Predicting the Contraceptive Method (By Udeshika Dissanayake) ¶

The objective of this study is to predict the contraceptive method (no use, long-term methods,or short-term methods) of a woman based on her demographic and socio-economic characteristics.

A data-set of 1473 married women with their demographic and socio-economic characteristics is used in this study. The Source for the data-set is the UCI Machine Learning Repository at, http://archive.ics.uci.edu/ml/datasets/Contraceptive+Method+Choice (http://archive.ics.uci.edu/ml/datasets/Contraceptive+Method+Choice) *(Dua, D. and Graff, C., 2013)*.

This study consists with two phases. Objective of the Phase I is to preprocess and explore the data-set in order to make it ready for Phase II. The objective of phase II is to build a model that predicts the contraceptive method of a women based on here demograpic and socio-econimic characteristics.

All the activities have been performed in `Python` package in this study and compiled from Jupyter Notebook (http://jupyter.org/).

This report does not cover the Phase I scope of work (Phase I report has been submitted under the previous assignment submission under this course). This report covers both narratives and the `Python` codesudes for the model building & evaluation which performed under the phase II.

Content of this report is organized as follows.

2. Overview

Data-set

The date-set contains contraceptive methods used & nine other demographic and socio-economic characteristics of 1473 married women in Indonesia, which obtains from National Indonesia Contraceptive Prevalence Survey in 1987. The data-set has 9 descriptive features and one target feature.

Target Feature

The response feature is contraceptive method which is given as:

$$\text{contraceptive method} = \begin{cases} long-term & \text{if the contraceptive method is long term} \\ short-term & \text{if the contraceptive method is short term} \\ no-use & \text{if no contraceptive method is used} \end{cases}$$

The target feature has three classes. Hence this can be classified as multinominal (multiclass) target feature.

Descriptive Features

Following are the descriptive features in the data-set.

- **`Wife's age`** : numerical
- **`Wife's education`** : categorical (low, medium low, medium high, high)
- **`Husband's education`** : categorical (low, medium low, medium high, high)
- **`Number of children ever born`** : numerical
- **`Wife's religion`** : binary (Non-Islam, Islam)
- **`Wife's now working?`** : binary (Yes, No)
- **`Husband's occupation`** : categorical (Cat1, Cat2, Cat3, Cat4)
- **`Standard-of-living index`** : categorical (low, medium low, medium high, high)
- **`Media exposure`** : binary (Good, Bad)

All the descriptive features are self-explanatory.

Methodology

The data-set that had been preprocessed under Phase I, is now being used in this Phase of work. The data-set has been further preprocessed in order to use it in Scikit-Learn functions. Those additional preprocessed steps that are used under this Phase are,

```
* Checking for missing values
* Discretizing wife's age feature
* Making ordinal categorical Features using Numeric-Integer-Encoding
* Splitting data-set into the set of descriptive features and the target
* Making nominal categorical features using One-Hot-Encoding
* Encoding target
* Scaling descriptive features
* Feature selection & ranking using Random Forest Importance (RFI)
* Splitting data-set into training and test sets
* Selecting evaluation strategy
```

Three different Machine-Learning (ML) classifier algorithms have been explored and built to predict the target feature. Those three ML algorithms are,

```
* K-Nearest Neighbors (KNN)
* (Gaussian) Naive Bayes (NB)
* Decision Trees (DT)
```

All the observation in the data-set (i.e. full data-set) have been used in for modeling keeping 70:30 ratio of observations for training and test perspectives. This comes down to 1031 observations for training set, while 442 observations for test set.

A pipeline technique is used in modeling: firstly, feature selection has been performed using Random Forest Importance method considering 3, 4, 5, ... 9 features. Secondly, the hyper-parameter tuning has been performed for each classifier and subsequently, 'accuracy' method is used to evaluate the performance.

After selecting the best model, 5-fold cross-validation was performed on test data and performed a paired t-test to determine whether there is any statistically significant difference in each model pair. Additionally, the

3. Data Preparation

Loadign Data-set

The data-set that had been preprocessed under phase I has been loaded

In [1]:

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

import numpy as np
import pandas as pd

#reading the data-set
df_con=pd.read_csv('Preprocessed_Data.csv')

#examining the dta-set
from IPython.display import Markdown, display
def printmd(string):
    display(Markdown(string))
print("Dimension of the data set is ({},{}).\n".format(df_con.shape[0],df_con.shape[1])
)

print("Data Types are: \n")
print(df_con.dtypes)
printmd("Random rows in the Data-set")
df_con.sample(6)
```

```
Dimension of the data set is (1473,10).

