QBUS6850: Tutorial 8 – Advanced Classification Techniques II

Objectives

- To learn how to build Random Forest;
- To learn how to use Random Forest and Adaboost model from scikit;

1. Random Forest Classification

引导

In sklearn tree's in a forest are built by bootstrapping the training set. So each tree is built from a slightly different set of data.

Sklearn also uses a random subset of features when deciding splits. This is to decrease the variance of the forest by introducing some randomness at the cost of increasing bias. The idea is that we will achieve a good spread of features used. If we were to split each tree using the same features then they would end up identical! Overall this strategy yields a better model.

The final classification of each class is given by averaging the probability output of each tree. In other words we caclulate predict_proba() from each tree and average them together. Then pick the most likely class.

Bank Customer Example

Lets use a random forest to classify customers in the bank customer dataset.

First load the data and build the train/test sets.

```
import numpy as np
import pandas as pd
from sklearn import ensemble
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import time

bank_df = pd.read_csv("bank.csv")

X = bank_df.iloc[:, 0:-1]
y = bank_df['y_yes']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Building a Random Forest in sklearn is just like using any other classifier. Create the object and then call the object's fit() function.

```
clf = ensemble.RandomForestClassifier(class_weight = 'balanced')
clf.fit(X_train, y_train)
```

Finally let's check the classification accuracy on the test set

```
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

Building an Optimal Forest

Of course we need to pick the best tree and there are many parameters to optimise. For a full list please refer to the documentation http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

The main ones to focus on are:

- number of trees
- max_depth

Here I will set up a grid of parameters to search through.

```
param_grid = {'n_estimators': np.arange(1,200,10),
             'max_depth': np.arange(1,20,1), }

clf_cv = GridSearchCV(ensemble.RandomForestClassifier(class_weight =
'balanced'), param_grid)

print("Running....")
tic = time.time()
clf_cv.fit(X_train, y_train)

toc = time.time()
print("Training time: {0}s".format(toc - tic))

clf = clf_cv.best_estimator_
print(clf)

y_pred = clf.predict(X_test)

print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

ExtraTreesClassifier

Sklearn provides another class for classifying using forests: ExtraTreesClassifier. This class implements "Extremely Randomised Forest". As in random forests, a random subset of candidate features is used. Additionally thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This reduces the variance of the model a even more but also increases bias.

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

2. Adaboost

AdaBoost is available in the sklearn ensemble library. It is used in the same way as every other sklearn class. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

The core principle is to fit a sequence of weak learners via boosting. Boosting is a process of increasing the weights of samples that were misclassified, then building a new classifier. The new classifier is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence.

By default AdaBoostClassifier uses DecisionTreeClassifier objects as the base classifier, however you can use a different classifier if you prefer. Check the docs for compatible classes.

Some paramters to tune are:

- n_estimators
- learning_rate

Below is an example of how to build an AdaBoost classifier. By default n_estimators = 50.

```
clf = ensemble.AdaBoostClassifier()


clf.fit(X_train, y_train)


y_pred = clf.predict(X_test)

print(classification_report(y_test, y_pred))print(confusion_matrix(y_test, y_pred))
```

Note that there are two available algorithms in AdaBoostClassifier: "SAMME" and "SAMME.R". The "SAMME.R" is the default algorithm used in Python and always return estimator weights as 1. "SAMME" will ouput unequal estimator weights (voting powers) for different estimators. Refer to the Python doc and following post for more information:

https://stackoverflow.com/questions/31981453/why-estimator-weight-in-samme-r-adaboost-algorithm-is-set-to-1

Combining Disparate Classifiers

We can go even further with ensemble classification. We can combine classes that are disparate and combine their predictions together for a more accurate classification. For example we could combine multiple RandomForests together or multiple Boosted Forests together!

This is implemented in sklearn with the VotingClassifier class http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html.

VotingClassifier provides two main parameters to specify:

voting scheme, how predictions from ensemble are combined
weights, we can weight each classifier's vote
The voting scheme is either "hard" or "soft". Hard scheme means majority voting, while soft means we sum up the class probabilites of each classifer then make a decision.

Example

Below I will use the iris dataset to demonstrate how using multiple classifiers together can slightly improve classification accuracy.

