labeled target variable.
- Random forest can handle both regression (numeric target variable) and classification (categorical target variable) problems effectively.
- Random forest creates multiple decision trees, which are trained on random subsets of the training data with replacement.
- Each tree in the random forest ensemble independently predicts the target variable, and the final output is the average of the predictions made by all the trees or the majority vote.
- Random forest can reduce overfitting by averaging the results of many decision trees, which helps to increase the accuracy and generalization performance of the model.
- Random forest is a robust algorithm that can handle missing data, outliers, and irrelevant features.
- The interpretability of random forest is lower than that of a single decision tree, but it can still provide feature importance measures that help in understanding the relevance of each input variable.

Working of Random Forest

1. Random forest is an ensemble method that creates multiple decision trees on random subsets of the training data with replacement.
2. Each decision tree in the random forest is trained on a random subset of the features, which helps to reduce overfitting and improve the model's generalization performance.
3. During training, the random forest algorithm selects the best split point in each decision tree based on the criterion of maximum information gain, Gini index or entropy.
4. Once all the decision trees are built, the random forest combines their predictions using either the majority voting or averaging method, depending on the problem type.
5. During the prediction phase, a new data point is run through each decision tree in the random forest, and each tree makes a prediction.

6. The final output of the random forest is either the majority vote or the average of all the predictions made by the decision trees.
7. Random forest is a robust algorithm that can handle missing data, outliers, and irrelevant features, making it a popular choice for various machine learning tasks.
8. The random forest algorithm can provide important measures of feature importance that help in understanding the relevance of each input variable.
9. Random forest can be computationally expensive and may require hyperparameter tuning to achieve optimal performance, but it can provide high accuracy and generalization performance for complex problems.

Building a Classifier

```
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
import matplotlib.pyplot as plt
```

```
iris = datasets.load_iris()
```

```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

```
print(iris.feature_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
print(iris.data[0:5])
print(iris.target)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Create a dataframe of given iris dataset

```
data=pd.DataFrame({
    'sepal length':iris.data[:,0],
    'sepal width':iris.data[:,1],
    'petal length':iris.data[:,2],
    'petal width':iris.data[:,3],
    'species':iris.target
})
data.head()
```

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Import train_test_split function

```
X=data[['sepal length', 'sepal width', 'petal length', 'petal width']]
y=data['species']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

```
clf=RandomForestClassifier(n_estimators=100)
clf.fit(X_train,y_train)
y_pred=clf.predict(X_test)
```

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9777777777777777
```

```
species_idx = clf.predict([[3, 5, 4, 2]])[0]
iris.target_names[species_idx]
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/base.py:439: UserWarning: X does no
  warnings.warn(
'virginica'
```

Finding Important Features

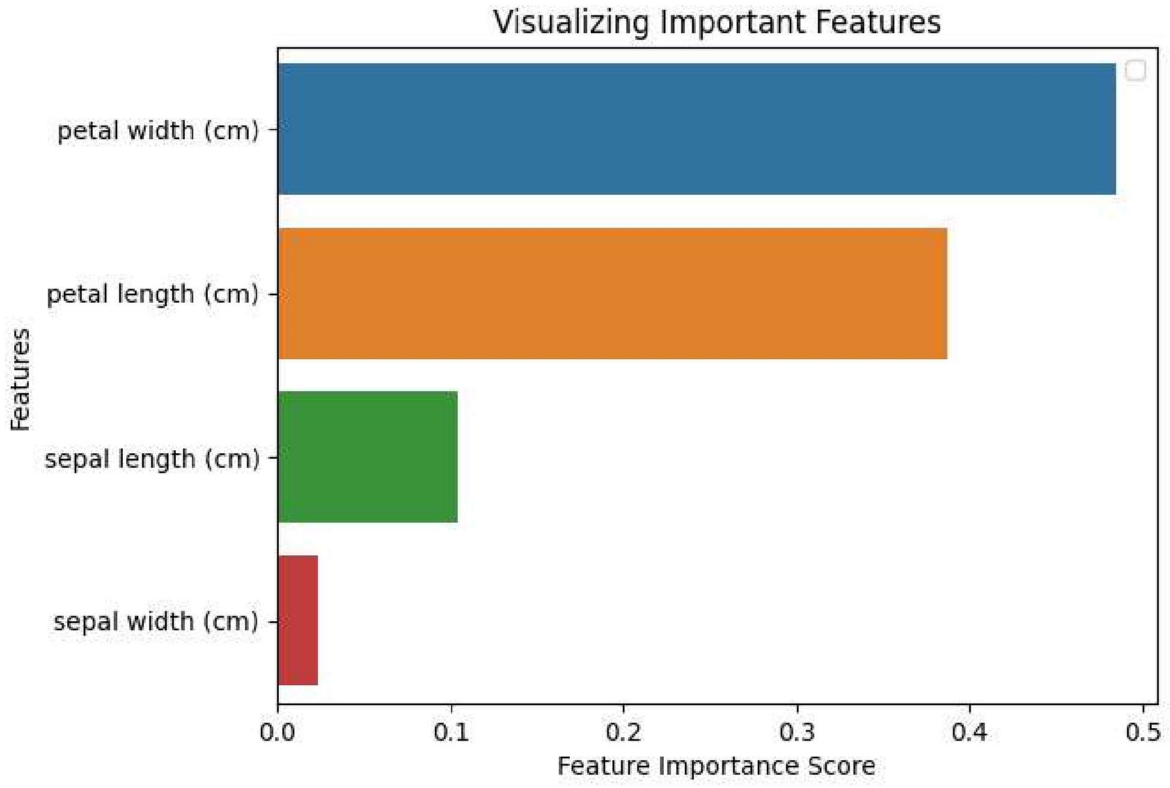
```
feature_imp = pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(asc
feature_imp
```

```
petal width (cm)     0.484564
petal length (cm)    0.387133
sepal length (cm)    0.104687
```

```
sepal width (cm)     0.023617
dtype: float64
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.barplot(x=feature_imp, y=feature_imp.index)
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualizing Important Features")
plt.legend()
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note tha
```



Generating the Model on Selected Features

```
X=data[['petal length', 'petal width','sepal length']]
y=data['species']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
from sklearn.ensemble import RandomForestClassifier
clf=RandomForestClassifier(n_estimators=100)
clf.fit(X_train,y_train)
y_pred=clf.predict(X_test)
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9111111111111111
```

Practical 9

Aim: Write a program to classify iris dataset using SVM. Experiment with different kernel functions.

SVM - Support Vector Machine

SVM stands for Support Vector Machines, which is a popular machine learning algorithm used for classification and regression analysis. It belongs to the family of supervised learning algorithms, which means it requires labeled data to train the model. SVM tries to find the best hyperplane that separates the different classes of data with the largest margin possible. The data points closest to the hyperplane are called support vectors and they define the position and orientation of the hyperplane.

SVM can be of two types:

1. Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
2. Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane

There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.

Support Vector

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Importing the necessary libraries

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
```

Load the iris dataset

```
iris = datasets.load_iris()
```

Split the dataset into features and labels

```
X = iris.data
y = iris.target
```

Split the dataset into training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Define the SVM model with different kernel functions

```
models = [
    {'label': 'Linear kernel', 'model': SVC(kernel='linear')},
    {'label': 'Polynomial kernel', 'model': SVC(kernel='poly', degree=3)},
```

```
    {'label': 'RBF kernel', 'model': SVC(kernel='rbf')},
    {'label': 'Sigmoid kernel', 'model': SVC(kernel='sigmoid')}
]
```

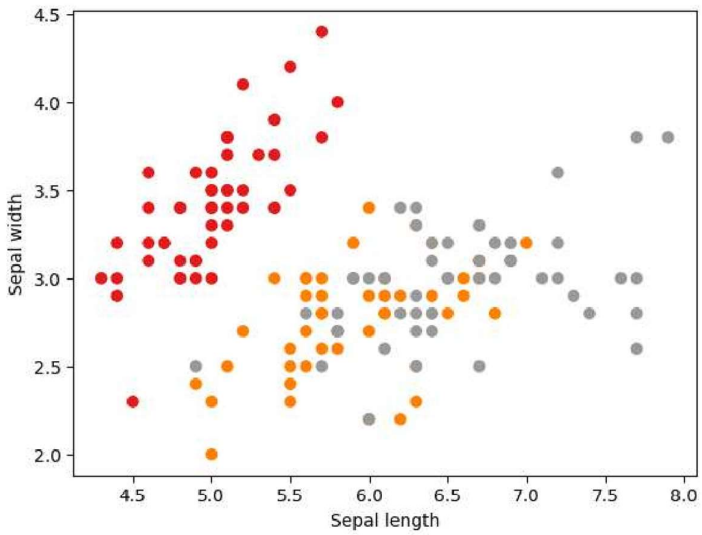
Train the SVM models and make predictions

```
for m in models:
    # Fit the SVM model on the training data
    m['model'].fit(X_train, y_train)

    # Make predictions on the testing data
    y_pred = m['model'].predict(X_test)

    # Calculate the accuracy of the model
    accuracy = np.mean(y_pred == y_test) * 100

    # Print the accuracy of the model
    print(f"{m['label']} accuracy: {accuracy}%")
```

```
 Linear kernel accuracy: 100.0%
 Polynomial kernel accuracy: 97.77777777777777%
 RBF kernel accuracy: 100.0%
 Sigmoid kernel accuracy: 22.22222222222222%
```

Plotting the first two features of the iris dataset

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```



Practical 10

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

Artificial Neural Network

The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

Error back-propagation algorithm theory

The error back-propagation algorithm consists of two big steps:

Feeding forward the input from the database to the input layer than to the hidden layers and finally to the output layer. Calculating the output error and feeding it backwards tuning the network's variables.

First of all, we need to load the database.

