Machine Learning Project – Predict the Age of Abalone

Thesis · June 2019

DOI: 10.13140/RG.2.2.21738.88009

1 author:

Yizhen Han
RMIT University
3 PUBLICATIONS **0** CITATIONS

SEE PROFILE

Chapter 1

Introduction

1.1 Objective

The objective of this project is to predicting the age of abalone from physical measurements using the 1994 abalone data "The Population Biology of Abalone (*Haliotis* species) in Tasmania. I. Blacklip Abalone (*H. rubra*) from the North Coast and Islands of Bass Strait". The data sets were sourced from the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/datasets/Abalone (http://archive.ics.uci.edu/ml/datasets/Abalone). This report is phase 2 of the project, which include model specification and model selection. Chapter 1 is introduction, chapter 2 is methodology, chapter 3 is data preparation, chapter 4 is feature selection, chapter 5 is about hyperparameter tuning, chapter 6 is model comparison, chapter 7 is limitations, chapter 8 presents a brief summary.

1.2 Data Sets

The UCI Machine Learning Repository provides one dataset abalone.data, it contains 4177 observations, 8 descriptives features and 1 target feature.

1.2.1 Target Feature

The target feature is the rings of abalone. It is an integer to describe the age of abalone, number of rings add 1.5 gives the age in years of them.

1.2.2 Descriptive Features

The variable description is produced here from abalone.names file:

Name / Data Type / Measurement Unit / Description

Sex / nominal / -- / M, F, and I (infant)

Length / continuous / mm / Longest shell measurement

Diameter / continuous / mm / perpendicular to length

Height / continuous / mm / with meat in shell

Whole weight / continuous / grams / whole abalone

Shucked weight / continuous / grams / weight of meat

Viscera weight / continuous / grams / gut weight (after bleeding)

Shell weight / continuous / grams / after being dried

Rings / integer / -- / +1.5 gives the age in years

Chapter 2

Methodology

In this chapter, we build the a series of classifiers to predict the target feature. In this report, target is the 'Rings' of abalone, which is a discrete integer from 1 to 29. In most cases, the target is treated as a classification problem, but due to its particularity, regression methods are also in our consideration.

For classification models, we use following classifier:

- K-Nearest Neighbors (KNN),
- Decision trees (DT),
- Random Forest (RF),
- Gaussian Naive Bayes (NB),
- Multi-layer Perceptron (MLP)

For regression models, algorithems are as follows:

- Linear Regression (LR),
- Logistic Regression(Logit).

Before model selection, we checked our dataset is cleaned in phase 1. Our modeling strategy begins are as follows:

1. Encoding categorical feature to numerical feature.
2. Scaling them based on different algorithm we used.

For KNN, MLP, we will use min-max scaler; for Gaussian NB, we use power transformation; for DT and RF, although it's not necessary to scale, we use min-max scaler in order to compare accuracy with other models.

1. Feature selection for all features.

F-score, mutual information and random forest method were used to select features. A paired t-test were performed on the highest accuracy obtained by each feature selection method and the number of input variables in the corresponding 1-NN model. See if any performance difference is statistically significant

1. Using selected feature to fit each model.

Using feature selection together with hyperparameter grid search, we conduct a 5-fold stratified cross-validation and 3 repetitions to fine-tune hyperparameters of each classifier using model accuracy as the performance metric.

1. Performance comparison.

After using hyperparameter search to find the best model for each algorithm, we conduct a 10-fold cross-validation on the test data and perform a paired t-test to see if any performance difference is statistically significant. In addition, we compare the classifiers with respect to their classification reports and confusion matrices on the test data.

Chapter 3

Data Preparation

In [1]:

```
import urllib
from IPython.display import display, HTML

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

import altair as alt
alt.renderers.enable('notebook')
import warnings
warnings.filterwarnings('ignore')

from sklearn import preprocessing
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.metrics import mean_squared_error
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression, LogisticRegressionCV
from sklearn.model_selection import train_test_split, KFold, cross_val_score, RepeatedStratifie
dKFold, StratifiedKFold, GridSearchCV
from sklearn import feature_selection as fs
from sklearn.preprocessing import PowerTransformer
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from scipy import stats
```

In [2]:

```
#Load data
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data"
set1 = urllib.request.Request(url)
abalone_p = urllib.request.urlopen(set1)
```

In [3]:

```
#Load columns
abalone = pd.read_csv(abalone_p, header = None, names =['Sex', 'Length', 'Diameter', 'Height',
'Whole weight', 'Shucked weight', 'Viscera weight', 'Shell weight', 'Rings'])
```

In [4]:

```
#Check data types
abalone.dtypes
```

Out[4]:

```
Sex                object
Length            float64
Diameter          float64
Height            float64
Whole weight      float64
Shucked weight    float64
Viscera weight    float64
Shell weight      float64
Rings               int64
dtype: object
```

In [5]:

```
#A brief view
abalone.head(3)
```

Out[5]:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.15	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.07	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.21	9

In [6]:

```
#Make a copy of original dataset and use the new dataset called "abl" for following steps.
abl = abalone.copy()
```

In [7]:

```
#It seems that no attributes contain NaN values, check it to make sure.
abl.isnull().sum(axis=0)
```

Out[7]:

```
Sex               0
Length            0
Diameter          0
Height            0
Whole weight      0
Shucked weight    0
Viscera weight    0
Shell weight      0
Rings             0
dtype: int64
```

In [8]:

```
#Show descriptive statistic table
display(HTML('<b>Table 1: Summary of continuous features</b>'))
display(abl.describe(include = ['int64','float64']))
```

Table 1: Summary of continuous features

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.(
mean	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.:
std	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.'
min	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.(
25%	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.'
50%	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.:
75%	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.3
max	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.(

Based on tables above, we have a clear view of the whole data. Since the dataset is clean, we will use this for following machine learning process.

one-hot-encoding

We remove the "Rings" feature from the full dataset and call it "target". The rest of the features are the descriptive features called "abl_all".

In [9]:

```
abl_all = abl.drop(columns="Rings")
target = abl['Rings']
```

In [10]:

```
abl_all = pd.get_dummies(abl_all)
abl_all.columns
```

Out[10]:

```
Index(['Length', 'Diameter', 'Height', 'Whole weight', 'Shucked weight',
       'Viscera weight', 'Shell weight', 'Sex_F', 'Sex_I', 'Sex_M'],
      dtype='object')
```

In [11]:

```
abl_all.shape
```

Out[11]:

```
(4177, 10)
```

In [12]:

```
target.value_counts()
```

Out[12]:

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

As we can see the classes of target are not balanced.

Scaling

In [13]:

```
abl_unscaled=abl_all.values
abl_minmax = preprocessing.MinMaxScaler().fit_transform(abl_unscaled)
```

Chapter 4

Feature selection

In the feature selection, we will use min-max scaled data. Since there are only 10 features, we will look at them all in the feature selection to see there importance. We will use F-Score(FS), Mutual Information(MI) and Random Forest Importance(RFI) to get ranking of features. For each feature selection ranking, the top-9 features selected by the filters are incrementally fed to the 1-NN algorithm using the scaled dataset. The highest accuracy obtained by each method and the number of input variablesin the corresponding 1-NN model are shown after feature selection. Then we run a paired t-test on three results. Features will be selected by the method with best accuracy.

