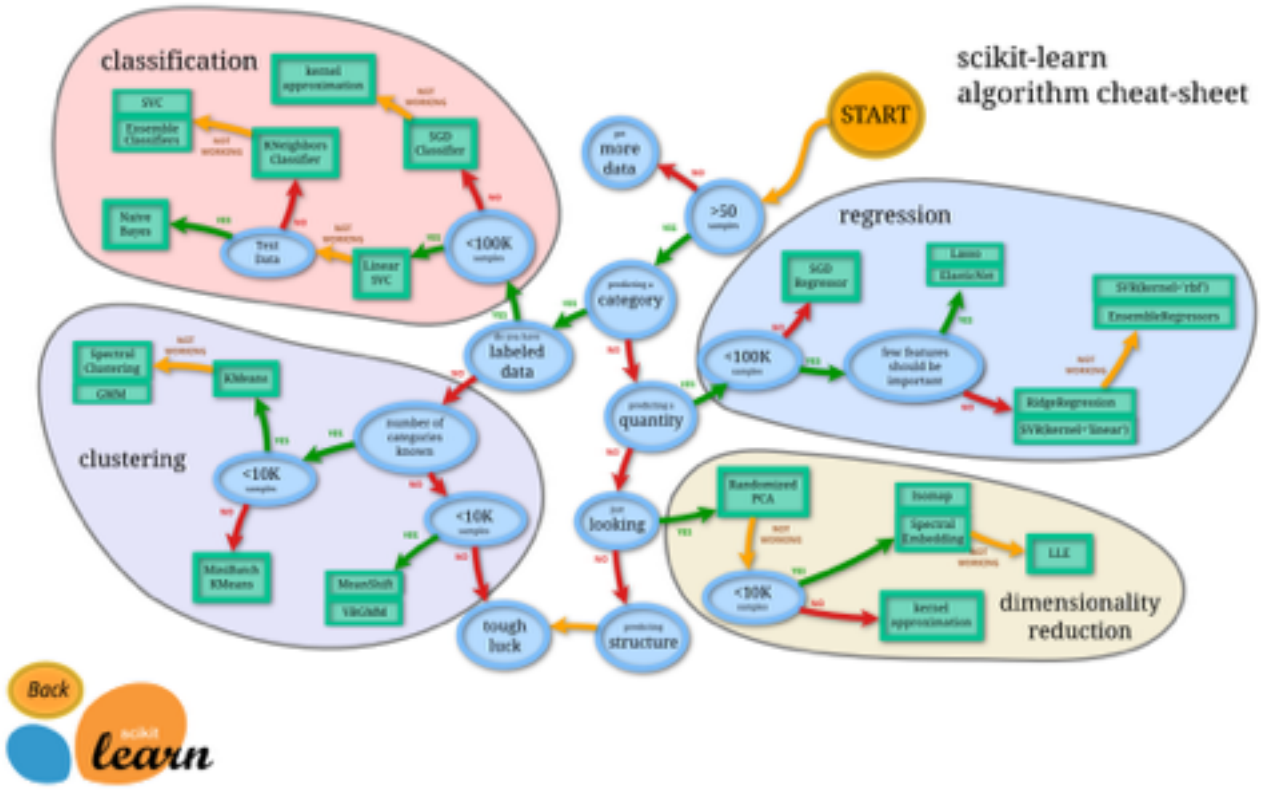
API design for machine learning software: experiences from the scikit-learn project

Sckit-learn 이란?

- Python 프로그래밍 언어를 위한 open source ML Library
- Machine Learning 관련 Library 중 가장 보편적으로 사용
- Python으로 작성 (일부는 Cython)
- 다양한 상황에서 재사용 할 수 있도록 interface 설계



1 Introduction

sklearn은 numeric 형의 패키지(Numpy)와 과학 관련 패키지(Scipy) 와 연결되도록 설계
Numpy는 연속적인 numeric array datatype을 이용해 빠른 computing array 연산을 할 수 있음
Scipy는 Python/Numpy에서 구현하거나 C/C++ 구현을 wrapping하여 이를 일반적인 숫자 연산으로 확장
이러한 Stack을 기반으로 Scipy를 도메인 별 툴킷으로 보완하기 위해 scikits이라는 일련의 라이브러리가 만들어짐
그리하여 2007 년에 시작된 scikit-learn 프로젝트는 다양한 분야의 연구원 (컴퓨터 과학, 신경 과학, 천체 물리학 등)의 12 명 이상의 핵심 개발자로 구성된 국제 팀에 의해 개발됨

Scikit-learn 프로젝트의 목표

1. 비전문가가 쉽게 접근
2. 다양한 과학적 영역에 효율적이고 잘 만들어진 머신러닝 도구를 제공

전통적인 러닝 알고리즘, model metric 및 selection 툴과 preprocessing 절차 등을 지원
sklearn을 사용하려면 기본적인 Python 프로그래밍 지식이 필요

