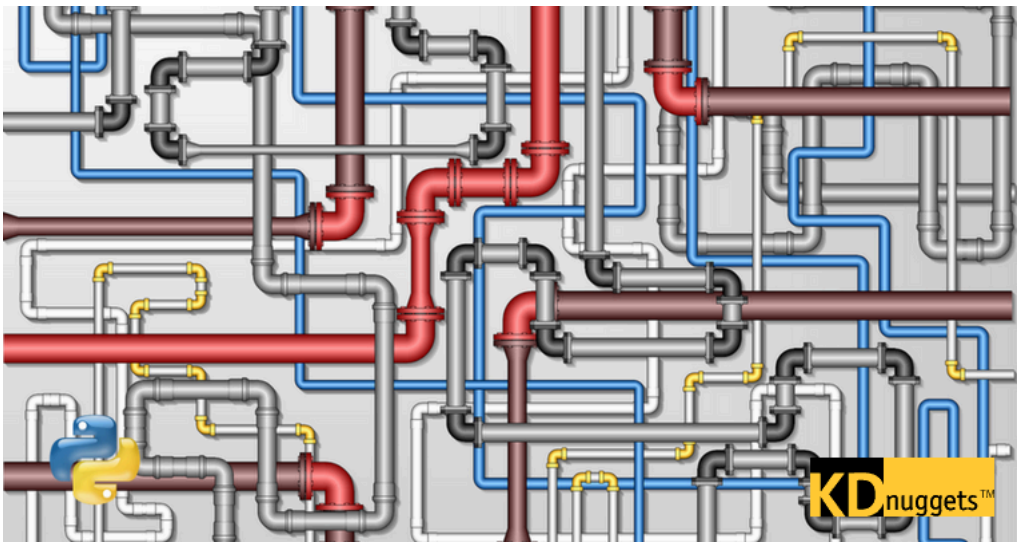
Managing Machine Learning Workflows with Scikit-learn Pipelines Part 1: A Gentle Introduction

Scikit-learn's Pipeline class is designed as a manageable way to apply a series of data transformations followed by the application of an estimator.

By **Matthew Mayo**, KDnuggets Managing Editor on December 7, 2017 in **Data Preprocessing**, **Pipeline**, **Python**, **scikit-learn**, **Workflow**

comments



Are you familiar with Scikit-learn Pipelines?

They are an extremely simple yet very useful tool for managing machine learning workflows.

A typical machine learning task generally involves data preparation to varying degrees. We won't get into the wide array of activities which make up data preparation here, but there are many. Such tasks are known for taking up a large proportion of time spent on any given machine learning task.

After a dataset is cleaned up from a potential initial state of massive disarray, however, there are still several less-intensive yet no less-important transformative data preprocessing steps such as feature extraction, feature scaling, and dimensionality reduction, to name just a few.

Search KDnuggets...

Latest Posts
Has Europe Gone Too Far? The Del Dance of Regulation and Innovatio
Get an Additional 30% off Courses Offer Ends in 3 Days!
5 LLM Tools I Can't Live Without
How To Improve the Performance RAG Model
7 Steps to Mastering Coding for Da Science
How to Create Interactive Visualiza in R

Top Posts

Blog
Top Posts
About
Topics

ultimately finish off with an estimator of some sort. This is where Scikit-learn Pipelines can be helpful.

Scikit-learn's Pipeline class is designed as a manageable way to apply a series of data transformations followed by the application of an estimator. In fact, that's really all it is:

> Pipeline of transforms with a final estimator.

That's it. Ultimately, this simple tool is useful for:

- Convenience in creating a coherent and easy-to-understand workflow
- Enforcing workflow implementation and the desired order of step applications
- Reproducibility
- Value in persistence of entire pipeline objects (goes to reproducibility and convenience)

So let's have a quick look at Pipelines. Specifically, here is what we will do.

Build 3 pipelines, each with a different estimator (classification algorithm), using default hyperparameters:

- Logisitic Regression
- Support Vector Machine
- Decision Tree

To demonstrate pipeline **transforms**, will perform:

- feature scaling
- dimensionality reduction, using PCA to project data onto 2 dimensional space

We will then end with fitting to our final **estimators**.

