StepwiseFeatureSelection

August 8, 2023

1 Stepwise Feature Selection

In this exercise, you will implement stepwise feature selection.

- You will train decision tree models on "cell2cell," a telecom company churn prediction data set.
- After using the optimal hyperparameter configuration that results in the best performing decision tree, you will perform feature selection to find the most important features in your training data for predicting customer churn.

Note: Some of the code cells in this notebook may take a while to run.

1.0.1 Import Packages

Before you get started, import a few packages. Run the code cell below.

[1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
```

We will also import the scikit-learn `DecisionTreeClassifier`, the `train_test_split()` function for splitting the data into training and test sets, and the metric `accuracy_score` to evaluate your model.

[2]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

1.1 Step 1. Load a 'ready-to-fit' Data Set

We will work with the "cell2celltrain" data set. This data set is already preprocessed, with the proper formatting, outliers and missing values taken care of, and all numerical columns scaled to the [0, 1] interval. One-hot encoding has been performed on all categorical columns. Run the cell below to load the data set and save it to DataFrame `df`.

[3]:
```
filename = os.path.join(os.getcwd(), "data", "cell2celltrain.csv")
df = pd.read_csv(filename, header=0)
```

1.2 Step 2: Create Training and Test Data Sets

1.2.1 Create Labeled Examples

The code cell obtains columns from our data and creates features and labels.

[4]:
```
y = df['Churn']
X = df.drop(columns = 'Churn', axis=1)
X.head()
```

[4]:
```
   CustomerID  ChildrenInHH  HandsetRefurbished  HandsetWebCapable  \
0     3000002         False               False               True
1     3000010          True               False              False
2     3000014          True               False              False
3     3000022         False               False               True
4     3000026         False               False              False

   TruckOwner  RVOwner  HomeownershipKnown  BuysViaMailOrder  \
0       False    False                True              True
1       False    False                True              True
2       False    False               False             False
3       False    False                True              True
4       False    False                True              True

   RespondsToMailOffers  OptOutMailings  ...  Occupation_Crafts  \
0                  True           False  ...                0.0
1                  True           False  ...                0.0
2                 False           False  ...                1.0
3                  True           False  ...                0.0
4                  True           False  ...                0.0

   Occupation_Homemaker  Occupation_Other  Occupation_Professional  \
0                   0.0               0.0                      1.0
1                   0.0               0.0                      1.0
2                   0.0               0.0                      0.0
3                   0.0               1.0                      0.0
4                   0.0               0.0                      1.0

   Occupation_Retired  Occupation_Self  Occupation_Student  Married_False  \
0                 0.0              0.0                 0.0            1.0
1                 0.0              0.0                 0.0            0.0
2                 0.0              0.0                 0.0            0.0
3                 0.0              0.0                 0.0            1.0
4                 0.0              0.0                 0.0            0.0

   Married_True  Married_nan
0           0.0          0.0
1           1.0          0.0
2           1.0          0.0
```

```
3            0.0           0.0
4            1.0           0.0

[5 rows x 84 columns]
```

1.2.2 Split Examples Into Training and Test Sets

The code cell below creates training and test data sets. Since we will be performing model selection, we will split 10% of our data to serve as a test set.

[5]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.10,␣
↪random_state=1234)
```

1.3 Step 3: Perform Decision Tree Model Selection Using Grid Search

For brevity, let's get back to where we left off in the previous exercise: We already performed a grid search over four different model configurations to find the optimal hyperparameter values for the maximum depth of the tree (`max_depth`) and the minimum number of samples required to be at a leaf node (`min_samples_leaf`).

Since we just re-used the same random seed to split our data into the training and test sets, it is safe to set the best hyperparameter values to what we identified in the previous exercise without re-running the grid search.

[6]:
```
# Do not edit this cell
best_params = {'max_depth':4, 'min_samples_leaf':50}
best_params
```

[6]:
```
{'max_depth': 4, 'min_samples_leaf': 50}
```

1.4 Step 4: Implement Stepwise Feature Selection

We will implement our version of stepwise feature selection to find the top performing model. We will continue to use the hyperparameter configuration that we have found to produce an optimal model; we will focus on finding the combination of features that produces the best performing model.