For comparison, we also do a performance with full set of features.

Performance with Full Set of Features

In [14]:

```
clf = KNeighborsClassifier(n_neighbors=1)
```

In [15]:

```
cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=3,
                                    random_state=999)
```

In [16]:

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

In [17]:

```
cv_results_full
```

Out[17]:

```
array([0.1964497 , 0.21190476, 0.2011976 , 0.20843373, 0.18016929,
       0.20236686, 0.21309524, 0.1882494 , 0.20528211, 0.21090909,
       0.18623962, 0.20570749, 0.21052632, 0.20192308, 0.22181818])
```

In [18]:

```
cv_results_full.mean().round(4)
```

Out[18]:

0.203

Feature Selection Using F-Score

Since there are only 10 features, we will look at them all in the feature selection to see there importance. We will use F-Score(FS), Mutual Information(MI) and Random Forest Importance(RFI) to get ranking of features. During the hyperparameter tuning phase, we will include all the three feature selection method and search top 6 based on their importance.

In [19]:

```
num_features = 10
```

In [20]:

```
fs_fit_fscore = fs.SelectKBest(fs.f_classif, k=num_features)
fs_fit_fscore.fit_transform(abl_minmax, target)
fs_indices_fscore = np.argsort(fs_fit_fscore.scores_)[::-1][0:num_features]
fs_indices_fscore
```

Out[20]:

```
array([1, 0, 6, 2, 3, 5, 4, 8, 7, 9], dtype=int64)
```

In [21]:

```
abl_all.dtypes
```

Out[21]:

```
Length            float64
Diameter          float64
Height            float64
Whole weight      float64
Shucked weight    float64
Viscera weight    float64
Shell weight      float64
Sex_F               uint8
Sex_I               uint8
Sex_M               uint8
dtype: object
```

In [22]:

```
best_features_fscore = abl_all.columns[fs_indices_fscore].values
best_features_fscore
```

Out[22]:

```
array(['Diameter', 'Length', 'Shell weight', 'Height', 'Whole weight',
       'Viscera weight', 'Shucked weight', 'Sex_I', 'Sex_F', 'Sex_M'],
      dtype=object)
```

In [23]:

```
feature_importances_fscore = fs_fit_fscore.scores_[fs_indices_fscore]
feature_importances_fscore
```

Out[23]:

```
array([196.43610528, 188.50942458, 147.23694566, 124.42771441,
       113.71233973, 103.72143783,  80.79767682,  68.87047717,
        15.78351983,  10.62934022])
```

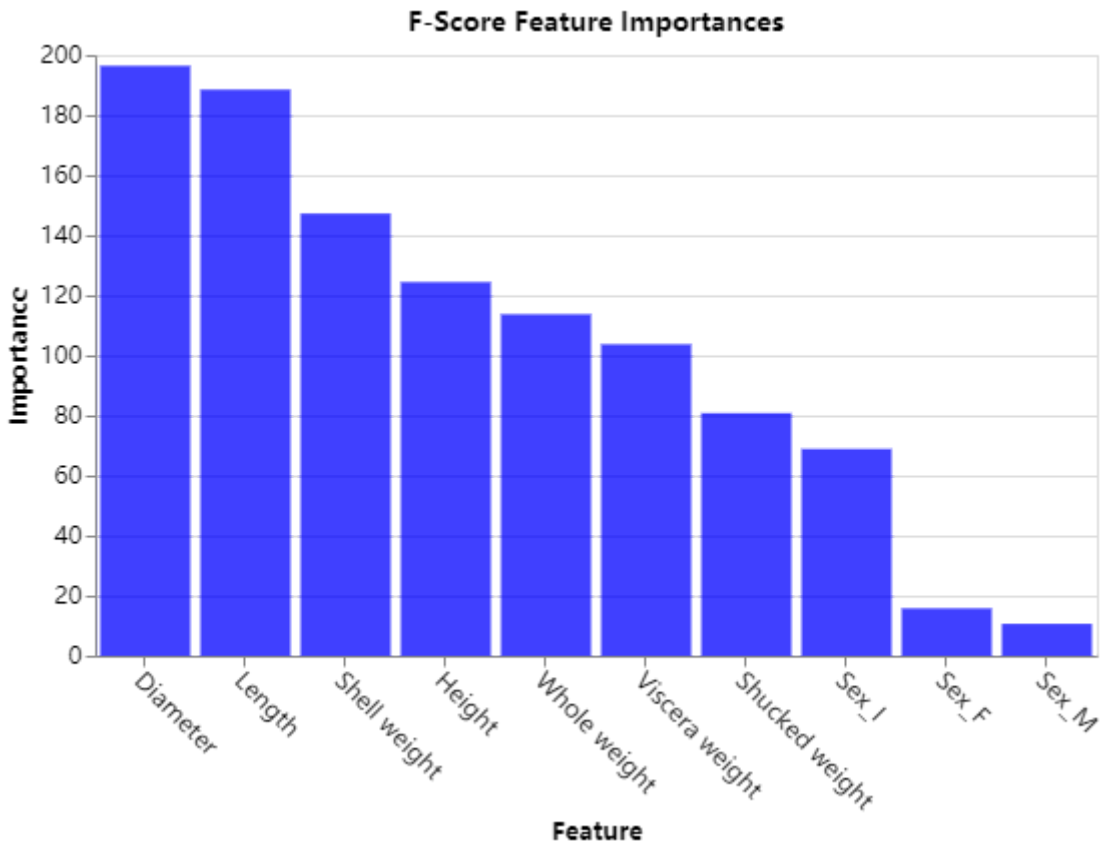
In [24]:

```
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
```

In [25]:

```
plot_imp(best_features_fscore, feature_importances_fscore, 'F-Score', 'blue')
```

Out[25]:



In [26]:

```
cv_results_fscore = cross_val_score(estimator=clf,
                                    X=abl_minmax[:, fs_indices_fscore[:8]],
                                    y=target,
                                    cv=cv_method,
                                    scoring=scoring_metric)
cv_results_fscore.mean().round(4)
```

Out[26]:

0.2037

Feature Selection Using Mutual Information

In [27]:

```
fs_fit_mutual_info = fs.SelectKBest(fs.mutual_info_classif, k=num_features)
fs_fit_mutual_info.fit_transform(abl_minmax, target)
fs_indices_mutual_info = np.argsort(fs_fit_mutual_info.scores_)[::-1][0:num_features]
best_features_mutual_info = abl_all.columns[fs_indices_mutual_info].values
best_features_mutual_info
```

Out[27]:

```
array(['Shell weight', 'Diameter', 'Whole weight', 'Viscera weight',
       'Length', 'Height', 'Shucked weight', 'Sex_I', 'Sex_F', 'Sex_M'],
      dtype=object)
```

In [28]:

```
feature_importances_mutual_info = fs_fit_mutual_info.scores_[fs_indices_mutual_info]
feature_importances_mutual_info
```

