# MATH2319 Machine Learning Project Phase - 2

## Identifying the factors affecting customer churn

Submitted by: Simarpreet Luthra (s3706588) & Pooja Dandge (s3698457)

Submitted to: Dr. Vural Aksakalli

## Table of Contents

## Introduction

The aim of this project is to investigate the reason behind discontinuing customers of a telecommunications company [1]. The dataset was obtained from the sample datasets provided in IBM Watson Analytics Community blog at https://community.watsonanalytics.com/wp-content/uploads/2015/03/WA_Fn-UseC_-Telco-Customer-Churn.csv. The dataset provides the customer demographics, opted services and account preferences of a telecommunications company [2]. The project has two phases. Phase 1 focused on data preprocessing and exploration. Phase 2 focuses on using Machine Learning models on customer attributes to classify customers who are more likely to churn. This would help in developing focused customer retention programs.

The target feature is churn, which is a binary variable that describes whether a customer churned or not. There are nineteen descriptive features and one customer ID feature, which is a unique identifier of each customer. The dataset has 7043 rows, each describing a different customer. Nearest Neighbors, Naive Bayes and Decision Tree classifiers were built to predict the binary target (churn).

The modeling strategy begins with encoding and scaling the features. The data is then split into training and test sample in the ratio 70:30. Before fitting any of the three models mentioned above,ten best features are selected using the powerful Random Forest Importance method inside a pipeline. Using feature selection together with hyperparameter search inside a single pipeline, a 5-fold stratified cross-validation to fine-tune hyperparameters of each classifier using recall as the performance metric as predicting churning customers correctly is more important than non-churning customers. Each model is built using parallel processing with "-2" cores. Since the target has more non-churning customers (unbalanced target class issue), stratification is crucial to ensure that each validation set has the same proportion of classes as in the original dataset. Each model is also tested for its sensitivity to hyperparameter tuning. Once the best model is identified for each of the three classifier types using a hyperparameter search on the training data, a 10-fold cross-validation is conducted on the test data and a paired t-test is performed to see if the difference in recall scores is statistically significant.

In [1]:
```
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np

import io
import requests
import os, ssl
```

The dataset was cleaned and pre-processed in Phase-1. Thus, the pre-processed dataset from Phase-1 is loaded and customerID column is dropped as it is just a unique identifier and not an attribute of customer.

In [2]:
```
# Read the cleaned preprocessed data from phase 1
churn = pd.read_csv("./churn_clean.csv", sep=',', decimal='.', header=0)
churn = churn.drop(columns='customerID')
```

In [3]:
```
churn['PaymentMethod'] = churn['PaymentMethod'].str.replace(' \(automatic\)',
"")
```

In [4]:
```
churn.shape
```

Out[4]:
```
(7043, 20)
```

In [5]:
```
print(f"\nData type for each feature:")
churn.dtypes
```

```
Data type for each feature:
```

Out[5]:
```
gender              object
SeniorCitizen        int64
Partner             object
Dependents          object
tenure               int64
PhoneService        object
MultipleLines       object
InternetService     object
OnlineSecurity      object
OnlineBackup        object
DeviceProtection    object
TechSupport         object
StreamingTV         object
StreamingMovies     object
Contract            object
PaperlessBilling    object
PaymentMethod       object
MonthlyCharges     float64
TotalCharges       float64
Churn               object
dtype: object
```

## Summary Statistics

The summary statistics for the full data are shown below.

In [6]:
```
churn.describe(include='all')
```

Out[6]:

| | gender | SeniorCitizen | Partner | Dependents | ... | Churn |
|---|---|---|---|---|---|---|
| count | 7043 | 7043.000000 | 7043 | 7043 | | 7043 |
| unique | 2 | NaN | 2 | 2 | | 2 |
| top | Male | NaN | No | No | | No |
| freq | 3555 | NaN | 3641 | 4933 | | 5174 |
| mean | NaN | 0.162147 | NaN | NaN | | NaN |
| std | NaN | 0.368612 | NaN | NaN | | NaN |
| min | NaN | 0.000000 | NaN | NaN | | NaN |
| 25% | NaN | 0.000000 | NaN | NaN | | NaN |
| 50% | NaN | 0.000000 | NaN | NaN | | NaN |
| 75% | NaN | 0.000000 | NaN | NaN | | NaN |
| max | NaN | 1.000000 | NaN | NaN | | NaN |

## Encoding the Target Feature

The descriptive features and the target feature are stored into separate variables. The target feature is then encoded into numeric values, 1 for churning customers and 0 for non-churning customers.

