Case Study: Scikit-Learn

Wanxuan Chen

The Scikit-Learn also called Sklearn and that is an open source machine learning tool base on the python language. Using the Sklearn with python, the users can easily and efficiently mining and process the data. Also the Sklearn can build on various environemnt. Python is a really common high-performance program language today, So it will be very easy to use it access the Sklearn project with Python language. If you want to begin a small machine learning project and need to pre-process the data, the sklearn will be a perfect tool for developer.

The building environment of sklearn require Python and some Python libraries, like Numpy, Pandas, Scipy and Matplotlib.

- Python: python opensource main language
- Numpy: open source library for large, multi-dimensional arrays and matrices operation
- Scipy: open source python library use for scentific computing
- Pandas: it offer data structure and operation for manipulating data. High-performance, easy way yo use the data structure and data analysis tool in python language
- matplotlib: The most common visualize libaray in python. It can process the data and plot quality figure in variety format.

In the Linux system, Installing from source requires the users to have installed the scikit-learn runtime dependencies, Python development headers and a working C/C++ compiler. To build scikit-learn on Windows you need a working C/C++ compiler in addition to numpy, scipy and setuptools. For python version higer than 3.5 ,It also need the build tool Visual Studio 2017.

Pytest is a good test framework for sklearn. The sklearn use the python as a main language, then we can use the pytest as a unit test framework to check. compared with unittest, pytest is more efficient and clear, easy way to use. Coverage.py is one of the most popular code coverage tools for Python. It uses code analysis tools and tracing hooks provided in Python standard library to measure coverage.

The CI platform choices for Sklearn are Travis CI and appveyor. Travis is used for testing on Linux platforms. Appveyor is used for testing on Windows platforms. The CircleCI is used to build the docs for viewing, for linting with flake8, and for testing with PyPy on Linux.

The Sklearn is a Python library and can easily use after installed. In the Sklearn library, there are many functions and Classes API, so that the developers can use API to complete all the function in Sklearn library. If the external project need to use the sklearn, the developers can write a sklearn module and integration after passed the pytest.

When the developer begin to use the Sklearn, they often hard to choose the right estimator for their job, because different estimator were built for specific types problem. The official document provide this flowchart to the users and provide appropriate method for developer.