Data Types are:

wife_age           int64
wife_edu          object
husb_edu          object
children           int64
wife_religion     object
wife-working      object
husb-occup        object
s-living_index    object
media_exp         object
contrac_mthd      object
dtype: object
```

Random rows in the Data-set

Out[1]:

	wife_age	wife_edu	husb_edu	children	wife_religion	wife-working	husb-occup	s-living_index	m
85	33	high	high	2	Islam	No	Cat1	high	
804	36	middle low	middle low	5	Islam	No	Cat3	high	
151	21	low	middle high	4	Islam	No	Cat3	middle low	
22	46	high	high	1	Other	No	Cat1	high	
139	20	high	high	1	Islam	No	Cat1	high	
58	21	middle low	middle high	0	Islam	No	Cat3	high	

Checking for missing values

The missing values can have a significant impact to the data model. Hence it is always a good practice to check and ensure that there are no missing values before building the model. As can be seen below, there are no missing values in the processed data-set.

In [2]:

```
df_con.isna().sum()
```

Out[2]:

```
wife_age          0
wife_edu          0
husb_edu          0
children          0
wife_religion     0
wife-working      0
husb-occup        0
s-living_index    0
media_exp         0
contrac_mthd      0
dtype: int64
```

Discretizing Numeric Features

Since this is a classification task, it is being assumed that transforming the numerical Age feature to ordinal variable will lead to a better performance in the model. As shown below, the Age feature has been transformed to ordinal variable with three levels:

In [3]:

```
contr_df = df_con.copy()
contr_df['wife_age'] = pd.qcut(contr_df['wife_age'], q=3,
                                    labels=['young', 'middle-aged', 'old'])
contr_df['wife_age'].value_counts()
```

Out[3]:

```
young          547
old            469
middle-aged    457
Name: wife_age, dtype: int64
```

Making Ordinal Categorical Features Numeric (using Integer-Encoding)

All descriptive attributes in the data-set (including target and descriptive features) need to be converted to numeric features, in order to use the data-set in Scikit-Learn functions. Integer-Encoding has been done for the ordinal categorical features as below. It is worth mentioning that "wife_edu", "husd_edu", "husb-occup", "s-livinh_index", and "media_exp" in this data-set do have natural ordering, hence consider as ordinal variables.

In [4]:

```
cleanup_nums = {"wife_age": {"young": 1, "middle-aged": 2, "old": 3},
                "wife_edu": {"low": 1, "middle low": 2, "middle high": 3, "high": 4},
                "husb_edu": {"low": 1, "middle low": 2, "middle high": 3, "high": 4},
                "husb-occup": {"Cat1": 1, "Cat2": 2, "Cat3": 3, "Cat4": 4},
                "s-living_index": {"low": 1, "middle low": 2, "middle high": 3, "high":
4},
                "media_exp": {"good": 0, "bad": 1}}
contr_df.replace(cleanup_nums, inplace=True)

#checking the data-set
print("Data Types are: \n")
print(contr_df.dtypes)
print("Random rows in the Data-set")
contr_df.sample(6)
```

```
Data Types are:

wife_age           int64
wife_edu           int64
husb_edu           int64
children           int64
wife_religion     object
wife-working      object
husb-occup         int64
s-living_index     int64
media_exp          int64
contrac_mthd      object
dtype: object
Random rows in the Data-set
```

Out[4]:

	wife_age	wife_edu	husb_edu	children	wife_religion	wife-working	husb-occup	s-living_index	n
473	2	4	4	4	Other	Yes	1	4	
416	3	4	3	5	Islam	No	1	4	
1330	3	4	4	5	Other	No	1	4	
458	3	4	4	4	Islam	No	1	4	
1184	2	3	4	2	Islam	No	3	4	
301	2	4	2	0	Islam	Yes	2	2	

Splitting data-set into the set of descriptive features and the target

Before encoding the target feature, it is required to split the data-set into descriptive features and the target feature. Then the target feature has been examined to see the distribution of each label. As shown below, the target features of No-use, Short-term and Long-term have 629, 511 and 333 instances, respectively.

In [5]:

```
contr_df_cat = contr_df.copy()
#Splitting data-set into the set of descriptive features and the target
Data = contr_df_cat.drop(columns = 'contrac_mthd')
target = contr_df_cat['contrac_mthd']

# Checking the count of instances in each label
target.value_counts()
```

Out[5]:

```
No-use        629
Short-term    511
Long-term     333
Name: contrac_mthd, dtype: int64
```

Making Nominal Categorical Features Numeric (using One-Hot-Encoding)

In the data-set, "wife_religion", and "wife-working" attributes are considered as nominal variables because they possess no natural ordering. The One-Hot-Encoding method has been used in for transforming these nominal variables to numerical.

In [6]:

```
categorical_cols = Data.columns[Data.dtypes==object].tolist()
categorical_cols
```

Out[6]:

```
['wife_religion', 'wife-working']
```

For two-level categorical features, drop_first option has been set to 'True' to encode the variable into a single column of 0 or 1. If there are ordinal categorical features with more than two levels, *Data = pd.get_dummies(Data)* function can be used.

In [7]:

```
for col in categorical_cols:
    n = len(Data[col].unique())
    if (n == 2):
        Data[col] = pd.get_dummies(Data[col], drop_first=True)

# use one-hot-encoding for categorical features with >2 levels
#Data = pd.get_dummies(Data)
```

In [8]:

```
Data.sample(5)
```

Out[8]:

	wife_age	wife_edu	husb_edu	children	wife_religion	wife-working	husb-occup	s-living_index	n
608	3	4	4	5	1	0	2	4	
863	1	4	3	1	0	0	3	4	
1111	1	4	4	1	0	1	1	3	
1373	2	4	4	2	0	1	1	4	
691	2	2	4	3	0	0	3	3	

Encoding Target

The target feature has been encoded to numeric values as shown below:

In [9]:

```
#Encoding Target
from sklearn import preprocessing
target = preprocessing.LabelEncoder().fit_transform(target)

#Checking the encoding
print(type(target))
np.unique(target, return_counts = True)
```

```
<class 'numpy.ndarray'>
```

Out[9]:

```
(array([0, 1, 2]), array([333, 629, 511], dtype=int64))
```

The 'Long-term' is labeled as 0, 'No-use' is labeled as 1 and 'Short-term' is labeled as 2 due to the fact that the LabelEncoder() function labels them in alphabetical order.

Scaling Descriptive Features

Feature scaling (Standardization) is performed to get the highly varying numeric data to a common scale. Even though the scaling is essential only for some of the models (such as KNN, Deep learning and SVMs), scaling is highly recommended for any model. Therefore, Min-Max Scaling has been used in this task, which can convert numerical data to common scale between 0 and 1.

In [10]:

```
from sklearn import preprocessing

#Data_df = Data.copy()

Data_scaler = preprocessing.MinMaxScaler()
Data_scaler.fit(Data)
Data = Data_scaler.fit_transform(Data)
```

Feature Selection & Ranking

In this section, best 6 features of the data-set has been examined using Random Forest Importance (RFI) to get an quick understanding on the importance of each feature. However, during the hyperparameter tuning phase inside the pipeline, the feature selection is performed in more systematic and detailed way using RFI, which eventually determines the optimal number of features for each classifier.

Selecting the best 6 features in the dataset using RFI (Random Forest Importance)

In [11]:

```
from sklearn.ensemble import RandomForestClassifier
num_features = 6
model_rfi = RandomForestClassifier(n_estimators=100)
model_rfi.fit(Data, target)
fs_indices_rfi = np.argsort(model_rfi.feature_importances_)[::-1][0:num_features]

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

Out[11]:

```
array(['children', 's-living_index', 'wife_edu', 'husb-occup', 'wife_age',
       'husb_edu'], dtype=object)
```

In [12]:

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

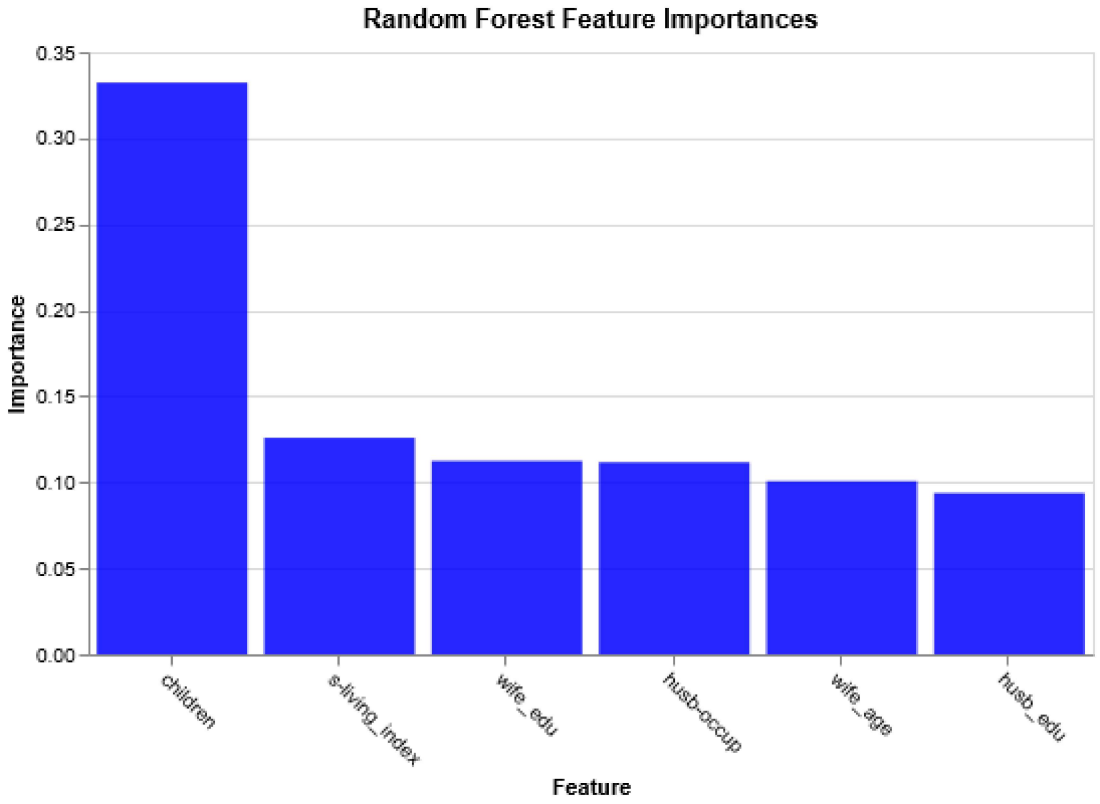
Out[12]:

```
array([0.33239353, 0.12593069, 0.11249967, 0.11167203, 0.1006196 ,
       0.09379219])
```

In [13]:

```
import altair as alt
alt.renderers.enable('notebook')

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=0.85,
                                color=color).encode(
        alt.X('features', title='Feature', sort=None, axis=alt.AxisConfig(labelAngle=45
)),
        alt.Y('importances', title='Importance')
    )

    return chart

#plotting
plot_imp(best_features_rfi, feature_importances_rfi, 'Random Forest', 'blue')
```

Out[13]:



As per the results above, best 6 features of the data-set are 'children', 's-living_index', 'wife_edu', 'husb-occup', 'wife_age', and 'husb_edu'.

