Chronic Kidney Disease Prediction

In this notebook, we will attempt to make machine learning models using **Chronic Kindey Disease Data**, which in return identifies whether a person may have chronic kidney disease or not.

Importing Libraries

In [1]:

```
# importing python libraries
%matplotlib inline
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('seaborn')

import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder
from sklearn.impute import KNNImputer
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

import keras
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

Load the Data

In [2]:

```
# Load the data
kidney_data = pd.read_csv('kidney_disease.csv')

print(kidney_data.shape)
print('Number of rows: %s' % str(kidney_data.shape[0]))
print('Number of columns: %s' % str(kidney_data.shape[1]))
```

```
(400, 26)
Number of rows: 400
Number of columns: 26
```

In [3]:

```
kidney_data.head(5)
```

Out[3]:

	id	age	bp	sg	al	su	rbc	pc	pcc	ba	...	pcv	wc	rc	htn	dm	cad	appet	pe	ane	classification
0	0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	...	44	7800	5.2	yes	yes	no	good	no	no	ckd
1	1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	...	38	6000	NaN	no	no	no	good	no	no	ckd
2	2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	...	31	7500	NaN	no	yes	no	poor	no	yes	ckd
3	3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	...	32	6700	3.9	yes	no	no	poor	yes	yes	ckd
4	4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	...	35	7300	4.6	no	no	no	good	no	no	ckd

5 rows × 26 columns

In [4]:

```
# get some statistical information
kidney_data.describe()
```

Out[4]:

	id	age	bp	sg	al	su	bgr	bu	sc	sod	pot
count	400.000000	391.000000	388.000000	353.000000	354.000000	351.000000	356.000000	381.000000	383.000000	313.000000	312.000000
mean	199.500000	51.483376	76.469072	1.017408	1.016949	0.450142	148.036517	57.425722	3.072454	137.528754	4.627244
std	115.614301	17.169714	13.683637	0.005717	1.352679	1.099191	79.281714	50.503006	5.741126	10.408752	3.193904
min	0.000000	2.000000	50.000000	1.005000	0.000000	0.000000	22.000000	1.500000	0.400000	4.500000	2.500000
25%	99.750000	42.000000	70.000000	1.010000	0.000000	0.000000	99.000000	27.000000	0.900000	135.000000	3.800000
50%	199.500000	55.000000	80.000000	1.020000	0.000000	0.000000	121.000000	42.000000	1.300000	138.000000	4.400000
75%	299.250000	64.500000	80.000000	1.020000	2.000000	0.000000	163.000000	66.000000	2.800000	142.000000	4.900000
max	399.000000	90.000000	180.000000	1.025000	5.000000	5.000000	490.000000	391.000000	76.000000	163.000000	47.000000

In [5]:

```
kidney_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 26 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   id              400 non-null    int64  
 1   age             391 non-null    float64
 2   bp              388 non-null    float64
 3   sg              353 non-null    float64
 4   al              354 non-null    float64
 5   su              351 non-null    float64
 6   rbc             248 non-null    object 
 7   pc              335 non-null    object 
 8   pcc             396 non-null    object 
 9   ba              396 non-null    object 
 10  bgr             356 non-null    float64
 11  bu              381 non-null    float64
 12  sc              383 non-null    float64
 13  sod             313 non-null    float64
 14  pot             312 non-null    float64
 15  hemo            348 non-null    float64
 16  pcv             330 non-null    object 
 17  wc              295 non-null    object 
 18  rc              270 non-null    object 
 19  htn             398 non-null    object 
 20  dm              398 non-null    object 
 21  cad             398 non-null    object 
 22  appet           399 non-null    object 
 23  pe              399 non-null    object 
 24  ane             399 non-null    object 
 25  classification  400 non-null    object 
dtypes: float64(11), int64(1), object(14)
memory usage: 81.4+ KB
```

In [6]:

```
# print dataset columns
kidney_data.columns
```

Out[6]:

```
Index(['id', 'age', 'bp', 'sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba', 'bgr',
       'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wc', 'rc', 'htn', 'dm', 'cad',
       'appet', 'pe', 'ane', 'classification'],
      dtype='object')
```

In [7]:

```
print(kidney_data['dm'])
```

```
0      yes
1       no
2      yes
3       no
4       no
      ...
395     no
396     no
397     no
398     no
399     no
Name: dm, Length: 400, dtype: object
```

Data Preprocessing

Our data contains missing, null and Nan values, and it also contains categorical values which need to be converted to numerical form. We will undergo data preprocessing steps in below section to make data ready for modelling. This step includes:

- Checking for null, missing and NaN values
- Removing null, missing and NaN values
- Getting the the ready in a form to apply machine learning

In [8]:

```
kidney_data.shape
```

Out[8]:

```
(400, 26)
```

We have a total of **400** samples and **26** features in data.

In [9]:

```
kidney_data = kidney_data.drop('id', axis=1)
```

In [10]:

```
kidney_data.head(3)
```

Out[10]:

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wc	rc	htn	dm	cad	appet	pe	ane	classification
0	48.0	80.0	1.02	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44	7800	5.2	yes	yes	no	good	no	no	ckd
1	7.0	50.0	1.02	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38	6000	NaN	no	no	no	good	no	no	ckd
2	62.0	80.0	1.01	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31	7500	NaN	no	yes	no	poor	no	yes	ckd

3 rows × 25 columns

In [11]:

```
# make column names easy to comprehend
'''
The column names are named in abbreviative manner. For making a comprehension about them, we will rename it in way that
it could be understood easily.
'''
kidney_data.columns = ['age', 'blood_pressure', 'specific_gravity',
                       'albumin', 'sugar', 'red_blood_cells', 'pus_cell',
                       'pus_cell_clumps', 'bacteria', 'blood_gulucose_random',
                       'blood_uera', 'serum_creatinine', 'sodium', 'potassium',
                       'haemoglobin', 'packed_cell_volume', 'white_blood_cell_count',
                       'red_blood_cell_count', 'hypertension', 'diabetes_mallitus',
                       'coronary_artery_disease', 'appetite', 'peda_edema',
                       'anaemia', 'class']
```

In [12]:

```
kidney_data.head(4)
```

Out[12]:

	age	blood_pressure	specific_gravity	albumin	sugar	red_blood_cells	pus_cell	pus_cell_clumps	bacteria	blood_gulucose_random	...
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...

4 rows × 25 columns

In [13]:

```
kidney_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 25 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   age                      391 non-null    float64
 1   blood_pressure           388 non-null    float64
 2   specific_gravity         353 non-null    float64
 3   albumin                  354 non-null    float64
 4   sugar                    351 non-null    float64
 5   red_blood_cells          248 non-null    object 
 6   pus_cell                 335 non-null    object 
 7   pus_cell_clumps          396 non-null    object 
 8   bacteria                 396 non-null    object 
 9   blood_gulucose_random    356 non-null    float64
 10  blood_uera               381 non-null    float64
 11  serum_creatinine         383 non-null    float64
 12  sodium                   313 non-null    float64
 13  potassium                312 non-null    float64
 14  haemoglobin              348 non-null    float64
 15  packed_cell_volume       330 non-null    object 
 16  white_blood_cell_count   295 non-null    object 
 17  red_blood_cell_count     270 non-null    object 
 18  hypertension             398 non-null    object 
 19  diabetes_mallitus        398 non-null    object 
 20  coronary_artery_disease  398 non-null    object 
 21  appetite                 399 non-null    object 
 22  peda_edema               399 non-null    object 
 23  anaemia                  399 non-null    object 
 24  class                    400 non-null    object 
dtypes: float64(11), object(14)
memory usage: 78.2+ KB
```

Converting Columns to Numerical Types

In [14]:

