KNN with scikit-learn - Lab

Introduction

In this lab, you'll learn how to use scikit-learn's implementation of a KNN classifier on the classic Titanic dataset from Kaggle!

Objectives

In this lab you will:

- Conduct a parameter search to find the optimal value for K
- Use a KNN classifier to generate predictions on a real-world dataset
- Evaluate the performance of a KNN model

Getting Started

Start by importing the dataset, stored in the `titanic.csv` file, and previewing it.

```
# Import pandas and set the standard alias
import pandas as pd

# Import the data from 'titanic.csv' and store it in a pandas DataFrame
raw_df = pd.read_csv('titanic.csv')

# Print the head of the DataFrame to ensure everything loaded correctly
raw_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0
3	4	1	1	Futrelle, Mrs. Jacques	female	35.0	1	0

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
				Heath (Lily May Peel)				
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0

Great! Next, you'll perform some preprocessing steps such as removing unnecessary columns and normalizing features.

Preprocessing the data

Preprocessing is an essential component in any data science pipeline. It's not always the most glamorous task as might be an engaging data visual or impressive neural network, but cleaning and normalizing raw datasets is very essential to produce useful and insightful datasets that form the backbone of all data powered projects. This can include changing column types, as in:

```
df['col_name'] = df['col_name'].astype('int')
```

Or extracting subsets of information, such as:

```
import re
df['street'] = df['address'].map(lambda x: re.findall('(.*)?\n', x)[0])
```

> **Note:** While outside the scope of this particular lesson, **regular expressions** (mentioned above) are powerful tools for pattern matching! See the regular expressions official documentation here.

Since you've done this before, you should be able to do this quite well yourself without much hand holding by now. In the cells below, complete the following steps:

1. Remove unnecessary columns (`'PassengerId'` , `'Name'` , `'Ticket'` , and `'Cabin'`)
2. Convert `'Sex'` to a binary encoding, where female is `0` and male is `1`
3. Detect and deal with any missing values in the dataset:
    - For `'Age'` , replace missing values with the median age for the dataset
    - For `'Embarked'` , drop the rows that contain missing values

4. One-hot encode categorical columns such as `'Embarked'`
5. Store the target column, `'Survived'` , in a separate variable and remove it from the DataFrame

```
# Drop the unnecessary columns
df = raw_df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1, inplace=False)
df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	male	22.0	1	0	7.2500	S
1	1	1	female	38.0	1	0	71.2833	C
2	1	3	female	26.0	0	0	7.9250	S
3	1	1	female	35.0	1	0	53.1000	S
4	0	3	male	35.0	0	0	8.0500	S

```
# Convert Sex to binary encoding
df['Sex'] = df['Sex'].map({'female': 0, 'male': 1})
df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	1	22.0	1	0	7.2500	S
1	1	1	0	38.0	1	0	71.2833	C
2	1	3	0	26.0	0	0	7.9250	S
3	1	1	0	35.0	1	0	53.1000	S
4	0	3	1	35.0	0	0	8.0500	S

```
# Find the number of missing values in each column
df.isna().sum()
```

```
Survived      0
Pclass        0
Sex           0
Age         177
SibSp         0
Parch         0
Fare          0
Embarked      2
dtype: int64
```

```
# Impute the missing values in 'Age'
df['Age'] = df['Age'].fillna(df.Age.median())
df.isna().sum()
```

```
Survived    0
Pclass      0
Sex         0
Age         0
SibSp       0
Parch       0
Fare        0
Embarked    2
dtype: int64
```

```
# Drop the rows missing values in the 'Embarked' column
df = df.dropna()
df.isna().sum()
```

```
Survived    0
Pclass      0
Sex         0
Age         0
SibSp       0
Parch       0
Fare        0
Embarked    0
dtype: int64
```

```
# One-hot encode the categorical columns
one_hot_df = pd.get_dummies(df)
one_hot_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked_C
0	0	3	1	22.0	1	0	7.2500	0
1	1	1	0	38.0	1	0	71.2833	1
2	1	3	0	26.0	0	0	7.9250	0
3	1	1	0	35.0	1	0	53.1000	0
4	0	3	1	35.0	0	0	8.0500	0

```
labels = one_hot_df['Survived']
one_hot_df.drop('Survived', axis=1, inplace=True)
```

Create training and test sets

Now that you've preprocessed the data, it's time to split it into training and test sets.