Splitting Data into Training and Test Sets

Data sampling is not required as the original data-set is not a significantly larger one: data-set only has 1473 observations. As shown below, the model has been trained and tuned on 1031 rows of training data and tested on 442 rows of test data. This is constituted from 70:30 ratio of training Vs test observations in the data-set.

In [14]:

```
from sklearn.model_selection import train_test_split

D_train, D_test, t_train, t_test = \
    train_test_split(Data, target, test_size = 0.3,
                     stratify=target, shuffle=True, random_state=999)

print(D_train.shape)
print(D_test.shape)
```

```
(1031, 9)
(442, 9)
```

Model Evaluation Strategy

Stratified 5-fold cross-validation with 3 repetitions has been used as the model evaluation strategy.

In [15]:

```
from sklearn.model_selection import RepeatedStratifiedKFold, GridSearchCV

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

4. Hyperparameter Tuning

Since a pipeline can be used to automate the work-flows of the machine learning, it has been used in for feature selection and hyperparameter tuning. The grid search for hyperparameter tuning of each classifiers has been performed via cross-validation approach. In addition to the hyperparameter tuning, feature selection using RFI has also been stacked to the pipeline along with the checks for best number of feature to be selected.

KNN, NB & DT classifiers have been chosen for this classification task. As shown below, each classifier has been optimized using the training data for optimal parameters.

K-Nearest Neighbors (KNN)

A dictionary for the hyperparameters of the KNN classifier is defined as below:

- value range for "number of neighbors (n_neighbors)": 1, 10, 20, 40, 60, 100
- values for p: 1 (Manhattan), 2 (Euclidean)

In [16]:

```
from sklearn.base import BaseEstimator, TransformerMixin

# custom function for RFI feature selection inside a pipeline
class RFIFeatureSelector(BaseEstimator, TransformerMixin):

    # class constructor
    def __init__(self, n_features_=6):
        self.n_features_ = n_features_
        self.fs_indices_ = None

    # override the fit function
    def fit(self, X, y):
        from sklearn.ensemble import RandomForestClassifier
        from numpy import argsort
        model_rfi = RandomForestClassifier(n_estimators=100)
        model_rfi.fit(X, y)
        self.fs_indices_ = argsort(model_rfi.feature_importances_)[::-1][0:self.n_featu
res_]
        return self

    # override the transform function
    def transform(self, X, y=None):
        return X[:, self.fs_indices_]
```

In [17]:

```
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
#from sklearn.metrics import roc_auc_score

pipe_KNN = Pipeline(steps=[('rfi_fs', RFIFeatureSelector()),
                           ('knn', KNeighborsClassifier())])

params_pipe_KNN = {'rfi_fs__n_features_': [3,4,5,6,7,8,9, Data.shape[1]],
                   'knn__n_neighbors': [1, 10, 20, 40, 60, 100],
                   'knn__p': [1, 2]}

gs_pipe_KNN = GridSearchCV(estimator=pipe_KNN,
                           param_grid=params_pipe_KNN,
                           cv=cv_method,
                           refit=True,
                           n_jobs=-2,
                           scoring='accuracy',
                           verbose=1)
```

In [18]:

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

```
Fitting 15 folds for each of 96 candidates, totalling 1440 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent worker
s.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:    7.7s
[Parallel(n_jobs=-2)]: Done 194 tasks      | elapsed:   18.6s
[Parallel(n_jobs=-2)]: Done 444 tasks      | elapsed:   36.8s
[Parallel(n_jobs=-2)]: Done 794 tasks      | elapsed:  1.0min
[Parallel(n_jobs=-2)]: Done 1244 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-2)]: Done 1440 out of 1440 | elapsed:  1.9min finished
C:\Users\udesh\Software\Anaconda2\lib\site-packages\sklearn\model_selectio
n\_search.py:841: DeprecationWarning: The default of the `iid` parameter w
ill change from True to False in version 0.22 and will be removed in 0.24.
This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

In [19]:

```
gs_pipe_KNN.best_params_
```

Out[19]:

```
{'knn__n_neighbors': 20, 'knn__p': 2, 'rfi_fs__n_features_': 5}
```

In [20]:

```
gs_pipe_KNN.best_score_
```

Out[20]:

```
0.5354025218234724
```

As shown above, the KNN model has a mean accuracy score of 0.535. The best parameters for the KNN model are 5 features with 20 nearest neighbors and $p = 2$.

Visualising the KNN model accuracy based on other parameter combinations

In [21]:

```
# custom function to format the search results as a Pandas data frame

def get_search_results(gs):

    def model_result(scores, params):
        scores = {'mean_score': np.mean(scores),
             'std_score': np.std(scores),
             'min_score': np.min(scores),
             'max_score': np.max(scores)}
        return pd.Series({**params,**scores})

    models = []
    scores = []

    for i in range(gs.n_splits_):
        key = f"split{i}_test_score"
        r = gs.cv_results_[key]
        scores.append(r.reshape(-1,1))

    all_scores = np.hstack(scores)
    for p, s in zip(gs.cv_results_['params'], all_scores):
        models.append((model_result(s, p)))

    pipe_results = pd.concat(models, axis=1).T.sort_values(['mean_score'], ascending=Fa
lse)

    columns_first = ['mean_score', 'std_score', 'max_score', 'min_score']
    columns = columns_first + [c for c in pipe_results.columns if c not in columns_firs
t]

    return pipe_results[columns]
```