In [7]:
```
churn_data = churn.drop(columns='Churn')
target = churn['Churn']
target.value_counts()
```

Out[7]:
```
No     5174
Yes    1869
Name: Churn, dtype: int64
```

In [8]:
```
target = target.replace({'No': 0, 'Yes': 1})
target.value_counts()
```

Out[8]:
```
0    5174
1    1869
Name: Churn, dtype: int64
```

## Encoding Categorical Descriptive Features

The categorical descriptive features are encoded using one-hot ending. If the variable has more than two levels, the variable is broken into dummy variables equal to the number of levels and each variable is encoded into zero and one representing presence and absence respectively. The variables with two levels are directly assigned zero and one values for their respective levels. Thus, the nineteen features are transformed into forty features.

In [9]:
```
categorical_cols = churn_data.columns[churn_data.dtypes==object].tolist()
```

In [10]:
```
categorical_cols
```

Out[10]:
```
['gender',
 'Partner',
 'Dependents',
 'PhoneService',
 'MultipleLines',
 'InternetService',
 'OnlineSecurity',
 'OnlineBackup',
 'DeviceProtection',
 'TechSupport',
 'StreamingTV',
 'StreamingMovies',
 'Contract',
 'PaperlessBilling',
 'PaymentMethod']
```

In [11]:
```
for col in categorical_cols:
    n = len(churn_data[col].unique())
    churn_data[col] = churn_data[col].str.replace(" ", '_')
    if (n == 2):
        churn_data[col] = pd.get_dummies(churn_data[col], drop_first=True)

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

In [12]:
```
churn_data.columns
```

Out[12]:
```
Index(['gender', 'SeniorCitizen', 'Partner', 'Dependents', 'tenure',
       'PhoneService', 'PaperlessBilling', 'MonthlyCharges', 'TotalCharges',
       'MultipleLines_No', 'MultipleLines_No_phone_service',
       'MultipleLines_Yes', 'InternetService_DSL',
       'InternetService_Fiber_optic', 'InternetService_No',
```

```
       'OnlineSecurity_No', 'OnlineSecurity_No_internet_service',
       'OnlineSecurity_Yes', 'OnlineBackup_No',
       'OnlineBackup_No_internet_service', 'OnlineBackup_Yes',
       'DeviceProtection_No', 'DeviceProtection_No_internet_service',
       'DeviceProtection_Yes', 'TechSupport_No',
       'TechSupport_No_internet_service', 'TechSupport_Yes',
'StreamingTV_No',
       'StreamingTV_No_internet_service', 'StreamingTV_Yes',
       'StreamingMovies_No', 'StreamingMovies_No_internet_service',
       'StreamingMovies_Yes', 'Contract_Month-to-month', 'Contract_One_year',
       'Contract_Two_year', 'PaymentMethod_Bank_transfer',
       'PaymentMethod_Credit_card', 'PaymentMethod_Electronic_check',
       'PaymentMethod_Mailed_check'],
      dtype='object')
```

In [13]:
```
churn_data.sample(5, random_state=999)
```

Out[13]:

| | gender | SeniorCitizen | Partner | Dependents | tenure | ... | PhoneService |
|---|---|---|---|---|---|---|---|
| 4210 | 0 | 1 | 1 | 1 | 61 | | 1 |
| 2099 | 0 | 0 | 1 | 0 | 40 | | 1 |
| 5302 | 0 | 0 | 1 | 1 | 10 | | 1 |
| 2791 | 1 | 0 | 0 | 0 | 57 | | 1 |
| 3471 | 0 | 0 | 0 | 0 | 35 | | 1 |

5 rows × 40 columns

## Scaling of Features

After encoding, the variables are scaled to the range of zero to one using Minmax scaler. For each observation (i) in the given column (x), minmax scaler is defined as $(x[i] - min(x))/(max(x) - min(x))$ where each observation is subtracted by minimum value of the column and divided by difference of maximum of column and minimum of column. Thus, the values are scaled between zero and one.

In [14]:
```
from sklearn import preprocessing

churn_data_df = churn_data.copy()

churn_data_scaler = preprocessing.MinMaxScaler()
churn_data_scaler.fit(churn_data)
churn_data = churn_data_scaler.fit_transform(churn_data)
```

In [15]:
```
pd.DataFrame(churn_data, columns=churn_data_df.columns).sample(5,
random_state=999)
```

Out[15]:

| | gender | SeniorCitizen | Partner | Dependents | ... | MultipleLines_No |
|---|---|---|---|---|---|---|
| 4210 | 0.0 | 1.0 | 1.0 | 1.0 | | 0.0 |
| 2099 | 0.0 | 0.0 | 1.0 | 0.0 | | 0.0 |
| 5302 | 0.0 | 0.0 | 1.0 | 1.0 | | 1.0 |
| 2791 | 1.0 | 0.0 | 0.0 | 0.0 | | 1.0 |
| 3471 | 0.0 | 0.0 | 0.0 | 0.0 | | 1.0 |

5 rows × 40 columns

## Feature Selection & Ranking

Random forest creates many weak classifiers and combines them all to make one strong classifier. Thus, since it goes through a lot of iterations, the feature importance, as predicted by Random forest classifier is very pronounced and effective [3]. A quick overview displays the ranking of ten most important features out of the forty features. This classifier is used in the model building pipeline to select ten most important feature based on each model.

In [16]:
```
from sklearn.ensemble import RandomForestClassifier

num_features = 10
model_rfi = RandomForestClassifier(n_estimators=100)
model_rfi.fit(churn_data, target)
fs_indices_rfi = np.argsort(model_rfi.feature_importances_)[::-
1][0:num_features]

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

Out[16]:
```
array(['TotalCharges', 'tenure', 'MonthlyCharges',
       'Contract_Month-to-month', 'OnlineSecurity_No', 'TechSupport_No',
       'gender', 'PaymentMethod_Electronic_check', 'PaperlessBilling',
       'InternetService_Fiber_optic'], dtype=object)
```

In [17]:
```
feature_importances_rfi = model_rfi.feature_importances_[fs_indices_rfi]
feature_importances_rfi
```

Out[17]:
```
array([0.17326062, 0.15240018, 0.15013565, 0.04653161, 0.0393398 ,
       0.03046817, 0.02821525, 0.02717345, 0.02473739, 0.02458039])
```

In [18]:
```
# Visualize Feature Importance
import seaborn as sns
import matplotlib.pyplot as plt
plt.style.use(u'ggplot')

df = pd.DataFrame({'Features': best_features_rfi,
                        'Importances': feature_importances_rfi})

ax = sns.barplot(y="Features",
x="Importances",data=df,color="red",edgecolor="black")
plt.show()
print("Figure-1: Top 10 features using Random Forest Importance")
```

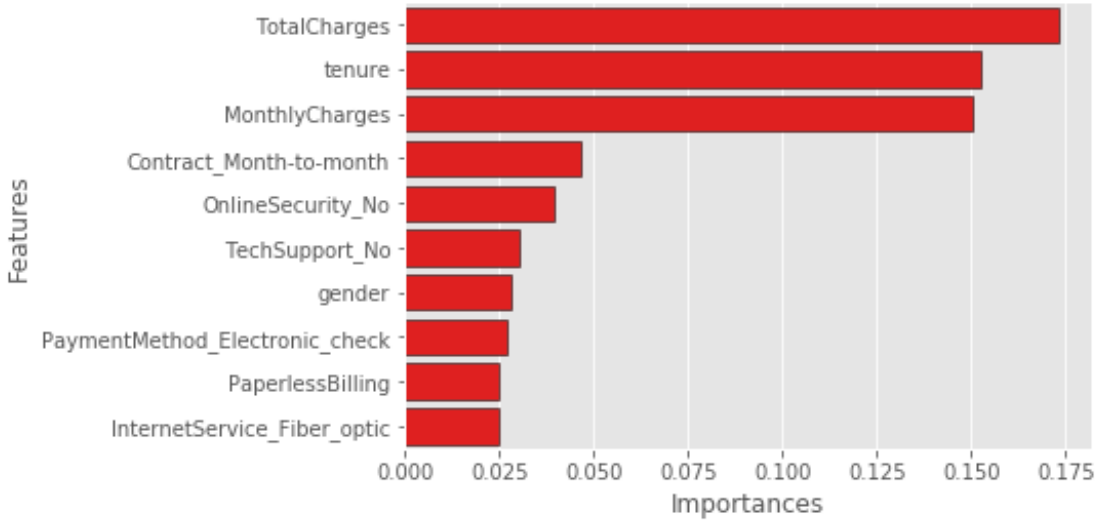


Figure-1: Top 10 features using Random Forest Importance

## Train-Test Splitting and Model Evaluation Strategy

The dataset is split into a 70:30 train:test ratio and target is stratified, target levels are equally proportionate in the train and test sample

```
In [19]:
from sklearn.model_selection import train_test_split

churn_data_train, churn_data_test, \
target_train, target_test = train_test_split(churn_data, target,
                                                    test_size = 0.3,
random_state=999,
                                                    stratify = target)

print(churn_data_train.shape)
print(churn_data_test.shape)
```

```
(4930, 40)
(2113, 40)
```