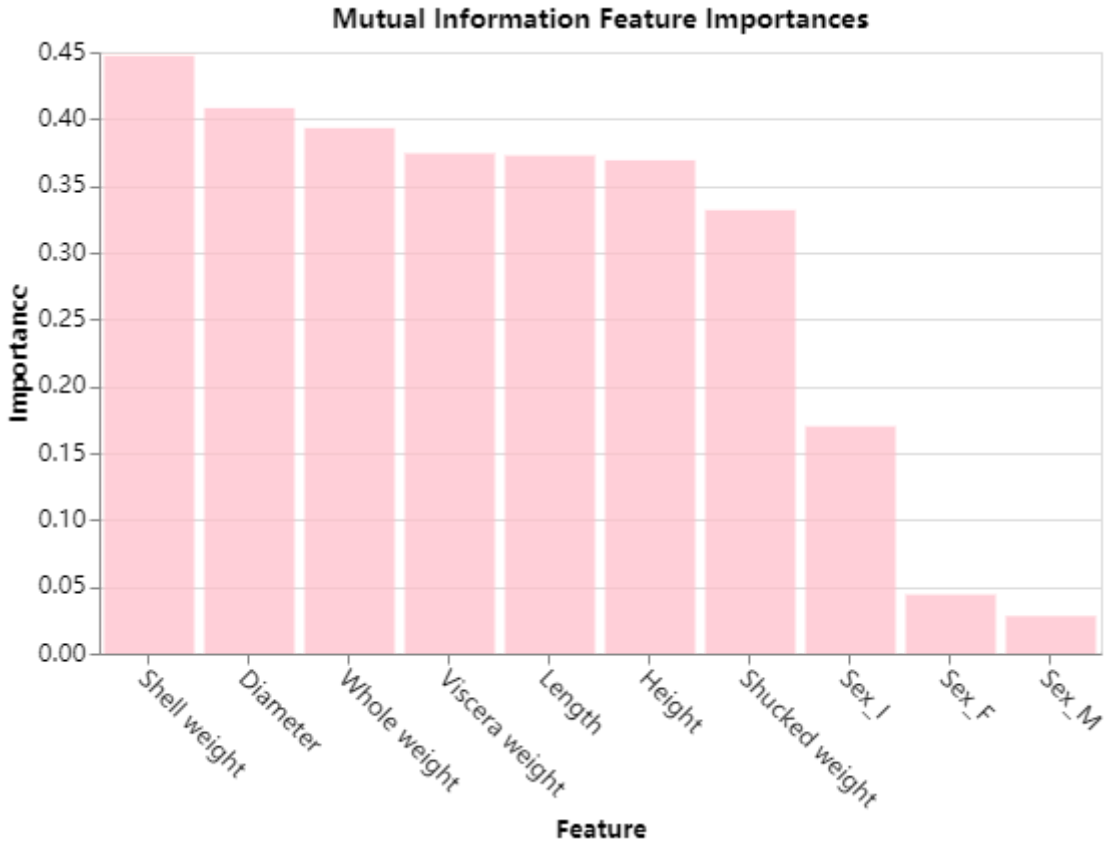
Out[28]:

```
array([0.44705293, 0.40807574, 0.39319245, 0.37425451, 0.37260109,
       0.36910541, 0.33180424, 0.16997897, 0.04406357, 0.02802034])
```

In [29]:

```
plot_imp(best_features_mutual_info, feature_importances_mutual_info, 'Mutual Information', 'pin
k')
```

Out[29]:



In [30]:

```
cv_results_mutual_info = cross_val_score(estimator=clf,
                            X=abl_minmax[:, fs_indices_mutual_info[:8]],
                            y=target,
                            cv=cv_method,
                            scoring=scoring_metric)
cv_results_mutual_info.mean().round(4)
```

Out[30]:

0.2037

Feature Selection Using Random Forest Importance¶

In [31]:

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

In [32]:

```
best_features_rfi = abl_all.columns[fs_indices_rfi].values
best_features_rfi
```

Out[32]:

```
array(['Shell weight', 'Shucked weight', 'Viscera weight', 'Whole weight',
       'Length', 'Diameter', 'Height', 'Sex_M', 'Sex_F', 'Sex_I'],
      dtype=object)
```

In [33]:

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

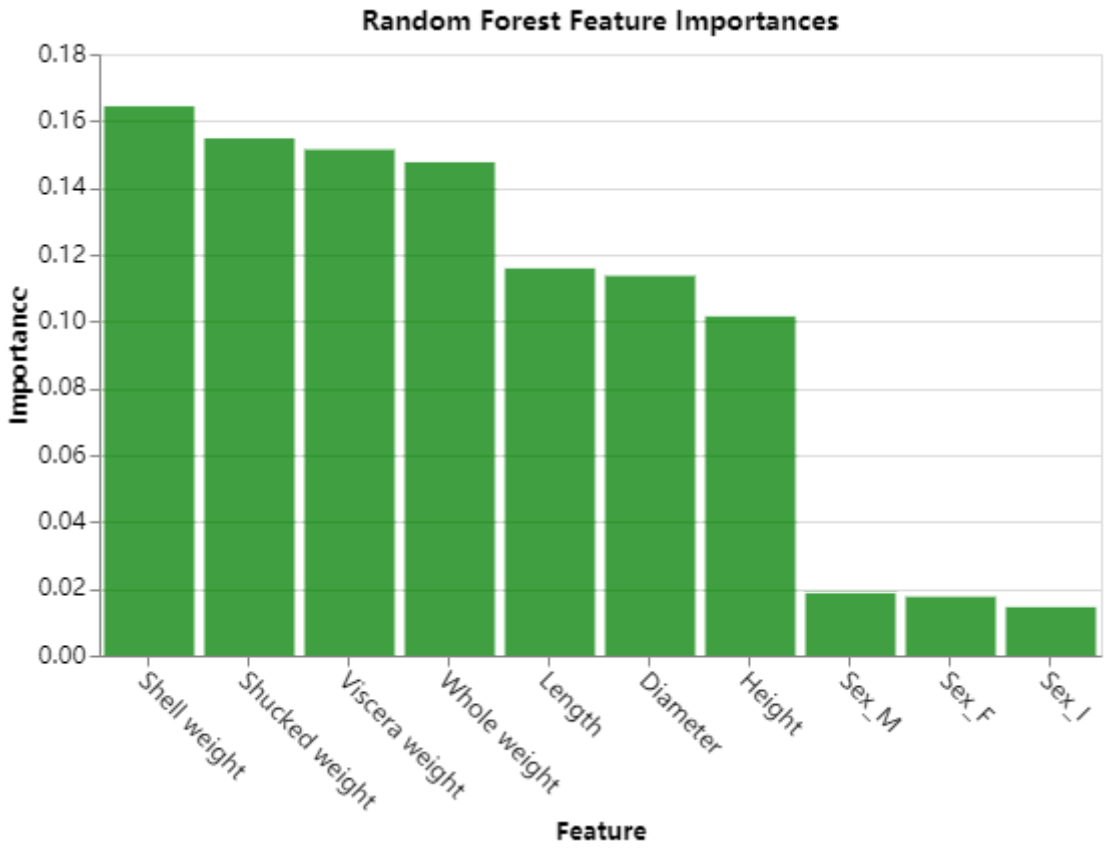
Out[33]:

```
array([0.16437818, 0.15479046, 0.15144102, 0.14764554, 0.11588212,
       0.11363829, 0.10149196, 0.01865731, 0.01758714, 0.01448798])
```

In [34]:

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

Out[34]:



In [35]:

```
cv_results_rfi = cross_val_score(estimator=clf,
                                 X=abl_minmax[:, fs_indices_rfi[:5]],
                                 y=target,
                                 cv=cv_method,
                                 scoring=scoring_metric)
cv_results_rfi.mean().round(4)
```

Out[35]:

0.2099

Feature selection comparison

Based on all feature selection, best results obtained by feeding top-ranked features as input to 1-NN are as follows:

	Number of features	Results
* Filter	8	0.2037
* FS	8	0.2037
* MI	5	0.2099

Then do a paired t-test to compare three method.