```
import numpy as np
from sklearn.model_selection import train_test_split

db = np.loadtxt("/content/drive/MyDrive/AIML/duke-breast-cancer.txt/duke-breast-cancer.txt
print("Database raw shape (%s,%s)" % np.shape(db))
```

```
Database raw shape (86,7130)
```

Now we have to shuffle it and then split it into training 90% and testing 10% so that the network can train itslef better. If needed you can also normalize the database.

```
np.random.shuffle(db)
y = db[:, 0]
x = np.delete(db, [0], axis=1)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1)
print(np.shape(x_train),np.shape(x_test))
```

```
(77, 7129) (9, 7129)
```

```
hidden_layer = np.zeros(72)
weights = np.random.random((len(x[0]), 72))
output_layer = np.zeros(2)
hidden_weights = np.random.random((72, 2))
```

Sum Function

s_i is the sum for [i]th perceptron from the layer.

```
def sum_function(weights, index_locked_col, x):
    result = 0
    for i in range(0, len(x)):
        result += x[i] * weights[i][index_locked_col]
    return result
```

Activation function

g(s_i) is the activation for the [i]th perceptron from the layer.

```
def activate_layer(layer, weights, x):
    for i in range(0, len(layer)):
        layer[i] = 1.7159 * np.tanh(2.0 * sum_function(weights, i, x) / 3.0)
```

Softmax Function

The softmax function, or normalized exponential function, is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector σ (z) of real values in the range (0, 1)that add up to 1.

```
def soft_max(layer):
    soft_max_output_layer = np.zeros(len(layer))
    for i in range(0, len(layer)):
        denominator = 0
        for j in range(0, len(layer)):
            denominator += np.exp(layer[j] - np.max(layer))
        soft_max_output_layer[i] = np.exp(layer[i] - np.max(layer)) / denominator
    return soft_max_output_layer


def recalculate_weights(learning_rate, weights, gradient, activation):
    for i in range(0, len(weights)):
```

```
            for j in range(0, len(weights[i])):
                weights[i][j] = (learning_rate * gradient[j] * activation[i]) + weights[i][j]
```

Recalculate weights function

Here we tune the network weights and hidden weights matrix. We are going to use this inside the back propagation function.

Back-propagation function

In this function we find out the output layer gradient and the hidden layer gradient to recalculate the network weights. Output gradient formula

```
def back_propagation(hidden_layer, output_layer, one_hot_encoding, learning_rate, x):
    output_derivative = np.zeros(2)
    output_gradient = np.zeros(2)
    for i in range(0, len(output_layer)):
        output_derivative[i] = (1.0 - output_layer[i]) * output_layer[i]
    for i in range(0, len(output_layer)):
        output_gradient[i] = output_derivative[i] * (one_hot_encoding[i] - output_layer[i]
    hidden_derivative = np.zeros(72)
    hidden_gradient = np.zeros(72)
    for i in range(0, len(hidden_layer)):
        hidden_derivative[i] = (1.0 - hidden_layer[i]) * (1.0 + hidden_layer[i])
    for i in range(0, len(hidden_layer)):
        sum_ = 0
        for j in range(0, len(output_gradient)):
            sum_ += output_gradient[j] * hidden_weights[i][j]
        hidden_gradient[i] = sum_ * hidden_derivative[i]
    recalculate_weights(learning_rate, hidden_weights, output_gradient, hidden_layer)
    recalculate_weights(learning_rate, weights, hidden_gradient, x)
```

Next we can one hot encode our output and start training our network iterative.

```
one_hot_encoding = np.zeros((2,2))
for i in range(0, len(one_hot_encoding)):
    one_hot_encoding[i][i] = 1
training_correct_answers = 0
for i in range(0, len(x_train)):
    activate_layer(hidden_layer, weights, x_train[i])
    activate_layer(output_layer, hidden_weights, hidden_layer)
    output_layer = soft_max(output_layer)
    training_correct_answers += 1 if y_train[i] == np.argmax(output_layer) else 0
    back_propagation(hidden_layer, output_layer, one_hot_encoding[int(y_train[i])], -1, x_
print("MLP Correct answers while learning: %s / %s (Accuracy = %s) on %s database." % (tra
                                                                                       tra
```

```
MLP Correct answers while learning: 47 / 77 (Accuracy = 0.6103896103896104) on Duke b
```

The accuracy of the test depends on the random generated weight's matrix and the learning rate. Using different learning rates and weight's will generate a different accuracy

```
testing_correct_answers = 0
for i in range(0, len(x_test)):
    activate_layer(hidden_layer, weights, x_test[i])
    activate_layer(output_layer, hidden_weights, hidden_layer)
    output_layer = soft_max(output_layer)
    testing_correct_answers += 1 if y_test[i] == np.argmax(output_layer) else 0
print("MLP Correct answers while testing: %s / %s (Accuracy = %s) on %s database" % (testi
                                                                                      testi
```

```
MLP Correct answers while testing: 9 / 9 (Accuracy = 1.0) on Duke breast cancer datal
```

Practical 11

Aim: Write a Program to implement K-Means clustering Algorithm.

Import required libraries

```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
```

```
df = pd.read_csv("/content/drive/MyDrive/AIML/Iris.csv")
df
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

```
df.head(n=10)
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
5	6	5.4	3.9	1.7	0.4	Iris-setosa
6	7	4.6	3.4	1.4	0.3	Iris-setosa
7	8	5.0	3.4	1.5	0.2	Iris-setosa
8	9	4.4	2.9	1.4	0.2	Iris-setosa
9	10	4.9	3.1	1.5	0.1	Iris-setosa

```
#finding different class labels
np.unique(df['Species'])
```

```
array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype=object)
```

```
#just for understanding, donot include in practical
df.shape
```

```
(150, 6)
```

```
df.info()
```

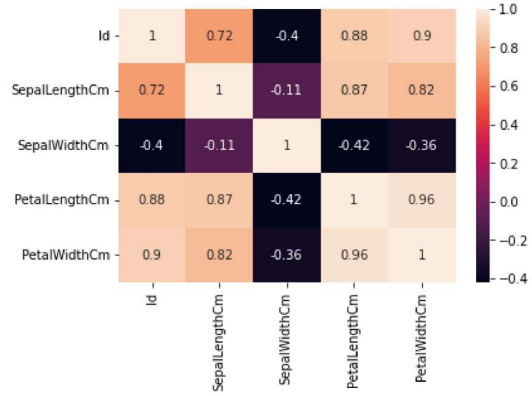
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

Heat map

A heatmap is a graphical representation where individual values of a matrix are represented as colors. A heatmap is very useful in visualizing the concentration of values between two dimensions of a matrix. This helps in finding patterns and gives a perspective of depth.

