INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 9: MACHINE LEARNING AND DECISION TREES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

9. MACHINE LEARNING AND DECISION TREES

BY FRED ANNEXSTEIN

CHAPTER 9 OVERVIEW

In this module we will be covering an introduction to machine learning libraries using python. We focus on the popular scikit-learn or sklearn library and demonstrate several examples of solving classification problems using sklearn's version of decision trees and random tree methods.

CHAPTER 9 OBJECTIVES

By the end of this chapter, you will be able to...

Understand some typical machine learning problems and solution approaches.

Understand the numpy library and ndarray type.

Run a machine learning library using several estimators or models.

Understand the interfaces of the sklearn library.

Recognize and describe the accuracy of machine learning models.

Understand the classification models known as Decision Trees and Random Forests.

Practice hyper-parameter tuning in the context of a lab for random forest tuning.

MACHINE LEARNING PROBLEMS

A machine learning problem typically considers a set of n samples of data, and then tries to predict properties of unknown data. Usually each sample is a simple list of numbers. Each number is the list is associated with a particular attribute or a feature of the problem.

Machine learning problems generally fall into the basic categories of either supervised learning or unsupervised learning. Classification problems are considered supervised learning since they are based on having data that have labeled classes identified. Density estimation and clustering are examples of unsupervised learning and do not depend on having labeled attributes.

When using Python for solving machine learning problems it is typical to store data in what is known as a Numpy array or an ndarray. This data type is popular for doing high-performance data manipulation required by machine learning algorithms. Ndarrys are useful within the context of general python programming, as an alternative to slow list operations.

NUMPY NDARRAY TYPE

We will be using the data type known as ndarrays in this chapter, rather than another sequence type such as a list. In general, in most programming languages, arrays are stored contiguously in memory and are used to efficiently store data of one fixed type of fixed size memory blocks.

The Numpy library defines a class ndarray for creating array objects. Each ndarray objects is used to represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.) Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

For machine learning applications we will use arrays to store table of attribute values (usually numbers), all of the same type, and indexed by a tuple of positive integers. The number of dimensions of the array is called the rank of the array. A tuple of integers giving the size of the array along each dimension is known as shape of the array.

Here is a sample script illustrating the

attributes of 1D and 2D arrays using numpy.ndarray type.

```
import numpy as np

# Creating 1D array object
arr1 = np.array( [ 1.0, 2.0, 3.0, 4.0] )

# Creating array object
arr2 = np.array( [[ 1, 2, 3],
                  [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr1))

# Printing array dimensions (axes)
print("No. of dimensions arr1: ", arr1.ndim)
print("No. of dimensions arr2: ", arr2.ndim)

# Printing shape of array
print("Shape of array arr1: ", arr1.shape)
print("Shape of array arr2: ", arr2.shape)

# Printing size (total number of elements)
of array
print("Size of array: ", arr1.size)
print("Size of array: ", arr2.size)

# Printing type of elements in array
print("Array stores elements of type: ",
arr1.dtype)
print("Array stores elements of type: ",
arr2.dtype)


Output:
Array is of type:  <class 'numpy.ndarray'>
No. of dimensions arr1:  1
No. of dimensions arr2:  2
```

```
Shape of array arr1:  (4,)
Shape of array arr2:  (2, 3)
Size of array:  4
Size of array:  6
Array stores elements of type:  float64
Array stores elements of type:  int64
```

SUPERVISED VS UNSUPERVISED LEARNING

Supervised learning problems have as input a data table with an additional attribute that we want to be able to predict. Supervised machine learning falls into two categories—classification and regression. Solutions to classification problems have a set of discrete possible outputs, which is a prediction of the class any new sample data belongs to.

In solutions to regression problems the desired output consists of one or more continuous variables. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight. Another example would be predicting local temperatures, given a weather time series.