We will write a program that fits a decision tree model to the training data then eliminates the least important feature and re-trains the model on a reduced set. We will iterate this process of eliminating and re-training until our stopping criterion is met: The accuracy of the updated model increases by less than 0.1. Once the stopping criterion is met, we will record which features we are left with and consider them the best ones to use.

Recall that all model selection is done on *validation data*, not the test set. Therefore, we will use the training data that we created before, and split the training data `X_train` and `y_train` into training and validation sets. We indirectly did this before by using cross-validation functions in `sklearn`. This time, we will perform only one split, and will do so manually. We will perform this split at every iteration of our stepwise feature selection process.

To accomplish a stepwise feature selection implementation, we will create a few functions: * `train_evaluate_DT_model()`: to train and evaluate a decision tree model * `feature_to_drop()`: to "drop" the lowest scoring feature * `stepwise_feature_selection()`: performs feature selection

1.4.1 a. Function to Train and Evaluate a Decision Tree

Use: The function will be used to train a model with a subset of training data.
Arguments: 1. X: features 2. y: label 3. max_depth 4. min_samples_leaf
Returns: 1. the accuracy score 2. the model object 3. a list of the training features

[7]:
```
def train_evaluate_DT_model(X_train, X_val, y_train, y_val, max_depth,␣
↪min_samples_leaf):

    #1. Create a DecisionTreeClassifier model object with the best hyperparam␣
↪values
    model = DecisionTreeClassifier(max_depth = max_depth,
                                   min_samples_leaf = min_samples_leaf)

    #2. Fit the model to the training data
    model.fit(X_train, y_train)

    #3. Make predictions on the validation data
    class_label_predictions = model.predict(X_val)

    #4. Compute the accuracy
    acc_score = accuracy_score(y_val, class_label_predictions)

    #. Return the accuracy score and the model object
    return acc_score, model
```

1.4.2 b. Function to Return the Worst Scoring Feature

Use: The function will use the feature importance scores from the `model` input to find the name of the feature that has the lowest score.
Arguments: 1. decision tree `model` object 2. `numpy` array of feature names
Returns: 1. the name of the lowest scoring feature.
In the code cell below, you will find the least important feature in your training data for predicting churn.
Perform the following steps:

1. Using `model.feature_importances_`, obtain scores corresponding to the importance of the predictive features. Save the result to the variable `feature_imp`.
2. Create a Pandas DataFrame out of all feature names and their measures of importance by using the `pd.DataFrame()` function. Call the function with a dictionary containing the following key/value pairs:

   - `'name': feature_names`
   - `'imp': feature_imp`

   Assign the DataFrame to the variable `df_features`.

3. Using the Pandas method `sort_values()`, sort the importance scores in the `imp` column in the new DataFrame `df_features` in descending order. Assign the resulting DataFrame to variable `df_sorted`.

4. Using `iloc`, extract the last value in `df_sorted`. Then extract the value in the `name` column and save it to the variable `lowest_scoring_feature`.

1.4.3 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[14]:
```
def feature_to_drop(model, feature_names):

    #1. Obtain "feature importance" scores from the model object and save the␣
 ↪list to the
    # variable 'feature_imp'
    # YOUR CODE HERE
    feature_imp=model.feature_importances_

    #2. Create a Pandas DataFrame with a list of all features and their scores.
    # Save the result to the variable 'df_features'
    # YOUR CODE HERE
    df_features=pd.DataFrame({'name':feature_names,'imp':feature_imp})

    #3. Sort df_features in descending order and
    # save the result to the variable 'df_sorted'
    # YOUR CODE HERE
    df_sorted=df_features.sort_values('imp',ascending=False)

    #4. Obtain the last feature name and save the result to variable␣
 ↪'lowest_scoring_feature'
    # YOUR CODE HERE
    lowest_scoring_feature=df_sorted['name'].iloc[-1]


    return lowest_scoring_feature
```

1.4.4 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[15]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testHFS

try:
    p, err = testHFS(df, feature_to_drop)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

1.4.5 b. Function to Perform Stepwise Feature Selection

Use: The function will iteratively remove the least important features from the training set, until the accuracy no longer improves by more than 1%. The function uses the two functions implemented above to accomplish this.