In the cell below:

- Import `train_test_split` from the `sklearn.model_selection` module
- Use `train_test_split()` to split the data into training and test sets, with a `test_size` of `0.25`. Set the `random_state` to 42

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(one_hot_df, labels, test_size=0.
```

Normalizing the data

The final step in your preprocessing efforts for this lab is to ***normalize*** the data. We normalize **after** splitting our data into training and test sets. This is to avoid information "leaking" from our test set into our training set (read more about data leakage here). Remember that normalization (also sometimes called ***Standardization*** or ***Scaling***) means making sure that all of your data is represented at the same scale. The most common way to do this is to convert all numerical values to z-scores.

Since KNN is a distance-based classifier, if data is in different scales, then larger scaled features have a larger impact on the distance between points.

To scale your data, use `StandardScaler` found in the `sklearn.preprocessing` module.

In the cell below:

- Import and instantiate `StandardScaler`
- Use the scaler's `.fit_transform()` method to create a scaled version of the training dataset
- Use the scaler's `.transform()` method to create a scaled version of the test dataset
- The result returned by `.fit_transform()` and `.transform()` methods will be numpy arrays, not a pandas DataFrame. Create a new pandas DataFrame out of this object

called `scaled_df`. To set the column names back to their original state, set the `columns` parameter to `one_hot_df.columns`

- Print the head of `scaled_df` to ensure everything worked correctly

```
# Import StandardScaler
from sklearn.preprocessing import StandardScaler

# Instantiate StandardScaler
scaler = StandardScaler()

# Transform the training and test sets
scaled_data_train = scaler.fit_transform(X_train)
scaled_data_test = scaler.transform(X_test)

# Convert into a DataFrame
scaled_df_train = pd.DataFrame(scaled_data_train, columns=one_hot_df.columns)
scaled_df_train.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Pclass	Sex	Age	SibSp	Parch	Fare	E
0	0.815528	-1.390655	-0.575676	-0.474917	-0.480663	-0.500108	-
1	-0.386113	-1.390655	1.550175	-0.474917	-0.480663	-0.435393	-
2	-0.386113	0.719086	-0.120137	-0.474917	-0.480663	-0.644473	-
3	-1.587755	0.719086	-0.120137	-0.474917	-0.480663	-0.115799	-
4	0.815528	-1.390655	-1.107139	0.413551	-0.480663	-0.356656	2

You may have noticed that the scaler also scaled our binary/one-hot encoded columns, too! Although it doesn't look as pretty, this has no negative effect on the model. Each 1 and 0 have been replaced with corresponding decimal values, but each binary column still only contains 2 values, meaning the overall information content of each column has not changed.

Fit a KNN model

Now that you've preprocessed the data it's time to train a KNN classifier and validate its accuracy.

In the cells below:

- Import `KNeighborsClassifier` from the `sklearn.neighbors` module
- Instantiate the classifier. For now, you can just use the default parameters
- Fit the classifier to the training data/labels
- Use the classifier to generate predictions on the test data. Store these predictions inside the variable `test_preds`

```
# Import KNeighborsClassifier
from sklearn.neighbors import KNeighborsClassifier

# Instantiate KNeighborsClassifier
clf = KNeighborsClassifier()

# Fit the classifier
clf.fit(scaled_data_train, y_train)

# Predict on the test set
test_preds = clf.predict(scaled_data_test)
```

Evaluate the model

Now, in the cells below, import all the necessary evaluation metrics from `sklearn.metrics` and complete the `print_metrics()` function so that it prints out ***Precision, Recall, Accuracy, and F1-Score*** when given a set of `labels` (the true values) and `preds` (the models predictions).

