Analysis and Insights from Customer Churn Project

Objective:

To identify the factors impacting customer churn.

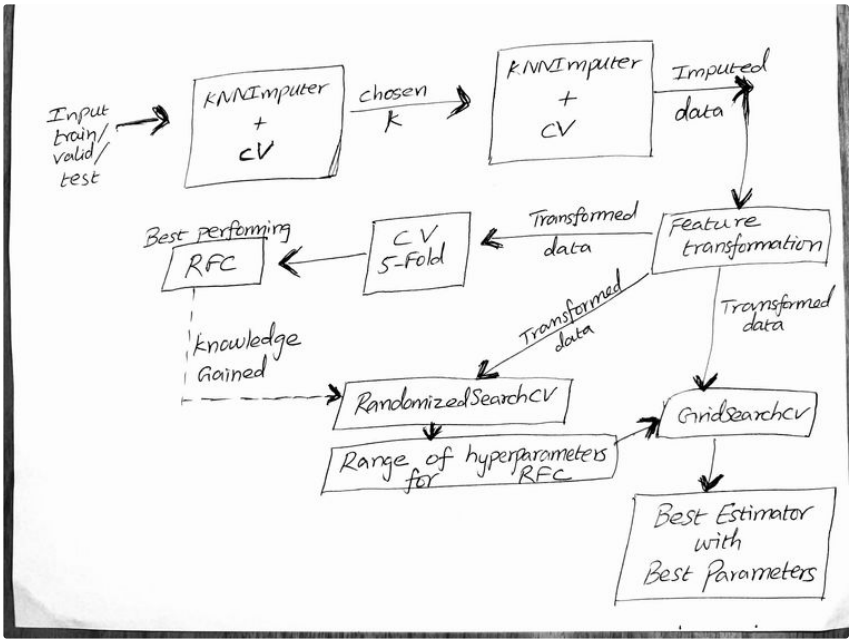
One way to accomplish the objective is to create a strong predictive model. Once we have verified the model's performance, we can utilize its feature selection attribute to identify the significant factors influencing churn.

Why machine learning?

1. Many ML models, especially supervised models, have feature selection attributes that can be used to discover significant features/factors affecting churn. A supervised model becomes possible with target variables indicating which customer churned and who has not, also called labels. Luckily, the churn dataset had labels, so we could build a supervised model and leverage the benefits.
2. Further ML techniques, such as cross-validation, can build robust training and hyperparameter tuning loops so that it becomes possible to confirm strong training, validation, and test splits without leaking data from the training to the validation set.
3. Supervised models have better predictive metrics, making the monitoring and validating processes much easier to implement. The predictive models have various metrics to validate a model's performance and the quality of its outcome.

Different ML techniques in the project

This pictorial demonstrates different machine learning techniques in this project to ensure model quality.



Cross-Validation

Throughout the cross-validation process, the training data gets split into train and validation data sets using the command:

```
kf = StratifiedKFold(n_splits=nsplit, shuffle=True)

for i, (train_index, val_index) in enumerate(kf.split(X, y)):
            x_train = X.loc[train_index]
            x_val = X.loc[val_index]
            y_train = y[train_index]
            y_val = y[val_index]
```

where:

A fewer minority class, i.e., 1, indicating that a customer has churned, exists in the data, occurring at 16% compared to the majority class label, 0 at 83% of occurrence. Stratified is chosen to ensure that each fold of the dataset has the same proportion of observations with a given label.

X represents the training data.

y represents the target variable.