Arguments: 1. X_train: features 2. y_train: label 3. max_depth 4. min_samples_leaf

Returns: The training data containing the best performing features

[16]:
```
def stepwise_feature_selection(X_train, y_train, max_depth, min_samples_leaf):
    """
    This function iteratively removes features from the original training set
    until the accuracy no longer improves by more than 1%.
    """

    best_training_features = X_train  # keeps track of the best performing␣
↪features
    total_num_features = X_train.shape[1] # start with 10 features
    last_acc_score = None  # keeps track of accuracy scores


    for num_features in reversed(range(1, total_num_features+1)):

        # resample to get new training and validation sets out of our training␣
↪data
        X_train_temp, X_val_temp, y_train_temp, y_val_temp  =␣
↪train_test_split(X_train,
                                                                  y_train, test_size=.
↪2,
                                                                  random_state=1234)

        # train model and get accuracy score
        acc, model = train_evaluate_DT_model(X_train_temp, X_val_temp,␣
↪y_train_temp,
                                                      y_val_temp, max_depth,␣
↪min_samples_leaf)
        print("Accuracy for top {0} features: {1}".format(num_features, acc))

        # if accuracy improves, save the training data so as to keep track that␣
↪it contains
        # the best performing features so far
        if last_acc_score and (acc > last_acc_score):
            best_training_features = X_train

        last_acc_score = acc

        # get lowest performing feature from the training data
        worst_feature_name = feature_to_drop(model, X_train_temp.columns.values)

        # remove lowest performing feature from training data
        X_train = X_train.drop(columns=[worst_feature_name])

    return best_training_features
```

The cell below calls the `stepwise_feature_selection()` function to perform feature selection. Note the following:

1. We will be working with our original training data `X_train` and `y_train`.
    - We will start by changing `X_train` so that it only contains the top 10 features we discovered above.
    - In the function `stepwise_feature_selection()`, we will be splitting `X_train` and `y_train` into training and validation (`X_val` and `y_val`) subsets.
2. We will train our model with the best hyperparameter values we discovered above.
3. The function `stepwise_feature_selection()` returns a DataFrame consisting of the training data that contains the features that result in the best performing DT model.

At the end of this process we will have: 1. The optimal hyperparameter configuration. 2. The optimal features.