```
#finding correlation of features
correl=df.corr()
sns.heatmap(correl,annot=True)
```

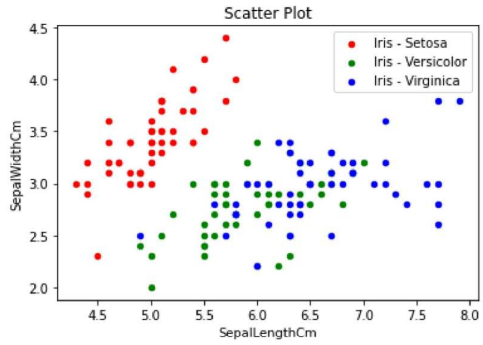
```
<AxesSubplot:>
```



Generate Scatter plot for Setosa, Versicolor, Virginca with x = SepalLengthCm and y = SepalWidthCm

```
ax = df[df.Species=='Iris-setosa'].plot.scatter(x='SepalLengthCm', y='SepalWidthCm',
                                                color='red', label='Iris - Setosa')
df[df.Species=='Iris-versicolor'].plot.scatter(x='SepalLengthCm', y='SepalWidthCm',
                                               color='green', label='Iris - Versicolor', ax=ax)
df[df.Species=='Iris-virginica'].plot.scatter(x='SepalLengthCm', y='SepalWidthCm',
                                              color='blue', label='Iris - Virginica', ax=ax)
ax.set_title("Scatter Plot")
```

```
Text(0.5, 1.0, 'Scatter Plot')
```



Check for null values in iris dataset

```
#checking for Null values
df.isnull().sum()
```

```
Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
```

```
Species          0
dtype: int64
```

Label Encoding - for encoding categorical features into numerical ones

```
#Label Encoding - for encoding categorical features into numerical ones
encoder = LabelEncoder()
df['Species'] = encoder.fit_transform(df['Species'])
df
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	0
1	2	4.9	3.0	1.4	0.2	0
2	3	4.7	3.2	1.3	0.2	0
3	4	4.6	3.1	1.5	0.2	0
4	5	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	2
146	147	6.3	2.5	5.0	1.9	2
147	148	6.5	3.0	5.2	2.0	2
148	149	6.2	3.4	5.4	2.3	2
149	150	5.9	3.0	5.1	1.8	2

150 rows × 6 columns

The values assign to class are

0 - Setosa

1 - Versicolor

2 - Virginica

```
#finding different class labels
np.unique(df['Species'])
```

```
array([0, 1, 2])
```

```
#DROPPING ID
df= df.drop(['Id'], axis = 1)
df.shape
```

```
(150, 5)
```

Coverting dataframe to np array and split the dataset

```
#converting dataframe to np array
data = df.values

X=data [:, 0:5]
Y= data [: , -1]

print(X.shape)
print(Y.shape)

#train-test split = 3:1

train_x = X[: 112, ]
train_y = Y[:112, ]

test_x = X[112:150, ]
test_y = Y[112:150, ]

print(train_x.shape)
print(train_y.shape)
print(test_x.shape)
print(test_y.shape)
```

```
(150, 5)
(150,)
(112, 5)
```

```
(38, 5)
(38,)
```

Using kmeans find training and testing dataset prediction

```
#KMeans

kmeans = KMeans(n_clusters=3)
kmeans.fit(train_x, train_y)

# training predictions
train_labels= kmeans.predict(train_x)

#testing predictions
test_labels = kmeans.predict(test_x)
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
```

```
#KMeans model accuracy

#training accuracy
print(accuracy_score(train_y, train_labels)*100)
#testing accuracy
print(accuracy_score(test_labels, test_y)*100)
```

```
99.10714285714286
94.73684210526315
```

```
#classification report for training set
print(classification_report(train_y, train_labels))
```

```
                  precision    recall  f1-score   support

             0.0       1.00      1.00      1.00        50
             1.0       0.98      1.00      0.99        50
             2.0       1.00      0.92      0.96        12

        accuracy                           0.99       112
       macro avg       0.99      0.97      0.98       112
    weighted avg       0.99      0.99      0.99       112
```

Practical 12

Aim: Implementation of any real time application using suitable machine learning technique.

Traffic Prediction

Importing Libraries

In [2]:
```
# Importing Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
import tensorflow
from statsmodels.tsa.stattools import adfuller
from sklearn.preprocessing import MinMaxScaler
from tensorflow import keras
from keras import callbacks
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense, LSTM, Dropout, GRU, Bidirectional
from tensorflow.keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error

import warnings
warnings.filterwarnings("ignore")
```

Loading Data

In [3]:
```
#Loading Data
data = pd.read_csv("../input/traffic-prediction-dataset/traffic.csv")
data.head()
```

Out[3]:

	DateTime	Junction	Vehicles	ID
0	2015-11-01 00:00:00	1	15	20151101001
1	2015-11-01 01:00:00	1	13	20151101011
2	2015-11-01 02:00:00	1	10	20151101021
3	2015-11-01 03:00:00	1	7	20151101031
4	2015-11-01 04:00:00	1	9	20151101041

About the data

This dataset is a collection of numbers of vehicles at four junctions at an hourly frequency. The CSV file provides four features:

- DateTime
- Junctions
- Vehicles
- ID

The sensors on each of these junctions were collecting data at different times, hence the traffic data from different time periods. Some of the junctions have provided limited or sparse data.

Data Exploration

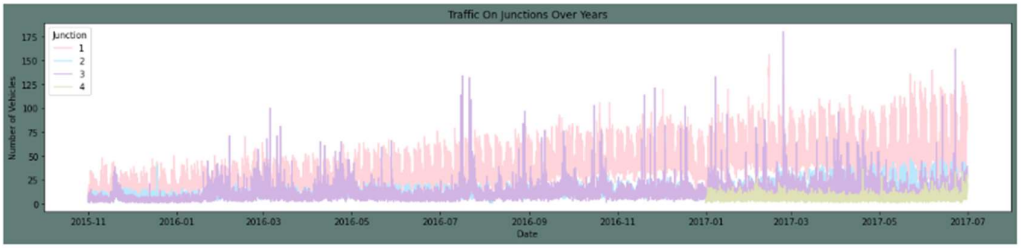
In [4]:
```
data["DateTime"]= pd.to_datetime(data["DateTime"])
data = data.drop(["ID"], axis=1) #dropping IDs
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48120 entries, 0 to 48119
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype         
---  ------    --------------  -----         
 0   DateTime  48120 non-null  datetime64[ns]
 1   Junction  48120 non-null  int64         
 2   Vehicles  48120 non-null  int64         
dtypes: datetime64[ns](1), int64(2)
memory usage: 1.1 MB
```

In [5]:
```
#df to be used for EDA
df=data.copy()
#Let's plot the Timeseries
colors = [ "#FFD4DB","#BBE7FE","#D3B5E5","#dfe2b6"]
plt.figure(figsize=(20,4),facecolor="#627D78")
Time_series=sns.lineplot(x=df['DateTime'],y="Vehicles",data=df, hue="Junction", palette=colors)
Time_series.set_title("Traffic On Junctions Over Years")
Time_series.set_ylabel("Number of Vehicles")
Time_series.set_xlabel("Date")
```

Out[5]:
```
Text(0.5, 0, 'Date')
```

Out[5]:
Text(0.5, 0, 'Date')



Noticeable information in the above plot:

- It can be seen here that the first junction is visibly having an upward trend.
- The data for the fourth junction is sparse starting only after 2017
- Seasonality is not evident from the above plot, So we must explore datetime composition to figure out more about it.

Feature Engineering

At this step, I am creating a few new features out of DateTime. Namely:

- Year
- Month
- Date in the given month
- Days of week
- Hour

In [6]:
```
#Exploring more features
df["Year"]= df['DateTime'].dt.year
df["Month"]= df['DateTime'].dt.month
df["Date_no"]= df['DateTime'].dt.day
df["Hour"]= df['DateTime'].dt.hour
df["Day"]= df.DateTime.dt.strftime("%A")
df.head()
```

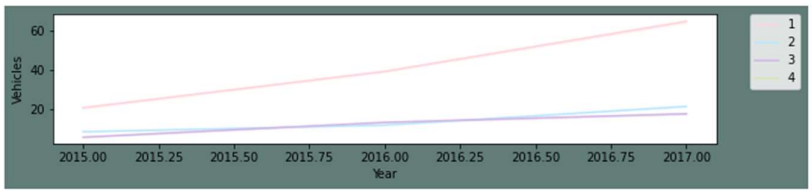
Out[6]:

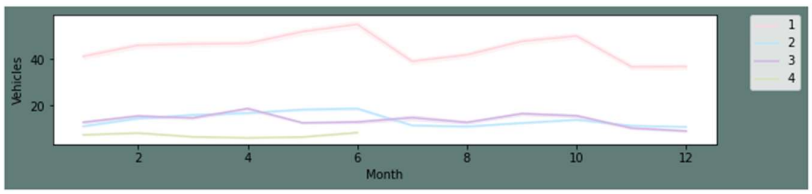
	DateTime	Junction	Vehicles	Year	Month	Date_no	Hour	Day
0	2015-11-01 00:00:00	1	15	2015	11	1	0	Sunday
1	2015-11-01 01:00:00	1	13	2015	11	1	1	Sunday
2	2015-11-01 02:00:00	1	10	2015	11	1	2	Sunday
3	2015-11-01 03:00:00	1	7	2015	11	1	3	Sunday
4	2015-11-01 04:00:00	1	9	2015	11	1	4	Sunday

