Models Accuracy

May 10, 2018

0.1 Overview:

This notebook will focus on shaping the data in ways that make it satisfactory for the machine learning process.

0.1.1 Scale the dataset

In general, learning algorithms benefit from standardization of the data set. If some outliers are present in the set, robust scalers or transformers are more appropriate. Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn.

```
In [1]: import pandas as pd
        import numpy as np
        import sys
        from sklearn.preprocessing import StandardScaler
        from sklearn.preprocessing import LabelEncoder
        from sklearn.model_selection import train_test_split
        from sklearn.model_selection import cross_val_score
        import random

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

```
In [2]: # START:OWN CODE
        sys.path.append('/Users/Kassi/Desktop/Gender_Recognition_by_Voice')
```

```
In [3]: path = '/Users/Kassi/Desktop/Gender_Recognition_by_Voice/voice.csv'
        voice_data = pd.read_csv(path)
        col_names = list(voice_data.columns.values)
        print("Total number of samples: {}".format(voice_data.shape[0]))
        print("Total number of male: {}".format(voice_data[voice_data.label == 'male'].shape[0
        print("Total number of female: {}".format(voice_data[voice_data.label == 'female'].sha
```

```
Total number of samples: 3168
Total number of male: 1584
Total number of female: 1584
```

Check if dataset contains NA's

```
In [4]: voice_data.isnull().any().any()

Out[4]: False
```

Fortunately, our dataset does not contain any missing values and therefore does not need cleaning.

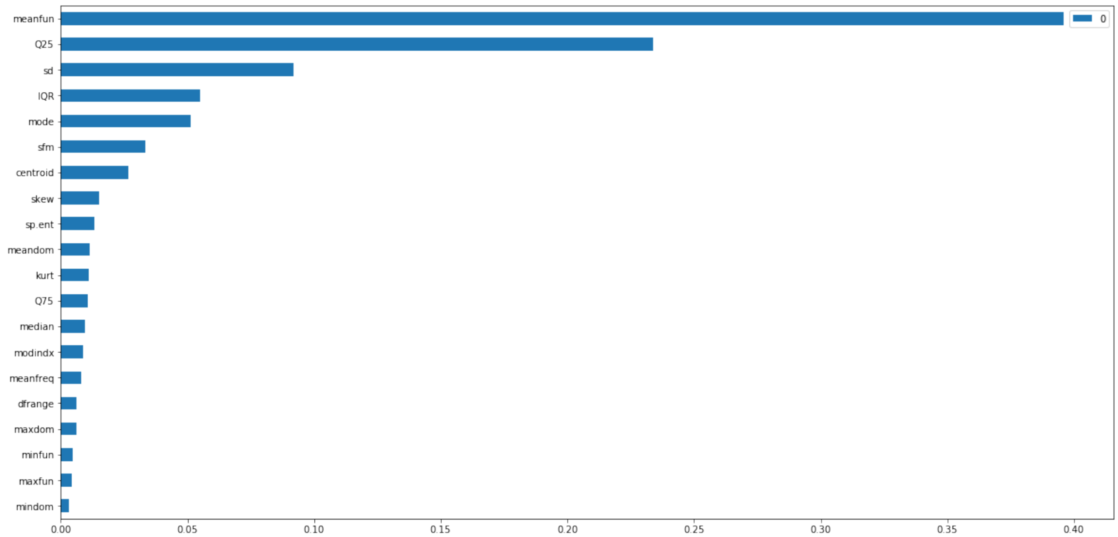
```
In [5]: voice_data = voice_data.values
        voices = voice_data[:, :-1]
        labels = voice_data[:, -1:]

In [6]: gender_encoder = LabelEncoder()
        labels = gender_encoder.fit_transform(labels)

In [7]: # 2 most significant features (IQR and meanfun)
        voices_two_features = voices[:,[5,12]]
        labels_two_features = labels

        # train_x_two_features.shape
        # labels_two_features.shape

In [8]: # Splitting the whole dataset into the Training set and Test set
        train_x, test_x, train_y, test_y = train_test_split(voices, labels, test_size=0.25, ran

        # Splitting the subset into the Training set and Test set
        train_x_two_features, test_x_two_features, train_y_two_features, test_y_two_features =

In [9]: # Feature Scaling (all features)
        # Learning algorithms benefit from standardization of the whole data set
        from sklearn.preprocessing import StandardScaler
        sc = StandardScaler()
        train_x = sc.fit_transform(train_x)
        test_x = sc.transform(test_x)

In [10]: # Feature Scaling (top 2 features)
         # Learning algorithms benefit from standardization of the subset
         sc = StandardScaler()
         train_x_two_features = sc.fit_transform(train_x_two_features)
         test_x_two_features = sc.transform(test_x_two_features)
```

0.1.2 Feature Importance

We used random forest to gain an insight on the importance of each feature. And again, we found that the IQR and Meanfun are two most significant features

```
In [11]: from sklearn.ensemble import RandomForestClassifier
         import matplotlib.pyplot as plt
         import seaborn as sns
```

```
from pandas.tools.plotting import scatter_matrix

clf = RandomForestClassifier()
clf.fit(voices, labels)
col_names = col_names[:-1]
#print(col_names)
importance = list(zip(clf.feature_importances_ , col_names))
#print(importance)
importance.sort()
pd.DataFrame(importance, index=[x for (_,x) in importance]).plot(kind = 'barh', figsi
plt.show()
```

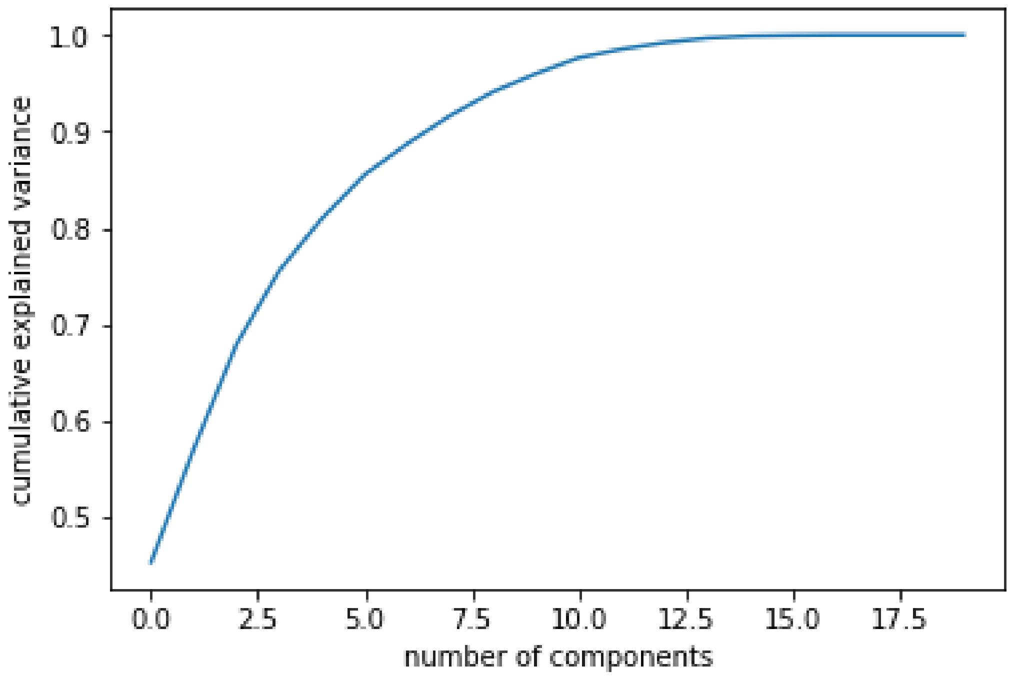


The plot above showed the top two features are still meanfun and IQR.

0.1.3 Principle Component Analysis(PCA)

The number of components is determined by looking at the cumulative explained variance ratio as a function of the number of components

```
In [12]: from sklearn.decomposition import PCA

         pca = PCA().fit(train_x)
         plt.plot(np.cumsum(pca.explained_variance_ratio_))
         plt.xlabel('number of components')
         plt.ylabel('cumulative explained variance');
         plt.show()
```



This curve quantifies how much of the total, 20-dimensional variance is contained within the first N components. For example, we see that with the digits the first 5 components contain approximately 85% of the variance, while we need around 10 components to describe close to 100% of the variance.