In [36]:

```
print(stats.ttest_rel(cv_results_rfi, cv_results_fscore))
print(stats.ttest_rel(cv_results_rfi, cv_results_mutual_info))
print(stats.ttest_rel(cv_results_mutual_info, cv_results_fscore)) # they are equal
```

```
Ttest_relResult(statistic=1.7643437698120972, pvalue=0.09947130549587538)
Ttest_relResult(statistic=1.7643437698120972, pvalue=0.09947130549587538)
Ttest_relResult(statistic=nan, pvalue=nan)
```

A p-value smaller than 0.05 indicates a statistically significant difference. Looking at these results, p-value less than 0.10, we can conclude that at a 90% significance level, feature selection by RFI have the best cross-validation results. In this case, we will use the top 5 feature based on RFI. We also do a paired t-test for the cross-validation results of RFI feature combination and full set.

In [37]:

```
print(stats.ttest_rel(cv_results_rfi, cv_results_full))
```

```
Ttest_relResult(statistic=2.006787672266217, pvalue=0.06448466181759005)
```

As we can see, at a 90% significance level, feature selection by RFI is also better than result of full set. Based on all above, we will choose top 5 feature in RFI feature selection.

In [38]:

```
abl_all.columns[fs_indices_rfi[:5]]
```

Out[38]:

```
Index(['Shell weight', 'Shucked weight', 'Viscera weight', 'Whole weight',
       'Length'],
      dtype='object')
```

In [39]:

```
abl_minmax_df = pd.DataFrame(data = abl_minmax,columns = abl_all.columns)
```

In [40]:

```
abl_rfi = abl_minmax_df[abl_all.columns[fs_indices_rfi[:5]].values]
```

In [41]:

```
abl_rfi = abl_rfi.values
```

Chapter 5

Hyperparameter tuning

k-NN

First, we need to define a dictionary of KNN parameters for the grid search. The KNN hyperparameters are as follows:

- number of neighbors (n_neighbors) and
- the distance metric p.

We will consider K values from 2 to a relatively large number 64 (which is the largest integer less than the square root of observation numbers), and p values of 1 (Manhattan) and 2 (Euclidean). Second, we pass the KNeighborsClassifier() and KNN_params as the model and the parameter dictionary into the GridSearchCV function. In addition, we include the repeated stratified CV method we defined previously (cv=cv_method). Also, we tell sklearn which metric to optimize, which is accuracy in our example (scoring='accuracy', refit='accuracy'). We will use the same grid search methodology for NB, DT, RF and MLP.

In [42]:

```
cv_method = RepeatedStratifiedKFold(n_splits=10,
                                    n_repeats=3,
                                    random_state=4)
```

In [43]:

```
params_KNN = {'n_neighbors': [2, 4, 8, 16, 32, 64],  # largest k should be less than sqrt(n) = sqrt(4177) which is 64
              'p': [1, 2]}
```

In [44]:

```
gs_KNN = GridSearchCV(estimator=KNeighborsClassifier(),
                      param_grid=params_KNN,
                      cv=cv_method,
                      verbose=1,  # verbose: the higher, the more messages
                      scoring='accuracy',
                      return_train_score=True)
```

In [45]:

```
gs_KNN.fit(abl_rfi, target);
```

```
Fitting 30 folds for each of 12 candidates, totalling 360 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 360 out of 360 | elapsed:  1.2min finished
```

In [46]:

```
gs_KNN.best_params_
```

Out[46]:

```
{'n_neighbors': 64, 'p': 1}
```

In [47]:

```
gs_KNN.best_score_
```

Out[47]:

```
0.2670975979570665
```

In [48]:

```
results_KNN = pd.DataFrame(gs_KNN.cv_results_['params'])
```

In [49]:

```
results_KNN['test_score'] = gs_KNN.cv_results_['mean_test_score']
```

In [50]:

```
results_KNN['metric'] = results_KNN['p'].replace([1,2,3], ["Manhattan", "Euclidean", "Minkowski"
])
results_KNN
```

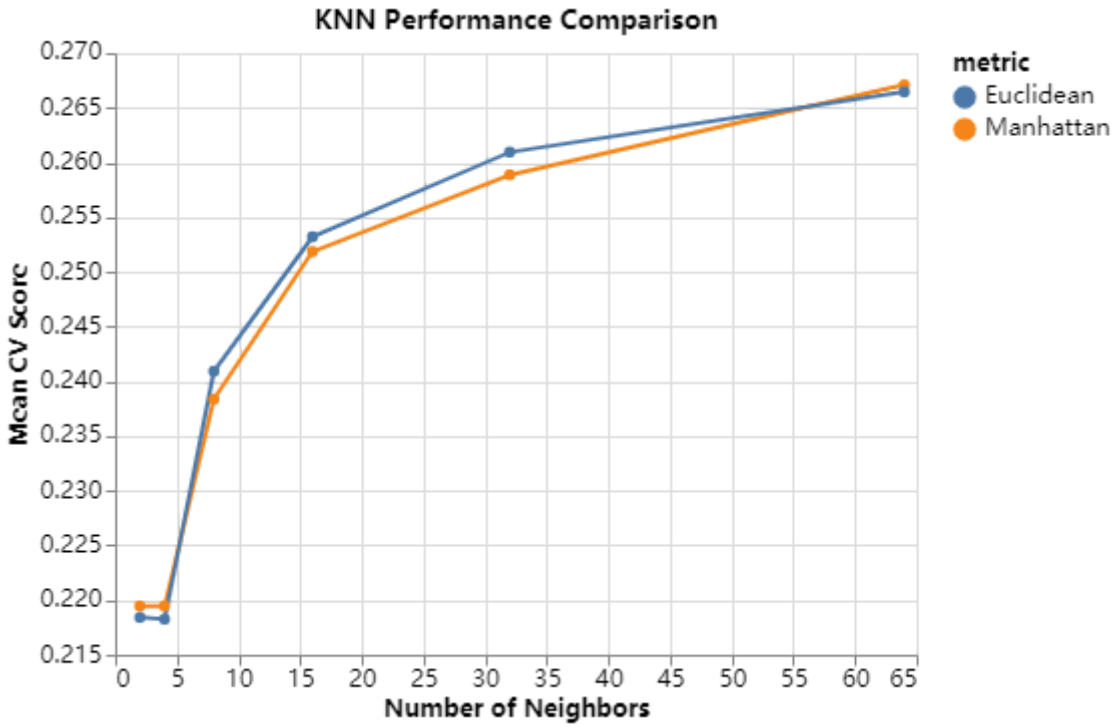
Out[50]:

	n_neighbors	p	test_score	metric
0	2	1	0.219456	Manhattan
1	2	2	0.218418	Euclidean
2	4	1	0.219456	Manhattan
3	4	2	0.218259	Euclidean
4	8	1	0.238369	Manhattan
5	8	2	0.240923	Euclidean
6	16	1	0.251855	Manhattan
7	16	2	0.253212	Euclidean
8	32	1	0.258878	Manhattan
9	32	2	0.260953	Euclidean
10	64	1	0.267098	Manhattan
11	64	2	0.266459	Euclidean