[ ]:
```
best_md = best_params['max_depth']
best_msl = best_params['min_samples_leaf']

best_performing_features = stepwise_feature_selection(X_train, y_train,
                                                      best_md, best_msl)
```

```
Accuracy for top 84 features: 0.7208619000979432
Accuracy for top 83 features: 0.7208619000979432
Accuracy for top 82 features: 0.7208619000979432
Accuracy for top 81 features: 0.7208619000979432
Accuracy for top 80 features: 0.7208619000979432
Accuracy for top 79 features: 0.7208619000979432
Accuracy for top 78 features: 0.7208619000979432
Accuracy for top 77 features: 0.7208619000979432
Accuracy for top 76 features: 0.7208619000979432
Accuracy for top 75 features: 0.7208619000979432
Accuracy for top 74 features: 0.7208619000979432
Accuracy for top 73 features: 0.7208619000979432
Accuracy for top 72 features: 0.7208619000979432
Accuracy for top 71 features: 0.7208619000979432
Accuracy for top 70 features: 0.7208619000979432
Accuracy for top 69 features: 0.7208619000979432
Accuracy for top 68 features: 0.7208619000979432
Accuracy for top 67 features: 0.7208619000979432
Accuracy for top 66 features: 0.7208619000979432
Accuracy for top 65 features: 0.7208619000979432
Accuracy for top 64 features: 0.7208619000979432
Accuracy for top 63 features: 0.7208619000979432
Accuracy for top 62 features: 0.7208619000979432
Accuracy for top 61 features: 0.7208619000979432
Accuracy for top 60 features: 0.7208619000979432
Accuracy for top 59 features: 0.7208619000979432
Accuracy for top 58 features: 0.7208619000979432
Accuracy for top 57 features: 0.7208619000979432
Accuracy for top 56 features: 0.7208619000979432
Accuracy for top 55 features: 0.7208619000979432
Accuracy for top 54 features: 0.7208619000979432
Accuracy for top 53 features: 0.7208619000979432
Accuracy for top 52 features: 0.7208619000979432
Accuracy for top 51 features: 0.7208619000979432
Accuracy for top 50 features: 0.7208619000979432
Accuracy for top 49 features: 0.7208619000979432
Accuracy for top 48 features: 0.7208619000979432
Accuracy for top 47 features: 0.7208619000979432
Accuracy for top 46 features: 0.7208619000979432
Accuracy for top 45 features: 0.7208619000979432
Accuracy for top 44 features: 0.7208619000979432
Accuracy for top 43 features: 0.7208619000979432
Accuracy for top 42 features: 0.7208619000979432
Accuracy for top 41 features: 0.7208619000979432
Accuracy for top 40 features: 0.7208619000979432
Accuracy for top 39 features: 0.7208619000979432
Accuracy for top 38 features: 0.7208619000979432
Accuracy for top 37 features: 0.7208619000979432
Accuracy for top 36 features: 0.7208619000979432
Accuracy for top 35 features: 0.7208619000979432
Accuracy for top 34 features: 0.7208619000979432
Accuracy for top 33 features: 0.7208619000979432
Accuracy for top 32 features: 0.7208619000979432
Accuracy for top 31 features: 0.7208619000979432
Accuracy for top 30 features: 0.7208619000979432
Accuracy for top 29 features: 0.7208619000979432
Accuracy for top 28 features: 0.7208619000979432
Accuracy for top 27 features: 0.7208619000979432
Accuracy for top 26 features: 0.7208619000979432
Accuracy for top 25 features: 0.7208619000979432
Accuracy for top 24 features: 0.7208619000979432
Accuracy for top 23 features: 0.7208619000979432
Accuracy for top 22 features: 0.7208619000979432
Accuracy for top 21 features: 0.7208619000979432
Accuracy for top 20 features: 0.7208619000979432
Accuracy for top 19 features: 0.7208619000979432
Accuracy for top 18 features: 0.7208619000979432
Accuracy for top 17 features: 0.7208619000979432
Accuracy for top 16 features: 0.7208619000979432
Accuracy for top 15 features: 0.7208619000979432
Accuracy for top 14 features: 0.7208619000979432
Accuracy for top 13 features: 0.7208619000979432
Accuracy for top 12 features: 0.7208619000979432
Accuracy for top 11 features: 0.7208619000979432
Accuracy for top 10 features: 0.7208619000979432
Accuracy for top 9 features: 0.7208619000979432
Accuracy for top 8 features: 0.7208619000979432
Accuracy for top 7 features: 0.7208619000979432
```

The code cell below outputs the best performing features given the heuristics that we decided upon for choosing the number of features.

[ ]:
```python
print("\nBest Performing training data (best features):\n")
best_performing_features.columns
```

1.5 Step 5. Fit the Best Model

Now that we have the hyperparameter configuration and features that produce the best decision tree model, we can fit a `DecisionTreeClassifier` with those values.

The code cell below fits *one* decision tree classifier using the best hyperparameters and features identified, tests the model on the test set (`X_test`), and obtains the final accuracy score of the model's class label predictions.

[ ]:
```python
# 1. Create a DecisionTreeClassifier model object
model = DecisionTreeClassifier(max_depth = best_md,
                               min_samples_leaf = best_msl)

# 2. Fit the model to the training data
model.fit(best_performing_features, y_train)



# 3. Make predictions on the test data
#(Appropriately, we are only using the features we trained on to test our␣
 ↪model)
X_test = X_test[['MonthlyMinutes', 'MonthsInService', 'CurrentEquipmentDays']]
class_label_predictions = model.predict(X_test)

# 4. Compute the accuracy
acc_score = accuracy_score(y_test, class_label_predictions)
```

```
print('Accuracy score: {0}'.format(acc_score))
```

[ ]:

```

```