KNNOptimization

August 8, 2023

1 k-Nearest Neighbors Optimization

In this exercise, you will train multiple KNN Classification models using using different values of hyperparameter K and compare the accuracy of each model. You will train the KNN models on "cell2cell" -- a telecom company churn prediction data set.

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 `KNeighborsClassifier`, the `train_test_split()` function for splitting the data into training and test sets, and the metric `accuracy_score` to evaluate our model.

[2]:
```
from sklearn.neighbors import KNeighborsClassifier
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.

1.1.1 Load a Data Set and Save it as a Pandas DataFrame

We will work with a new data set called "cell2celltrain." 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.

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

[4]:
```
df.shape
```

[4]:
```
(51047, 58)
```

[5]:
```
df.head()
```

[5]:
```
   CustomerID  Churn ServiceArea  ChildrenInHH  HandsetRefurbished  \
0     3000002   True    SEAPOR503         False               False
1     3000010   True    PITHOM412          True               False
2     3000014  False    MILMIL414          True               False
3     3000022  False    PITHOM412         False               False
4     3000026   True    OKCTUL918         False               False

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

   BuysViaMailOrder  ...  HandsetModels  CurrentEquipmentDays    AgeHH1  \
0              True  ...       0.487071             -0.077013  1.387766
1              True  ...      -0.616775              3.019920  0.392039
2             False  ...      -0.616775              3.019920 -0.241605
3              True  ...       2.694763              0.305179 -0.060564
4              True  ...       1.590917              1.857585  0.663601

     AgeHH2  RetentionCalls  RetentionOffersAccepted  \
0 -0.883541        4.662897                  -0.1283
1  0.871495       -0.180167                  -0.1283
2  0.202910       -0.180167                  -0.1283
3 -0.883541       -0.180167                  -0.1283
4  1.372934       -0.180167                  -0.1283

   ReferralsMadeBySubscriber  IncomeGroup  AdjustmentsToCreditRating  \
0                  -0.169283    -0.103411                  -0.140707
1                  -0.169283     0.215243                  -0.140707
2                  -0.169283     0.533896                  -0.140707
3                  -0.169283     0.533896                  -0.140707
4                  -0.169283     1.489856                   2.469282

  HandsetPrice
0    -0.864858
1    -0.864858
2    -0.368174
3    -1.195980
4    -1.195980

[5 rows x 58 columns]
```

1.1.2 Remove String Columns

To train a k-Nearest Neighbors model on our dataset, we must first remove those features for which computing the distance is impossible: the string-valued, categorical variables. Inspect the data type of each column in the code cell below.

[6]:
```
df.dtypes
```

[6]:
```
CustomerID                   int64
Churn                         bool
ServiceArea                 object
ChildrenInHH                  bool
HandsetRefurbished            bool
HandsetWebCapable             bool
TruckOwner                    bool
RVOwner                       bool
HomeownershipKnown            bool
BuysViaMailOrder              bool
RespondsToMailOffers          bool
OptOutMailings                bool
NonUSTravel                   bool
OwnsComputer                  bool
HasCreditCard                 bool
NewCellphoneUser              bool
NotNewCellphoneUser           bool
OwnsMotorcycle                bool
MadeCallToRetentionTeam       bool
CreditRating                object
PrizmCode                   object
Occupation                  object
Married                     object
MonthlyRevenue             float64
MonthlyMinutes             float64
TotalRecurringCharge       float64
DirectorAssistedCalls      float64
OverageMinutes             float64
RoamingCalls               float64
PercChangeMinutes          float64
PercChangeRevenues         float64
DroppedCalls               float64
BlockedCalls               float64
UnansweredCalls            float64
CustomerCareCalls          float64
ThreewayCalls              float64
ReceivedCalls              float64
OutboundCalls              float64
InboundCalls               float64
PeakCallsInOut             float64
OffPeakCallsInOut          float64
```

```
DroppedBlockedCalls           float64
CallForwardingCalls           float64
CallWaitingCalls              float64
MonthsInService               float64
UniqueSubs                    float64
ActiveSubs                    float64
Handsets                      float64
HandsetModels                 float64
CurrentEquipmentDays          float64
AgeHH1                        float64
AgeHH2                        float64
RetentionCalls                float64
RetentionOffersAccepted       float64
ReferralsMadeBySubscriber     float64
IncomeGroup                   float64
AdjustmentsToCreditRating     float64
HandsetPrice                  float64
dtype: object
```

The code cell below finds all columns of type `object`.

[7]:
```
to_exclude = list(df.select_dtypes(include=['object']).columns)
print(to_exclude)
```

