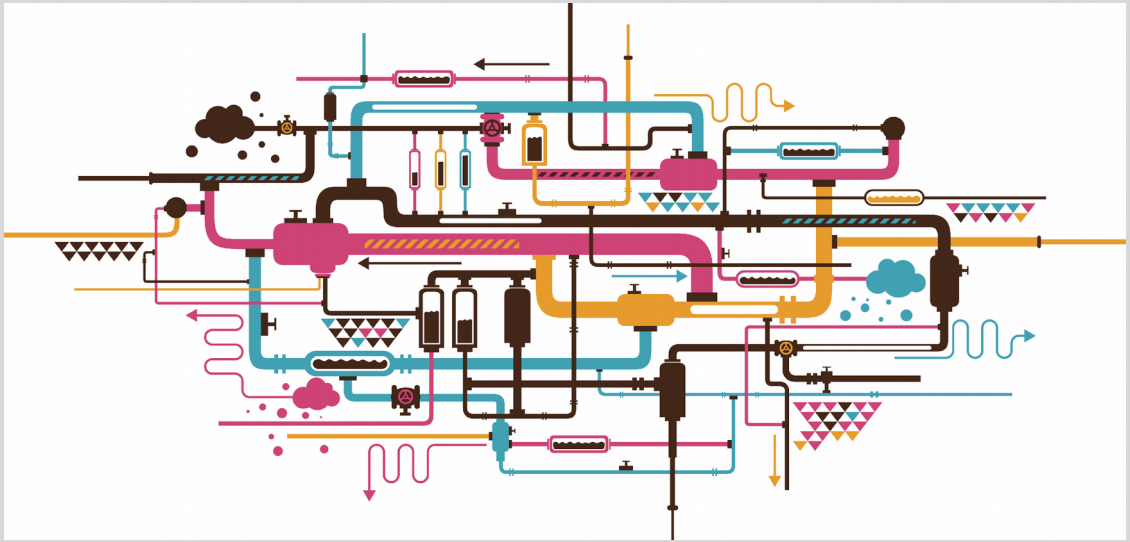


DigitalHouse > Coding School

DATA SCIENCE

UNIDAD 3
MÓDULO 5

Pipelines

Octubre 2017

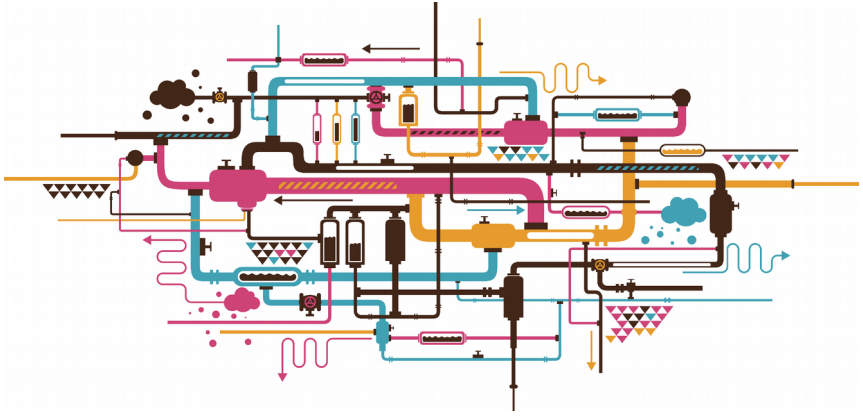
PIPELINES

OBJETIVOS DE LA CLASE

1. **Crear pipelines para limpiar y manipular datos**

2. **Crear un transformador custom**

3. **Usar pipelines en combinación con clasificación, FeatureUnion y GridSearch**



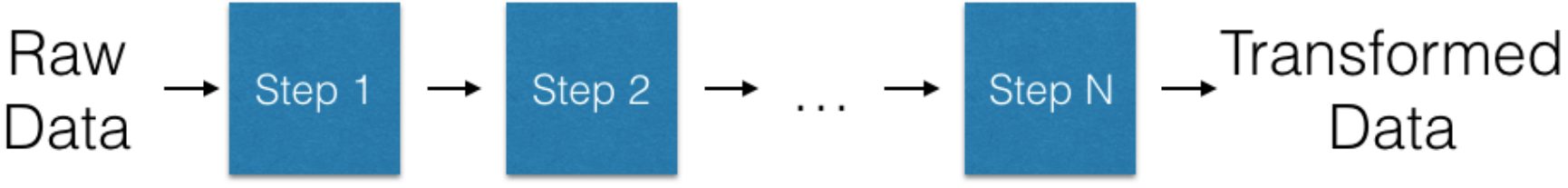
PIPELINES

Un pipeline es una serie de pasos automatizados para transformar nuestros datos con el objetivo de asegurar su validez y consistencia.

