Parkinsons Disease dataset Classification

Outline

- Introduction
- EDA
- Feature Engineering
- Cassification
- Conclusion

Introduction

In this report we will analyse the parkinsons dataset, we will try to fit multiple classification algorithms, tune their hyperparameters, and finally, compare them to figure out which one perform better in the status feature prediction.

Data Set Characteristics: Multivariate Number of Instances: 197 Area: Life Attribute Characteristics: Real Number of Attributes: 23 Date Donated: 2008-06-26 Associated Tasks: Classification Missing Values? N/A

Source:

The dataset was created by Max Little of the University of Oxford, in collaboration with the National Centre for Voice and Speech, Denver, Colorado, who recorded the speech signals. The original study published the feature extraction methods for general voice disorders.

Data Set Information:

This dataset is composed of a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease (PD). Each column in the table is a particular voice measure, and each row corresponds one of 195 voice recording from these individuals ("name" column). The main aim of the data is to discriminate healthy people from those with PD, **according to "status" column which is set to 0 for healthy and 1 for PD.**

The data is in ASCII CSV format. The rows of the CSV file contain an instance corresponding to one voice recording. There are around six recordings per patient, the name of the patient is identified in the first column.

Attribute Information:

- Matrix column entries (attributes):
- name - ASCII subject name and recording number

- MDVP:Fo(Hz) - Average vocal fundamental frequency
- MDVP:Fhi(Hz) - Maximum vocal fundamental frequency
- MDVP:Flo(Hz) - Minimum vocal fundamental frequency
- MDVP:Jitter(%),MDVP:Jitter(Abs),MDVP:RAP,MDVP:PPQ,Jitter:DDP - Several measures of variation in fundamental frequency
- MDVP:Shimmer,MDVP:Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,MDVP:APQ,Shimmer:DDA - Several measures of variation in amplitude
- NHR,HNR - Two measures of ratio of noise to tonal components in the voice
- status - Health status of the subject (one) - Parkinson's, (zero) - healthy
- RPDE,D2 - Two nonlinear dynamical complexity measures
- DFA - Signal fractal scaling exponent
- spread1,spread2,PPE - Three nonlinear measures of fundamental frequency variation

Exploratory Data Analysis

In [1]:

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

In [2]:

```
data = pd.read_csv("parkinsons.data", delimiter=',')
```

In [3]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   name              195 non-null    object 
 1   MDVP:Fo(Hz)       195 non-null    float64
 2   MDVP:Fhi(Hz)      195 non-null    float64
 3   MDVP:Flo(Hz)      195 non-null    float64
 4   MDVP:Jitter(%)    195 non-null    float64
 5   MDVP:Jitter(Abs)  195 non-null    float64
 6   MDVP:RAP          195 non-null    float64
 7   MDVP:PPQ          195 non-null    float64
 8   Jitter:DDP        195 non-null    float64
 9   MDVP:Shimmer      195 non-null    float64
 10  MDVP:Shimmer(dB)  195 non-null    float64
 11  Shimmer:APQ3      195 non-null    float64
 12  Shimmer:APQ5      195 non-null    float64
 13  MDVP:APQ          195 non-null    float64
 14  Shimmer:DDA       195 non-null    float64
 15  NHR               195 non-null    float64
 16  HNR               195 non-null    float64
 17  status            195 non-null    int64  
 18  RPDE              195 non-null    float64
 19  DFA               195 non-null    float64
 20  spread1           195 non-null    float64
 21  spread2           195 non-null    float64
 22  D2                195 non-null    float64
 23  PPE               195 non-null    float64
dtypes: float64(22), int64(1), object(1)
memory usage: 36.7+ KB
```

In [4]:

```
data.head()
```

Out[4]:

	name	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)
0	phon_R01_S01_1	119.992	157.302	74.997	0.00784	0.00007
1	phon_R01_S01_2	122.400	148.650	113.819	0.00968	0.00008
2	phon_R01_S01_3	116.682	131.111	111.555	0.01050	0.00009
3	phon_R01_S01_4	116.676	137.871	111.366	0.00997	0.00009
4	phon_R01_S01_5	116.014	141.781	110.655	0.01284	0.00011

5 rows × 24 columns

In [5]:

```
data['status'].value_counts(normalize=True)
```

Out[5]:

```
1    0.753846
0    0.246154
Name: status, dtype: float64
```

In [6]:

```
data.insert(len(data.columns)-1, 'status', data.pop('status'))
```

In [7]:

```
data.drop(columns='name', inplace=True)
```

In [8]:

```
#sns.pairplot(data, hue='status',  corner=True)
```

In [9]:

```
pd.set_option('display.max_columns', 21)
data.describe()
```

Out[9]:

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
mean	154.228641	197.104918	116.324631	0.006220	0.000044	0.003306
std	41.390065	91.491548	43.521413	0.004848	0.000035	0.002968
min	88.333000	102.145000	65.476000	0.001680	0.000007	0.000680
25%	117.572000	134.862500	84.291000	0.003460	0.000020	0.001660
50%	148.790000	175.829000	104.315000	0.004940	0.000030	0.002500
75%	182.769000	224.205500	140.018500	0.007365	0.000060	0.003835
max	260.105000	592.030000	239.170000	0.033160	0.000260	0.021440

8 rows × 23 columns

In [10]:

```
from sklearn.preprocessing import StandardScaler
```

In [11]:

```
num_cols = [x for x in data.columns if x!='status']
```

In [12]:
```
scaler = StandardScaler()

data[num_cols] = scaler.fit_transform(data[num_cols])
```

In [13]:
```
data.head()
```

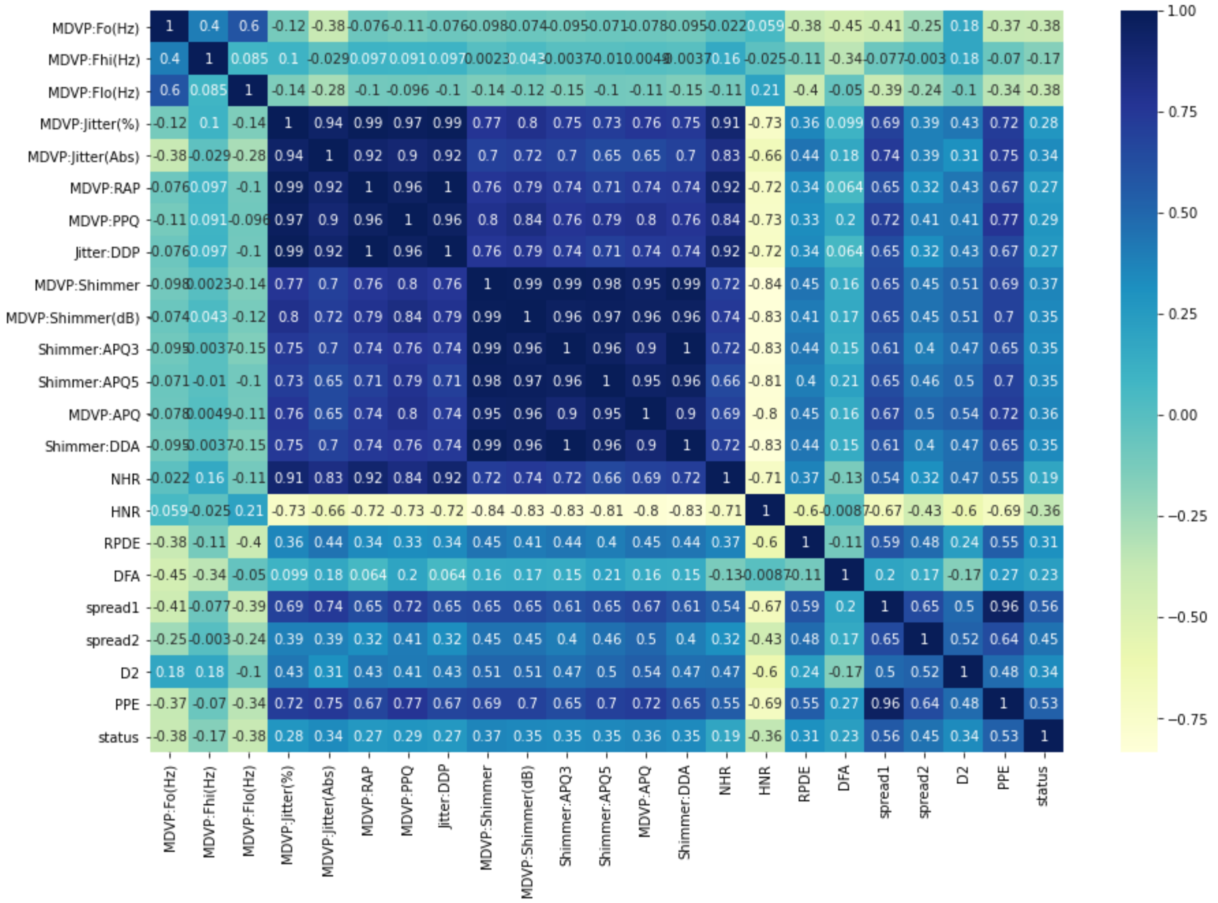
Out[13]:

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDV
0	-0.829300	-0.436165	-0.952037	0.334914	0.749759	0.132963	0.7
1	-0.770972	-0.530974	-0.057721	0.715418	1.037674	0.453892	1.2
2	-0.909476	-0.723168	-0.109875	0.884991	1.325589	0.720770	1.5
3	-0.909622	-0.649092	-0.114229	0.775389	1.325589	0.578885	1.2
4	-0.925657	-0.606245	-0.130608	1.368893	1.901418	1.095750	2.0

5 rows × 23 columns

In [14]:
```
Corr = data.corr()
```

In [15]:
```
# plotting correlation heatmap
plt.figure(figsize = (15,10))
sns.heatmap(Corr, cmap="YlGnBu", annot=True)