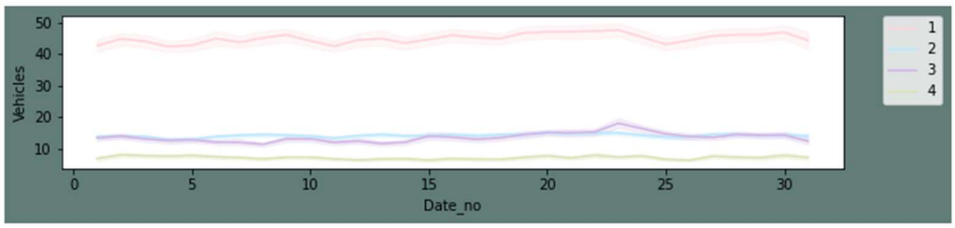
Exploratory Data Analysis

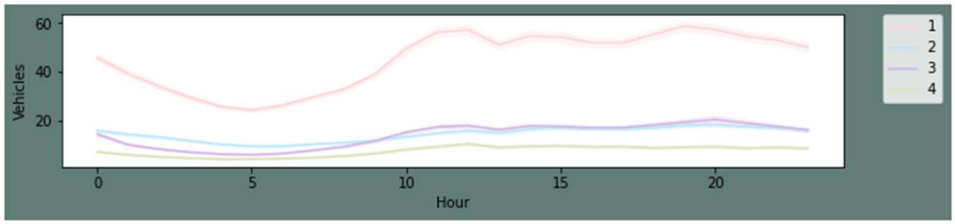
Plotting the newly created features

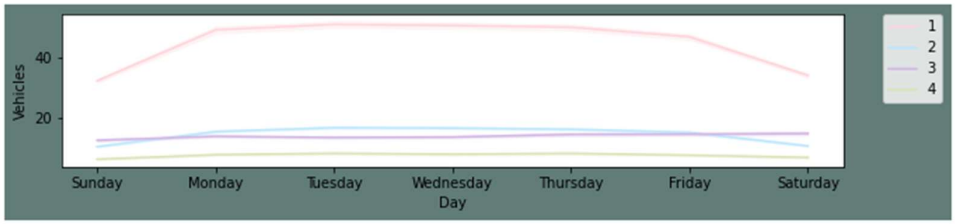
In [7]:
```
#Let's plot the Timeseries
new_features = [ "Year","Month", "Date_no", "Hour", "Day"]

for i in new_features:
    plt.figure(figsize=(10,2),facecolor="#627D78")
    ax=sns.lineplot(x=df[i],y="Vehicles",data=df, hue="Junction", palette=colors )
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```









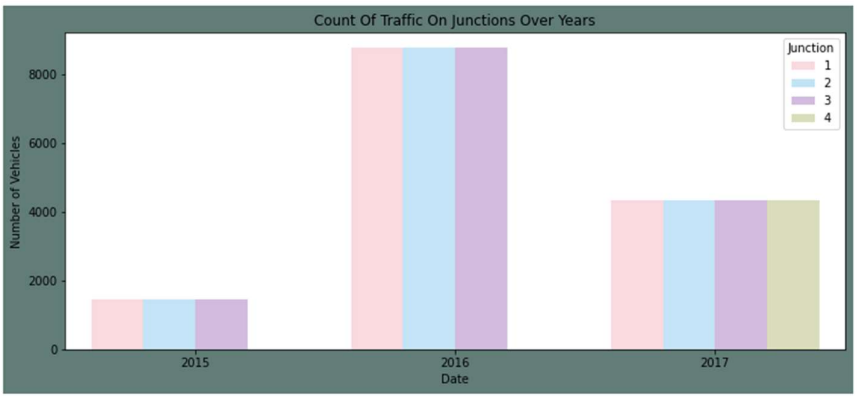


In [8]:

```
plt.figure(figsize=(12,5),facecolor="#627D78")
count = sns.countplot(data=df, x =df["Year"], hue="Junction", palette=colors)
count.set_title("Count Of Traffic On Junctions Over Years")
count.set_ylabel("Number of Vehicles")
count.set_xlabel("Date")
```

Out[8]:

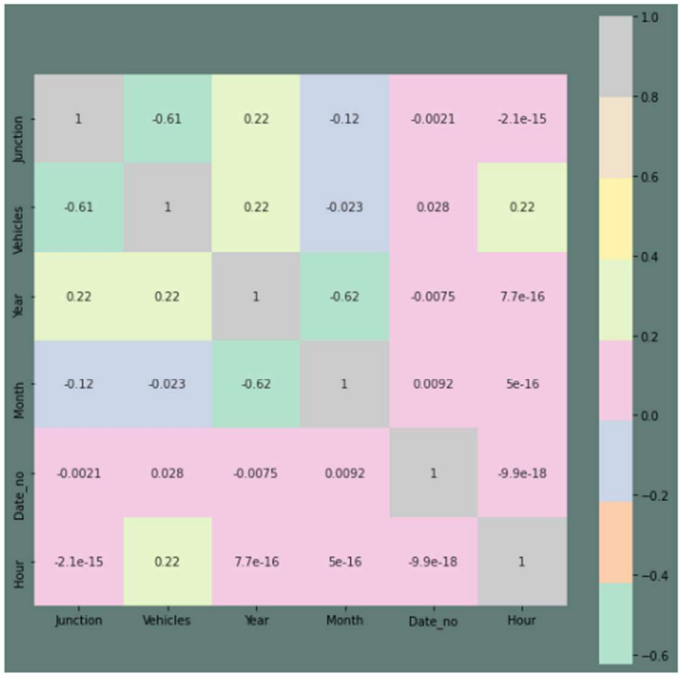
```
Text(0.5, 0, 'Date')
```



The count plot shows that there is an increase in the number of vehicles between 2015 and 2016. However, it is inconclusive to say the same about 2017 as we have limited data for 2017 ie till the 7th month.

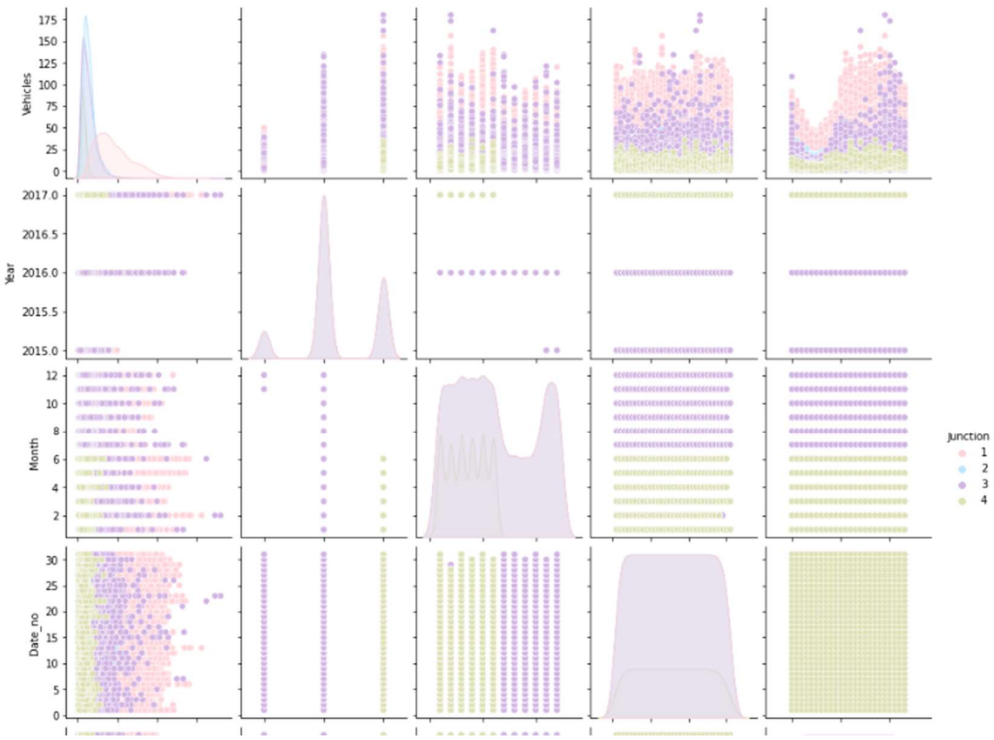
In [9]:
```
corrmat = df.corr()
plt.subplots(figsize=(10,10),facecolor="#627D78")
sns.heatmap(corrmat,cmap= "Pastel2",annot=True,square=True, )
```

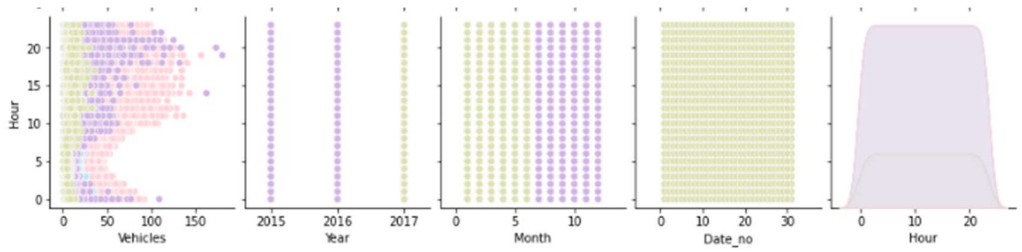
Out[9]:
```
<AxesSubplot:>
```



In [10]:
```
sns.pairplot(data=df, hue= "Junction",palette=colors)
```

Out[10]:
```
<seaborn.axisgrid.PairGrid at 0x7f3e9e1dbf50>
```





Data Transformation and Preprocessing

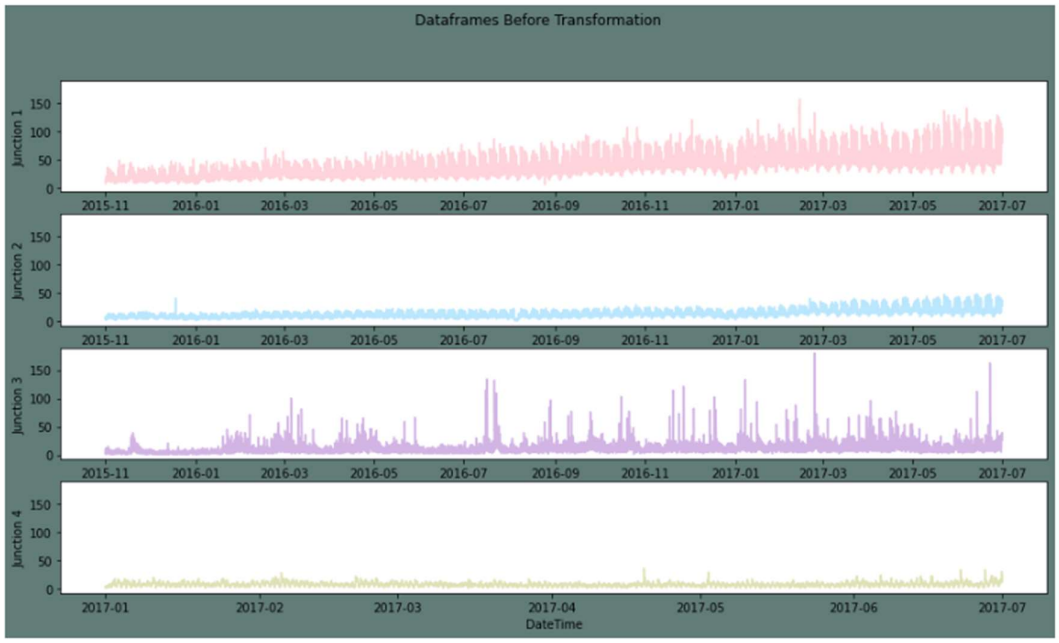
In [11]:
```
#Pivoting data fron junction
df_J = data.pivot(columns="Junction", index="DateTime")
df_J.describe()
```

Out[11]:

	Vehicles			
Junction	1	2	3	4
count	14592.000000	14592.000000	14592.000000	4344.000000
mean	45.052906	14.253221	13.694010	7.251611
std	23.008345	7.401307	10.436005	3.521455
min	5.000000	1.000000	1.000000	1.000000
25%	27.000000	9.000000	7.000000	5.000000
50%	40.000000	13.000000	11.000000	7.000000
75%	59.000000	17.000000	18.000000	9.000000
max	156.000000	48.000000	180.000000	36.000000

In [12]:
```
#Creating new sets
df_1 = df_J[[('Vehicles', 1)]]
df_2 = df_J[[('Vehicles', 2)]]
df_3 = df_J[[('Vehicles', 3)]]
df_4 = df_J[[('Vehicles', 4)]]
df_4 = df_4.dropna() #Junction 4 has limited data only for a few months