```
['ServiceArea', 'CreditRating', 'PrizmCode', 'Occupation', 'Married']
```

The code cell below removes these columns.

[8]:
```
df.drop(columns = to_exclude, axis = 1, inplace=True)
```

[9]:
```
print(df.shape)
```

```
(51047, 53)
```

1.2 Step 2: Create Labeled Examples from the Data Set for the Training Phase

Let's obtain columns from our data set to create labeled examples. The code cell below carries out the following steps:

- Gets the `Churn` column from DataFrame `df` and assigns it to the variable `y`. This is our label.
- Gets all other columns from DataFrame `df` and assigns them to the variable `X`. These are our features.

Execute the code cell below and inspect the results. You will see that we have 51047 labeled examples. Each example contains 52 features and one label (`Churn`).

[10]:
```
y = df['Churn']
X = df.drop(columns = 'Churn', axis=1)

print("Number of examples: " + str(X.shape[0]))
print("\nNumber of Features:" + str(X.shape[1]))
```

```
print(str(list(X.columns)))
```

```
Number of examples: 51047

Number of Features:52
['CustomerID', 'ChildrenInHH', 'HandsetRefurbished', 'HandsetWebCapable',
'TruckOwner', 'RVOwner', 'HomeownershipKnown', 'BuysViaMailOrder',
'RespondsToMailOffers', 'OptOutMailings', 'NonUSTravel', 'OwnsComputer',
'HasCreditCard', 'NewCellphoneUser', 'NotNewCellphoneUser', 'OwnsMotorcycle',
'MadeCallToRetentionTeam', 'MonthlyRevenue', 'MonthlyMinutes',
'TotalRecurringCharge', 'DirectorAssistedCalls', 'OverageMinutes',
'RoamingCalls', 'PercChangeMinutes', 'PercChangeRevenues', 'DroppedCalls',
'BlockedCalls', 'UnansweredCalls', 'CustomerCareCalls', 'ThreewayCalls',
'ReceivedCalls', 'OutboundCalls', 'InboundCalls', 'PeakCallsInOut',
'OffPeakCallsInOut', 'DroppedBlockedCalls', 'CallForwardingCalls',
'CallWaitingCalls', 'MonthsInService', 'UniqueSubs', 'ActiveSubs', 'Handsets',
'HandsetModels', 'CurrentEquipmentDays', 'AgeHH1', 'AgeHH2', 'RetentionCalls',
'RetentionOffersAccepted', 'ReferralsMadeBySubscriber', 'IncomeGroup',
'AdjustmentsToCreditRating', 'HandsetPrice']
```

1.3 Step 3: Create Training and Test Data Sets

Now that we have specified examples, we will need to split them into a training set that we will use to train our model, and a test set, which we will use to understand the performance of our model on new data. To do so, we will use the `train_test_split()` function from `sklearn`.

In the code cell below, use the `train_test_split()` function to create training and test sets. Consult the previous "KNN Demo" exercise to refresh your memory on how to accomplish this, or consult the online documentation for the `train_test_split()` function.

You will call `train_test_split()` function with the following arguments:

1. Feature DataFrame `X`.
2. Label DataFrame `Y`.
3. A test set that is a third of the size of the data set. More specifically, use the parameter `test_size=0.33`.
4. A seed value of 1234. More specifically, use the parameter `random_state=1234`.

The `train_test_split()` method will return four outputs (data subsets). Assign these outputs to the following variable names, using the following order: `X_train`, `X_test`, `y_train`, `y_test`.

Note that you will be able to accomplish this using one line of code.

1.4 Graded Cell

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

[11]:
```
# YOUR CODE HERE
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=.
↪33,random_state=1234)
```

1.5 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.

[12]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testSplit

try:
    p, err = testSplit(X_train, X_test, y_train, y_test, df)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

1.5.1 Get the Dimensions of the Training and Test Data Sets

[13]:
```
X_train.shape
```

[13]:
```
(34201, 52)
```

[14]:
```
X_test.shape
```

[14]:
```
(16846, 52)
```

1.5.2 Glance at the Training Data

[15]:
```
X_train.head()
```

[15]:
```
       CustomerID  ChildrenInHH  HandsetRefurbished  HandsetWebCapable  \
10351     3081630          True               False               True
33816     3269538         False               False               True
36668     3292822         False               False               True
12787     3100870          True               False               True
2635      3020642         False               False               True

