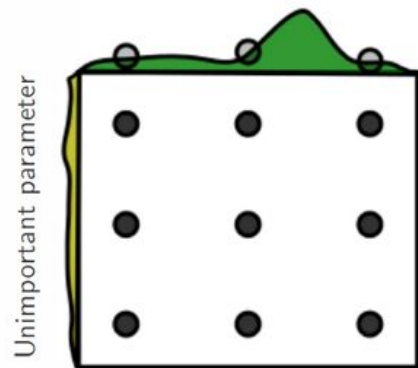
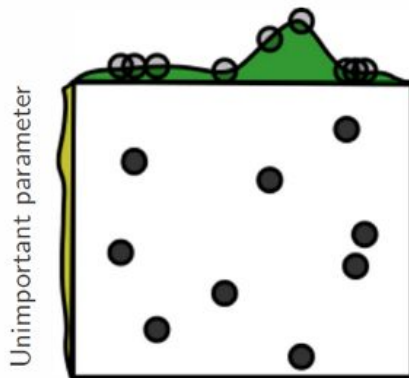


Grid Layout



Important parameter

Random Layout



Important parameter

DigitalHouse >
Coding School

DATA SCIENCE

MÓDULO 4

Grid Search - Tuneo de
Hiperparámetros-Pipelines

Evaluación Avanzada de Modelos



1

Conocer a GridSearch: un método para experimentar sobre hiper-parámetros

2

Comparar GridSearch y RandomSearch, dos aproximaciones a la búsqueda de hiperparámetros óptimos

3

Implementar GridSearch de sklearn para autoajustar un modelo

4

Introducir pipelines

Regresión lineal

Calcula los parámetros ideales para minimizar la suma de los errores al cuadrado.

- **Preprocesamiento**: Los coeficientes dan cuenta perfectamente de cualquier unidad en la que estén expresados los datos. Por lo tanto, estandarizar o reescalar no cambia los resultados
- **Variables categóricas**: Admite. Hay que crear $k-1$ variables dummy por cada categórica para evitar la multicolinealidad perfecta.
- **Hiperparámetros**: En Scikit la decisión sobre si el modelo tiene o no intercepto.

Regresiones Ridge, Lasso, Elastic Net

Calcula los parámetros ideales para minimizar la suma de los errores al cuadrado al mismo tiempo penalizando el valor de los Betas, usando la norma 1 del vector (Lasso) o la norma 2 (Ridge) o ambas (Elastic Net).

- **Peprocesamiento**: El método requiere que los features estén en la misma unidad. Si las variables no están todas en la misma escala, arbitrariamente vamos a penalizar los coeficientes de unas más que los de otras. Esto puede funcionar bien pero habitualmente no es lo que queremos.
- **Variables categóricas**: Admite. Hay que crear $k-1$ variables dummy por cada categórica para evitar la multicolinealidad perfecta. Si queremos ponderar cada dummy en el término de penalización igual que las demás variables entonces también hay que estandarizarlas.
- **Hiperparámetros**: Alpha (a veces llamado Lambda), que regula cuánto se penaliza el valor de los coeficientes. A mayor alpha mayor sesgo y menor varianza.

K Nearest Neighbors (KNN)

Clasifica cada punto nuevo buscando los “k” datos del conjunto de entrenamiento más cercanos y promediando la clase de éstos.

- **Preprocesamiento**: El método se basa en una matriz de distancias. Si los features no se encuentran todos en la misma escala una dimensión podría tener arbitrariamente más peso que otras a la hora de determinar las distancias.
- **Variables categóricas**: La medida de distancia por default (euclídea) no tiene sentido matemático con variables categóricas. Se puede usar otras medidas de distancia como base y el algoritmo funciona.
- **Hiperparámetros**:
 - K: La cantidad de vecinos que se usa en la clasificación. A mayor K, mayor sesgo y menor varianza.
 - Weight: Si dentro de los K vecinos más cercanos, se quiere ponderar más a algunos para hacer la clasificación.

Regresión Logística

Calcula la probabilidad $P(Y=1|X)$ aplicando una función "sigmoidea" a una regresión lineal para obtener valores entre 0 y 1.

- **Preprocesamiento**: Igual que en regresión lineal, los parámetros pueden dar cuenta de las distintas unidades. Sin embargo, cómo los estimadores de los coeficientes se obtienen mediante descenso gradiente por lo que si los features no se encuentran todos en la misma escala una dimensión podría tener arbitrariamente más peso que otras.
- **Variables categóricas**: Admite. Hay que crear $k-1$ variables dummy por cada categórica para evitar la multicolinealidad perfecta..
- **Hiperparámetros**:
 - No tiene

Regresión Logística Regularizada

Calcula la probabilidad $P(Y=1|X)$ aplicando una función “sigmoidea” a una regresión lineal para obtener valores entre 0 y 1 y aplica un coeficiente de penalización sobre el valor absoluto de los coeficientes.

- **Estandarización**: Además del motivo anterior hay que remover las unidades de las variables por la misma razón que en Ridge y Lasso.
- **Variables categóricas**: Admite. Hay que crear $k-1$ variables dummy por cada categórica para evitar la multicolinealidad perfecta. .
- **Hiperparámetros**:
 - C. Se comporta al revés que Lambda, un C más alto hace que el modelo elija coeficientes también más altos, es decir, regulariza menos el modelo. A mayor C menor sesgo y mayor varianza.

Naive Bayes

Calcula la probabilidad de Y en base a cada uno de los features de forma independiente y se calcula la productoria de todas las dimensiones de X.

- **Preprocesamiento**: No requiere remover las unidades. En ningún momento se hacen comparaciones entre distintos features, con lo cual todas pueden tener unidades distintas y no es un problema para el modelo.
- **Variables categóricas**: Admite. Hay que crear k-1 variables dummy por cada categórica para evitar la multicolinealidad perfecta. .
- **Hiperparámetros**:
 - No tiene

Hiperparámetro: aquella/s característica/s del modelo que no se “aprenden” de forma directa en los estimadores.

Son valores que tiene que definir quien implementa el modelo.

Algunos modelos de scikit learn transforman los hiperparámetros en parámetros. Por ejemplo el modelo `RidgeCV()`, elige el mejor valor de alpha, por lo cual en este modelo alpha es un parámetro y no un hiperparámetro.

En el modelo `Ridge()` de la misma biblioteca, alpha ocupa el lugar de parámetro C.

- Dos grandes métodos (no los únicos) de búsqueda de hiperparámetros:
 - **Grid Search:** se busca encontrar el mejor hiperparámetro dentro de una “grilla” (grid) especificada de forma manual. Se realiza una búsqueda exhaustiva para cada valor de la grilla y se elige el parámetro que minimiza una determinada métrica de error (generalmente calculada mediante cross-validation)
 - **Random Search:** dado que grid search es exhaustiva, puede ser computacionalmente intensiva si el espacio de búsqueda es muy grande. Por eso, random search, realiza la búsqueda en un subset (seleccionado aleatoriamente) de parámetros, achicando el espacio de búsqueda.

- GridSearch es la estrategia mediante la cual buscamos los hiperparametros óptimos para un modelo predictivo
- ¿Cómo lo hace? Experimenta con diferentes tipos de combinaciones hiperparametros
- Se trata de buscar las características óptimas, para el problema que queremos abordar, y validando el efecto de ésta selección sobre la performance del modelo con ***cross-validation***
- Se denomina grid, porque la idea es hacer un “retículo” con todos los hiperparametros ensayados y sus resultados en el modelo

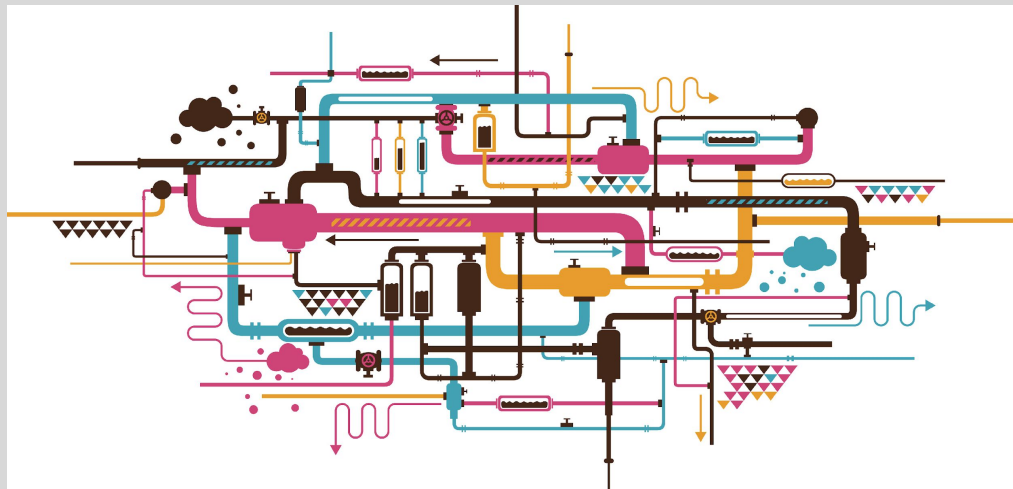
A la hora de implementar en sklearn una búsqueda sobre los hiperparámetros tenemos que tener en cuenta las siguientes cuestiones:

- Elegir un estimador, es decir, un modelo sobre el cual queremos trabajar
- Elegir un espacio de parámetros donde vamos a hacer la búsqueda.
- Elegir un método de búsqueda sobre los modelos candidatos (Random Search o Gridsearch)
- Un esquema de validación cruzada, donde se deben elegir la cantidad de particiones.
- La métrica de evaluación para elegir el mejor modelo

Práctica Guiada : usando GridSearch en K- Nearests Neighbours



- Los **hiperparámetros** son características de un modelo que no se “aprenden” directamente de los datos
- Se puede realizar una estimación de la combinación óptima usando como método una búsqueda exhaustiva a lo largo de un “grid” de valores
- Se elige la combinación que minimiza alguna métrica de error o de costo



DigitalHouse >
Coding School

DATA SCIENCE

MÓDULO 4

Pipelines

Pipelines



1

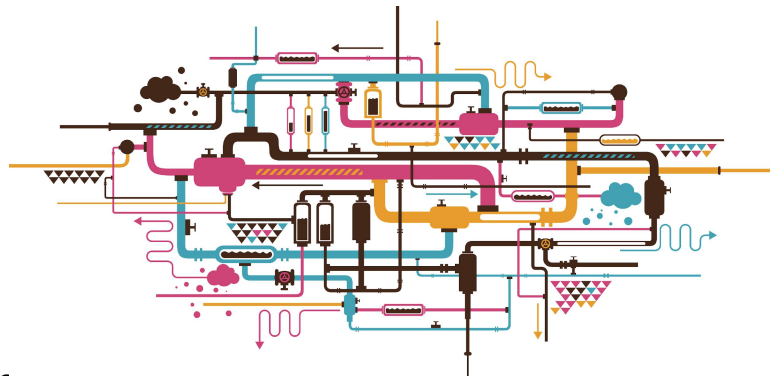
Crear pipelines para limpiar y manipular datos

2

Crear un transformador custom

3

Usar pipelines en combinación con clasificación, FeatureUnion y GridSearch



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>

PRACTICA GUIADA

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 de las páginas, dentro del campo boilerplate (json)

Z	AA
_errors_r	"label"
9575"	"0"

PIPELINES

Algunos atributos



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

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))])
```

Dentro de un Pipeline, cada paso individual también puede ser reemplazado como si fuera un parámetro e incluso ignorados si son no-finales y se setean a None:

```
>>> from sklearn.linear_model import LogisticRegression

>>> params = dict(reduce_dim=[None, PCA(5), PCA(10)],

...               clf=[SVC(), LogisticRegression()],

...               clf__C=[0.1, 10, 100])
```

En params estamos preparando la posibilidad de generar un pipeline que ejecute o no un paso de reducción con PCA(5) o PCA(10), luego aplique un modelo de SVC o de Regresión Logística, con distintos valores del parámetro C.

PIPELINES + GRIDSEARCH



GridSearchCV genera candidatos desde una grilla de parámetros y valores que se setean en *param_grid* probando exhaustivamente todas las combinaciones.

Ejemplo:

```
param_grid = [  
    {'C': [1, 10, 100, 1000], 'kernel': ['linear']},  
    {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},  
]
```

Dos grillas serán exploradas:

1. Usando un Kernel de tipo lineal y los valores en [1, 10, 100, 1000] para el parámetro C.
2. Usando un Kernel RBF, y la combinatoria de los valores de C [1, 10, 100, 1000] y los valores de gamma en [0.001, 0.0001].

GridSearchCV implementa todas las combinaciones y retiene aquella con mejores resultados.

```
pipeline = Pipeline ([  
    ('vect', CountVectorizer ()),  
    ('tfidf', TfidfTransformer ()),  
    ('clf', SGDClassifier ()),  
])
```

Genero un Pipeline con:

1. Un vectorizador de texto
2. Un transformador de la matriz obtenida
3. Un clasificador lineal.

```
parameters = {  
    'vect__max_df': (0.5, 0.75, 1.0),  
    'vect__ngram_range': ((1, 1), (1, 2)),  
    'clf__alpha': (0.00001, 0.000001),  
    'clf__penalty': ('l2', 'elasticnet'),  
}
```

Se genera una lista con los parámetros a probar:

1. max_df (frecuencia de corte) y ngram_range (unigramas y bigramas) para 'vect'
2. alpha de 10EXP-5 y 10EXP-6 y l2 o elastic net para la regularización

```
grid_search = GridSearchCV (pipeline, parameters, n_jobs=-1, verbose=1)
```

Se configura la búsqueda con el pipeline y los parámetros posibles. Se van a ejecutar todos los pasos, tantas veces como combinatoria de parámetros haya (en este caso 24)

```
print ("Performing grid search...")
grid_search.fit(data.data, data.target)

print ("Best score: %0.3f" % grid_search.best_score_)

print ("Best parameters set:")
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):

    print (" \t%s: %r" % (param_name, best_parameters[param_name]))
```

```
Performing grid search...
Best score: 0.940
Best parameters set:
  clf__alpha: 9.9999999999999995e-07
  clf__n_iter: 50
  clf__penalty: 'elasticnet'
  tfidf__use_idf: True
  vect__max_n: 2
  vect__max_df: 0.75
  vect__max_features: 50000
```


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>

Data Manipulators

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

Data Imputation

- Imputer

Function Transformer

- FunctionTransformer

Label Manipulators

- LabelBinarizer
- LabelEncoder

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:

```
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

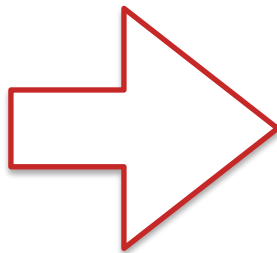
    def fit(self, *):
        return self

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]]
```

2. La otra manera de generar un Custom Transformer es usando la función `FunctionTransformer` del módulo de pre-procesamiento.

Ejemplo: Crear un transformador custom que devuelve el logaritmo del input.

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p)
>>> X = np.array([[0, 1], [2, 3]])
>>> transformer.transform(X)
array([[ 0.          ,  0.69314718],
       [ 1.09861229,  1.38629436]])
```

¿En qué difieren ambos métodos?

`FunctionTransformer` usa una función existente (object o user-defined).

FEATURE UNION

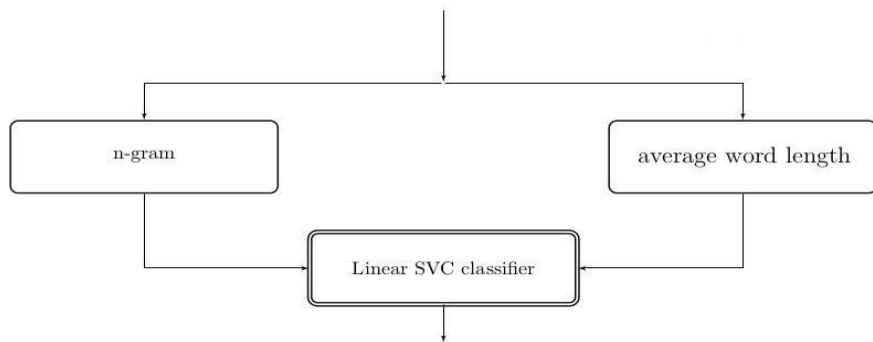


Hay casos en los que nos va a interesar juntar features transformadas aplicando distintos métodos, y luego correr pasos siguientes.