```
In [13]: pca = PCA(n_components = 10)
         pca.fit(train_x)
         pca_train_x = pca.transform(train_x)
         pca_test_x = pca.transform(test_x)
```

```
In [14]: # Store variables in the current directory and use it later
         %store train_x
         %store test_x
         %store train_y
         %store test_y

         %store voices
         %store labels

         %store train_x_two_features
         %store test_x_two_features
         %store train_y_two_features
         %store test_y_two_features

         %store pca_train_x
         %store pca_test_x
```

```
Stored 'train_x' (ndarray)
Stored 'test_x' (ndarray)
Stored 'train_y' (ndarray)
Stored 'test_y' (ndarray)
Stored 'voices' (ndarray)
Stored 'labels' (ndarray)
Stored 'train_x_two_features' (ndarray)
Stored 'test_x_two_features' (ndarray)
Stored 'train_y_two_features' (ndarray)
Stored 'test_y_two_features' (ndarray)
Stored 'pca_train_x' (ndarray)
Stored 'pca_test_x' (ndarray)
```

In [15]:
```
# Importing Jupyter Notebooks as Modules
# code get from http://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Im

import io, os, sys, types
from IPython import get_ipython
from nbformat import read
from IPython.core.interactiveshell import InteractiveShell

def find_notebook(fullname, path=None):
    """find a notebook, given its fully qualified name and an optional path

    This turns "foo.bar" into "foo/bar.ipynb"
    and tries turning "Foo_Bar" into "Foo Bar" if Foo_Bar
    does not exist.
    """
    name = fullname.rsplit('.', 1)[-1]
    if not path:
        path = ['']
    for d in path:
        nb_path = os.path.join(d, name + ".ipynb")
        if os.path.isfile(nb_path):
            return nb_path
        # let import Notebook_Name find "Notebook Name.ipynb"
        nb_path = nb_path.replace("_", " ")
        if os.path.isfile(nb_path):
            return nb_path


class NotebookLoader(object):
    """Module Loader for Jupyter Notebooks"""
    def __init__(self, path=None):
        self.shell = InteractiveShell.instance()
        self.path = path
```

```
    def load_module(self, fullname):
        """import a notebook as a module"""
        path = find_notebook(fullname, self.path)

        print ("importing Jupyter notebook from %s" % path)

        # load the notebook object
        with io.open(path, 'r', encoding='utf-8') as f:
            nb = read(f, 4)


        # create the module and add it to sys.modules
        # if name in sys.modules:
        #    return sys.modules[name]
        mod = types.ModuleType(fullname)
        mod.__file__ = path
        mod.__loader__ = self
        mod.__dict__['get_ipython'] = get_ipython
        sys.modules[fullname] = mod

        # extra work to ensure that magics that would affect the user_ns
        # actually affect the notebook module's ns
        save_user_ns = self.shell.user_ns
        self.shell.user_ns = mod.__dict__

        try:
          for cell in nb.cells:
            if cell.cell_type == 'code':
                # transform the input to executable Python
                code = self.shell.input_transformer_manager.transform_cell(cell.source
                # run the code in themodule
                exec(code, mod.__dict__)
        finally:
            self.shell.user_ns = save_user_ns
        return mod


class NotebookFinder(object):
    """Module finder that locates Jupyter Notebooks"""
    def __init__(self):
        self.loaders = {}

    def find_module(self, fullname, path=None):
        nb_path = find_notebook(fullname, path)
        if not nb_path:
            return

        key = path
```

```
            if path:
                # lists aren't hashable
                key = os.path.sep.join(path)

            if key not in self.loaders:
                self.loaders[key] = NotebookLoader(path)
            return self.loaders[key]
```

```
In [16]: sys.meta_path.append(NotebookFinder())
```

```
In [17]: import SVM
         print("Run linear svm with all features(Built-in Algorithm):")
         SVM.run_linear_svm(train_x, test_x, train_y, test_y)
         print("------------------------------------------------------------")
         print("Run rbf svm with all features(Built-in Algorithm):")
         SVM.run_rbf_svm(train_x, test_x, train_y, test_y)
         print("============================================================")
         print("Run linear svm with dimensionality reduction using PCA(Built-in Algorithm)")
         SVM.run_linear_svm(pca_train_x, pca_test_x, train_y, test_y)
         print("------------------------------------------------------------")
         print("Run rbf svm with dimensionality reduction using PCA(Built-in Algorithm)")
         SVM.run_rbf_svm(pca_train_x, pca_test_x, train_y, test_y)
         print("============================================================")
         print("Run linear svm with top 2 features(Built-in Algorithm):")
         SVM.run_linear_svm(train_x_two_features, test_x_two_features, train_y_two_features, te
         print("============================================================")
         print("Run rbf svm with top 2 features(Built-in Algorithm):")
         SVM.run_rbf_svm(train_x_two_features, test_x_two_features, train_y_two_features, test_
```

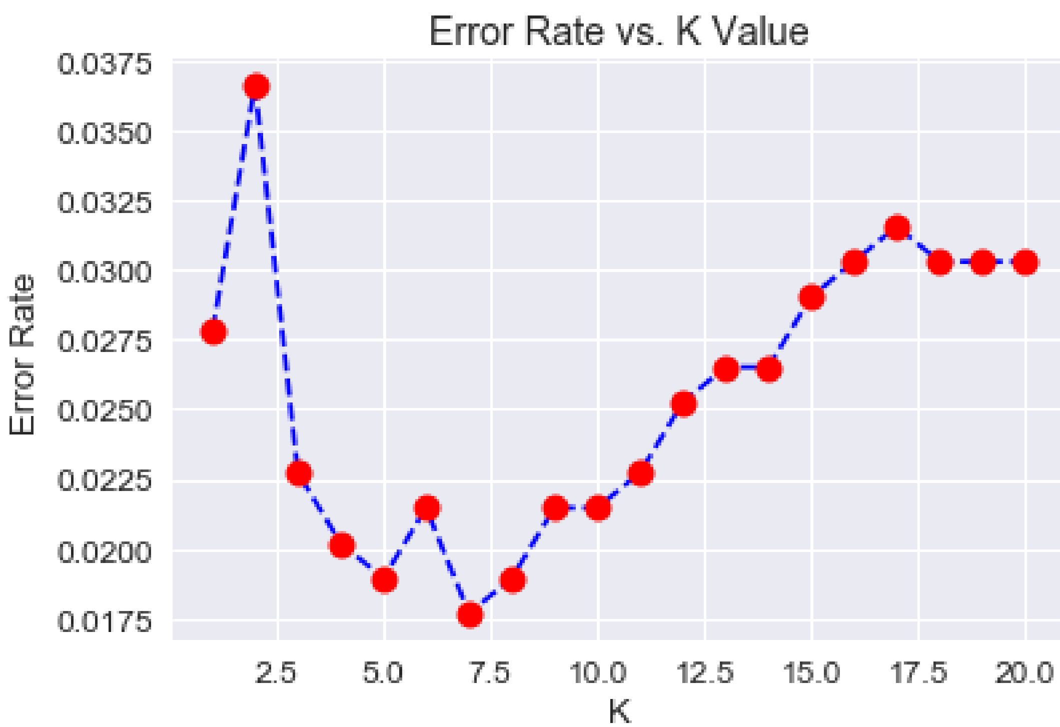
```
importing Jupyter notebook from SVM.ipynb
Run linear svm with all features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.9743290243
Out-of-sample accuracy for svm with Linear kernel: 0.9722076268
------------------------------------------------------------
Run rbf svm with all features(Built-in Algorithm):
In-sample accuracy for svm with RBF kernel: 0.9806448362
Out-of-sample accuracy for svm with RBF kernel: 0.9722235491
============================================================
Run linear svm with dimensionality reduction using PCA(Built-in Algorithm)
In-sample accuracy for svm with Linear kernel: 0.9751702488
Out-of-sample accuracy for svm with Linear kernel: 0.9734654884
------------------------------------------------------------
Run rbf svm with dimensionality reduction using PCA(Built-in Algorithm)
In-sample accuracy for svm with RBF kernel: 0.9781202858
Out-of-sample accuracy for svm with RBF kernel: 0.9633787119
============================================================
Run linear svm with top 2 features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.9629720840
```

```
Out-of-sample accuracy for svm with Linear kernel: 0.9709338428
============================================================
Run rbf svm with top 2 features(Built-in Algorithm):
In-sample accuracy for svm with RBF kernel: 0.9659132604
Out-of-sample accuracy for svm with RBF kernel: 0.9734654884
```

```
In [18]: import KNN
         print("with all features(Built-in Algorithm):")
         KNN.run_k_nearest_neighbour(train_x, test_x, train_y, test_y)
         print("============================================================")
         print("with all features(Self-Written Algorithm):")
         KNN.run_my_k_nearest_neighbour(train_x, test_x, train_y, test_y)
         print("============================================================")
         print("with top 2 features(Built-in Algorithm):")
         KNN.run_k_nearest_neighbour(train_x_two_features, test_x_two_features, train_y_two_fea
         print("============================================================")
         print("with top 2 features(Self-Written Algorithm):")
         KNN.run_my_k_nearest_neighbour(train_x_two_features, test_x_two_features, train_y_two_
         print("============================================================")
         print("with dimensionality reduction using PCA(Built-in Algorithm)")
         KNN.run_k_nearest_neighbour(pca_train_x, pca_test_x, train_y, test_y)
         print("============================================================")
         print("with dimensionality reduction using PCA(Self-Written Algorithm)")
         KNN.run_my_k_nearest_neighbour(pca_train_x, pca_test_x, train_y, test_y)
```

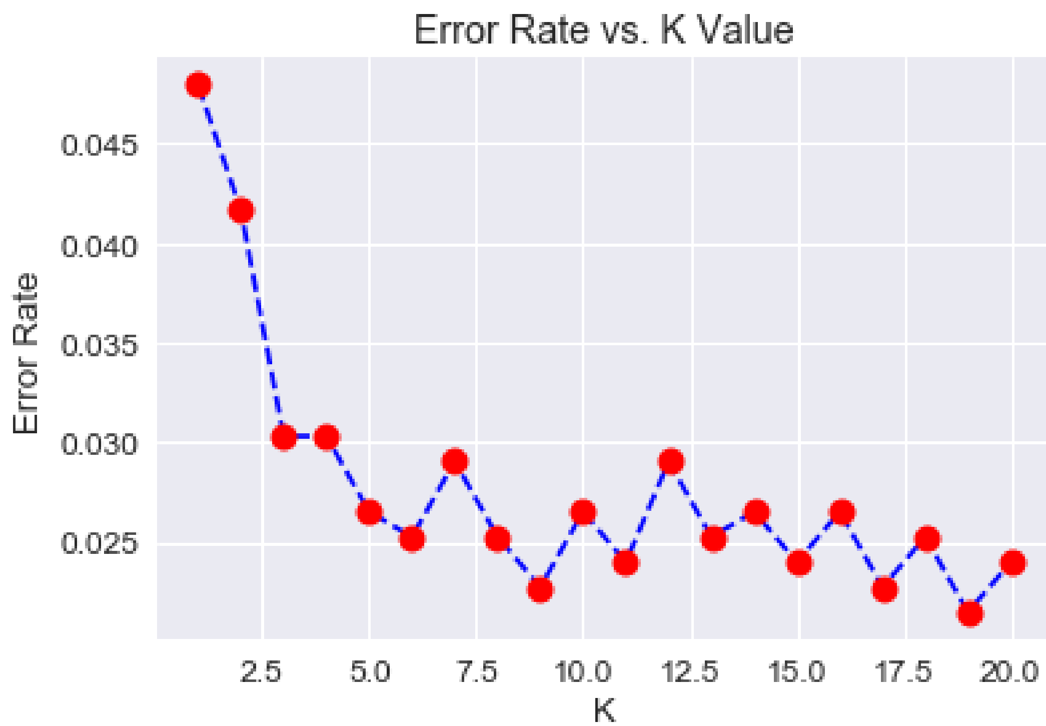
```
importing Jupyter notebook from KNN.ipynb