In contrast, for unsupervised learning problems the training data consists of a set samples without any corresponding target values. The goal in such problems may be to discover groups of similar examples within

the data - this is the problem of data clustering. Another example is to determine the distribution of data within the input space — this is the problem of density estimation. Finally, there is the example of transforming the data from a high-dimensional space down to a small number (two or three dimensions) for the purpose of analysis or visualization - this is the problem of dimensionality reduction.

CLASSIFICATION

In a classification problem, samples belong to two or more target classes, and the goal is to learn a model from the already labeled data, which can be used to predict the class of any unlabeled data.Classification is a discrete form of supervised learning where one has a limited number of categories for each of the say n samples provided.

Binary classification uses two classes, such as “true” of “false” for boolean outputs. For example, we may want to learn to label input email messages as “spam” or “not spam” in an email classification application.

Multi-classification uses more than two classes. For example, in the problem of digit image recognition we may have 10 classes

of digit possibilities 0 through 9, which the case in a digit-image dataset.

A multi-classification scheme looking at movie descriptions might try to classify them by genre, such as “action,” “adventure,” “fantasy,” “romance,” “history” and so forth.

TRAINING SET AND TESTING SET

Machine learning is concerned with learning a model to representing some patterns or properties of a data set, and then testing those properties against the model using another data set. A common practice is to evaluate an algorithm by splitting a given learning data set into two. We call one of those sets the training set, on which we set to learn some properties. The other set we call the testing set, on which we run test to see if the model accurately predicts the properties, hence producing an accuracy score. The next examples should help clarify the use of these sets.

SCIKIT-LEARN

Scikit-learn or simply sklearn has many python library of modules for many machine learning tasks. In this chapter we will begin to explore the library’s features for various

classification, regression and clustering algorithms.

The sklearn modules are designed to interoperate with standard numerical and scientific python libraries, specifically numpy and scipy. Sklearn is largely written in python, and uses numpy extensively for high-performance linear algebra and array operations.

If you need to install scikit-learn, you can do so using the following command.

```
$ conda install -c anaconda scikit-learn
```

In this introductory chapter we will demonstrate learning experiments with sci-kit learn's tree-based models known as *decision tree classifiers*. This classifier is ready to use right out of the box when using this sklearn package.

SKLEARN DATASETS

The sklearn software packages comes with a few small standard datasets that do not require you to download files from some external website. These datasets come with a small number of samples with a limited number of features, which makes it easy to experiment with. Using these data sets will

help you to understand how sklearn expects data to be named and shaped. The data sets can also be found at the following link: https://scikit-learn.org/stable/datasets/toy_dataset.html

At the end of the chapter will be links to information on loading external data sets when using sklearn.

LOADING AN EXAMPLE DATASET FROM SKLEARN

In this example we show how to load the classic dataset with sampled data about Iris flowers containing flower names and measurements of the organs of mature flowers. This data set will show us how a solution to a basic classification problem is solved using sklearn. We first input the data for our program.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
```

The call of the function `datasets.load_iris()` creates and returns a dictionary like object called a Bunch object, as we can see from using the type() command.

```
>>> type(iris)
sklearn.utils.Bunch
```

Within the Bunch is the data (measurements), the target values (referencing flower names) and various metadata, as we can see using the dir() command.

```
>>> dir(iris)
['DESCR',
 'data',
 'data_module',
 'feature_names',
 'filename',
 'frame',
 'target',
 ‘target_names']
```

The `.data` member of the iris Bunch is simply a 2D numpy ndarray with shape pair determined by the number of samples, and the number of attributes.

```
>>> type(iris.data)
numpy.ndarray
>>> iris.data.shape
(150, 4)
```

So this iris data set has 150 samples each with 4 features each representing a measurement of a different part of the sample flower. The first flower sample in the data has

four measurements [5.1, 3.5, 1.4, 0.2], as we see below.

```
>>> iris.data
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       …
```

Each sample also has a labeled target of one of three possible class values [0,1,2], representing three species of irises, and stored in a numpy ndarray referenced by the `.target` member of iris Bunch.