#Dropping level one in dfs's index as it is a multi index data frame
list_dfs = [df_1, df_2, df_3, df_4]
for i in list_dfs:
    i.columns= i.columns.droplevel(level=1)

#Function to plot comparitive plots of dataframes
def Sub_Plots4(df_1, df_2,df_3,df_4,title):
    fig, axes = plt.subplots(4, 1, figsize=(15, 8),facecolor="#627D78", sharey=True)
    fig.suptitle(title)
    #J1
    pl_1=sns.lineplot(ax=axes[0],data=df_1,color=colors[0])
    #pl_1=plt.ylabel()
    axes[0].set(ylabel ="Junction 1")
    #J2
    pl_2=sns.lineplot(ax=axes[1],data=df_2,color=colors[1])
    axes[1].set(ylabel ="Junction 2")
    #J3
    pl_3=sns.lineplot(ax=axes[2],data=df_3,color=colors[2])
    axes[2].set(ylabel ="Junction 3")
    #J4
    pl_4=sns.lineplot(ax=axes[3],data=df_4,color=colors[3])
    axes[3].set(ylabel ="Junction 4")
```

```
#Plotting the dataframe to check for stationarity
Sub_Plots4(df_1.Vehicles, df_2.Vehicles,df_3.Vehicles,df_4.Vehicles,"Dataframes Before Transformatio
n")
```



A time series is stationary if it does not have a trend or seasonality. However, in the EDA, we saw a weekly seasonality and an upwards trend over the years. In the above plot, it is again established that Junctions one and two have an upward trend. If we limit the span we will be able to further see the weekly seasonality. I will be spairing that step at this point and moving on with the respective transforms on datasets.

Steps for Transforming:

- Normalizing
- Differencing

In [13]:

```
# Normalize Function
def Normalize(df,col):
    average = df[col].mean()
    stdev = df[col].std()
    df_normalized = (df[col] - average) / stdev
    df_normalized = df_normalized.to_frame()
    return df_normalized, average, stdev

# Differencing Function
def Difference(df,col, interval):
    diff = []
    for i in range(interval, len(df)):
        value = df[col][i] - df[col][i - interval]
        diff.append(value)
    return diff
```

In accordance with the above observations, Differencing to eliminate the seasonality should be performed as follows:

- For Junction one, I will be taking a difference of weekly values.
- For junction two, The difference of consecutive days is a better choice
- For Junctions three and four, the difference of the hourly values will serve the purpose.

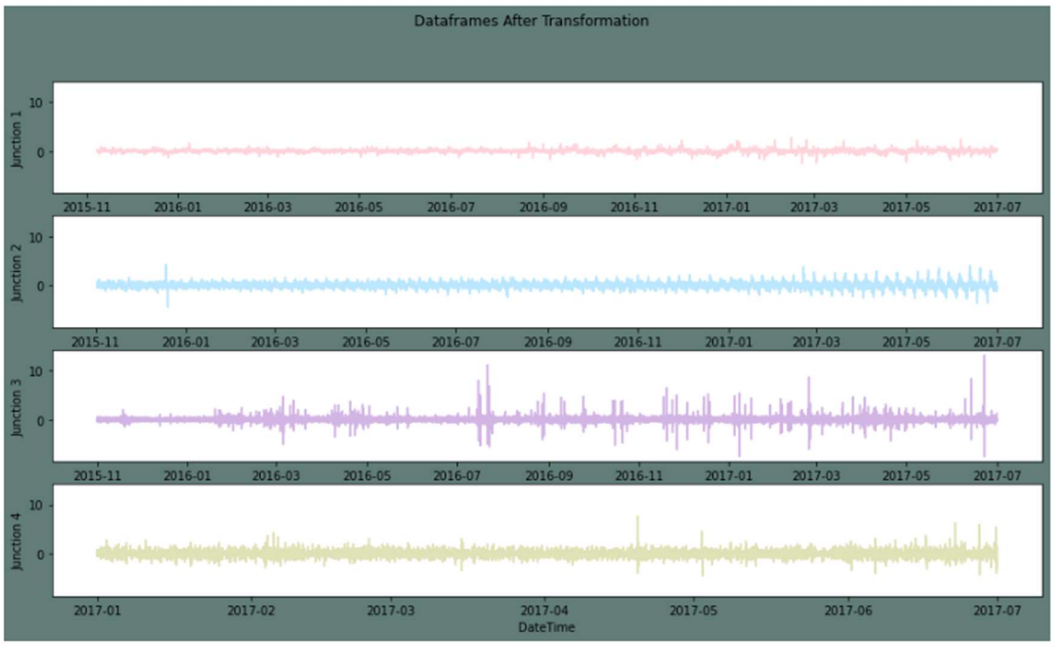
In [14]:

```
#Normalizing and Differencing to make the series stationary
df_N1, av_J1, std_J1 = Normalize(df_1, "Vehicles")
Diff_1 = Difference(df_N1, col="Vehicles", interval=(24*7)) #taking a week's diffrence
df_N1 = df_N1[24*7:]
df_N1.columns = ["Norm"]
df_N1["Diff"]= Diff_1

df_N2, av_J2, std_J2 = Normalize(df_2, "Vehicles")
Diff_2 = Difference(df_N2, col="Vehicles", interval=(24)) #taking a day's diffrence
df_N2 = df_N2[24:]
df_N2.columns = ["Norm"]
df_N2["Diff"]= Diff_2

df_N3, av_J3, std_J3 = Normalize(df_3, "Vehicles")
Diff_3 = Difference(df_N3, col="Vehicles", interval=1) #taking an hour's diffrence
df_N3 = df_N3[1:]
df_N3.columns = ["Norm"]
df_N3["Diff"]= Diff_3

df_N4, av_J4, std_J4 = Normalize(df_4, "Vehicles")
Diff_4 = Difference(df_N4, col="Vehicles", interval=1) #taking an hour's diffrence
df_N4 = df_N4[1:]
df_N4.columns = ["Norm"]
df_N4["Diff"]= Diff_4
```

Plots of Transformed Dataframe

In [15]:

```
Sub_Plots4(df_N1.Diff, df_N2.Diff,df_N3.Diff,df_N4.Diff,"Dataframes After Transformation")
```



In [16]:

```
#Stationary Check for the time series Augmented Dickey Fuller test
def Stationary_check(df):
    check = adfuller(df.dropna())
    print(f"ADF Statistic: {check[0]}")
    print(f"p-value: {check[1]}")
    print("Critical Values:")
    for key, value in check[4].items():
        print('\t%s: %.3f' % (key, value))
    if check[0] > check[4]["1%"]:
        print("Time Series is Non-Stationary")
    else:
        print("Time Series is Stationary")


#Checking if the series is stationary

List_df_ND = [ df_N1["Diff"], df_N2["Diff"], df_N3["Diff"], df_N4["Diff"]]
print("Checking the transformed series for stationarity:")
for i in List_df_ND:
    print("\n")
    Stationary_check(i)
```

```
Checking the transformed series for stationarity:


    ADF Statistic: -15.265303390415458
    p-value: 4.798539876396407e-28
    Critical Values:
            1%: -3.431
            5%: -2.862
            10%: -2.567
    Time Series is Stationary


    ADF Statistic: -21.795891026940097
    p-value: 0.0
    Critical Values:
            1%: -3.431
            5%: -2.862
            10%: -2.567
    Time Series is Stationary


    ADF Statistic: -28.001759908832778
    p-value: 0.0
    Critical Values:
            1%: -3.431
            5%: -2.862
            10%: -2.567
    Time Series is Stationary


    ADF Statistic: -17.97909256305229
    p-value: 2.778787532595421e-30
    Critical Values:
            1%: -3.432
            5%: -2.862
            10%: -2.567
```

In [17]:

```
#Differencing created some NA values as we took a weeks data into consideration while difrencing
df_J1 = df_N1["Diff"].dropna()
df_J1 = df_J1.to_frame()

df_J2 = df_N2["Diff"].dropna()
df_J2 = df_J2.to_frame()

df_J3 = df_N3["Diff"].dropna()
df_J3 = df_J3.to_frame()

df_J4 = df_N4["Diff"].dropna()
df_J4 = df_J4.to_frame()

#Splitting the dataset
def Split_data(df):
    training_size = int(len(df)*0.90)
    data_len = len(df)
    train, test = df[0:training_size],df[training_size:data_len]
    train, test = train.values.reshape(-1, 1), test.values.reshape(-1, 1)
    return train, test
#Splitting the training and test datasets
J1_train, J1_test = Split_data(df_J1)
J2_train, J2_test = Split_data(df_J2)
J3_train, J3_test = Split_data(df_J3)
J4_train, J4_test = Split_data(df_J4)
```

```
#Target and Feature
def TnF(df):
    end_len = len(df)
    X = []
    y = []
    steps = 32
    for i in range(steps, end_len):
        X.append(df[i - steps:i, 0])
        y.append(df[i, 0])
    X, y = np.array(X), np.array(y)
    return X ,y

#fixing the shape of X_test and X_train
def FeatureFixShape(train, test):
    train = np.reshape(train, (train.shape[0], train.shape[1], 1))
    test = np.reshape(test, (test.shape[0],test.shape[1],1))
    return train, test

#Assigning features and target
X_trainJ1, y_trainJ1 = TnF(J1_train)
X_testJ1, y_testJ1 = TnF(J1_test)
X_trainJ1, X_testJ1 = FeatureFixShape(X_trainJ1, X_testJ1)

X_trainJ2, y_trainJ2 = TnF(J2_train)
X_testJ2, y_testJ2 = TnF(J2_test)
X_trainJ2, X_testJ2 = FeatureFixShape(X_trainJ2, X_testJ2)

X_trainJ3, y_trainJ3 = TnF(J3_train)
X_testJ3, y_testJ3 = TnF(J3_test)
X_trainJ3, X_testJ3 = FeatureFixShape(X_trainJ3, X_testJ3)
```

```
X_trainJ4, y_trainJ4 = TnF(J4_train)
X_testJ4, y_testJ4 = TnF(J4_test)
X_trainJ4, X_testJ4 = FeatureFixShape(X_trainJ4, X_testJ4)
```

Model Building

In [18]:

```
#Model for the prediction
def GRU_model(X_Train, y_Train, X_Test):
    early_stopping = callbacks.EarlyStopping(min_delta=0.001,patience=10, restore_best_weights=True)
    #callback delta 0.01 may interrupt the learning, could eliminate this step, but meh!

    #The GRU model
    model = Sequential()
    model.add(GRU(units=150, return_sequences=True, input_shape=(X_Train.shape[1],1), activation='ta
nh'))
    model.add(Dropout(0.2))
    model.add(GRU(units=150, return_sequences=True, input_shape=(X_Train.shape[1],1), activation='ta
nh'))
    model.add(Dropout(0.2))
    model.add(GRU(units=50, return_sequences=True, input_shape=(X_Train.shape[1],1), activation='tan
h'))
    model.add(Dropout(0.2))
    model.add(GRU(units=50, return_sequences=True, input_shape=(X_Train.shape[1],1), activation='tan
h'))
    model.add(Dropout(0.2))
    #model.add(GRU(units=50, return_sequences=True,  input_shape=(X_Train.shape[1],1),activation='tan
h'))
    #model.add(Dropout(0.2))
    model.add(GRU(units=50, input_shape=(X_Train.shape[1],1), activation='tanh'))
    model.add(Dropout(0.2))
    model.add(Dense(units=1))
    #Compiling the model
    model.compile(optimizer=SGD(decay=1e-7, momentum=0.9),loss='mean_squared_error')
    model.fit(X_Train,y_Train, epochs=50, batch_size=150,callbacks=[early_stopping])
    pred_GRU= model.predict(X_Test)
    return pred_GRU
```

```
#To calculate the root mean squred error in predictions
def RMSE_Value(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))
    return rmse

#To plot the comparitive plot of targets and predictions
def PredictionsPlot(test,predicted,m):
    plt.figure(figsize=(12,5),facecolor="#627D78")
    plt.plot(test, color=colors[m],label="True Value",alpha=0.5 )
    plt.plot(predicted, color="#627D78",label="Predicted Values")
    plt.title("GRU Traffic Prediction Vs True values")
    plt.xlabel("DateTime")
    plt.ylabel("Number of Vehicles")
    plt.legend()
    plt.show()
```

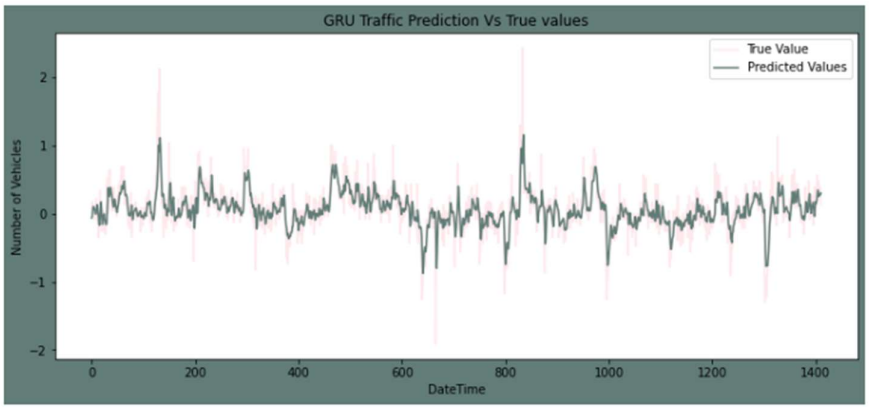
Fitting the model

Fitting the first junction and plotting the predictions and testset

In [19]:
```
#Predictions For First Junction
PredJ1 = GRU_model(X_trainJ1,y_trainJ1,X_testJ1)
```

In [20]:
```
#Results for J1
RMSE_J1=RMSE_Value(y_testJ1,PredJ1)
PredictionsPlot(y_testJ1,PredJ1,0)
```