# displaying heatmap
plt.show()
```



We see that the 'Jitter:DDP' feature is perfectly correlated with 'MDVP:RAP'. The same applies to

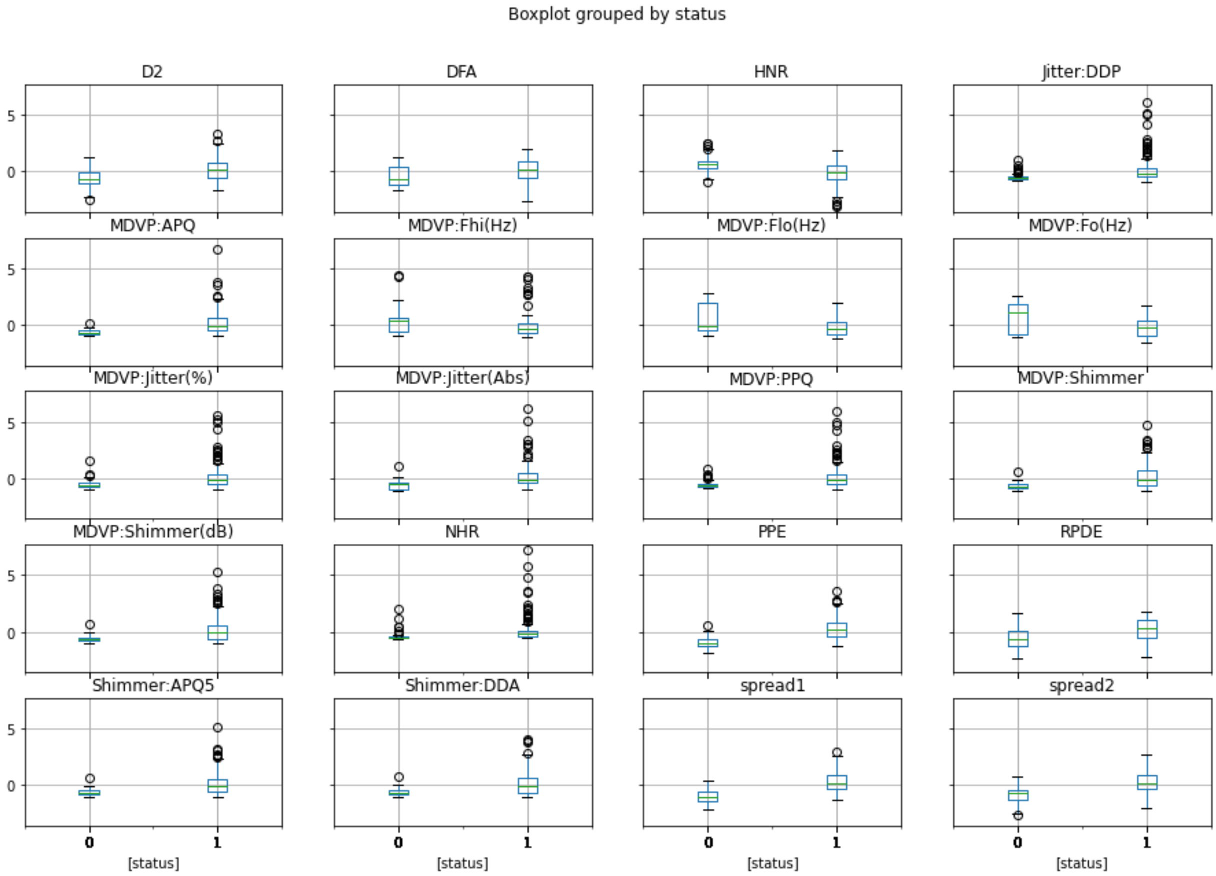
'Shimmer:DDA' and 'Shimmer:APQ3'. That's why we're going to drop duplicates.

Feature Engineering

In this section, we will drop perfectly correlated columns, and reduce the number of highly correlated ones using Principal Component Analysis (PCA).

In [16]:
```
data.drop(columns = ['Shimmer:APQ3', 'MDVP:RAP'],axis=1,  inplace=True)
```

In [17]:
```
data.boxplot(by='status', figsize=(15,10))
plt.show()
```



In [18]:
```
Corr = data.corr()
```

In [19]:
```
for x in range(Corr.shape[0]):
    Corr.iloc[x,x] = 0.0

Corr
```

Out[19]:

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)
MDVP:Fo(Hz)	0.000000	0.400985	0.596546	-0.118003	-0.382027
MDVP:Fhi(Hz)	0.400985	0.000000	0.084951	0.102086	-0.029198
MDVP:Flo(Hz)	0.596546	0.084951	0.000000	-0.139919	-0.277815
MDVP:Jitter(%)	-0.118003	0.102086	-0.139919	0.000000	0.935714
MDVP:Jitter(Abs)	-0.382027	-0.029198	-0.277815	0.935714	0.000000

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)
MDVP:PPQ	-0.112165	0.091126	-0.095828	0.974256	0.897778
Jitter:DDP	-0.076213	0.097150	-0.100488	0.990276	0.922913
MDVP:Shimmer	-0.098374	0.002281	-0.144543	0.769063	0.703322
MDVP:Shimmer(dB)	-0.073742	0.043465	-0.119089	0.804289	0.716601
Shimmer:APQ5	-0.070682	-0.009997	-0.101095	0.725561	0.648961
MDVP:APQ	-0.077774	0.004937	-0.107293	0.758255	0.648793
Shimmer:DDA	-0.094732	-0.003733	-0.150737	0.746635	0.697170
NHR	-0.021981	0.163766	-0.108670	0.906959	0.834972
HNR	0.059144	-0.024893	0.210851	-0.728165	-0.656810
RPDE	-0.383894	-0.112404	-0.400143	0.360673	0.441839
DFA	-0.446013	-0.343097	-0.050406	0.098572	0.175036
spread1	-0.413738	-0.076658	-0.394857	0.693577	0.735779
spread2	-0.249450	-0.002954	-0.243829	0.385123	0.388543
D2	0.177980	0.176323	-0.100629	0.433434	0.310694
PPE	-0.372356	-0.069543	-0.340071	0.721543	0.748162
status	-0.383535	-0.166136	-0.380200	0.278220	0.338653

In [20]:

```
Corr.columns
```

Out[20]:

```
Index(['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)', 'MDVP:Jitter(%)',
       'MDVP:Jitter(Abs)', 'MDVP:PPQ', 'Jitter:DDP', 'MDVP:Shimmer',
       'MDVP:Shimmer(dB)', 'Shimmer:APQ5', 'MDVP:APQ', 'Shimmer:DDA', 'NHR',
       'HNR', 'RPDE', 'DFA', 'spread1', 'spread2', 'D2', 'PPE', 'status'],
      dtype='object')
```

In [21]:

```
Corr.unstack().sort_values(ascending=False).drop_duplicates()
```

Out[21]:

```
Jitter:DDP        MDVP:Jitter(%)       0.990276
Shimmer:DDA       MDVP:Shimmer         0.987626
MDVP:Shimmer      MDVP:Shimmer(dB)     0.987258
Shimmer:APQ5      MDVP:Shimmer         0.982835
MDVP:Jitter(%)    MDVP:PPQ             0.974256
                                         ...
HNR               MDVP:APQ            -0.800407
                  Shimmer:APQ5        -0.813753
                  Shimmer:DDA         -0.827130
                  MDVP:Shimmer(dB)    -0.827805
                  MDVP:Shimmer        -0.835271
Length: 211, dtype: float64
```

In [22]:

```
filtered_list = []
for col in Corr.columns:
    features = sorted(list(Corr[(Corr > 0.9)][col][Corr[(Corr > 0.9)][col]>0.9].inde
    filtered_list.append(features)
```

In [23]:

```
unique=[]
```

```
for x in filtered_list:
    if x not in unique and x!=[]:
        unique.append(x)
```

In [24]:

```
unique
```

Out[24]:

```
[['Jitter:DDP', 'MDVP:Jitter(Abs)', 'MDVP:PPQ', 'NHR'],
 ['Jitter:DDP', 'MDVP:Jitter(%)'],
 ['MDVP:Jitter(%)', 'MDVP:Jitter(Abs)', 'MDVP:PPQ', 'NHR'],
 ['MDVP:APQ', 'MDVP:Shimmer(dB)', 'Shimmer:APQ5', 'Shimmer:DDA'],
 ['MDVP:APQ', 'MDVP:Shimmer', 'Shimmer:APQ5', 'Shimmer:DDA'],
 ['MDVP:APQ', 'MDVP:Shimmer', 'MDVP:Shimmer(dB)', 'Shimmer:DDA'],
 ['MDVP:Shimmer', 'MDVP:Shimmer(dB)', 'Shimmer:APQ5'],
 ['PPE'],
 ['spread1']]
```

Interestingly, when we look at these features description, we find that the main correlations are between the features that measure the frequency and amplitude.

- MDVP:Jitter(%),MDVP:Jitter(Abs),MDVP:RAP,MDVP:PPQ,Jitter:DDP - Several measures of variation in fundamental frequency
- MDVP:Shimmer,MDVP:Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,MDVP:APQ,Shimmer:DDA - Several measures of variation in amplitude

From this point, we will create two lists from these features, and perform dimentionality reduction to create a total of two new features that are representative of their variations.

In [25]:

```
features_l1=['Jitter:DDP', 'MDVP:Jitter(Abs)', 'MDVP:PPQ', 'NHR','MDVP:Jitter(%)']
features_l2=['MDVP:APQ', 'MDVP:Shimmer', 'Shimmer:APQ5', 'Shimmer:DDA','MDVP:Shimmer
```

In [26]:

```
feat_list = {'freq_measures':features_l1, 'amp_measures':features_l2}
```

In [27]:

```
data1 = data.copy()
```

In [28]:

```
data = data1.copy()
```

In [29]:

```
from sklearn.decomposition import PCA

PCAmod = PCA(n_components=1)
```

In [30]:

```
for f_list in feat_list:

    PCAmod.fit(data[feat_list[f_list]])
    tr_data = PCAmod.transform(data[feat_list[f_list]])
    tr_list = pd.DataFrame(tr_data, columns=['{}'.format(f_list)])
    data = pd.concat([tr_list, data], axis=1)
```

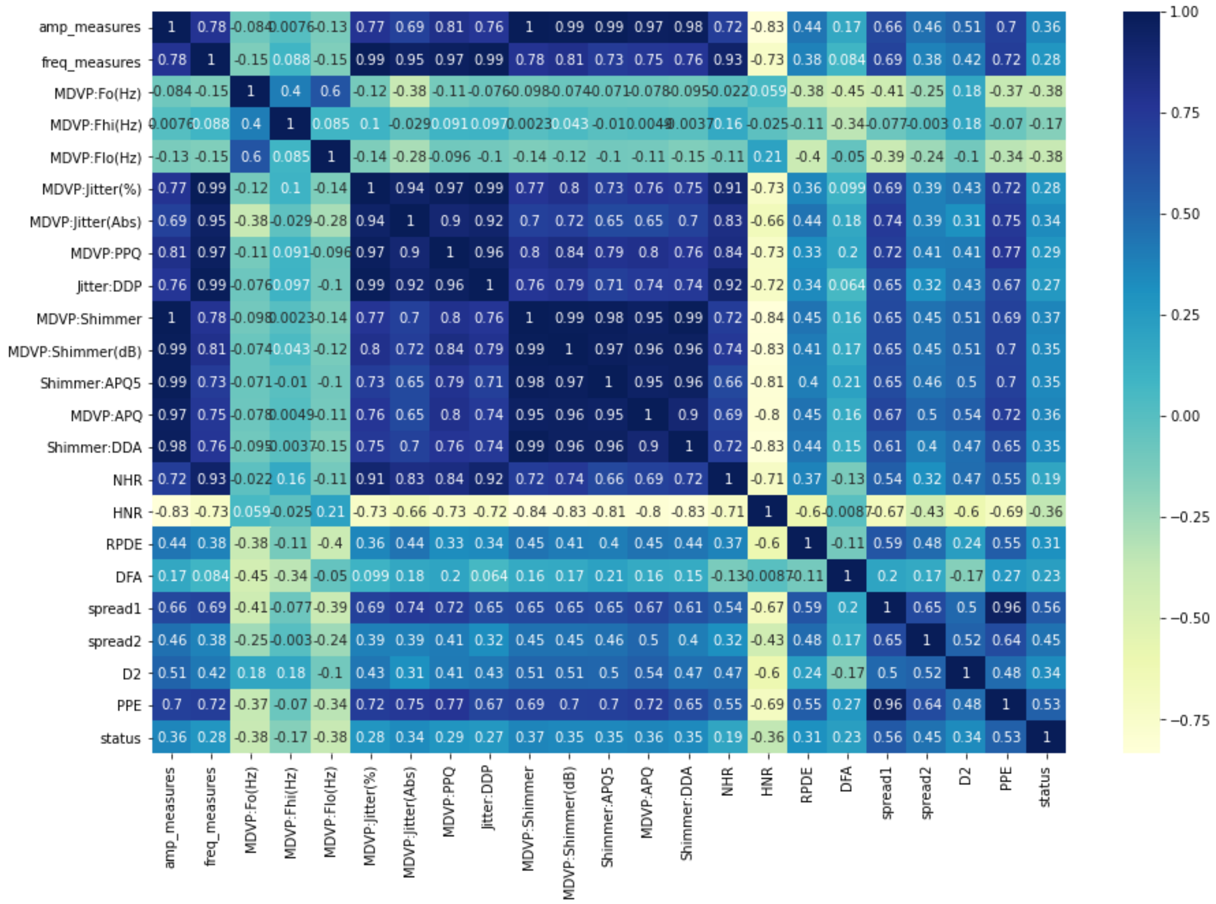
In [31]:

```
data1 = data.copy()
```

In [32]:

```
# plotting correlation heatmap
plt.figure(figsize = (15,10))
```

```
sns.heatmap(data.corr(), cmap="YlGnBu", annot=True)

# displaying heatmap
plt.show()
```



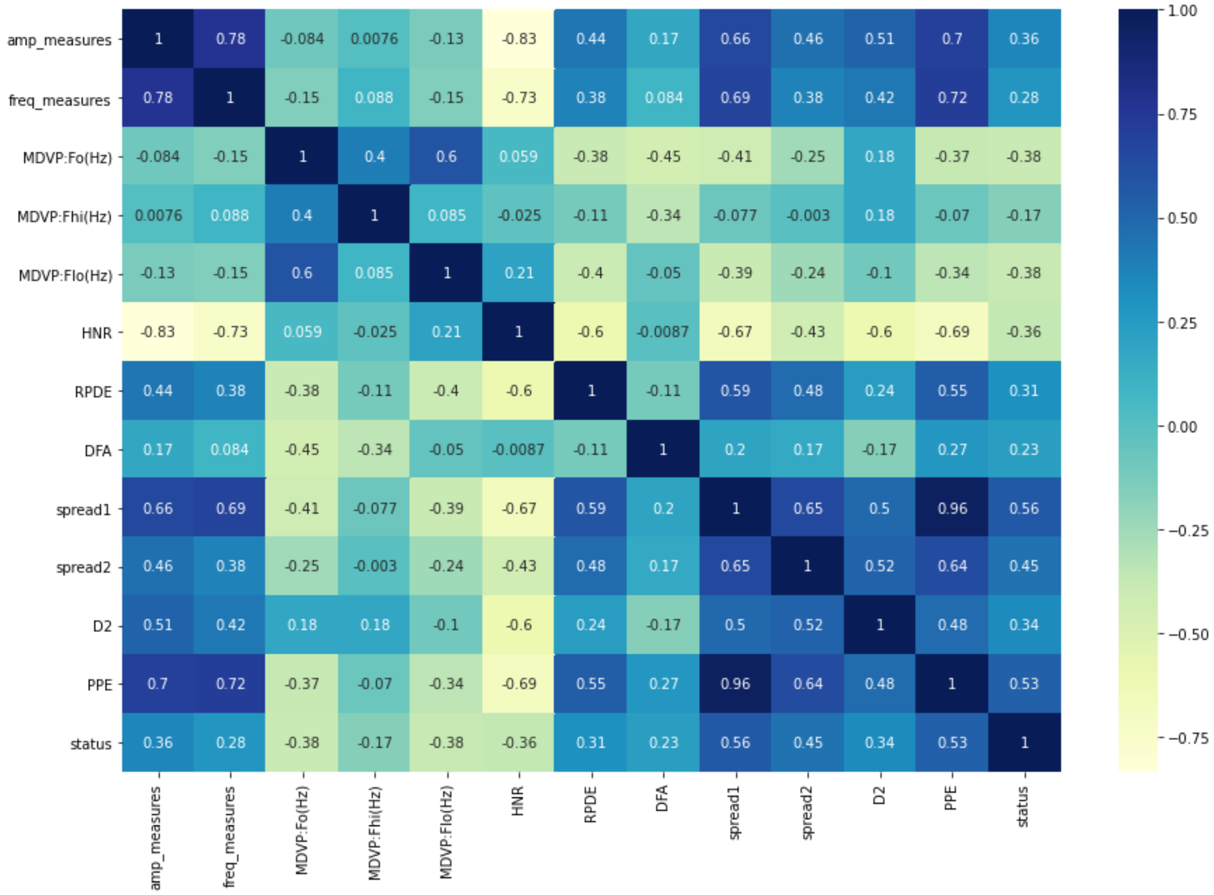
We see that the features that we gathered in list1 and list2 are highly correlated with the newly created features, which means that the PCA was successful, and thus we will drop those lists since reducing dimensions is our primary goal.

In [33]:

```
for f_list in feat_list:
    data.drop(columns=feat_list[f_list], inplace=True)
```

In [34]:

```
# plotting correlation heatmap
plt.figure(figsize = (15,10))
sns.heatmap(data.corr(), cmap="YlGnBu", annot=True)