5-fold stratified cross-validation is used in the model hyperparameter tuning to ensure that the accuracy result is not buffed/debuffed due to a lucky split.

```
In [20]:
from sklearn.model_selection import StratifiedKFold, GridSearchCV

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

RFIFeatureSelector function is written for random forest feature selection in the pipeline to select the set of best features for each model iteration.

In [21]:

```
from sklearn.base import BaseEstimator, TransformerMixin

# custom function for RFI feature selection inside a pipeline
# here we use n_estimators=100
class RFIFeatureSelector(BaseEstimator, TransformerMixin):

    # class constructor
    # make sure class attributes end with a "_"
    # per scikit-learn convention to avoid errors
    def __init__(self, n_features_=10):
        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_features_]
        return self

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

## Hyperparameter Tuning

### K-Nearest Neighbors (KNN)

Using Pipeline, feature selection and grid search are stacked to fine-tune the number the number of neighbors and distance metrics for the model. The results suggest that the accuracy drops as the neighbors are increased to 20 and then sharply increases. Further,using Manhattan distance metric gives higher accuracy than Euclidean distance. The highest recall value for KNN training sample is 53.5%, which is a very slight improvement over random guess.

In [22]:
```
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier

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

params_pipe_KNN = {'rfi_fs__n_features_': [10],
                   '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='recall',
                           verbose=1)
```

In [23]:
```
gs_pipe_KNN.fit(churn_data_train, target_train);
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:   24.2s
[Parallel(n_jobs=-2)]: Done  60 out of  60 | elapsed:   32.3s finished
```

In [24]:
```
gs_pipe_KNN.best_params_
```

Out[24]:
```
{'knn__n_neighbors': 100, 'knn__p': 1, 'rfi_fs__n_features_': 10}
```

In [25]:
```
gs_pipe_KNN.best_score_
```

Out[25]:
```
0.5397354130949018
```

In [26]:
```
# 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=False)

    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_first]

    return pipe_results[columns]
```

In [27]:
```
results_KNN = get_search_results(gs_pipe_KNN)

results_KNN.head()
```

Out[27]:

| | mean_score | std_score | max_score | min_score | knn__n_neighbors | knn__p | rfi_fs__n_features_ |
|---|---|---|---|---|---|---|---|
| 10 | 0.539712 | 0.029435 | 0.591603 | 0.509579 | 100.0 | 1.0 | 10.0 |
| 11 | 0.528253 | 0.022690 | 0.553435 | 0.494253 | 100.0 | 2.0 | 10.0 |
| 8 | 0.519096 | 0.035249 | 0.564885 | 0.465649 | 60.0 | 1.0 | 10.0 |
| 9 | 0.503826 | 0.031823 | 0.553435 | 0.454198 | 60.0 | 2.0 | 10.0 |
| 0 | 0.498424 | 0.042635 | 0.580153 | 0.455939 | 1.0 | 1.0 | 10.0 |

In [28]:
```
ax = sns.pointplot(y="mean_score",
x="knn__n_neighbors",hue="knn__p",data=results_KNN,palette="muted",join=True,
legend=False)
h, l = ax.get_legend_handles_labels()
plt.legend(title="Distance Metric",handles= h,labels =
["Manhattan","Euclidian"],loc="lower right", frameon=True,facecolor="white")
plt.xlabel("Number of neighbors")
plt.ylabel("Average Recall")
plt.show()
print("Figure-2: Hyperparameter tuning for Nearest Neighbors")
```