```
The root mean squared error is 0.245881146563882.
```

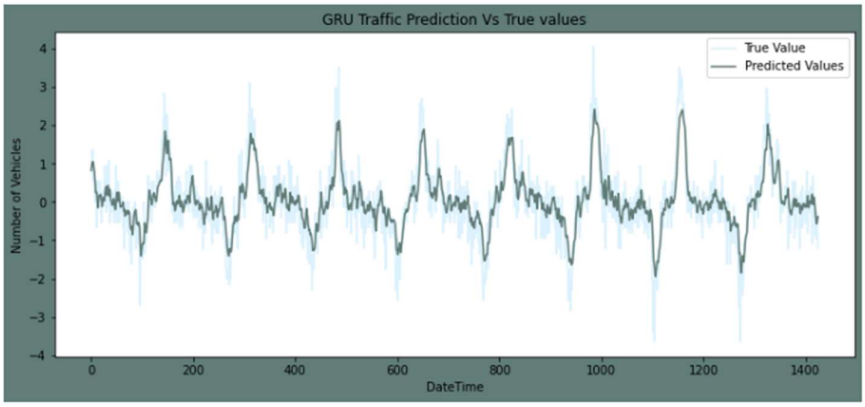


Fitting the second junction and plotting the predictions and testset

In [21]:
```
#Predictions For Second Junction
PredJ2 = GRU_model(X_trainJ2,y_trainJ2,X_testJ2)
```

In [22]:
```
#Results for J2
RMSE_J2=RMSE_Value(y_testJ2,PredJ2)
PredictionsPlot(y_testJ2,PredJ2,1)
```

```
The root mean squared error is 0.5585970393765944.
```

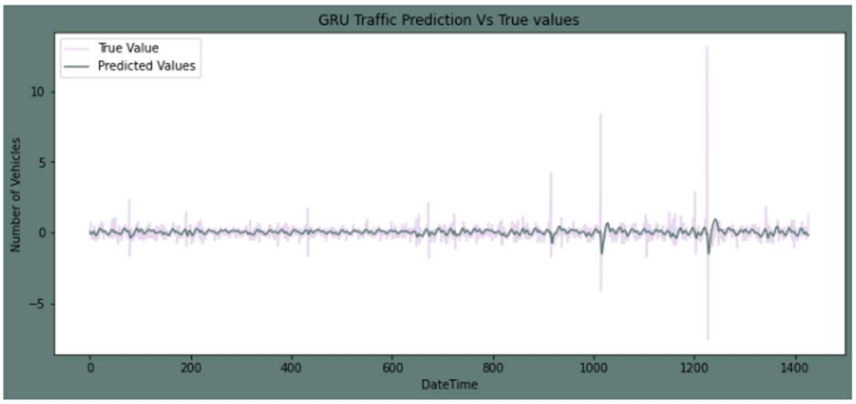


Fitting the third junction and plotting the predictions and testset

In [23]:
```
#Predictions For Third Junction
PredJ3 = GRU_model(X_trainJ3,y_trainJ3,X_testJ3)
```

In [24]:
```
#Results for J3
RMSE_J3=RMSE_Value(y_testJ3,PredJ3)
PredictionsPlot(y_testJ3,PredJ3,2)
```

```
The root mean squared error is 0.6061366783632264.
```

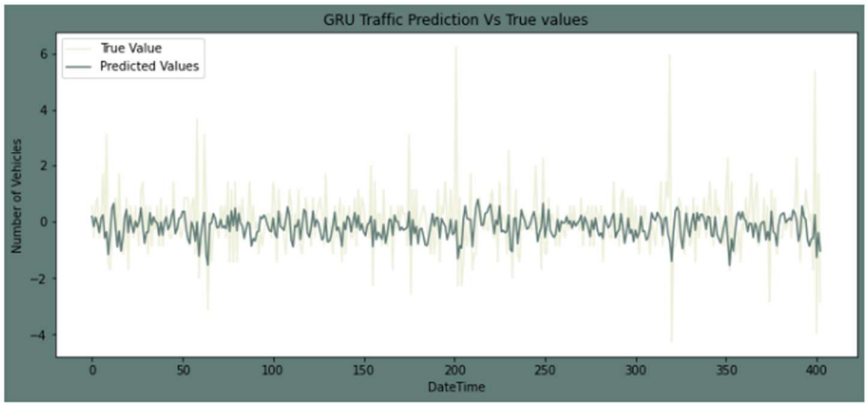


Fitting the fourth junction and plotting the predictions and testset

In [25]:
```
#Predictions For Forth Junction
PredJ4 = GRU_model(X_trainJ4,y_trainJ4,X_testJ4)
```

In [26]:
```
#Results for J4
RMSE_J4=RMSE_Value(y_testJ4,PredJ4)
PredictionsPlot(y_testJ4,PredJ4,3)
```

```
The root mean squared error is 1.0241982484501175.
```



The results of the model

Out[27]:

	Junction	RMSE
0	Junction1	0.245881
1	Junction2	0.558597
2	Junction3	0.606137
3	Junction4	1.024198

The Root Mean Square Error is quite a subjective marker for evaluating the performance. Thus, in this project, I am including the outcome plots as well.

Inversing the Transformation of Data

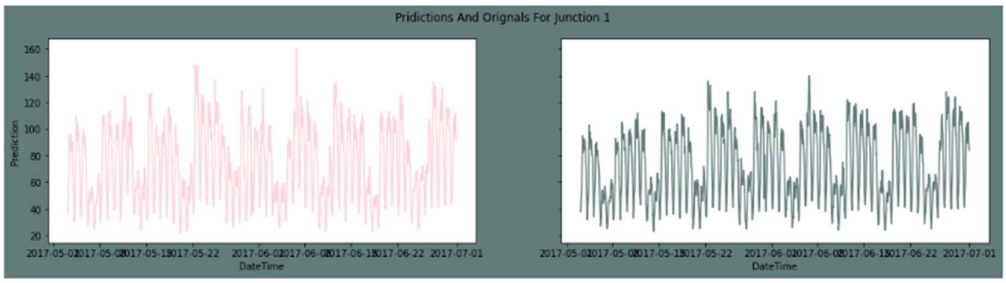
In [28]:

```
# Functions to inverse transforms and Plot comparitive plots
# invert differenced forecast
def inverse_difference(last_ob, value):
    inversed = value + last_ob
    return inversed
#Plotting the comparison
def Sub_Plots2(df_1, df_2,title,m):
    fig, axes = plt.subplots(1, 2, figsize=(18,4), sharey=True,facecolor="#627D78")
    fig.suptitle(title)

    pl_1=sns.lineplot(ax=axes[0],data=df_1,color=colors[m])
    axes[0].set(ylabel ="Prediction")