# displaying heatmap
plt.show()
```

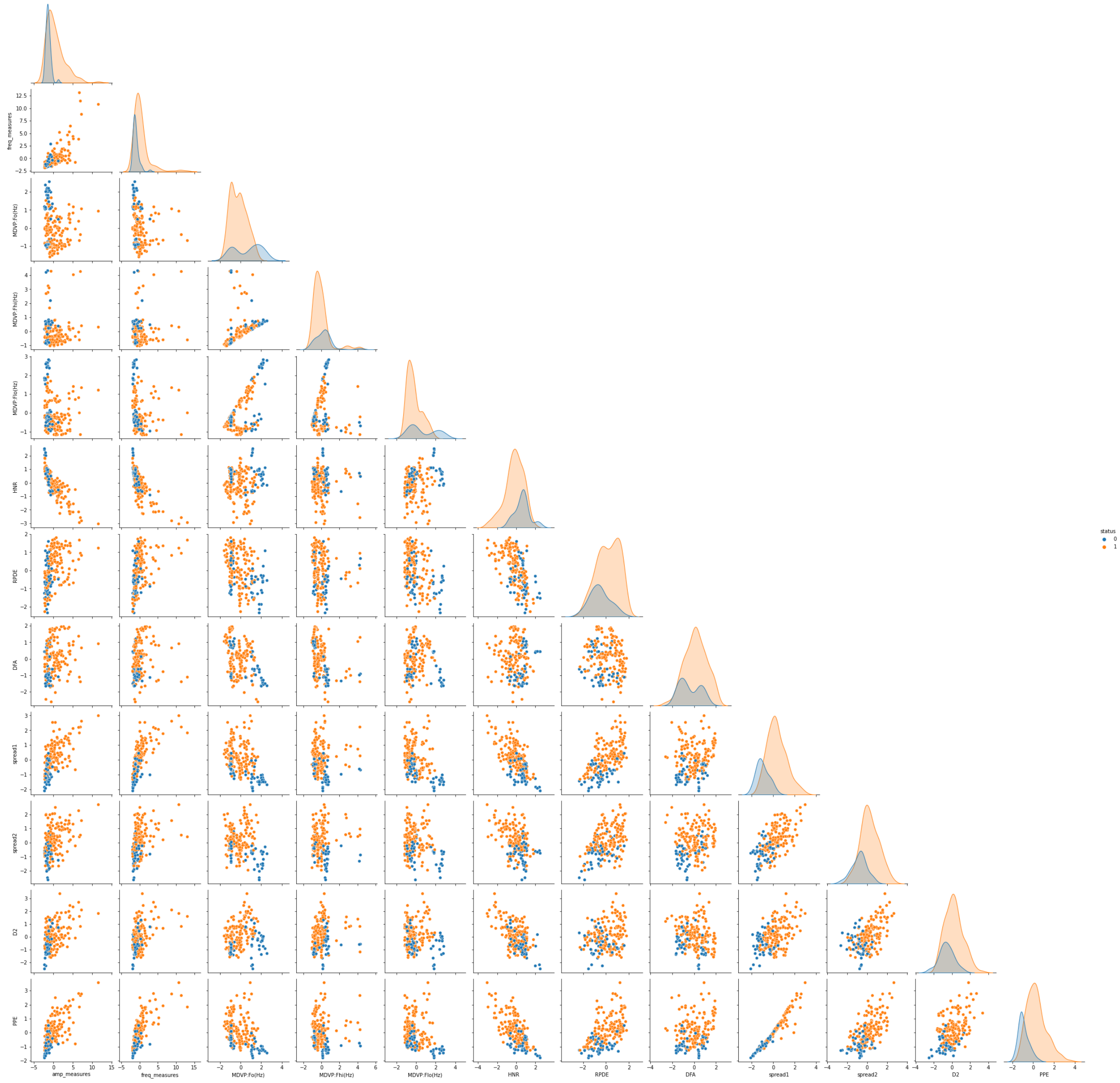


In [35]:
```
data.shape
```

Out[35]: (195, 13)

In [36]:
```
sns.pairplot(data, hue='status', corner=True)
```

Out[36]: <seaborn.axisgrid.PairGrid at 0x1594c9a3520>



Machine Learning ('status' Classification)

Here we will fit several algorithms to our data, mainly Logistic Regression, K Neighbors Classifier, SVC and XGBClassifier. And we will compare their performances based of the F1_score, since the status feature is constructed with inbalanced classes as we saw in EDA section :

0.753846 : Parkinsons Disease (1)

0.246154 : Healthy(0)

In [37]:

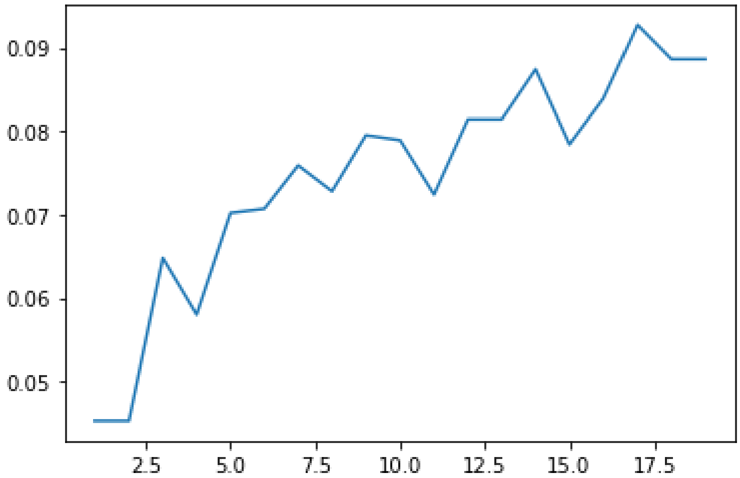
```
#import classifier algorithm here
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
import warnings
warnings.filterwarnings('ignore')

