Break Through Tech - Machine Learning Week 5

April 21, 2023

1 Assignment 5: Model Selection for KNN

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

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
```

In this assignment, you will:

1. Load the "cell2celltrain" data set.
2. Perform a grid search to identify and fit a cross-validated optimal KNN classifier.
3. Fit the optimal KNN classifier to the training data and make predictions on the test data.
4. Display a confusion matrix for the model.
5. Plot a precision-recall curve for the model.

Note: Some of the evaluation metrics we will be using are suited for binary classification models that produce probabilities. For this reason, we will be using `predict_proba()` method to produce class label probability predictions. Recall that KNN is *not* a probabilistic method. Because of this, `predict_proba()` does not output true probabilities. What it does is the following: For n_neighbors=k, it identifies the closest k points to a given input point. It then counts up the likelihood, among these k points, of belonging to one of the classes and uses that as the class "probabilities." We will be using KNN for the sake of demonstrating how to use these evaluation metrics.

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

1.1 Part 1: Load the 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`.

[2]:
```python
# Do not remove or edit the line below:
filename = os.path.join(os.getcwd(), "data", "cell2celltrain.csv")
```

Task: Load the data and save it to DataFrame `df`.

[3]:
```python
# YOUR CODE HERE

#Solution:
df = pd.read_csv(filename, header=0)
```

1.2 Part 2: Create Training and Test Data Sets

1.2.1 Create Labeled Examples

Task: Create labeled examples from DataFrame `df`. In the code cell below, carry out the following steps:

- Get the `Churn` column from DataFrame `df` and assign it to the variable `y`. This will be our label.
- Get all other columns from DataFrame `df` and assign them to the variable `X`. These will be our features.

[4]:
```python
# YOUR CODE HERE

#Solution:

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

1.2.2 Split Labeled Examples Into Training and Test Sets

Task: In the code cell below, create training and test sets out of the labeled examples.

1. Use Scikit-learn's `train_test_split()` function to create the data sets.
2. Specify:
   - A test set that is 10 percent of the size of the data set.
   - A seed value of '1234'.

[5]:
```python
# YOUR CODE HERE

#Solution:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.10,␣
↪random_state=1234)
```

[6]:
```python
X_train.head()
```

[6]:
```
       CustomerID  ChildrenInHH  HandsetRefurbished  HandsetWebCapable  \
45106     3356966         False               False               True
38896     3310250          True               False               True
```

```
29853     3237338          False              False              True
33048     3263222          False              False              True
21061     3165118          False              False              True

       TruckOwner  RVOwner  HomeownershipKnown  BuysViaMailOrder  \
45106       False    False                True             False
38896       False    False                True              True
29853       False    False                True             False
33048       False    False                True              True
21061       False    False               False             False

       RespondsToMailOffers  OptOutMailings  ...  Occupation_Crafts  \
45106                 False           False  ...                0.0
38896                  True           False  ...                0.0
29853                 False           False  ...                0.0
33048                  True           False  ...                0.0
21061                 False           False  ...                0.0

       Occupation_Homemaker  Occupation_Other  Occupation_Professional  \
45106                   0.0               1.0                      0.0
38896                   0.0               1.0                      0.0
29853                   0.0               1.0                      0.0
33048                   0.0               0.0                      0.0
21061                   0.0               1.0                      0.0

       Occupation_Retired  Occupation_Self  Occupation_Student  Married_False  \
45106                 0.0              0.0                 0.0            0.0
38896                 0.0              0.0                 0.0            0.0
29853                 0.0              0.0                 0.0            1.0
33048                 1.0              0.0                 0.0            0.0
21061                 0.0              0.0                 0.0            0.0

       Married_True  Married_nan
45106           0.0          1.0
38896           0.0          1.0
29853           0.0          0.0
33048           1.0          0.0
21061           0.0          1.0

[5 rows x 84 columns]
```

1.3 Part 3: Perform KNN Model Selection Using `GridSearchSV()`

Our goal is to find the optimal choice of hyperparameter K.

1.3.1 Set Up a Parameter Grid

Task: Create a dictionary called `param_grid` that contains 10 possible hyperparameter values for K. The dictionary should contain the following key/value pair:

- A key called 'n_neighbors'
- A value which is a list consisting of 10 values for the hyperparameter K

For example, your dictionary would look like this: `{'n_neighbors': [1, 2, 3,..]}`

The values for hyperparameter K will be in a range that starts at 2 and ends with $\sqrt{num\_examples}$, where `num_examples` is the number of examples in our training set `X_train`. Use the NumPy np.linspace() function to generate these values, then convert each value to an `int`.

[7]:
```
#num_examples = # YOUR CODE HERE
#param_grid = # YOUR CODE HERE



### Solution:
num_examples=X_train.shape[0]
param_grid = dict(n_neighbors = [int(x) for x in np.linspace(2, np.
 ↪sqrt(num_examples), num=10)])




param_grid
```

[7]:
```
{'n_neighbors': [2, 25, 49, 72, 96, 119, 143, 167, 190, 214]}
```

1.3.2 Perform Grid Search Cross-Validation

Task: Use `GridSearchCV` to search over the different values of hyperparameter K to find the one that results in the best cross-validation (CV) score.

Complete the code in the cell below.

[8]:
```
print('Running Grid Search...')

# 1. Create a KNeighborsClassifier model object without supplying arguments.
#    Save the model object to the variable 'model'

# YOUR CODE HERE

#Solution:
model = KNeighborsClassifier()

# 2. Run a grid search with 5-fold cross-validation and assign the output to␣
 ↪the object 'grid'.
#    * Pass the model and the parameter grid to GridSearchCV()
#    * Set the number of folds to 5

# YOUR CODE HERE
```

```
#Solution:
grid = GridSearchCV(model, param_grid, cv=5)


# 3. Fit the model (use the 'grid' variable) on the training data and assign␣
↪the fitted model to the
#    variable 'grid_search'

# YOUR CODE HERE
#Solution:
grid_search = grid.fit(X_train, y_train)

print('Done')
```

```
Running Grid Search...
Done
```

Task: Retrieve the value of the hyperparameter K for which the best score was attained. Save the result to the variable `best_k`.

[9]:
```
# YOUR CODE HERE


### Solution:
best_k = grid_search.best_params_['n_neighbors']
best_k
```

[9]:
```
96
```

1.4 Part 4: Fit the Optimal KNN Model and Make Predictions

Task: Initialize a `KNeighborsClassifier` model object with the best value of hyperparameter `K` and fit the model to the training data. The model object should be named `model_best`.

[10]:
```
# 1. Create the  model object below and assign to variable 'model_best'
# YOUR CODE HERE

#Solution:
model_best = KNeighborsClassifier(best_k)

# 2. Fit the model to the training data below
# YOUR CODE HERE

# Solution:
model_best.fit(X_train, y_train)
```

[10]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=96, p=2,
                     weights='uniform')
```

Task: Test your model on the test set (`X_test`).

1. Use the `predict_proba()` method to use the fitted model `model_best` to predict class probabilities for the test set. Note that the `predict_proba()` method returns two columns, one column per class label. The first column contains the probability that an unlabeled example belongs to class `False` (Churn is "False") and the second column contains the probability that an unlabeled example belongs to class `True` (Churn is "True"). Save the values of the *second* column to a list called `probability_predictions`.

2. Use the `predict()` method to use the fitted model `model_best` to predict the class labels for the test set. Store the outcome in the variable `class_label_predictions`. Note that the `predict()` method returns the class label (True or False) per unlabeled example.

[11]:
```
# 1. Make predictions on the test data using the predict_proba() method
# YOUR CODE HERE


# Solution:
pp = model_best.predict_proba(X_test)
probability_predictions = []
for i in pp:
    probability_predictions.append(i[1])

# 2. Make predictions on the test data using the predict() method
# YOUR CODE HERE

### Solution:
class_label_predictions = model_best.predict(X_test)
```

1.5 Part 5: Evaluate the Accuracy of the Model

Task: Create a confusion matrix to evaluate your model. In the code cell below, perform the following steps:

1. Compute and print the model's accuracy score using `accuracy_score`.

2. Call the `confusion_matrix()` function with the arguments:

    1. `y_test`
    2. `class_label_predictions`
    3. The parameter `labels`. Assign the parameter a list containing two items: `True` and `False`. Note: these correspond to the two possible labels contained in `class_label_predictions`.

3. Save the resulting confusion matrix to the variable `c_m`.

4. Use the Pandas `pd.DataFrame()` function to create a DataFrame out of the confusion matrix. Supply it the following arguments:

    1. The confusion matrix `c_m`

2. The parameter columns with the value: `['Predicted: Customer Will Leave', 'Predicted: Customer Will Stay']`
3. The parameter index with the value: `['Actual: Customer Will Leave', 'Actual: Customer Will Stay']`

[12]:
```
# Compute and print the model's accuracy score
# YOUR CODE HERE
#solution
acc_score = accuracy_score(y_test, class_label_predictions)

print('Accuracy score: ' + str(acc_score))

# Create a confusion matrix
# YOUR CODE HERE

#solution
c_m = confusion_matrix(y_test, class_label_predictions, labels=[True, False])


# Create a Pandas DataFrame out of the confusion matrix for display
print('Confusion Matrix for the model: ')
# YOUR CODE HERE

#solution
pd.DataFrame(
c_m,
columns=['Predicted: Customer Will Leave', 'Predicted: Customer Will Stay'],
index=['Actual: Customer Will Leave', 'Actual: Customer Will Stay']
)
```

```
Accuracy score: 0.7134182174338883
Confusion Matrix for the model:
```

[12]:
```
                             Predicted: Customer Will Leave  \
Actual: Customer Will Leave                               0
Actual: Customer Will Stay                                0

                             Predicted: Customer Will Stay
Actual: Customer Will Leave                           1463
Actual: Customer Will Stay                            3642
```

1.6 Part 6: Plot the Precision-Recall Curve

Recall that scikit-learn defaults to a 0.5 classification threshold. Sometimes we may want a different threshold.

The precision-recall curve shows the trade-off between precision and recall for different classification thresholds. Scikit-learn's `precision_recall_curve()` function computes precision-recall pairs for different probability thresholds. For more information, consult the Scikit-learn documentation.

Let's first import the function.

[13]:
```
from sklearn.metrics import precision_recall_curve
```

Task: You will use `precision_recall_curve()` to compute precision-recall pairs. In the code cell below, call the function with the arguments `y_test` and `probability_predictions`. The function returns three outputs. Save the three items to the variables `precision`, `recall`, and `thresholds`, respectively.

[14]:
```
#precision, recall, thresholds = # YOUR CODE HERE

#Solution:
precision, recall, thresholds = precision_recall_curve(y_test,␣
↪probability_predictions)
```

The code cell below uses seaborn's `lineplot()` function to visualize the precision-recall curve. Variable `recall` will be on the x axis and `precision` will be on the y-axis.

[15]:
```
fig = plt.figure()
ax = fig.add_subplot(111)

sns.lineplot(x=recall, y=precision, marker = 'o')

plt.title("Precision-recall curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.show()
```