This technique is implemented in the following contexts:

KNNImputer - made dynamic

We take the input data, be it train/valid/test, and send it through a cross-validation loop where KNNImputer replaces the missing values by considering the values from a group of neighbors (referred to as k in KNN) that exist in the data. After imputing, there is a need to validate how good they are. We do that by fitting a Random Forest Classifier with a fixed set of parameters on the imputed training data.

```
model = RandomForestClassifier(max_depth=12, random_state=2076)

model.fit(xtrain_imputed, y_train)

We test the model’s performance by predicting the imputed validation data.

y_pred = model.predict(xval_imputed)

F1_score is used to validate how good the predictions are.

f1_value = round(f1_score(y_val, y_pred), 2)
```

Dynamically choosing the hyperparameter K in KNNImputer

(from sklearn.impute import KNNImputer)

The training and validation sets are imputed separately, through 5-fold cross-validation, to ensure that info on neighbors from the training data is not leaked into the validation data. We choose from a range of integer values for neighbors, and for every value, a 5-fold cross-validation is run. Thus for every value we store the corresponding f1_score. The maximum f1_score is looked up from a dictionary containing

and the respective neighbor value is returned as outcome.

This neighbor value is then dynamically assigned to the KNNImputer used to impute the missing values in

The k in KNN (k here represents the number of neighbors to consider during the imputation process) using KNNImputer that can later be used as the cross-validated k in KNNImputer to impute the missing values in X_train, X_val, and X_test where X_train represents training data, X_val represents validation data, and X_test represents testing

Curse of Dimensionality

```
for n in range(3, 13):
        element_average = []
        for i, (train_index, val_index) in enumerate(kf.split(X, y)):
            x_train = X.loc[train_index]
            x_val = X.loc[val_index]
            y_train = y[train_index]
            y_val = y[val_index]
            x_train = scaler.fit_transform(x_train)
            x_val = scaler.fit_transform(x_val)
            train_imputer = KNNImputer(n_neighbors=n, weights='uniform', metric='nan_euclidean')
            xtrain_imputed = train_imputer.fit_transform(x_train)
            val_imputer = KNNImputer(n_neighbors=3, weights='uniform', metric='nan_euclidean')
            xval_imputed = val_imputer.fit_transform(x_val)
            model = RandomForestClassifier(max_depth=12, random_state=2076)
            model.fit(xtrain_imputed, y_train)
            y_pred = model.predict(xval_imputed)
            f1_value = round(f1_score(y_val, y_pred), 2)
```

Point to note:

A default value for parameter n_neighbors worked well for the validation data in every iteration.

RandomizedSearchCV

- I used this technique to choose the best-performing model from a choice of models using RandomizedSearchCV.
- To narrow down to a range of hyperparameter values that can be used in GridSearch later to identify the optimum hyperparameters for the final model.
- To identify overfitting when it happens and tackle it by taking appropriate measures.

The best parameters were chosen after this process:

```
Best score reached: 0.76

Best parameters: {'n_estimators': 13, 'min_samples_split': 8, 'min_samples_leaf': 4, 'max_features': 10, 'max_depth': 13}
```

GridSearchCV

Hyperparameter tuning is also done manually in the randomized search process, but grid search is an exhaustive search over a list of chosen hyperparameters with a range of values narrowed down from the random search process. The best parameter values after this process look like this:

```
##################################################
The best parameters are {'max_depth': 13, 'max_features': 9, 'min_samples_leaf': 3, 'min_samples_split': 8, 'n_estimators': 17}

##################################################
Mean cross-validated score of the best_estimator is 0.7776813234474537
```

After the grid search, the tuned model is chosen as the final model, ready to predict the unseen data.

Feature Engineering

Train-Test Split

The churn data is split into training and test sets using the command:

train_data, test_data = train_test_split(df_churn, test_size=test_size, random_state=random_state)

The training is used to train an ML model, whereas the test set is kept aside for testing purposes and used only once when the final trained model gets ready to predict using the completely unseen data, thereby reflecting a production scenario.

Feature Transformation

All the following transformations are validated using F1_score through a well-established cross-validation process.

