Eric Hansen

Dec 21, 2021 · 3 min read · Listen

Pickling The Particular Pipelines

A brief foray into two less-often-inspected SKLearn scenarios: custom features within a Pipeline and how to Pickle that Pipeline.

If you've used and read up on SKlearn, there's a good chance you've seen Pipelines mentioned as a powerful tool for ensuring streamlined workflow, avoiding data leakage, and creating a process flow suitable for serialization using Python's pickle module.

There are some incredible comprehensive guides already out there that give overviews at a variety of levels. Recently, though, I had difficulty finding a guide on how to perform two tasks:

1. My particular model involved creating some custom features (i.e. not a prepackaged transformation) based on multiple existing columns of my dataset. This proved to be non-trivial to fit it into my pipeline.

2. When the pipeline was finished, I wanted to pickle it, as I'd heard was an advantage of enclosing my process-flow in a pipeline. There was a dearth of examples of how to do this.

So, let's approach these two tasks here.

First, begin with a simple dataset, three numeric columns. Suppose the target column is related to the two predictors in a non-linear way.

```
import pandas as pd
import numpy as np

col_d = {'numeric1':[1, 2, 5, 11, 12],
```

```
df_x = df.drop('target', axis=1)
df_y = df['target']

#suppose we wanted to create non-linear columns because we had reason
to believe
# such a relationship held; and perhaps even knew there was the
modulus operation happening
# but suppose we were off by a multiple, let's say 3

def feat_eng(df):
    df['n1sq'] = np.power(df['numeric1'], 2)
    df['n2sqrt'] = np.power(df['numeric2'], 1/2)
    df['mod2'] = 3*((np.power(df['numeric1'], 2) * np.power(
df['numeric2'], 1/2)) % 2)
    return df

df_xe= feat_eng(df_x)
print(df_xe)
```

So, we have added a function to compute some custom-designed features to our dataframe at this point. As shown above, it'd be no problem to simply pass the dataframe through the feat_eng() function — in normal workflow. However, if we want to create a pipeline, this creates an obstacle. Each step in a pipeline needs to support .fit and .transform methods. Fortunately for us, we can write a class wrapper around our feat_eng() function (and even inherit from existing classes) to make it work.

```
from sklearn.base import TransformerMixin, BaseEstimator
class customFeats(TransformerMixin, BaseEstimator):
    '''object wrapper for engineered features, suitable for
pipelining'''
    def transform(self, X):
        X = feat_eng(X)
        return X

     def fit(self, X, y=None):
        return self
```

For our purposes, .transform should apply our feat_eng() function, and .fit should just

Finally(for part 1), we can wrap the entire process in our pipeline. I've included a standard scaler and used a random forest classifier for this example; of course this can be replaced with other transforms and classifiers as necessary.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

pipe = Pipeline(steps=[
    ('customFeats', customFeats()),
    ('scaler', StandardScaler()),
    ('RFclf', RandomForestClassifier( criterion='gini',
n_estimators=10))
    ])

#

col_d_test = {'numeric1':[1, 3, 5, 11, 12],
        'numeric2': [144, 36, 225, 121, 81],
        'target':[0, 0, 1, 1, 0 ]}
#note, target = ((n1**2*n2**(1/2))%2)

df_test = pd.DataFrame(col_d_test)
df_x_test = df.drop('target', axis=1)
df_y_test = df['target']

pipe.fit(df_x, df_y)
print('Training set accuracy: {:.4}%'.format(100*pipe.score(df_x,
df_y)))
print('Test set accuracy: {:.4}%'.format(100*pipe.score(df_x_test,
df_y_test)))
```

Great! Onto part 2 — putting the newly created pipeline into python's pickle function. This will likely be straighforward, using the 'with' treatment to ease the file I/O handling.

```
import pickle
with open('clf_model_1.pkl', 'wb') as f:
    pickle.dump(pipe,f)
```

```
if unpickled_pipe.score(df_x_test, df_y_test) == pipe.score(df_x_test,
df_y_test):
    print('Pickled and unpickled models produce same scores!')
```

This works great for me, but some users do run into some difficulty with the pipe step and their own custom feature functions. One solution involves saving the feat_eng() function as a .py file, then importing it and saving it using joblib, but the details of that are beyond the scope of this blog entry.

In closing, pipeline functionality is tremendously useful and there are many great guides out there for it. I hope that this provided you with some useful examples that were otherwise hard to find!

Following is a github gist with the full code from this post: