BTT004_week5_assignment-solution

June 7, 2024

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 continue practicing the fifth step of the machine learning life cycle and perform model selection to find the best performing KNN model for a classification problem.

You will complete the following tasks:

1. Build your DataFrame and define your ML problem
2. Create labeled examples from the data set
3. Split the data into training and test data sets
4. Perform a grid search to identify the optimal value of *K* for a KNN classifier
5. Fit the optimal KNN classifier to the training data and make predictions on the test data
6. Evaluate the accuracy of the model
7. 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 the `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. Build Your DataFrame and Define Your ML Problem

Load a Data Set and Save it as a Pandas DataFrame We will work with the "cell2celltrain" data set. This version of the data set has been preprocessed and is ready for modeling.

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

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

[3]:
```
# YOUR CODE HERE

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

Define the Label This is a binary classification problem in which we will predict customer churn. The label is the `Churn` column.

Identify Features Our features will be all of the remaining columns in the dataset.

1.2 Part 2: Create Labeled Examples from the Data Set

Task: In the code cell below, create labeled examples from DataFrame `df`.

[4]:
```
# YOUR CODE HERE

#Solution:

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

1.3 Part 3: Create Training and Test Data Sets

Task: In the code cell below, create training and test sets out of the labeled examples. Create a test set that is 10 percent of the size of the data set.

[5]:
```
# YOUR CODE HERE

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

1.4 Part 4: Perform KNN Model Selection Using `GridSearchSV()`

Our goal is to find the optimal choice of hyperparameter K. We will then train a KNN model using that value of K.

1.4.1 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]}
```

1.4.2 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. Note: This will take a few minutes to run.

[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
```

1.5 Part 5: Train 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]:
```
# YOUR CODE HERE

#Solution:
model_best = KNeighborsClassifier(best_k)
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)
```

1.6 Part 6: Evaluate the Accuracy of the Model

Task: Compute and print the model's accuracy score using `accuracy_score()`.

[12]:
```
# YOUR CODE HERE
#solution
acc_score = accuracy_score(y_test, class_label_predictions)
print('Accuracy score: ' + str(acc_score))
```

```
Accuracy score: 0.7134182174338883
```

Task: Create a confusion matrix to evaluate your model. Use the Confusion Matrix Demo as a reference.

[13]:
```
# YOUR CODE HERE

#solution
c_m = confusion_matrix(y_test, class_label_predictions, labels=[True, False])
pd.DataFrame(
c_m,
columns=['Predicted: Customer Will Leave', 'Predicted: Customer Will Stay'],
index=['Actual: Customer Will Leave', 'Actual: Customer Will Stay']
)
```

[13]:

	Predicted: Customer Will Leave	Predicted: Customer Will Stay
Actual: Customer Will Leave	0	1463
Actual: Customer Will Stay	0	3642

1.7 Part 7: Plot the Precision-Recall Curve

Recall that scikit-learn defaults to a 0.5 classification threshold. Sometimes we may want a different threshold. We can use the precision-recall curve to show 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.

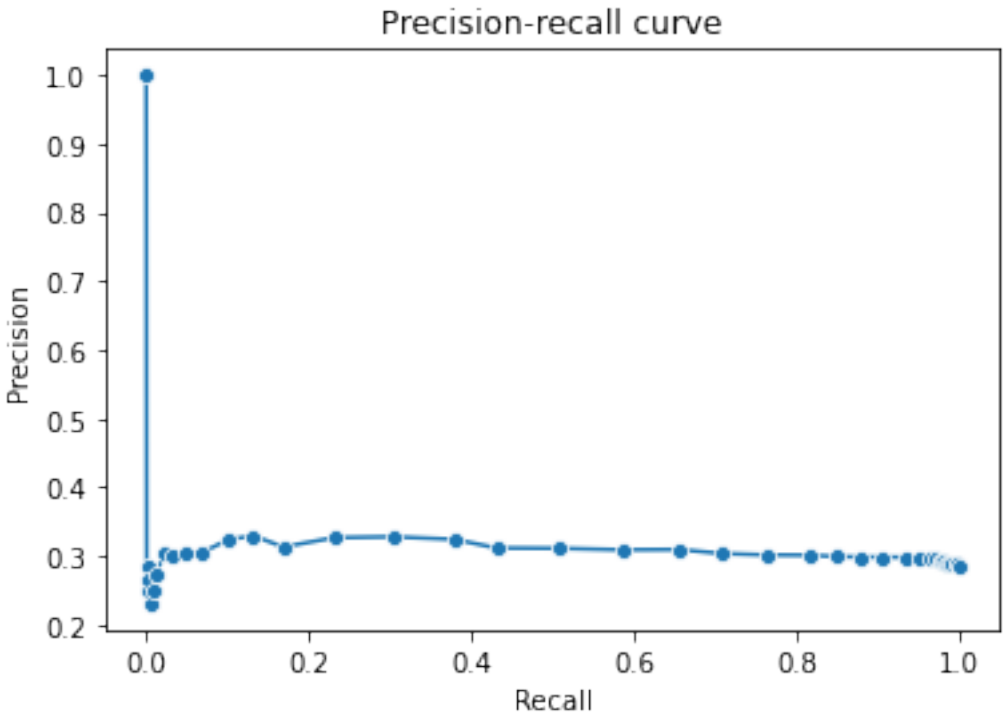
[14]:
```
from sklearn.metrics import precision_recall_curve
```

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.

[15]:
```
#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.

[16]:
```
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()
```


Precision-recall curve
Precision
Recall
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.2
0.4
0.6
0.8
1.0