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold, cross_val_predict
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report,
```

In [38]:
```python
X = data[[x for x in data.columns if x!='status']]
y = data['status']
model_comparison = list()
```

In [39]:
```python
kf = StratifiedKFold(shuffle=True, random_state=22, n_splits=4 )
```

In [40]:
```python
error_rates = list() # 1-accuracy

ks = range(1,20)
scores=[]

for k in ks:

    predictions = cross_val_predict(KNeighborsClassifier(n_neighbors=k, weights='dis
    error = 1-round(f1_score(y, predictions), 4)
    scores.append(error)

plt.plot(ks, scores)
```

Out[40]: [<matplotlib.lines.Line2D at 0x1594b44e4c0>]



In [41]:
```python
knn = KNeighborsClassifier(n_neighbors=4, weights='distance')
knn = knn.fit(X, y)

y_pred = knn.predict(X)
score = f1_score(predictions, y)
model_comparison.append(('KNN', score))
score
```

Out[41]: 0.9113924050632911

In [42]:
```python
confusion_matrix(y, predictions)
```

Out[42]:
```
array([[ 23,  25],
       [  3, 144]], dtype=int64)
```

In [43]:
```python
from sklearn.metrics import classification_report

print(classification_report(y, predictions))
```

```
              precision    recall  f1-score   support
```

```
              0       0.88      0.48      0.62        48
              1       0.85      0.98      0.91       147

       accuracy                           0.86       195
      macro avg       0.87      0.73      0.77       195
   weighted avg       0.86      0.86      0.84       195
```

In [44]:
```
params = { 'penalty' : ['none','l1','l2','elasticnet'],
           'solver' : ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
           'C' : [0.02, 0.015, 0.01, 0.005] }

grid = GridSearchCV(LogisticRegression(), params, cv=kf, scoring='f1')
```

In [45]:
```
grid.fit(X, y)
```

Out[45]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.02, 0.015, 0.01, 0.005],
                         'penalty': ['none', 'l1', 'l2', 'elasticnet'],
                         'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag',
                                    'saga']},
             scoring='f1')
```

In [46]:
```
grid.best_score_, grid.best_params_
```

Out[46]:
```
(0.9097944991404907, {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'})
```

In [47]:
```
model_comparison.append(('Logistic_Regression', grid.best_score_))
```

In [48]:
```
params = {'kernel' : ['poly', 'rbf', 'sigmoid'],
          'C' : [3,3.5,4,4.5,5] }

grid = GridSearchCV(SVC(), params, cv=kf, scoring='f1')
```

In [49]:
```
grid.fit(X, y)
```

Out[49]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=SVC(),
             param_grid={'C': [3, 3.5, 4, 4.5, 5],
                         'kernel': ['poly', 'rbf', 'sigmoid']},
             scoring='f1')
```

In [50]:
```
grid.best_score_, grid.best_params_
```

Out[50]:
```
(0.9379457273794383, {'C': 4, 'kernel': 'rbf'})
```

In [51]:
```
model_comparison.append(('SVC', grid.best_score_))
```

In [52]:
```
params = {'learning_rate':[0.18,0.17,0.16],
          'max_depth':[1,2,3] }

grid = GridSearchCV(XGBClassifier( n_estimators=140, seed=27, verbosity=0),
                    params, cv=kf, scoring='f1')
```

In [53]:
```
grid.fit(X, y)
```

Out[53]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
                                     enable_categorical=False, gamma=None,
                                     gpu_id=None, importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimators=140, n_jobs=None,
                                     num_parallel_tree=None, predictor=None,
                                     random_state=None, reg_alpha=None,
                                     reg_lambda=None, scale_pos_weight=None,
                                     seed=27, subsample=None, tree_method=None,
                                     validate_parameters=None, verbosity=0),
             param_grid={'learning_rate': [0.18, 0.17, 0.16],
                         'max_depth': [1, 2, 3]},
             scoring='f1')
```

In [54]:
```
grid.best_score_, grid.best_params_
```

Out[54]:
```
(0.9463612717037375, {'learning_rate': 0.17, 'max_depth': 2})
```

In [55]:
```
model_comparison.append(('XGBoost_Classifier', grid.best_score_))
```

In [56]:
```
Comparison = pd.DataFrame(model_comparison, columns=('Model','F1_Score'))
Comparison.sort_values(by='F1_Score', ascending=False).reset_index(drop=True)
```

Out[56]:

	Model	F1_Score
0	XGBoost_Classifier	0.946361
1	SVC	0.937946
2	KNN	0.911392
3	Logistic_Regression	0.909794

Without Dimensionality Reduction (DR)

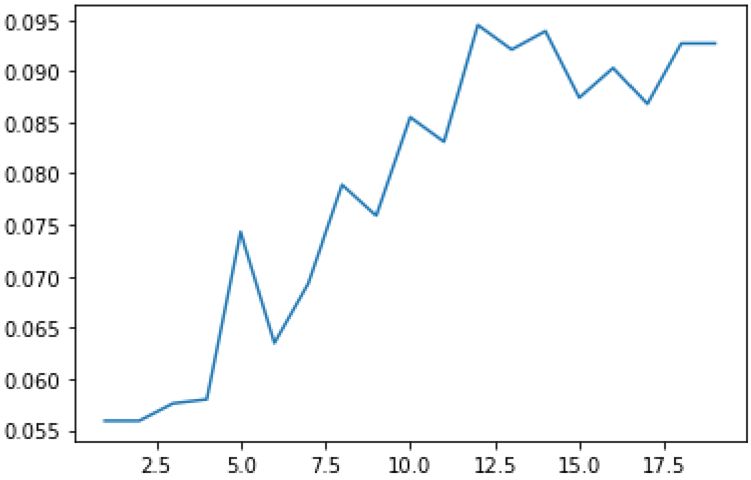
Now we will try the same algorithms but this time using the whole dataset, without performing dimensionality reduction.