Cada paso se “alimenta” del paso previo.

Al ser re-utilizables, los pipelines permiten ejecutar exactamente las mismas transformaciones sobre distintos datasets, asegurando consistencia en la operación

Al agrupar operaciones, también proveen un mayor nivel de abstracción



PIPELINES en SCIKIT-LEARN

Clase Pipeline del módulo sklearn (class sklearn.pipeline.Pipeline(steps)).

Steps es una lista de tuplas (key, value):
- Key: Nombre dado al paso
- Value: el transformador usado

Todos los pasos menos el último deben ser transformadores:
- Implementar el método *Transform*
- Implementar el método *Fit*

El último paso solo necesita implementar fit (puede ser transformador o clasificador)

http://scikit-learn.org/stable/modules/pipeline.html

DEMO I

Evergreen Stumbleupon Kaggle Competition

Kaggle(www.kaggle.com): red social para data scientists

StumbleUpon (www.StumbleUpon.com): “curador” de contenidos

2 Tipos de Contenidos:

- Efímeros: Pierden relevancia con el paso del tiempo. Ej: Noticias, recetas, etc.
- Perennes (Evergreen): Mantienen relevancia y pueden ser recomendados por más tiempo.

Competencia:

- Realizar esta distinción (clasificación) sin participación humana.

https://www.kaggle.com/c/stumbleupon/overview

- 7395 ejemplos
- 27 campos
- Nos vamos a concentrar en el título y cuerpo de las páginas, dentro del campo boilerplate

"url"	"urlid"	"boilerplate"	"alchemy_categor y"	"alchemy_categor y_score"	"avglinksize"	"commonlin k"
"http:// www.b om/ne 2010-1 predic hologr air-bre batteri by-201						

PIPELINES

Algunos atributos

PIPELINES: MAKE_PIPELINE()

Otra manera de crear un pipeline es utilizando el comando make_pipeline.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe1 = make_pipeline(StandardScaler(), LogisticRegression())

pipe2 = Pipeline(steps=[('standardscaler',StandardScaler()),
                        ('logistic_regr',LogisticRegression())
                       ])
```

Ambos pipelines creados son idénticos.

```
pipe1
```

```
Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))])
```

```
pipe2
```

```
Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))])
```

Para make_pipeline no hace falta dar un nombre a cada paso, lo toma automáticamente del transformador

PIPELINES: ATRIBUTOS

Los estimadores del pipeline se guardan como una lista en el atributo *steps*:

```
>>> pipe.steps[0]
('reduce_dim', PCA(copy=True, iterated_power='auto',n_components=None,
random_state=None, svd_solver='auto', tol=0.0, whiten=False))
```

… y como un dict en *named_steps* :

```
>>> pipe.named_steps['reduce_dim']
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
svd_solver='auto', tol=0.0, whiten=False)
```

Los parameters de un estimador se pueden acceder usando *<estimador>__<parámetro>* :

```
>>> pipe.set_params(clf__C=10)

Pipeline(steps=[('reduce_dim', PCA(copy=True,iterated_power='auto',
n_components=None, random_state=None,svd_solver='auto', tol=0.0,whiten=False)),
('clf', SVC(C=10, cache_size=200,class_weight=None,coef0=0.0,
decision_function_shape=None, degree=3,gamma='auto',kernel='rbf', max_iter=-1,
probability=False,random_state=None,shrinking=True, tol=0.001, verbose=False))])
```

Podemos crear un pipeline con dos pasos:

```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.decomposition import PCA
estimators = [('reduce_dim', PCA()), ('clf', SVC())]
pipe = Pipeline(estimators)
pipe
```

Y luego cambiar los modelos que ejecutan esos pasos:

```
from sklearn.linear_model import LogisticRegression
param_grid = dict(reduce_dim=[None, PCA(5), PCA(10)],
                  clf=[SVC(), LogisticRegression()],
                  clf__C=[0.1, 10, 100])
grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

PREPROCESAMIENTO

Esta práctica nos permitirá conocer el módulo de pre-procesamiento.

El mismo viene repleto de clases muy útiles para dicha operación.