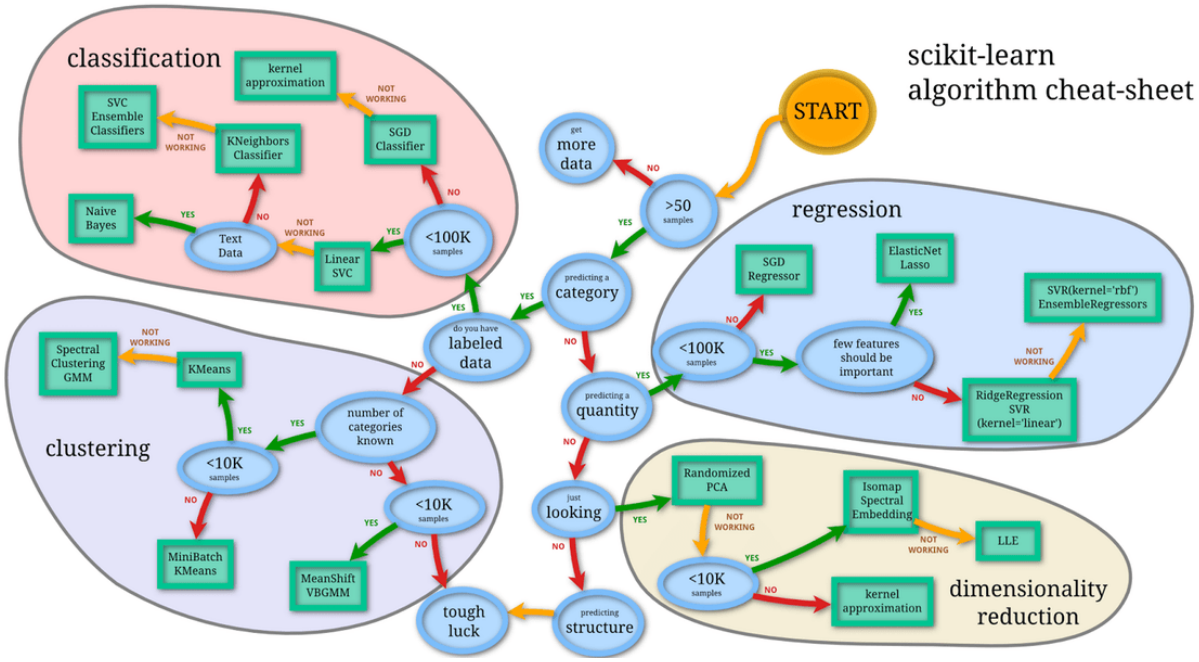


Figure. Sklearn Algorithm Flowchart

The machine learning problem can be divided to few categories, supervised learning and unsupervised learning. There are some main model in the Sklearn library:

- Classification:
  - SVC
  - Nearest Neighbors
  - Decision Tree
  - Random Forest
- Regression:
  - Linear Regression
  - Polynomial Regression
- Clustering:
  - K-mean

    - Spectral Clustering
- Dimensionality Reduction:
    - It can reduce the sample dimension
    - PCA
    - ICA
- Model Selection and Evaluation:
    - Cross-Validation
    - Grid Search
- Pre-Process
    - Normalization
    - Standardizatiopn
    - Mean Removal

There is feature request in the github issue is that adding a function to show the feature importances with trees. For adding this feature, the developer just added some new function on it.

This the pull request to add the feature.

```
+     def feature_importances(self):
+         """Compute the feature importances of all features.
+
+         The importance I of a feature is computed as the (normalized) total
+         reduction of error brought by that feature.
+
+         .. math::
+
+             I(f) = \sum_{nodes A for which f is used} n_samples(A) * \Delta err
+
+         Returns
+         -------
+         importances : array of shape = [n_features]
+             The feature importances.
+         """
+         tree = self.tree_
+         importances = np.zeros(self.n_features_)
+
+         for node in xrange(tree.node_count):
+             if tree.children[node, 0] == tree.children[node, 1] == Tree.LEAF:
+                 continue
+
+             else:
+                 importances[tree.feature[node]] += \
+                     tree.n_samples[node] * (tree.init_error[node] -
+                                             tree.best_error[node])
+
+         importances /= np.sum(importances)
+
+         return importances
+
```

The feature used to show the feature importance in decision tree model and that can provide a directly figure for users.

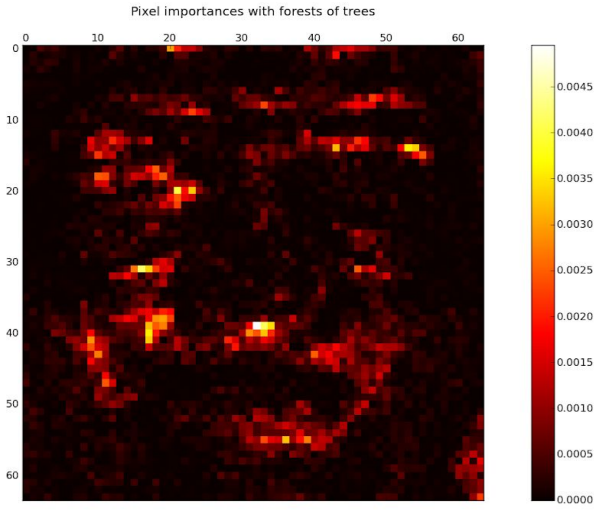


Figure. Pixel Importance With Forest of Trees

The another defects is adding a additional ‘TimeSeriesSplit’ Functionality. This function can provide a more flexible function to deal with financial data.The additional test_size parameter allows users to control the number of samples used for each test split and use all remaining samples for training. This feature only need to modify the function and add a parameter to achieve it.

```
>>> x = np.arange(252 * 4)
>>> cv = TimeSeriesSplit(n_splits=4, max_train_size=504)
>>> for train_index, test_index in cv.split(x):
...     print("Train Size:", len(train_index), "Test Size:", len(test_index))
Train Size: 204 Test Size: 201
Train Size: 405 Test Size: 201
Train Size: 504 Test Size: 201
Train Size: 504 Test Size: 201
```