Use self-written KNeighbors Classifier:
Correct rate is 0.972222222222 when K = 1.
Correct rate is 0.963383838384 when K = 2.
Correct rate is 0.977272727273 when K = 3.
Correct rate is 0.979797979798 when K = 4.
Correct rate is 0.981060606061 when K = 5.
Correct rate is 0.978535353535 when K = 6.
Correct rate is 0.982323232323 when K = 7.
Correct rate is 0.981060606061 when K = 8.
Correct rate is 0.978535353535 when K = 9.
Correct rate is 0.978535353535 when K = 10.
Correct rate is 0.977272727273 when K = 11.
Correct rate is 0.974747474747 when K = 12.
Correct rate is 0.973484848485 when K = 13.
Correct rate is 0.973484848485 when K = 14.
Correct rate is 0.97095959596 when K = 15.
Correct rate is 0.969696969697 when K = 16.
Correct rate is 0.968434343434 when K = 17.
Correct rate is 0.969696969697 when K = 18.
Correct rate is 0.969696969697 when K = 19.
```

```
Highest correct rate 0.982323232323 occurs at K = 7.
with all features(Built-in Algorithm):
```



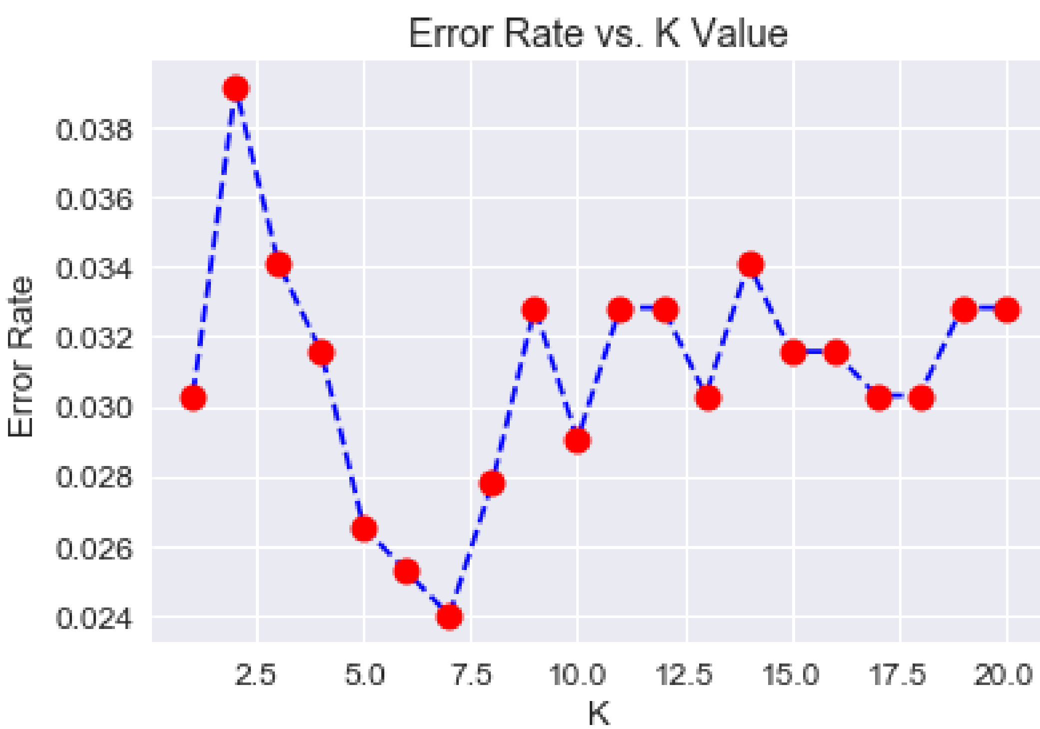

```
Lowest error is 0.0176767676768 occurs at k=7.
In-sample accuracy for KNN: 0.9696947768
Out-of-sample accuracy for KNN: 0.9507602898
===========================================================
with all features(Self-Written Algorithm):

Use self-written KNeighbors Classifier:
Correct rate is 0.972222222222 when K = 1.
Correct rate is 0.963383838384 when K = 2.
Correct rate is 0.977272727273 when K = 3.
Correct rate is 0.979797979798 when K = 4.
Correct rate is 0.981060606061 when K = 5.
Correct rate is 0.978535353535 when K = 6.
Correct rate is 0.982323232323 when K = 7.
Correct rate is 0.981060606061 when K = 8.
Correct rate is 0.978535353535 when K = 9.
Correct rate is 0.978535353535 when K = 10.
Correct rate is 0.977272727273 when K = 11.
Correct rate is 0.974747474747 when K = 12.
Correct rate is 0.973484848485 when K = 13.
```

```
Correct rate is 0.973484848485 when K = 14.
Correct rate is 0.97095959596 when K = 15.
Correct rate is 0.969696969697 when K = 16.
Correct rate is 0.968434343434 when K = 17.
Correct rate is 0.969696969697 when K = 18.
Correct rate is 0.969696969697 when K = 19.
Highest correct rate 0.982323232323 occurs at K = 7.
===========================================================
with top 2 features(Built-in Algorithm):
```




```
Lowest error is 0.0214646464646 occurs at k=19.
In-sample accuracy for KNN: 0.9654922078
Out-of-sample accuracy for KNN: 0.9633468673
===========================================================
with top 2 features(Self-Written Algorithm):

Use self-written KNeighbors Classifier:
Correct rate is 0.95202020202 when K = 1.
Correct rate is 0.958333333333 when K = 2.
Correct rate is 0.969696969697 when K = 3.
Correct rate is 0.969696969697 when K = 4.
Correct rate is 0.973484848485 when K = 5.
Correct rate is 0.974747474747 when K = 6.
```

```
Correct rate is 0.97095959596 when K = 7.
Correct rate is 0.974747474747 when K = 8.
Correct rate is 0.977272727273 when K = 9.
Correct rate is 0.973484848485 when K = 10.
Correct rate is 0.97601010101 when K = 11.
Correct rate is 0.97095959596 when K = 12.
Correct rate is 0.974747474747 when K = 13.
Correct rate is 0.973484848485 when K = 14.
Correct rate is 0.97601010101 when K = 15.
Correct rate is 0.973484848485 when K = 16.
Correct rate is 0.977272727273 when K = 17.
Correct rate is 0.974747474747 when K = 18.
Correct rate is 0.978535353535 when K = 19.
Highest correct rate 0.978535353535 occurs at K = 19.
============================================================
with dimensionality reduction using PCA(Built-in Algorithm)
```




```
Lowest error is 0.0239898989899 occurs at k=7.
In-sample accuracy for KNN: 0.9633816261
Out-of-sample accuracy for KNN: 0.9520181514
============================================================
with dimensionality reduction using PCA(Self-Written Algorithm)
```

```
Use self-written KNeighbors Classifier:
Correct rate is 0.969696969697 when K = 1.
Correct rate is 0.960858585859 when K = 2.
Correct rate is 0.965909090909 when K = 3.
Correct rate is 0.968434343434 when K = 4.
Correct rate is 0.973484848485 when K = 5.
Correct rate is 0.974747474747 when K = 6.
Correct rate is 0.97601010101 when K = 7.
Correct rate is 0.972222222222 when K = 8.
Correct rate is 0.967171717172 when K = 9.
Correct rate is 0.97095959596 when K = 10.
Correct rate is 0.967171717172 when K = 11.
Correct rate is 0.967171717172 when K = 12.
Correct rate is 0.969696969697 when K = 13.
Correct rate is 0.965909090909 when K = 14.
Correct rate is 0.968434343434 when K = 15.
Correct rate is 0.968434343434 when K = 16.
Correct rate is 0.969696969697 when K = 17.
Correct rate is 0.969696969697 when K = 18.
Correct rate is 0.967171717172 when K = 19.
Highest correct rate 0.97601010101 occurs at K = 7.
```

In [19]:
```python
import Random_Forest
print("Run random forest with all features(Built-in Algorithm):")
Random_Forest.run_random_forest_classifier(train_x, test_x, train_y, test_y)
print("=================================================================================
print("Run random forest with top 2 features(Built-in Algorithm):")
Random_Forest.run_random_forest_classifier(train_x_two_features, test_x_two_features,
print("=================================================================================
print("Run random forest with dimensionality reduction using PCA(Built-in Algorithm)")
Random_Forest.run_random_forest_classifier(pca_train_x, pca_test_x, train_y, test_y)
```

