Scikit-learn's Transformers

- v0.20 and beyond -

Tom Dupré la Tour - PyParis 14/11/2018
Scikit-learn's Transformers
from sklearn.preprocessing import StandardScaler

model = StandardScaler()
X_train_2 = model.fit(X_train).transform(X_train)
X_test_2 = model.transform(X_test)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier

model = make_pipeline(StandardScaler(),
                      SGDClassifier(loss='log'))

y_pred = model.fit(X_train, y_train).predict(X_test)
Pipeline

```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier

model = make_pipeline(StandardScaler(),
                      SGDClassifier(loss='log'))

y_pred = model.fit(X_train, y_train).predict(X_test)
```

Advantages

- Clear overview of the pipeline
- Correct cross-validation
- Easy parameter grid-search
- Caching intermediate results
Transformers before v0.20

- Dimensionality reduction: PCA, KernelPCA, FastICA, NMF, etc.
- Scalers: StandardScaler, MaxAbsScaler, etc.
- Encoders: OneHotEncoder, LabelEncoder, MultiLabelBinarizer
- Expansions: PolynomialFeatures
- Imputation: Imputer
- Custom 1D transforms: FunctionTransformer
- Quantiles: QuantileTransformer (v0.19)
- and also: Binarizer, KernelCenterer, RBFSampler,...
New in v0.20
v0.20: Easier data science pipeline

Many new Transformers

- ColumnTransformer (new)
- PowerTransformer (new)
- KBinsDiscretizer (new)
- MissingIndicator (new)
- SimpleImputer (new)
- OrdinalEncoder (new)
- TransformedTargetRegressor (new)

Transformer with significant improvements

- OneHotEncoder handles categorical features.
- MaxAbsScaler, MinMaxScaler, RobustScaler, StandardScaler, PowerTransformer, and QuantileTransformer, handles missing values (NaN).
v0.20: Easier data science pipeline

- SimpleImputer (new) handles categorical features.
- MissingIndicator (new)
v0.20: Easier data science pipeline

- SimpleImputer (new) handles categorical features.
- MissingIndicator (new)
- OneHotEncoder handles categorical features.
- OrdinalEncoder (new)
v0.20: Easier data science pipeline

- SimpleImputer (new) handles categorical features.
- MissingIndicator (new)
- OneHotEncoder handles categorical features.
- OrdinalEncoder (new)
- MaxAbsScaler, MinMaxScaler, RobustScaler, StandardScaler, PowerTransformer, and QuantileTransformer, handles missing values (NaN).
ColumnTransformer (new)

```python
don from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

umeric = make_pipeline(
    SimpleImputer(strategy='median'),
    StandardScaler())

categorical = make_pipeline(
    # new: 'constant' strategy, handles categorical features
    SimpleImputer(strategy='constant', fill_value='missing'),
    # new: handles categorical features
    OneHotEncoder())

preprocessing = make_column_transformer(
    [(['age', 'fare'], numeric),
     ([ 'sex', 'pclass'], categorical)],
    remainder='drop')

model = make_pipeline(preprocessing,
                       LogisticRegression())
```
PowerTransformer (new)
KBinsDiscretizer (new)
TransformedTargetRegressor (new)
TransformedTargetRegressor

```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                    func=np.log,
                                    inverse_func=np.exp)

y_pred = model.fit(X_train, y_train).predict(X_test)
```
Glossary of Common Terms and API Elements (new)

Joblib backend system (new)

- New pluggable backend system for Joblib
- New default backend for single host multiprocessing (loky)
  - Does not break third-party threading runtimes
- Ability to delegate to dask/distributed for cluster computing
Nearest Neighbors
Nearest Neighbors Classifier

Input data

KNeighborsClassifier(k=1)

KNeighborsClassifier(k=2)

KNeighborsClassifier(k=4)

KNeighborsClassifier(k=8)

KNeighborsClassifier(k=16)
Nearest Neighbors in scikit-learn

Used in:

- KNeighborsClassifier, RadiusNeighborsClassifier
- KNeighborsRegressor, RadiusNeighborsRegressor, LocalOutlierFactor
- TSNE, Isomap, SpectralEmbedding
- DBSCAN, SpectralClustering
Nearest Neighbors

Computed with brute force, KDTree, or BallTree, ...
Nearest Neighbors

Computed with brute force, **KDT**ree, or **BallTree**, ...

... or with approximated methods (random projections)

- annoy (by Spotify)
- faiss (by Facebook research)
- nmslib
- ...
Nearest Neighbors benchmark

https://github.com/erikbern/ann-benchmarks
Nearest Neighbors

- scikit-learn API -
Trees and wrapping estimator

- **KDTTree** and **BallTree**:  
  - Not proper scikit-learn estimators  
  - `query`, `query_radius`, which return `(indices, distances)`
Trees and wrapping estimator

- **KDTREE and BallTree:**
  - Not proper scikit-learn estimators
  - `query`, `query_radius`, which return `(indices, distances)`

- **NearestNeighbors:**
  - scikit-learn estimator, but without `transform` or `predict`
  - `kneighbors`, `radius_neighbors`, which return `(distances, indices)`
Nearest Neighbors call

- **KernelDensity, NearestNeighbors:**
  - Create an instance of BallTree or KDTree
Nearest Neighbors call

- **KernelDensity**, **NearestNeighbors**:
  - Create an instance of BallTree or KDTree

- **KNeighborsClassifier**, **KNeighborsRegressor**, **RadiusNeighborsClassifier**, **RadiusNeighborsRegressor**, **LocalOutlierFactor**:
  - Inherit `fit` and `kneighbors` (weird) from **NearestNeighbors**
Nearest Neighbors call