Figure.Before Add test_size.

```
>>> x = np.arange(252 * 4)
>>> cv = TimeSeriesSplit3(n_splits=4, max_train_size=504, test_size=126)
>>> for train_index, test_index in cv.split(x):
....     print("Train Size:", len(train_index), "Test Size:", len(test_index))
Train Size: 504 Test Size: 126
Train Size: 504 Test Size: 126
Train Size: 504 Test Size: 126
Train Size: 504 Test Size: 126
```

Figure. Feature added

Demo:

```
from sklearn.datasets import load_breast_cancer
from sklearn.neighbors import KNeighborsClassifier      #KNN
from sklearn.linear_model import LogisticRegression     #Logistic Regression
from sklearn.tree import DecisionTreeClassifier         #Decision Tree
from sklearn.ensemble import RandomForestClassifier     #Random Forest
from sklearn.neural_network import MLPClassifier        #Neural Network
from sklearn.svm import SVC                             #SVM
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import export_graphviz
import matplotlib.pylab as plt
import numpy as np
import graphviz



#load the breast cancer data from sklearn and display the description of dataset
cancer = load_breast_cancer()
# print(cancer.DESCR)
print(cancer.feature_names)
# print(cancer.target_names)
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
```

'concave points error' 'symmetry error' 'fractal dimension error'

'worst radius' 'worst texture' 'worst perimeter' 'worst area'

'worst smoothness' 'worst compactness' 'worst concavity'

'worst concave points' 'worst symmetry' 'worst fractal dimension']

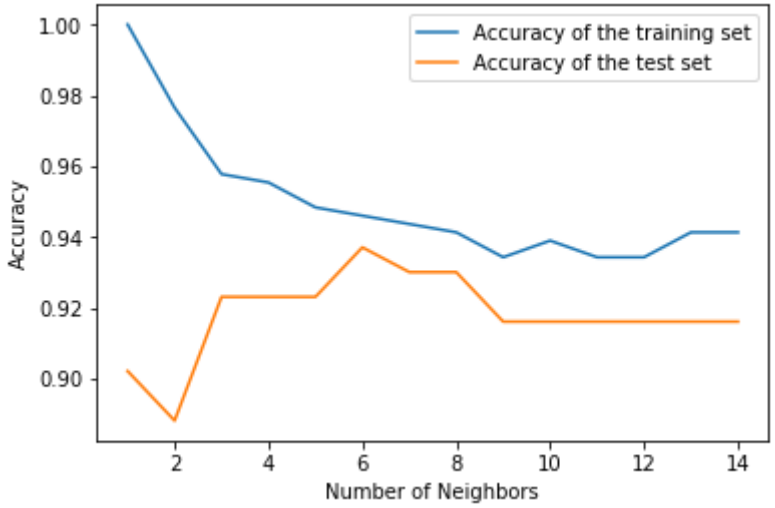
In [8]:

```
# ----------KNN Classifier
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []

# try KNN for diffrent k nearest neighbor from 1 to 15
neighbors_setting = range(1, 15)

for n_neighbors in neighbors_setting:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)
    training_accuracy.append(knn.score(X_train, y_train))
    test_accuracy.append(knn.score(X_test, y_test))

plt.plot(neighbors_setting, training_accuracy, label='Accuracy of the training set')
plt.plot(neighbors_setting, test_accuracy, label='Accuracy of the test set')
plt.ylabel('Accuracy')
plt.xlabel('Number of Neighbors')
plt.legend()
```

```
print("Accuracy of the training set for 6NN: {:3f}".format(training_accuracy[5]))

print("Accuracy of the test set for 6NN: {:3f}".format(test_accuracy[5]))

# From the plot can observe the neighbor 6 has best result
```

