MATH2319 - Machine Learning

Course project - Predicting Chronic Kidney Disease

Luke Daws

S3322003

Table of Contents

1. Introduction

For this course project we will be attempting to compare three different algorithms that will be used to predict early chronic kidney disease. I will be the following three models:

- KNN
- Decision trees
- Random forest

I will also do some feature selection and ranking with random forest importance as well as hyperparameter tuning of the 3 models.

The dataset was taken from the UCI Machine Learning Repository[1]. This dataset has 400 observations, 24 descriptive features and 1 target feature, which is the `class` feature. The `class` is a binary feature, where the value `ckd` indicates a confirmation of chronic kidney disease and `notckd` indicates a negative confirmation.

1.A Feature description

Below outlines the headings of each feature and the information about the values of each one. This information is provided in the `chronic_kidney_disease.info` file at the UCI repository:

- `age` - age (*in years*), (numerical)
- `bp` - blood pressure (*mm/Hg*), (numerical)
- `sg` - specific gravity (*1.005,1.010,1.015,1.020,1.025*), (nominal)
- `al` - albumin (*0,1,2,3,4,5*), (nominal)
- `su` - sugar (*0,1,2,3,4,5*), (nominal)
- `rbc` - red blood cells (*normal,abnormal*), (nominal)
- `pc` - pus cell (*normal,abnormal*), (nominal)
- `pcc` - pus cell clumps (*present,notpresent*), (nominal)
- `ba` - bacteria (*present,notpresent*), (nominal)
- `bgr` - blood glucose random (*mgs/dl*), (numerical)
- `bu` - blood urea (*mgs/dl*), (numerical)
- `sc` - serum creatinine (*mgs/dl*), (numerical)
- `sod` - sodium (*mEq/L*), (numerical)
- `pot` - potassium (*mEq/L*), (numerical)
- `hemo` - hemoglobin (*gms*), (numerical)
- `pcv` - packed cell volume** (numerical)
- `wbcc` - white blood cell count (*cells/cumm*), (numerical)
- `rbcc` - red blood cell count (*millions/cmm*), (numerical)
- `htn` - hypertension (*yes,no*), (nominal)
- `dm` - diabetes mellitus (*yes,no*), (nominal)
- `cad` - coronary artery disease (*yes,no*), (nominal)
- `appet` - appetite (*good,poor*), (nominal)
- `pe` - pedal edema (*yes,no,*) (nominal)
- `ane` -anemia (*yes,no*), (nominal)
- `class` - Target (*ckd,notckd*), (nominal)

** For `pcv - packed cell volume` no measurements units were provided.

---

2. Data preparation

This dataset originally comes as an `.arff` file. For this project we are required to supply our dataset as a `.csv` file. For ease of conversion, excel was used to convert the file into the required `.csv` file type. During this conversion line `371` showed a missing value exists under the feature `dm`. This is due to an extra `,` in the original `chronic_kidney_disease.arff` file, which has caused all values for that observation (from the `dm` feature onwards) to be shifted across by one feature. This issue was fixed in excel before saving as a `csv` file and the `.csv` file provided for this project has had that issue rectified.

2.A Importing packages

In [1]:

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

import numpy as np
import pandas as pd

from sklearn import preprocessing

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold
from scipy import stats
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

import altair as alt
```

2.B Loading dataset

In the `chronic_kidney_disease.info` file it is explained that any missing values have been replaced with an `?`, so they will be converted into NaN values, using the `na_values=["?"]` argument, as the dataset is loaded.

In [2]:

```
dataset = pd.read_csv('chronic_kidney_disease.csv', na_values=["?"], sep = ',')

print(f"Number of observations and features in the dataset: {dataset.shape}")
dataset.head()
```

```
Number of observations and features in the dataset: (400, 25)
```

Out[2]:

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44	7800
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38	6000
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31	7500
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32	6700
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35	7300

5 rows × 25 columns

2.C Dataset observation

First before we begin touching the data we will get an overview of it to make sure everything is being displayed correctly and see what sort of date preprocessing will need to be done.

In [3]:

```
print("The number of missing values for each feature:")
print(dataset.isna().sum())
```

```
The number of missing values for each feature:
age        9
bp        12
sg        47
al        46
su        49
rbc      152
pc        65
pcc        4
ba         4
bgr       44
bu        19
sc        17
sod       87
pot       88
hemo      52
pcv       70
wbcc     105
rbcc     130
htn        2
dm         2
cad        2
appet      1
pe         1
ane        1
class      0
dtype: int64
```

In [4]:

```
dataset.dtypes
```

Out[4]:

```
age      float64
bp       float64
sg       float64
al       float64
su       float64
rbc       object
pc        object
pcc       object
ba        object
bgr      float64
bu       float64
sc       float64
sod      float64
pot      float64
hemo     float64
pcv       object
wbcc      object
rbcc      object
htn       object
dm        object
cad       object
appet     object
pe        object
ane       object
class     object
dtype: object
```

by comparing the data types to what they should be in the `chronic_kidney_disease.info` file we can see that the `pcv`, `wbcc` and `rbcc` features are listed as objects when they should be numerical. We can look further into these features to see why it has been listed as an `object` and not a `float64.

In [5]:

```
wrong_type_cols = ['pcv','wbcc','rbcc']

for wrong_type_cols in wrong_type_cols:
    print('Column ' + wrong_type_cols + ':')
    print(dataset[wrong_type_cols].value_counts(), '\n')
```

```
Column pcv:
52      21
41      21
44      19
48      19
40      16
43      14
42      13
45      13
28      12
32      12
36      12
50      12
33      12
37      11
34      11
35       9
46       9
29       9
```

2.D Correcting values

We can see there are values that have a `\t` in front of them. This is most likely caused by the way the data has been recorded. In this case it seems that there is a leading tab space before the values so when it is imported in to this notebook that tab space is recorded as part of the value. So we will go through these columns and strip the `\t` from the values.

In [6]:

```
wrong_type_cols = ['pcv','wbcc','rbcc']

for wrong_type_cols in wrong_type_cols:
    dataset[wrong_type_cols] = dataset[wrong_type_cols].str.strip()
```

In [7]:

```
wrong_type_cols = ['pcv','wbcc','rbcc']

for wrong_type_cols in wrong_type_cols:
    print('Column ' + wrong_type_cols + ':')
    print(dataset[wrong_type_cols].value_counts(), '\n')
```

```
Column pcv:
41    21
52    21
44    19
48    19
40    16
43    15
42    13
45    13
32    12
36    12
50    12
28    12
33    12
34    11
37    11
29     9
30     9
46     9
```

Now that the `\t` has been removed there are a few `?` characters that didn't get turned into `NaN` when the file was imported so we will go ahead and convert those.

In [8]:

```
dataset = dataset.replace('?', np.nan)
```

Now that we have fixed the string values and converted the `?` values in to `NaN` we can attempt to convert those features into there correct data type (numeric).

In [9]:

```
wrong_type_cols = ['pcv','wbcc','rbcc']
dataset[wrong_type_cols] = dataset[wrong_type_cols].astype(np.number)
dataset.dtypes
```

Out[9]:

```
age      float64
bp       float64
sg       float64
al       float64
su       float64
rbc       object
pc        object
pcc       object
ba        object
bgr      float64
bu       float64
sc       float64
sod      float64
pot      float64
hemo     float64
pcv      float64
wbcc     float64
rbcc     float64
htn       object
dm        object
cad       object
appet     object
pe        object
ane       object
class     object
dtype: object
```

Now that the numerical features have been set to the correct data types we should check the catagorical data to see if they have the correct values.

In [10]:

```
obj_type_cols = ['rbc','pc','pcc','ba','htn','dm','cad','appet','pe','ane','class']

for obj_type_cols in obj_type_cols:
    print('Column ' + obj_type_cols + ':')
    print(dataset[obj_type_cols].value_counts(), '\n')
```

```
Column rbc:
normal      201
abnormal     47
Name: rbc, dtype: int64

Column pc:
normal      259
abnormal     76
Name: pc, dtype: int64

Column pcc:
notpresent    354
present        42
Name: pcc, dtype: int64

Column ba:
notpresent    374
present        22
Name: ba, dtype: int64

Column htn:
no     251
yes    147
Name: htn, dtype: int64

Column dm:
no       258
yes      134
\tno       3
\tyes      2
 yes       1
Name: dm, dtype: int64

Column cad:
no      362
yes      34
\tno      2
Name: cad, dtype: int64

Column appet:
good    317
poor     82
Name: appet, dtype: int64

Column pe:
no     323
yes     76
Name: pe, dtype: int64

Column ane:
no     339
yes     60
Name: ane, dtype: int64
```

```
Column class:
ckd       248
notckd    150
ckd\t       2
Name: class, dtype: int64
```

Again we seem to have the same problem where the `\t` has been included in the values when importing the file.

In [11]:

```
obj_type_cols = ['rbc','pc','pcc','ba','htn','dm','cad','appet','pe','ane','class']

for obj_type_cols in obj_type_cols:
    dataset[obj_type_cols] = dataset[obj_type_cols].str.strip()

obj_type_cols = ['rbc','pc','pcc','ba','htn','dm','cad','appet','pe','ane','class']

for obj_type_cols in obj_type_cols:
    print('\nColumn ' + obj_type_cols + ':')
    print(dataset[obj_type_cols].value_counts(), '\n')
```

```
Column rbc:
normal      201
abnormal     47
Name: rbc, dtype: int64


Column pc:
normal      259
abnormal     76
Name: pc, dtype: int64


Column pcc:
notpresent    354
present        42
Name: pcc, dtype: int64


Column ba:
notpresent    374
present        22
Name: ba, dtype: int64


Column htn:
no     251
yes    147
Name: htn, dtype: int64


Column dm:
no     261
yes    137
Name: dm, dtype: int64


Column cad:
no     364
yes     34
Name: cad, dtype: int64


Column appet:
good    317
poor     82
Name: appet, dtype: int64
```

```
Column pe:
no     323
yes     76
Name: pe, dtype: int64


Column ane:
no     339
yes     60
Name: ane, dtype: int64


Column class:
ckd       250
notckd    150
Name: class, dtype: int64
```

Now that the values have been stripped, we can see that each unique value in the observations are what they should be as per the `chronic_kidney_disease.info` file.

2.E Redundant features

Next, we should look to see if there are any redundant features, this can included if any features have an **ID-like** quality, This means that each observation has a unique value, **Constant features**, which is when two or more features have their values match throughout the observations and also **date** or **time** features.

We can quickly remove any **ID-like** features with the following code:

In [12]:

```
dataset = dataset.loc[:, dataset.nunique() != 1]
dataset.shape
```

Out[12]:

```
(400, 25)
```

As we can see there is still 25 features so there is no **ID-like** features.

when looking for **constant features** we are looking for features that have the same amount of unique values as well as the same amount of those unique features. When looking at the features from this dataset earlier there isn't any features that fall under that definition, so we do not have to worry about **constant features**.

Finally by looking through the `chronic_kidney_disease.info` file we know there is no **date** or **time** features.

2.F Missing values

As we are told in the `chronic_kidney_disease.info` file there are missing values so we will check each feature for them.

In [13]:

```
print("The number of missing values for each feature:")
print(dataset.isna().sum())
```

```
The number of missing values for each feature:
age        9
bp        12
sg        47
al        46
su        49
rbc      152
pc        65
pcc        4
ba         4
bgr       44
bu        19
sc        17
sod       87
pot       88
hemo      52
pcv       71
wbcc     106
rbcc     131
htn        2
dm         2
cad        2
appet      1
pe         1
ane        1
class      0
dtype: int64
```

As we can see above, there is a lot of missing values throughout this dataset. Ranging from just 1 missing in the `appet`, `pe` and `ane` features and up to 152 in the `rbc` feature. The most important thing to notice here is, there are no missing values for the target feature `class`. This means that we can still use all the observations depending on how we plan to deal with the missing values in the rest of the features.

The easiest way of dealing with missing values is to just remove the observation from the dataset, known as **complete case analysis**[2]. When removing observations this way it is important to note that this can introduce a bias into the dataset if the missing values are not distributed randomly throughout the dataset[2].

In [14]:

```
droppeddata=dataset.dropna(axis=0)
print(f"Dataset shape with missing values: {dataset.shape}")
print(f"Dataset shape where the observations have been dropped: {droppeddata.shape}")
```

```
Dataset shape with missing values: (400, 25)
Dataset shape where the observations have been dropped: (158, 25)
```

Complete case analysis removes 60.5% of the total observations from the dataset. Leaving us with 158 observations. Even though we are left with a much smaller dataset we can still use this with our models further down but for this project we are required to have at least 200 observations. so we will try some alternate ways of dealing with the missing values. One method is to check each feature and if there is over 60% of the values missing, then this feature should be removed[2].

In [15]:

```
print("The percentage of missing values for each feature:")
print(dataset.isna().sum()/400*100)
```

```
The percentage of missing values for each feature:
age       2.25
bp        3.00
sg       11.75
al       11.50
su       12.25
rbc      38.00
pc       16.25
pcc       1.00
ba        1.00
bgr      11.00
bu        4.75
sc        4.25
sod      21.75
pot      22.00
hemo     13.00
pcv      17.75
wbcc     26.50
rbcc     32.75
htn       0.50
dm        0.50
cad       0.50
appet     0.25
pe        0.25
ane       0.25
class     0.00
dtype: float64
```

So we can see that the `rbc` feature has the highest missing values totalling 38%. This doesn't exceed the 60% threshold we use to determine if the feature should be removed, therefore removing the features is not the best option.

This leaves us with the option of **Imputation**[2]. **Imputation** replaces the missing values with plausible estimated values. For continuous features, the mean or median can be used and for categorical features the mode can be used. it is important to consider that if **imputation** replaces a large amount of data, it can shift the central tendency of the feature[2]. So **imputation** should be considered carefully. It is recommended that if the feature is missing over 30% of it's values than **imputation** is to be used reluctantly and over 50% it is advised against using it at all[2]. As the features `rbc` and `rbcc` are both missing over 30% of it's values, we'll try and remove the observations where they are missing values in those two features.

In [16]:

```
mask = (dataset['rbc'].isnull() | dataset['rbcc'].isnull())
df = dataset.loc[~mask]
print(f"the shape of the dataset when observations with missing values from the 'rbc' and '
```

```
the shape of the dataset when observations with missing values from the 'rb
c' and 'rbcc' features are removed: (199, 25)
```

Unfortunately this leaves us again with just too few observations (we require 200). So as the feature `rbc` has

the highest percentage of missing values, we will remove the observations that have missing values from that feature and the rest will use **imputation**.

It should be noted that we are only going doing this method of fixing missing values as we have a requirement as part of the course project (min. 200 observations), in most circumstances it is best to go with the simplest method. In this case it would be **Complete case analysis** (the removal of observations with missing values).

In [17]:

```
mask = (dataset['rbc'].isnull())
df = dataset.loc[~mask]
print(f"the shape of the dataset when observations with missing values from the 'rbc' featu
print("\nThe missing values for each feature:")
df.isna().sum()
```

```
the shape of the dataset when observations with missing values from the 'rb
c' feature are removed: (248, 25)

The missing values for each feature:
```

Out[17]:

```
age       4
bp        7
sg        2
al        3
su        3
rbc       0
pc        9
pcc       3
ba        3
bgr      23
bu       13
sc       10
sod      33
pot      33
hemo     22
pcv      23
wbcc     38
rbcc     49
htn       2
dm        2
cad       2
appet     1
pe        1
ane       1
class     0
dtype: int64
```

We can see now that we have enough observations (248) to continue using this dataset for the project. There are no longer any missing values in the `rbc` feature but there are still some missing in the rest.

2.G Imputation

The next step in dealing with the missing values in our case is to replace all the missing values. For numerical values we can use either the mean or the median (We will use the median for our data) and for catagorical we can use the mode. Part of this code was taken from the answers to assignment 1[3]

In [18]:

```
data_clean = df.copy()

cat_col = df.columns[df.dtypes==object]
num_col = df.columns[df.dtypes==float]

for cat_col in cat_col:
    mode = data_clean[cat_col].mode()[0]
    print('Categorical Column ' + cat_col + ': Mode = ' + mode)
    data_clean[cat_col] = data_clean[cat_col].fillna(mode)

for num_col in num_col:
    median = np.round(data_clean[num_col].median(), 3)
    print('Numerical Column ' + num_col + ': Median = ' + str(median))
    data_clean[num_col] = data_clean[num_col].fillna(median)
```

```
Categorical Column rbc: Mode = normal
Categorical Column pc: Mode = normal
Categorical Column pcc: Mode = notpresent
Categorical Column ba: Mode = notpresent
Categorical Column htn: Mode = no
Categorical Column dm: Mode = no
Categorical Column cad: Mode = no
Categorical Column appet: Mode = good
Categorical Column pe: Mode = no
Categorical Column ane: Mode = no
Categorical Column class: Mode = notckd
Numerical Column age: Median = 50.0
Numerical Column bp: Median = 80.0
Numerical Column sg: Median = 1.02
Numerical Column al: Median = 0.0
Numerical Column su: Median = 0.0
Numerical Column bgr: Median = 119.0
Numerical Column bu: Median = 40.0
Numerical Column sc: Median = 1.15
Numerical Column sod: Median = 139.0
Numerical Column pot: Median = 4.5
Numerical Column hemo: Median = 13.85
Numerical Column pcv: Median = 42.0
Numerical Column wbcc: Median = 7900.0
Numerical Column rbcc: Median = 4.9
```

In [19]:

```
data_clean.isna().sum()
```

Out[19]:

```
age      0
bp       0
sg       0
al       0
su       0
rbc      0
pc       0
pcc      0
ba       0
bgr      0
bu       0
sc       0
sod      0
pot      0
hemo     0
pcv      0
wbcc     0
rbcc     0
htn      0
dm       0
cad      0
appet    0
pe       0
ane      0
class    0
dtype: int64
```

As we can see now there are no more missing values.

2.H Encoding

The dataframe we have created here `data_clean` will be used below in the data exploration section but to use the machine learning algorithms, one-hot-encoding of the categorical features needs to take place. We will start by creating a dataframe without the target feature `class` and make a separate dataframe with the target feature.

In [20]:

```
data_clean_encoded = data_clean.drop(columns = 'class')

data_target = data_clean['class']
```

When we looked through our categorical features earlier and if we look at the descriptions of the feature in the `chronic_kidney_disease.info` file, we find that all the categorical features are binary. We can use this piece of code from assignment 1[3] to one-hot-encode all the categorical features.

In [21]:

```
cat_col = df.columns[df.dtypes==object]

for col in cat_col:
    if col == 'class':
        continue
    q = len(data_clean_encoded[col].unique())
    if (q == 2):
        data_clean_encoded[col] = pd.get_dummies(data_clean_encoded[col], drop_first=True)

print(data_clean_encoded.shape)

data_clean_encoded.head()
```

```
(248, 24)
```

Out[21]:

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	hemo	pcv	wbcc	rbcc	htn	dı
2	62.0	80.0	1.010	2.0	3.0	1	1	0	0	423.0	...	9.6	31.0	7500.0	4.9	0	
3	48.0	70.0	1.005	4.0	0.0	1	0	1	0	117.0	...	11.2	32.0	6700.0	3.9	1	
4	51.0	80.0	1.010	2.0	0.0	1	1	0	0	106.0	...	11.6	35.0	7300.0	4.6	0	
7	24.0	80.0	1.015	2.0	4.0	1	0	0	0	410.0	...	12.4	44.0	6900.0	5.0	0	
8	52.0	100.0	1.015	3.0	0.0	1	0	1	0	138.0	...	10.8	33.0	9600.0	4.0	1	

5 rows × 24 columns

We know need to encode the target feature so that 1 corrosponds to the positive class (`ckd`) and 0 corrosponds to the negative class (`notckd`).

In [22]:

```
print(data_target.value_counts())


target_mapping = {'notckd': 0, 'ckd': 1}
data_target = data_target.replace(target_mapping)

print(data_target.value_counts())
```

```
notckd    141
ckd       107
Name: class, dtype: int64
0    141
1    107
Name: class, dtype: int64
```

2.I Scaling

Next, we need to normalise the data using scaling. For this project we will go with min-max scaling, which will be done usng the `scikit learn` module.

In [23]:

```
data_clean_encoded_scale = data_clean_encoded.copy()
col_names_data_clean_encoded_scale = data_clean_encoded_scale.columns

data_clean_encoded_scale = preprocessing.MinMaxScaler().fit_transform(data_clean_encoded_sc

data_clean_encoded_scale_df = pd.DataFrame(data_clean_encoded_scale, columns=col_names_data

print(data_clean_encoded_scale_df.shape)

data_clean_encoded_scale_df.head()
```

```
(248, 24)
```

Out[23]:

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	hemo	pcv	
0	0.740741	0.500000	0.25	0.4	0.6	1.0	1.0	0.0	0.0	0.856838	...	0.442177	0.488889	0.2
1	0.567901	0.333333	0.00	0.8	0.0	1.0	0.0	1.0	0.0	0.202991	...	0.551020	0.511111	0.1
2	0.604938	0.500000	0.25	0.4	0.0	1.0	1.0	0.0	0.0	0.179487	...	0.578231	0.577778	0.2
3	0.271605	0.500000	0.50	0.4	0.8	1.0	0.0	0.0	0.0	0.829060	...	0.632653	0.777778	0.1
4	0.617284	0.833333	0.50	0.6	0.0	1.0	0.0	1.0	0.0	0.247863	...	0.523810	0.533333	0.3

5 rows × 24 columns

Now that we have encoded all the categorical features and the target feature, and done min-max scaling. we need to add the target feature back into the scaled dataframe

In [24]:

```
data_clean_ml = data_clean_encoded_scale_df.assign(target = data_target.values)

data_clean_ml.head()
```

Out[24]:

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc	
0	0.740741	0.500000	0.25	0.4	0.6	1.0	1.0	0.0	0.0	0.856838	...	0.488889	0.219008	0.4
1	0.567901	0.333333	0.00	0.8	0.0	1.0	0.0	1.0	0.0	0.202991	...	0.511111	0.185950	0.3
2	0.604938	0.500000	0.25	0.4	0.0	1.0	1.0	0.0	0.0	0.179487	...	0.577778	0.210744	0.4
3	0.271605	0.500000	0.50	0.4	0.8	1.0	0.0	0.0	0.0	0.829060	...	0.777778	0.194215	0.4
4	0.617284	0.833333	0.50	0.6	0.0	1.0	0.0	1.0	0.0	0.247863	...	0.533333	0.305785	0.3

5 rows × 25 columns

The dataset is now ready to be used with any of the various algorithims we plan on using.

3. Data exploration

3.A Summary statistics

Our first look into our dataset will begin with the summary statistics. This will provide us with a good overview of the data and can also highlight any unusual values in the dataset. The summary statistics are split into the numerical and categorical summaries for ease of viewing:

In [25]:

```
print("This is the numerical features summary statistics:")
data_clean.describe(include = np.number).round(2)
```

```
This is the numerical features summary statistics:
```

Out[25]:

	age	bp	sg	al	su	bgr	bu	sc	sod	pot	hemo	
count	248.00	248.00	248.00	248.00	248.00	248.00	248.00	248.00	248.00	248.00	248.00	2
mean	49.22	75.16	1.02	1.03	0.33	138.08	56.95	2.69	138.56	4.71	13.20	
std	16.66	11.70	0.01	1.42	0.95	69.34	54.74	4.03	6.89	3.55	2.88	
min	2.00	50.00	1.00	0.00	0.00	22.00	1.50	0.40	111.00	2.50	3.10	
25%	38.00	70.00	1.01	0.00	0.00	100.00	26.75	0.80	135.00	3.90	11.10	
50%	50.00	80.00	1.02	0.00	0.00	119.00	40.00	1.15	139.00	4.50	13.85	
75%	61.25	80.00	1.02	2.00	0.00	137.25	53.00	2.35	142.00	4.90	15.20	
max	83.00	110.00	1.02	5.00	5.00	490.00	391.00	32.00	163.00	47.00	17.80	

In [26]:

```
print("This is the categorical features summary statistics:")
data_clean.describe(include = np.object).round(2)
```

```
This is the categorical features summary statistics:
```

Out[26]:

	rbc	pc	pcc	ba	htn	dm	cad	appet	pe	ane	class
count	248	248	248	248	248	248	248	248	248	248	248
unique	2	2	2	2	2	2	2	2	2	2	2
top	normal	normal	notpresent	notpresent	no	no	no	good	no	no	notckd
freq	201	190	219	230	181	190	231	207	206	217	141

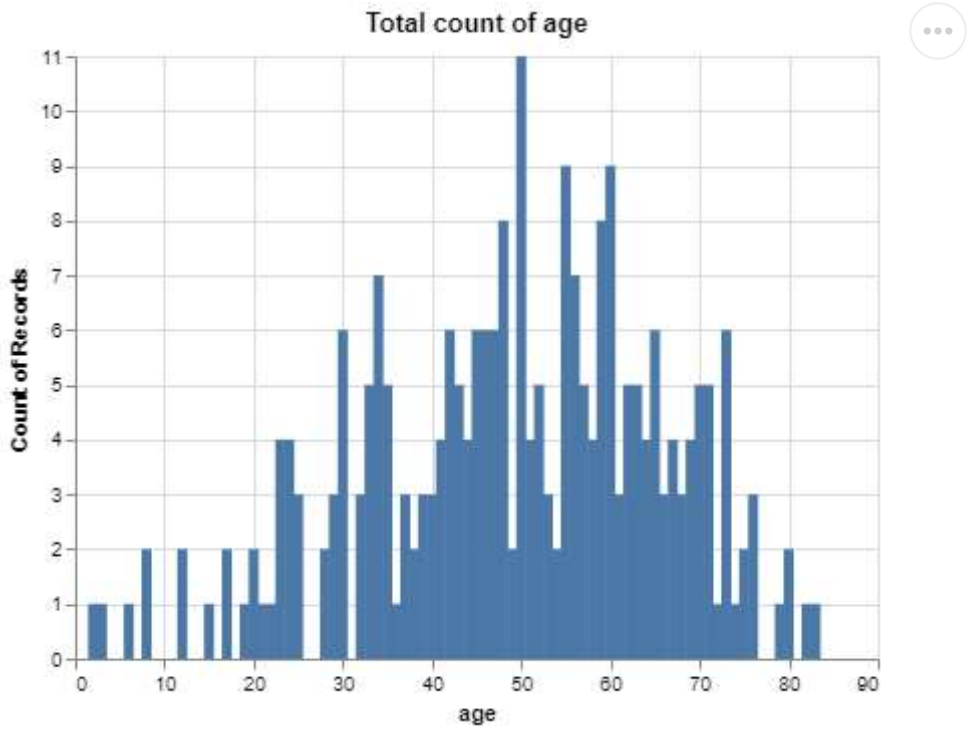
The first thing we are looking at here is that the count on all features is the same this means that all the missing values have been accounted for and there is an equal amount of observations for each feature. Using the summary statistics we can look for unusual values. The best use of this is with the age feature. We can see that the max age recorded is `80` and the minimum is `2`. The max age is fine and not something that jumps out as odd but the min age of `2` seems a little odd so we can look further into it.

3.B Univariate visualisations

In [27]:

```
alt.Chart(data_clean, title='Total count of age').mark_bar().encode(x='age', y='count()')
```

Out[27]:



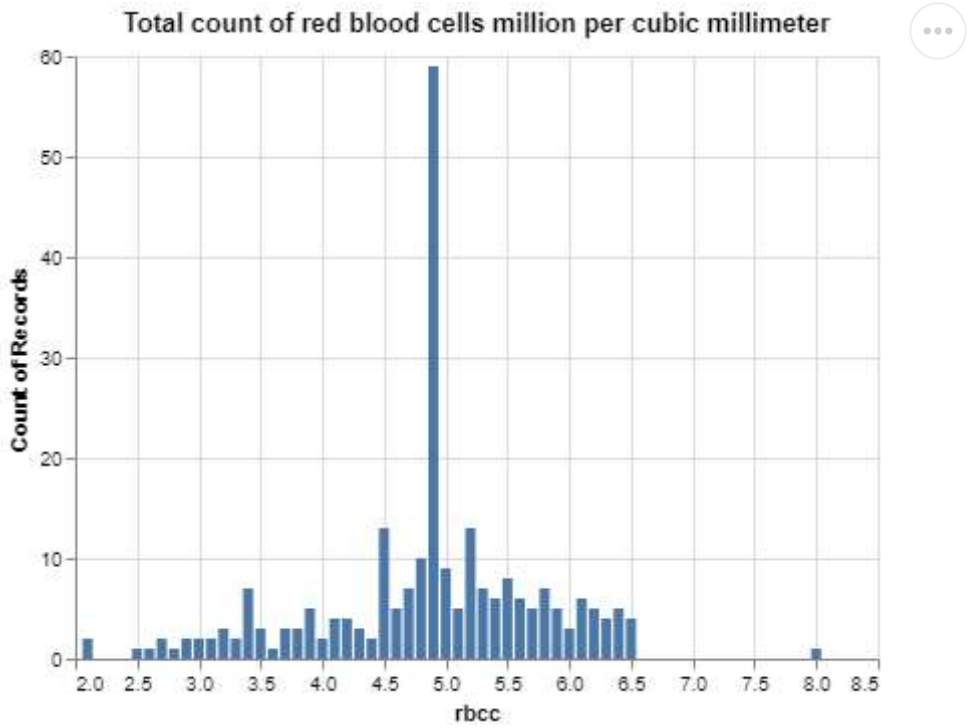
Looking at the histogram above we can see that there are quite a few counts of ages on the lower end. So in terms of the min age being 2, we wouldn't consider that unreasonable as they may have tested patients of that age. We can also see that this is left skewed.

When we were dealing with missing values, we decided to completely remove all the observations `rbc` feature as it was over 30% missing and it was the one feature missing the most values. the second highest feature missing values was the `rbcc` feature. this feature was missing 32.75% of its value but we decided to use the median to replace those values. Let's have a look at the histogram for that feature and see what kind of effect it has had on our dataset.

In [28]:

```
alt.Chart(data_clean, title='Total count of red blood cells million per cubic millimeter').
```

Out[28]:



AS you can see there is a huge spike in values at 4.9, to approximately 60 counts. This shows that replacing values when a significant amount of values are missing can cause some unusual behavours in the features.

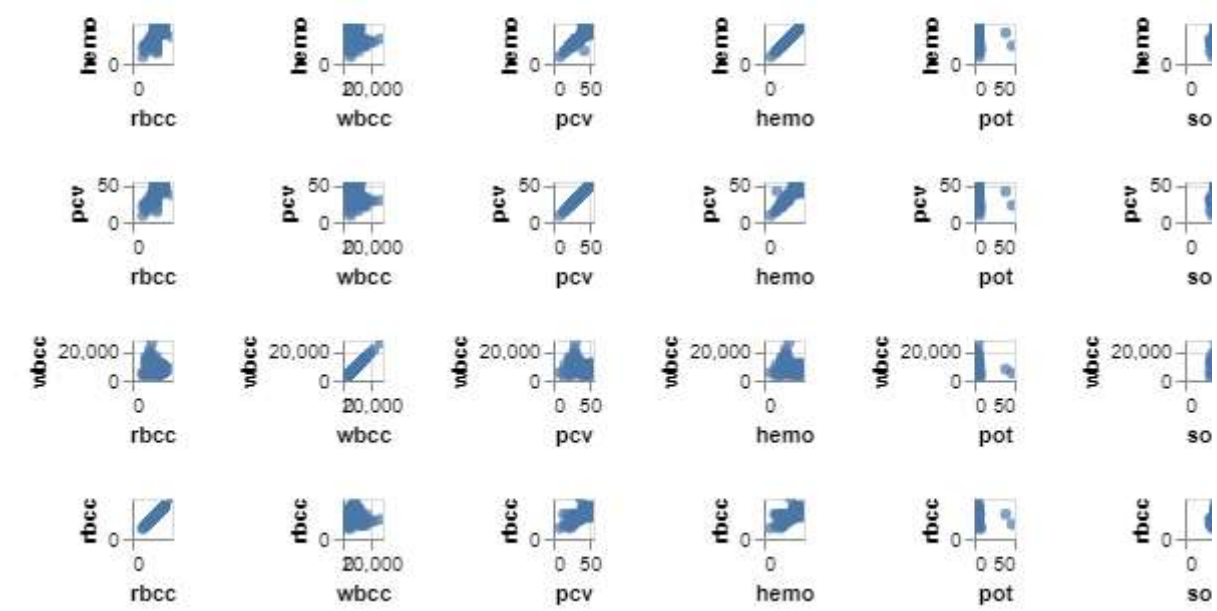
3.C Bivariant visualisations

When exploring bivariant data visualisations, if there's no particular features you want to compare, it can be easier to start out with a scatter matrix. With a scatter matrix you can see all the variable compared with each other all displayed at once and then you can begin to focus on the ones that seem interesting. A scatter matrix doesn't work that well with categorical features expecially not binay features, so they have been excluded from this scatter matrix.

In [29]:

```
alt.Chart(data_clean).mark_circle().encode(
    alt.X(alt.repeat("column"), type='quantitative'),
    alt.Y(alt.repeat("row"), type='quantitative'),
    color='Origin:N'
).properties(
    width=20,
    height=20
).repeat(
    row=['age', 'bp','sg','al','su','bgr','bu','sc','sod','pot','hemo','pcv','wbcc','rbcc']
    column=['rbcc', 'wbcc','pcv','hemo','pot','sod', 'sc','bu','bgr','su','al','sg','bp','a
).interactive()
```

Out[29]:

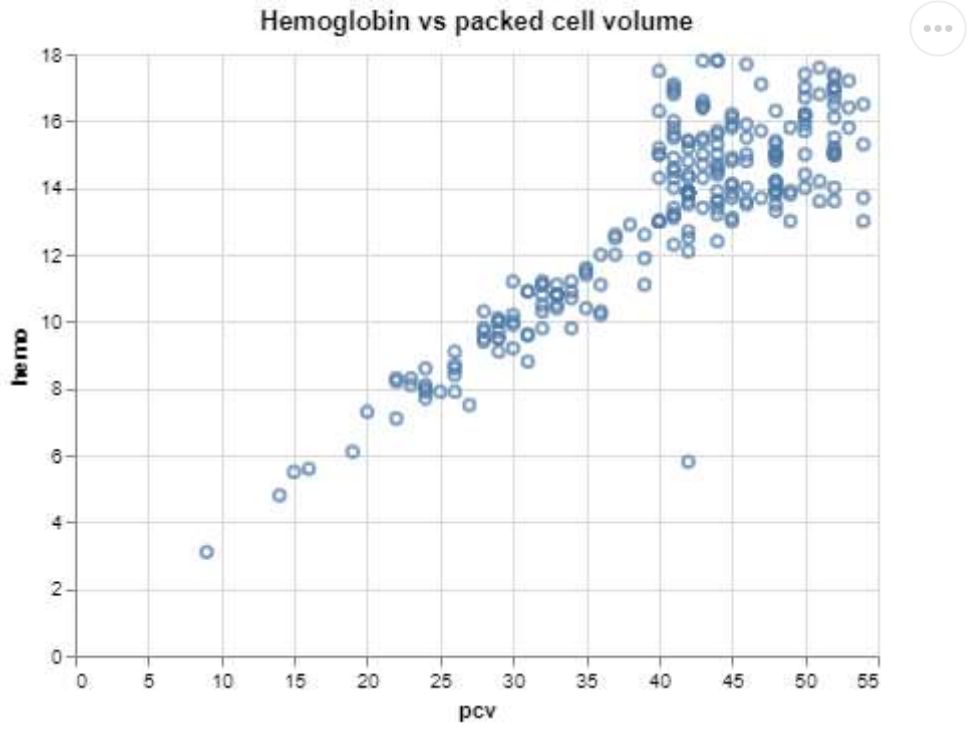


Unfortunatly with with so many features being compared with each other a scatter matrix can become unweidly. But we can still see some interesting scatter plots that we can focus in on. One in particular is the `pcv` vs `hemo` scatter plot.

In [30]:

```
alt.Chart(data_clean, title = 'Hemoglobin vs packed cell volume').mark_point().encode(
    x='pcv:Q',
    y='hemo:Q',
)
```

Out[30]:

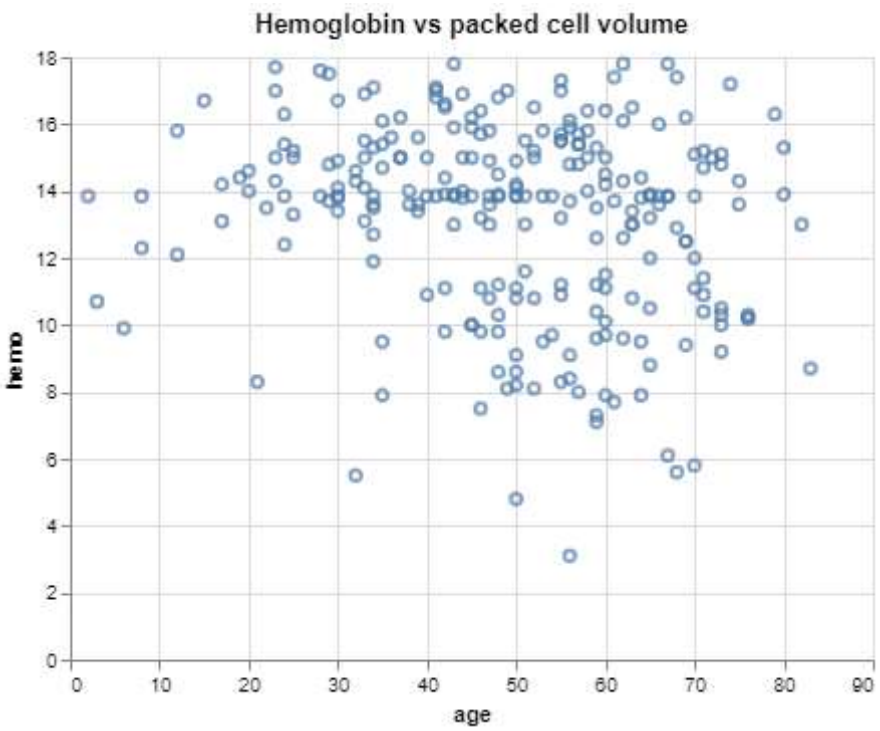


Here we can see that there is a very strong positve trend with the increase in hemoglobin there could be a correlation in increased packed cell volume.

In [31]:

```
alt.Chart(data_clean, title = 'Hemoglobin vs packed cell volume').mark_point().encode(
    x='age:Q',
    y='hemo:Q',
)
```

Out[31]:



This graph was chosen specifically because there doesn't seem to be any obvious trends and the data seems to be more or less distributed pretty evenly across the graph. As this isn't the most interesting of graphs now it would be interesting to see if adding another variable provides more information.

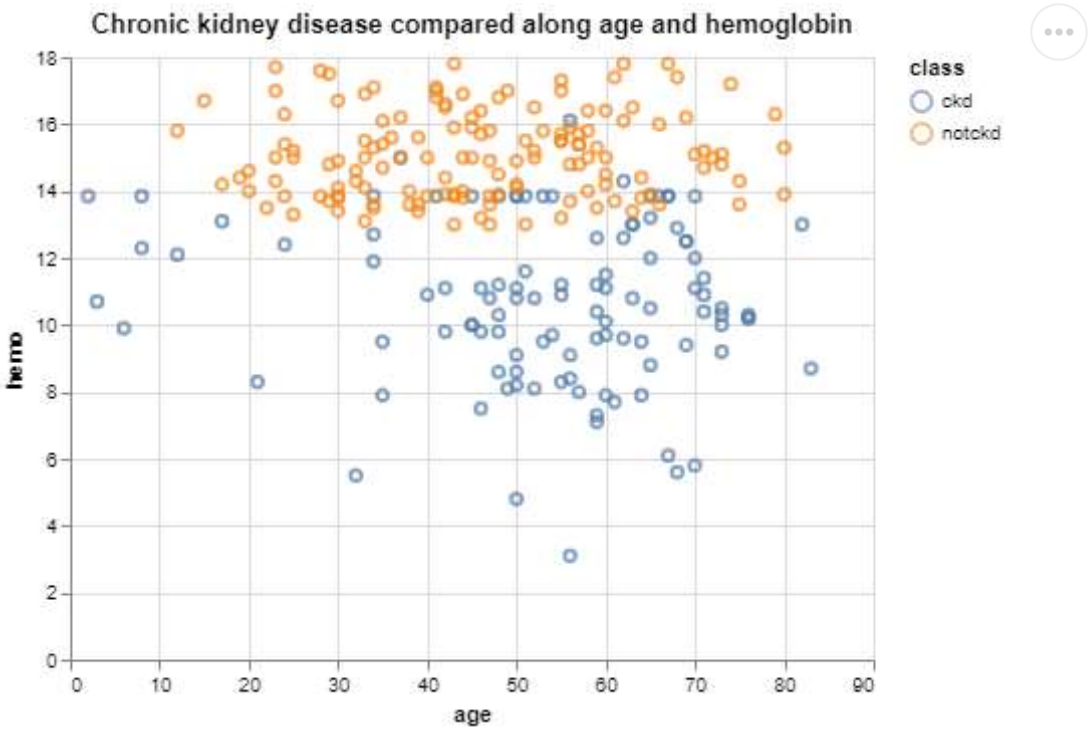
3.D Trivariant visulisations

This first visulisation we will look at is the `Hemoglobin vs packed cell volume` graph where the with the taget value as the third variable.

In [32]:

```
alt.Chart(data_clean, title = 'Chronic kidney disease compared along age and hemoglobin' ).
    x='age:Q',
    y='hemo:Q',
    color='class:N',
    tooltip=['Name:N', 'url:N']
)
```

Out[32]:



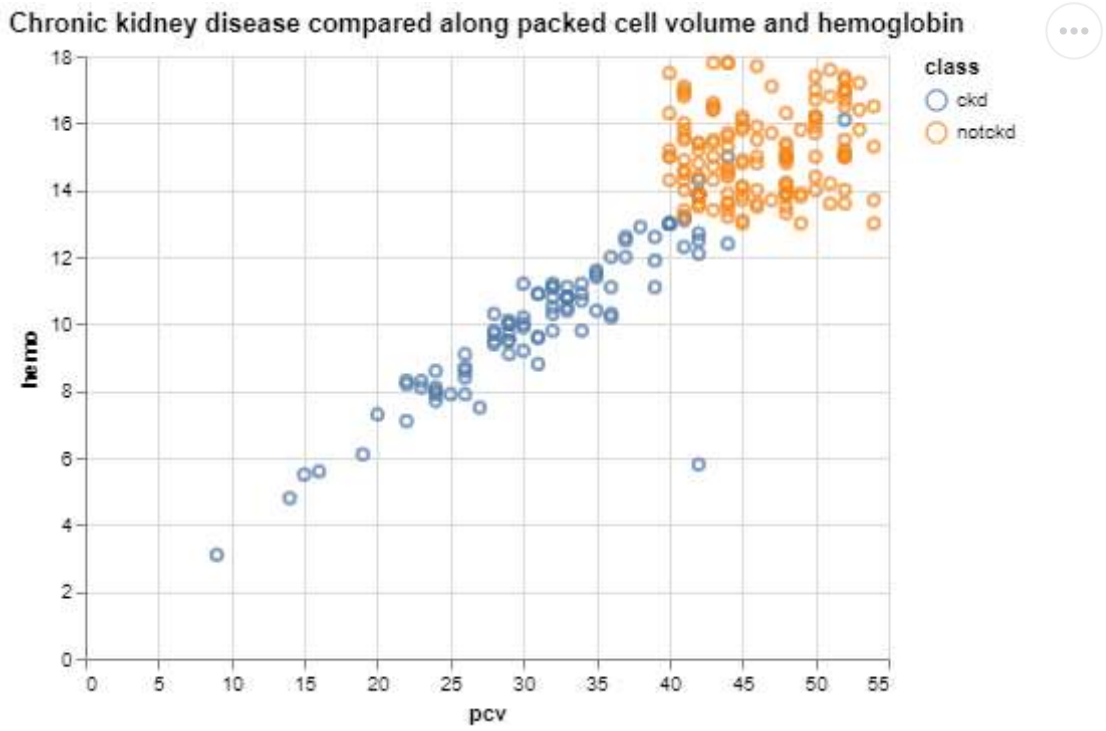
While the first graph that just compared the features of `age` and `hemo` didn't provided us with any information, when we add the target feature `class` which tells us whether or no some one does have chronic kidney disease, we can see that those with approximately 14gms and above in hemoglobin don't have chronic kidney disease but it also shows that the `age` feature doesn't have an effect on both the other features.

Now we can try the graph of `hemo` vs `pcv` where there seems to be a strong correlation of the two but is there a correlation with the target feature.

In [33]:

```
alt.Chart(data_clean, title = 'Chronic kidney disease compared along packed cell volume and
    x='pcv:Q',
    y='hemo:Q',
    color='class:N',
    tooltip=['Name:N', 'url:N']
)
```

Out[33]:



Interestingly enough we can see that with a higher hemoglobin count and higher packed cell volume there is less chance of chronic kidney disease. with this just this information we could almost start modelling a prediction with linear regression.

---

4 Predictive modelling

Now that we have prepared the data for analysis and briefly explored it, we will start working towards tuning and refining the three predictive models we will use on our dataset.

Firstly, we will explore feature selection and ranking, There are a few different algorithms for selecting the best features. For this project we will only be using one, random forest importance, which we will compare the performance with all the features using a paired t-test.

The next stage will be cross validation and hyperparameter tuning of the KNN, random forest and decision tree algorithms. The performance of these algorithms will be compared to see which of the tree performed the best on our dataset.

A lot of the code for this section is used from various tutorials found in the Math2319 machine learning class at RMIT[4], where this project is due, and also amongst the tutorials found at the www.featureranking.com (http://www.featureranking.com) website[5]

Before moving into the feature selection, we will be defining some variables that will be used. The number of features we will be wanting to define as the best is `10` and we will be using the `roc_auc` scoring metric. We will use the `DecisionTreeClassifier` with a max depth of `5` as a wrapper to assess the performance of the feature selection and ranking.

In [34]:

```
num_features = 10
scoring_metric = 'roc_auc'
clf = DecisionTreeClassifier(max_depth=5, random_state=999)
Data = data_clean_ml.drop(columns = 'target').values
target = data_clean_ml['target'].values
```

We will use the stratified 5-fold cross-validation. This will be used in both the feature selection performance assessment as well as the model performance assessments.

In [35]:

```
cv_method = StratifiedKFold(n_splits=5, shuffle=True, random_state=999)
```

4.A Feature selection

Before we choose which features to select we need to assess how well our classifier performs using all the features so we have something to compare with.

4.A.a Full set of features

In [36]:

```
cv_results_full = cross_val_score(estimator=clf,
                                  X=Data,
                                  y=target,
                                  cv=cv_method,
                                  scoring=scoring_metric)
cv_results_full
```

Out[36]:

```
array([1.        , 1.        , 0.97619048, 0.95238095, 1.        ])
```

In [37]:

```
cv_results_full.mean().round(3)
```

Out[37]:

```
0.986
```

4.A.b Random forest importance

We will now use random forest importance to see which are the 10 most important features. This will be performed using 100 trees.

In [38]:

```
model_rfi = RandomForestClassifier(n_estimators=100)
model_rfi.fit(Data, target)
fs_indices_rfi = np.argsort(model_rfi.feature_importances_)[::-1][0:num_features]
```

Now that the model has been run we will see the 10 best features.

In [39]:

```
best_features_rfi = data_clean_ml.columns[fs_indices_rfi].values
print(best_features_rfi)
```

```
['al' 'sc' 'sg' 'pcv' 'htn' 'hemo' 'pc' 'bu' 'rbcc' 'bgr']
```

Now that we know what the top 10 most important features are we will create a histogram so we can visualise it and get an idea of how important they are compared to each other.

In [40]:

```
feature_importances_rfi = model_rfi.feature_importances_[fs_indices_rfi]
feature_importances_rfi
```

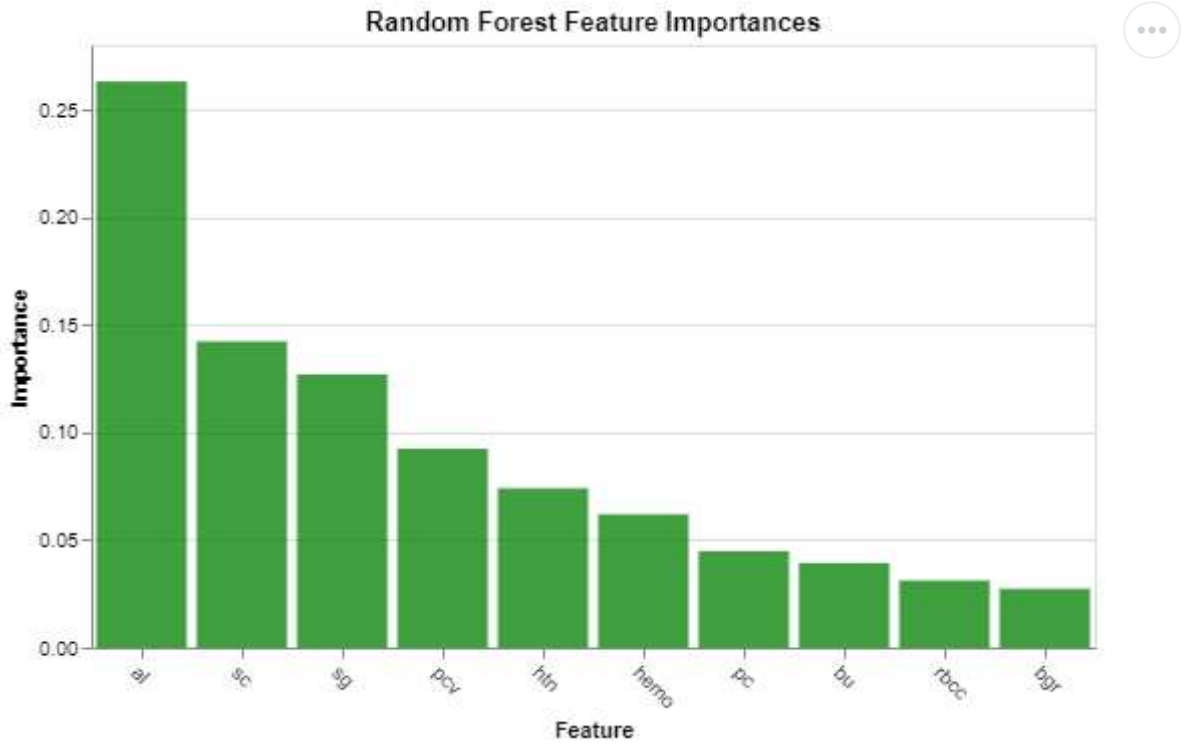
Out[40]:

```
array([0.26311367, 0.14226777, 0.12698815, 0.09246515, 0.07406507,
       0.06194331, 0.04481714, 0.03928791, 0.03118154, 0.02738165])
```

In [41]:

```
def plot_imp(best_features, scores, method_name, color):

    df = pd.DataFrame({'features': best_features,
                       'importances': scores})

    chart = alt.Chart(df,
                      width=500,
                      title=method_name + ' Feature Importances'
                     ).mark_bar(opacity=0.75,
                                color=color).encode(
        alt.X('features', title='Feature', sort=None, axis=alt.AxisConfig(labelAngle=45)),
        alt.Y('importances', title='Importance')
    )

    return chart

plot_imp(best_features_rfi, feature_importances_rfi, 'Random Forest', 'green')
```

Out[41]:



What we see here is that the feature `al` is the most important feature of the lot. `hemo`, `sg` and `sc` are approximatly as important as each other.

Now we assesshow well it performs with our classifer.

In [42]:

```
cv_results_rfi = cross_val_score(estimator=clf,
                                 X=Data[:, fs_indices_rfi],
                                 y=target,
                                 cv=cv_method,
                                 scoring=scoring_metric)
cv_results_rfi.mean().round(3)
```

Out[42]:

```
0.986
```

In [43]:

```
print(f"This is the result of using full features: {cv_results_full.mean().round(3)}")
print(f"This is the result of using 10 features selected by Random forest importance: {cv_r
```

```
This is the result of using full features: 0.986
This is the result of using 10 features selected by Random forest importanc
e: 0.986
```

We can see that both performed very well, both receiving a score of `0.986` . We will still do a paired t-test to check which ones are statistically significant.

4.A.c Paired t-test

In [44]:

```
print(f"P-Value for Full vs Random forest importance: {stats.ttest_rel(cv_results_full, cv_
```

```
P-Value for Full vs Random forest importance: nan
```

Unfortunately we recieve a `nan` out put this is due to getting the exact same result from using the full features aswell as random forest importance `0.986` , we could say that using either methods will work on our dataset. Since there seems to be no difference, we will use the full features from the dataset.

4.B Hyperparameter Tuning and Visualisation

Our next step is to find what are the optimal parameters we will use for our algorithms. We will split our data up into 70% training data and 30% test data.

In [45]:

```
D_train, D_test, t_train, t_test = train_test_split(Data, target, test_size = 0.3, random_s
```

4.B.a KNN

we will start with the KNN algorithm. in order to tune the algorithm, we need to provide the parameters it will be testing. We will define the number of neighbors `n_neighbors` using between `1-7` . Then we'll select the `p` values which are the different distances (1= Manhattan, 2 = Euclidean, and 5 = Minkowski.

In [46]:

```
params_KNN = {'n_neighbors': [1, 2, 3, 4, 5, 6, 7],
              'p': [1, 2, 5]}
```

With those defined we need to select model and parameters for the `GridSearchCV` function which are `KNeighborsClassifier()` and `KNN_params` respectivly. we will use the same `CV method` used in the feature selection and use the same scoring_metric to optimize. This process will be very similar for the rest of the hyperparameter tuning of the other algorithms.

In [47]:

```
gs_KNN = GridSearchCV(estimator=KNeighborsClassifier(),
                      param_grid=params_KNN,
                      cv=cv_method,
                      verbose=1,
                      scoring=scoring_metric,
                      return_train_score=True)
gs_KNN.fit(D_train, t_train);
```

```
Fitting 5 folds for each of 21 candidates, totalling 105 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 105 out of 105 | elapsed:    0.8s finished
C:\Users\pooli\Anaconda3\lib\site-packages\sklearn\model_selection\_search.p
y:841: DeprecationWarning: The default of the `iid` parameter will change fr
om True to False in version 0.22 and will be removed in 0.24. This will chan
ge numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

With the KNN model now fitted to the data. We can see what the best parameters for this dataset are.

In [48]:

```
print(f"Best parameters for KNN: {gs_KNN.best_params_}")
```

```
Best parameters for KNN: {'n_neighbors': 1, 'p': 2}
```

After our stratified 5-fold cross validation, we can observe that the optimal number of neighbors is `1` (`n_neighbors`) and the optimal distance to use is `2` (`Euclidean`). We can display the score of the best parameter as well as display the score for the rest of the parameters.

In [49]:

```
print(f"Score of the best parameters: {gs_KNN.best_score_}")
print(f"\nScore of the all the parameters: \n{gs_KNN.cv_results_['mean_test_score']}")
```

```
Score of the best parameters: 0.9934489402697496

Score of the all the parameters:
[0.97384393 0.99344894 0.99344894 0.97384393 0.99344894 0.99344894
 0.98034682 0.99344894 0.99344894 0.98689788 0.99344894 0.99344894
 0.98689788 0.99344894 0.99344894 0.98689788 0.99344894 0.99344894
 0.98689788 0.99344894 0.99344894]
```

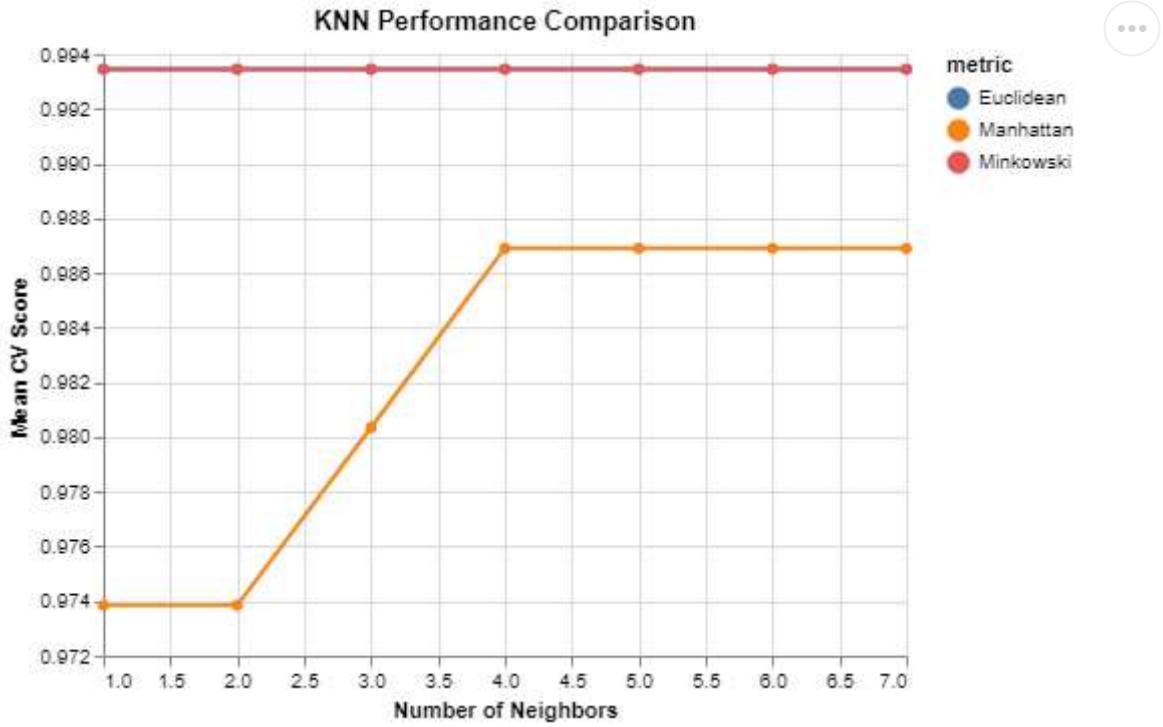
It's good to visualise how these performances went over all the parameters. So, we will put together line graph

to see how much better our best parameters did.

In [50]:

```
results_KNN = pd.DataFrame(gs_KNN.cv_results_['params'])
results_KNN['test_score'] = gs_KNN.cv_results_['mean_test_score']
results_KNN['metric'] = results_KNN['p'].replace([1,2,5], ["Manhattan", "Euclidean", "Minko

alt.Chart(results_KNN,
          title='KNN Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('n_neighbors', title='Number of Neighbors'),
    alt.Y('test_score', title='Mean CV Score', scale=alt.Scale(zero=False)),
    color='metric'
)
```

Out[50]:



With the KNN model now fitted to the data. We can see what the best parameters for this dataset are. we can see that as the number of neighbors increases the worse the overall performance got. Overall the `Minkowski` did the best along the neighbors but the `Euclidean` distance was just as good in some spots. We can see that the best number of neighbors is `1` and that the `Euclidean` Distance is best used at that number of neighbors, which confirms our best parameters we got earlier.

4.B.b Decision tree

Next is the Decision tree algorithm. We will be assessing `criterion` (either `gini` or `entropy`), `max_depth` (from `1-8`) and the `min_samples_split` (either `2` or `3`).

In [51]:

```
df_classifier = DecisionTreeClassifier(random_state=999)

params_DT = {'criterion': ['gini', 'entropy'],
             'max_depth': [1, 2, 3, 4, 5, 6, 7, 8],
             'min_samples_split': [2, 3]}

gs_DT = GridSearchCV(estimator=df_classifier,
                     param_grid=params_DT,
                     cv=cv_method,
                     verbose=1,
                     scoring=scoring_metric)

gs_DT.fit(D_train, t_train);
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.

Fitting 5 folds for each of 32 candidates, totalling 160 fits

[Parallel(n_jobs=1)]: Done 160 out of 160 | elapsed:    0.3s finished
```

In [52]:

```
print(f"Best parameters for DT: {gs_DT.best_params_}")
```

```
Best parameters for DT: {'criterion': 'gini', 'max_depth': 1, 'min_samples_s
plit': 2}
```

In [53]:

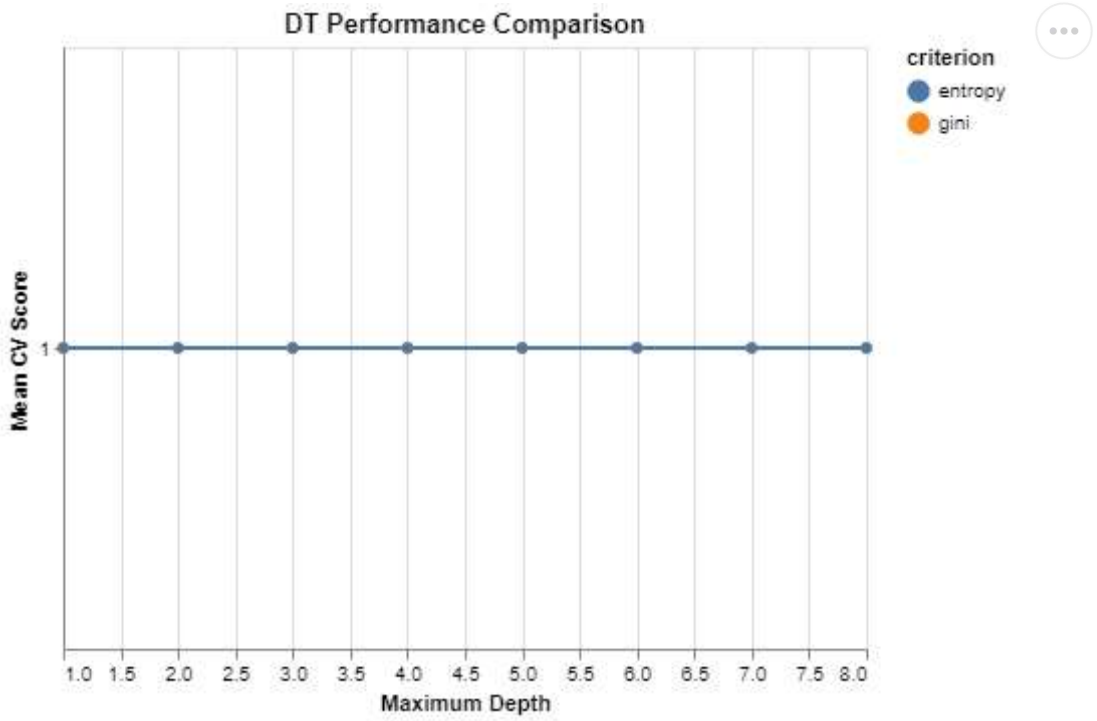
```
print(f"Score of the best parameters: {gs_DT.best_score_}")
print(f"\nScore of the all the parameters: \n{gs_DT.cv_results_['mean_test_score']}")
```

```
Score of the best parameters: 0.9738439306358383

Score of the all the parameters:
[0.97384393 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393
 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393
 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393
 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393
 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393 0.97384393
 0.97384393 0.97384393]
```

In [54]:

```
results_DT = pd.DataFrame(gs_DT.cv_results_['params'])
results_DT['test_score'] = gs_DT.cv_results_['mean_test_score']
results_DT.columns

alt.Chart(results_DT,
          title='DT Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('max_depth', title='Maximum Depth'),
    alt.Y('test_score', title='Mean CV Score', aggregate='average', scale=alt.Scale(zero=Fa
    color='criterion'
)
```

Out[54]:



The DT Performance Comparison graph above shows that for both the entropy and gini criterion they seem to follow the same path. They both perform best with a maximum depth of 1. So, in this case the best parameters for the Decision tree algorithm is what was suggested above `criterion: gini, max_depth: 1, min_samples_split: 2` .

4.B.c Random Forest

For the Random forest algorithm. We will be assessing `max_features` (`auto` , `sqrt` and `log2`), `min_samples_leaf` (`1` , `2` , `4`) and `n_estimators` (`10` , `50` , `100` , `200` , `400` , `600` , `800`).