In [22]:

```
results_KNN = get_search_results(gs_pipe_KNN)
results_KNN.head()
```

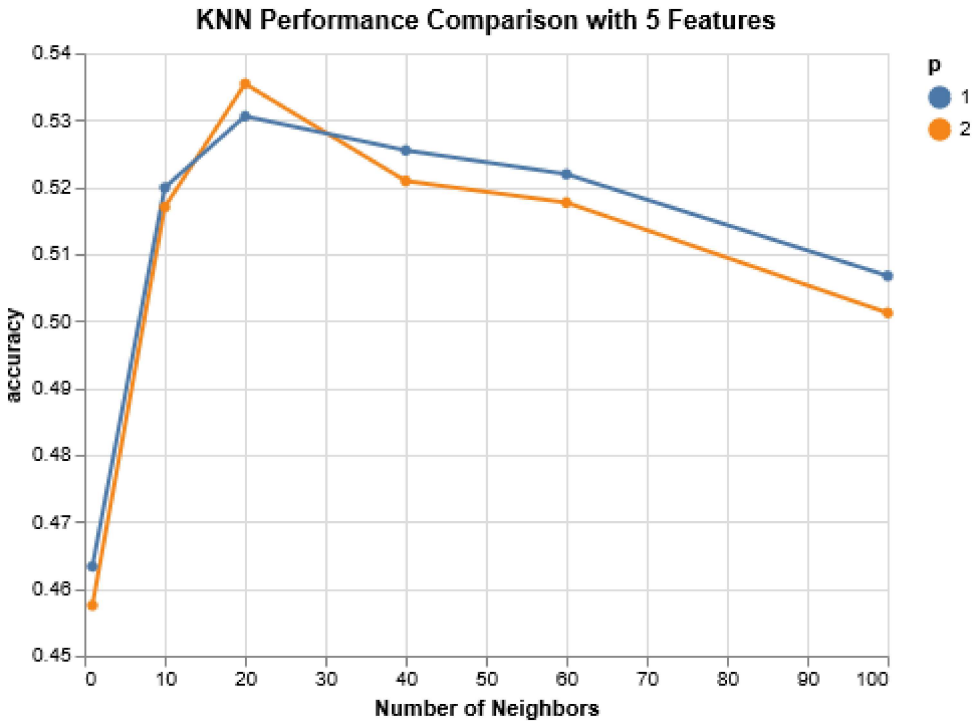
Out[22]:

	mean_score	std_score	max_score	min_score	knn__n_neighbors	knn__p	rfi_fs__n_fea
42	0.535442	0.030889	0.595122	0.487923	20.0	2.0	
35	0.534826	0.030386	0.619512	0.492754	20.0	1.0	
67	0.531927	0.030572	0.590244	0.473430	60.0	1.0	
51	0.531637	0.037078	0.590244	0.439614	40.0	1.0	
34	0.530589	0.026493	0.570048	0.487923	20.0	1.0	

In [23]:

```
results_KNN_5_features = results_KNN[results_KNN['rfi_fs__n_features_'] == 5.0]

alt.Chart(results_KNN_5_features,
          title='KNN Performance Comparison with 5 Features'
         ).mark_line(point=True).encode(
    alt.X('knn__n_neighbors', title='Number of Neighbors'),
    alt.Y('mean_score', title='accuracy', scale=alt.Scale(zero=False)),
    alt.Color('knn__p:N', title='p')
)
```

Out[23]:



(Gaussian) Naive Bayes (NB)

A dictionary for the hyperparameters of the NB classifier is defined as below:

- var_smoothing: start with 10 and end with 10^{-2} with 10 different values for random search over only 20 different values

In [24]:

```
from sklearn.preprocessing import PowerTransformer
D_train_transformed = PowerTransformer().fit_transform(D_train)
```

In [40]:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import RandomizedSearchCV

pipe_NB = Pipeline([('rfi_fs', RFIFeatureSelector()),
                    ('nb', GaussianNB())])

params_pipe_NB = {'rfi_fs__n_features_': [3,4,5,6,7,8,9, Data.shape[1]],
                  'nb__var_smoothing': np.logspace(0,-2, num=10)}

n_iter_search = 20
gs_pipe_NB = RandomizedSearchCV(estimator=pipe_NB,
                           param_distributions=params_pipe_NB,
                           cv=cv_method,
                           refit=True,
                           n_jobs=-2,
                           scoring='accuracy',
                           n_iter=n_iter_search,
                           verbose=1)

gs_pipe_NB.fit(D_train_transformed, t_train);
```

```
Fitting 15 folds for each of 20 candidates, totalling 300 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent worker
s.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:    2.8s
[Parallel(n_jobs=-2)]: Done 194 tasks      | elapsed:   11.5s
[Parallel(n_jobs=-2)]: Done 300 out of 300 | elapsed:   17.9s finished
```

In [41]:

```
gs_pipe_NB.best_params_
```

Out[41]:

```
{'rfi_fs__n_features_': 5, 'nb__var_smoothing': 0.0774263682681127}
```

In [42]:

```
gs_pipe_NB.best_score_
```

Out[42]:

```
0.5337859683155513
```

As shown above, NB model has a mean accuracy score of 0.534. The best parameters for the NB model are 5 features with 0.077 var_smoothing.

Visualising the NB model accuracy based on other parameter combinations

In [43]:

```
results_NB = get_search_results(gs_pipe_NB)
results_NB.head()
```

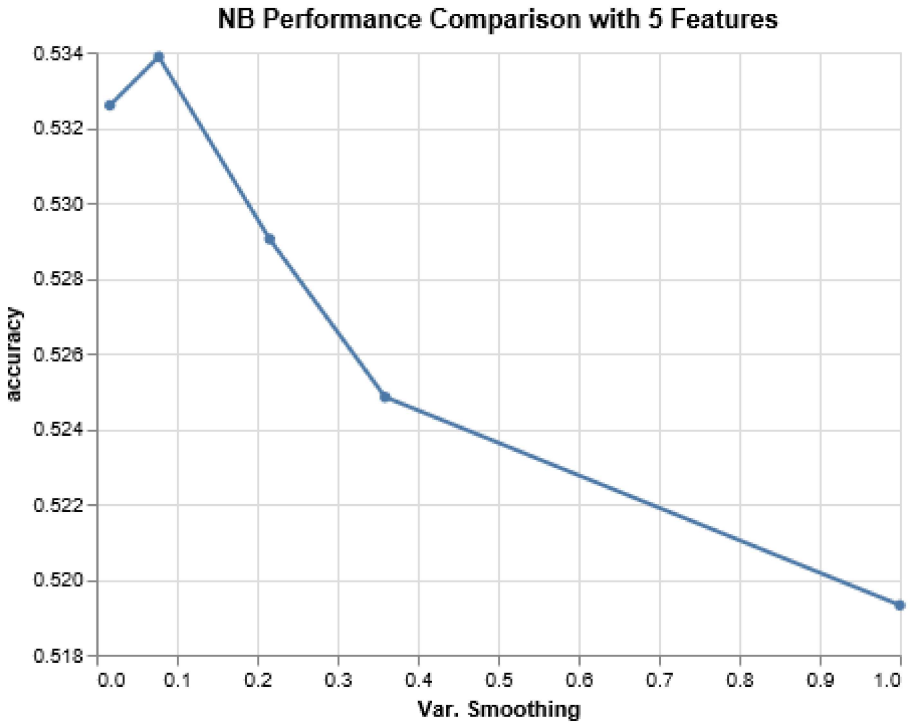
Out[43]:

	mean_score	std_score	max_score	min_score	rfi_fs__n_features_	nb__var_smoothing
5	0.533882	0.031144	0.595122	0.473430	5.0	0.077426
8	0.532590	0.029704	0.585366	0.473430	5.0	0.016681
18	0.529038	0.029615	0.590244	0.483092	5.0	0.215443
17	0.527428	0.036078	0.604878	0.473430	9.0	0.129155
19	0.526136	0.036811	0.604878	0.473430	9.0	0.027826

In [44]:

```
results_NB_5_features = results_NB[results_NB['rfi_fs__n_features_'] == 5.0]

alt.Chart(results_NB_5_features,
          title='NB Performance Comparison with 5 Features'
         ).mark_line(point=True).encode(
    alt.X('nb__var_smoothing', title='Var. Smoothing'),
    alt.Y('mean_score', title='accuracy', scale=alt.Scale(zero=False))
)
```

Out[44]:



Decision Trees (DT)

A dictionary for the hyperparameters of the DT classifier is defined as below:

- Value range for maximum depth (max_depth): 3, 4, 5, 6, 7
- Value for minimum sample split (min_samples_split): 2, 3, 4, 5, 6 , 7

In [49]:

```
from sklearn.tree import DecisionTreeClassifier

pipe_DT = Pipeline([('rfi_fs', RFIFeatureSelector()),
                    ('dt', DecisionTreeClassifier(criterion='gini'))])

params_pipe_DT = {'rfi_fs__n_features_': [3,4,5,6,7,8,9, Data.shape[1]],
                  'dt__max_depth': [3, 4, 5, 6, 7],
                  'dt__min_samples_split': [2, 3, 4, 5, 6, 7]}

gs_pipe_DT = GridSearchCV(estimator=pipe_DT,
                          param_grid=params_pipe_DT,
                          cv=cv_method,
                          refit=True,
                          n_jobs=-2,
                          scoring='accuracy',
                          verbose=1)

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

```
Fitting 15 folds for each of 240 candidates, totalling 3600 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent worker
s.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:    2.6s
[Parallel(n_jobs=-2)]: Done 194 tasks      | elapsed:   10.9s
[Parallel(n_jobs=-2)]: Done 444 tasks      | elapsed:   24.8s
[Parallel(n_jobs=-2)]: Done 794 tasks      | elapsed:   46.9s
[Parallel(n_jobs=-2)]: Done 1244 tasks      | elapsed:  1.2min
[Parallel(n_jobs=-2)]: Done 1794 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-2)]: Done 2444 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-2)]: Done 3194 tasks      | elapsed:  3.1min
[Parallel(n_jobs=-2)]: Done 3600 out of 3600 | elapsed:  3.4min finished
C:\Users\udesh\Software\Anaconda2\lib\site-packages\sklearn\model_selectio
n\_search.py:841: DeprecationWarning: The default of the `iid` parameter w
ill change from True to False in version 0.22 and will be removed in 0.24.
This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

In [50]:

```
gs_pipe_DT.best_params_
```

Out[50]:

```
{'dt__max_depth': 5, 'dt__min_samples_split': 3, 'rfi_fs__n_features_': 6}
```

In [51]:

```
gs_pipe_DT.best_score_
```

Out[51]:

```
0.5638538635628839
```

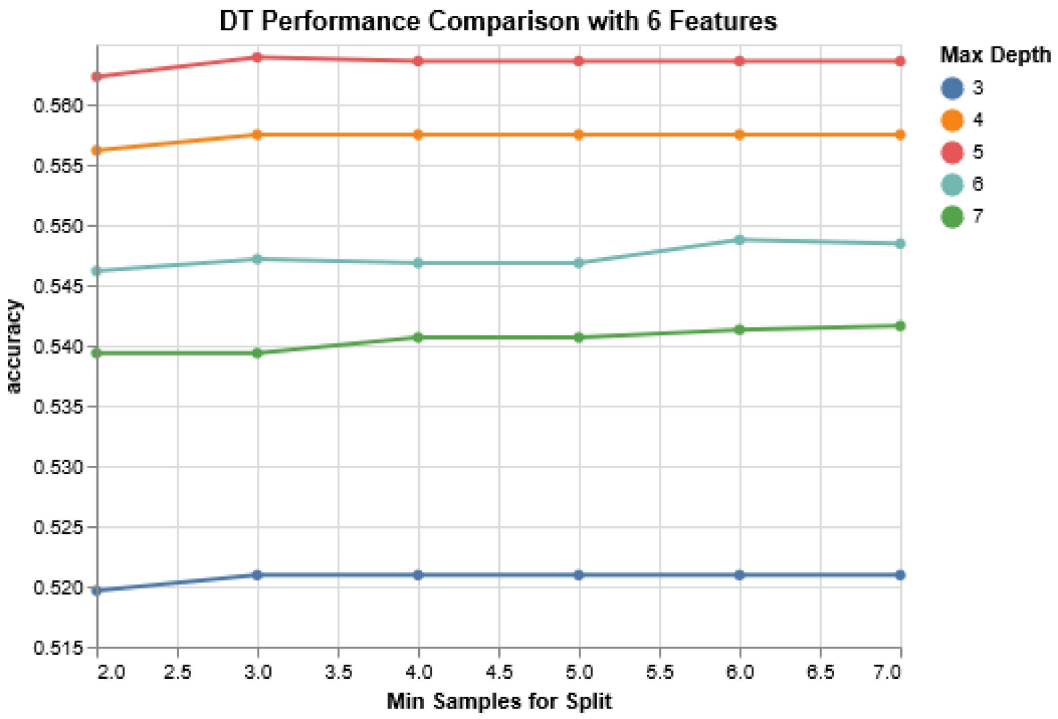
As shown above , DT model has a mean accuracy score of 0.564. The best parameters for the DT model are 6 features with maximum depth of 5 and minimum split value of 3.

Visualising the DT model accuracy based on other parameter combinations

In [52]:

```
results_DT = get_search_results(gs_pipe_DT)

results_DT_6_features = results_DT[results_DT['rfi_fs__n_features_'] == 6.0]

alt.Chart(results_DT_6_features,
          title='DT Performance Comparison with 6 Features'
         ).mark_line(point=True).encode(
    alt.X('dt__min_samples_split', title='Min Samples for Split'),
    alt.Y('mean_score', title='accuracy', scale=alt.Scale(zero=False)),
    alt.Color('dt__max_depth:N', title='Max Depth')
)
```

Out[52]:



Mean accuracy scores for each classifiers

In [53]:

```
print("Mean accuracy score for KNN model: " ,gs_pipe_KNN.best_score_)
print("Mean accuracy score for NB model: " ,gs_pipe_NB.best_score_)
print("Mean accuracy score for DT model: " ,gs_pipe_DT.best_score_)
```

```
Mean accuracy score for KNN model:  0.5354025218234724
Mean accuracy score for NB model:  0.5337859683155513
Mean accuracy score for DT model:  0.5638538635628839
```

Based on above accuracy scores, it can be stated that the best model out of the three explored is DT model.

5. Performance Comparison

After optimizing each of three classifiers using the training data, test data has been fitted in cross-validation manner in order to compare the performance of each model. Pairwise t-test has been conducted to determine whether the performance difference of each pair of models is statistically significant. This is done because cross-validation is a random process.

Paired t-test has been performed for the below model pairs with accuracy score.

- DT vs. KNN
- DT vs. NB
- KNN vs. NB

In [54]:

```
from sklearn.model_selection import cross_val_score

#Stratified 5-fold cross-validation with 3 repetitions
cv_results_KNN = cross_val_score(estimator=gs_pipe_KNN.best_estimator_,
                                 X=D_test,
                                 y=t_test,
                                 cv=cv_method,
                                 n_jobs=-2,
                                 scoring='accuracy')
cv_results_KNN.mean()
```

Out[54]:

```
0.46319344158360043
```

In [55]:

```
D_test_transformed = PowerTransformer().fit_transform(D_test)

cv_results_NB = cross_val_score(estimator=gs_pipe_NB.best_estimator_,
                                X=D_test_transformed,
                                y=t_test,
                                cv=cv_method,
                                n_jobs=-2,
                                scoring='accuracy')
cv_results_NB.mean()
```

Out[55]:

```
0.47942325228260907
```

In [56]:

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

Out[56]:

```
0.526338060966112
```

The `stats.ttest_rel` function from the `SciPy` module has been used to run the t-tests on **test data**.

In [57]:

```
from scipy import stats

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

```
Ttest_relResult(statistic=3.5398348605584475, pvalue=0.003266205917545575
4)
Ttest_relResult(statistic=2.878710634761897, pvalue=0.012142085409240956)
Ttest_relResult(statistic=-0.9833375363617465, pvalue=0.3421309032814237)
```

As per the results above, DT is statistically the best classifier model at 95% significance level. This conclusion has been taken due the fact that p-value is smaller than 0.05, which in other words says there is a statistically significant difference between model pairs.