Afterward, and almost completely unrelated, in order to make this a little more like a full-fledged workflow (it still isn't, but closer), we will:

- Followup with scoring test data
- Compare pipeline model accuracies
- Identify the "best" model, meaning that which has the highest accuracy on our test data
- Persist (save to file) the entire pipeline of the "best" model

Granted, given that we will use default hyperparameters, this likely won't result in the most accurate possible models, but it will provide a sense of how to use simple pipelines. We will come back to the question of more complex modeling, hyperparameter tuning, and model evaluation afterward.

Oh, and for additional simplicity, we are using the iris dataset. The code is well-commented, and should be easy to follow.

```
1  from sklearn.datasets import load_iris
2  from sklearn.model_selection import train_test_split
3  from sklearn.preprocessing import StandardScaler
4  from sklearn.decomposition import PCA
```

7 Steps to Mastering Coding for Da Science
How Natural Language Processing Unstructured Data is Improving Healthcare Outcomes
Partial Functions in Python: A Guid Developers
10 GitHub Repositories for Deep Learning Enthusiasts
5 LLM Tools I Can't Live Without
Free Courses That Are Actually Fre Google Cloud Edition
Free Courses That Are Actually Fre Programming Edition
Ollama Tutorial: Running LLMs Loc Made Super Simple
How to Write Basic SQL Queries in BigQuery

```
from sklearn import tree

# Load and split the data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

# Construct some pipelines
pipe_lr = Pipeline([('scl', StandardScaler()),
                    ('pca', PCA(n_components=2)),
                    ('clf', LogisticRegression(random_state=42))])

pipe_svm = Pipeline([('scl', StandardScaler()),
                    ('pca', PCA(n_components=2)),
                    ('clf', svm.SVC(random_state=42))])

pipe_dt = Pipeline([('scl', StandardScaler()),
                    ('pca', PCA(n_components=2)),
                    ('clf', tree.DecisionTreeClassifier(random_state=42))])

# List of pipelines for ease of iteration
pipelines = [pipe_lr, pipe_svm, pipe_dt]

# Dictionary of pipelines and classifier types for ease of reference
pipe_dict = {0: 'Logistic Regression', 1: 'Support Vector Machine', 2: 'Decision Tree'}

# Fit the pipelines
for pipe in pipelines:
        pipe.fit(X_train, y_train)

# Compare accuracies
for idx, val in enumerate(pipelines):
        print('%s pipeline test accuracy: %.3f' % (pipe_dict[idx], val.score(X_test, y_test)))

# Identify the most accurate model on test data
best_acc = 0.0
best_clf = 0
best_pipe = ''
for idx, val in enumerate(pipelines):
        if val.score(X_test, y_test) > best_acc:
                best_acc = val.score(X_test, y_test)
                best_pipe = val
                best_clf = idx
print('Classifier with best accuracy: %s' % pipe_dict[best_clf])

# Save pipeline to file
joblib.dump(best_pipe, 'best_pipeline.pkl', compress=1)
print('Saved %s pipeline to file' % pipe_dict[best_clf])
```

pipelines-1.py hosted with ❤ by GitHub view raw

Let's run our script and see what happens.

```
$ python3 pipelines.py
```

```
Logistic Regression pipeline test accuracy: 0.933
Support Vector Machine pipeline test accuracy: 0.900
Decision Tree pipeline test accuracy: 0.867
Classifier with best accuracy: Logistic Regression
Saved Logistic Regression pipeline to file
```

So there you have it; a simple implementation of Scikit-learn pipelines. In this particular case, our logistic regression-based pipeline with default parameters scored the highest

we did want to test out a series of different hyperparameters? Can we use grid search? Can we incorporate automated methods for tuning these hyperparameters? Can AutoML fit in to this picture somewhere? What about using cross-validation?

Over the next couple of posts we will take a look at these additional issues, and see how these simple pieces fit together to make pipelines much more powerful than they may first appear to be given our initial example.

Related:

- 7 Steps to Mastering Data Preparation with Python
- Machine Learning Workflows in Python from Scratch Part 1: Data Preparation
- Machine Learning Workflows in Python from Scratch Part 2: k-means Clustering

---

More On This Topic

- A Gentle Introduction to Natural Language Processing
- A Gentle Introduction to Support Vector Machines
- A New Way of Managing Deep Learning Datasets
- Managing Your Reusable Python Code as a Data Scientist
- 4 Steps for Managing a Data Science Project
- Managing Model Drift in Production with MLOps