```
>> iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2])
-
```

The values in the target array are just indexes for the class of each flower. The names of each flower species is stored in an array of three strings called *.target_names.*

```
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'],
dtype='<U10')
```

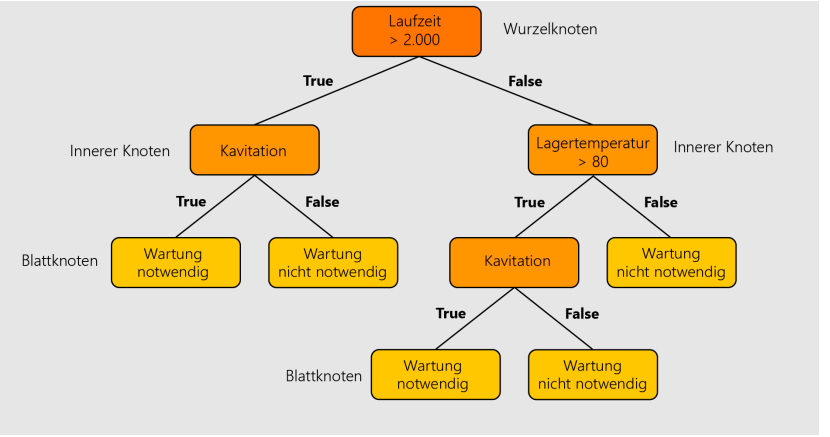
The dtype attribute at the end dtype='<U10' of the array indicates that the data type held by the numpy ndarray is unicode and this gives information about how Numpy stores the array.

CLASSIFICATION VIA DECISION TREES

Decision trees are labeled binary trees (See Chapter 6) and are very useful for constructing a supervised learning model. Decision trees are often used in classification and regression problems. The goal of a decision tree is to create a model that can accurately predict the value of a target variable by applying easy decision rules. These rules are inferred from the data during the learning process. A tree can be seen as a piecewise constant approximation, that is as the value of an attribute crosses a threshold the classification can change.

Decision trees contain at each non-leaf node an if-then-else decision rule followed by new branches of the tree. The deeper the tree, the more complex the decision rules become, and as a result deep-trees may exhibit over-fitting, which is considered a problem of poor generalization from the data.

The basic structure of a decision tree is shown in the figure below, where the yellow leaf nodes label the class, and the orange internal nodes are labeled with T/F decision test.



DECISION TREE CLASSIFIERS IN SKLEARN

To use decision trees within sklearn we use the tree module method called DecisionTreeClassifier, which takes as input two arrays: an array X, of shape `(n_samples, n_features)` holding the training samples, and an array Y of integer values, with shape `(n_samples)`, holding the class labels for the training samples

A DecisionTreeClassifier is capable of both binary classification — where the labels are [-1, 1]), and multiclass classification — where the labels are [0, …, K-1]. In our example using the iris data we have target multiclass classification with labels [0,1,2]

TRAINING AND TESTING A DECISION TREE CLASSIFIER

We will use the standard sklearn function called train_test_split() to split up our data set to create training and testing data sets.

We then call the method tree.DecisionTreeClassifier() to construct a decision tree object named **`clf`** (a standard name for classifier in sklearn). We then fit the data by calling the fit function. And finally we test the accuracy of the model on both the training data and the testing data. Here is the script.

```
from sklearn.model_selection import
train_test_split
from sklearn.datasets import load_iris
from sklearn import tree

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=0)

clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Once we fit the model we can test for accuracy both on the training data and on the testing data. In this example, we find the model is 100% accurate on training data, which is not surprising, since the model saw all the labels of the training data. We find that the model is 97.3% accurate on testing data, which is a pretty solid result.

```
>>> clf.score(X_train, Y_train))
1.0
>>> clf.score(X_test, Y_test))
0.9736842105263158
```

SKLEARN.TREE.DECISIONTREECLASSIFIER

The decision tree classifier has an attribute called tree_ which will allows access to low level attributes such as node_count, the total number of nodes, and max_depth, the maximal depth of the tree. It also stores the entire binary tree structure, represented as a number of parallel arrays. Each node of the tree has a unique index value spanning the arrays, that is the ith element of each array holds information about the node `i`. The standard convention is that node 0 is the tree's root.