In [51]:

```
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[51]:



As we can see, the parameters of best model in k-NN algorithm is k = 64, p = 1 with the best accuracy are 0.2671. However, a relatively large k didn't make sense. We will use other model.

Decision tree

In [52]:

```
df_classifier = DecisionTreeClassifier(random_state=4)

params_DT = {'criterion': ['gini', 'entropy'],
             'max_depth': [1, 2, 3, 4, 5, 6, 7, 8],
             'min_samples_split': [2, 3]}
cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=3,
                                    random_state=999)
gs_DT = GridSearchCV(estimator=df_classifier,
                     param_grid=params_DT,
                     cv=cv_method,
                     verbose=1,
                     scoring='accuracy')

gs_DT.fit(abl_rfi, target);
```

```
Fitting 15 folds for each of 32 candidates, totalling 480 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 480 out of 480 | elapsed:   10.7s finished
```

In [53]:

```
gs_DT.best_params_
```

Out[53]:

```
{'criterion': 'gini', 'max_depth': 4, 'min_samples_split': 2}
```

In [54]:

```
gs_DT.best_score_
```

Out[54]:

```
0.26270848296225363
```

In [55]:

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

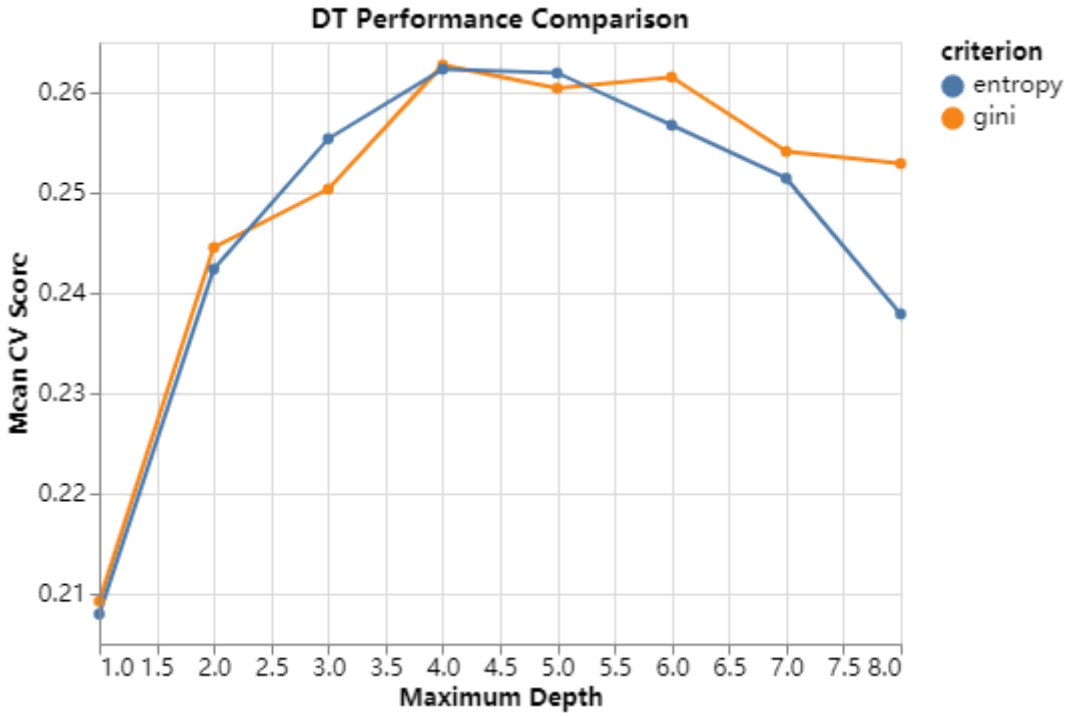
Out[55]:

```
Index(['criterion', 'max_depth', 'min_samples_split', 'test_score'], dtype='objec
t')
```

In [56]:

```
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=False
)),
    color='criterion'
)
```

Out[56]:



As we can see, the parameters of best model in DT algorithm is max_depth = 4, min_samples_split = 2 using gini criterion. The best accuracy are 0.2627.

Random forest

In [57]:

```
df_classifier = RandomForestClassifier(random_state=4)

params_RF = {'criterion': ['gini', 'entropy'],
             'max_depth': [1, 2, 3, 4, 5, 6, 7, 8],
             'min_samples_split': [2, 3]}
cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=3,
                                    random_state=999)
gs_RF = GridSearchCV(estimator=df_classifier,
                     param_grid=params_RF,
                     cv=cv_method,
                     verbose=1,
                     scoring='accuracy')

gs_RF.fit(abl_rfi, target);
```

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

Fitting 15 folds for each of 32 candidates, totalling 480 fits

[Parallel(n_jobs=1)]: Done 480 out of 480 | elapsed:   37.9s finished
```

In [58]:

```
gs_RF.best_params_
```

Out[58]:

```
{'criterion': 'entropy', 'max_depth': 4, 'min_samples_split': 2}
```

In [59]:

```
gs_RF.best_score_
```

Out[59]:

```
0.27531721331098874
```

In [60]:

```
gs_RF.cv_results_['mean_test_score']
```

Out[60]:

```
array([0.21809911, 0.21809911, 0.26135185, 0.26135185, 0.2711675 ,
       0.2711675 , 0.27324236, 0.27324236, 0.26941186, 0.26893305,
       0.26909265, 0.27180592, 0.26446413, 0.26629958, 0.2633469 ,
       0.26374591, 0.21211396, 0.21211396, 0.25528689, 0.25528689,
       0.27196553, 0.27196553, 0.27531721, 0.27531721, 0.27156652,
       0.2708483 , 0.27164632, 0.2709281 , 0.26645918, 0.26653898,
       0.25760115, 0.2633469 ])
```

In [61]:

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

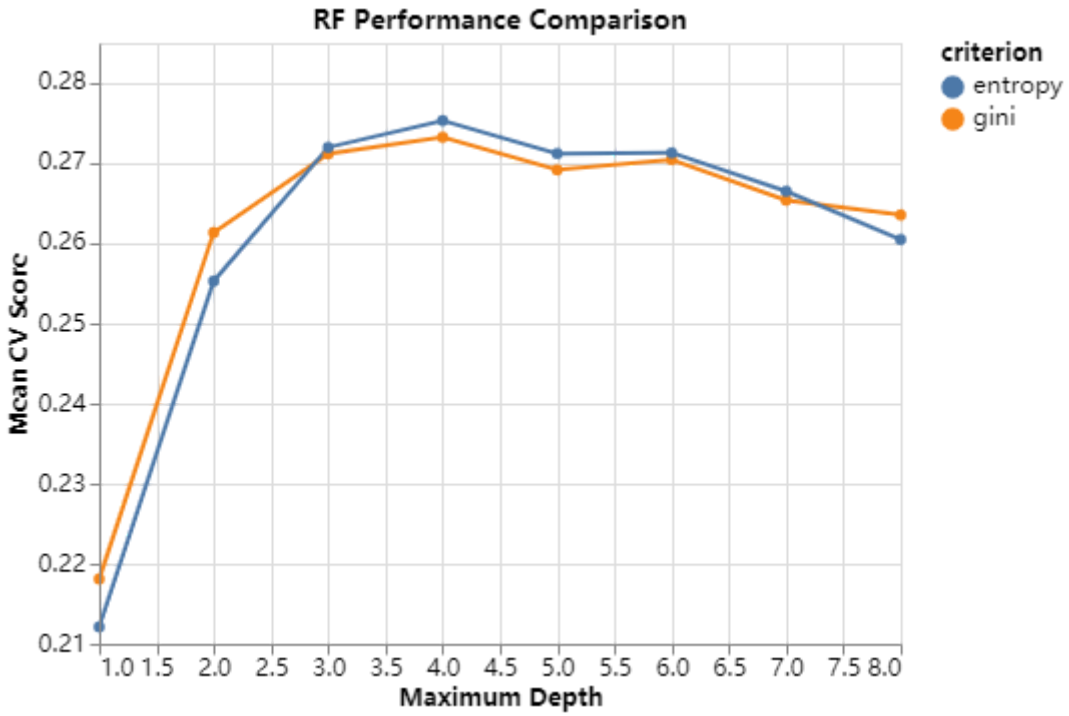
Out[61]:

```
Index(['criterion', 'max_depth', 'min_samples_split', 'test_score'], dtype='objec
t')
```

In [62]:

```
alt.Chart(results_RF,
          title='RF 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=False
)),
    color='criterion'
)
```

Out[62]:



As we can see, the parameters of best model in RF algorithm is max_depth = 4, min_samples_split = 2 using gini criterion. The best accuracy are 0.2730.

GaussianNB

In Gaussian NB, we will conduct the grid search in the "logspace", that is, we will search over the powers of 10. We will start with 10^0 and end with 10^{-9} and we will try 100 different values.

In [63]:

```
np.logspace(0,-9, num=10)
```

Out[63]:

```
array([1.e+00, 1.e-01, 1.e-02, 1.e-03, 1.e-04, 1.e-05, 1.e-06, 1.e-07,
       1.e-08, 1.e-09])
```

Before classifier, we need power transformed data.

In [64]:

```
abl_power = PowerTransformer().fit_transform(abl_unscaled)
```

In [65]:

```
abl_power_df = pd.DataFrame(data = abl_power,columns = abl_all.columns)
```

In [66]:

```
abl_power_rfi = abl_power_df[abl_all.columns[fs_indices_rfi[:5]].values]
abl_power_rfi = abl_power_rfi.values
```

In [67]:

```
nb_classifier = GaussianNB()

params_NB = {'var_smoothing': np.logspace(0,-9, num=100)}

cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=3,
                                    random_state=999)

gs_NB = GridSearchCV(estimator=nb_classifier,
                     param_grid=params_NB,
                     cv=cv_method,
                     verbose=1,
                     scoring='accuracy')

Data_transformed = PowerTransformer().fit_transform(abl_rfi)

gs_NB.fit(abl_power_rfi, target);
```

```
Fitting 15 folds for each of 100 candidates, totalling 1500 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 1500 out of 1500 | elapsed:   12.8s finished
```

In [68]:

```
gs_NB.best_params_
```

Out[68]:

```
{'var_smoothing': 0.8111308307896871}
```

In [69]:

```
gs_NB.best_score_
```

Out[69]:

0.2607932327826989

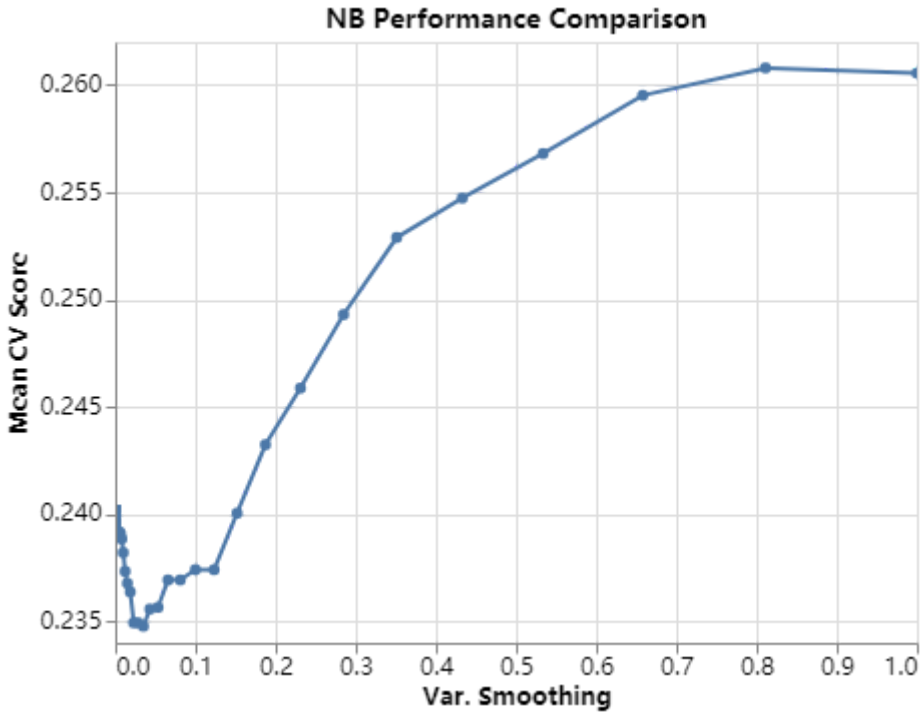
In [70]:

```
results_NB = pd.DataFrame(gs_NB.cv_results_['params'])
results_NB['test_score'] = gs_NB.cv_results_['mean_test_score']
```

In [71]:

```
alt.Chart(results_NB,
          title='NB Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('var_smoothing', title='Var. Smoothing'),
    alt.Y('test_score', title='Mean CV Score', scale=alt.Scale(zero=False))
).interactive()
```

Out[71]:



As we can see, the parameters of best model in RF algorithm is var_smoothing = 0.8111, the best accuracy are 0.2608.