```
kidney_data['packed_cell_volume'] = pd.to_numeric(kidney_data['packed_cell_volume'],
                                                  errors='coerce')
kidney_data['white_blood_cell_count'] = pd.to_numeric(kidney_data['white_blood_cell_count'],
                                                      errors='coerce')
kidney_data['red_blood_cell_count'] = pd.to_numeric(kidney_data['white_blood_cell_count'],
                                                      errors='coerce')
```

In [15]:

```
print(kidney_data['white_blood_cell_count'].dtype)
print(kidney_data['packed_cell_volume'].dtype)
print(kidney_data['red_blood_cell_count'].dtype)
```

```
float64
float64
float64
```

Extract Column Types

In [16]:

```
# extract categorical & numerical columns
cat_cols = [i for i in kidney_data.columns if kidney_data[i].dtype=='object']
numeric_cols = [i for i in kidney_data.columns if kidney_data[i].dtype!='object']
```

In [17]:

```
print('Number of categorical columns: %s' % str(len(cat_cols)))
print('Number of numerical columns: %s' % str(len(numeric_cols)))
```

```
Number of categorical columns: 11
Number of numerical columns: 14
```

In [18]:

```
# check for missing values in each column
for c in cat_cols:
    print(f'{c} has {kidney_data[c].unique()} values\n')
```

```
red_blood_cells has [nan 'normal' 'abnormal'] values

pus_cell has ['normal' 'abnormal' nan] values

pus_cell_clumps has ['notpresent' 'present' nan] values

bacteria has ['notpresent' 'present' nan] values

hypertension has ['yes' 'no' nan] values

diabetes_mallitus has ['yes' 'no' ' yes' '\tno' '\tyes' nan] values

coronary_artery_disease has ['no' 'yes' '\tno' nan] values

appetite has ['good' 'poor' nan] values

peda_edema has ['no' 'yes' nan] values

anaemia has ['no' 'yes' nan] values

class has ['ckd' 'ckd\t' 'notckd'] values
```

In [19]:

```
# replacing incorrect values in columns
kidney_data['diabetes_mallitus'] = kidney_data['diabetes_mallitus'].replace(to_replace={'\tno': 'no', '\tyes': 'yes', 'yes':'yes
kidney_data['coronary_artery_disease'] = kidney_data['coronary_artery_disease'].replace(to_replace='\tno', value='no')
kidney_data['class'] = kidney_data['class'].replace(to_replace={'ckd\t': 'ckd', 'ckd': 'ckd', 'notckd': 'not ckd'})
```

In [20]:

```
kidney_data.head(5)
```

Out[20]:

	age	blood_pressure	specific_gravity	albumin	sugar	red_blood_cells	pus_cell	pus_cell_clumps	bacteria	blood_gulucose_random	...
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...

5 rows × 25 columns

In [21]:

```
# map labels as 0 or 1
# we will map as 0: no disease, 1: disease
kidney_data['class'] = kidney_data['class'].map({'ckd': 0, 'not ckd': 1})
```

In [22]:

```
kidney_data['class'].head(4)
```

Out[22]:

```
0    0
1    0
2    0
3    0
Name: class, dtype: int64
```

In [23]:

```
# check for null or missing values
kidney_data.isna().sum().sort_values(ascending=False)
```

Out[23]:

```
red_blood_cells            152
red_blood_cell_count       106
white_blood_cell_count     106
potassium                   88
sodium                      87
packed_cell_volume          71
pus_cell                    65
haemoglobin                 52
sugar                       49
specific_gravity            47
albumin                     46
blood_gulucose_random       44
blood_uera                  19
serum_creatinine            17
blood_pressure              12
age                          9
bacteria                     4
pus_cell_clumps              4
hypertension                 2
diabetes_mallitus            2
coronary_artery_disease      2
appetite                     1
peda_edema                   1
anaemia                      1
class                        0
dtype: int64
```

In [24]:

```
# checking for sum of null values in each column
kidney_data.isna().sum().to_numpy()
```

Out[24]:

```
array([  9,  12,  47,  46,  49, 152,  65,   4,   4,  44,  19,  17,  87,
        88,  52,  71, 106, 106,   2,   2,   2,   1,   1,   1,   0],
      dtype=int64)
```

Each column in our data contains missing, null or NaN values, which need to be cleaned. We will perform imputation technique to remove these null values.

Performing Imputation to Remove Missing Values

In [25]:

```
# making an imputation function
def random_imputation(x):
    random_sample = kidney_data[x].dropna().sample(kidney_data[x].isna().sum())
    random_sample.index = kidney_data[kidney_data[x].isnull()].index
    kidney_data.loc[kidney_data[x].isnull(), x] = random_sample

# define imputation mode
def imputation_mode(x):
    mode = kidney_data[x].mode()[0]
    kidney_data[x] = kidney_data[x].fillna(mode)
```

In [26]:

```
# apply function to columns
for c in numeric_cols:
    random_imputation(c)
```

In [27]:

```
# check for null or missing values
kidney_data[numeric_cols].isna().sum().sort_values(ascending=False)
```

Out[27]:

```
age                        0
blood_pressure             0
specific_gravity           0
albumin                    0
sugar                      0
blood_gulucose_random      0
blood_uera                 0
serum_creatinine           0
sodium                     0
potassium                  0
haemoglobin                0
packed_cell_volume         0
white_blood_cell_count     0
red_blood_cell_count       0
dtype: int64
```

In [28]:

```
# clean categorical columns
random_imputation('red_blood_cells')
random_imputation('pus_cell')

for c in cat_cols:
    imputation_mode(c)
```

In [29]:

```
# check for null or missing values
kidney_data[cat_cols].isna().sum().sort_values(ascending=False)
```

Out[29]:

```
red_blood_cells            0
pus_cell                   0
pus_cell_clumps            0
bacteria                   0
hypertension               0
diabetes_mallitus          0
coronary_artery_disease    0
appetite                   0
peda_edema                 0
anaemia                    0
class                      0
dtype: int64
```

We can now see that there is no missing, null or NaN value left in categorical features of dataset.

In [30]:

```
# check data again
kidney_data.head(5)
```

Out[30]:

	age	blood_pressure	specific_gravity	albumin	sugar	red_blood_cells	pus_cell	pus_cell_clumps	bacteria	blood_gulucose_random	...
0	48.0	80.0	1.020	1.0	0.0	normal	normal	notpresent	notpresent	121.0	...
1	7.0	50.0	1.020	4.0	0.0	abnormal	normal	notpresent	notpresent	100.0	...
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...

5 rows × 25 columns

In [31]:

```
# checking for sum of null values in each column
kidney_data.isna().sum().to_numpy()
```

Out[31]:

```
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], dtype=int64)
```

Label Encoding

For converting all categorical columns to number form, we will perform **Label Encoding** to each of the feature, so that each feature may have all numbered values.

In [32]:

```
# initialize label encoder
encoder = LabelEncoder()

# convert all categorical cols to labels
for column in cat_cols:
    kidney_data[column] = encoder.fit_transform(kidney_data[column])
```

In [33]:

```
kidney_data.head(5)
```

Out[33]:

	age	blood_pressure	specific_gravity	albumin	sugar	red_blood_cells	pus_cell	pus_cell_clumps	bacteria	blood_gulucose_random	...	pa
0	48.0	80.0	1.020	1.0	0.0	1	1	0	0	121.0	...	
1	7.0	50.0	1.020	4.0	0.0	0	1	0	0	100.0	...	
2	62.0	80.0	1.010	2.0	3.0	1	1	0	0	423.0	...	
3	48.0	70.0	1.005	4.0	0.0	1	0	1	0	117.0	...	
4	51.0	80.0	1.010	2.0	0.0	1	1	0	0	106.0	...	

5 rows × 25 columns

Conclusion

After performing a complete data preprocessing process to our data, we have come out with the result that:

- There is no null, missing or NaN valued feature left
- There is no feature containing categorical data
- Data is clean, preprocessed and is ready to be modeled

In [34]:

```
# kidney_data.to_csv('cleaned_kidney_data.csv')
```

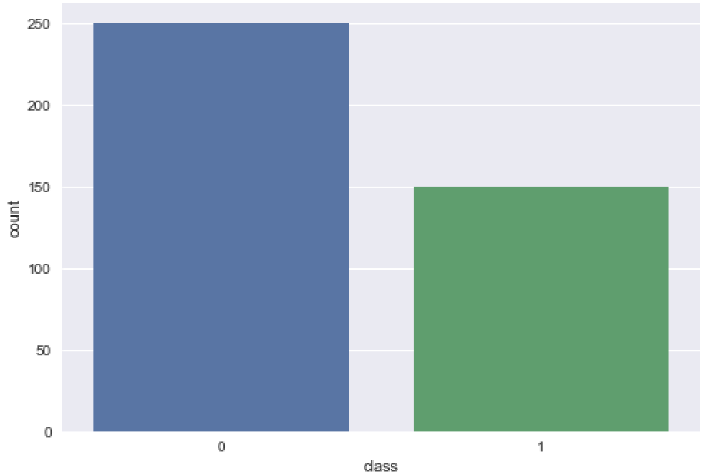
Data Visualizations

In [35]:

```
# make data visualizations
sns.countplot(x=kidney_data['class'])
```

Out[35]:

```
<AxesSubplot:xlabel='class', ylabel='count'>
```

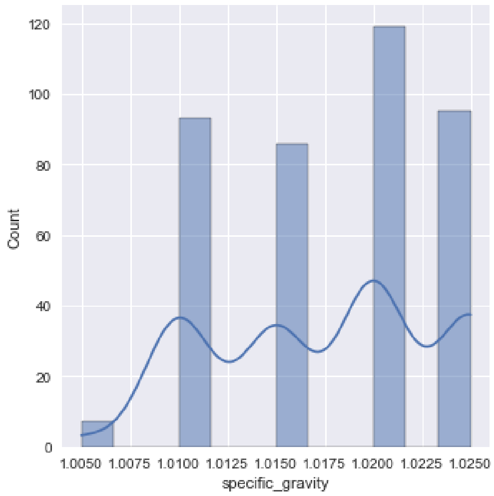


In [36]:

```
sns.displot(x=kidney_data['specific_gravity'], kde=True)
```

Out[36]:

```
<seaborn.axisgrid.FacetGrid at 0x22e50dfee50>
```



Set Predictors (X) and Target (Y) Values

In [37]:

```
X = kidney_data.drop('class', axis=1)
y = kidney_data['class']

X.shape, y.shape
```

Out[37]:

```
((400, 24), (400,))
```

Feature Scaling

For scaling the features to remove bias in data, we will use Min-Max Scaling as tool. The Min-Max Scaler performs such that:

- All values are scaled in a given range
- For our data, the range is set to (0, 1)

In [38]:

```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

X_scaled[3]
```

Out[38]:

```
array([0.52272727, 0.15384615, 0.        , 0.8       , 0.        ,
       1.        , 0.        , 1.        , 0.        , 0.20299145,
       0.13992298, 0.04497354, 0.67192429, 0.        , 0.55102041,
       0.51111111, 0.18595041, 0.18595041, 1.        , 0.5       ,
       0.        , 1.        , 1.        , 1.        ])
```

Data Splicing

For applying model, data needs to be split. We will use a ratio of **80-20** for our data such that:

- 80% data is kept in training set
- 20% data is kept in testing set

In [39]:

```
# make train/test splits

X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, y,
                                                    test_size=0.2, stratify=y)

print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)
```

```
(320, 24)
(80, 24)
(320,)
(80,)
```

Apply Models

We will use 6 (3 machine learning & 3 deep learning) models for classification task:

- Support Vector Machine (SVM)
- Random Forest Clasifier
- Decision Tree Classifier
- Artificial Neural Network (ANN)
- Multi-Layered Perceptron (MLP)

a. Applying Support Vector Machine (SVM)

In [40]:

```
svc = SVC()
svc.fit(X_train, Y_train)
```

Out[40]:

```
SVC()
```

In [41]:

```
# make predictions of test set
y_pred_svm = svc.predict(X_test)
y_pred_svm[:10]
```

Out[41]:

```
array([1, 1, 0, 1, 0, 0, 1, 0, 0, 0], dtype=int64)
```

In [42]:

```
# calculate accuracies on train and test datasets
print('Accuracy of Support Vector Machine on train data: %.5f' % svc.score(X_train, Y_train))
print('Accuracy of Support Vector Machine on test data: %.2f' % svc.score(X_test, Y_test))
```

```
Accuracy of Support Vector Machine on train data: 0.98438
Accuracy of Support Vector Machine on test data: 1.00
```

In [43]:

```
df_svm = pd.DataFrame({'Actual': Y_test, 'Predicted': y_pred_svm})
df_svm.head(6)
```

Out[43]:

	Actual	Predicted
265	1	1
379	1	1
135	0	0
259	1	1
39	0	0
120	0	0

In [49]:

```
# make confusion matrix
cm = confusion_matrix(Y_test, y_pred_svm)
cm
```

Out[49]:

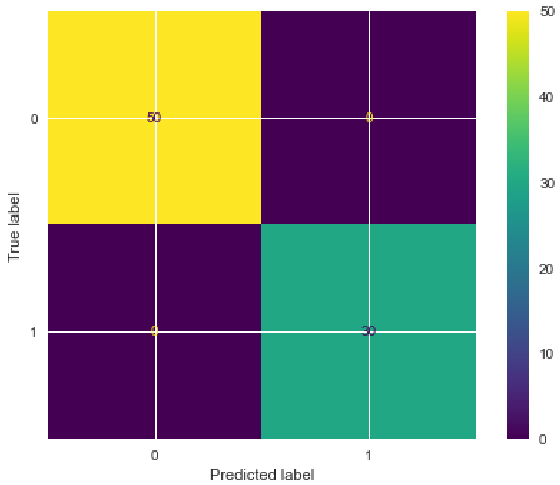
```
array([[50,  0],
       [ 0, 30]], dtype=int64)
```

In [54]:

```
disp = ConfusionMatrixDisplay(cm)
disp.plot()
```

Out[54]:

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x22e514eaac0>



In [55]:

```
print(classification_report(Y_test, y_pred_svm))
```

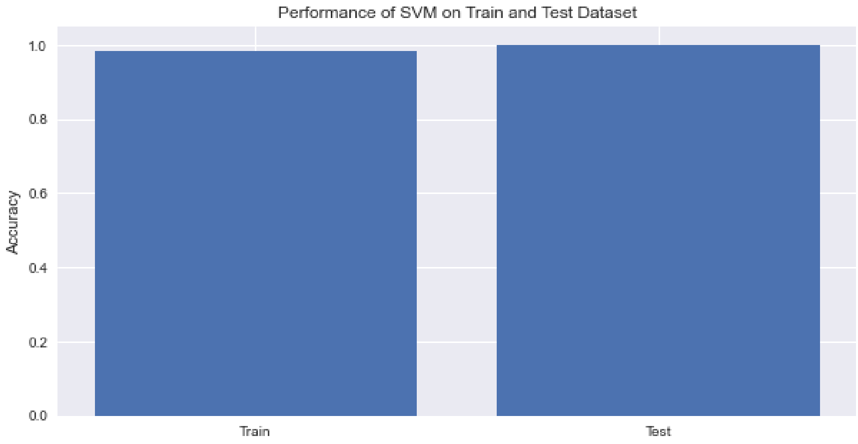
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        30

    accuracy                           1.00        80
   macro avg       1.00      1.00      1.00        80
weighted avg       1.00      1.00      1.00        80
```

In [57]:

```
plt.figure(figsize=(10, 5))
plt.ylabel('Accuracy')
plt.title('Performance of SVM on Train and Test Dataset')
clf = ['Train', 'Test']
acc = [svc.score(X_train, Y_train), accuracy_score(Y_test, y_pred_svm)]

plt.bar(clf, acc)
```

Out[57]:

```
<BarContainer object of 2 artists>
```



b. Decision Tree Classifier

In [58]:

```
dt = DecisionTreeClassifier(criterion='gini',
                            splitter='best')
# fit the model
dt.fit(X_train, Y_train)
```

Out[58]:

```
DecisionTreeClassifier()
```

In [59]:

```
# make predictions using dt
y_hat_dt = dt.predict(X_test, )
```

In [60]:

```
# calculate accuracies on train and test datasets
print('Accuracy of Decision Tree on train data: %.3f' % dt.score(X_train, Y_train))
print('Accuracy of Decision Tree on test data: %.2f' % dt.score(X_test, Y_test))
```

```
Accuracy of Decision Tree on train data: 1.000
Accuracy of Decision Tree on test data: 0.97
```

In [61]:

```
# make confusion matrix
cm = confusion_matrix(Y_test, y_hat_dt, labels=dt.classes_)
cm
```

Out[61]:

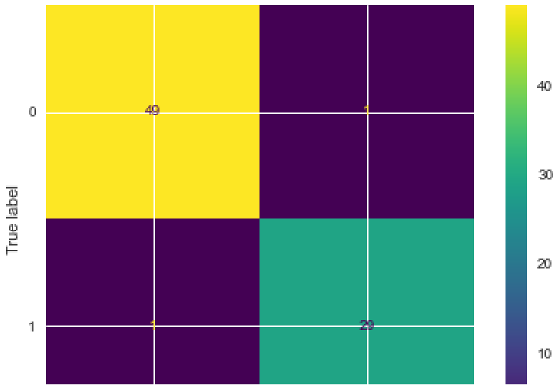
```
array([[49,  1],
       [ 1, 29]], dtype=int64)
```

In [62]:

```
matrix = ConfusionMatrixDisplay(cm, display_labels=dt.classes_)
matrix.plot()
```

Out[62]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x22e518a4bb0>
```



c. Random Forest Classifier

In [63]:

```
rf = RandomForestClassifier(n_estimators=100,
                            criterion='gini',
                            max_features='auto')
rf.fit(X_train, Y_train)
```

Out[63]:

```
RandomForestClassifier()
```

In [64]:

```
y_hat_rf = rf.predict(X_test)
```

In [65]:

```
# calculate accuracies on train and test datasets
print('Accuracy of Random Forest on train data: %.5f' % rf.score(X_train, Y_train))
print('Accuracy of Random Forest on test data: %.3f' % rf.score(X_test, Y_test))
```

```
Accuracy of Random Forest on train data: 1.00000
Accuracy of Random Forest on test data: 1.000
```

In [66]:

```
# make confusion matrix
cm = confusion_matrix(Y_test, y_hat_rf)
cm
```

Out[66]:

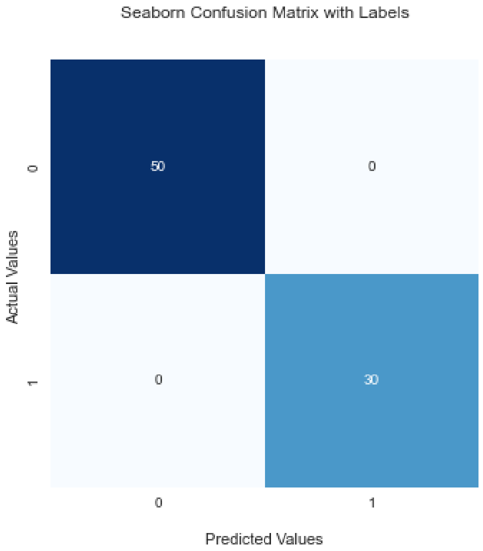
```
array([[50,  0],
       [ 0, 30]], dtype=int64)
```

In [67]:

```
ax = sns.heatmap(cm, square=True, annot=True,
                 cmap='Blues', fmt='d',
                 cbar=False)

ax.set_title('Seaborn Confusion Matrix with Labels\n\n')
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ')
```

Out[67]:

```
Text(125.71000000000001, 0.5, 'Actual Values ')
```



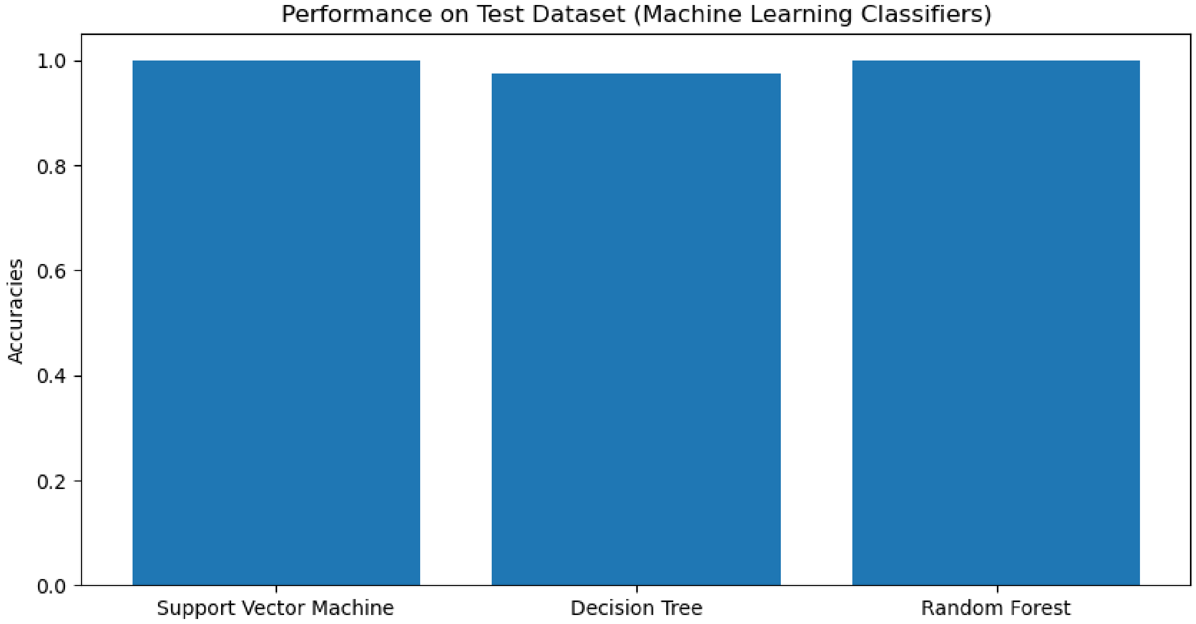
In [68]:

```
print(classification_report(Y_test, y_hat_rf))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        30

    accuracy                           1.00        80
   macro avg       1.00      1.00      1.00        80
weighted avg       1.00      1.00      1.00        80
```

In [121]:

```
# score = den.evaluate(X_test, Y_test, verbose=0)
from sklearn.metrics import log_loss

plt.figure(figsize=(10, 5))
plt.ylabel('Accuracies')
plt.title('Performance on Test Dataset (Machine Learning Classifiers)')
clf = ['Support Vector Machine', 'Decision Tree', 'Random Forest']
acc = [accuracy_score(Y_test, y_pred_svm), accuracy_score(Y_test, y_hat_dt), accuracy_score(Y_test, y_hat_rf)]

