Machine learning avec Spark

DRIO 5101A : Big Data Analytics avec Spark

Thibaud Vienne - ESIEE Paris

Fonctionnement du cours

- 11 heures de cours :
  - Big Data : Une introduction (2h)
  - Panorama de l'écosystème Hadoop (3h)
  - Le framework Spark (3h)
  - Machine learning avec Spark (3h)

- 15 heures de TP :
  - TP 1 : Tutorial + Comptage de mots avec Spark.
  - TP 2 : Analyse de logs Apache.
  - TP 3 : Prédiction de dates de sorties de chansons.
  - TP 4 : Classification de clics internet.

- Evaluation finale

Sommaire

1. Le machine Learning en contexte Big Data
2. Rappels de Machine Learning
3. La librairie MLLib
4. La librairie Spark ML

Machine Learning en contexte Big Data

Machine learning : En local

- Le machine Learning est une composante incontournable du travail de data scientist. Il consiste en la mise en place d'algorithmes dans le but d'obtenir des prédictions / informations sur un jeu de données.

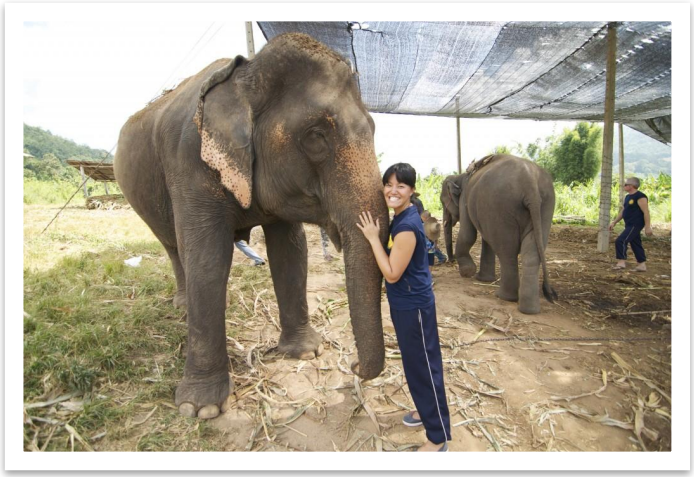
- A l'heure actuelle, de nombreuses librairies permettent de travailler sur du machine Learning. Les plus connues en python:
  - Scikit-Learn, le projet le plus avancé actuellement.
  - Les projets nltk et gensim pour le text processing.
  - Tensorflow et PyTorch, spécialisés dans les réseaux de neurones…

Machine learning : En distribué

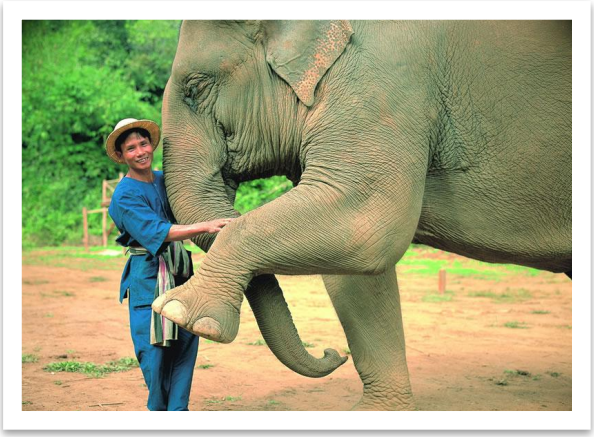
- Les librairies de machine Learning en environnement distribué se font plus rares. En effet, le développement des algorithmes est autrement plus compliqué et nécessite l'utilisation de structures de calculs distribués (Hadoop MapReduce, Spark Core…).

- Dans l'écosystème Hadoop-Spark, il existe principalement trois projets dédiés au machine Learning:
  - Le projet Apache Mahout , basé sur Hadoop MapReduce.
  - La librairie MLlib, module historique du Framework Spark.
  - La librairie Spark ML, apparue depuis la version Spark 1.2.

Machine learning : Apache mahout



Apache Mahout 2011-2015 R.I.P



Machine learning : Apache mahout

- Le Projet de Machine Learning Apache Mahout est complètement à l'arrêt et n'est plus mis à jour depuis Juillet 2015.

- Cela tient au fait que Mahout fonctionne au-dessus de Hadoop MapReduce ! Comme vu précédemment, Hadoop MapReduce est une implémentation beaucoup plus lente que Spark Core. En effet, pour un même traitement, Hadoop MapReduce écrit et charge les données sur disque (HDFS) entre deux itérations.

- Les traitements de Machine Learning y sont de 10 à 100x plus lents que les librairies basées sur Spark.

Machine learning : Spark MLlib et ML

- Les projets MLlib et Spark ML sont tous deux issus du Framework Spark. Pour être plus précis, le module MLlib se décompose en deux « packages » distincts, MLlib et Spark ML.

- Ces deux librairies possèdent d'importantes différences dans leur implémentation, notamment:
  - Spark MLlib travaille sur des objets de type RDDs.
  - Spark ML travaille sur des objets dataframes issus du module Spark SQL.

- **Note** : Depuis la version Spark 2.0 (juillet 2016), la librairie MLlib est dite dépréciée. Cela signifie qu'elle reste disponible actuellement mais qu'il n'y aura plus de contributions à son actif.

- **Note**: Les deux librairies seront présentées lors de ce cours. Lors des TPs, le travail s'effectuera majoritairement avec Spark ML.

Algorithmes de Machine Learning distribués

Rappels

L'exemple de la régression linéaire

Descente de gradient

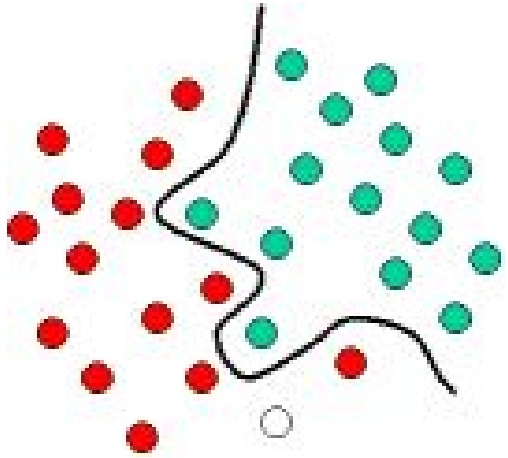
Rappels : Machine learning

- Concernant le Machine Learning:
  - Cela consiste à apprendre de la connaissance à partir de données.

- En machine learning, on distingue principalement deux types d'apprentissage :

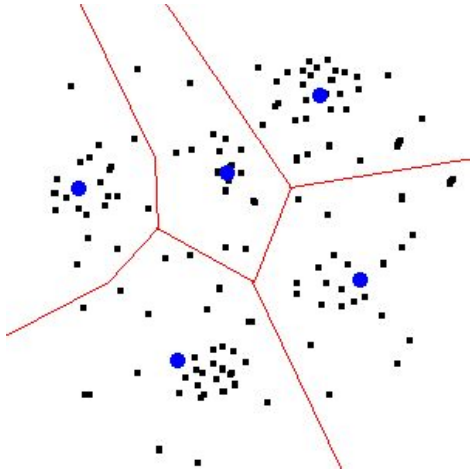
Apprentissage supervisé

Données *= observations + étiquettes*
Connaissance *→ relation entrée-sortie*

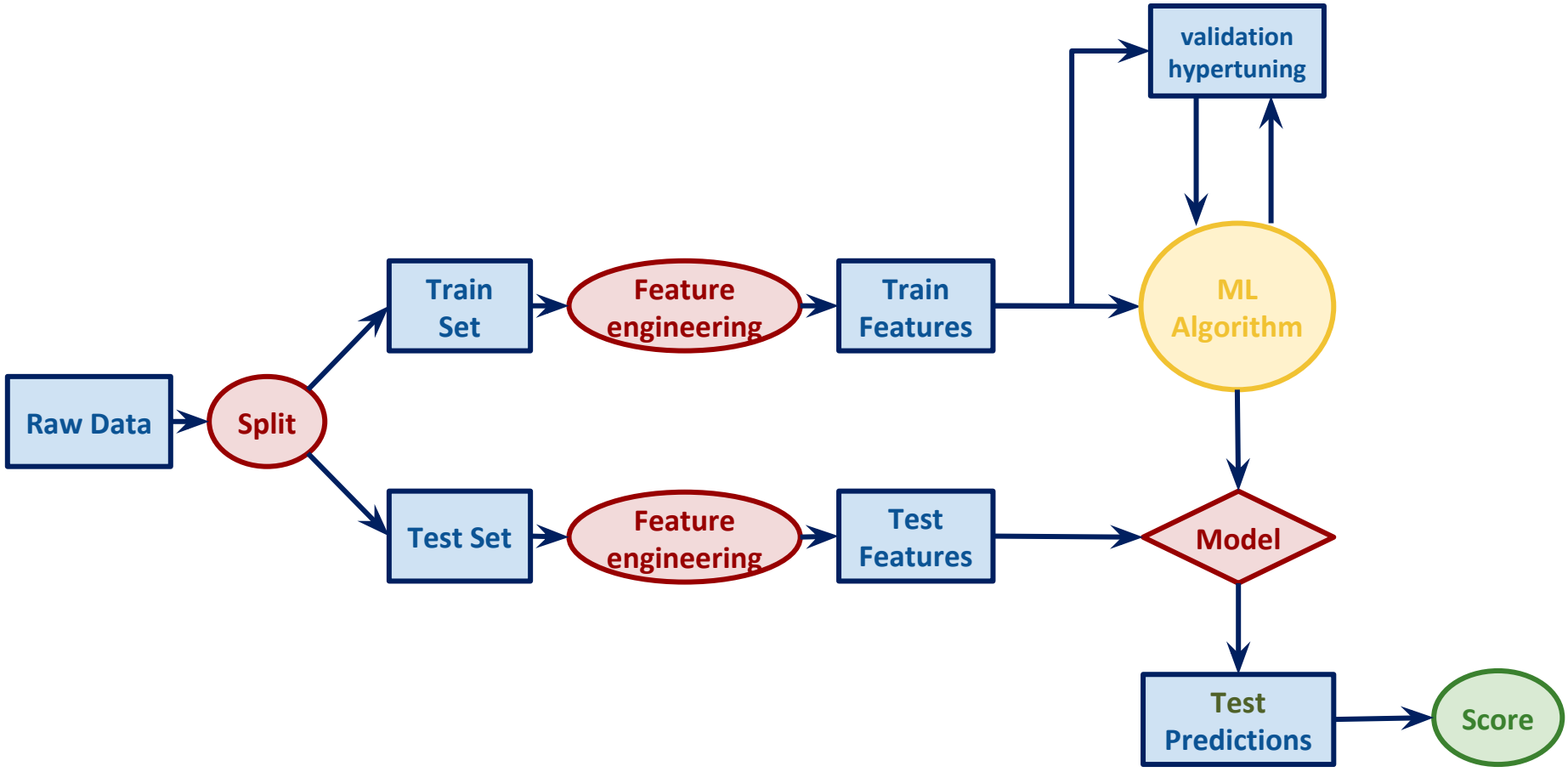


Apprentissage non-supervisé

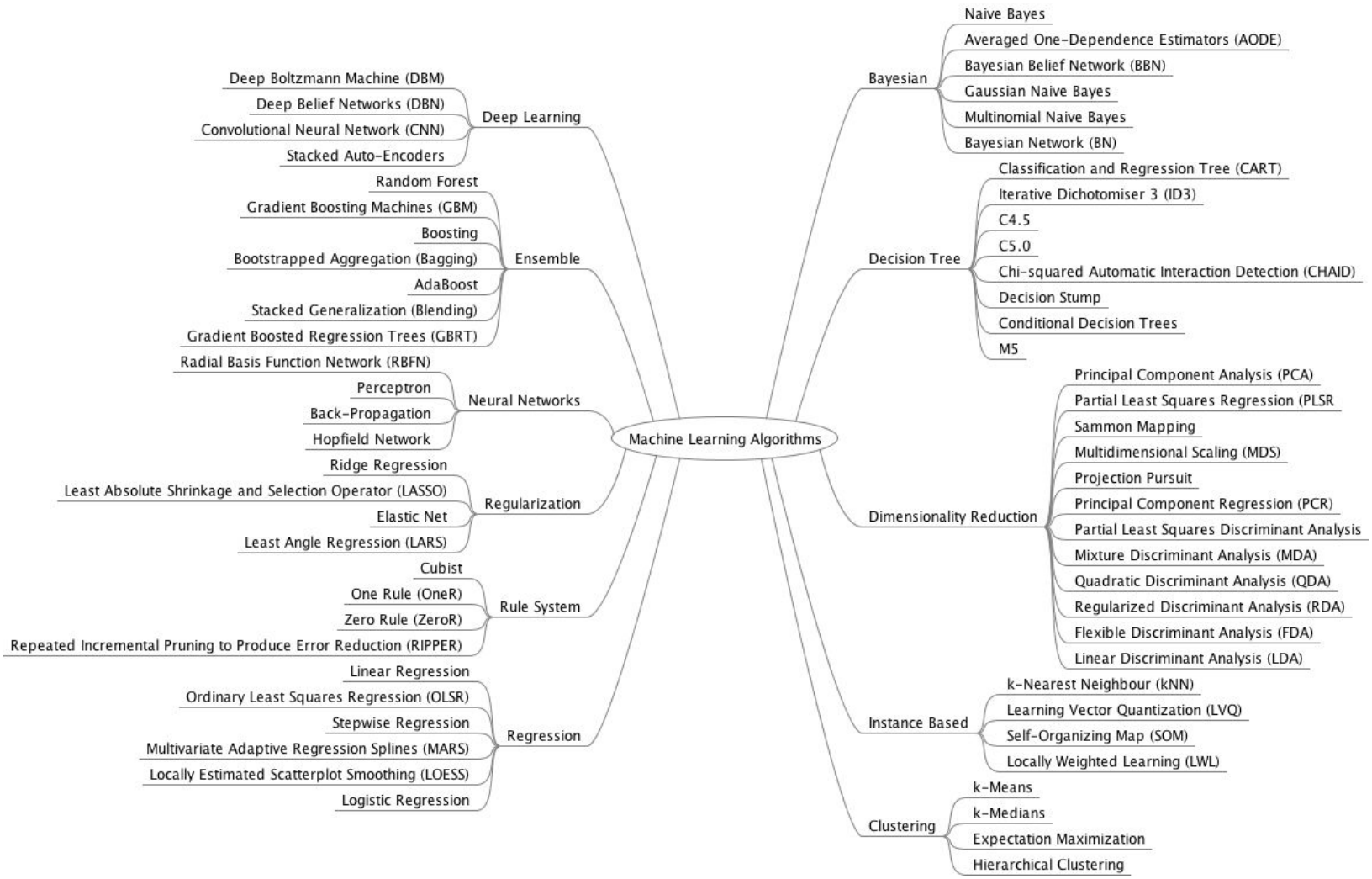
Données *= observations*
Connaissance *→ structures latentes*



Rappels : Pipeline machine learning



Rappels : Les algorithmes



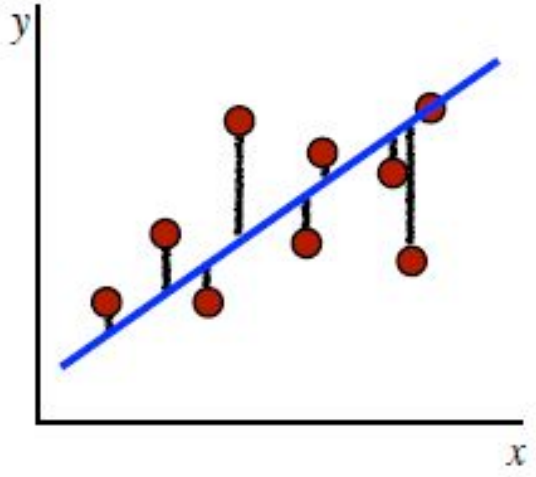
La régression linéaire : Définition

- En machine learning supervisé, la régression linéaire consiste à:
  - Trouver la « meilleure » approximation linéaire d'une variable / plusieurs variables.
  - x = une variable explicative / un vecteur de variables explicatives.
  - y = la variable expliquée → *valeur continue*

- Dans le cadre d'une régression linéaire (1D - 1 variable explicative), nous devons alors déterminer w0 et w1 tel que :

$$y \approx \hat{y} = w_0 + w_1 x$$

Intercept / Offset — Slope



La régression linéaire : Résolution

- Dans le cas d'une régression linéaire (simple / multivariée), il est possible de calculer la solution à ce problème d'optimisation par la recherche d'un minimum local tel que :
  - $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \rightarrow$ *données d'apprentissage.*
  - x = une variable explicative / un vecteur de variables explicatives.
  - y = la variable expliquée → valeur continue
  - Il faut alors trouver le vecteur **w** minimisant le MSE sur les données :

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)})^2 \quad = \quad \min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

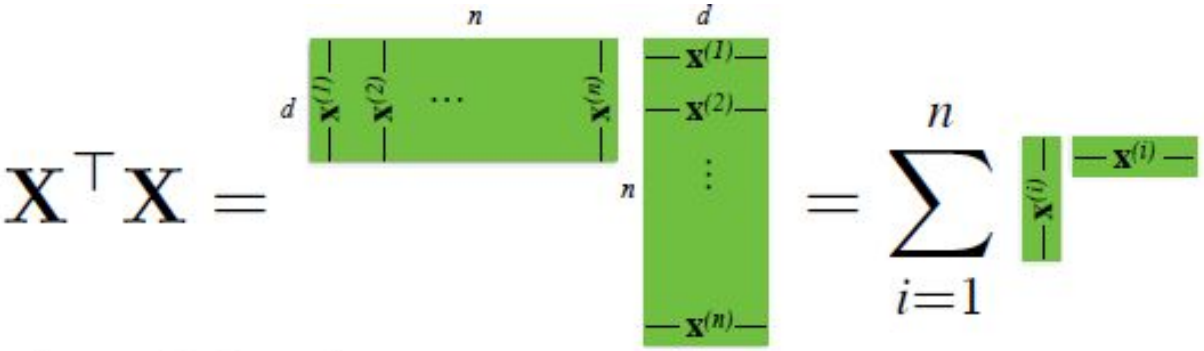
- En recherchant le minimum global au travers de dérivées / gradients, la solution explicite est alors donnée par :

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

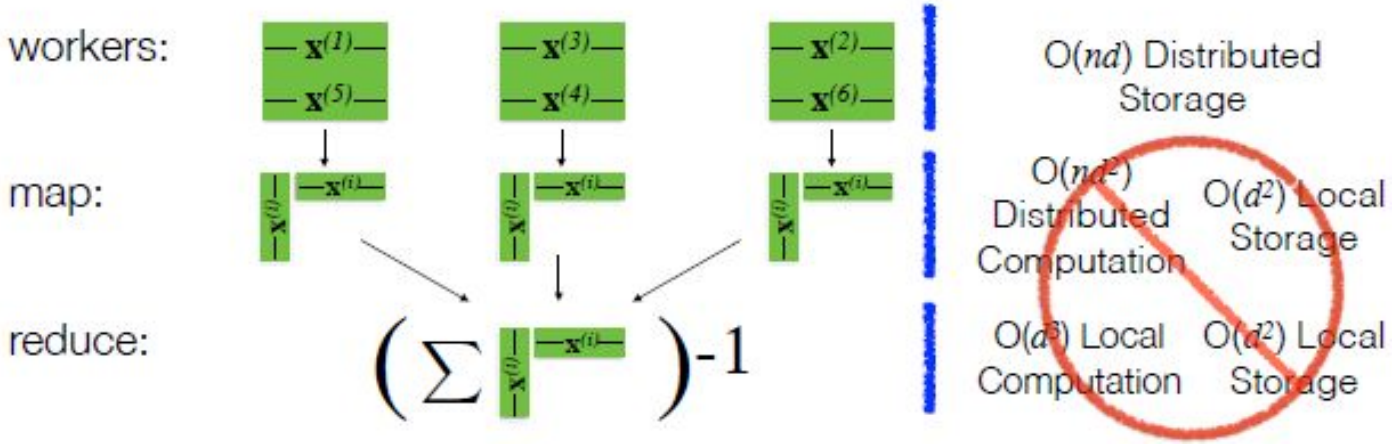
La régression linéaire : Problème

- La solution montrée précédemment ne peut être calculée de façon explicite en contexte distribué.

- En effet, la complexité du problème (en terme de stockage et de calcul) évolue de façon carrée ou cubique. Cela est alors rendu impossible en présence de grands jeux de données.



Example: $n = 6$; 3 workers



Descente de Gradient

- Pour pallier à ce problème, nous utilisons une méthode d'approximation efficace appelée "descente de gradient". Cet algorithme est alors capable d'approcher suffisamment la solution grâce au mécanisme suivant :

Ainsi, pour une régression linéaire, en posant *f(w) = MSE :*

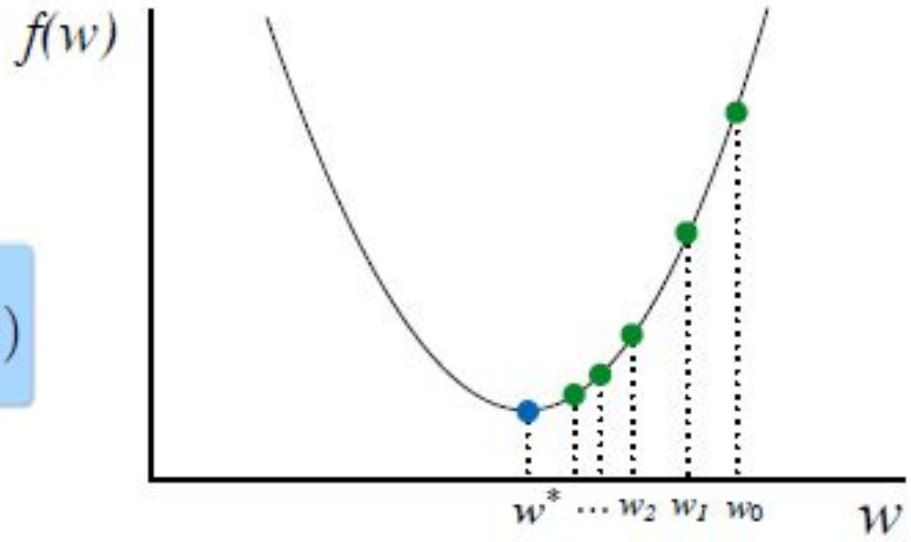
$$f(\mathbf{w}) = \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)})^2$$

- L'algorithme itératif consiste à :
  - Choisir un point initial $\mathbf{w}_0$
  - Itérer jusqu'à convergence avec la règle :

Step Size

$$\textbf{Update Rule: } \mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$$

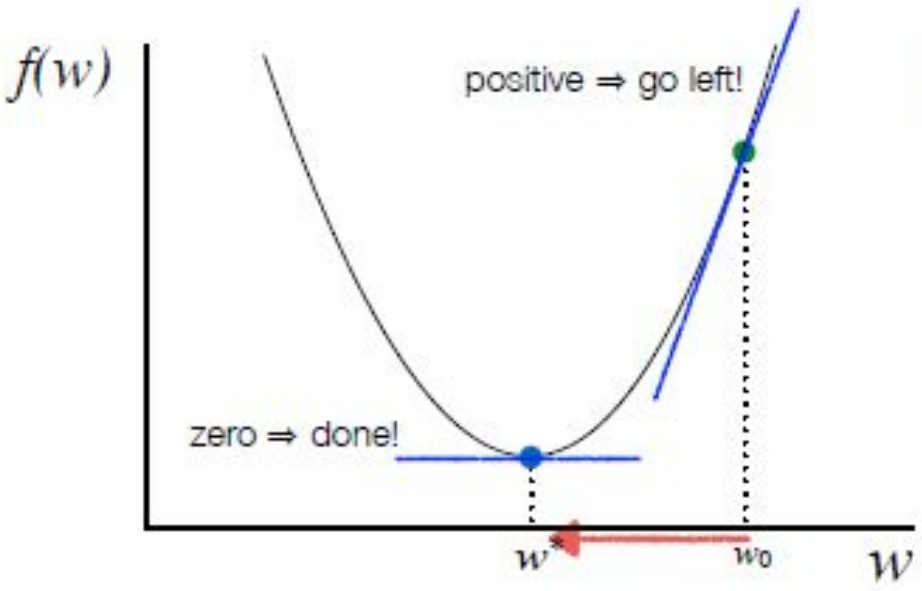
Negative Slope

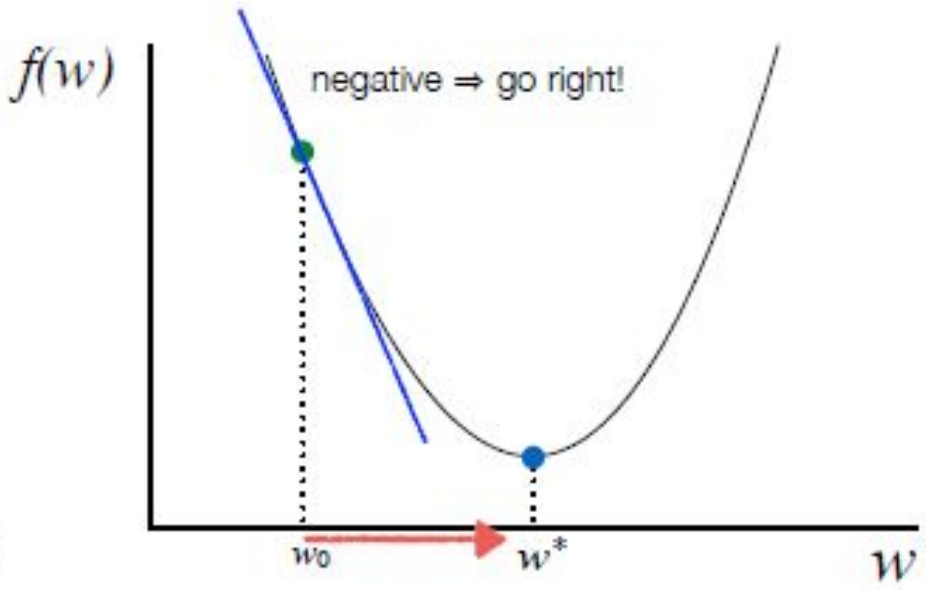


Descente de Gradient

- La règle d'apprentissage se basant alors sur le signe de la dérivée ("sens" de la pente), l'algorithme est alors capable au bout de suffisamment d'itération d'atteindre le minimum local / global.

- Cette méthode est, de façon analogue, comparable à la descente d'une bille dans un ravin.





Descente de Gradient

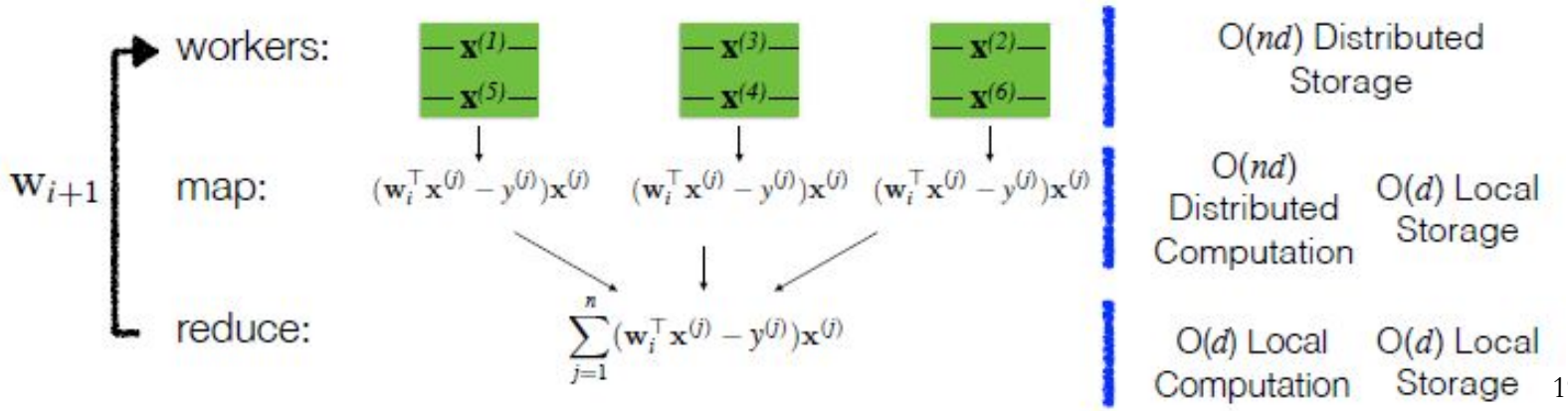
- Au vu de sa faible complexité (stockage et calcul), la descente de gradient devient alors une très bonne alternative au calcul de solution de problèmes d'optimisation.

- Une partie des algorithmes de machine learning peuvent alors distribués efficacement grâce à la descente de gradient. C'est le mécanisme implémenté dans les algorithmes Spark MLLib et Spark ML.

Vector Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$

Compute summands in parallel!
note: workers must all have $\mathbf{w}_i$

Example: $n = 6$; 3 workers



Descente de Gradient : Quelques remarques

- Dans Spark MLlib et Spar ML, quel que soit l'algorithme considéré, la parallélisation de l'algorithme et la méthode utilisée sont entièrement transparentes à l'utilisateur.

- La descente de gradient permet l'approximation de plusieurs algorithmes de machine learning (régression linéaire, régression logistique…). Pour d'autres algorithmes, d'autres techniques et mécanismes sont adoptés. C'est le cas par exemple des arbres de décision.

- Certains algorithmes se parallélisant peu voire pas du tout. dans ce cas, ceux-ci possèdent des temps de traitement élevés et/ou ne sont pas implémentés dans Spark. C'est le cas par exemple de :
    - l'algorithme ensembliste "Gradient Boosting" et assimilés qui nécessitent la création d'arbres de décision en série (processus non parallélisable). L'algorithme existe dans Spark, cependant les temps de latence seront, proportionnellement, bien plus élevés que pour un Random Forest (création d'arbres de décision indépendants donc parallélisable).
    - Le "SVM" (Support Vector Machine) n'est à l'heure actuelle pas parallélisable dans la majorité des cas. Seule la version avec kernel linéaire existe dans Spark MLLib et Spark ML.

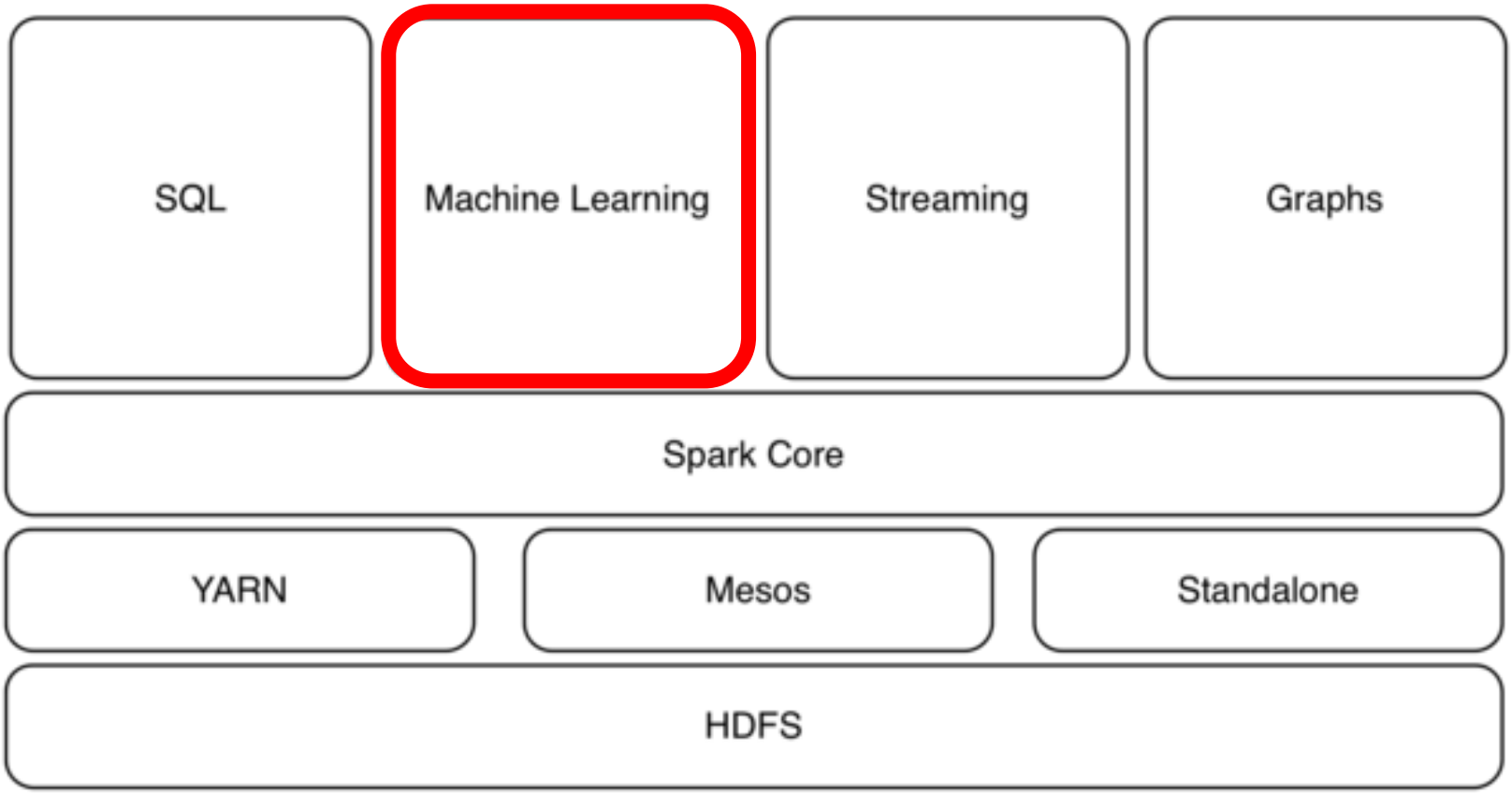
Spark MLlib

Spark MLlib

MLlib en pratique

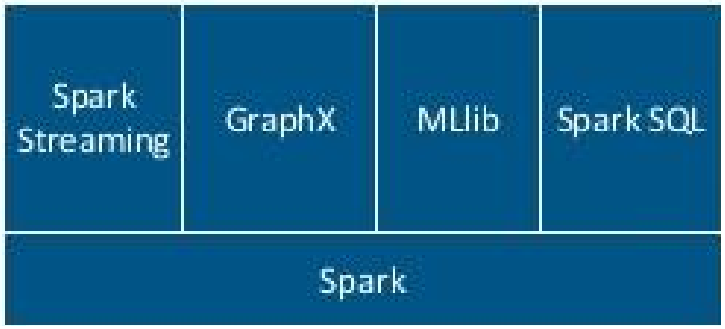
Fonctionnalités MLlib

Spark MLlib


SQL
Machine Learning
Streaming
Graphs
Spark Core
YARN
Mesos
Standalone
HDFS

Spark MLlib : Présentation

- La librairie MLlib est la librairie de Machine Learning historique de Spark.
- Au niveau écosystème, elle se place au-dessus du Spark Core. La librairie MLlib travaille exclusivement sur des RDDs.
- Pour travailler elle accepte exclusivement trois types de données qu'elle construit souvent avec des fonctions de type mapping:
  - Les objets Vector : utilisés pour les algorithmes non supervisés.
  - Les objets LabeledPoint : pour les algorithmes supervisés.
  - Les objets Rating : pour les systèmes de recommandations.
- La librairie possède une démarche de fonctionnement hétérogène qui peuvent la rendre difficile à prendre en main au premier abord.



Spark MLlib : Les objets Vector

- Un objet **« Vector »** est un tableau de nombres réels. On peut les construire de deux façons.
  - **Type « dense »** : chaque entrée doit être spécifiée.
  - **Type « sparse »** : seules les entrées non-nulles, avec leurs positions, sont spécifiées.

- Les « **Vector** » sont utilisés pour travailler avec des algorithmes non-supervisés. Ils contiennent obligatoirement des doubles.

- Un exemple de construction:
  - Soit un vecteur tel que (1.0, 0.0, 3.0).
  - Celui-ci peut se construire de deux façons avec l'API Spark MLlib.

```
from pyspark.mllib.linalg import Vectors

# Création d'un vector "dense"
vect_dense = Vectors.dense([1.0, 0.0, 3.0])

# Crée d'un vecteur "sparse"
vect_parse = Vectors.sparse(3, [0, 2], [1.0, 3.0])
```

Spark MLlib : Les objets LabeledPoint

- Un objet **« LabeledPoint »** est un vecteur dense/sparse associé avec un label.
- Il est utilisé pour travailler avec des algorithmes supervisés. Le label est en machine Learning, la variable à prédire, souvent nommée par défaut « Y ».
- Pour stocker un label, il est là encore nécessaire d'utiliser un double.
- Pour de la classification, le label doit prendre les valeurs 0.0, 1.0, 2.0 et ainsi de suite…

```
from pyspark.mllib.regression import LabeledPoint


# Création d'un LabeledPoint ( label + vecteur dense )
labeled_point = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Accéder au label
label = labeled_point.label

#Accès à l'attribut features
features = labeled_point.features
```

Spark MLlib : Les objets Rating

- Les objets « **Rating** » sont utilisés spécialement dans les systèmes de recommandation.

- Pour créer un objet Rating, celui-ci a besoin de trois paramètres:
  - Un entier représentant l'utilisateur (un ID).
  - Un entier représentant un item (un film, un produit…)
  - Un rating sous forme de double (la note attribuée)

```
from pyspark.mllib.recommendation import Rating

# Création d'un rating pour l'utilisateur 47 du produit n°55 avec la note de 9.0 / 10
rating = Rating(47, 55, 9.0)
```

MLlib en pratique : Création d'objets

- Il est bien sûr possible et conseillé de construire des RDDs Vectors/LabeledPoint/Ratings depuis l'utilisation de fonction mapping.

- Un exemple:

rdd[0]	rdd[1]	rdd[2]	rdd[3] (label)
14.78	0.0	5.78	0.0
15.27	0.0	6.25	1.0
12.25	1.0	7.77	0.0

```
from pyspark.mllib.regression import LabeledPoint

# Mapping d'un rdd de machine learning
rdd_mllib = rdd.map(lambda row : LabeledPoint(row[3], [row[0], row[1], row[2]]))
```

MLlib en pratique : Training

- Sous MLlib, tous les algorithmes sont associés à une classe python.
- En python, toutes ces classes possèdent une méthode « **.train()** » qui permet d'entrainer l'algorithme sur des RDDs de *type Vectors / LabeledPoint / Ratings*.
- Le tuning de l'algorithme se fait au travers des paramètres pouvant être passés à la méthode « .train() ».
- La méthode retourne un objet de classe « **Model** ».

```
from pyspark.mllib.regression import LabeledPoint, LogisticRegressionWithLBFGS

# Entrainement de l'algorithme
model = LogisticRegressionWithLBFGS.train(rdd_mllib, iterations=100, regParam=0.1, regType='l2')
```

MLlib en pratique : Prediction

- Une fois l'algorithme entraîné, la méthode retourne un objet de classe « Model ».
- Celui-ci possède à son tour une méthode « **.predict()** » qui va utiliser le modèle entraîné pour prédire le label / la variable recherchée.
- **Note :** Les traitements s'effectuent en distribué. Ne pas oublier d'utiliser une méthode de mapping pour prédire les résultats.
- **Note :** Un mapping sous forme d'un tuple <*valeur_vraie, prédiction*> facilite grandement l'évaluation de notre modèle.

```
from pyspark.mllib.regression import LabeledPoint, LogisticRegressionWithLBFGS

# Entrainement de l'algorithme
model = LogisticRegressionWithLBFGS.train(rdd_mllib_train, iterations=100, regParam=0.1, regType='l2')

# Effectuer les prédictions d'après le modèle
rdd_pred = rdd_mllib_test.map(lambda r: (r.label, model.predict(r.features)))
```

Fonctionnalités MLlib : Version 2.2 (1)

- Algorithmes supervisés de classification (*pyspark.mllib.classification)* :
  - Régression logistique régularisée (*classification.LogisticRegressionWithLBFGS*).
  - Support Vector Machine (*classification.SVMWithSGD*).
  - Naive Bayes (*classification.NaiveBayes*).

- Algorithmes supervisés de régression (*pyspark.mllib.regression)* :
  - Régression linéaire régularisée *(regression.LinearRegressionWithSGD).*
  - Régression isotonique (*regression.IsotonicRegressionModel).*

- Techniques d'arbres (*pyspark.mllib.tree)* :
  - Arbres de décisions (*tree.DecisionTree).*
  - Random Forest (*tree.RandomForest).*
  - Gradient Boosting (*tree.GradientBoostedTreesModel*).

Fonctionnalités MLlib : Version 2.2 (2)

- Algorithmes non-supervisés de clustering (*pyspark.mllib.clustering)*:
  - K-means (*clustering.Kmeans).*
  - Bisecting K-means (*clustering.BisectingKMeans*).
  - Gaussian Mixture (*clustering.GaussianMixture*).
  - Latent Dirichlet Allocation (*clustering.LDA*).

- Algorithmes de recommandation (*pyspark.mllib.recommendation)*:
  - Alternating Least Squares *(recommendation.ALS).*

- Statistiques (*pyspark.mllib.stat)*:
  - Statistiques descriptives (mean, variance, min, max…).
  - Test du Chi2 de Pearson *(stat.Statistics.chiSqTest).*
  - Calcul de corrélations (*stat.Statistics.corr).*
  - Test de Komolgorov Smirnov (*stat.kolmogorovSmirnovTest).*
  - *…*

Fonctionnalités MLlib : version 2.2 (3)

- Création et manipulation de features (*pyspark.mllib.feature*):
  - Normalisation de features (*feature.Normalizer).*
  - Rescaling de features (*feature.StandardScaler).*
  - Hashing Term Frequency et Inverse Document Frequency
  - Word2Vec (*feature.Word2Vec*).
  - Sélection de features (*features.ChiSqSelector).*

- Evaluation de modèles (*pyspark.mllib.evaluation)*:
  - Métriques binaires et multi-classes (precision, recall, sensitivité, précision, F-score …).
  - Métriques de régression (MAE, MSE, RMSE, R2, ...).
  - Métriques pour les systèmes de recommandation.

Fonctionnalités MLlib : Quelques remarques

- La librairie MLLib, bien que très efficace, possède quelques inconvénients (manque de souplesse, fonctionnement hétérogène, pas de pipelines…).
- La libraire Spark ML se propose d'effacer ces problèmes par une API plus robuste et un fonctionnement proche de scikit-learn.
- **Note :** Depuis la version Spark 2.0 (juillet 2016), la librairie MLlib est dite dépréciée. Elle reste disponible mais ne sera plus mise à jour.
- **Note :** Plusieurs autres fonctionnalités de MLlib seront vues au cours des TPs à venir.

Spark ML

Spark ML

Spark ML en pratique

Fonctionnalités Spark ML

Spark ML : Présentation

- Spark ML est une librairie de machine Learning sortie depuis la version 1.2 de Spark.
- Elle a pour but de combler les lacunes de MLlib et notamment d'uniformiser le fonctionnement des algorithmes de machine Learning.
- Au niveau architecture, Spark ML travaille exclusivement avec des dataframes, elle est donc communément placée au-dessus du module Spark SQL.

Spark ML

- La librairie Spark ML est basée sur plusieurs objets abstraits qui utilisés ensemble permettent de réaliser des programmes de Machine learning.

- On retrouve principalement 4 types d'objets :

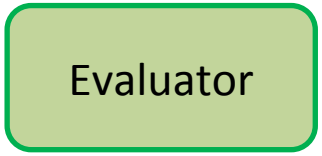
Les dataframes

	account	campaign	date	successes	trials	rate
455	1	Campaign #76	2012-08-14 11:56:20 -0400	2	2	1.000000
449	1	Campaign #78	2012-08-14 12:06:20 -0400	2	2	1.000000
438	1	Campaign #87	2012-08-14 18:06:30 -0400	27	118	0.228814
431	1	Campaign #95	2012-08-15 00:07:42 -0400	22	118	0.186441
422	1	Campaign #99	2012-08-15 01:27:48 -0400	25	120	0.208333

Les Estimators

Les Transformers

Les Evaluators



Spark ML : Les Transformers

- Les « **transformers »** sont des objets qui transforment un dataframe en un autre dataframe.
- Dans Spark ML, ils possèdent une méthode « **.transform()** » qui prend un entrée un dataframe et retourne un nouveau dataframe.
- Généralement, ces méthodes ne transforment pas radicalement un dataframe mais se contentent d'ajouter une ou plusieurs colonnes.

Spark ML : Les Estimators

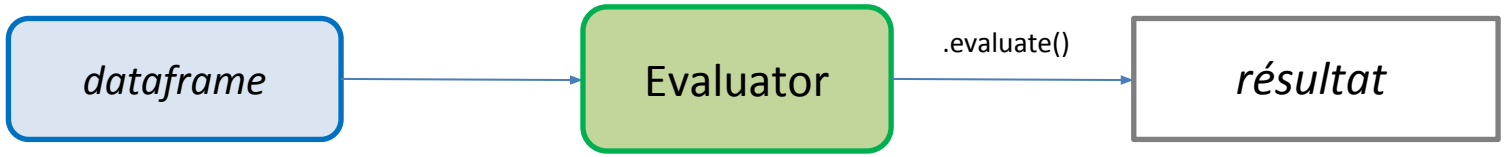
- Les « **estimators** » sont des objets, qui, à partir d'un dataframe créent un objet de type transformer.

- Pour cela, ils possèdent tous une méthode « .fit() » qui prend en entrée un dataframe et retourne en sortie un transformer.

- **Note :** contrairement à d'autres librairies de machine Learning, les « estimators » ne sont pas uniquement des algorithmes de machine Learning.

Spark ML : Les Evaluators

- Les « **evaluators »** permettent d'évaluer un modèle.
- Pour cela, ils possèdent tous une méthode « **.evaluate()** » qui prend en entrée un dataframe et retourne en sortie un résultat de performance en local à l'utilisateur.
- **Note :** Il existe de nombreux objets Evaluators. Le choix de celui-ci dépend du type de problème posé (classification/régression, supervisé/non-supervisé, etc…)



Spark ML en pratique

- De façon analogue à MLlib, il est obligatoire de préparer les features et les labels avant de pouvoir les utiliser dans un algorithmes de machine Learning.

- **Concernant les datasets, ceux-ci doivent :**
  - Être splittés en un dataset de training et un dataset de Test.

- **Concernant les features, ceux-ci doivent:**
  - Être représentées sous forme de doubles.
  - Être encodées pour les variables catégorielles multi-classes.
  - Être assemblées dans une seule et même colonne d'un dataframe.

- **Concernant les labels/variables à prédire (apprentissage supervisé), ceux-ci doivent:**
  - Être représentées sous forme de double.
  - Être encodées pour les variables catégorielles multi-classes.

Spark ML en pratique : Splitting

- En machine Learning, il est nécessaire de séparer nos données entre un dataset d'apprentissage et un dataset de test.
- Cela permet ainsi de pouvoir évaluer la capacité de notre algorithme à effectuer des prédictions sur des données non apprises.
- La méthode la plus simple pour splitter un dataframe est d'utiliser la méthode « **.randomSplit()** » de l'API Spark dataframe.
- Cette méthode splitte les données de façon aléatoire. Cette méthode est donc statistiquement plus viable qu'une simple coupure..

```
# Splitting d'un dataframe de façon aléatoire
(df_train, df_test) = df.randomSplit([0.75, 0.25])
```

Spark ML en pratique : StringIndexer

- Spark ML ne peut pas travailler sur des variables au format String. Il est donc nécessaire lorsque l'on possède des features/labels au format string de les indexer.

- Pour cela, on utilise un estimator appelé « **StringIndexer** ».

- Lors de la création d'un objet de type StringIndexer, il faut obligatoirement lui spécifier les arguments suivants.
  - inputCol : le nom de la colonne sur lequel sera appliqué le StringIndexer.
  - outputCol : le nom de la colonne indexée créée grâce au StringIndexer.

- **Note** : un StringIndexer est un estimator et non pas un transformer parce qu'il est nécessaire pour Spark d'évaluer l'ensemble des catégories d'une colonne avant transformation.

Colonne catégorielle	Colonne indexée
« A »	0.0
« B »	1.0
« C »	2.0
« A »	0.0

Spark ML en pratique : StringIndexer

- Pour indexer les données, on commence par créer un objet de type « StringIndexer ».
- On fitte ensuite les données avec la méthode « **.fit()** ». On obtient ensuite un transformer « indexer_model ».
- On transforme ensuite les données avec la méthode « **.transform()** » de l'*indexer_model*.

```
from pyspark.ml.feature import StringIndexer

# Instanciation d'un objet StringIndexer
indexer = StringIndexer(inputCol='colonne_categorielle', outputCol='colonne_indexee')
indexer_model = indexer.fit(df)

# transformation des données
df_indexed = indexer_model.transform(df)
df_indexed.show(4)
```

Spark ML en pratique : OneHotEncoder

- Lorsque les variables catégorielles possèdent plus de deux classes, il est nécessaire après la réalisation de l'indexation, de les encoder.

- Pour cela, on peut utiliser dans Spark ML un « transformer » appelé « OneHotEncoder ».

- Lors de l'instanciation de l'objet, il est nécessaire de spécifier les paramètres suivants:
  - intputCol : le nom de la colonne indexée.
  - outputCol : le nom que portera la colonne encodée.

Colonne catégorielle	Colonne indexée	Colonne encodée
« A »	0.0	[1.0 , 0.0 , 0.0 , 0.0]
« B »	1.0	[0.0 , 1.0 , 0.0 , 0.0]
« C »	2.0	[0.0 , 0.0 , 1.0 , 0.0]
« D »	3.0	[0.0 , 0.0 , 0.0 , 1.0]

```
from pyspark.ml.feature import OneHotEncoder

# One Hot Encoding des données
encoder = OneHotEncoder(inputCol='colonne_indexee', outputCol='colonne_encodee')
df_encoded = encoder.transform(df_indexed)
```

Spark ML en pratique : VectorAssembler

- Dans Spark ML, pour pouvoir passer des features à un l'algorithme de machine Learning, il est nécessaire de rassembler celles-ci au sein d'une seule et même colonne.

- Pour réaliser cette opération, on utilise un objet appellé « **VectorAssembler** ». C'est un Transformer.

- Comme pour le StringIndexer ou le OneHotEncoder, il est nécessaire de lui spécifier à minima les paramètres suivants:
  - inputCols : liste de colonnes à assembler.
  - outputCol : nom de la colonne assemblée en sortie.

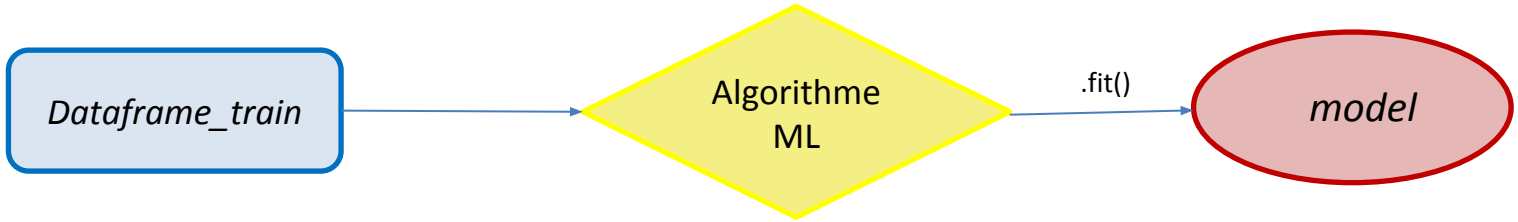
Age	marital_status	Features
37	[0.0 , 1.0 , 0.0]	[37 , [0.0 , 1.0 , 0.0]]
28	[1.0 , 0.0 , 0.0]	[28 , [1.0 , 0.0 , 0.0]]

```
from pyspark.ml.feature import VectorAssembler

# Assemblage des features avec VectorAssembler
feature_assembler = VectorAssembler(inputCols=["age", "marital_status"], outputCol="features")
df_final = feature_assembler.transform(df_encoded)
```

Spark ML en pratique : Training

- Dans Spark ML, chaque algorithme est relié à une classe.
- Les algorithmes de machine Learning Spark ML sont tous des estimators. Ils possèdent donc une méthode « **.fit()** » qui va permettre d'apprendre le dataset de training.
- En sortie, la méthode « **.fit()** » retourne un modèle (transformer) qui va permettre de faire des prédictions sur le dataset de test.



Spark ML en pratique : Training

- Pour mettre en place un algorithme de machine Learning, on commence par créer l'estimator. Pour cela, il prend généralement en compte les paramètres suivants:
  - featuresCol : le nom de la colonne features créée précédemment.
  - labelCol (apprentissage supervisé) : nom de la colonne « label ».
  - predictionCol : nom de la colonne qui contiendra les prédictions.
  - des paramètres de tuning (nombre d'itérations, régularisation etc…)

- On peut ensuite appliquer la méthode « .fit() » sur notre dataset de training et ainsi récupérer le modèle de prédiction.

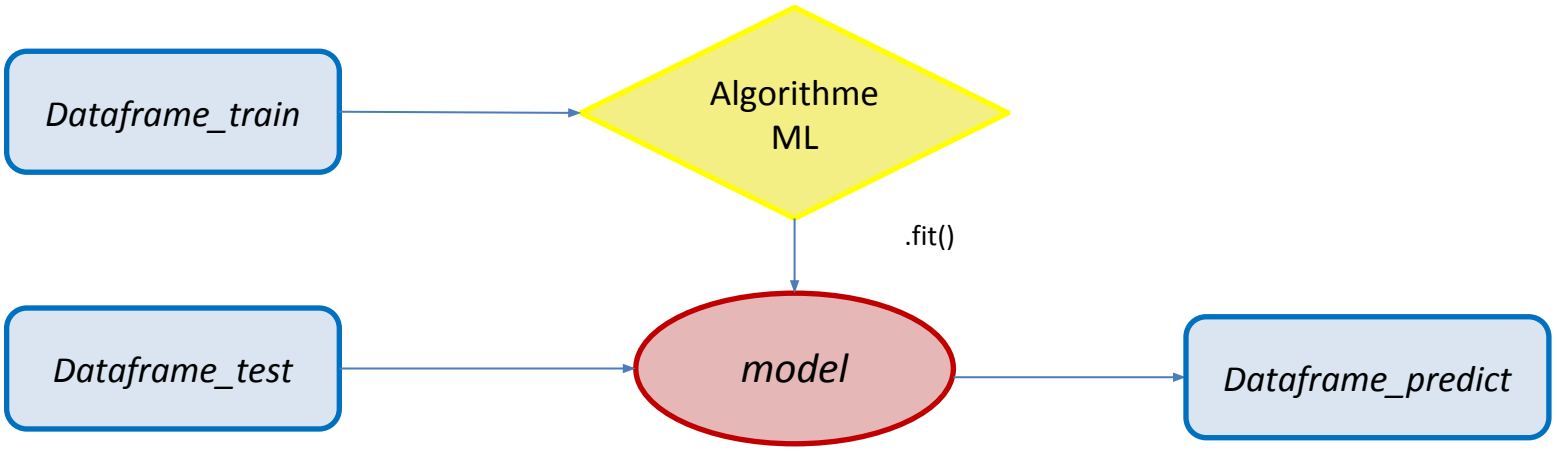
```
from pyspark.ml.classification import LogisticRegression

# Instanciation de l'algorithme
lr = LogisticRegression(featuresCol="colonne_features", labelCol="target_indexed",predictionCol="colonne_prediction",
                        maxIter=200, elasticNetParam=0.1)

# Training
lr_model = lr.fit(df_train)
```

Spark ML en pratique : Prédictions

- Le model est un transformer issu de l'apprentissage d'un algorithme de machine Learning.
- Il possède ainsi une méthode « .**transform()** » qui va transformer le dataframe en lui ajoutant une colonne prédiction.
- **Note :** Il ne prend pas de paramètres, ceux-ci ont été assignés lors de l'instanciation de l'estimator algorithme.



Spark ML en pratique : Prédictions

- Maintenant que nous avons créé un modèle de prédiction avec la méthode « **.fit()** », il est possible d'effectuer des prédictions avec ce modèle.

- Pour cela, on utilise sa méthode « **.transform()** ».

```
from pyspark.ml.classification import LogisticRegression

# Instanciation de l'algorithme
lr = LogisticRegression(featuresCol="colonne_features", labelCol="target_indexed",predictionCol="colonne_prediction",
                        maxIter=200, elasticNetParam=0.1)

# Training
lr_model = lr.fit(df_train)

# Predicting
df_pred = lr_model.transform(df_test)
```

Spark ML en pratique : Evaluation

- Lors de la mise en place d'un algorithme de machine Learning, il est essentiel et nécessaire d' évaluer son modèle pour connaître sa capacité de prédiction.

- Pour évaluer un modèle, on instancie un objet de type **Evaluator**. On utilise ensuite la méthode « **.evaluate()** » pour obtenir la performance du modèle.

- Dans Spark ML, plusieurs evaluators sont implémentés:
  - RegressionEvaluator : Pour les problèmes de régression.
  - BinaryClassificationEvaluator : Pour les problèmes de classification.
  - …

- **Note :** Généralement, lors de l'instanciation d'un evaluator, on peut / doit lui spécifier les paramètres suivants:
  - rawPredictionCol: colonne des prédictions.
  - labelCol : colonne des données réelles.
  - metricName : le nom de la métrique à utiliser (mse, rmse, areaUnderROC, areaUnderPR…)

Spark ML en pratique : Evaluation

- Exemple pour une classification :

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Instanciation d'un evaluator de classification
evaluator = BinaryClassificationEvaluator(rawPredictionCol='prediction',
labelCol='label', metricName='areaUnderROC')

# Evaluation des prédictions sur le test set
evaluator.evaluate(df_test_pred)
```

- Exemple pour une régression :

```
from pyspark.ml.evaluation import RegressionEvaluator

# Instanciation d'un evaluator de regression
evaluator = RegressionEvaluator(predictionCol='prediction',
labelCol='prix', metricName='rmse')

# Evaluation du modèle
evaluator.evaluate(df_test_pred)
```

Spark ML en pratique : les pipelines

- Le pipeline est un objet Spark ML permettant d'enchaîner des traitements de données de manière simple afin de gagner en fluidité en lisibilité dans le code.
- On les utilise bien souvent lorsque de nombreuses opérations sont menées sur les datasets.
- Les pipelines sont des estimators. Ils possèdent donc une méthode « **.fit()** » qui va retourner un model.
- Pour les utiliser, il faut leur préciser les paramètres suivants:
  - Stages : une liste chronologique d'estimators et transformer à appliquer sur le dataframe

Spark ML en pratique : les pipelines

- Un pipeline est un estimator. Il s'utilise de la même façon que tout autre estimateur (exemple : StringIndexer…)

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
```

Live Coding !

- Chargement des données :
  - Création du SparkContext et du SQLContext.
  - Chargement des données.
  - Conversion des colonnes au bon typage.
  - Recherche et gestion des valeurs nulles.

- Mise en place d'un Vectorizer :
  - Tokenization des données.
  - Vectorisation des données.

- Premier modèle de machine learning :
  - Séparation des données en un jeu d'apprentissage et un jeu de test.
  - Utilisation d'un modèle Naive Bayes.
  - Evaluation de la précision du modèle.

- Un modèle plus avancé :
  - Utilisation de nouvelles features.
  - Utilisation d'un SVM à kernel linéaire.

Fonctionnalités Spark ML : Version 2.2 (1)

- Algorithmes supervisés de classification (*pyspark.ml.classification)* :
  - Régression logistique régularisée.
  - SVM avec kernel linéaire (SVC).
  - Naive Bayes.
  - Arbres de décisions, Random Forest, Gradient Boosting Regressor.
  - Perceptron multi-couches.
  - Solutions aux problèmes de classification multi-classes.

- Algorithmes supervisés de régression (*pyspark.ml.regression*):
  - Régression (linéaire / généralisée) et régularisée.
  - Régression isotonique.
  - Arbres de régressions, Random Forest Regressor, Gradient Boosting Regressor.
  - Modèles de survie - Accelerated Failure Time (AFT).

Fonctionnalités Spark ML : Version 2.2 (2)

- Algorithmes non-supervisés de clustering (*pyspark.ml.clustering)* :
  - KMeans.
  - Bisecting K-means.
  - Gaussian Mixture.
  - Latent Dirichlet Allocation (LDA).

- Algorithmes de recommandation (*pyspark.ml.recommendation*):
  - Alternating Least Squares (ALS).

- Fonctionnalités statistiques (*pyspark.ml.stat)*:
  - Test du Chi2 (indépendance de variables).
  - Calcul de corrélation entre deux variables (Pearson et Spearman).

- Fonctionnalités d'évaluation de modèles (*pyspark.ml.evaluation)*:
  - Métriques de classification binaires et multi-classes.
  - Métriques de régression.

Fonctionnalités Spark ML : Version 2.2 (3)

- Fonctionnalités de features engineering (*pyspark.ml.feature*):
  - Binarisation de variable, indexation de variables catégorielles, one-hot-encoding…
  - Discrétisation de features (bucketing).
  - Complétion de valeurs nulles ou manquantes.
  - Fonctionnalités de rescaling, fonctionnalités de normalisation.
  - Count Vectorizer (Hashing TF), Tf-Idf Vectorizer (Hashing TF et Idf).
  - Traitement de texte (stop words, tokenization, …)
  - Word2Vec.
  - Sélection de modèles.
  - Analyse en Composantes Principales (ACP).
  - …

- Fonctionnalités de Tuning (*pyspark.ml.tuning*):
  - Train-Test Split.
  - Cross-validation.
  - Hyper Tuning de paramètres (pipelines)

Conclusion

- Au cours de l'unité, nous avons évoqué la plupart des fonctionnalités du framework Spark.

- Spark possède deux librairies relatives à la manipulation et l'exploitation de données :
  - La première, **Spark Core**, est la brique centrale de Spark. Elle repose sur la manipulation de RDD (Resilient Distributed Datasets), objets assimilables à des "listes de listes" en python.
  - Le deuxième, **Spark SQL**, s'appuie sur le Spark Core afin de proposer un niveau d'abstraction et de structuration des données supplémentaire. On travaille ainsi sur des objets de type dataframe et on effectue les traitements colonne par colonne.

- Concernant le machine learning, Spark possède là encore deux librairies possibles.
  - **La librairie MLlib**, historique, qui effectue des traitements de machine learning parallélisés sur des objets de type RDD. La librairie est maintenant dépréciée à cause de son manque de souplesse et d'ergonomie.
  - **La librairie Spark ML**, apparue depuis la version 1.2 et dont l'ergonomie est très proche de scikit-learn. Spark ML travaille directement sur les dataframes Spark SQL et est à l'heure actuelle, une des solutions les plus utilisées pour le machine learning en contexte Big Data.