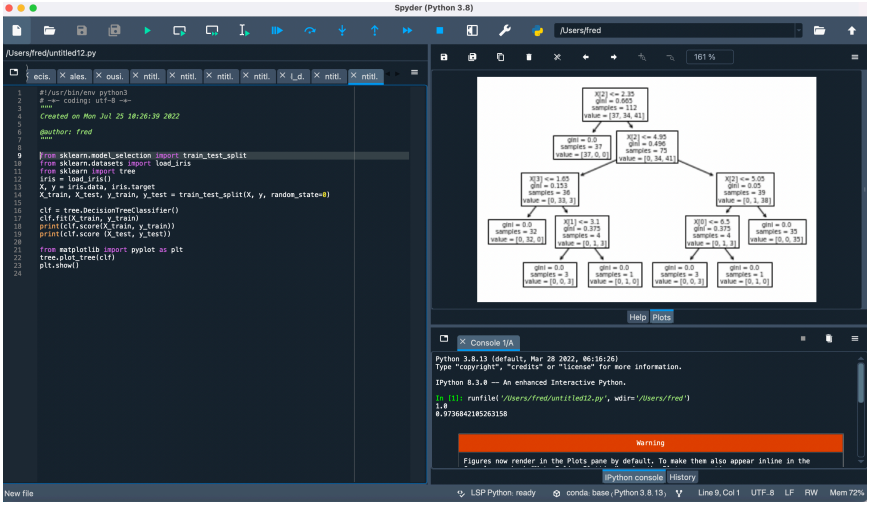
Some of the information arrays only apply to either leaves or internal split nodes. In this case the array values of certain indexes may not be relevant and should be ignored. For example, the information arrays `feature` and `threshold` only apply to split nodes. The values for leaf nodes in these arrays are therefore ignored.

Here are some of these tree information arrays stored in a sklearn decision tree,

`children_left[i]`: id of the left child of node `i` or -1 if leaf node
`children_right[i]`: id of the right child of node `i` or -1 if leaf node
`feature[i]`: feature used for splitting node `i`
`threshold[i]`: threshold value at node `i`
`n_node_samples[i]`: the number of training samples reaching node `i`
`impurity[i]`: the impurity at node

VISUALIZATION OF THE DECISION TREE

We can visualize the resulting sklearn decision tree to see each of the nodes and the logic of each split. We do this by referencing matplotlib and adding three lines of code at the end of the above sample



script:

```
from matplotlib import pyplot as plt
tree.plot_tree(clf)
plt.show()
```

Visualization of Decision tree for Iris dataset.

GINI SCORING

In the image generated above you will see that each node has a gini score associated with it. This score is a statistically measured value between 0 and 1 and used by the algorithm to determine the next decision variable to pick for the branching question.

The gini score is a measure of statistical dispersion, and is intended to represent the inequality between the classes. For

decision trees, more inequality is usually better since we are using information to determine the particular target. The gini score was initially developed by the statistician and sociologist Corrado Gini and used to measure income inequality.

For decision trees, lower gini scores are better. A gini score of 0 expresses no impurity or perfect targeting, where all the samples remaining are of the same class. While a gini score of 1 (or 100%) expresses maximal impurity where all classes are equally represented.

Consider a dataset D that contains samples from k=3 classes. The probability or equivalently the fraction of samples belonging to class i at a given node can be denoted as p[i] and the gini score of D is defined as 1 minus the sum of the squares of these probabilities. Here is the gina-score function in code.

```
gini-score = lambda p,k:
       1 - sum([ p[i]**2 for i in range(k)])
```

We can allow our tree to be determined or generated by those branching questions which minimize the gini scores.