# loss = [log_loss(Y_test, y_pred_svm), log_loss(Y_test, y_hat_dt), log_loss(Y_test, y_hat_rf)]

pps = plt.bar(clf, acc)

for p in pps:
   height = p.get_height()
   ax.annotate('{}'.format(height),
      xy=(p.get_x() + p.get_width() / 2, height),
      xytext=(0, 3), # 3 points vertical offset
      textcoords="offset points",
      ha='center', va='bottom')

plt.show()
# plt.bar(clf, loss)
```



In [80]:

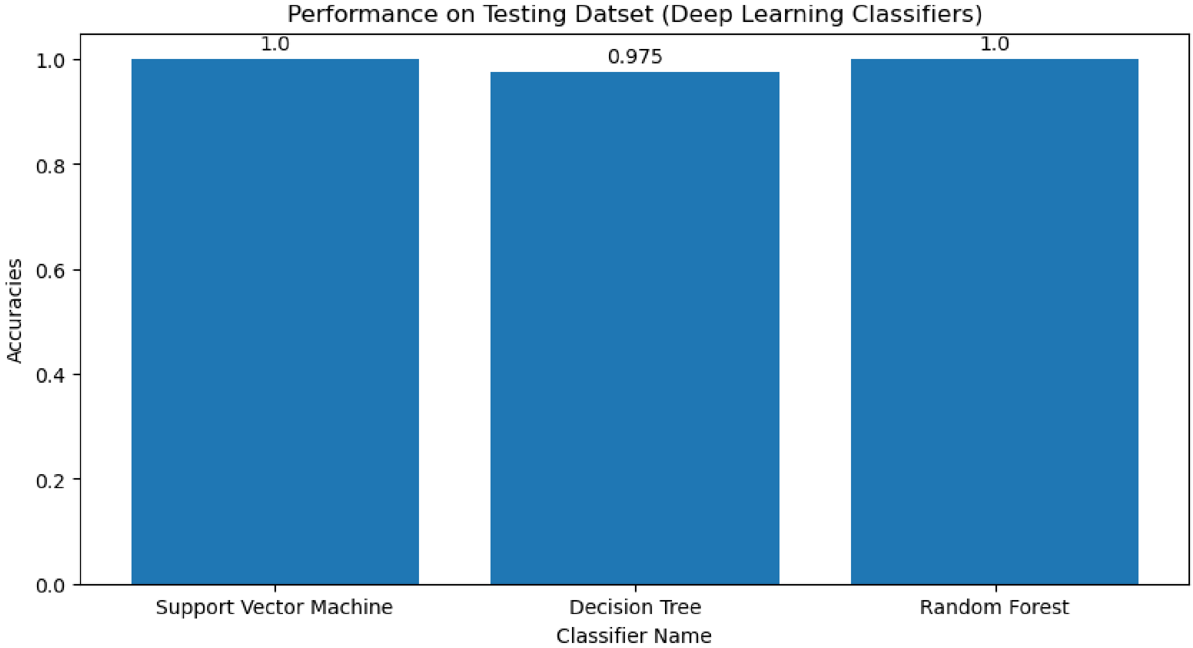
```
log_loss(Y_test, y_hat_rf)
```

Out[80]:

```
9.992007221626413e-16
```

In [122]:

```
fig, ax = plt.subplots(figsize=(10, 5))

ax.set_xlabel('Classifier Name')
ax.set_ylabel('Accuracies')

ax.set_title('Performance on Testing Datset (Deep Learning Classifiers)')
# ax.set_xticks(x)
# ax.set_xticklabels(years)


pps = ax.bar(clf, acc)
for p in pps:
   height = p.get_height()
   ax.annotate('{}'.format(height),
      xy=(p.get_x() + p.get_width() / 2, height),
      xytext=(0, 3), # 3 points vertical offset
      textcoords="offset points",
      ha='center', va='bottom')

plt.show()
```



d. Multi-Layered Perceptron (MLP)

In [84]:

```
# initialize MLP classifier
mlp = MLPClassifier(hidden_layer_sizes=(100,),
                    activation='relu', solver='sgd', max_iter=500)

# fit the model
mlp.fit(X_train, Y_train)
```

```
C:\Anaconda\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:692: ConvergenceWarning: Stochast
Optimizer: Maximum iterations (500) reached and the optimization hasn't converged yet.
  warnings.warn(
```

Out[84]:

```
MLPClassifier(max_iter=500, solver='sgd')
```

In [85]:

```
y_hat_mlp = mlp.predict(X_test)
```

In [86]:

```
# calculate accuracies on train and test datasets
print('Accuracy of MLP on train data: %.5f' % rf.score(X_train, Y_train))
print('Accuracy of MLP on test data: %.3f' % rf.score(X_test, Y_test))
```

```
Accuracy of MLP on train data: 1.00000
Accuracy of MLP on test data: 1.000
```

In [87]:

```
# make confusion matrix
cm = confusion_matrix(Y_test, y_hat_mlp)
cm
```

Out[87]:

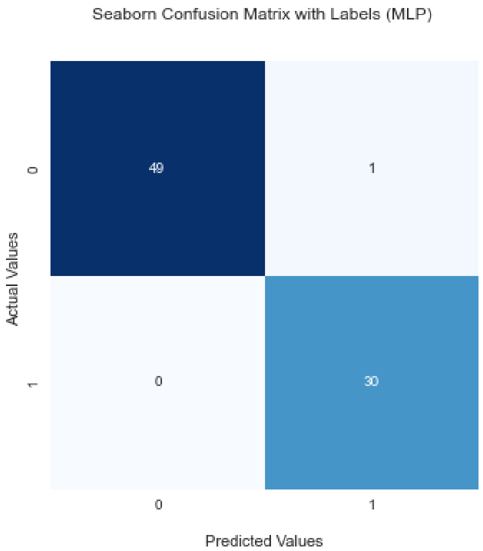
```
array([[49,  1],
       [ 0, 30]], dtype=int64)
```

In [88]:

```
ax = sns.heatmap(cm, square=True, annot=True,
                 cmap='Blues', fmt='d',
                 cbar=False)

ax.set_title('Seaborn Confusion Matrix with Labels (MLP) \n\n')
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ')
```

Out[88]:

```
Text(125.71000000000001, 0.5, 'Actual Values ')
```



In [89]:

```
print(classification_report(Y_test, y_hat_mlp))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.97      1.00      0.98        30

    accuracy                           0.99        80
   macro avg       0.98      0.99      0.99        80
weighted avg       0.99      0.99      0.99        80
```

e. Applying Deep Neural Network Network (DNN)

In [90]:

```
# make ann function
def deep_neural_network():
    model = Sequential()
    model.add(Dense(512, activation='relu'))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    return model

# call the function
dnn = deep_neural_network()
```

In [91]:

```
# compiling the model
opt = Adam(0.01)