- Assuming 0s in Tenure represent the customers with the organization for less than a month, I have replaced them with 0.5.
- Replaced 'Mobile Phone' with 'Phone' in PreferredLoginDevice.
- Did the following categorical transformations in PreferredPaymentMode:

```
x["PreferredPaymentMode"] = x["PreferredPaymentMode"].map({"Debit Card": "debit", "Credit Card": "credit", "E wallet": "wallet",
                                                           "UPI": "upi", "COD": "cash_delivery", "CC": "credit",
                                                           "Cash on Delivery": "cash_delivery"})
```

- Replaced the outlier values in WarehouseToHome with null values and imputed them along with the other existing nulls in the column through KNNImputer technique.

```
x['WarehouseToHome'] = x.apply(lambda row: np.nan if row['WarehouseToHome']>120 else row['WarehouseToHome'], axis=1)
```

- Replaced the null values in columns PreferredPaymentMode, PreferedOrderCat, and MaritalStatus through frequency encoding:

```
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1098)

for i, (train_index, val_index) in enumerate(kf.split(x, y)):
    x_train = x.loc[train_index]
    x_val = x.loc[val_index]
    y_train = y[train_index]
    y_val = y[val_index]


    x_train = frequency_encoding(x_train, "PreferredPaymentMode")
    x_val = frequency_encoding(x_val, "PreferredPaymentMode")
    x_train = frequency_encoding(x_train, "PreferedOrderCat")
    x_val = frequency_encoding(x_val, "PreferedOrderCat")
    x_train = frequency_encoding(x_train, "MaritalStatus")
    x_val = frequency_encoding(x_val, "MaritalStatus")
```

- CityTier has values 1, 2, and 3. It's possible that an ML model would misunderstand these values, thinking 3 is greater than 1. To avoid that, I swapped the values in CityTier as below:

```
x["CityTier"] = x["CityTier"].map({1: 3, 2: 2, 3:1})
```

Complain represents customers' dissatisfaction and therefore it does not make sense to have a value 1 represent it. After experimenting and validating, I changed the complain values into -1:

```
x["Complain"].replace(to_replace=1, value=-1, inplace=True)
```

Challenges

Imbalanced Class

The churn data has an imbalanced class issue, too:

```
train_data["Churn"].value_counts(dropna=False)
```

```
Churn
0    4396
1     884
Name: count, dtype: int64
```

Experimentation is the key:

- I used the stratify option while splitting the training data in every cross-validation process.
- I examined resampling techniques such as SMOTETomek and SMOTEENN to increase the number of minority samples, but they led to poor performance and did not work in this data context.

Overfitting

With a relatively small data size as the churn data, overfitting is expected. The model (here refers to the chosen Random Forest Classifier model) did overfit with a mean training f1_score at 99%

Some of the measures undertaken to tackle it are:

- Reducing the number of estimators started to show some positive changes in reducing overfitting.
- Altering some feature values and experimenting with dropping some correlated ones through calculating Pearson Correlation Coefficient and multicollinearity provided some direction and refined overall.
- Adding the hyperparameter, min_samples_leaf improved the effort quite a lot.

Preventing data leak

Meticulously dealt with this concept by carefully splitting the training data into validation through the benefits of cross-validation.

- Feature transformation logic involving replacing null values through frequency encoding and KNNImputer imputation method is carried out separately on train, validation, and test datasets to prevent any leaking of training data into validation data.

```
x_train = np.array(x_train)
x_val = np.array(x_val)
train_imputer = KNNImputer(n_neighbors=nneighbors[0], weights='uniform', metric='nan_euclidean')
xtrain_imputed = train_imputer.fit_transform(x_train)
val_imputer = KNNImputer(n_neighbors=3, weights='uniform', metric='nan_euclidean')
xval_imputed = val_imputer.fit_transform(x_val)
```

- When experimented with resampling techniques such as SMOTETomek, the logic of applying separately on train and validation sets continued.

The highlighted part of the following code sample illustrates how frequency encoding of the features PreferredPaymentMode, PreferredOrderCat, and MaritalStatus is done separately on the training and validation sets.