In [55]:

```
df_classifier = RandomForestClassifier(random_state=999)

params_RF = {'max_features': ['auto', 'sqrt', 'log2'],
             'min_samples_leaf': [1, 2, 4],
             'n_estimators': [10, 50, 100, 200, 400, 600, 800]}

gs_RF = GridSearchCV(estimator=df_classifier,
                     param_grid=params_RF,
                     cv=cv_method,
                     verbose=1,
                     scoring=scoring_metric)

gs_RF.fit(D_train, t_train);
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.

Fitting 5 folds for each of 63 candidates, totalling 315 fits

[Parallel(n_jobs=1)]: Done 315 out of 315 | elapsed:  1.2min finished
```

In [56]:

```
print(f"Best parameters for KNN: {gs_RF.best_params_}")
```

```
Best parameters for KNN: {'max_features': 'auto', 'min_samples_leaf': 1, 'n_
estimators': 50}
```

In [57]:

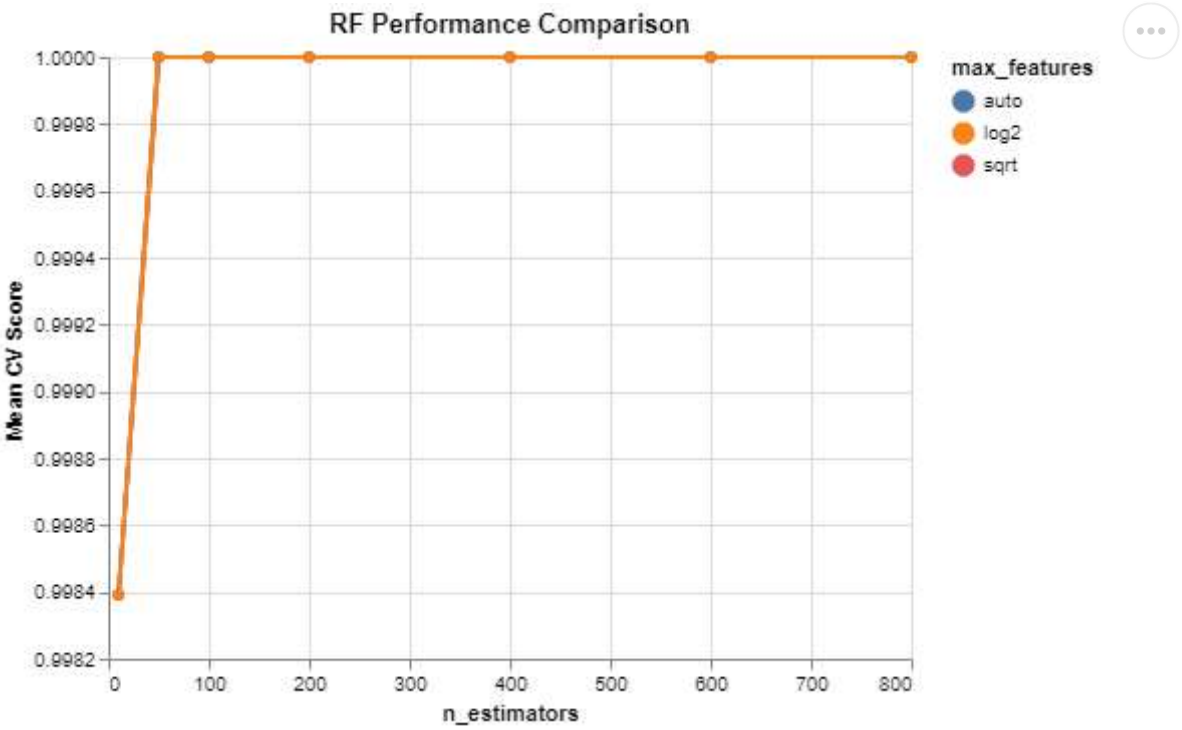
```
print(f"Score of the best parameters: {gs_RF.best_score_}")
print(f"\nScore of the all the parameters: \n{gs_RF.cv_results_['mean_test_score']}")
```

```
Score of the best parameters: 1.0

Score of the all the parameters:
[0.99862083 1.         1.         1.         1.         1.
 1.         0.99793124 1.         1.         1.         1.
 1.         1.         0.99862083 1.         1.         1.
 1.         1.         1.         0.99862083 1.         1.
 1.         1.         1.         1.         0.99793124 1.
 1.         1.         1.         1.         1.         0.99862083
 1.         1.         1.         1.         1.         1.
 0.99862083 1.         1.         1.         1.         1.
 1.         0.99793124 1.         1.         1.         1.
 1.         1.         0.99862083 1.         1.         1.
 1.         1.         1.        ]
```

In [58]:

```
results_RF = pd.DataFrame(gs_RF.cv_results_['params'])
results_RF['test_score'] = gs_RF.cv_results_['mean_test_score']
results_RF.columns

alt.Chart(results_RF,
          title='RF Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('n_estimators', title='n_estimators'),
    alt.Y('test_score', title='Mean CV Score', aggregate='average', scale=alt.Scale(zero=Fa
    color='max_features'
)
```

Out[58]:



The RF Performance Comparison graph above shows that for `max_features` they all follow the same path when the `n_estimators` increase. The best paramaters for the Random forest algorithm the following are the best parameters: `max_features: auto, min_samples_leaf: 1, n_estimators: 50`

4.C Prediction evaluation

Now that we done feature selection and hyperparameter tuning we are ready to evaluate how well the algorithms will run under our parameters. As a reminder we have decided to use the full features for each algorithm.

In [59]:

```
cv_results_KNN = cross_val_score(estimator=gs_KNN.best_estimator_,
                                 X=D_test,
                                 y=t_test,
                                 cv=cv_method,
                                 n_jobs=-2,
                                 scoring=scoring_metric)
cv_results_KNN.mean()
```

Out[59]:

```
0.9857142857142858
```

In [60]:

```
cv_results_DT = cross_val_score(estimator=gs_DT.best_estimator_,
                                X=D_test,
                                y=t_test,
                                cv=cv_method,
                                n_jobs=-2,
                                scoring=scoring_metric)
cv_results_DT.mean()
```

Out[60]:

```
0.9857142857142858
```

In [61]:

```
cv_results_RF = cross_val_score(estimator=gs_RF.best_estimator_,
                                X=D_test,
                                y=t_test,
                                cv=cv_method,
                                n_jobs=-2,
                                scoring=scoring_metric)
cv_results_RF.mean()
```

Out[61]:

```
1.0
```

using the scoring method of `roc_auc` we can see all the algorithms scored extremely high. Interestingly enough the KNN algorithm scored exactly the same as the Decision tree. also, the Random forest algorithm scored `1.0` which is kind of unusual.

We will do some further evaluations. We have used the same random stat throughout the hyperparameter tuning so we are fitting and traing on the exact same data throughout. we can use a paired t-test to see if either of the algorithms are significantly better than the other for the dataset.

In [62]:

```
print(stats.ttest_rel(cv_results_KNN, cv_results_DT))
print(stats.ttest_rel(cv_results_KNN, cv_results_RF))
print(stats.ttest_rel(cv_results_DT, cv_results_RF))
```

```
Ttest_relResult(statistic=nan, pvalue=nan)
Ttest_relResult(statistic=-1.0, pvalue=0.373900966300059)
Ttest_relResult(statistic=-1.0, pvalue=0.373900966300059)
```

Because KNN and the Decision tree both achieved the same score we receive the `nan` output so they can't be compared. As for the Random forest though we can see that the output is higher than 0.05 therefore Random forest does not have a statistically significant difference of the other two. this test shows none of the algorithms are better than the other.

In [63]:

```
pred_KNN = gs_KNN.predict(D_test)
pred_DT = gs_DT.predict(D_test)
pred_RF = gs_RF.predict(D_test)
```

4.C.a Confusion matrix

In [64]:

```
print("\nConfusion matrix for K-Nearest Neighbor")
print(metrics.confusion_matrix(t_test, pred_KNN))
print("\nConfusion matrix for Decision Tree")
print(metrics.confusion_matrix(t_test, pred_DT))
print("\nConfusion matrix for Random Forest")
print(metrics.confusion_matrix(t_test, pred_RF))
```

```
Confusion matrix for K-Nearest Neighbor
[[44  0]
 [ 0 31]]

Confusion matrix for Decision Tree
[[44  0]
 [ 1 30]]

Confusion matrix for Random Forest
[[44  0]
 [ 0 31]]
```

Our confusion matrices show that for KNN and Random forest it has managed to correctly predict all the true positives (`44`) and true negatives (`31`). The Decision tree on the other hand managed to have `1` observation predicted incorrectly, it predicted it as a false negative, meaning it predicted a case where the patient did have chronic kidney disease but it said that they didn't.

We'll look further into this in the Classification report below.

4.C.b Classification report

In [65]:

```
print("\nClassification report for K-Nearest Neighbor")
print(metrics.classification_report(t_test, pred_KNN))
print("\nClassification report for Decision Tree")
print(metrics.classification_report(t_test, pred_DT))
print("\nClassification report for Random Forest")
print(metrics.classification_report(t_test, pred_RF))
```

```
Classification report for K-Nearest Neighbor
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        44
           1       1.00      1.00      1.00        31

   micro avg       1.00      1.00      1.00        75
   macro avg       1.00      1.00      1.00        75
weighted avg       1.00      1.00      1.00        75


Classification report for Decision Tree
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        44
           1       1.00      0.97      0.98        31

   micro avg       0.99      0.99      0.99        75
   macro avg       0.99      0.98      0.99        75
weighted avg       0.99      0.99      0.99        75


Classification report for Random Forest
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        44
           1       1.00      1.00      1.00        31

   micro avg       1.00      1.00      1.00        75
   macro avg       1.00      1.00      1.00        75
weighted avg       1.00      1.00      1.00        75
```

So as we can see in the classification report for both KNN and Random forest the precision, recall and f1-score are `1` meaning it was able to predict with correctly patients with chronic kidney disease and those without and not misclassify any, which is what we saw in the confusion matrix for both of them. the Decision tree on the other hand only received a `0.98` on precision and a recall of `0.97` this due to the misclassification of the observation of a nonchronic kidney disease patient as someone who did have it.

4.C.c ROC curve

At the moment it seems unusual to score so high on these test butt we can continue on with further evaluation. We can use a ROC curve but as the algorithms are already scoring so high (and no false positives) we may not get any more useful information out of it.