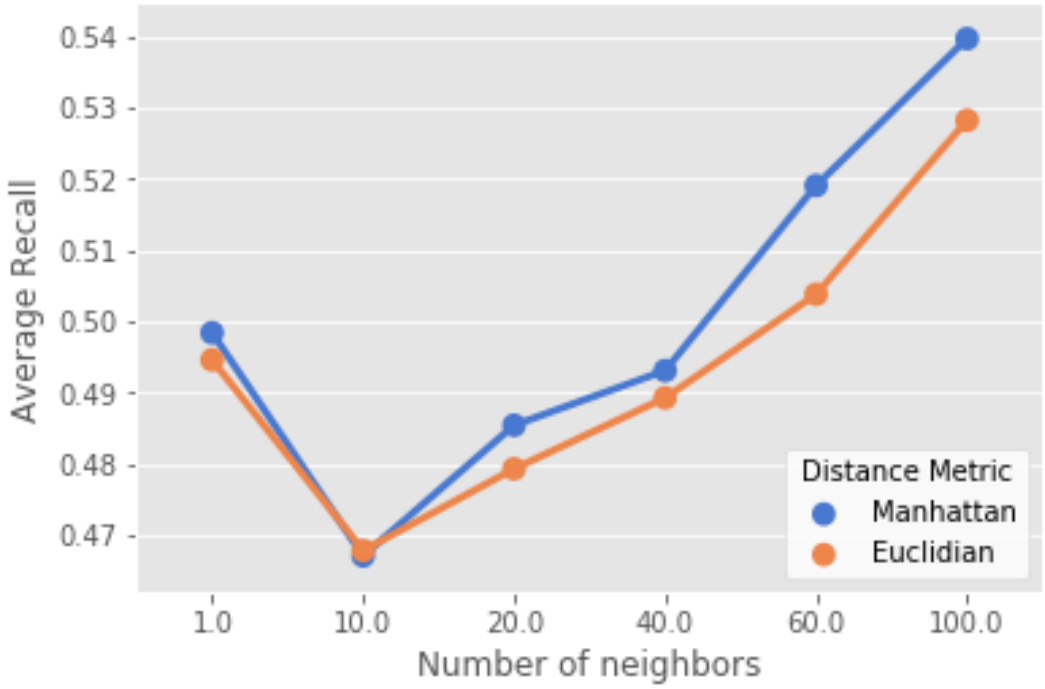


Figure-2: Hyperparameter tuning for Nearest Neighbors

## (Gaussian) Naive Bayes (NB)

A Gaussian Naive Bayes model was implemented by optimizing var_smoothing (a variant of Laplace smoothing) as we do not have any prior information about our dataset. By default, the var_smoothing parameter's value is 10^−9 . A random search is performed over 20 different values to find the optimal variance smoothing. Since NB requires each descriptive feature to follow a Normal distribution, we first perform a power transformation on the input data before model fitting. The accuracy sharply drops as variance smoothing is increased. Thus, value of var__smoothing is assumed to be 0.037 giving a recall value of 74.8%

In [29]:
```
from sklearn.preprocessing import PowerTransformer
churn_data_train_transformed =
PowerTransformer().fit_transform(churn_data_train)
```

In [30]:
```
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_': [10],
                  'nb__var_smoothing': np.logspace(1,-3, num=200)}

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='recall',
                           n_iter=n_iter_search,
                           verbose=1)

gs_pipe_NB.fit(churn_data_train_transformed, target_train);
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:   11.9s
[Parallel(n_jobs=-2)]: Done 100 out of 100 | elapsed:   26.0s finished
```

In [31]:
```
gs_pipe_NB.best_params_
```

Out[31]:
```
{'rfi_fs__n_features_': 10, 'nb__var_smoothing': 0.005291978735958442}
```

In [32]:
```
gs_pipe_NB.best_score_
```

Out[32]:
```
0.7331745427473619
```

In [33]:
```
results_NB = get_search_results(gs_pipe_NB)
results_NB.head()
```

Out[33]:

| | mean_score | std_score | max_score | min_score | rfi_fs__n_features_ | nb__var_smoothing |
|---|---|---|---|---|---|---|
| 11 | 0.733158 | 0.034354 | 0.801527 | 0.709924 | 10.0 | 0.005292 |
| 3 | 0.732403 | 0.027540 | 0.786260 | 0.709924 | 10.0 | 0.003827 |
| 0 | 0.730856 | 0.037834 | 0.805344 | 0.701149 | 10.0 | 0.002300 |
| 14 | 0.730856 | 0.037834 | 0.805344 | 0.701149 | 10.0 | 0.003037 |
| 6 | 0.730104 | 0.028284 | 0.786260 | 0.709924 | 10.0 | 0.002524 |