MLP

In this model, we optimizes the log-loss function using LBFGS or stochastic gradient descent. We also use grid search to find best model. However, duel to the longer running time in MLP, we didn't tried large number of hidden_layer_sizes and set max_iter to 100. The output might be improved by adding other possible hyperparameters. Min-max scaled data were used in this section.

In [72]:

```
from sklearn.neural_network import MLPClassifier
df_classifier = MLPClassifier(max_iter = 100)

params_MLP = {'hidden_layer_sizes': [5, 10, 20],
              'alpha': [0.0001, 0.001],
              'solver': ['sgd', 'adam'],
              'learning_rate': ['constant','adaptive']}
cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=3,
                                    random_state=999)
gs_MLP = GridSearchCV(estimator=df_classifier,
                      param_grid=params_MLP,
                      cv=cv_method,
                      verbose=1,
                      scoring='accuracy')

gs_MLP.fit(abl_rfi, target);
```

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

Fitting 15 folds for each of 24 candidates, totalling 360 fits

[Parallel(n_jobs=1)]: Done 360 out of 360 | elapsed:  8.5min finished
```

In [73]:

```
gs_MLP.best_params_
```

Out[73]:

```
{'alpha': 0.0001,
 'hidden_layer_sizes': 20,
 'learning_rate': 'constant',
 'solver': 'adam'}
```

In [74]:

```
gs_MLP.best_score_
```

Out[74]:

```
0.2639055143244753
```

In [75]:

```
results_MLP = pd.DataFrame(gs_MLP.cv_results_['params'])
results_MLP['test_score'] = gs_MLP.cv_results_['mean_test_score']
```

In [76]:

```
results_MLP
```

Out[76]:

	alpha	hidden_layer_sizes	learning_rate	solver	test_score
0	0.0001	5	constant	sgd	0.174368
1	0.0001	5	constant	adam	0.240763
2	0.0001	5	adaptive	sgd	0.175006
3	0.0001	5	adaptive	adam	0.245232
4	0.0001	10	constant	sgd	0.175884
5	0.0001	10	constant	adam	0.253212
6	0.0001	10	adaptive	sgd	0.167744
7	0.0001	10	adaptive	adam	0.254968
8	0.0001	20	constant	sgd	0.185141
9	0.0001	20	constant	adam	0.263906
10	0.0001	20	adaptive	sgd	0.178198
11	0.0001	20	adaptive	adam	0.262389
12	0.0010	5	constant	sgd	0.175485
13	0.0010	5	constant	adam	0.242838
14	0.0010	5	adaptive	sgd	0.172053
15	0.0010	5	adaptive	adam	0.246429
16	0.0010	10	constant	sgd	0.177719
17	0.0010	10	constant	adam	0.256165
18	0.0010	10	adaptive	sgd	0.177001
19	0.0010	10	adaptive	adam	0.258399
20	0.0010	20	constant	sgd	0.183545
21	0.0010	20	constant	adam	0.260235
22	0.0010	20	adaptive	sgd	0.180512
23	0.0010	20	adaptive	adam	0.262389

As we can see, the parameters of best model in MLP algorithm are alpha = 0.0001, hidden_layer_sizes = 20, learning_rate = adaptive, and solver = adam, the best accuracy are 0.2608.

Linear regression

Although this is a regression problem, since there are a large number of levels in our target, we tried to fit it with regression algorithms. To begin with, we use linear regression which is the simplest regression model.

In this model, normal k-fold validation were used. Model will be fitted in each validation.

In [77]:

```
kf = KFold(n_splits=5,random_state=4)
```

In [78]:

```
test = []
prediction = []
model = LinearRegression()

for train_index,test_index in kf.split(abl_rfi):
    X_train,X_test = abl_rfi[train_index],abl_rfi[test_index]
    y_train, y_test = target.values[train_index], target.values[test_index]
    model.fit(X_train, y_train)
    pred = model.predict(X_test)

    prediction += [pred[x] for x in range(len(pred))]
    test += [y_test[x] for x in range(len(y_test))]
```

In [79]:

```
error = mean_squared_error(test,prediction)
```

In [80]:

```
print(error) # mean squared error
print(error**0.5) # root mean square
```

5.49549572604576
2.344247368782945

The RMSE of linear regression is 2.34. However, since our target 'Rings' should be integers, linear models may not be able to simulate raw data very well. Then we use logistic regression.

In [81]:

```
test = []
prediction = []
model = LogisticRegression()


for train_index,test_index in kf.split(abl_rfi):
    X_train,X_test = abl_rfi[train_index],abl_rfi[test_index]
    y_train, y_test = target.values[train_index], target.values[test_index]
    model.fit(X_train, y_train)
    pred = model.predict(X_test)

    prediction += [pred[x] for x in range(len(pred))]
    test += [y_test[x] for x in range(len(y_test))]
```

In [82]:

```
error = mean_squared_error(test,prediction)
print(error) # mean squared error
print(error**0.5) # root mean square
```

8.482164232702896
2.9124155322863694

Although the RMSE of logistic are higher than linear, this is because the integer data. For logistic regression, in order to compare with other filters, we can also get the accuracy using LogisticRegressionCV().

In [83]:

```
# LogisticRegression cross validation
model = LogisticRegressionCV()
cv_results_logic = cross_val_score(model,abl_rfi,target.values,cv = 5)
```

In [84]:

```
cv_results_logic.mean().round(4)
```

Out[84]:

0.2036

Chapter 6

Model comparison

Based on all models, best accuracy obtained by feeding top-ranked features as input to 1-NN are as follows:

* Model	Best accuracy
* KNN	0.2671
* DT	0.2627
* RF	0.2730
* NB	0.2608
* MLP	0.2646

We have optimized each one of all models on the test data in a cross-validated fashion. Since cross validation is a random process, we perform pairwise t-tests to determine if any difference between the performance of any two optimized classifiers is statistically significant.

First, we perform 10-fold stratified cross-validation on each best model (without any repetitions). Second, we conduct a paired t-test for the accuracy between the RF model and other models since RF model have the best accuracy.

In [85]:

```
cv_method_ttest = StratifiedKFold(n_splits=10, random_state=111)

cv_results_KNN = cross_val_score(estimator=gs_KNN.best_estimator_,
                                 X=abl_rfi,
                                 y=target,
                                 cv=cv_method_ttest,
                                 n_jobs=-2,
                                 scoring='accuracy')
cv_results_KNN.mean()
```

Out[85]:

0.2673751003388005

In [86]:

```
cv_results_DT = cross_val_score(estimator=gs_DT.best_estimator_,
                                X=abl_rfi,
                                y=target,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='accuracy')
cv_results_DT.mean()
```

Out[86]:

0.26241255725499774

In [87]:

```
cv_results_RF = cross_val_score(estimator=gs_RF.best_estimator_,
                                X=abl_rfi,
                                y=target,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='accuracy')
cv_results_RF.mean()
```

Out[87]:

0.2673877175725291

In [88]:

```
cv_results_NB = cross_val_score(estimator=gs_NB.best_estimator_,
                                X=abl_power_rfi,
                                y=target,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='accuracy')
cv_results_NB.mean()
```

Out[88]:

0.260558472568088

In [89]:

```
cv_results_MLP = cross_val_score(estimator=gs_MLP.best_estimator_,
                                 X=abl_rfi,
                                 y=target,
                                 cv=cv_method_ttest,
                                 n_jobs=-2,
                                 scoring='accuracy')
cv_results_MLP.mean()
```

Out[89]:

0.26375452714665765

In [90]:

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

```
Ttest_relResult(statistic=0.002654717262444256, pvalue=0.9979397567467918)
Ttest_relResult(statistic=0.9309212625772436, pvalue=0.3761900801527821)
Ttest_relResult(statistic=0.855234509237703, pvalue=0.4146080896671649)
Ttest_relResult(statistic=0.644084508839602, pvalue=0.5355881288168893)
```

A p-value larger than 0.05 indicates there is no statistically significant difference. Looking at these results, we conclude that at a 95% significance level, although RF have higher score, we can't say that RF is statistically better than other models in this competition (in terms of accuracy).

To find best model, we shall consider the following metrics to evaluate models based on the test set:

- Precision
- Recall
- F1 Score (the harmonic average of precision and recall)
- Confusion Matrix

In [91]:

```
pred_KNN = gs_KNN.predict(abl_rfi)
pred_DT = gs_DT.predict(abl_rfi)
pred_RF = gs_RF.predict(abl_rfi)
pred_NB = gs_NB.predict(abl_power_rfi)
pred_MLP = gs_MLP.predict(abl_rfi)
```

In [92]:

```
#print("\nClassification report for K-Nearest Neighbor")
#print(metrics.classification_report(target, pred_KNN))
#print("\nClassification report for Decision Tree")
#print(metrics.classification_report(target, pred_DT))
print("\nClassification report for Random Forest")
print(metrics.classification_report(target, pred_RF))
#print("\nClassification report for Gaussian Naive Bayes")
#print(metrics.classification_report(target, pred_NB))
#print("\nClassification report for Multi-layer Perceptron")
#print(metrics.classification_report(target, pred_MLP))
```

```
Classification report for Random Forest
              precision    recall  f1-score   support

           1       0.00      0.00      0.00         1
           2       0.00      0.00      0.00         1
           3       0.41      0.47      0.44        15
           4       0.49      0.49      0.49        57
           5       0.46      0.44      0.45       115
           6       0.38      0.22      0.28       259
           7       0.32      0.49      0.39       391
           8       0.29      0.40      0.33       568
           9       0.29      0.26      0.27       689
          10       0.23      0.50      0.32       634
          11       0.28      0.27      0.27       487
          12       0.00      0.00      0.00       267
          13       0.00      0.00      0.00       203
          14       0.00      0.00      0.00       126
          15       0.00      0.00      0.00       103
          16       0.00      0.00      0.00        67
          17       0.00      0.00      0.00        58
          18       0.00      0.00      0.00        42
          19       0.00      0.00      0.00        32
          20       0.00      0.00      0.00        26
          21       0.00      0.00      0.00        14
          22       0.00      0.00      0.00         6
          23       0.00      0.00      0.00         9
          24       0.00      0.00      0.00         2
          25       0.00      0.00      0.00         1
          26       0.00      0.00      0.00         1
          27       0.00      0.00      0.00         2
          29       0.00      0.00      0.00         1

    accuracy                           0.28      4177
   macro avg       0.11      0.13      0.12      4177
weighted avg       0.23      0.28      0.24      4177
```

RF have slightly better f1-score and weighted avg recall than other models, followed by KNN. KNN have better precision score at the same time. Other classification are similar, we didn't print them all because of the large number of target levels. There are same situation in confusion matrix.

In [93]:

```
print("\nConfusion matrix for K-Nearest Neighbor")
print(metrics.confusion_matrix(target, pred_RF)) # Other confusion matrix are similar.
```

Confusion matrix for K-Nearest Neighbor

```
[[  0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   7   8   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   7  28  15   6   1   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   1  16  51  28  19   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   3  28  58 133  30   4   3   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   2  13  34 192 120  19  10   1   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   5  11 108 227 133  81   3   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   8  62 159 177 258  25   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   2  42  90 107 317  76   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   3  23  55  56 219 131   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0  13  38  40 112  64   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   1   5  26  27  99  45   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   3  16  20  64  23   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   1  16  14  50  22   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   1   7   6  33  20   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   2   6  33  17   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   4   3  21  14   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   1   2  19  10   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   2  17   7   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   1   1   6   6   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   1   3   2   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   1   7   1   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   2   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   1   1   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]
```

Based on the classification report and confusion matrix, RF have best overall performance, although not significant.

Chapter 7

Limitations and Proposed Solutions

Our modeling strategy has a few flaws and limitations.

First of all, we can observe that every model's accuracy are below 0.3, which is relatively low and hard for forecasting. This might due to the large number of levels and the highly imbalance in our target. As we can see from the confusion matrix, a significant number of targets are predicted into adjacent categories. There are several possible solutions.

For the imbalanced dataset, we can use oversampling technique to get rid of this imbalance. However, based on the classification report and original dataset, some level of target are extremely small, which would lost relatively in oversampling. We will consider remove some levels to get better performance.

Another way to solve this is by regrouping the target items into different groups using binning technique. By doing this, we can keep a better balance between exactness of prediction and model accuracy.

Secondly, since our goal is to pridict the target 'Rings' to get the age of abalone, we can regard this question as regression. However, we need a constant method to compare the performance between regression models and classification models.

Thirdly, in MLP classifier, we only worked with a small subset of all hyperparameters for shorter run times, With higher computing power, we can improve it by further expanding the hyperparameter search space by including other parameters of this classification method. There is similar situation in RF classifier. Since RF model slightly outperforms the other models, we can perhaps improve it by consider extra trees and other ensemble methods built on trees as potentially better models.

We may also add AUC scores into model selection and comparison.

Chapter 8

Summary

In this report, five classification algorithms and two regression algorithems were used to predict the target feature. For each method, cross validation method were used. We use grid search in hyperparameter tuning and get the best performance of each model based on their accuracy. In the end, we present and discuss the results in forms of accuracy scores, confusion matrix, classification report (recall, precision, F1-score). Based on the above analysis, RF model have the best accuracy as well as best recall and f1-score among all classification models. However, it's not significant higher than other models. Further analysis is needed due to the limitation.

References

- Peter E. (2012). Machine Learning in Action, Manning Publications Co., Greenwich.
- Vural A. (2019). ML Tutorials [online]. Available at https://www.featureranking.com/ml-tutorials/ (https://www.featureranking.com/ml-tutorials/) [Accessed 2019-06-09].
- Warwick N, Tracy L, et al. (1994). UCI Machine Learning Repository: Abalone Data Set [online]. Available at https://archive.ics.uci.edu/ml/datasets/Abalone (https://archive.ics.uci.edu/ml/datasets/Abalone) [Accessed 2019-06-09].
- Wu, K., Lu, BL., Utiyama, M. et al. (2008). An empirical comparison of min–max-modular k-NN with different voting methods to large-scale text categorization.[online]. Available at https://doi.org/10.1007/s00500-007-0242-3 (https://doi.org/10.1007/s00500-007-0242-3) [Accessed 2019-06-09].