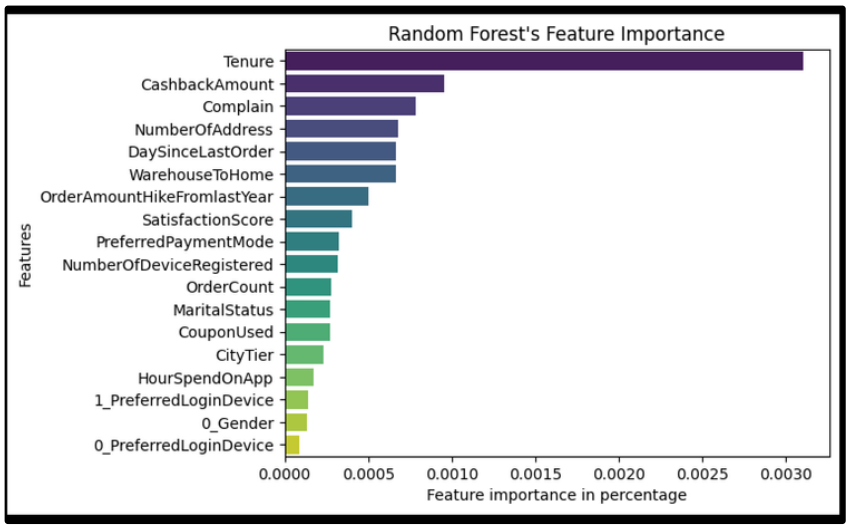
```python
#This function imputes the missing values in the training data using the neighbors selected from the function cv_knn.
def feature_engineering_cv(df, date, version, model_name):
    x = df.copy()
    x.columns = x.columns.str.strip()
    x.drop("CustomerID", axis=1, inplace=True)
    avg_f1_score = []
    decimals = 2
    # Imputing only the 0s in Tenure
    x['Tenure'] = x.apply(lambda row: 0.5 if row['Tenure']==0.0 else row['Tenure'], axis=1)
    x["PreferredLoginDevice"] = x["PreferredLoginDevice"].replace(to_replace=['Mobile Phone'], value='Phone')
    x["PreferredPaymentMode"] = x["PreferredPaymentMode"].map({"Debit Card": "debit", "Credit Card": "credit", "E wallet": "wallet", "UPI": "u
    x["PreferedOrderCat"] = x["PreferedOrderCat"].replace(to_replace=['Mobile Phone', "Laptop & Accessory"], value=['Mobile', "Laptop_accessor
    x['WarehouseToHome'] = x.apply(lambda row: np.nan if row['WarehouseToHome']>120 else row['WarehouseToHome'], axis=1)
    x = feature_encoding(x, "PreferredLoginDevice")
    x = feature_encoding(x, "Gender")
    x["Complain"].replace(to_replace=1, value=-1, inplace=True)
    x["CityTier"] = x["CityTier"].map({1: 3, 2: 2, 3:1})
    x["CityTier"] = x["CityTier"].astype('int32')
    x["Complain"] = x["Complain"].astype('int32')
    x["NumberOfDeviceRegistered"] = x["NumberOfDeviceRegistered"].astype('int32')
    x["SatisfactionScore"] = x["SatisfactionScore"].astype('int32')
    x["NumberOfAddress"] = x["NumberOfAddress"].astype('int32')
    x["OrderAmountHikeFromlastYear"] = x["OrderAmountHikeFromlastYear"].astype('float32')
    x["CashbackAmount"] = x["CashbackAmount"].astype('float32')
    x.drop('1_Gender', axis=1, inplace=True)
    # Imputing missing values in WarehouseToHome, HourSpendOnApp, OrderAmountHikeFromlastYear, CouponUsed, OrderCount, DaySinceLastOrder,
    #Tenure
    nneighbors = cv_knn(x, 5, date, model_name)
    y = x["Churn"]
    x.drop("Churn", axis=1, inplace=True)
    X_train, X_val, y_train, y_val = train_test_split(x, y, test_size=0.05, stratify=y, shuffle=True, random_state=2098)
    X_train = frequency_encoding(X_train, "PreferredPaymentMode")
    X_val = frequency_encoding(X_val, "PreferredPaymentMode")
    X_train = frequency_encoding(X_train, "PreferedOrderCat")
    X_val = frequency_encoding(X_val, "PreferedOrderCat")
    X_train = frequency_encoding(X_train, "MaritalStatus")
    X_val = frequency_encoding(X_val, "MaritalStatus")
    X_train["PreferredPaymentMode"] = X_train["PreferredPaymentMode"].apply(lambda x: round(x, decimals))
    X_train["MaritalStatus"] = X_train["MaritalStatus"].apply(lambda x: round(x, decimals))
    X_train["PreferedOrderCat"] = X_train["PreferedOrderCat"].apply(lambda x: round(x, decimals))
    X_train.drop('PreferedOrderCat', axis=1, inplace=True)
    X_val.drop('PreferedOrderCat', axis=1, inplace=True)
    train_imputer = KNNImputer(n_neighbors=nneighbors[0], weights='uniform', metric='nan_euclidean')
    xtrain_imputed = train_imputer.fit_transform(X_train)
    val_imputer = KNNImputer(n_neighbors=3, weights='uniform', metric='nan_euclidean')
    xval_imputed = val_imputer.fit_transform(X_val)
    # smt = SMOTETomek(sampling_strategy='minority', random_state=3050)
    # xtrain_resampled, ytrain_resampled = smt.fit_resample(xtrain_imputed, y_train)
    return xtrain_imputed, xval_imputed, y_train, y_val, X_train.columns.to_list()
```