In [57]:
```
X1 = data1[[x for x in data1.columns if x!='status']]
y1 = data1['status']
model_comparison1 = list()
```

In [58]:
```
kf = StratifiedKFold(shuffle=True, random_state=22, n_splits=4 )
```

In [59]:
```python
error_rates1 = list() # 1-accuracy

ks = range(1,20)
scores1=[]

for k in ks:

    predictions1 = cross_val_predict(KNeighborsClassifier(n_neighbors=k, weights='di
    error1 = 1-round(f1_score(y1, predictions1), 4)
    scores1.append(error1)

plt.plot(ks, scores1)
```

Out[59]:
```
[<matplotlib.lines.Line2D at 0x1594cf238b0>]
```



In [60]:
```python
knn = KNeighborsClassifier(n_neighbors=4, weights='distance')
knn = knn.fit(X1, y1)

y_pred = knn.predict(X1)
score1 = f1_score(predictions, y1)
model_comparison1.append(('KNN', score1))
score1
```

Out[60]:
```
0.9113924050632911
```

In [61]:
```python
confusion_matrix(y1, predictions)
```

Out[61]:
```
array([[ 23,  25],
       [  3, 144]], dtype=int64)
```

In [62]:
```python
from sklearn.metrics import classification_report

print(classification_report(y1, predictions))
```

```
              precision    recall  f1-score   support

           0       0.88      0.48      0.62        48
           1       0.85      0.98      0.91       147

    accuracy                           0.86       195
   macro avg       0.87      0.73      0.77       195
weighted avg       0.86      0.86      0.84       195
```

In [63]:
```python
params = { 'penalty' : ['none','l1','l2','elasticnet'],
```

```
                'solver' : ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
                'C' : [0.02, 0.015, 0.01, 0.005] }

grid = GridSearchCV(LogisticRegression(), params, cv=kf, scoring='f1')
```

In [64]:
```
grid.fit(X1, y1)
```

Out[64]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.02, 0.015, 0.01, 0.005],
                         'penalty': ['none', 'l1', 'l2', 'elasticnet'],
                         'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag',
                                    'saga']},
             scoring='f1')
```

In [65]:
```
grid.best_score_, grid.best_params_
```

Out[65]:
```
(0.9085443037974683, {'C': 0.015, 'penalty': 'l2', 'solver': 'newton-cg'})
```

In [66]:
```
model_comparison1.append(('Logistic_Regression', grid.best_score_))
```

In [67]:
```
params = {'kernel' : ['poly', 'rbf', 'sigmoid'],
          'C' : [1,1.5,2,2.5,3] }

grid = GridSearchCV(SVC(), params, cv=kf, scoring='f1')
```

In [68]:
```
grid.fit(X1, y1)
```

Out[68]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=SVC(),
             param_grid={'C': [1, 1.5, 2, 2.5, 3],
                         'kernel': ['poly', 'rbf', 'sigmoid']},
             scoring='f1')
```

In [69]:
```
grid.best_score_, grid.best_params_
```

Out[69]:
```
(0.9358868979122144, {'C': 2, 'kernel': 'rbf'})
```

In [70]:
```
model_comparison1.append(('SVC', grid.best_score_))
```

In [71]:
```
params = {'learning_rate':[0.18,0.17,0.16],
          'max_depth':[2,3,4] }

grid = GridSearchCV(XGBClassifier( n_estimators=140, seed=27, verbosity=0),
                    params, cv=kf, scoring='f1')
```

In [72]:
```
grid.fit(X1, y1)
```

Out[72]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=4, random_state=22, shuffle=True),
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
```

```
                                                 enable_categorical=False, gamma=None,
                                                 gpu_id=None, importance_type=None,
                                                 interaction_constraints=None,
                                                 learning_rate=None, max_delta_step=None,
                                                 max_depth=None, min_child_weight=None,
                                                 missing=nan, monotone_constraints=None,
                                                 n_estimators=140, n_jobs=None,
                                                 num_parallel_tree=None, predictor=None,
                                                 random_state=None, reg_alpha=None,
                                                 reg_lambda=None, scale_pos_weight=None,
                                                 seed=27, subsample=None, tree_method=None,
                                                 validate_parameters=None, verbosity=0),
                         param_grid={'learning_rate': [0.18, 0.17, 0.16],
                                     'max_depth': [2, 3, 4]},
                         scoring='f1')
```

In [73]:

```
grid.best_score_, grid.best_params_
```

Out[73]:

```
(0.9504798191637103, {'learning_rate': 0.17, 'max_depth': 3})
```

In [74]:

```
model_comparison1.append(('XGBoost_Classifier', grid.best_score_))
```

In [75]:

```
Comparison1 = pd.DataFrame(model_comparison1, columns=('Model','F1_Score'))
Comparison1.sort_values(by='F1_Score', ascending=False).reset_index(drop=True)
```

Out[75]:

	Model	F1_Score
0	XGBoost_Classifier	0.950480
1	SVC	0.935887
2	KNN	0.911392
3	Logistic_Regression	0.908544

We see that the models ranking didn't differ after performing dimensionality reduction, let's compare them with the models using dimensionality reduction.

In [78]:

```
Comparison.set_index('Model').join(Comparison1.
                                   set_index('Model'),
                                   lsuffix='_With-DR',
                                   rsuffix='_Without-DR').sort_values(by='F1_Score_W
```

Out[78]:

	F1_Score_With-DR	F1_Score_Without-DR
Model		
XGBoost_Classifier	0.946361	0.950480
SVC	0.937946	0.935887
KNN	0.911392	0.911392
Logistic_Regression	0.909794	0.908544

We see that the Dimensionality Reduction powered-up the SVC and Logistic_Regression performance, while it didn't affect KNN model at all, it impacted negatively the greedy algorithm XGBoost_Classifier.

Conclusion

We can see that these models perform great in predicting the target feature, achieving F1_score greater than 90% with the XGBoost algorithm in the first position.

We can go further in improving the model performance by tuning other hyperparameters, or even building a Voting Classifier.

Other improvements would involve creating combinations between variables.

Acknowledgments:

This dataset was downloaded from the UCI Machine Learning Repository website

'Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection', Little MA, McSharry PE, Roberts SJ, Costello DAE, Moroz IM. BioMedical Engineering OnLine 2007, 6:23 (26 June 2007)