Accuracy of the training set for 6NN: 0.946009

Accuracy of the test set for 6NN: 0.937063



In [13]:

```
print("Accuracy of the training set for 6NN: {:3f}".format(training_accuracy[5]))

print("Accuracy of the test set for 6NN: {:3f}".format(test_accuracy[5]))
```

Accuracy of the training set for 6NN: 0.946009

Accuracy of the test set for 6NN: 0.937063

In [9]:

```
#----------------Logistic Regression(classification)-------------------#

#----binary classification (tumor: benign, malignant)--------#

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,

stratify=cancer.target, random_state=42)
```

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

```
print('Accuracy on the training set: {:.3f}'.format(log_reg.score(X_train,y_train)))
print('Accuracy on the training set: {:.3f}'.format(log_reg.score(X_test,y_test)))
```

```
#It seems as it does better than KNN
```

```
Accuracy on the training set: 0.953
Accuracy on the training set: 0.958
```

In [14]:

```
#Regularization: prevention of overfitting
log_reg100 = LogisticRegression(C=100)
log_reg100.fit(X_train, y_train)
print('Accuracy on the training set after regularization: {:.3f}'.format(log_reg100.score(X_train,
y_train)))
print('Accuracy on the test set after regularization: {:.3f}'.format(log_reg100.score(X_test,
y_test)))
```

```
Accuracy on the training set after regularization: 0.972
Accuracy on the test set after regularization: 0.965
```

In [18]:

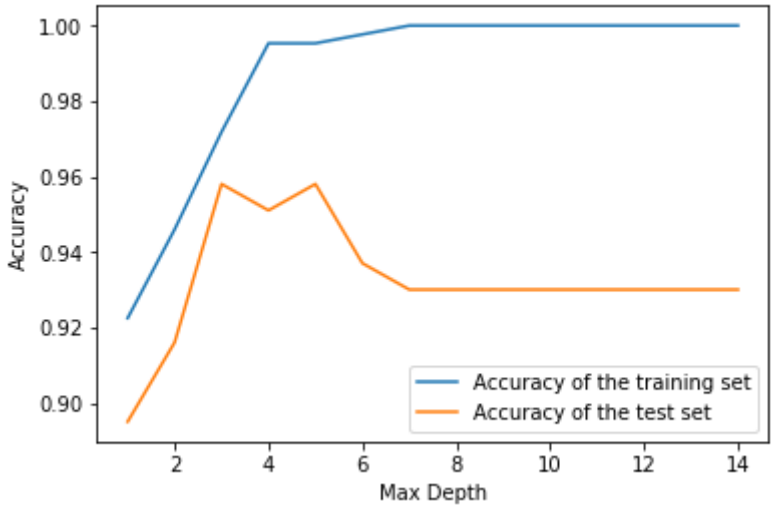
```
#----------------- Decision Tree
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=42)

Tree_training_accuracy = []
```

```
Tree_test_accuracy = []

max_dep = range(1,15)

for md in max_dep:
    tree = DecisionTreeClassifier(max_depth=md,random_state=0)
    tree.fit(X_train,y_train)
    Tree_training_accuracy.append(tree.score(X_train, y_train))
    Tree_test_accuracy.append(tree.score(X_test, y_test))

plt.plot(max_dep,Tree_training_accuracy, label='Accuracy of the training set')
plt.plot(neighbors_setting,Tree_test_accuracy, label='Accuracy of the test set')
plt.ylabel('Accuracy')
plt.xlabel('Max Depth')
plt.legend()

# By having larger max_depth (>5), we overfit the model into training data, so the accuracy for
training set become
# but the accuracy for test set decrease


# by looking at plot, best result accurs when max_depth is 3
print('Accuracy on the training set: {:.3f}'.format(Tree_training_accuracy[2]))
print('Accuracy on the training set: {:.3f}'.format(Tree_test_accuracy[2]))

#This result is similar with the logistic regression
#This method do not need to pre-process data and standard features
```