In [34]:

```
ax = sns.lineplot(y="mean_score",
x="nb__var_smoothing",style="rfi_fs__n_features_",data=results_NB,markers=Tru
e,legend=False)
plt.xlabel("Variance Smoothing")
plt.ylabel("Average Recall")
plt.show()
print("Figure-3: Hyperparameter tuning for Naive Bayes")
```

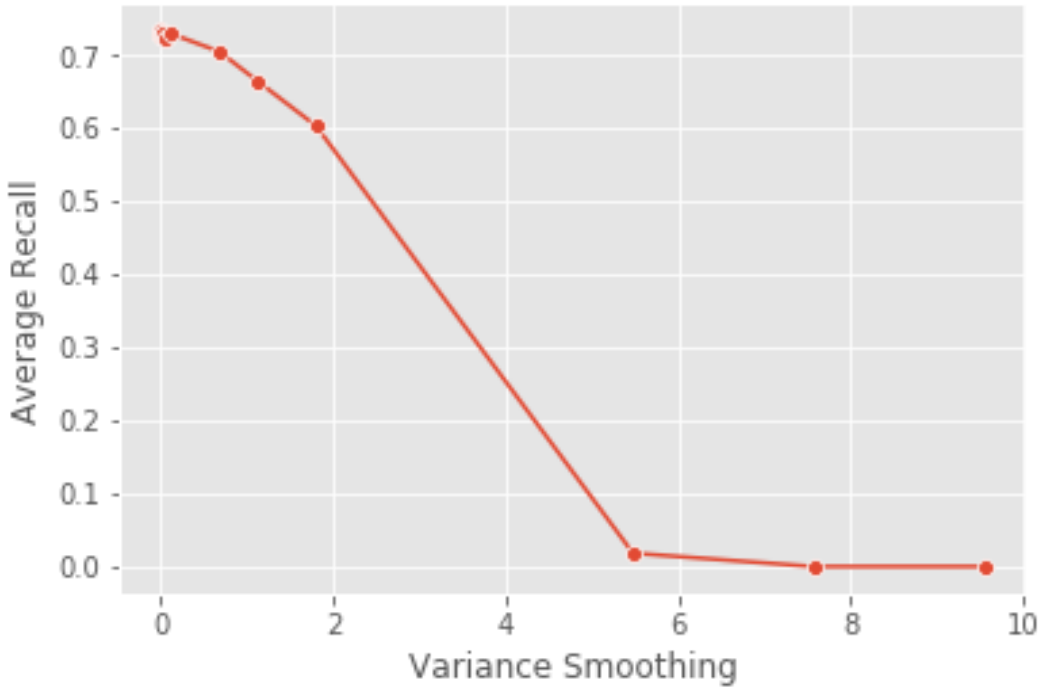


Figure-3: Hyperparameter tuning for Naive Bayes

## Decision Trees (DT)

Maximum depth and Minimum sample split are tuned for decision trees and it can be clearly observed that maximum depth equal to five outperforms all the smaller and larger models and has the best accuracy at min sample split equal to two. The best recall value from decision trees is 57.2%

In [35]:
```
from sklearn.tree import DecisionTreeClassifier

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

params_pipe_DT = {'rfi_fs__n_features_': [10],
                  'dt__max_depth': [3, 4, 5, 6, 10],
                  'dt__min_samples_split': [2, 5, 10, 50, 100]}

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

gs_pipe_DT.fit(churn_data_train, target_train);
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done  44 tasks      | elapsed:   12.7s
[Parallel(n_jobs=-2)]: Done 125 out of 125 | elapsed:   37.0s finished
```

In [36]:
```
gs_pipe_DT.best_params_
```

Out[36]:
```
{'dt__max_depth': 5, 'dt__min_samples_split': 2, 'rfi_fs__n_features_': 10}
```

In [37]:
```
gs_pipe_DT.best_score_
```

Out[37]:
```
0.5550521580741713
```

In [38]:
```
results_DT = get_search_results(gs_pipe_DT)
```

In [39]:
```
results_DT.head()
```

Out[39]:

| | mean_score | std_score | max_score | min_score | dt__ max_depth | dt__min_ sample_split | rfi_fs__ n_features_ |
|---|---|---|---|---|---|---|---|
| 10 | 0.555044 | 0.035119 | 0.59542 | 0.51341 | 5.0 | 2.0 | 10.0 |
| 12 | 0.554277 | 0.034296 | 0.59542 | 0.51341 | 5.0 | 10.0 | 10.0 |
| 14 | 0.554277 | 0.034296 | 0.59542 | 0.51341 | 5.0 | 100.0 | 10.0 |
| 13 | 0.554277 | 0.034296 | 0.59542 | 0.51341 | 5.0 | 50.0 | 10.0 |
| 11 | 0.554277 | 0.034296 | 0.59542 | 0.51341 | 5.0 | 5.0 | 10.0 |