FROM DECISION TREES TO RANDOM FORESTS

One way of improving the results of the decision tree model for large and more complex data sets is to use more than one decision tree and take a weighted average of the results.

This is the goal of general machine learning strategy called *ensemble methods***.** The idea here is to combine the predictions of several base estimators. Two families of ensemble methods are usually distinguished. In *averaging methods*, the driving principle is to build several estimators independently and then to average their predictions. The combined estimator is usually better than any of the single base estimators. Statistics often show that by taking averages, the variance of the output is usually reduced.

By contrast, in *boosting methods* the base estimators are built sequentially and one tries to reduce the bias of the combined estimator in each stage. The motivation is to combine several weak models to produce a more powerful ensemble.

The sklearn.ensemble module includes an averaging algorithm based on randomized *decision trees* called the RandomForest algorithm. The prediction of the ensemble is given as the averaged integer prediction of individual decision tree classifiers.

As with the other classifiers, forest classifiers have to be fitted with two arrays: an array X of shape `(n_samples, n_features)` holding the training samples, and an array Y of shape `(n_samples)` holding the target values.

In random forests each tree in the ensemble is built from a sample drawn with replacement from the training set. Furthermore, when splitting each node during the construction of each tree, the best split is found either from all input features or a random subset of size `max_features`.

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit producing rules too closely matching the data. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant, and thus yields an overall better model.

In contrast to the original algorithm designers, the sklearn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

READING DATA FILES WITH PANDAS

We will be using a library called Pandas to help us read in a data file into our programs. You may need to install pandas with the following command line:

```
$ conda install pandas
```

With pandas we can read in a .csv datafile as follows:

```
data = pd.read_csv('/Users/fred/Desktop/
                                    datafile.csv')
```

We will be covering more features of the important Pandas library in the following chapters.

LAB #9

HEART DISEASE CLASSIFICATION

For this lab you will run computer experiments to find good classifiers for the standard heart disease data set. Please download this dataset to your Desktop from Canvas.

Here is starter code, which you can read in the data and experiment with it. Please try to compile and run this code first. For this lab you will take this starter code and add code to carry out a set of experiments.

```
import pandas as pd
```

```
import numpy as np

heart_disease = pd.read_csv('/Users/fred/
Desktop/heart.csv')

X = heart_disease.drop(['target'] , axis=1)
Y = heart_disease['target']

from sklearn.ensemble import
RandomForestClassifier

clf =
RandomForestClassifier(n_estimators=10)

from sklearn.model_selection import
train_test_split

X_train, X_test, Y_train, Y_test =
train_test_split(X, Y, test_size = 0.2)

clf.fit(X_train, Y_train)
y_pred = clf.predict(X_test)


print(clf.score(X_train, Y_train))
print(clf.score(X_test, Y_test))
```

It is recommended that you follow this video tutorial on using sklearn. For the lab you should view the video between the 5:00 minute and 19:00 minute marks. The video will explain all the steps used in the starter code above, as well as showing you how to run an experiment with manual hyper-parameter tuning.

https://www.youtube.com/watch?v=BXkqEXjBf5s&ab_channel=MachineLearni

ng

For the lab you will vary the number of default estimators for this classifier, as shown in the video. This tuning parameter is already named n_estimators, and this parameter specifies the number of random trees used by the ensemble classifier.

You will write code to search to find the number that gives the best accuracy on the test data, X_test. Report the scores for your results given by clf.score(X_test, Y_test) over a wide range of n_estimator values.

For the lab you should summarize your results in a concise data table. Finally, put this data table as a comment at the top of your code submission.

OPTIONAL ADDITIONAL PROJECT

Use the classic Titanic dataset to fit a binary classification decision tree model using sklearn's DecisionTree and Random Forest Classifiers. What is the accuracy you are able to achieve.