La finalidad es re-utilizar al máximo las funciones existentes

http://scikit-learn.org/stable/modules/preprocessing.html

Práctica:

1. Juntarse de a pares
2. Seleccionar una función por grupo
3. Leer la documentación de la misma
4. Explicar el funcionamiento al resto de la clase.

Data Manipulators

- Binarizer
- KernelCenterer
- MaxAbsScaler
- MinMaxScaler
- Normalizer
- OneHotEncoder
- PolynomialFeatures
- RobustScaler
- StandardScaler

Data Imputation

- Imputer

Function Transformer

- FunctionTransformer

Label Manipulators

- LabelBinarizer
- LabelEncoder
- MultiLabelBinarizer

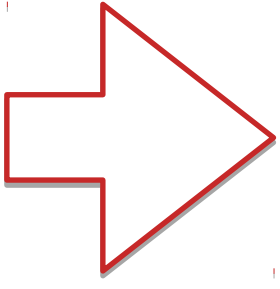
Más allá de la riqueza del módulo de preprocesamiento, podemos encontrar casos donde no sean suficiente y nos sea conveniente crear transformadores custom. Tenemos dos maneras de hacerlo:

1. Extender la BaseClass en Scikit-Learn.

En este ejemplo creamos un transformador muy simple que devuelve la entrada multiplicada por un factor X:

```python
from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np

class FeatureMultiplier(BaseEstimator, TransformerMixin):
    def __init__(self, factor):
        self.factor = factor

    def transform(self, X, *_):
        return X * self.factor

    def fit(self, *_):
        return self

fm = FeatureMultiplier(2)

test = np.diag((1,2,3,4))
print test

fm.transform(test)
```

```
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

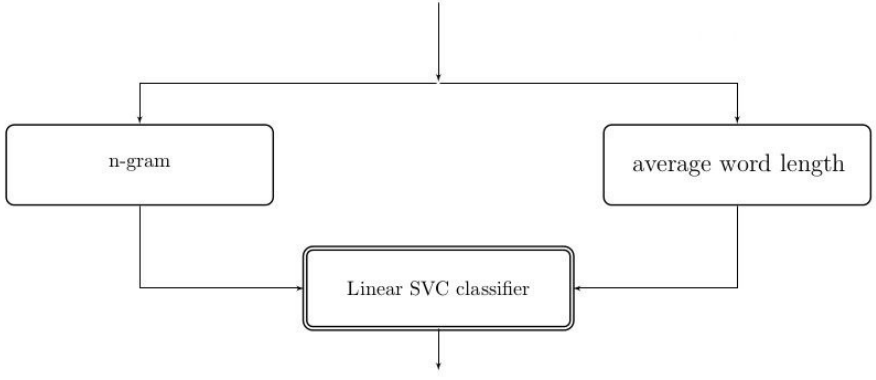


```
[[[2, 0, 0, 0],
  [0, 4, 0, 0],
  [0, 0, 6, 0],
  [0, 0, 0, 8]]
```

FEATURE UNION

Hay casos en los que nos va a interesar juntar features transformadas aplicando distintos métodos, y luego correr pasos siguientes.
Para ello, podemos usar FeatureUnion, que combina varios transformadores, en un nuevo transformador armado con la combinación de los outputs de cada transformador incluido.
Cada transformador es aplicado de manera independiente y en paralelo, y los vectores de salida son combinados.

Ejemplo: un Pipeline con una Unión entre una matriz de palabras con CountVectorizer y el tamaño promedio de palabra (CustomTransformer), y un modelo que clasifique en base a ambas