- **KernelDensity**, **NearestNeighbors**:  
  - Create an instance of BallTree or KDTree

- **KNeighborsClassifier**, **KNeighborsRegressor**,  
  **RadiusNeighborsClassifier**, **RadiusNeighborsRegressor**,  
  **LocalOutlierFactor**:  
  - Inherit fit and kneighbors (weird) from NearestNeighbors

- **TSNE**, **DBSCAN**, **Isomap**, **LocallyLinearEmbedding**:  
  - Create an instance of NearestNeighbors
Nearest Neighbors call

- **KernelDensity, NearestNeighbors:**
  - Create an instance of BallTree or KDTree

- **KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor, LocalOutlierFactor:**
  - Inherit `fit` and `kneighbors` (weird) from NearestNeighbors

- **TSNE, DBSCAN, Isomap, LocallyLinearEmbedding:**
  - Create an instance of NearestNeighbors

- **SpectralClustering, SpectralEmbedding:**
  - Call `kneighbors_graph`, which creates an instance of NearestNeighbors
Copy of NearestNeighbors parameters in each class

```python
params = [algorithm, leaf_size, metric, p, metric_params, n_jobs]

# sklearn.neighbors
NearestNeighbors(n_neighbors, radius, *params)
KNeighborsClassifier(n_neighbors, *params)
KNeighborsRegressor(n_neighbors, *params)
RadiusNeighborsClassifier(radius, *params)
RadiusNeighborsRegressor(radius, *params)
LocalOutlierFactor(n_neighbors, *params)

# sklearn.manifold
TSNE(metric)
Isomap(n_neighbors, neighbors_algorithm, n_jobs)
LocallyLinearEmbedding(n_neighbors, neighbors_algorithm, n_jobs)
SpectralEmbedding(n_neighbors, n_jobs)

# sklearn.cluster
SpectralClustering(n_neighbors, n_jobs)
DBSCAN(eps, *params)
```
Different handling of precomputed neighbors in X

- Handle precomputed distance matrices:
  - TSNE, DBSCAN, SpectralEmbedding, SpectralClustering,
  - LocalOutlierFactor, NearestNeighbors
  - KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor
  - (not Isomap)
Different handling of precomputed neighbors in X

- Handle precomputed distance matrices:
  - TSNE, DBSCAN, SpectralEmbedding, SpectralClustering,
  - LocalOutlierFactor, NearestNeighbors
  - KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor
  - (not Isomap)

- Handle precomputed sparse neighbors graphs:
  - DBSCAN, SpectralClustering
Different handling of precomputed neighbors in $X$

- Handle precomputed distance matrices:
  - TSNE, DBSCAN, SpectralEmbedding, SpectralClustering,
  - LocalOutlierFactor, NearestNeighbors
  - KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor
  - (not Isomap)

- Handle precomputed sparse neighbors graphs:
  - DBSCAN, SpectralClustering

- Handle objects inheriting NearestNeighbors:
Different handling of precomputed neighbors in X

- Handle precomputed distance matrices:
  - TSNE, DBSCAN, SpectralEmbedding, SpectralClustering,
  - LocalOutlierFactor, NearestNeighbors
  - KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor
  - (not Isomap)

- Handle precomputed sparse neighbors graphs:
  - DBSCAN, SpectralClustering

- Handle objects inheriting NearestNeighbors:
  - LocalOutlierFactor, NearestNeighbors

- Handle objects inheriting BallTree/KDTree:
  - LocalOutlierFactor, NearestNeighbors
  - KNeighborsClassifier, KNeighborsRegressor, RadiusNeighborsClassifier, RadiusNeighborsRegressor
Challenges

Consistent API, avoid copying all parameters,
Changing the API? difficult without breaking code
Use approximated nearest neighbors from other libraries
Proposed solution

Precompute sparse graphs in a Transformer

[#10482]
Precomputed sparse nearest neighbors graph

Steps:

1. Make all classes accept precomputed sparse neighbors graph
Precomputed sparse nearest neighbors graph

Steps:

1. Make all classes accept precomputed sparse neighbors graph

2. Pipeline: Add KNeighborsTransformer and RadiusNeighborsTransformer

```python
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsTransformer
from sklearn.manifold import TSNE

graph = KNeighborsTransformer(n_neighbors=n_neighbors,
                               mode='distance', metric=metric)
tsne = TSNE(metric='precomputed', method="barnes_hut")

model_1 = make_pipeline(graph, tsne)
model_2 = TSNE(metric=metric, method="barnes_hut")
```
Precomputed sparse nearest neighbors graph

Improvements:

1. All parameters are accessible in the transformer
Precomputed sparse nearest neighbors graph

Improvements:

1. All parameters are accessible in the transformer
2. Caching properties of the pipeline (memory="path/to/cache")
Precomputed sparse nearest neighbors graph

Improvements:

1. All parameters are accessible in the transformer
2. Caching properties of the pipeline (memory="path/to/cache")
3. Allow custom nearest neighbors estimators

# Example:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSNE with AnnoyTransformer:</td>
<td></td>
<td>46.222 sec</td>
</tr>
<tr>
<td>TSNE with KNeighborsTransformer:</td>
<td></td>
<td>79.842 sec</td>
</tr>
<tr>
<td>TSNE with internal NearestNeighbors:</td>
<td></td>
<td>79.984 sec</td>
</tr>
</tbody>
</table>
Thank you for your attention!

tomdlt.github.io/decks/2018_pyparis

@tomdl10