#Disadvantage is easy to be overfitting

Accuracy on the training set: 0.972

Accuracy on the training set: 0.958



In [16]:

```
print('Feature importances: {}'.format(tree.feature_importances_))

type(tree.feature_importances_)
```

```
Feature importances: [0.         0.         0.         0.         0.         0.
 0.         0.72468105 0.         0.         0.01277192 0.
 0.         0.         0.00826156 0.         0.         0.01702539
 0.         0.         0.05899273 0.12550655 0.00838371 0.03452044
 0.00985664 0.         0.         0.         0.         0.        ]
```

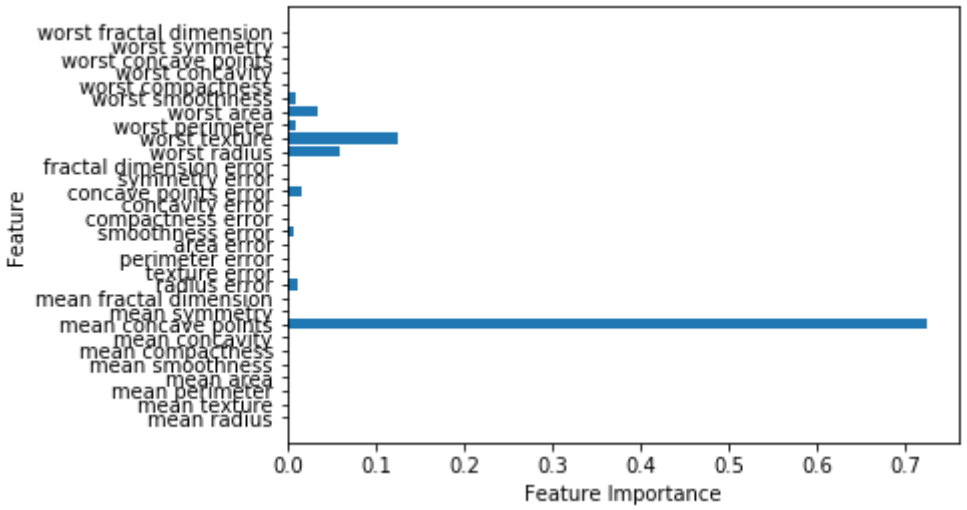
Out[16]:

```
numpy.ndarray
```

In [17]:

```
#Feature Importance

n_feature = cancer.data.shape[1]

plt.barh(range(n_feature), tree.feature_importances_, align='center')
```

```
plt.yticks(np.arange(n_feature), cancer.feature_names)

plt.xlabel('Feature Importance')

plt.ylabel('Feature')

plt.show()
```



In [19]:

```
# ---------- Random Forests

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)


forest = RandomForestClassifier(n_estimators=100, random_state=0)

forest.fit(X_train,y_train)


#you can tune parameter such as:

# - n_job (how many cores)(n_job=-1 => all cores)

# - max_depth

# - max_feature



print('Accuracy on training set data: {:.3f}'.format(forest.score(X_train,y_train)))
```

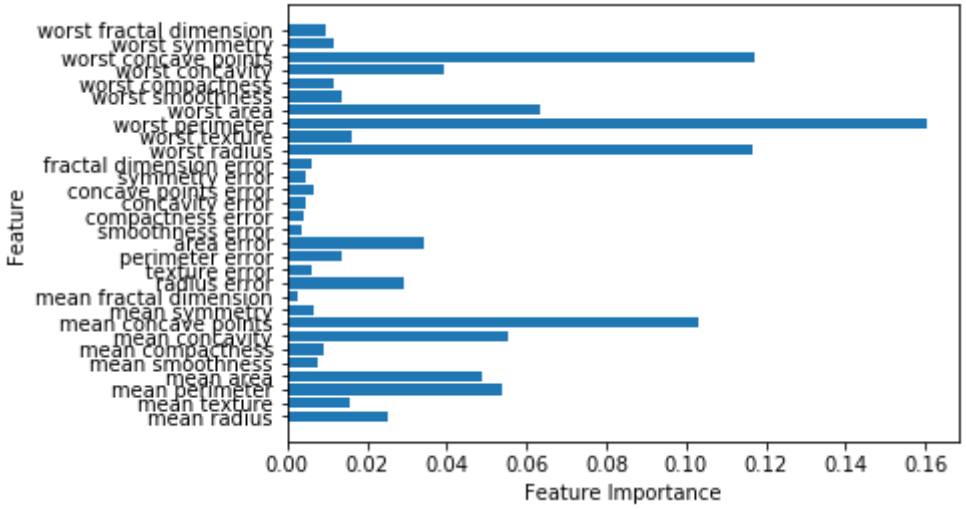
```
print('Accuracy on test set data: {:.3f}'.format(forest.score(X_test,y_test)))
```

acc for training data: 1.000

acc for test data: 0.972

In [20]:

```
#Feature Importance
n_feature = cancer.data.shape[1]
plt.barh(range(n_feature), forest.feature_importances_, align='center')
plt.yticks(np.arange(n_feature), cancer.feature_names)
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
plt.show()
```

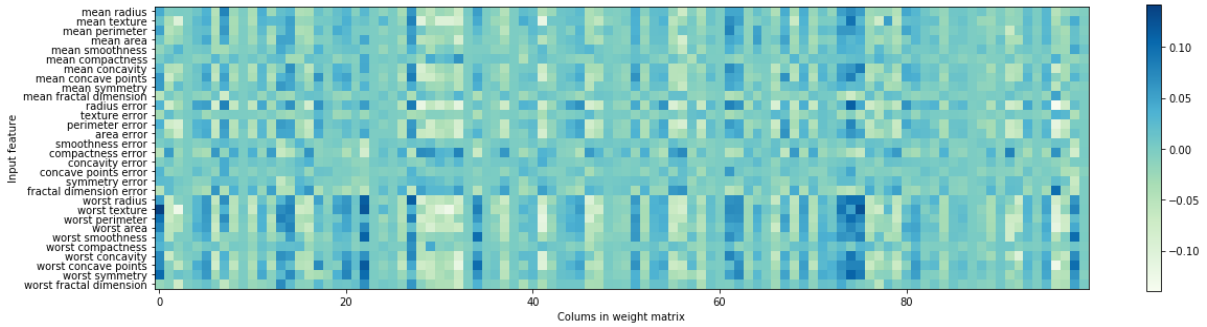


In [23]:

```
# ------------ Neural Network
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit(X_train).transform(X_train)
```

```
X_test_scaled = scaler.fit(X_test).transform(X_test)

mlp.fit(X_train_scaled,y_train)

plt.figure(figsize=(20,5))

plt.imshow(mlp.coefs_[0],interpolation='None',cmap='GnBu')

plt.yticks(range(30),cancer.feature_names)

plt.xlabel('Colums in weight matrix')

plt.ylabel('Input feature')

plt.colorbar()

print('Accuracy on training data: {:.3f}'.format(mlp.score(X_train_scaled, y_train)))

print('Accuracy on test data: {:.3f}'.format(mlp.score(X_test_scaled, y_test)))

# NN can get better result in larger datasets

# we can tune a lot of parameter

# but data may need pre-processing
```

```
Accuracy on training data: 0.988

Accuracy on test data: 0.972
```



In [32]:

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
```

```
min_train = X_train.min(axis=0)
```

```
range_train = (X_train - min_train).max(axis=0)


X_train_scaled = (X_train - min_train)/range_train

X_test_scaled = (X_test - min_train)/range_train


svm = SVC(C=1000)

svm.fit(X_train_scaled, y_train)

print('The accuracy on the training subset: {:.3f}'.format(svm.score(X_train_scaled, y_train)))

print('The accuracy on the test subset: {:.3f}'.format(svm.score(X_test_scaled, y_test)))

#can work well on high-dimensional data with relatively small sample size

#don't perform well on high-dimensional data with many samples (i.e. > 100k)

#preprocessing may be required => implies knowledge and understanding of hyper-parameters

#harder to inspect and visualize
```

```
The accuracy on the training subset: 0.988

The accuracy on the test subset: 0.972
```