Finally, use `print_metrics()` to print the evaluation metrics for the test predictions stored in `test_preds`, and the corresponding labels in `y_test`.

```
# Import the necessary functions
from sklearn.metrics import precision_score, recall_score, accuracy_score, f1_score
```

```
# Complete the function
def print_metrics(labels, preds):
    print("Precision Score: {}".format(precision_score(labels, preds)))
    print("Recall Score: {}".format(recall_score(labels, preds)))
    print("Accuracy Score: {}".format(accuracy_score(labels, preds)))
    print("F1 Score: {}".format(f1_score(labels, preds)))

print_metrics(y_test, test_preds)
```

```
Precision Score: 0.7058823529411765
Recall Score: 0.7317073170731707
Accuracy Score: 0.7892376681614349
F1 Score: 0.718562874251497
```

> Interpret each of the metrics above, and explain what they tell you about your model's capabilities. If you had to pick one score to best describe the performance of the model, which would you choose? Explain your answer.

Write your answer below this line:

Improve model performance

While your overall model results should be better than random chance, they're probably mediocre at best given that you haven't tuned the model yet. For the remainder of this notebook, you'll focus on improving your model's performance. Remember that modeling is an ***iterative process***, and developing a baseline out of the box model such as the one above is always a good start.

First, try to find the optimal number of neighbors to use for the classifier. To do this, complete the `find_best_k()` function below to iterate over multiple values of K and find the value of K that returns the best overall performance.

The function takes in six arguments:

- `X_train`
- `y_train`

- `X_test`
- `y_test`
- `min_k` (default is 1)
- `max_k` (default is 25)

> Pseudocode Hint:

1. Create two variables, `best_k` and `best_score`
2. Iterate through every ***odd number*** between `min_k` and `max_k + 1` .
    i. For each iteration:
        a. Create a new `KNN` classifier, and set the `n_neighbors` parameter to the current value for k, as determined by the loop
        b. Fit this classifier to the training data
        c. Generate predictions for `X_test` using the fitted classifier
        d. Calculate the ***F1-score*** for these predictions
        e. Compare this F1-score to `best_score` . If better, update `best_score` and `best_k`
3. Once all iterations are complete, print the best value for k and the F1-score it achieved

```python
def find_best_k(X_train, y_train, X_test, y_test, min_k=1, max_k=25):
    best_k = 0
    best_score = 0.0
    for k in range(min_k, max_k+1, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        preds = knn.predict(X_test)
        f1 = f1_score(y_test, preds)
        if f1 > best_score:
            best_k = k
            best_score = f1

    print("Best Value for k: {}".format(best_k))
    print("F1-Score: {}".format(best_score))
```

```python
find_best_k(scaled_data_train, y_train, scaled_data_test, y_test)
# Expected Output:

# Best Value for k: 17
# F1-Score: 0.7468354430379746
```

```
Best Value for k: 17
F1-Score: 0.7468354430379746
```

If all went well, you'll notice that model performance has improved by 3 percent by finding an optimal value for k. For further tuning, you can use scikit-learn's built-in `GridSearch()` to perform a similar exhaustive check of hyperparameter combinations and fine tune model performance. For a full list of model parameters, see the sklearn documentation !

(Optional) Level Up: Iterating on the data

As an optional (but recommended!) exercise, think about the decisions you made during the preprocessing steps that could have affected the overall model performance. For instance, you were asked to replace the missing age values with the column median. Could this have affected the overall performance? How might the model have fared if you had just dropped those rows, instead of using the column median? What if you reduced the data's dimensionality by ignoring some less important columns altogether?

In the cells below, revisit your preprocessing stage and see if you can improve the overall results of the classifier by doing things differently. Consider dropping certain columns, dealing with missing values differently, or using an alternative scaling function. Then see how these different preprocessing techniques affect the performance of the model. Remember that the `find_best_k()` function handles all of the fitting; use this to iterate quickly as you try different strategies for dealing with data preprocessing!

Summary

Well done! In this lab, you worked with the classic Titanic dataset and practiced fitting and tuning KNN classification models using scikit-learn! As always, this gave you another opportunity to continue practicing your data wrangling skills and model tuning skills using Pandas and scikit-learn!

Releases

No releases published

Packages

No packages published

Contributors 4

LoreDirick Lore Dirick

mike-kane Mike Kane

sumedh10 Sumedh Panchadhar

mathymitchell

Languages

● **Jupyter Notebook** 100.0%