```
importing Jupyter notebook from Random_Forest.ipynb
Run random forest with all features(Built-in Algorithm):
```

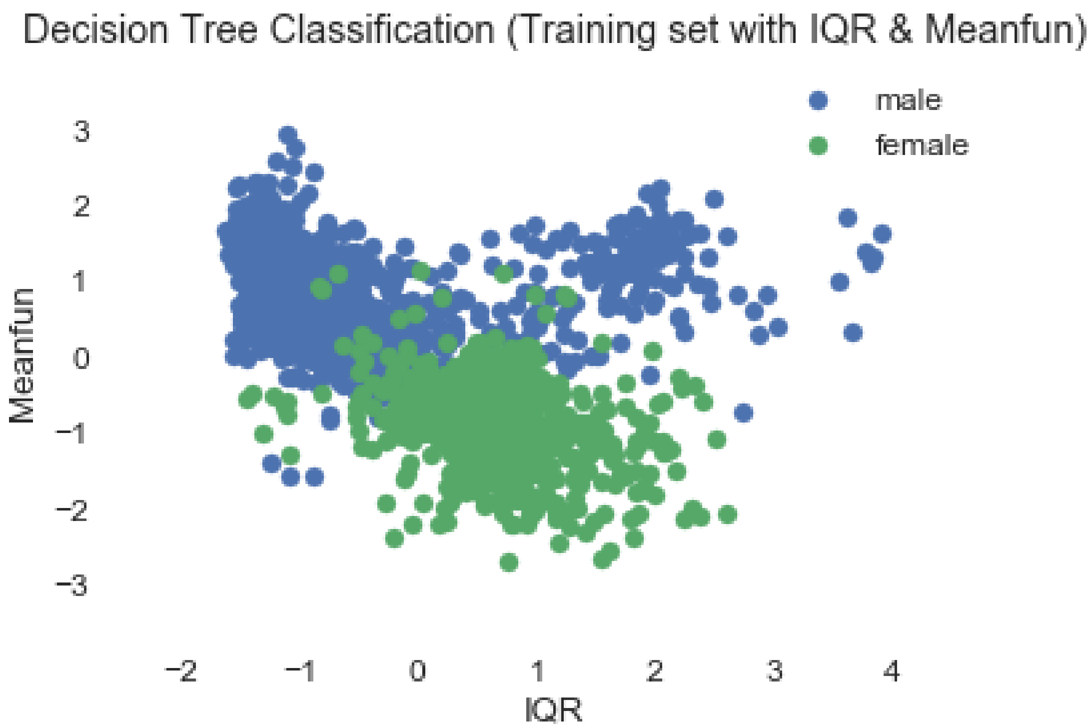

```
Lowest error of 0.0151515151515 occurs at n=241.
The highest in-sample accuracy in Random Forest is 0.984848484848 when n=241.
Out-of-sample accuracy in Random Forest:0.9671522968
================================================================================

Run random forest with top 2 features(Built-in Algorithm):
```

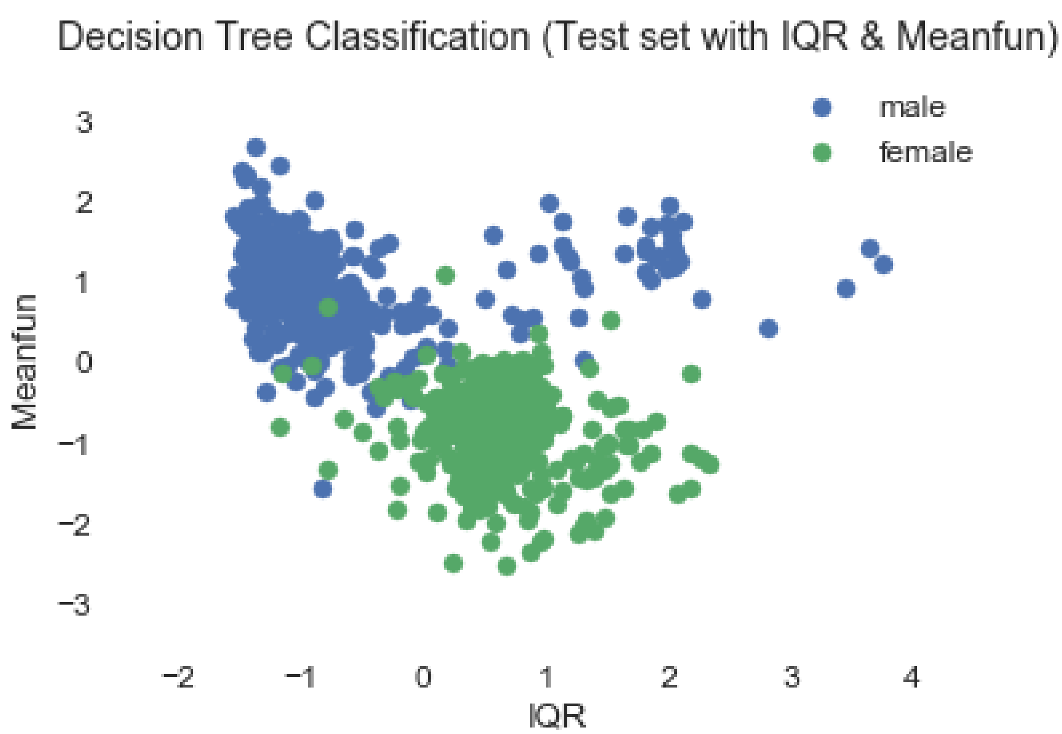

```
Lowest error of 0.030303030303 occurs at n=31.
The highest in-sample accuracy in Random Forest is 0.969696969697 when n=31.
Out-of-sample accuracy in Random Forest:0.9696839424
================================================================================

Run random forest with dimensionality reduction using PCA(Built-in Algorithm)
```


```
Lowest error of 0.030303030303 occurs at n=151.
The highest in-sample accuracy in Random Forest is 0.969696969697 when n=151.
Out-of-sample accuracy in Random Forest:0.9608629886
```

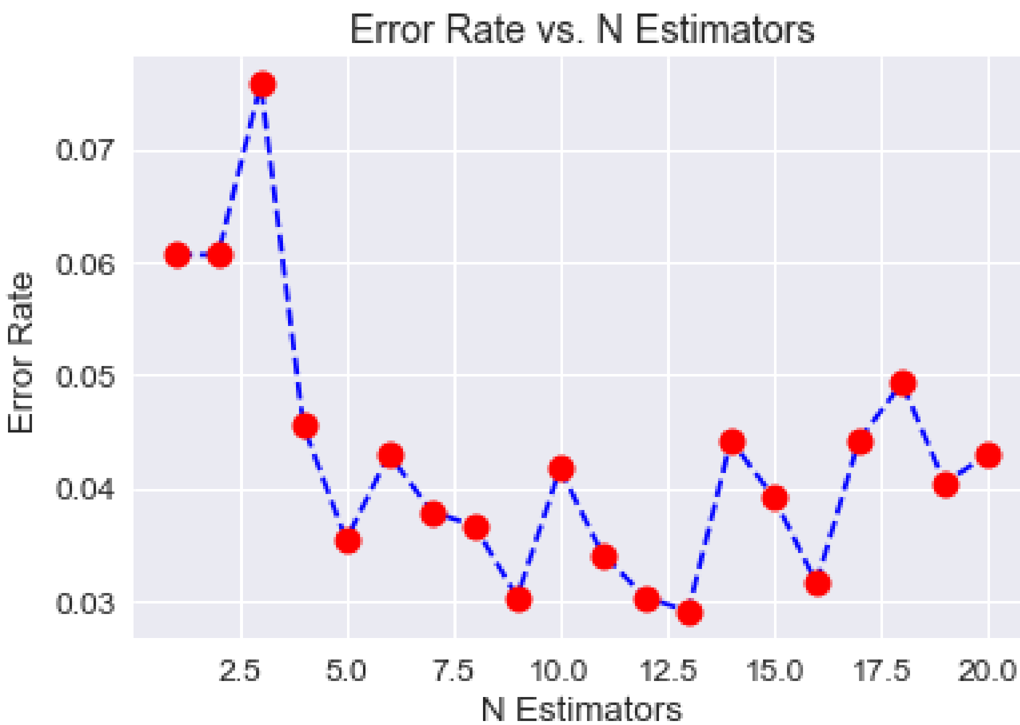
```
In [20]: import Decision_Tree
         print("Run decision tree with all features(Built-in Algorithm):")
         Decision_Tree.run_Decision_tree(train_x, test_x, train_y, test_y)
         print("==============================================================================
         print("Run decision tree with top 2 features(Built-in Algorithm):")
         Decision_Tree.run_Decision_tree(train_x_two_features, test_x_two_features, train_y_two
         print("==============================================================================
         print("Run decision tree with dimensionality reduction using PCA(Built-in Algorithm)")
         Decision_Tree.run_Decision_tree(pca_train_x, pca_test_x, train_y, test_y)
```

```
importing Jupyter notebook from Decision_Tree.ipynb
```

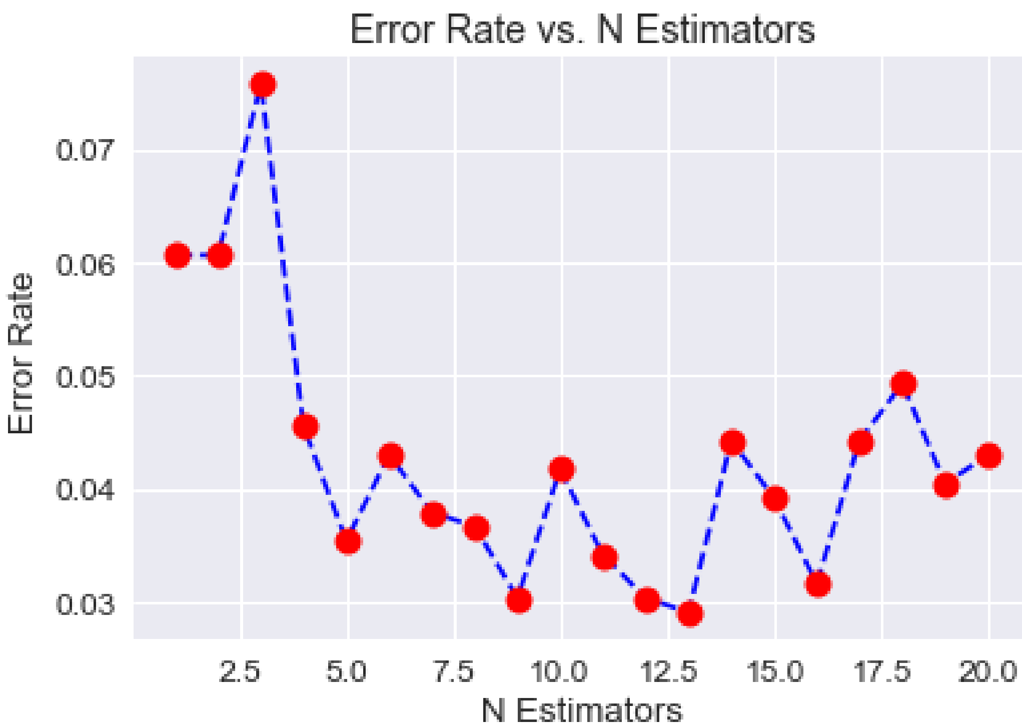

Decision Tree Classification (Training set with IQR & Meanfun)
male
female
Meanfun
IQR
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4


Decision Tree Classification (Test set with IQR & Meanfun)
male
female
Meanfun
IQR
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4

```
In-Sample Accuracy:100.0000%
Out-of-Sample Accuracy:95.8333%
```




```
Lowest error of 0.0290404040404 occurs at n=13.
In-Sample Accuracy:100.0000%
Out-of-Sample Accuracy:95.7071%

with all features(Built-in Algorithm):
```




```
Lowest error of 0.0290404040404 occurs at n=13.
In-Sample Accuracy:100.0000%
Out-of-Sample Accuracy:95.7071%
```

```
In [21]: from sklearn.linear_model import LogisticRegression
         import Logistic_Regression
         print("Run Logistic Regression with all features(Built-in Algorithm):")
         Logistic_Regression.run_Logistic_Regression(train_x,test_x,train_y,test_y)
         print("===============================================================================
         print("Run Logistic Regression with top 2 features(Built-in Algorithm):")
         Logistic_Regression.run_Logistic_Regression(train_x_two_features, test_x_two_features
         print("===============================================================================
         print("Run Logistic Regression with dimensionality reduction using PCA(Built-in Algor
         Logistic_Regression.run_Logistic_Regression(pca_train_x, pca_test_x, train_y, test_y)
```

```
importing Jupyter notebook from Logistic_Regression.ipynb
In-sample accuracy for Logistic Regression: 0.9718035894
Out-of-sample accuracy for Logistic Regression: 0.9747392724
Run Logistic Regression with all features(Built-in Algorithm):
In-sample accuracy for Logistic Regression: 0.9718035894
Out-of-sample accuracy for Logistic Regression: 0.9747392724
===============================================================================
```

```
Run Logistic Regression with top 2 features(Built-in Algorithm):
In-sample accuracy for Logistic Regression: 0.9633949132
Out-of-sample accuracy for Logistic Regression: 0.9696759812
===============================================================================

Run Logistic Regression with dimensionality reduction using PCA(Built-in Algorithm)
In-sample accuracy for Logistic Regression: 0.9726456947
Out-of-sample accuracy for Logistic Regression: 0.9747392724
```

```
In [28]: import Neural_Network
         print("Run Neural Network with all features(Built-in Algorithm):")
         Neural_Network.run_neural_network(train_x,train_y,test_x,test_y)
         print("===============================================================================
         print("Run Neural Network with all features(self-written Algorithm):")
         Neural_Network.run_my_neural_network(train_x,train_y,test_x,test_y)
```

```
Run Neural Network with all features(Built-in Algorithm):
In-sample accuracy in Neural Network using Keras package (with all features):1.0
Out-of-sample accuracy in Neural Network using Keras package (with 2 features):0.982323232323
===============================================================================

Run Neural Network with all features(self-written Algorithm):
Epoch 1: cost=0.7508
Epoch 2: cost=0.7387
Epoch 3: cost=0.7233
Epoch 4: cost=0.7041
Epoch 5: cost=0.6809
Epoch 6: cost=0.6547
Epoch 7: cost=0.6267
Epoch 8: cost=0.5983
Epoch 9: cost=0.5714
Epoch 10: cost=0.5464
Epoch 11: cost=0.5234
Epoch 12: cost=0.5027
Epoch 13: cost=0.4842
Epoch 14: cost=0.4680
Epoch 15: cost=0.4539
Epoch 16: cost=0.4414
Epoch 17: cost=0.4305
Epoch 18: cost=0.4210
Epoch 19: cost=0.4125
Epoch 20: cost=0.4051
Epoch 21: cost=0.3984
Epoch 22: cost=0.3924
Epoch 23: cost=0.3871
Epoch 24: cost=0.3823
```

```
Epoch 25: cost=0.3780
Epoch 26: cost=0.3742
Epoch 27: cost=0.3707
Epoch 28: cost=0.3675
Epoch 29: cost=0.3646
Epoch 30: cost=0.3620
Epoch 31: cost=0.3596
Epoch 32: cost=0.3573
Epoch 33: cost=0.3552
Epoch 34: cost=0.3533
Epoch 35: cost=0.3515
Epoch 36: cost=0.3498
Epoch 37: cost=0.3483
Epoch 38: cost=0.3469
Epoch 39: cost=0.3456
Epoch 40: cost=0.3443
Epoch 41: cost=0.3431
Epoch 42: cost=0.3419
Epoch 43: cost=0.3407
Epoch 44: cost=0.3396
Epoch 45: cost=0.3386
Epoch 46: cost=0.3376
Epoch 47: cost=0.3366
Epoch 48: cost=0.3357
Epoch 49: cost=0.3348
Epoch 50: cost=0.3340
Epoch 51: cost=0.3332
Epoch 52: cost=0.3325
Epoch 53: cost=0.3318
Epoch 54: cost=0.3312
Epoch 55: cost=0.3306
Epoch 56: cost=0.3301
Epoch 57: cost=0.3296
Epoch 58: cost=0.3291
Epoch 59: cost=0.3286
Epoch 60: cost=0.3281
Epoch 61: cost=0.3276
Epoch 62: cost=0.3272
Epoch 63: cost=0.3268
Epoch 64: cost=0.3264
Epoch 65: cost=0.3260
Epoch 66: cost=0.3257
Epoch 67: cost=0.3253
Epoch 68: cost=0.3250
Epoch 69: cost=0.3247
Epoch 70: cost=0.3244
Epoch 71: cost=0.3241
Epoch 72: cost=0.3239
```

```
Epoch 73: cost=0.3236
Epoch 74: cost=0.3234
Epoch 75: cost=0.3232
Epoch 76: cost=0.3229
Epoch 77: cost=0.3227
Epoch 78: cost=0.3225
Epoch 79: cost=0.3223
Epoch 80: cost=0.3221
Epoch 81: cost=0.3219
Epoch 82: cost=0.3218
Epoch 83: cost=0.3216
Epoch 84: cost=0.3215
Epoch 85: cost=0.3213
Epoch 86: cost=0.3212
Epoch 87: cost=0.3211
Epoch 88: cost=0.3209
Epoch 89: cost=0.3208
Epoch 90: cost=0.3207
Epoch 91: cost=0.3206
Epoch 92: cost=0.3204
Epoch 93: cost=0.3203
Epoch 94: cost=0.3202
Epoch 95: cost=0.3201
Epoch 96: cost=0.3200
Epoch 97: cost=0.3199
Epoch 98: cost=0.3199
Epoch 99: cost=0.3198
Epoch 100: cost=0.3196
Epoch 101: cost=0.3195
Epoch 102: cost=0.3194
Epoch 103: cost=0.3193
Epoch 104: cost=0.3192
Epoch 105: cost=0.3191
Epoch 106: cost=0.3190
Epoch 107: cost=0.3190
Epoch 108: cost=0.3189
Epoch 109: cost=0.3189
Epoch 110: cost=0.3188
Epoch 111: cost=0.3188
Epoch 112: cost=0.3187
Epoch 113: cost=0.3187
Epoch 114: cost=0.3187
Epoch 115: cost=0.3186
Epoch 116: cost=0.3186
Epoch 117: cost=0.3185
Epoch 118: cost=0.3185
Epoch 119: cost=0.3185
Epoch 120: cost=0.3184
```