dnn.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [92]:

```
# fit the model to data
history = dnn.fit(X_train, Y_train,
                    epochs=50, verbose=1, validation_split=0.2)
```

```
Epoch 1/50
8/8 [==============================] - 3s 99ms/step - loss: 0.3385 - accuracy: 0.7930 - val_loss: 0.0699 - val_acc
uracy: 0.9688
Epoch 2/50
8/8 [==============================] - 0s 11ms/step - loss: 0.2572 - accuracy: 0.9297 - val_loss: 0.2584 - val_acc
uracy: 0.9531
Epoch 3/50
8/8 [==============================] - 0s 12ms/step - loss: 0.2215 - accuracy: 0.9258 - val_loss: 0.1041 - val_acc
uracy: 0.9844
Epoch 4/50
8/8 [==============================] - 0s 12ms/step - loss: 0.1539 - accuracy: 0.9258 - val_loss: 0.0889 - val_acc
uracy: 0.9688
Epoch 5/50
8/8 [==============================] - 0s 12ms/step - loss: 0.1440 - accuracy: 0.9570 - val_loss: 0.0759 - val_acc
uracy: 0.9688
Epoch 6/50
8/8 [==============================] - 0s 32ms/step - loss: 0.0726 - accuracy: 0.9766 - val_loss: 0.0397 - val_acc
uracy: 0.9688
Epoch 7/50
8/8 [==============================] - 0s 32ms/step - loss: 0.0682 - accuracy: 0.9805 - val_loss: 0.0359 - val_acc
uracy: 0.9688
Epoch 8/50
8/8 [==============================] - 0s 17ms/step - loss: 0.0695 - accuracy: 0.9727 - val_loss: 0.1008 - val_acc
uracy: 0.9375
Epoch 9/50
8/8 [==============================] - 0s 18ms/step - loss: 0.0541 - accuracy: 0.9805 - val_loss: 0.0369 - val_acc
uracy: 0.9844
Epoch 10/50
8/8 [==============================] - 0s 21ms/step - loss: 0.0433 - accuracy: 0.9844 - val_loss: 0.0920 - val_acc
uracy: 0.9688
Epoch 11/50
8/8 [==============================] - 0s 12ms/step - loss: 0.0304 - accuracy: 0.9805 - val_loss: 0.0421 - val_acc
uracy: 0.9844
Epoch 12/50
8/8 [==============================] - 0s 17ms/step - loss: 0.0203 - accuracy: 0.9922 - val_loss: 0.0271 - val_acc
uracy: 0.9688
Epoch 13/50
8/8 [==============================] - 0s 14ms/step - loss: 0.0339 - accuracy: 0.9883 - val_loss: 0.0994 - val_acc
uracy: 0.9531
Epoch 14/50
8/8 [==============================] - 0s 19ms/step - loss: 0.0266 - accuracy: 0.9883 - val_loss: 0.0281 - val_acc
uracy: 0.9844
Epoch 15/50
8/8 [==============================] - 0s 37ms/step - loss: 0.0136 - accuracy: 0.9922 - val_loss: 0.0839 - val_acc
uracy: 0.9688
Epoch 16/50
8/8 [==============================] - 0s 18ms/step - loss: 0.0211 - accuracy: 0.9961 - val_loss: 0.0395 - val_acc
uracy: 0.9844
Epoch 17/50
8/8 [==============================] - 0s 19ms/step - loss: 0.0274 - accuracy: 0.9883 - val_loss: 0.0253 - val_acc
uracy: 0.9844
Epoch 18/50
8/8 [==============================] - 0s 14ms/step - loss: 0.0113 - accuracy: 0.9961 - val_loss: 0.1137 - val_acc
uracy: 0.9531
Epoch 19/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0214 - accuracy: 0.9922 - val_loss: 0.0300 - val_acc
uracy: 0.9844
Epoch 20/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0128 - accuracy: 0.9961 - val_loss: 0.0693 - val_acc
uracy: 0.9688
Epoch 21/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0190 - accuracy: 0.9922 - val_loss: 0.0738 - val_acc
uracy: 0.9688
Epoch 22/50
8/8 [==============================] - 0s 17ms/step - loss: 0.0369 - accuracy: 0.9844 - val_loss: 0.0402 - val_acc
uracy: 0.9844
Epoch 23/50
8/8 [==============================] - 0s 21ms/step - loss: 0.0124 - accuracy: 0.9961 - val_loss: 0.0260 - val_acc
uracy: 0.9844
Epoch 24/50
8/8 [==============================] - 0s 29ms/step - loss: 0.0164 - accuracy: 0.9883 - val_loss: 0.1892 - val_acc
uracy: 0.9375
Epoch 25/50
8/8 [==============================] - 0s 29ms/step - loss: 0.0624 - accuracy: 0.9688 - val_loss: 0.0260 - val_acc
uracy: 0.9844
Epoch 26/50
8/8 [==============================] - 0s 15ms/step - loss: 0.0427 - accuracy: 0.9844 - val_loss: 0.0483 - val_acc
uracy: 0.9844
Epoch 27/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0457 - accuracy: 0.9805 - val_loss: 0.0756 - val_acc
uracy: 0.9688
Epoch 28/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0176 - accuracy: 0.9922 - val_loss: 0.0799 - val_acc
uracy: 0.9688
Epoch 29/50
```

```
8/8 [==============================] - 0s 12ms/step - loss: 0.0070 - accuracy: 1.0000 - val_loss: 0.0372 - val_acc
uracy: 0.9688
Epoch 30/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0202 - accuracy: 0.9922 - val_loss: 0.0709 - val_acc
uracy: 0.9844
Epoch 31/50
8/8 [==============================] - 0s 12ms/step - loss: 0.0554 - accuracy: 0.9766 - val_loss: 0.2836 - val_acc
uracy: 0.9375
Epoch 32/50
8/8 [==============================] - 0s 14ms/step - loss: 0.0518 - accuracy: 0.9727 - val_loss: 0.0330 - val_acc
uracy: 0.9688
Epoch 33/50
8/8 [==============================] - 0s 46ms/step - loss: 0.0324 - accuracy: 0.9844 - val_loss: 0.0348 - val_acc
uracy: 0.9844
Epoch 34/50
8/8 [==============================] - 0s 35ms/step - loss: 0.0291 - accuracy: 0.9844 - val_loss: 0.0946 - val_acc
uracy: 0.9375
Epoch 35/50
8/8 [==============================] - 0s 14ms/step - loss: 0.0131 - accuracy: 0.9961 - val_loss: 0.0286 - val_acc
uracy: 0.9844
Epoch 36/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0080 - accuracy: 0.9961 - val_loss: 0.0441 - val_acc
uracy: 0.9688
Epoch 37/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0049 - accuracy: 0.9961 - val_loss: 0.0304 - val_acc
uracy: 0.9844
Epoch 38/50
8/8 [==============================] - 0s 12ms/step - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.0353 - val_acc
uracy: 0.9844
Epoch 39/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.0568 - val_acc
uracy: 0.9688
Epoch 40/50
8/8 [==============================] - 0s 17ms/step - loss: 0.0015 - accuracy: 1.0000 - val_loss: 0.0444 - val_acc
uracy: 0.9844
Epoch 41/50
8/8 [==============================] - 0s 13ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.0301 - val_acc
uracy: 0.9844
Epoch 42/50
8/8 [==============================] - 0s 63ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.1102 - val_acc
uracy: 0.9688
Epoch 43/50
8/8 [==============================] - 0s 15ms/step - loss: 0.0035 - accuracy: 0.9961 - val_loss: 0.1026 - val_acc
uracy: 0.9844
Epoch 44/50
8/8 [==============================] - 0s 16ms/step - loss: 0.0433 - accuracy: 0.9844 - val_loss: 0.0601 - val_acc
uracy: 0.9844
Epoch 45/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0800 - accuracy: 0.9844 - val_loss: 0.0515 - val_acc
uracy: 0.9688
Epoch 46/50
8/8 [==============================] - 0s 13ms/step - loss: 0.0253 - accuracy: 0.9961 - val_loss: 0.0333 - val_acc
uracy: 0.9844
Epoch 47/50
8/8 [==============================] - 0s 11ms/step - loss: 0.0132 - accuracy: 0.9922 - val_loss: 0.0498 - val_acc
uracy: 0.9688
Epoch 48/50
8/8 [==============================] - 0s 13ms/step - loss: 0.0036 - accuracy: 1.0000 - val_loss: 0.0481 - val_acc
uracy: 0.9688
Epoch 49/50
8/8 [==============================] - 0s 15ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.0480 - val_acc
uracy: 0.9688
Epoch 50/50
8/8 [==============================] - 0s 39ms/step - loss: 0.0197 - accuracy: 0.9844 - val_loss: 0.0438 - val_acc
uracy: 0.9688
```

In [93]:

```
# print model summary
dnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (32, 512)                 12800     
                                                                 
 dense_1 (Dense)             (32, 256)                 131328    
                                                                 
 dense_2 (Dense)             (32, 128)                 32896     
                                                                 
 dense_3 (Dense)             (32, 1)                   129       
                                                                 
=================================================================
Total params: 177,153
Trainable params: 177,153
Non-trainable params: 0
_________________________________________________________________
```