       TruckOwner  RVOwner  HomeownershipKnown  BuysViaMailOrder  \
10351        True     True                True              True
33816        True    False                True              True
36668       False    False               False             False
12787       False    False                True              True
2635        False    False                True             False

       RespondsToMailOffers  OptOutMailings  ...  HandsetModels  \
10351                  True           False  ...      -0.616775
33816                  True           False  ...      -0.616775
36668                 False           False  ...      -0.616775
```

```
12787                  True           False  ...       0.487071
2635                  False           False  ...      -0.616775

       CurrentEquipmentDays     AgeHH1    AgeHH2  RetentionCalls  \
10351              1.826064  0.210998 -0.883541       -0.180167
33816             -0.167636  1.116204  0.202910       -0.180167
36668              0.104233 -1.418373 -0.883541       -0.180167
12787             -0.782294  1.025683  1.289361       -0.180167
2635               3.019920  0.120477 -0.883541       -0.180167

       RetentionOffersAccepted  ReferralsMadeBySubscriber  IncomeGroup  \
10351                  -0.1283                  -0.169283     1.489856
33816                  -0.1283                  -0.169283     0.215243
36668                  -0.1283                  -0.169283    -1.378025
12787                  -0.1283                  -0.169283     1.489856
2635                   -0.1283                  -0.169283     1.489856

       AdjustmentsToCreditRating  HandsetPrice
10351                  -0.140707     -0.368174
33816                  -0.140707     -0.368174
36668                  -0.140707     -0.368174
12787                  -0.140707     -0.864858
2635                   -0.140707     -0.368174

[5 rows x 52 columns]
```

1.6 Step 4: Fit a KNN Classification Model and Evaluate the Model

The code cell below contains a shell of a function named `train_test_knn()`. This function should train a KNN classifier on the training data, test the resulting model on the test data, and compute and return the accuracy score of the resulting predicted class labels on the test data. The accuracy score is a fraction between 0 and 1 indicating the fraction of predictions that match the true value in the test set.

Your task is to fill in the function to make it work.

Inspect the function definition `train_test_knn(X_train, X_test, y_train, y_test, k)`. The function expects the test and train datasets as well as a value for hyperparameter k - the number of neighbors. Note that by default, the Scikit-learn `KNeighborsClassifier` class uses the Euclidean distance as its distance function.

In the code cell below:

1. Use `KNeighborsClassifier()` to create a model object, and assign the result to the variable `model`. Call the method with one parameter: `n_neighbors = k`.
2. Call the `model.fit()` method to fit the model to the training data. The first argument should be `X_train` and the second argument should be `y_train`.
3. Call the `model.predict()` method with the argument `X_test` to use the fitted model to predict values for the test data. Store the outcome in the variable `class_label_predictions`.
4. Call the `accuracy_score()` function; the first argument should be `y_test` and the second argument should be `class_label_predictions`. Assign the result to variable `acc_score`.

You might find it useful to consult the "KNN Demo" exercise or the `KNeighborsClassifier` Scikit-learn online documentation for a refresher on how to accomplish these tasks.

1.7 Graded Cell

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

[20]:
```
def train_test_knn(X_train, X_test, y_train, y_test, k):
    '''
    Fit a k Nearest Neighbors classifier to the training data X_train, y_train.
    Return the accuracy of resulting predictions on the test data.
    '''

    # 1. Create the  KNeighborsClassifier model object below and assign to␣
↪variable 'model'
    # YOUR CODE HERE
    model=KNeighborsClassifier(n_neighbors=k)

    # 2. Fit the model to the training data below
    # YOUR CODE HERE
    model.fit(X_train,y_train)

    # 3. Make predictions on the test data below and assign the result to the␣
↪variable 'class_label_predictions'
    # YOUR CODE HERE
    class_label_predictions=model.predict(X_test)

    # 4. Compute the accuracy here and save the result to the variable␣
↪'acc_score'
    # YOUR CODE HERE
    acc_score=accuracy_score(y_test,class_label_predictions)


    return acc_score
```

1.7.1 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.

[21]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testFunction

try:
    p, err = testFunction(train_test_knn, df)
    print(err)
except Exception as e:
```

```
    print("Error!\n" + str(e))
```

```
Correct!
```

Train on Different Values of Hyperparameter K For a fixed data set and a chosen distance function, varying the value of the parameter `k` may have a substantial effect on the performance of the model. The optimal value of `k` depends on the data.

Running the code below will train three KNN classifiers using the `train_test_knn()` function just implemented, and using three values of `k`: 10, 100, and 1000. It will print the accuracy score of each model and save the scores to list `acc1`. This may take a few seconds.

[ ]:
```
k_values = [10, 100, 1000]

acc1 = []

for k in k_values:
    score = train_test_knn(X_train, X_test, y_train, y_test, k)
    print('k=' + str(k) + ', accuracy score: ' + str(score))
    acc1.append(float(score))
```

```
k=10, accuracy score: 0.6938739166567731
k=100, accuracy score: 0.710198266650837
```

Next we will train three more KNN classifiers for the same values of `k`, but this time using only a subset of the training data -- just the first 1500 examples.

[ ]:
```
k_values = [10, 100, 1000]

acc2 = []

for k in k_values:
    score = train_test_knn(X_train[:1500], X_test, y_train[:1500], y_test, k)
    print('k=' + str(k) + ', accuracy score: ' + str(score))
    acc2.append(float(score))
```

Let's visualize the results:

[ ]:
```
# Visualizing accuracy:
fig = plt.figure()
ax = fig.add_subplot(111)
p1 = sns.lineplot(x=k_values, y=acc1, color='b', marker='o', label = 'Full␣
 ↪training set')
p2 = sns.lineplot(x=k_values, y=acc2, color='r', marker='o', label = 'First␣
 ↪1500 of the training examples')

plt.title('Accuracy of the kNN predictions, for k$\in{10,100,1000}$')
```

```
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
ax.set_xlabel('k')
ax.set_ylabel('Accuracy on the test set')
plt.show()
```

Let's work with more than three values of k.

The code bellow trains 40 KNN classifiers with different values of k (1-40). Inspect the accuracy scores and note the optimal value for *k*.

This may take a while to compute -- we are fitting ~40 models!

[ ]:
```
acc1_40 = []
print("Accuracy scores for full training data:")
for k in range(1,41):
    score = train_test_knn(X_train, X_test, y_train, y_test, k)
    print('k=' + str(k) + ', accuracy score: ' + str(score))
    acc1_40.append(float(score))
```

The cell below accomplishes the same thing above, but using a subset of the data - the first 50 examples in the training set.

[ ]:
```
acc2_40 = []
print("\nAccuracy scores for 50 examples in training data:")
for k in range(1,41):
    score = train_test_knn(X_train[:50], X_test, y_train[:50], y_test, k)
    print('k=' + str(k) + ', accuracy score: ' + str(score))
    acc2_40.append(float(score))
```

Let's visualize the resulting accuracy values, as before:

[ ]:
```
x = [i for i in range(1,41)]

fig = plt.figure()
ax = fig.add_subplot(111)
p1 = plt.plot(x, acc1_40, 'b-', label = 'Full training set')
p2 = plt.plot(x, acc2_40, 'r-', label = 'First 50 of the training examples')

plt.title('Accuracy of the kNN predictions, for $k\in(1, 40), k\in\mathbb{N}$')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
ax.set_xlabel('k')
ax.set_ylabel('Accuracy on the test set')
plt.show()
```

You are encouraged to think about the takeaways from looking at these plots. See if you can decide what seems to be the optimal value of k. Think furthermore about what is the improvement in learning gained by having additional data.

1.7.2 The Importance of Scaling

Note that Euclidean distance is *not* scale invariant. Features with higher norms will in general dominate the neighborhood. Hence, if the features with the highest norms are also *not* strongly predictive of the target variable, these features will harm the performance of the model. It is often best to rescale the features before running KNN. The dataset that you loaded for this exercise already has this step done.

[ ]:
```

```