```
Epoch 121: cost=0.3184
Epoch 122: cost=0.3184
Epoch 123: cost=0.3184
Epoch 124: cost=0.3183
Epoch 125: cost=0.3183
Epoch 126: cost=0.3183
Epoch 127: cost=0.3183
Epoch 128: cost=0.3182
Epoch 129: cost=0.3182
Epoch 130: cost=0.3182
Epoch 131: cost=0.3182
Epoch 132: cost=0.3181
Epoch 133: cost=0.3181
Epoch 134: cost=0.3181
Epoch 135: cost=0.3181
Epoch 136: cost=0.3181
Epoch 137: cost=0.3181
Epoch 138: cost=0.3180
Epoch 139: cost=0.3180
Epoch 140: cost=0.3180
Epoch 141: cost=0.3180
Epoch 142: cost=0.3180
Epoch 143: cost=0.3180
Epoch 144: cost=0.3180
Epoch 145: cost=0.3180
Epoch 146: cost=0.3179
Epoch 147: cost=0.3179
Epoch 148: cost=0.3179
Epoch 149: cost=0.3179
Epoch 150: cost=0.3179
Epoch 151: cost=0.3179
Epoch 152: cost=0.3179
Epoch 153: cost=0.3179
Epoch 154: cost=0.3179
Epoch 155: cost=0.3179
Epoch 156: cost=0.3179
Epoch 157: cost=0.3178
Epoch 158: cost=0.3178
Epoch 159: cost=0.3178
Epoch 160: cost=0.3178
Epoch 161: cost=0.3178
Epoch 162: cost=0.3178
Epoch 163: cost=0.3178
Epoch 164: cost=0.3178
Epoch 165: cost=0.3178
Epoch 166: cost=0.3178
Epoch 167: cost=0.3178
Epoch 168: cost=0.3178
```

```
Epoch 169: cost=0.3178
Epoch 170: cost=0.3178
Epoch 171: cost=0.3178
Epoch 172: cost=0.3178
Epoch 173: cost=0.3178
Epoch 174: cost=0.3177
Epoch 175: cost=0.3177
Epoch 176: cost=0.3177
Epoch 177: cost=0.3177
Epoch 178: cost=0.3177
Epoch 179: cost=0.3177
Epoch 180: cost=0.3177
Epoch 181: cost=0.3177
Epoch 182: cost=0.3177
Epoch 183: cost=0.3177
Epoch 184: cost=0.3177
Epoch 185: cost=0.3177
Epoch 186: cost=0.3177
Epoch 187: cost=0.3177
Epoch 188: cost=0.3177
Epoch 189: cost=0.3177
Epoch 190: cost=0.3177
Epoch 191: cost=0.3177
Epoch 192: cost=0.3177
Epoch 193: cost=0.3177
Epoch 194: cost=0.3177
Epoch 195: cost=0.3177
Epoch 196: cost=0.3177
Epoch 197: cost=0.3177
Epoch 198: cost=0.3177
Epoch 199: cost=0.3177
Epoch 200: cost=0.3177
Epoch 201: cost=0.3177
Epoch 202: cost=0.3177
Epoch 203: cost=0.3177
Epoch 204: cost=0.3177
Epoch 205: cost=0.3177
Epoch 206: cost=0.3177
Epoch 207: cost=0.3177
Epoch 208: cost=0.3177
Epoch 209: cost=0.3177
Epoch 210: cost=0.3176
Epoch 211: cost=0.3176
Epoch 212: cost=0.3176
Epoch 213: cost=0.3176
Epoch 214: cost=0.3176
Epoch 215: cost=0.3176
Epoch 216: cost=0.3176
```

```
Epoch 217: cost=0.3176
Epoch 218: cost=0.3176
Epoch 219: cost=0.3176
Epoch 220: cost=0.3176
Epoch 221: cost=0.3176
Epoch 222: cost=0.3176
Epoch 223: cost=0.3176
Epoch 224: cost=0.3176
Epoch 225: cost=0.3176
Epoch 226: cost=0.3176
Epoch 227: cost=0.3176
Epoch 228: cost=0.3176
Epoch 229: cost=0.3176
Epoch 230: cost=0.3176
Epoch 231: cost=0.3176
Epoch 232: cost=0.3176
Epoch 233: cost=0.3176
Epoch 234: cost=0.3176
Epoch 235: cost=0.3176
Epoch 236: cost=0.3176
Epoch 237: cost=0.3176
Epoch 238: cost=0.3176
Epoch 239: cost=0.3176
Epoch 240: cost=0.3176
Epoch 241: cost=0.3176
Epoch 242: cost=0.3176
Epoch 243: cost=0.3176
Epoch 244: cost=0.3176
Epoch 245: cost=0.3176
Epoch 246: cost=0.3176
Epoch 247: cost=0.3176
Epoch 248: cost=0.3176
Epoch 249: cost=0.3176
Epoch 250: cost=0.3176
Epoch 251: cost=0.3176
Epoch 252: cost=0.3176
Epoch 253: cost=0.3176
Epoch 254: cost=0.3176
Epoch 255: cost=0.3176
Epoch 256: cost=0.3176
Epoch 257: cost=0.3176
Epoch 258: cost=0.3176
Epoch 259: cost=0.3176
Epoch 260: cost=0.3176
Epoch 261: cost=0.3176
Epoch 262: cost=0.3176
Epoch 263: cost=0.3176
Epoch 264: cost=0.3176
```

```
Epoch 265: cost=0.3176
Epoch 266: cost=0.3176
Epoch 267: cost=0.3176
Epoch 268: cost=0.3176
Epoch 269: cost=0.3176
Epoch 270: cost=0.3176
Epoch 271: cost=0.3176
Epoch 272: cost=0.3176
Epoch 273: cost=0.3176
Epoch 274: cost=0.3176
Epoch 275: cost=0.3176
Epoch 276: cost=0.3176
Epoch 277: cost=0.3176
Epoch 278: cost=0.3176
Epoch 279: cost=0.3176
Epoch 280: cost=0.3176
Epoch 281: cost=0.3176
Epoch 282: cost=0.3176
Epoch 283: cost=0.3176
Epoch 284: cost=0.3176
Epoch 285: cost=0.3176
Epoch 286: cost=0.3176
Epoch 287: cost=0.3176
Epoch 288: cost=0.3176
Epoch 289: cost=0.3176
Epoch 290: cost=0.3176
Epoch 291: cost=0.3176
Epoch 292: cost=0.3176
Epoch 293: cost=0.3176
Epoch 294: cost=0.3176
Epoch 295: cost=0.3176
Epoch 296: cost=0.3176
Epoch 297: cost=0.3176
Epoch 298: cost=0.3176
Epoch 299: cost=0.3176
Epoch 300: cost=0.3176
Epoch 301: cost=0.3176
Epoch 302: cost=0.3176
Epoch 303: cost=0.3176
Epoch 304: cost=0.3176
Epoch 305: cost=0.3176
Epoch 306: cost=0.3176
Epoch 307: cost=0.3176
Epoch 308: cost=0.3176
Epoch 309: cost=0.3176
Epoch 310: cost=0.3176
Epoch 311: cost=0.3176
Epoch 312: cost=0.3176
```

```
Epoch 313: cost=0.3176
Epoch 314: cost=0.3176
Epoch 315: cost=0.3176
Epoch 316: cost=0.3176
Epoch 317: cost=0.3176
Epoch 318: cost=0.3176
Epoch 319: cost=0.3176
Epoch 320: cost=0.3176
Epoch 321: cost=0.3176
Epoch 322: cost=0.3176
Epoch 323: cost=0.3176
Epoch 324: cost=0.3176
Epoch 325: cost=0.3176
Epoch 326: cost=0.3176
Epoch 327: cost=0.3176
Epoch 328: cost=0.3176
Epoch 329: cost=0.3176
Epoch 330: cost=0.3176
Epoch 331: cost=0.3176
Epoch 332: cost=0.3176
Epoch 333: cost=0.3176
Epoch 334: cost=0.3176
Epoch 335: cost=0.3176
Epoch 336: cost=0.3176
Epoch 337: cost=0.3176
Epoch 338: cost=0.3176
Epoch 339: cost=0.3176
Epoch 340: cost=0.3176
Epoch 341: cost=0.3176
Epoch 342: cost=0.3176
Epoch 343: cost=0.3176
Epoch 344: cost=0.3176
Epoch 345: cost=0.3176
Epoch 346: cost=0.3176
Epoch 347: cost=0.3176
Epoch 348: cost=0.3176
Epoch 349: cost=0.3176
Epoch 350: cost=0.3176
Epoch 351: cost=0.3176
Epoch 352: cost=0.3176
Epoch 353: cost=0.3176
Epoch 354: cost=0.3176
Epoch 355: cost=0.3176
Epoch 356: cost=0.3176
Epoch 357: cost=0.3176
Epoch 358: cost=0.3176
Epoch 359: cost=0.3176
Epoch 360: cost=0.3176
```