In [94]:

```
# evaluate the model
score = dnn.evaluate(X_test, Y_test)
print("%s: %.2f%%" % (dnn.metrics_names[1], score[1] * 100))
```

```
3/3 [==============================] - 0s 16ms/step - loss: 3.7984e-04 - accuracy: 1.0000
accuracy: 100.00%
```

Visualization of Results

We will make a graph that shows the loss and accuracy of model on training and testing data. We will make 2 graphs as:

- One graph shows the accuracy on training and testing datasets
- Second graph shows the loss on training and testing datasets

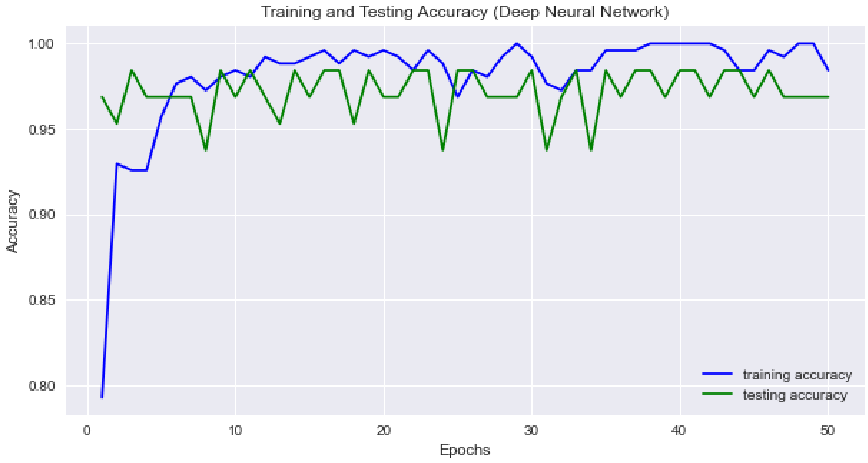
Graph 1 - Train/Test Accuracy

In [95]:

```
his = history.history
epochs = range(1, len(his['loss']) + 1)
loss = his['loss']
val_loss = his['val_loss']
acc = his['accuracy']
val_acc = his['val_accuracy']

# make a graph of train/test acc
plt.figure(figsize=(10, 5))
plt.title('Training and Testing Accuracy (Deep Neural Network)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(epochs, acc, label='training accuracy', color='blue')
plt.plot(epochs, val_acc, label='testing accuracy', color='g')
plt.legend()
```

Out[95]:

```
<matplotlib.legend.Legend at 0x22e54fe9a30>
```



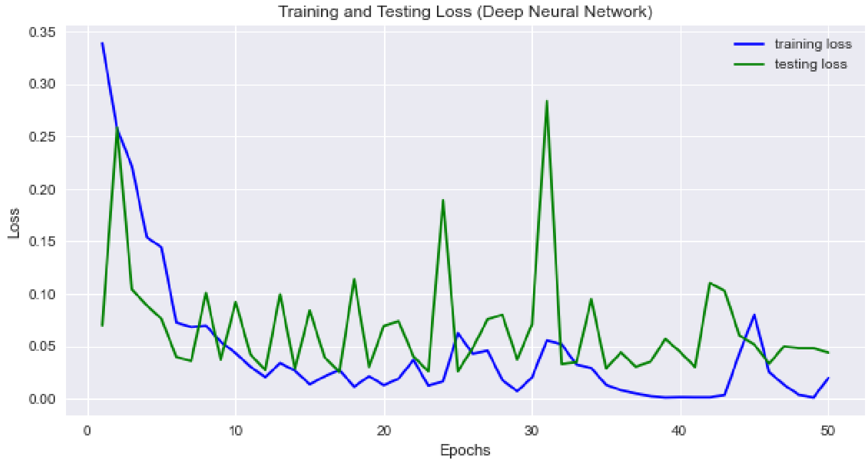
Graph 2 - Train/Test Loss

In [96]:

```
# make a graph of train/test loss
plt.figure(figsize=(10, 5))
plt.title('Training and Testing Loss (Deep Neural Network)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(epochs, loss, label='training loss', color='blue')
plt.plot(epochs, val_loss, label='testing loss', color='g')
plt.legend()
```

Out[96]:

```
<matplotlib.legend.Legend at 0x22e54f9a280>
```



In [97]:

```
# check predictions made by deep neural network
y_hat_dnn = dnn.predict(X_test)
y_hat_dnn[:5]
```

Out[97]:

```
array([[9.9976277e-01],
       [9.9999982e-01],
       [1.2190923e-19],
       [9.9992186e-01],
       [1.3222543e-24]], dtype=float32)
```

In [98]:

```
# make class predictions with dnn
predictions = (dnn.predict(X_test) > 0.5).astype(int)
predictions[:5]
```

Out[98]:

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

In [99]:

```
# make classification report
print(classification_report(Y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        30

    accuracy                           1.00        80
   macro avg       1.00      1.00      1.00        80
weighted avg       1.00      1.00      1.00        80
```

In [100]:

```
# print confusion matrix
confusion_matrix(Y_test, predictions)
```

Out[100]:

```
array([[50,  0],
       [ 0, 30]], dtype=int64)
```

e. Deep Neural Network (DNN) - V2

In [101]:

```
# make ann function
def deep_neural_network_v2():
    model = Sequential()
    model.add(Dense(256, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    return model

# call the function
dnn_v2 = deep_neural_network_v2()
```

In [102]:

```
# compiling the model
opt = Adam(0.01)

dnn_v2.compile(optimizer=opt,
               loss='binary_crossentropy',
               metrics=['accuracy'])
```

In [103]:

```python
# fit the model to data
history_v2 = dnn_v2.fit(X_train, Y_train,
                        epochs=100, verbose=1, validation_split=0.2)
```

```
Epoch 1/100
8/8 [==============================] - 2s 44ms/step - loss: 0.3465 - accuracy: 0.8281 - val_loss: 0.0880 - val
_accuracy: 0.9531
Epoch 2/100
8/8 [==============================] - 0s 11ms/step - loss: 0.2038 - accuracy: 0.9219 - val_loss: 0.1509 - val
_accuracy: 0.9531
Epoch 3/100
8/8 [==============================] - 0s 13ms/step - loss: 0.1918 - accuracy: 0.9531 - val_loss: 0.1235 - val
_accuracy: 0.9688
Epoch 4/100
8/8 [==============================] - 0s 41ms/step - loss: 0.0930 - accuracy: 0.9688 - val_loss: 0.0773 - val
_accuracy: 0.9688
Epoch 5/100
8/8 [==============================] - 0s 18ms/step - loss: 0.0616 - accuracy: 0.9805 - val_loss: 0.0654 - val
_accuracy: 0.9688
Epoch 6/100
8/8 [==============================] - 0s 11ms/step - loss: 0.0477 - accuracy: 0.9805 - val_loss: 0.0791 - val
_accuracy: 0.9688
Epoch 7/100
```

In [104]:

```python
# evaluate the model
score_v2 = dnn_v2.evaluate(X_test, Y_test)
print("%s: %.2f%%" % (dnn_v2.metrics_names[1], score_v2[1] * 100))
```