```
from sklearn.pipeline import Pipeline, FeatureUnion
pipeline = Pipeline([('feats', FeatureUnion([
    ('ngram', ngram_count_pipeline),
    ('ave', AverageWordLengthExtractor())
  ])),
  ('clf', LinearSVC())
])
```

HERRAMIENTAS GRÁFICAS

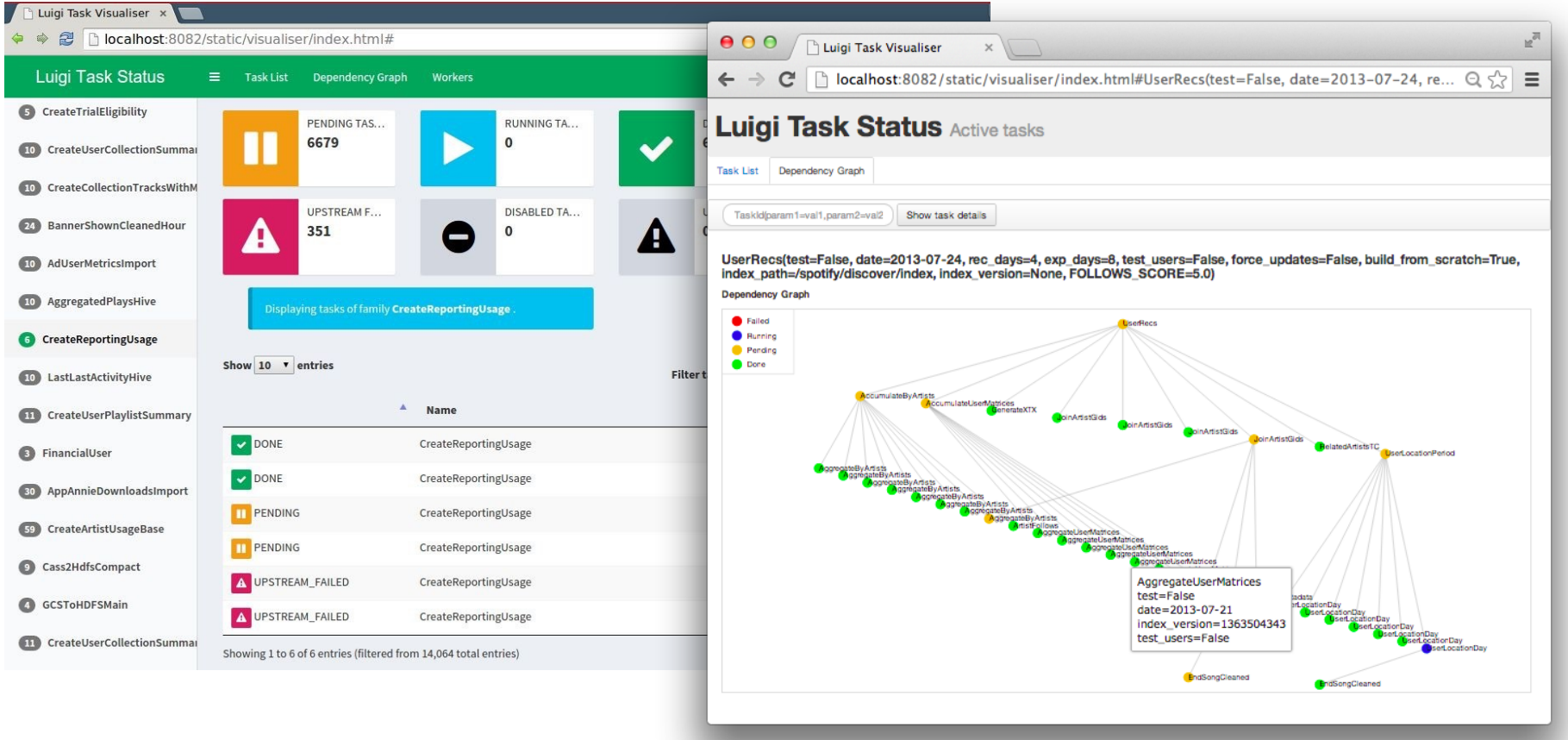
Hemos podido observar la potencia de los Pipelines, más en combinación con GridSearch y FeatureUnion, además de las funciones de pre-procesamiento existentes y la posibilidad de extenderlos.

Sin embargo, en un ambiente complejo será conveniente utilizar una herramienta gráfica.

Dicha herramienta deberá soportar:

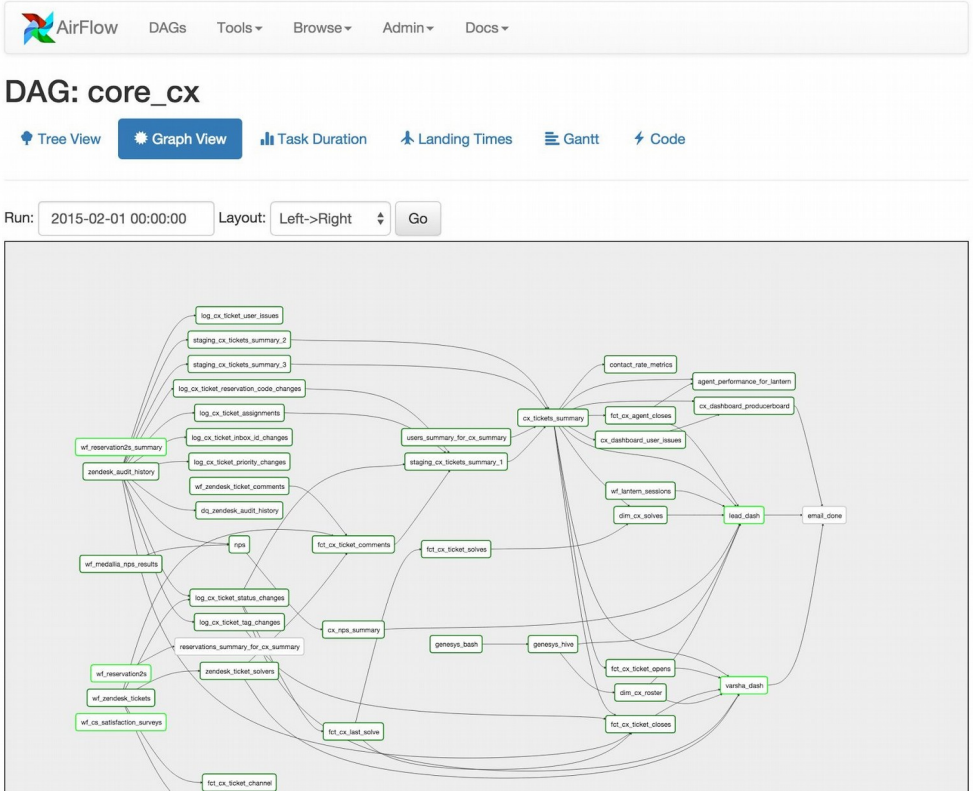
- Manejo de Ambientes (Dev, Test, Prod, Sandbox)