In [40]:

```
ax = sns.pointplot(y="mean_score",
x="dt__min_samples_split",hue="dt__max_depth",data=results_DT,palette="muted"
,join=True,legend=True)
h, l = ax.get_legend_handles_labels()
plt.legend(title="Max depth",loc="lower right",
frameon=True,facecolor="white")
plt.xlabel("Min Sample Split")
plt.ylabel("Average Recall")
plt.show()
print("Figure-4: Hyperparameter tuning for Decision Tree")
```

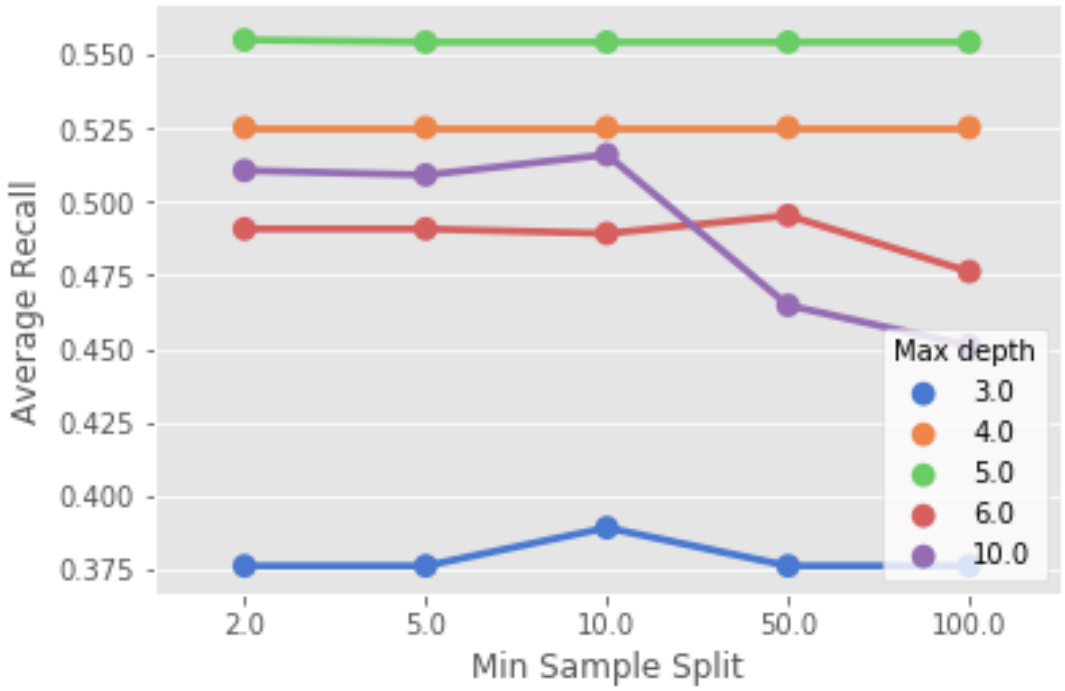


Figure-4: Hyperparameter tuning for Decision Tree

## Performance Comparison

10-fold cross-validation is performed while predicting the test sample and the results are consistent with the training results that is Naive Bayes outperforms the other two algorithms by a factor of about 20%. The paired t-test values for Naive Bayes with other two algorithms is far less than 0.05, Thus, the accuracy difference is significant. The accuracy difference for KNN and DT is insignificant as the paired t-test value is greater than 0.05. This is further supported by the fact thatboth the algorithms have very close accuracy and KNN accuracy was lower than DT while training and higher while testing.