Model evaluation using other methods

Precision, Recall, F1 Score & Confusion Matrix can also used to evaluate the model as below:

In [58]:

```
pred_KNN = gs_pipe_KNN.predict(D_test)
```

In [59]:

```
pred_NB = gs_pipe_NB.predict(D_test_transformed)
```

In [60]:

```
pred_DT = gs_pipe_DT.predict(D_test)
```

In [61]:

```
from sklearn import metrics
print("\nClassification report for K-Nearest Neighbor")
print(metrics.classification_report(t_test, pred_KNN))
print("\nClassification report for Naive Bayes")
print(metrics.classification_report(t_test, pred_NB))
print("\nClassification report for Decision Tree")
print(metrics.classification_report(t_test, pred_DT))
```

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

           0       0.35      0.36      0.36       100
           1       0.56      0.50      0.53       189
           2       0.40      0.45      0.42       153

   micro avg       0.45      0.45      0.45       442
   macro avg       0.44      0.44      0.44       442
weighted avg       0.46      0.45      0.45       442


Classification report for Naive Bayes
              precision    recall  f1-score   support

           0       0.39      0.50      0.44       100
           1       0.60      0.59      0.59       189
           2       0.46      0.37      0.41       153

   micro avg       0.50      0.50      0.50       442
   macro avg       0.48      0.49      0.48       442
weighted avg       0.50      0.50      0.49       442


Classification report for Decision Tree
              precision    recall  f1-score   support

           0       0.42      0.47      0.44       100
           1       0.71      0.58      0.64       189
           2       0.45      0.51      0.48       153

   micro avg       0.53      0.53      0.53       442
   macro avg       0.52      0.52      0.52       442
weighted avg       0.55      0.53      0.54       442
```

In [62]:

```
print("\nConfusion matrix for K-Nearest Neighbor")
print(metrics.confusion_matrix(t_test, pred_KNN))
print("\nConfusion matrix for Naive Bayes")
print(metrics.confusion_matrix(t_test, pred_NB))
print("\nConfusion matrix for Decision Tree")
print(metrics.confusion_matrix(t_test, pred_DT))
```

```
Confusion matrix for K-Nearest Neighbor
[[36 25 39]
 [31 94 64]
 [35 49 69]]

Confusion matrix for Naive Bayes
[[ 50  20  30]
 [ 39 112  38]
 [ 40  56  57]]

Confusion matrix for Decision Tree
[[ 47  14  39]
 [ 22 110  57]
 [ 43  32  78]]
```

Depending on which organization uses the model, the "precision" or the "recall" may become key performance indicator. For an example, if government organization that runs a program to find number of womens who do not use any contraceptive method, "recall" would be the best performance indicator. According to above classification report, its is evident that DT is the best method with higher recall value.

6. Limitations

Since the size of the data set is significantly small (only 1473), the accuracy rate of the models developed could be considerably low to represent entire county (Indonesia). Also, this data-set could be a biased one, considering the small size of it. As a well know limitation of the supervised machine learning is that it requires large number of data to achieve a reasonably accurate model. Therefore, if a larger data set is available for this exercise, then more accurate model could have been developed. In oppose to the supervised machine learning, deep learning would have been a better approach for this kind of problems with limited data-sets.

There is a room to improve the model in future by considering more parameters and more ensemble methods during the hyperparameter tuning process.

There are few assumptions for the paired sample t-test and it can impact to the evaluation. Main assumptions in paired sample t-test are;

- The data-set is normally distributed and has no outliers
- Variable in the data-set is independent and continuous

7. Summary

Contraceptive method data-set is a multi-class classification problem and K-Nearest Neighbors (KNN), Naive Bayes (NB) & Decision trees (DT) classifiers have been examined for this classification problem.

When evaluating the training set, the DT model with 6 of the best features selected by Random Forest Importance (RFI) produces the highest cross-validated accuracy score. Similarly, when evaluating the test set, the DT model performed the best on accuracy score of approximately 55%. Therefore, it can be concluded that the DT method is the most suitable model for this classification problem based on the given data-set.

References

- Dua, D. and Graff, C (2019). UCI Machine Learning Repository: Contraceptive Method Choice Data Set [online]. Available at https://archive.ics.uci.edu/ml/datasets/Contraceptive+Method+Choice (https://archive.ics.uci.edu/ml/datasets/Contraceptive+Method+Choice) [Accessed 2019-06-02]
- Aksakalli V, Yenice Z (2019). Feature Ranking [online]. Available at https://www.featureranking.com/ (https://www.featureranking.com/)
- Aksakalli V (2019). Machine Learning (1910) Lecture Notes, RMIT University, Melbourne, delivered Semester-1 2019.
- Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/ (http://www.scipy.org/) [Accessed 2019-05-26].
- Pedregosa et al. (2011). Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830.
- Travis E, Oliphant (2006). A guide to NumPy, USA: Trelgol Publishing.