- Governance
- Seguridad
- Dependencias
- Alarmas
- Conectividad
- Priorización de procesamiento
- Estadísticas
- Errores
- Excepciones
- Reporting
- Etc.

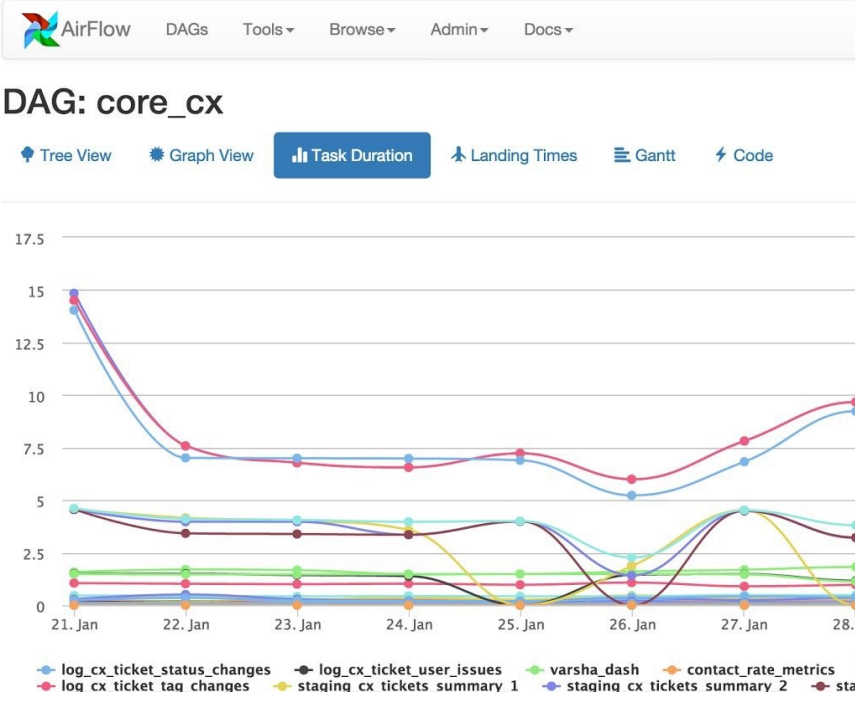
Un ejemplo de estas herramientas es Luigi (creada por Spotify): https://github.com/spotify/luigi



HERRAMIENTAS GRÁFICAS: AirFlow

O AirFlow (creada por AirBnb) https://github.com/apache/incubator-airflow





CONCLUSIONES

Pipelines

Los conceptos vistos hoy nos permiten:

- Generar soluciones más robustas
- Re-utilizar código
- Optimizar nuestra elección de modelos y parámetros
- Armar workflows de pro-cesamiento complejos
- Crear transformadores custom y adosarlos en los workflows