2 Core API

Scikit-learn 내의 모든 객체는 3개의 보완적인 interface로 구성된 기본 common API를 가지고 있음

1. **모델 생성 및 fitting을 위한 estimator interface**
2. **예측하기 위한 predictor interface**
3. **데이터를 변환하기 위한 transformer interface**

2.1 General principles

간단한 규칙만을 정하고, 객체가 구현해야만 하는 메소드의 수를 최소한으로 제한
sklearn API는 다음과 같은 원칙을 준수하도록 설계

Consistency (일관성)

- 모든 객체는 일관된 interface를 공유
- 이 interface는 모든 객체에 대해 일관된 방식으로 문서화

Inspection (검사)

- 생성자 매개 변수 및 매개 변수 값은 공용 속성으로 표시

Non-proliferation of classes (클래스의 비확산 / 제한된 객체 계층 구조)

- 알고리즘만 파이썬 클래스로 표현가능한 유일한 객체
- 데이터 세트는 표준 형식(Numpy 배열, Scipy sparse 행렬)으로 표현
- 매개 변수 이름은 표준 파이썬 문자열을 사용

Composition (구성)

- 다양한 머신러닝 작업을 시퀀스 또는 데이터 변환 조합으로 표현
- 일부 작업은 다른 알고리즘을 매개변수화 한 메타 알고리즘으로 제작 가능

Sensible defaults (합리적인 기본값)

- 모델에 사용자 지정 매개변수가 필요한 경우 라이브러리는 적절한 기본값을 정의
- 작업에 대한 baseline solution을 제공

2.2 Data representation

sklearn에서는 가능한 한 matrix 표현에 가까운 data representation을 사용

- 밀도가 높은 데이터 -> NumPy의 multi-dimension array
- 희소 데이터 -> SciPy의 sparse matrix

효율적인 NumPy 및 SciPy 벡터화 연산에 의존 할 수 있는 이점이 있음 + 코드를 짧고 읽기 쉽게 유지
text data의 경우 두 라이브러리를 위용해 효율적으로 vectorization 가능
하나의 Sample이 아닌 Batch Sample 차리에 최적화
Batch 처리는 Python 함수 호출 고유의 오버 헤드를 방지할 수 있음

2.3 Estimators

Estimator interface는 sklearn 라이브러리의 핵심
객체의 인스턴스화 메커니즘을 정의하고 훈련 데이터에서 모델을 학습하기위한 적합한 방법을 제공
모든 알고리즘은 이 Estimator interface에서 구현되는 객체로 제공

Estimator 초기화와 실제 학습은 partial function application과 같은 방식으로 분리
Estimator의 생성자는 실제 데이터를 보거나 실제 학습을 수행하지 않으며, 주어진 매개변수를 객체에 연결
실제 학습은 fit 메소드로 수행 (training data와 함께 호출) -> 항상 호출 된 estimator object를 리턴
이후 입력되는 데이터의 predictions 또는 transformations을 수행하는 데 사용
partial application 관점에서 fit은 데이터(X)에서 해당 데이터의 모델링 결과(Y)까지의 함수

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

? 왜 fit과 (prediction or transformation)을 분리한 걸까 ?

- 현재 single object가 estimator 및 model로 이중 목적을 수행하고 있다고 볼 수 있음
- user 관점에서, 2 개의 결합 된 인스턴스를 갖는 것은 사용이 번거로우며,
  특히 newcomers이 인스턴스를 혼동하여 사용할 가능성이 높아짐
- developer 관점에서는 인스턴스를 분리하면 병렬 클래스 계층 구조가 생성되고
  프로젝트의 전체 유지 관리 복잡성이 증가
- 이러한 이유로, decoupling estimators from models을 진행하지 않았음

모든 Estimator는 동일한 interface를 공유 -> 생성자를 바꾸면 다른 학습 알고리즘 사용 가능
Estimator는 기본 ML 말고도 preprocessing, feature extraction 구현 가능
이러한 Estimator를 사용하는 Design Pattern은 General Principles을 준수할 수 있는 유용한 패턴

2.4 Predictors

predictor interface는 estimator의 학습 된 매개 변수를 기반으로 X_test에 대한 예측을 생성하는 방법을 추가함
-> Estimator 개념을 확장한 것이며, 예측은 새로운 데이터로 일반화 됨

Supervised learning estimators의 경우 일반적으로 모델에 의해 예측된 레이블 반환
Unsupervised learning estimators도 predict interface를 구현할 수 있으며, K-means의 경우 X_test에 대한 군집 레이블 (integer indices) 반환
일부 predictors는 클래스 확률(class probabilities)을 반환하는 예측 proba 방법도 제공
또한 predictors는 입력 데이터 batch에 대한 성과를 평가하기 위해 score 함수를 제공함

- Regression에서 y_test와 y_pred 사이의 결정 계수(coefficient of determination)
- Classification에서 accuracy
- Unsupervised estimators는 주어진 데이터의 likelihood을 계산하기 위해 score 함수를 노출 가능

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=10)
km.fit(X_train)
clust_pred = km.predict(X_test)
```

2.5 Transformers

일부 Estimators는 데이터 수정 및 필터링을 위해 transformer interface를 구현
새로운 데이터 X_test를 입력으로 사용하고 변환 된 버전의 X_test를 출력으로 생성
아래 알고리즘은 모두 라이브러리 내에 transformers로 제공

- Preprocessing
- Feature selection
- Feature extraction
- Dimensionality reduction

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
```

메소드 체인을 적용하여 한줄로 작성 가능
fit(X_train) .transform(X_train)을 fit_transform(X_train)으로 작성하여 반복계산을 방지할 수 있음

```
X_train = StandardScaler().fit(X_train).transform(X_train)
```

```
X = vectorizer.fit_transform(corpus)
```

3 Advanced API

Core API를 기반으로 meta-estimators 구축, 복잡한 estimators 구성 및 모델 선택을 위한 advanced API 메커니즘이 존재함

3.1 Meta-estimators

일부 머신 러닝 알고리즘은 간단한 알고리즘으로 매개 변수화 된(parametrized) meta-algorithms으로 자연스럽게 표현 가능 -> sklearn에서 meta-estimators로 구현
(EX. ensemble methods, multi-class 및 multi-label 분류 체계)
meta-estimators은 기존의 base estimator을 입력으로 사용하여 학습 및 예측에 내부적으로 사용

3.2 Pipelines and feature unions

sklearn API의 두드러진 특징은 여러 base estimators에서 new estimators를 작성할 수 있다는 것
Composition 메커니즘을 사용하여 ML workflow를 단일 estimator 객체로 결합 할 수 있으며 일반적인 estimator가 사용될 수 있는 모든 곳에서 사용할 수 있음
특히, sklearn의 model selection routines을 composite estimators에 적용 할 수 있으므로 복잡한 workflow에서 모든 매개 변수를 전체적으로 최적화 할 수 있음

composite estimator를 구성할 수 있는 방법

1. Pipeline objects

- sequentially하게 수행
- pipe의 마지막 estimator가 pileline의 역할 정의
  (마지막 estimator가 predictor 인 경우 파이프 라인 자체를 predictor)

2. FeatureUnion objects

- parallel하게 수행
- 여러 transformers를 하나의 transformers로 결합하여 출력을 연결

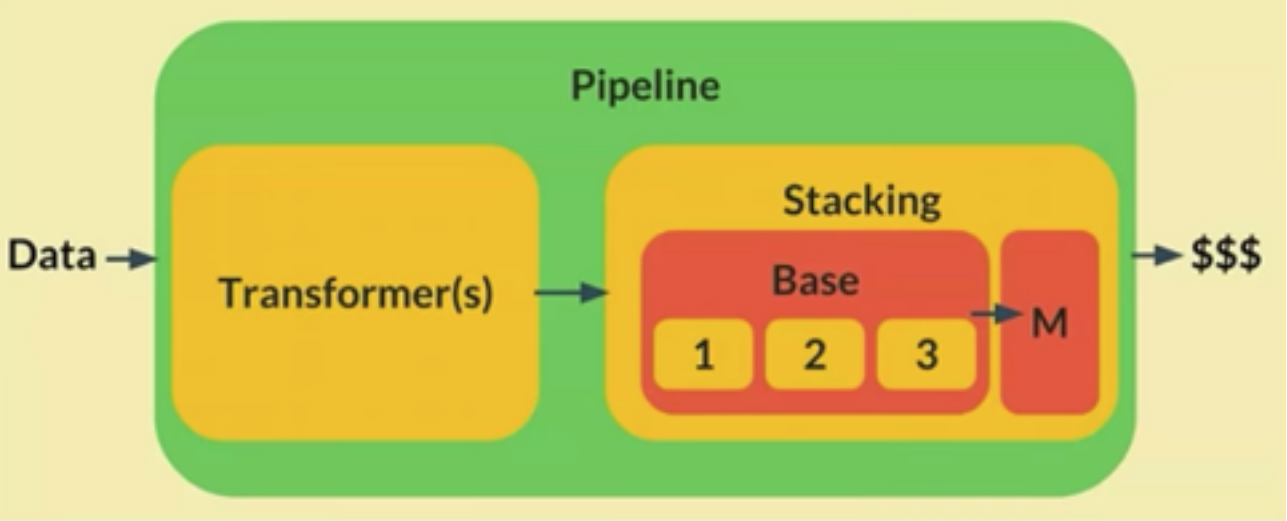
Pileline과 FeatureUnion을 결합하여 복잡하고 중첩 된 workflow를 만들 수 있음

```
from sklearn.pipeline import FeatureUnion, Pipeline
from sklearn.decomposition import PCA, KernelPCA
from sklearn.feature_selection import SelectKBest

union = FeatureUnion([("pca", PCA()),
                      ("kpca", KernelPCA(kernel="rbf"))])

Pipeline([("feat_union", union),
          ("feat_sel", SelectKBest(k=10)),
          ("log_reg", LogisticRegression(penalty="l2"))
         ]).fit(X_train, y_train).predict(X_test)
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

train_X, test_X, train_y, test_y = train_test_split(X, y)

est_list = [('scaler', StandardScaler()),
            ('logistic', LogisticRegression())]
pipe = Pipeline(est_list)
pipe.fit(train_X, train_y)
scores=pipe.predict(test_x)
```

```
from sklearn.ensemble import GradientBoostingClassifier

def score_iris(est):
    X, y = load_iris(return_X_y=True)
    train_X, test_X, train_y, test_y = train_test_split(X, y)

    est_list = [('scaler', StandardScaler()),
                ('your_estimator', est)]
    pipe = Pipeline(est_list)
    pipe.fit(train_X, train_y)
    scores=pipe.predict(test_X)
    return pipe, scores

gbt = GradientBoostingClassifier(n_estimators=50)
pipe, scores = score_iris(gbt)
```

```
from civismlext.stacking import StackedClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.datasets import load_iris

iris_x, iris_y = load_iris(return_X_y=True)
est_list = [('logistic', LogisticRegression()),
            ('rf', RandomForestClassifier()),
            ('gbt', GradientBoostingClassifier()),
            ('meta', LogisticRegression())]

stacker = StackedClassifier(est_list)
stacker.fit(iris_x, iris_y)
scores = stacker.predict(iris_x)
```