    pl_2=sns.lineplot(ax=axes[1],data=df_2["Vehicles"],color="#627D78")
    axes[1].set(ylabel ="Orignal")
```

the inverse transform on the first junction

In [29]:

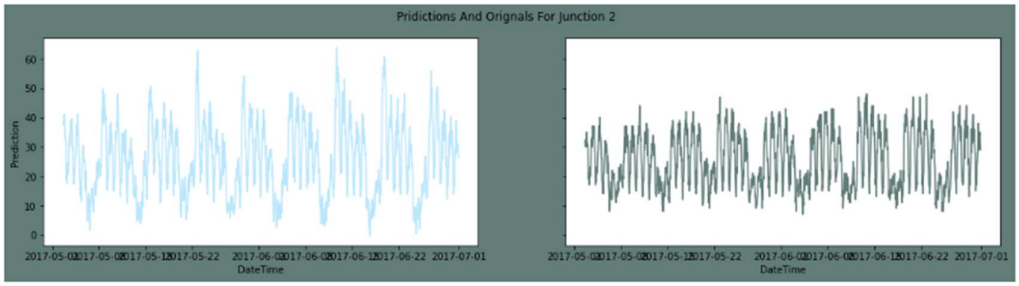
```
# invert the differenced forecast for Junction 1
recover1 = df_N1.Norm[-1412:-1].to_frame()
recover1["Pred"]= PredJ1
Transform_reverssed_J1 = inverse_difference(recover1.Norm, recover1.Pred).to_frame()
Transform_reverssed_J1.columns = ["Pred_Normed"]
#Invert the normalizeation J1
Final_J1_Pred = (Transform_reverssed_J1.values* std_J1) + av_J1
Transform_reverssed_J1["Pred_Final"] =Final_J1_Pred
#Plotting the Predictions with orignals
Sub_Plots2(Transform_reverssed_J1["Pred_Final"], df_1[-1412:-1],"Pridictions And Orignals For Junction 1", 0)
```



the inverse transform on the second junction

In [30]:

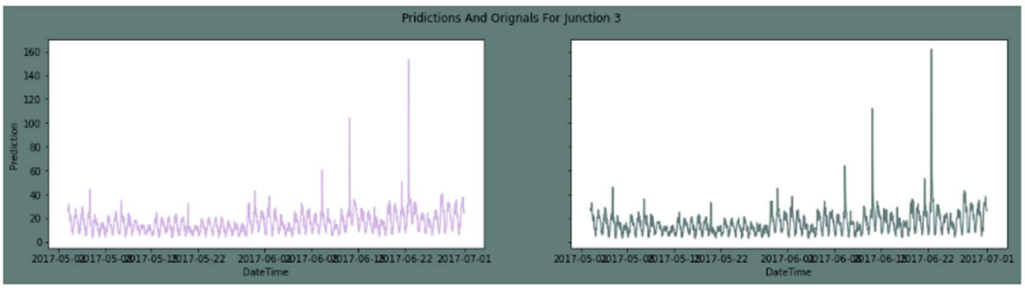
```
#Invert the differenced J2
recover2 = df_N2.Norm[-1426:-1].to_frame() #len as per the diff
recover2["Pred"]= PredJ2
Transform_reverssed_J2 = inverse_difference(recover2.Norm, recover2.Pred).to_frame()
Transform_reverssed_J2.columns = ["Pred_Normed"]
Final_J2_Pred = (Transform_reverssed_J2.values* std_J2) + av_J2
Transform_reverssed_J2["Pred_Final"] =Final_J2_Pred
#Plotting the Predictions with orignals
Sub_Plots2(Transform_reverssed_J2["Pred_Final"], df_2[-1426:-1],"Pridictions And Orignals For Junction 2", 1)
```



The inverse transform on the third junction

In [31]:

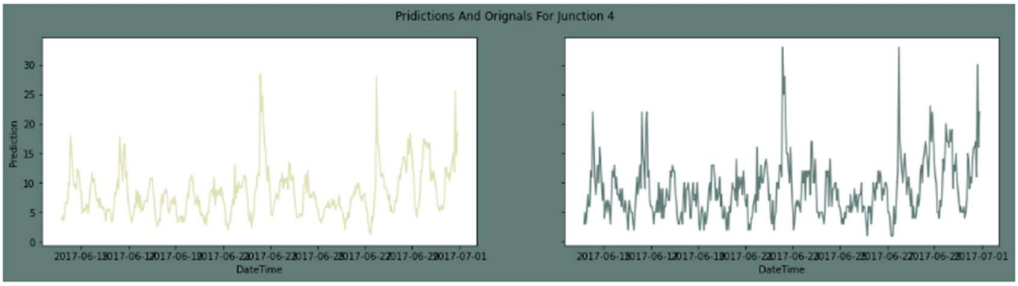
```
#Invert the differenced J3
recover3 = df_N3.Norm[-1429:-1].to_frame() #len as per the diff
recover3["Pred"]= PredJ3
Transform_reverssed_J3 = inverse_difference(recover3.Norm, recover3.Pred).to_frame()
Transform_reverssed_J3.columns = ["Pred_Normed"]
#Invert the normalizeation J3
Final_J3_Pred = (Transform_reverssed_J3.values* std_J3) + av_J3
Transform_reverssed_J3["Pred_Final"] =Final_J3_Pred
Sub_Plots2(Transform_reverssed_J3["Pred_Final"], df_3[-1429:-1],"Pridictions And Orignals For Junction 3", 2)
```



The inverse transform on the fourth junction

In [32]:

```
#Invert the differenced J4
recover4 = df_N4.Norm[-404:-1].to_frame()  #len as per the testset
recover4["Pred"]= PredJ4
Transform_reverssed_J4 = inverse_difference(recover4.Norm, recover4.Pred).to_frame()
Transform_reverssed_J4.columns = ["Pred_Normed"]
#Invert the normalizeation J4
Final_J4_Pred = (Transform_reverssed_J4.values* std_J4) + av_J4
Transform_reverssed_J4["Pred_Final"] =Final_J4_Pred
Sub_Plots2(Transform_reverssed_J4["Pred_Final"], df_4[-404:-1],"Pridictions And Orignals For Junctio
n 4", 3)
```



Summary

In this project, I trained a GRU Neural network to predicted the traffic on four junctions. I used a normalisation and differencing transform to achieve a stationary timeseries. As the Junctions varry in trends and seasonality, I took diffrent approach for each junction to make it stationary. I applyied the root mean squred error as the evaluation metric for the model. In addition to that I plotted the Predictions alongside the original test values. Take aways from the data analysis:

The Number of vehicles in Junction one is rising more rapidly compaired to junction two and three. The sparsity of data in juction four bars me from making any conclusion on the same.

The Junction one's traffic has a stronger weekly seasonality as well as hourly seasonality. Where as other junctions are significantly linear.

```
# define the CNN architecture
model = keras.Sequential(
    [
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=input_shape),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

We compile the model in Keras for two main reasons:

1. **Configuration**: The compile() method in Keras allows us to configure the model for training by setting the optimizer, loss function, and evaluation metrics. We specify the optimizer that will be used to update the weights of the model during training, the loss function that will be minimized, and the metrics that will be used to evaluate the performance of the model.
2. **Initialization**: The compile() method also initializes the model's parameters, including the weights and biases of the layers. This is necessary before training the model because the initial values of the parameters affect the model's ability to learn the patterns in the data.

```
# compile the model
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

Model Fitting

We do model fitting in Keras to train the neural network on a given dataset. Model fitting involves optimizing the model's parameters (i.e., the weights and biases of the layers) so that the model can learn to make accurate predictions on new data.

During model fitting, the neural network is trained on the training data by adjusting the model's parameters to minimize the difference between the predicted output and the true output.

```
# train the model
model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(X_test, y_test))
```

```
Epoch 1/10
11/11 [==============================] - 18s 1s/step - loss: 401.8176 - accuracy: 0.5681 - val_loss: 8.0222 - val_accuracy: 0.7027
Epoch 2/10
11/11 [==============================] - 14s 1s/step - loss: 1.9345 - accuracy: 0.7014 - val_loss: 0.5993 - val_accuracy: 0.6892
Epoch 3/10
11/11 [==============================] - 12s 1s/step - loss: 0.4008 - accuracy: 0.8232 - val_loss: 0.6177 - val_accuracy: 0.7365
Epoch 4/10
11/11 [==============================] - 12s 1s/step - loss: 0.3301 - accuracy: 0.8899 - val_loss: 0.6513 - val_accuracy: 0.7703
Epoch 5/10
11/11 [==============================] - 13s 1s/step - loss: 0.2107 - accuracy: 0.9536 - val_loss: 0.6761 - val_accuracy: 0.7432
Epoch 6/10
11/11 [==============================] - 13s 1s/step - loss: 0.1369 - accuracy: 0.9652 - val_loss: 0.7315 - val_accuracy: 0.7568
Epoch 7/10
11/11 [==============================] - 10s 951ms/step - loss: 0.0591 - accuracy: 0.9826 - val_loss: 0.8399 - val_accuracy: 0.7230
Epoch 8/10
11/11 [==============================] - 12s 1s/step - loss: 0.0646 - accuracy: 0.9855 - val_loss: 1.0543 - val_accuracy: 0.7297
Epoch 9/10
11/11 [==============================] - 13s 1s/step - loss: 0.0772 - accuracy: 0.9768 - val_loss: 0.7717 - val_accuracy: 0.7027
Epoch 10/10
11/11 [==============================] - 12s 1s/step - loss: 0.0634 - accuracy: 0.9855 - val_loss: 0.7432 - val_accuracy: 0.7095
<keras.callbacks.History at 0x7f4e70666790>
```

Model Evaluation

we evaluate the model in Keras to measure its performance on unseen data and estimate its ability to generalize to new data.

Model evaluation helps us to determine if the model is overfitting or underfitting, and to identify areas where the model can be improved.

```
# evaluate the model
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.7431941032409668
Test accuracy: 0.7094594836235046
```

```
# deploy the model for helmet detection
img = cv2.imread('/content/drive/MyDrive/AIML/Deploy_Model/BikesHelmets13.png')
img_arr = cv2.resize(img, (input_shape[0], input_shape[1]))
img_arr = np.array([img_arr])
# prediction = model.predict_classes(img_arr)
prediction = model.predict(img_arr)

# Get the predicted class index
predicted_class = np.argmax(prediction)

# Compare the predicted class index to the class index for helmets
if predicted_class == 0:
    print('Helmet detected')
else:
    print('No helmet detected')
```

```
1/1 [==============================] - 0s 131ms/step
Helmet detected
```