In [41]:
```
from sklearn.model_selection import cross_val_score

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

cv_results_KNN = cross_val_score(estimator=gs_pipe_KNN.best_estimator_,
                                 X=churn_data_test,
                                 y=target_test,
                                 cv=cv_method_ttest,
                                 n_jobs=-2,
                                 scoring='recall')
cv_results_KNN.mean()
```

Out[41]:
```
0.5561090225563909
```

In [42]:
```
churn_data_test_transformed =
PowerTransformer().fit_transform(churn_data_test)

cv_results_NB = cross_val_score(estimator=gs_pipe_NB.best_estimator_,
                                X=churn_data_test_transformed,
                                y=target_test,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='recall')
cv_results_NB.mean()
```

Out[42]:
```
0.7451754385964913
```

In [43]:
```
cv_results_DT = cross_val_score(estimator=gs_pipe_DT.best_estimator_,
                                X=churn_data_test,
                                y=target_test,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='recall')
cv_results_DT.mean()
```

Out[43]:
```
0.5207080200501253
```

In [44]:
```
from scipy import stats

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

```
Ttest_relResult(statistic=-10.750336633788288, pvalue=1.953292430807435e-06)
Ttest_relResult(statistic=-1.5826164964472782, pvalue=0.1479683822072602)
Ttest_relResult(statistic=-10.715315327250472, pvalue=2.0076135263617794e-06)
```

In [45]:
```
pred_KNN = gs_pipe_KNN.predict(churn_data_test)
```

In [46]:
```
churn_data_test_transformed =
PowerTransformer().fit_transform(churn_data_test)
pred_NB = gs_pipe_NB.predict(churn_data_test_transformed)
```

In [47]:
```
pred_DT = gs_pipe_DT.predict(churn_data_test)
```

In [48]:
```
from sklearn import metrics
print("\nClassification report for K-Nearest Neighbor")
print(metrics.classification_report(target_test, pred_KNN))
print("\nClassification report for Naive Bayes")
print(metrics.classification_report(target_test, pred_NB))
print("\nClassification report for Decision Tree")
print(metrics.classification_report(target_test, pred_DT))
```

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

           0       0.83      0.89      0.86      1552
           1       0.62      0.51      0.56       561

   micro avg       0.79      0.79      0.79      2113
   macro avg       0.73      0.70      0.71      2113
weighted avg       0.78      0.79      0.78      2113


Classification report for Naive Bayes
              precision    recall  f1-score   support

           0       0.89      0.77      0.83      1552
           1       0.54      0.75      0.63       561
```

```
   micro avg       0.77      0.77      0.77      2113
   macro avg       0.72      0.76      0.73      2113
weighted avg       0.80      0.77      0.78      2113


Classification report for Decision Tree
              precision    recall  f1-score   support

           0       0.84      0.86      0.85      1552
           1       0.59      0.56      0.58       561

   micro avg       0.78      0.78      0.78      2113
   macro avg       0.72      0.71      0.71      2113
weighted avg       0.78      0.78      0.78      2113
```

In [49]:

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

```
Confusion matrix for K-Nearest Neighbor
[[1377  175]
 [ 275  286]]

Confusion matrix for Naive Bayes
[[1198  354]
 [ 142  419]]

Confusion matrix for Decision Tree
[[1334  218]
 [ 245  316]]
```

Even though Naive Bayes predictions have lower precision. However, the higher recall value makes the model more favorable in this scenario as prediction of interest is the rate of churning customers and the factors affecting the churn and not the non-churning customers.

## Limitations and Proposed Solutions

The modeling strategy has a flaws and limitations. Even though Random forest is a very good predictor of most important features. However since it averages over many iterations, it is not physically possible to understand the method used to rank the important features. Secondly, to reduce the computation load, the number of experiments for hyperparameter tuning were limited, which could lead to a sub-optimal parameter choice and might be the reason for poor accuracy by Decision Trees and Nearest Neighbors algorithms. Furthermore, the algorithms used are quite basic, so Random Forest and other ensemble models could be considered as potentially better models and used for predictions.

## Conclusion

The Gaussian Naive Bayes model with ten best features selected using Random Forest Importance and variance smoothing of 0.036 proved to be the best model in this scenario to predict churning customers using the recall metric on both the training and test data. It far surpasses the Decision Tree and Nearest Neighbor algorithms for churn prediction. Even though those algorithms have a better precision, the prediction of churning customers is really poor and are hence rejected.

## References

[1] "Churn Rate", Investopedia, 2019. [Online]. Available: https://www.investopedia.com/terms/c/churnrate.asp.

[2] "Using Customer Behavior Data to Improve Customer Retention", IBM Analytics Communities, 2019. [Online]. Available: https://www.ibm.com/communities/analytics/watson-analytics-blog/predictive-insights-in-the-telco-customer-churn-data-set/.

[3] 2019. [Online]. Available: https://www.researchgate.net/post/What_methods_are_suitable_for_feature_selection_to_improve_classification_accuracy.