3.3 Model selection

estimators의 생성자에 설정된 hyper-parameters는 학습 알고리즘의 동작을 결정하고,
결과적으로 보이지 않는 데이터에 대한 결과 모델의 성능을 결정
결국 model selection의 목적은 주어진 hyper-parameter space와 관련하여best hyper-parameter combination을 찾는 것

sklearn에서 제공하는 두 개의 model selection meta-estimators

1. GridSearchCV

- 주어진 hyper-parameters list의 grid한 조합(cartesian product)를 열거

2. RandomizedSearchCV

- 매개변수 분포에서 고정된 횟수를 sampling하여 grid search에서 조합 폭발을 회피

model selection 알고리즘에서 선택적으로 다양한 cross-validation scheme와 score function을 제공

- k-fold, stratified k-fold
- accuracy, AUC, F1 score for classification, R2 score, MSE for regression

```
from sklearn.grid_search import GridSearchCV
from sklearn.svm import SVC

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

clf = GridSearchCV(SVC(), param_grid, scoring="f1", cv=10)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

3.4 Extending scikit-learn

용이한 코드 재사용 및 구현을 단순화하며 불필요한 클래스의 도입을 막기 위해,
duck typing의 Python 원칙은 코드베이스 전체에서 활용 -> 확장성과 유연성 제공
(Pipelines & grid search -> duck typing 적용)
위에 제시된 API의 General Principles을 준수한다면, Custom estimator를 만들어 사용 가능

```
class BaseEstimator:
    """Base class for all estimators in scikit-learn
```

```
class KMeans(BaseEstimator, ClusterMixin, TransformerMixin):
    """K-Means clustering
```

```
class CountVectorizer(BaseEstimator, VectorizerMixin):
    """Convert a collection of text documents to a matrix of token counts
```

```
class LinearModel(BaseEstimator, metaclass=ABCMeta):
    """Base class for Linear Models"""
```

```
class MeanClassifier(BaseEstimator, ClassifierMixin):
    """An example of classifier"""
```

```
from sklearn.grid_search import GridSearchCV

X_train = [i for i in range(0, 100, 5)]
X_test = [i + 3 for i in range(-5, 95, 5)]
tuned_params = {"intValue" : [-10,-1,0,1,10]}

gs = GridSearchCV(MeanClassifier(), tuned_params)

# for some reason I have to pass y with same shape
# otherwise gridsearch throws an error. Not sure why.
gs.fit(X_test, y=[1 for i in range(20)])

gs.best_params_ # {'intValue': -10} # and that is what we expect :)
```

다른 toolkit 및 context에서 재사용 할 수 있도록,
sklearn을 사용하는 user code는 sklearn 라이브러리에 연결되어서는 안됨
("바나나"를 원하는 사용자는"바나나와 정글 전체를 들고있는 고릴라"를 얻지 않아야합니다!)