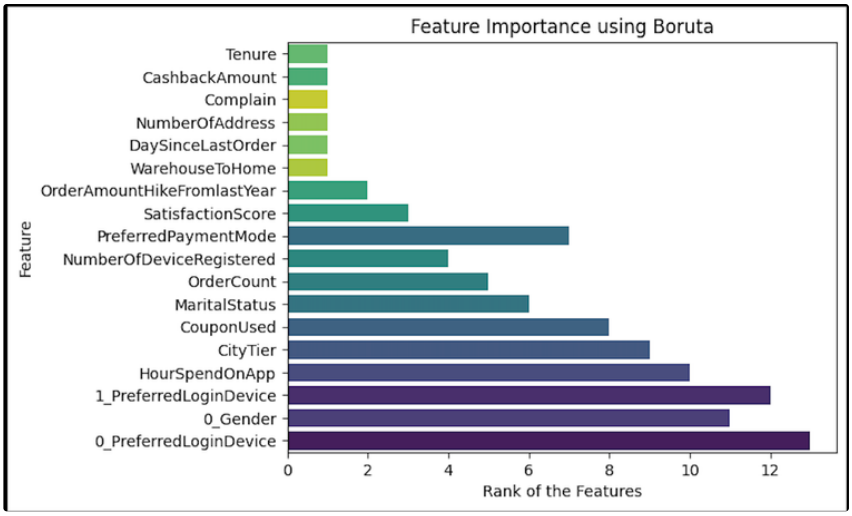
Feature Importance

Random Forest's Vs. Boruta feature importance

Both have predicted different features as significant. But I choose to trust Boruta over Random Forest Classifier's (referred as RFC) feature selection attribute because of the following:

1. RFC predicts Tenure as the most significant, but at the same time, the rest of the features are way less predictable. We know From EDA that some features, such as CityTier and HourSpendOnApp, have displayed strong positive and (not so strong) negative correlations with the target variable, Churn.
2. Moreover, Boruta is a wrapper function over the RFC feature selection attribute, which enhances its capability to identify the most relevant features in a dataset. I like it because it provides a good global optimization for feature selection. Some of the features that have gained importance through this technique, such as PreferredLoginDevice, CityTier, and HourSpendOnApp seem to make sense.





Final model

The final trained Random Forest model is chosen based on the model's ability to showcase a consistent range of differences between the mean training and validation score throughout the cross-validation phase. The model successfully performed at the expected range on the unseen data, i.e., the test data that was set aside. This ensures the model's generalization ability so we can trust that it uses significant factors as accurately as possible for its final predictions.