In [66]:

```
t_prob = gs_KNN.predict_proba(D_test)
t_prob[0:10]
```

Out[66]:

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

In [67]:

```
fpr, tpr, _ = metrics.roc_curve(t_test, t_prob[:, 1])
roc_auc = metrics.auc(fpr, tpr)
roc_auc
```

Out[67]:

```
1.0
```

In [68]:

```
df = pd.DataFrame({'fpr': fpr, 'tpr': tpr})
df
```

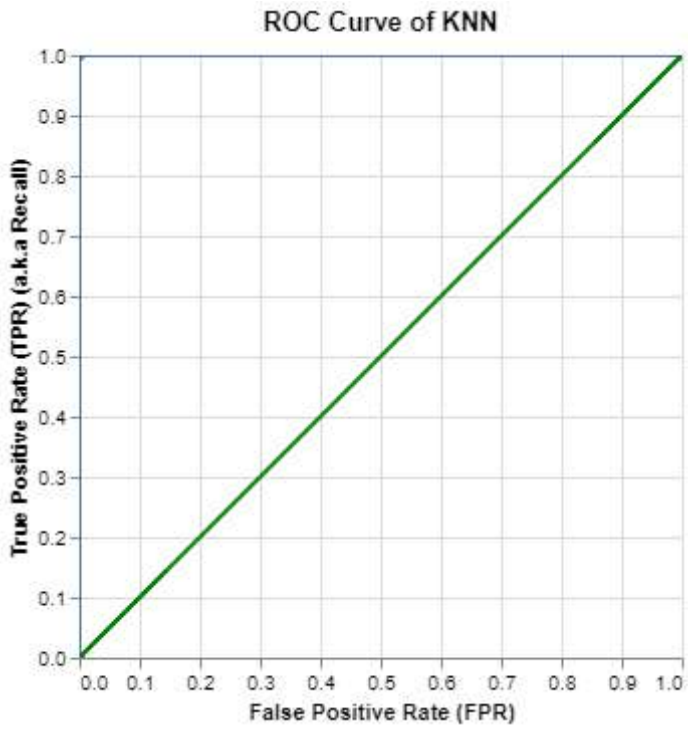
Out[68]:

	fpr	tpr
0	0.0	0.0
1	0.0	1.0
2	1.0	1.0

In [69]:

```
base = alt.Chart(df,
                 title='ROC Curve of KNN'
                ).properties(width=300)

roc_curve = base.mark_line(point=True).encode(
    alt.X('fpr', title='False Positive Rate (FPR)',  sort=None),
    alt.Y('tpr', title='True Positive Rate (TPR) (a.k.a Recall)'),
)

roc_rule = base.mark_line(color='green').encode(
    x='fpr',
    y='fpr',
    size=alt.value(2)
)

(roc_curve + roc_rule).interactive()
```

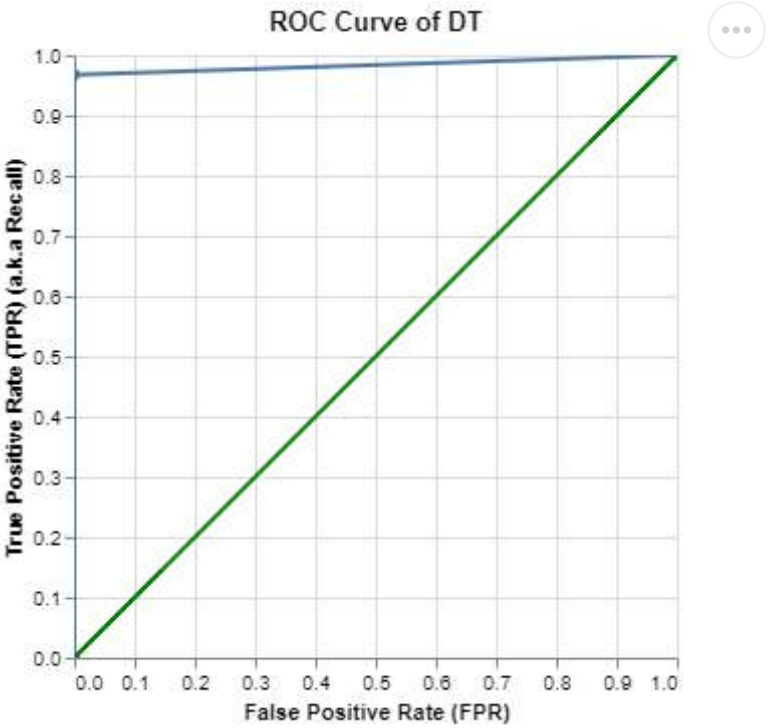
Out[69]:



In [70]:

```
t_prob = gs_DT.predict_proba(D_test)
fpr, tpr, _ = metrics.roc_curve(t_test, t_prob[:, 1])
roc_auc = metrics.auc(fpr, tpr)
df = pd.DataFrame({'fpr': fpr, 'tpr': tpr})

base = alt.Chart(df,
                 title='ROC Curve of DT'
                ).properties(width=300)

roc_curve = base.mark_line(point=True).encode(
    alt.X('fpr', title='False Positive Rate (FPR)',  sort=None),
    alt.Y('tpr', title='True Positive Rate (TPR) (a.k.a Recall)'),
)

roc_rule = base.mark_line(color='green').encode(
    x='fpr',
    y='fpr',
    size=alt.value(2)
)

(roc_curve + roc_rule).interactive()
```

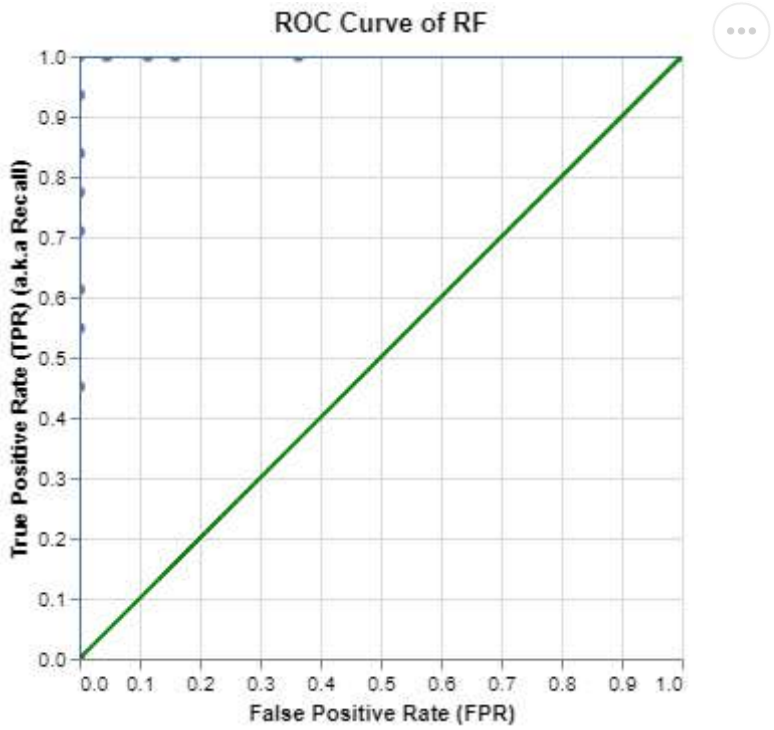
Out[70]:



In [71]:

```
t_prob = gs_RF.predict_proba(D_test)
fpr, tpr, _ = metrics.roc_curve(t_test, t_prob[:, 1])
roc_auc = metrics.auc(fpr, tpr)
df = pd.DataFrame({'fpr': fpr, 'tpr': tpr})

base = alt.Chart(df,
                 title='ROC Curve of RF'
                ).properties(width=300)

roc_curve = base.mark_line(point=True).encode(
    alt.X('fpr', title='False Positive Rate (FPR)',  sort=None),
    alt.Y('tpr', title='True Positive Rate (TPR) (a.k.a Recall)'),
)

roc_rule = base.mark_line(color='green').encode(
    x='fpr',
    y='fpr',
    size=alt.value(2)
)

(roc_curve + roc_rule).interactive()
```

Out[71]:



As predicted the ROC curve is basically a right angle due to the lack of false positives. The only one different is

the decision tree. If we didn't have any suspicions of out algorithms this would show that they are working extraordinary well.

5. Conclusion

The chronic kidney disease dataset we use needed extensive data cleaning. There were many missing values that needed to be dealt with. We looked at **complete case analysis** were we would remove observations that has any missing values but as we were required for the assignment to have a minimum of at least 200 observations we opted only to remove observations that had missing values in the `rbc` feature. This was chosen as it had the most missing values (`38%`). With the rest of the missing values they were wither replaced with the median for numerical features and the mode for categorical ones.

After some quick data exploration, we discovered that patients with higher hemoglobulin counts generally didn't have chronic kidney disease and also the same was discovered when patients had a high hemoglobulin count as well as a higher packed cell count.

Moving on to predictive modelling, we used Random forest importance for feature selection. the top ten features were `al` , `pcv` , `sg` , `sc` , `hemo` , `bu` , `htn` , `pc` , `dm` , `bgr` . We compared this with the full set of features using the `DecisionTreeClassifier` a wrapper to assess the performance and found they performed equally as well (`0.986`) when using the `roc_auc` scoring method. With this result we decided to use the full set with the hyperparameter tuning and predicting.

As for tuning the parameters we found the best parameters for the following three algorithms where as follows:

-KNN: 1 neighbor Euclidean distance

-Decision tree: gini criterion Max depth of 1 2 min samples split

-Random forest: Max features was auto 1 Min samples leaf 50 estimators

The data was trained on a 70:30 split and it was found that both the KNN and Random forest algorithms where predicting perfectly no false positives or false negatives, whereas the decision tree scored 1 false negative. This may be due to the models overfitting the data. This ay be due to the dataset being so small, therefore more data, like in most cases would help to see if the models are being overfitted and still able to predicate with such accuracy, which is highly doubted. It may also have to do with the data cleaning as in most cases it would have been preferred to remove all the observations with missing values rather than replace them.

If for some reason these models where this accurate KNN should be chosen just as it runs faster than the random forest algorithm. and the decision tree shouldn't be chosen as it was the only one to misclassify an observation. the misclassification on its own isn't a problem it more that it misclassified someone who had chronic kidney disease as someone who didn't which would be a worse outcome than the opposite prediction.

6. References

1. Lichman, M. (2013). UCI Machine Learning Repository [online]. Available at https://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease (https://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease) [Accessed 31/05/2020]
2. Kelleher J, Namee B, D'Arcy A. (2015) Fundamentals of machine learning for preditive data analytics. Massachusetts Institute of Technology.
3. Aksakalli, V. RMIT [online]. Available at https://rmit.instructure.com/courses/67061/pages/assignment-1-materials?module_item_id=2327584 (https://rmit.instructure.com/courses/67061/pages/assignment-1-materials?module_item_id=2327584) [Accessed 31/05/2020]

4. Aksakalli, V. RMIT [online]. Available at https://rmit.instructure.com/courses/67061 (https://rmit.instructure.com/courses/67061) [Accessed 31/05/2020]
5. Aksakalli, V. feature ranking [online]. Available at www.featureranking.com (http://www.featureranking.com) [Accessed 31/05/2020]