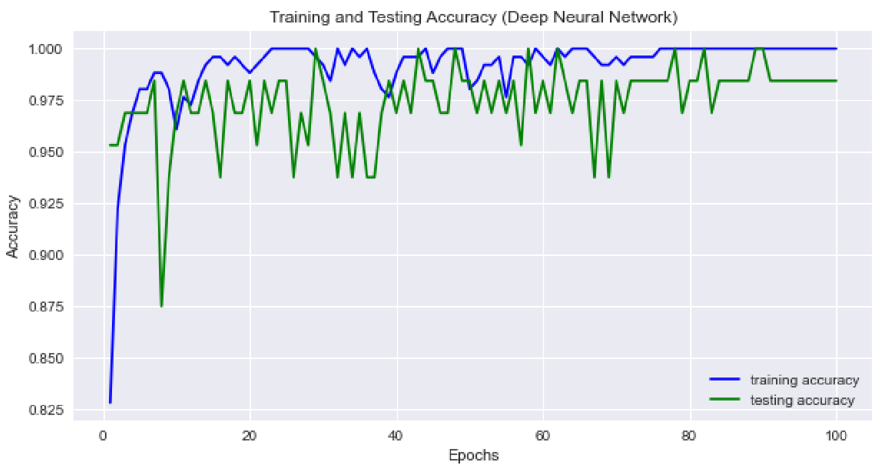
```
3/3 [==============================] - 1s 7ms/step - loss: 6.1505e-04 - accuracy: 1.0000
accuracy: 100.00%
```

In [105]:

```python
his = history_v2.history
epochs = range(1, len(his['loss']) + 1)
loss = his['loss']
val_loss = his['val_loss']
acc = his['accuracy']
val_acc = his['val_accuracy']

# make a graph of train/test acc
plt.figure(figsize=(10, 5))
plt.title('Training and Testing Accuracy (Deep Neural Network)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(epochs, acc, label='training accuracy', color='blue')
plt.plot(epochs, val_acc, label='testing accuracy', color='g')
plt.legend()
```

Out[105]:

```
<matplotlib.legend.Legend at 0x22e56449580>
```

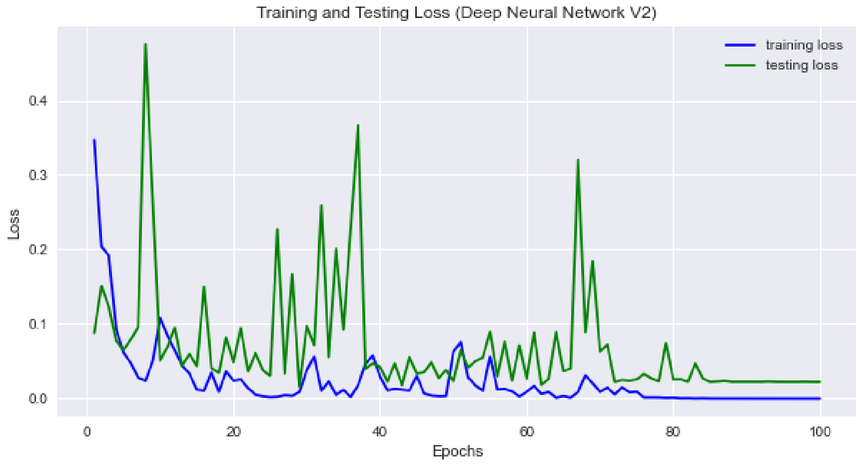


In [106]:

```
# make a graph of train/test loss
plt.figure(figsize=(10, 5))
plt.title('Training and Testing Loss (Deep Neural Network V2)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(epochs, loss, label='training loss', color='blue')
plt.plot(epochs, val_loss, label='testing loss', color='g')
plt.legend()
```

Out[106]:

```
<matplotlib.legend.Legend at 0x22e564172e0>
```



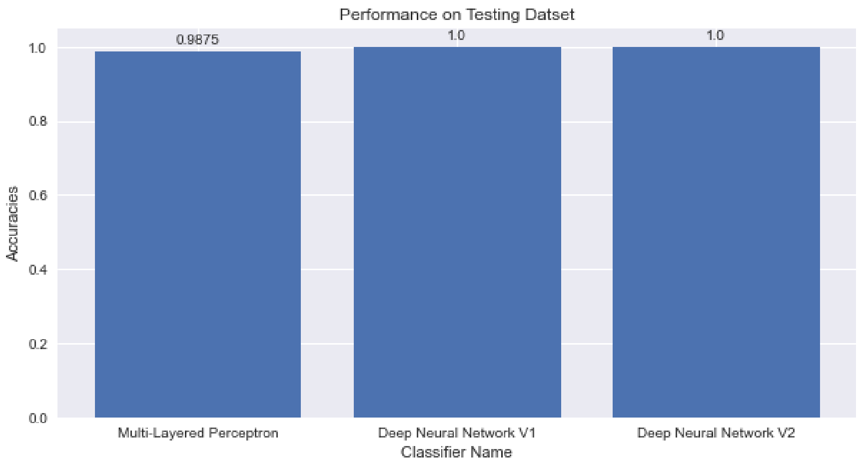
Commulative Comparison of Deep Learning Classifiers

In [109]:

```
score = dnn.evaluate(X_test, Y_test, verbose=0)
score_v2 = dnn_v2.evaluate(X_test, Y_test, verbose=0)

# plt.figure(figsize=(10, 5))
# plt.ylabel('Accuracies')
# plt.title('Performance on Test Dataset (Deep Learning)')
clf = ['Multi-Layered Perceptron', 'Deep Neural Network V1', 'Deep Neural Network V2']
acc = [accuracy_score(Y_test, y_hat_mlp), score[1], score_v2[1]]

# plt.bar(clf, acc)
```

In [110]:

```
fig, ax = plt.subplots(figsize=(10, 5))

ax.set_xlabel('Classifier Name')
ax.set_ylabel('Accuracies')

ax.set_title('Performance on Testing Datset')
# ax.set_xticks(x)
# ax.set_xticklabels(years)


pps = ax.bar(clf, acc)
for p in pps:
   height = p.get_height()
   ax.annotate('{}'.format(height),
      xy=(p.get_x() + p.get_width() / 2, height),
      xytext=(0, 3), # 3 points vertical offset
      textcoords="offset points",
      ha='center', va='bottom')

plt.show()
```



Using Stacked Ensemble Learning

In below section, we will merge multiple classifiers to make a meta estimator, which will be used for making classification. The classifiers will be used are:

- Logistic Regression
- Decision Tree
- K Nearest Neighbors
- Support Vector Machine

In [111]:

```
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import RepeatedStratifiedKFold, cross_val_score
from sklearn.naive_bayes import GaussianNB
```

In [112]:

```
# make a stacking model
def make_staks():
    # defining base models
    level_0 = list()

    level_0.append(('logistic regression:', LogisticRegression()))
    level_0.append(('cart:', DecisionTreeClassifier()))
    level_0.append(('knn:', KNeighborsClassifier()))
    level_0.append(('svm', SVC()))
    level_0.append(('gnb', GaussianNB()))

    # defining the meta estimator
    level_1 = LogisticRegression()

    # make a stacking model
    model = StackingClassifier(estimators=level_0,
                               final_estimator=level_1,
                               cv=10)
    return model
```

In [113]:

```
def get_model():
    models = dict()

    models['lr'] = LogisticRegression()
    models['cart'] = DecisionTreeClassifier()
    models['knn'] = KNeighborsClassifier()
    models['svm'] = SVC()
    models['gnb'] = GaussianNB()
    models['fusion'] = make_staks()

    return models
```

In [114]:

```
def evaluate_model(model, X, Y):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    score = cross_val_score(model, X, Y, scoring='accuracy',
                           cv=cv, n_jobs=-1, error_score='raise')
    return score
```

In [115]:

```
%%time

models = get_model()
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model, X_test, Y_test)
    results.append(scores)
    names.append(name)
    print('> %s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
```

```
> lr 0.988 (0.037)
> cart 0.908 (0.096)
> knn 0.925 (0.089)
> svm 0.988 (0.037)
> gnb 0.988 (0.037)
> fusion 0.988 (0.037)
CPU times: total: 1.05 s
Wall time: 16.4 s
```

In [116]:

```
# plot model performance for comparison
plt.style.use('default')
plt.figure(figsize=(10, 5))
plt.title('Test Accuracies of Multiple Classfiers & Fusion')
plt.boxplot(results, labels=names, showmeans=True)
plt.show()
```