```
Epoch 361: cost=0.3176
Epoch 362: cost=0.3176
Epoch 363: cost=0.3176
Epoch 364: cost=0.3176
Epoch 365: cost=0.3176
Epoch 366: cost=0.3176
Epoch 367: cost=0.3176
Epoch 368: cost=0.3176
Epoch 369: cost=0.3176
Epoch 370: cost=0.3176
Epoch 371: cost=0.3176
Epoch 372: cost=0.3176
Epoch 373: cost=0.3176
Epoch 374: cost=0.3176
Epoch 375: cost=0.3176
Epoch 376: cost=0.3176
Epoch 377: cost=0.3176
Epoch 378: cost=0.3176
Epoch 379: cost=0.3176
Epoch 380: cost=0.3176
Epoch 381: cost=0.3176
Epoch 382: cost=0.3176
Epoch 383: cost=0.3176
Epoch 384: cost=0.3176
Epoch 385: cost=0.3176
Epoch 386: cost=0.3176
Epoch 387: cost=0.3176
Epoch 388: cost=0.3176
Epoch 389: cost=0.3176
Epoch 390: cost=0.3176
Epoch 391: cost=0.3176
Epoch 392: cost=0.3176
Epoch 393: cost=0.3176
Epoch 394: cost=0.3176
Epoch 395: cost=0.3176
Epoch 396: cost=0.3176
Epoch 397: cost=0.3176
Epoch 398: cost=0.3176
Epoch 399: cost=0.3176
Epoch 400: cost=0.3176
Epoch 401: cost=0.3176
Epoch 402: cost=0.3176
Epoch 403: cost=0.3176
Epoch 404: cost=0.3176
Epoch 405: cost=0.3176
Epoch 406: cost=0.3176
Epoch 407: cost=0.3176
Epoch 408: cost=0.3176
```

```
Epoch 409: cost=0.3176
Epoch 410: cost=0.3176
Epoch 411: cost=0.3176
Epoch 412: cost=0.3176
Epoch 413: cost=0.3176
Epoch 414: cost=0.3176
Epoch 415: cost=0.3176
Epoch 416: cost=0.3176
Epoch 417: cost=0.3176
Epoch 418: cost=0.3176
Epoch 419: cost=0.3176
Epoch 420: cost=0.3176
Epoch 421: cost=0.3176
Epoch 422: cost=0.3176
Epoch 423: cost=0.3176
Epoch 424: cost=0.3176
Epoch 425: cost=0.3176
Epoch 426: cost=0.3176
Epoch 427: cost=0.3176
Epoch 428: cost=0.3176
Epoch 429: cost=0.3176
Epoch 430: cost=0.3176
Epoch 431: cost=0.3176
Epoch 432: cost=0.3176
Epoch 433: cost=0.3176
Epoch 434: cost=0.3176
Epoch 435: cost=0.3176
Epoch 436: cost=0.3176
Epoch 437: cost=0.3176
Epoch 438: cost=0.3176
Epoch 439: cost=0.3176
Epoch 440: cost=0.3176
Epoch 441: cost=0.3176
Epoch 442: cost=0.3176
Epoch 443: cost=0.3176
Epoch 444: cost=0.3176
Epoch 445: cost=0.3176
Epoch 446: cost=0.3176
Epoch 447: cost=0.3176
Epoch 448: cost=0.3176
Epoch 449: cost=0.3176
Epoch 450: cost=0.3176
Epoch 451: cost=0.3176
Epoch 452: cost=0.3176
Epoch 453: cost=0.3176
Epoch 454: cost=0.3176
Epoch 455: cost=0.3176
Epoch 456: cost=0.3176
```

```
Epoch 457: cost=0.3176
Epoch 458: cost=0.3176
Epoch 459: cost=0.3176
Epoch 460: cost=0.3176
Epoch 461: cost=0.3176
Epoch 462: cost=0.3176
Epoch 463: cost=0.3176
Epoch 464: cost=0.3176
Epoch 465: cost=0.3176
Epoch 466: cost=0.3176
Epoch 467: cost=0.3176
Epoch 468: cost=0.3176
Epoch 469: cost=0.3176
Epoch 470: cost=0.3176
Epoch 471: cost=0.3176
Epoch 472: cost=0.3176
Epoch 473: cost=0.3176
Epoch 474: cost=0.3176
Epoch 475: cost=0.3176
Epoch 476: cost=0.3176
Epoch 477: cost=0.3176
Epoch 478: cost=0.3176
Epoch 479: cost=0.3176
Epoch 480: cost=0.3176
Epoch 481: cost=0.3176
Epoch 482: cost=0.3176
Epoch 483: cost=0.3176
Epoch 484: cost=0.3176
Epoch 485: cost=0.3176
Epoch 486: cost=0.3176
Epoch 487: cost=0.3176
Epoch 488: cost=0.3176
Epoch 489: cost=0.3176
Epoch 490: cost=0.3176
Epoch 491: cost=0.3176
Epoch 492: cost=0.3176
Epoch 493: cost=0.3176
Epoch 494: cost=0.3176
Epoch 495: cost=0.3176
Epoch 496: cost=0.3176
Epoch 497: cost=0.3176
Epoch 498: cost=0.3176
Epoch 499: cost=0.3176
Epoch 500: cost=0.3176
In-sample accuracy in Neural Network: 0.99495
Out-of-sample accuracy in Neural Network: 0.97601
```

```
In [29]: import Gaussian_Naive_Bayes
         print("Run Gaussian Naive Bayes with all features(Built-in Algorithm):")
         Gaussian_Naive_Bayes.run_naive_bayes(train_x, test_x, train_y, test_y)
         print("================================================================================
         print("Run Gaussian Naive Bayes with all features(Self-Written Algorithm):")
         Gaussian_Naive_Bayes.run_my_gaussian_naive_bayes(train_x, test_x, train_y, test_y)
         print("================================================================================
         print("Run Gaussian Naive Bayes with top 2 features(Built-in Algorithm):")
         Gaussian_Naive_Bayes.run_naive_bayes(train_x_two_features, test_x_two_features, train_
         print("================================================================================
         print("Run Gaussian Naive Bayes with dimensionality reduction using PCA(Built-in Algo
         Gaussian_Naive_Bayes.run_naive_bayes(pca_train_x, pca_test_x, train_y, test_y)
```

```
importing Jupyter notebook from Gaussian_Naive_Bayes.ipynb
Correct rate is 0.900252525253
             precision    recall  f1-score   support

          0       0.89      0.90      0.89       367
          1       0.91      0.90      0.91       425

avg / total       0.90      0.90      0.90       792

Run Gaussian Naive Bayes with all features(Self-Written Algorithm):
Correct rate is 0.900252525253
             precision    recall  f1-score   support

          0       0.89      0.90      0.89       367
          1       0.91      0.90      0.91       425

avg / total       0.90      0.90      0.90       792

===============================================================================

Run Gaussian Naive Bayes with top 2 features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.9659123721
Out-of-sample accuracy for svm with Linear kernel: 0.9747313112
===============================================================================

Run Gaussian Naive Bayes with dimensionality reduction using PCA(Built-in Algorithm)
In-sample accuracy for svm with Linear kernel: 0.9482405343
Out-of-sample accuracy for svm with Linear kernel: 0.9507523286
Run Gaussian Naive Bayes with all features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.8943739953
Out-of-sample accuracy for svm with Linear kernel: 0.8711328716
===============================================================================

Run Gaussian Naive Bayes with all features(Self-Written Algorithm):
Correct rate is 0.900252525253
```

```
             precision    recall  f1-score   support

          0       0.89      0.90      0.89       367
          1       0.91      0.90      0.91       425

avg / total       0.90      0.90      0.90       792

===============================================================================

Run Gaussian Naive Bayes with top 2 features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.9659123721
Out-of-sample accuracy for svm with Linear kernel: 0.9747313112
===============================================================================

Run Gaussian Naive Bayes with dimensionality reduction using PCA(Built-in Algorithm)
In-sample accuracy for svm with Linear kernel: 0.9482405343
Out-of-sample accuracy for svm with Linear kernel: 0.9507523286
```

In [30]:
```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
from sklearn import svm
def getModels(train_x, test_x, train_y, test_y):
    # KNN
    error_rate = []
    kvals = range(1,21)  # range of k parameters to test
    for i in kvals:
        knn = KNeighborsClassifier(n_neighbors=i)
        knn.fit(train_x,train_y)
        pred_y_i = knn.predict(test_x)
        error_rate.append(np.mean(pred_y_i != test_y))
    kloc = error_rate.index(min(error_rate))
    clf = KNeighborsClassifier(kvals[kloc], 'uniform')
    clf1 =clf.fit(train_x, train_y)

    # Decision Tree
    clf2 = tree.DecisionTreeClassifier()
    clf2 = clf2.fit(train_x, train_y)

    # Random Forest
    error_rate = []
    nvals = range(1,301,30)
    for i in nvals:
        clf = RandomForestClassifier(n_estimators=i)
        clf.fit(train_x,train_y)
        pred_y_i = clf.predict(test_x)
```

```python
                error_rate.append(np.mean(pred_y_i != test_y))
            nloc = error_rate.index(min(error_rate))
            clf3 = RandomForestClassifier(n_estimators= nvals[nloc])
            clf3 = clf3.fit(train_x, train_y)