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())
```

1)

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>


The image displays two screenshots of the Luigi Task Visualiser interface, a web-based tool for monitoring and managing data pipeline tasks.

Left Screenshot: Luigi Task Status Overview

- Luigi Task Status** (Header)
- Task List** (Tab)
- Dependency Graph** (Tab)
- Workers** (Tab)
- Task Status Summary:**
 - PENDING TASKS: 6679
 - RUNNING TASKS: 0
 - DONE TASKS: 6946
 - UPSTREAM FAILED: 351
 - DISABLED TASKS: 0
 - UPSTREAM DEPENDENCY FAILED: 0
- Task List:** A list of tasks, including **CreateReportingUsage**, which is highlighted.
- Task Details:** A table showing the status of tasks for the **CreateReportingUsage** family. The table has columns for **Name** and **Status**. The status is **DONE** for most tasks, and **UPSTREAM_FAILED** for some.
- Showing 1 to 6 of 6 entries (filtered from 14,064 total entries)**

Right Screenshot: Luigi Task Status Active tasks

- Luigi Task Status** (Header)
- Task List** (Tab)
- Dependency Graph** (Tab)
- Task Details:** A detailed view of the **CreateReportingUsage** task, showing its parameters and dependencies.
- Task Parameters:**
 - `UserRecs(test=False, date=2013-07-24, rec_days=4, exp_days=8, test_users=False, force_updates=False, build_from_scratch=True, index_path=/spotify/discover/index, index_version=None, FOLLOWS_SCORE=5.0)`
- Dependency Graph:** A graph showing the dependencies between tasks. The graph is a directed acyclic graph (DAG) with nodes representing tasks and edges representing dependencies. The nodes are color-coded: red for Failed, blue for Running, yellow for Pending, and green for Done.
- Task Details:** A popup window showing the details of the **AggregateUserMatrices** task, including its parameters and dependencies.

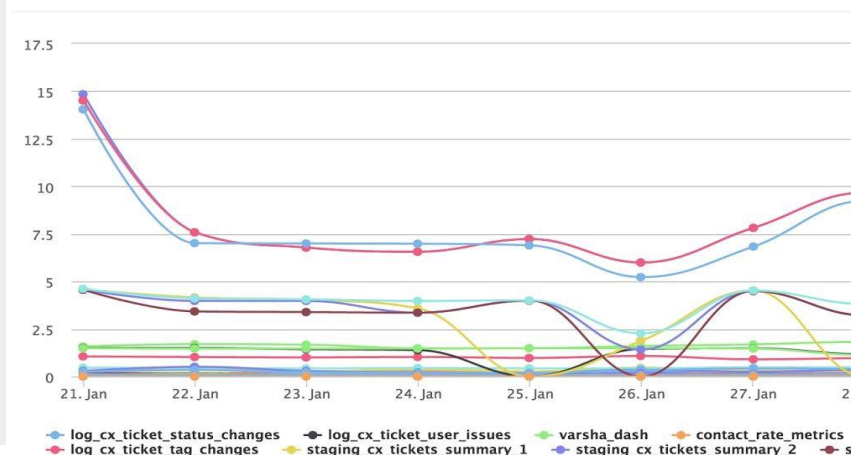

[AirFlow](#)
[DAGs](#)
[Tools ▾](#)
[Browse ▾](#)
[Admin ▾](#)
[Docs ▾](#)

 Tree View
 Graph View
 Task Duration
 Landing Times
 Gantt
 Code

[illegible]


[AirFlow](#)
[DAGs](#)
[Tools ▾](#)
[Browse ▾](#)
[Admin ▾](#)
[Docs ▾](#)

 Tree View
 Graph View
 Task Duration
 Landing Times
 Gantt
 Code



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