The final trained Random Forest Classifier model's performance on test data:

Predicted x_test and y_test

```
##################################################
confusion_matrix for test data is
 [[281   5]
 [ 13  51]]
##################################################
```

True y_test	
0	286
1	64

Of 64 churned values, i.e., minority class, the model got 51 right, a difference of 13 (that the model predicted churn but did not churn).

The final trained Random Forest Classifier model's performance after the randomized search process:

Predicted x_val and y_val

```
##################################################

confusion_matrix is
 [[213   7]
 [ 12  32]]

##################################################
```

where the actual y values are:

True y_val	
0	220
1	44

Of 44 true churned values, i.e., minority class, the model got 32 right, a difference of 12 (that the model predicted churn but did not churn).

Insights and Recommendations

The significant features listed in order of their significance, using the Bortuga algorithm, are as follows:

1. 0_PreferredLoginDevice
2. 1_PreferredLoginDevice
3. 0_Gender
4. HourSpendOnApp
5. CityTier
6. CouponUsed
7. PreferredPaymentMode

0_PreferredLoginDevice, 1_PreferredLoginDevice, and 0_Gender

From EDA (Exploratory Data Analysis), we have gathered an approx. 61% of single male customers have churned. These customers fall into a risky zone. Let's see some steps to address this:

Analysis: Further, segment this group to identify more specific patterns to understand whether there are certain age groups, income levels, or specific geographic locations that are more prone to churn.

Pain points: I wish to understand why they are leaving. Because more than 70% of customers use mobile phones (learned from EDA), it's possible to analyze from that angle, as other mobile shopping apps may provide better options concerning `PreferedOrderCat` /experiences.

Proactive monitoring:

To effectively mitigate customer churn, particularly within the identified high-risk segments, I propose a proactive enhancement of the user experience (UX). This strategy includes rigorously monitoring user interactions within the app, focusing on key landing pages. We could plan to employ robust statistical techniques, such as A/B testing and multivariate analysis, to assess the effectiveness of the page layouts critically, call-to-action (CTA) buttons, and the overall navigation flow.

Calculate stats, such as the month-over-month churn rate for both single male customer segments, to learn of any alarming increasing trend.

HourSpendOnApp

We've observed instances of '0' in the HourSpendOnApp field. Assuming these zeros are accurate and not data entry errors, they suggest that a segment of the customers spends less than an hour on the business app. Within this group of 258 customers, 29% have already churned. Given that the vast majority (95%) of the customer base spends an average of 2.7 hours on the app, the remaining customers in this low-engagement segment are likely at a heightened risk of churn. This discrepancy in app usage time highlights a significant engagement gap that we must proactively address.

The OrderAmountHikeFromlastYear for this segment stays the same as that of the customer segments who spend over 2 hours at approx. 14% A similar trend continues between the segments with other factors such as **OrderCount**, **CouponUsed**, and **CashbackAmount.** The only minor variance is observed in the **DaySinceLastOrder**, which differs by two days. So, it is unclear what might be causing this. Some potential areas to focus on are:

Buying patterns: are they making fewer purchases or more sporadic? And learn their average purchase value compared to that of other segments. Alternatively, the fact that this segment spends less than an hour on the app might not be as concerning as initially thought.

Relevant content: to enhance the recommendation engine to show more relevant products that align with their interests and past behavior.

Conduct qualitative research, such as customer interviews or surveys, with a representative sample from this segment to understand their needs and experiences.

PreferredPaymentMode

Approx. 40% of customers use debit cards as their preferred payment mode. This does not consider the types of cards stored in a wallet and used via UPI mode. It's possible to assume that customers using debit cards are more cautious about spending money than customers who use credit cards. This could lead to a longer duration in DaySinceLastOrder. It's worth comparing this group's behavior with customers using other payment methods (like credit cards) to understand differences in their spending habits, purchase frequency, and average transaction values.