```
from sklearn import datasets
iris = datasets.load_iris()
X, y = iris.data[:, 1:3], iris.target

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.model_selection import cross_val_score

clf1 = LogisticRegression(random_state=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()

eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')

for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random Forest',
'Naive Bayes', 'Ensemble']):
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))
```

Multi-class Classification (and dealing with text, optional)

In this section I want to introduce a more complex case. First the data contains more than two classes. Second the dataset contains only text data.

> The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. The data is organized into 20 different newsgroups, each corresponding to a different topic. Some of the newsgroups are very closely related to each other (e.g. comp.sys.ibm.pc.hardware / comp.sys.mac.hardware), while others are highly unrelated (e.g misc.forsale / soc.religion.christian). Here is a list of the 20 newsgroups, partitioned (more or less) according to subject matter:

source: http://qwone.com/~jason/20Newsgroups/, https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups*

The goal is: given a text document, correctly assign it to the newsgroup from which it came from.

We can directly use decision trees, forests etc for multi-class classification without any modification.

The difficulty is transforming the text data into a numeric representation. Recall that a decision tree operates on features and threshold values for those features. Text does not satisfy this requirement.

Bag of Words (Vectorising Text)

A simple method of dealing with text is to treat each word in a corpus as a feature. For each document we count the number or frequency of each word.

Below is a simple example. Note that X is a sparse matrix data type and some columns will be out of order.

```
from sklearn.feature_extraction.text import CountVectorizer

count_vectorizer = CountVectorizer()

corpus = ['This is the first document.',
        'This is the second second document.']

X = count_vectorizer.fit_transform(corpus)
print(count_vectorizer.get_feature_names())
print(X)
```

Then you print out the tree rules as if-else statement:

```
# extract true rules
tree_to_code(clf, feature_names)
```

In large datasets there will be lots of repeated words such as "a", "is" and "the" that don't carry much useful information. These terms should be ignored or given very small weights.

We can use the tf–idf transform to boost the weights of uncommon words (which are likely domain specific) and shrink common words.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()

tfidf = tfidf_vectorizer.fit_transform(corpus)

print(tfidf_vectorizer.get_feature_names())

print(tfidf)
```

N-grams

Unfortunately this simple method has two major drawbacks:

- it cannot handle phrases (multiple word expressions), which removes the order or dependancy information
- it cannot handle typos

So it is suggested that you use n-grams. Actually we have already been using unigrams so far.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(analyzer='word', ngram_range=(1, 2))

tfidf = tfidf_vectorizer.fit_transform(corpus)

print(tfidf_vectorizer.get_feature_names())

print(tfidf)
```

Usenet Newsgroups

Now we are armed with the right tools to tackle the problem at hand of classifying text documents.

Let's transform the text documents to an n-gram representation and build a random forest to classify new documents.

For the sake of running time I am going to manually pick the number of trees as 100 in my forest. Normally you should use CV to pick the best number of trees and the tree depth.

```
from sklearn.datasets import fetch_20newsgroups

categories = None
```

```
remove = ('headers', 'footers', 'quotes')

data_train = fetch_20newsgroups(subset='train', categories=categories,
                                shuffle=True, random_state=42,
                                remove=remove)

data_test = fetch_20newsgroups(subset='test', categories=categories, shuffle=True,
random_state=42, remove=remove)


X_train = data_train.data
y_train = data_train.target

X_test = data_test.data
y_test = data_test.target

# tfidf_vectorizer = TfidfVectorizer(analyzer='word', ngram_range=(1, 2))
tfidf_vectorizer = TfidfVectorizer(sublinear_tf=True, max_df=0.5,
stop_words='english')

print("Fitting and Transforming")
usenet_tfidf = tfidf_vectorizer.fit_transform(X_train)
print("Done")

usenet_clf = ensemble.RandomForestClassifier(n_estimators=100)

print("Training....")
usenet_clf.fit(usenet_tfidf, y_train)
print("Training completed.")

y_pred = usenet_clf.predict(tfidf_vectorizer.transform(X_test))

print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```