            # SVM
            clf4 = svm.SVC()
            clf4 = clf4.fit(train_x, train_y)

            # Naive Bayes
            clf5 = GaussianNB()
            clf5 = clf5.fit(train_x, train_y)

            return [clf1, clf2, clf3, clf4, clf5]
```

In [31]:
```python
# Majority vote
def run_majority_voting(train_x, test_x, train_y, test_y):
    models = getModels(train_x, test_x, train_y, test_y);
    correct = 0;
    for i in range(len(test_x)):
        count = 0;
        for j in range(len(models)):
            if (models[j].predict(test_x[i].reshape(1,-1))[0] == test_y[i]):
                count = count + 1;
            else:
                count = count - 1;
        if (count > 0):
            correct = correct + 1;

    accuracy = (correct*1.0)/len(test_x)
    print("Majority vote accuracy : %.10f" % accuracy)
```

In [32]:
```python
print("Run majority voting with all features(Built-in Algorithm):")
run_majority_voting(train_x, test_x, train_y, test_y)
print("=======================================================================
print("Run majority voting with top 2 features(Built-in Algorithm):")
run_majority_voting(train_x_two_features, test_x_two_features, train_y_two_features, t
print("=======================================================================
print("Run majority voting with dimensionality reduction using PCA(Built-in Algorithm
run_majority_voting(pca_train_x, pca_test_x, train_y, test_y)
```

```
Run majority voting with all features(Built-in Algorithm):
Majority vote accuracy : 0.9823232323
===============================================================================
Run majority voting with top 2 features(Built-in Algorithm):
Majority vote accuracy : 0.9747474747
===============================================================================
Run majority voting with dimensionality reduction using PCA(Built-in Algorithm)
```

```
Majority vote accuracy : 0.9747474747
```

```
In [33]: # Our test data
         path = '/Users/Kassi/Desktop/Gender_Recognition_by_Voice/voice_test.csv'
         voice_data_test = pd.read_csv(path)
         print("Total number of samples: {}".format(voice_data_test.shape[0]))
         print("Total number of male: {}".format(voice_data_test[voice_data_test.label == 'male
         print("Total number of female: {}".format(voice_data_test[voice_data_test.label == 'fe
         print("Correlation between each feature")
```

```
Total number of samples: 460
Total number of male: 230
Total number of female: 230
Correlation between each feature
```

```
In [34]: #### Check if dataset contains NA's
         voice_data_test.isnull().any().any()
```

```
Out[34]: False
```

```
In [35]: voice_data_test = voice_data_test.values
         voices_test = voice_data_test[:, :-1]
         labels_test = voice_data_test[:, -1:]
```

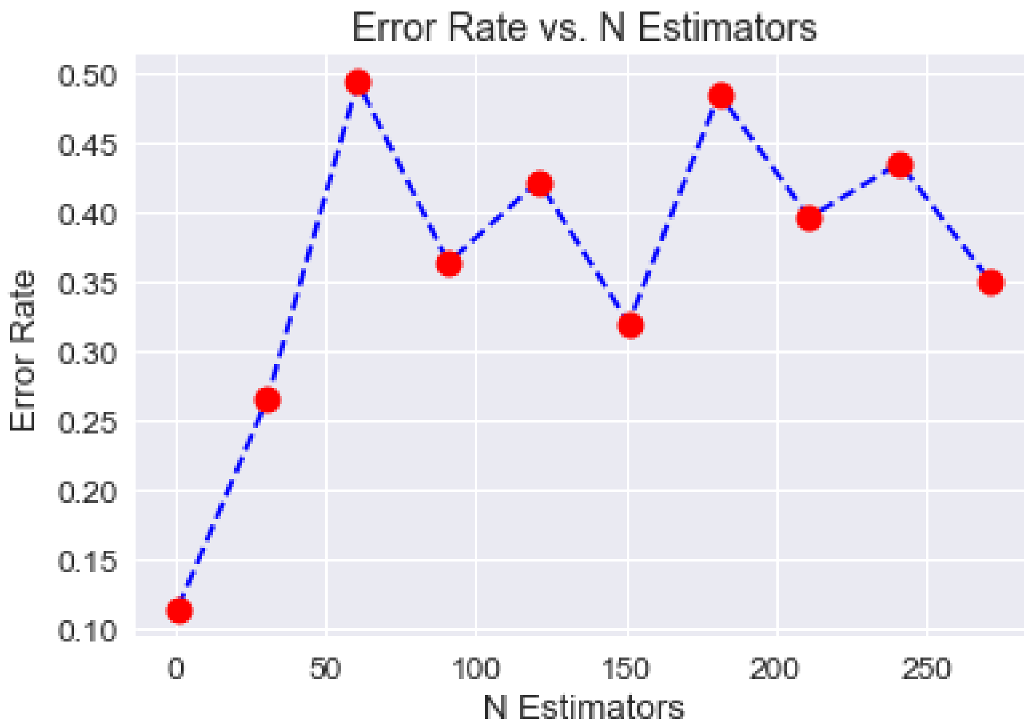
```
In [36]: gender_encoder = LabelEncoder()
         labels_test = gender_encoder.fit_transform(labels_test)
         # labels_test
```

```
In [37]: # randomly shuffle our data
         voices_tmp = []
         lables_tmp = []
         index_shuf = range(len(voices_test))
         random.shuffle(index_shuf)
         for i in index_shuf:
             voices_tmp.append(voices_test[i])
             lables_tmp.append(labels_test[i])
         voices_test = np.array(voices_tmp)
         labels_test = np.array(lables_tmp)
```

```
In [38]: print("Run random forest with all features(Built-in Algorithm):")
         Random_Forest.run_random_forest_classifier(train_x, voices_test, train_y,labels_test)
         print("=================================================================================
         print("Run majority voting with all features(Built-in Algorithm):")
         run_majority_voting(train_x, voices_test, train_y, labels_test)
         print("=================================================================================
         print("Run linear svm with all features(Built-in Algorithm):")
         SVM.run_linear_svm(train_x, voices_test, train_y, labels_test)
```

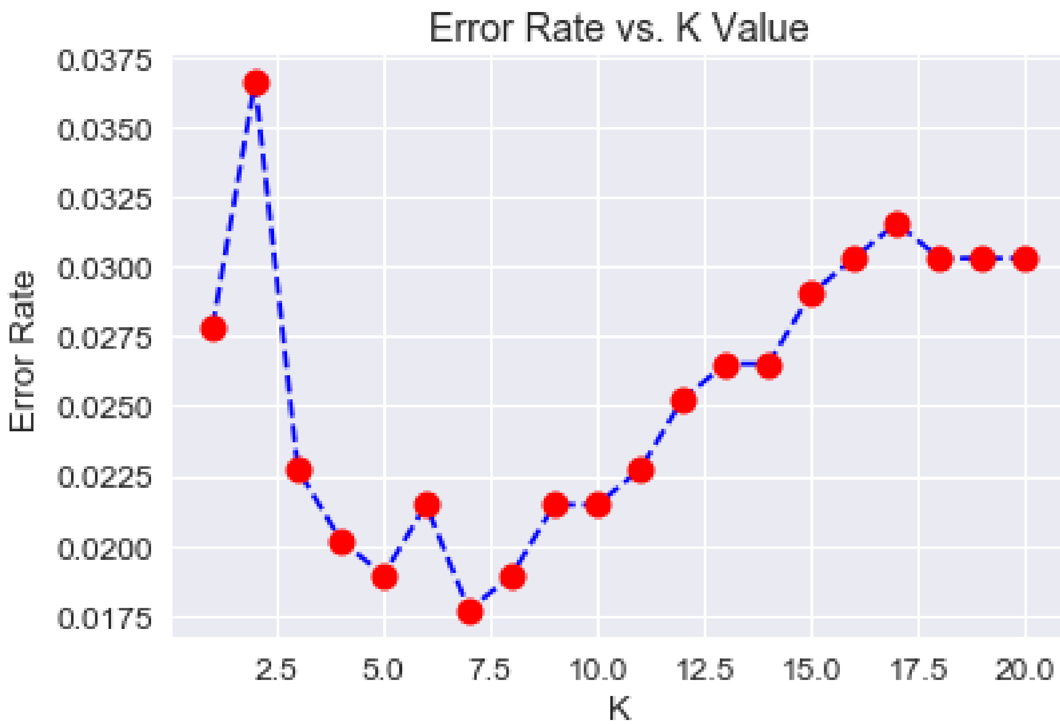
```
print("=================================================================================
print("Run rbf svm with all features(Built-in Algorithm):")
SVM.run_rbf_svm(train_x, voices_test, train_y, labels_test)
print("=================================================================================
print("with all features(Built-in Algorithm):")
KNN.run_k_nearest_neighbour(train_x, test_x, train_y, test_y)
```

```
Run random forest with all features(Built-in Algorithm):
```




```
Lowest error of 0.113043478261 occurs at n=1.
The highest in-sample accuracy in Random Forest is 0.886956521739 when n=1.
Out-of-sample accuracy in Random Forest:0.7543478261
===============================================================================

Run majority voting with all features(Built-in Algorithm):
Majority vote accuracy : 0.6478260870
===============================================================================

Run linear svm with all features(Built-in Algorithm):
In-sample accuracy for svm with Linear kernel: 0.9743290243
Out-of-sample accuracy for svm with Linear kernel: 0.7913043478
===============================================================================
```

```
Run rbf svm with all features(Built-in Algorithm):
In-sample accuracy for svm with RBF kernel: 0.9806448362
Out-of-sample accuracy for svm with RBF kernel: 0.5847826087
================================================================================

with all features(Built-in Algorithm):
```




```
Lowest error is 0.0176767676768 occurs at k=7.
In-sample accuracy for KNN: 0.9696947768
Out-of-sample accuracy for KNN: 0.9507602898
```

```
In [ ]: # END:OWN CODE
```