

## AU 332 ARTIFICIAL INTELLIGENCE: PRINCIPLES AND TECHNIQUES

---

By: Ming Cheng (517021910750) Chunhui Wang (517021910047)

HW#: 5

December 14, 2019

## I. INTRODUCTION

### A. Purpose

In this lab, we use *sklearn* to implement several machine learning algorithms and predict the forest cover type (the predominant kind of tree cover) from strictly cartographic variables (as opposed to remotely sensed data). The data is from the forests located in Roosevelt National Forest of northern Colorado and provided by US Geological Survey and USFS.

The training data contains several features of the forest as well as the *cover\_type* which is the label. The testing data, however, only contains the features. Our goal is to test several machine learning algorithms on the training data, obtain the accuracy through k-fold cross validation and analyze the differences.

### B. Equipment

The few requirements are listed below:

- Python 3.
- pandas
- sklearn
- L<sup>A</sup>T<sub>E</sub>X.

## II. IMPLEMENTATION OF EACH ALGORITHM

In this part, we implement several machine learning algorithms using *sklearn*, obtain the accuracy and analyze the result. After choosing the appropriate value of parameters, we make our best to get as high accuracy as possible.

**We have tried six different algorithms: Random Forest, Extra Tree, KNN, Naive Bayes, Logistic Regression and SVM. Among them, there are three algorithms meet the requirement (with accuracy greater than 80%): Random Forest, Extra Tree and KNN.**

**We will talk about the three algorithms as well as k-fold cross validation in this section. And we will also talk about the rest of the algorithms in the section of Discussions.** s

### A. Random Forest Algorithm

We use *sklearn* to implement the Random Forest Algorithm. The code is shown below:

After analysis, we think the number of trees as the most important factor that make a difference. Therefore, the arguments of this function are training data, training labels, testing data, testing labels and number of trees. Note that we use k-fold cross validation to split the whole dataset (with labels) into training data and testing data.

In this function:

1. *model = RandomForestClassifier(n\_estimators = numberOfTree)*  
To build our model.
2. *model.fit(features, labels)*  
To fit our model with input data.

```

def randomForest(features, labels, features_test, labels_test, numberOfTree):
    model = RandomForestClassifier(n_estimators = numberOfTree)

    model.fit(features, labels)

    predict = model.predict(features_test)

    score = accuracy_score(labels_test, predict)

    data = open("final/rdnf/rdnf_" + str(numberOfTree) + ".txt", "a", newline = '')

    print('Tree num:', numberOfTree, '\t', score, file = data)

    data.close()

    if os.path.exists("final/rdnf/predict_randomforest_" + str(numberOfTree) + ".csv") == False:

        with open("final/rdnf/predict_randomforest_" + str(numberOfTree) + ".csv", "w", newline = '') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(
                ["Id", "Cover_Type"])
            for i in range(0, len(predict)):
                writer.writerow([15121 + i, predict[i]])

    return score

```

FIG. 1: code of Random Forest

3. `score = accuracy_score(labels_test, predict)`  
To get the prediction through the algorithm.
4. `score = accuracy_score(labels_test, predict)`  
To compute the accuracy.

Also, we write the prediction result into a csv file which contains the "Id" and "Cover\_Type".

We use 5-fold cross validation to get five different results each time of training and compute the average of them to get the final accuracy. After we do the whole training process, we can get the following figure:

As we can see from the figure, the Random Forest Algorithm can get a very satisfactory result, which is around **86.25%**. Also, we can see that when number of trees is equal to 250, we can get the highest accuracy, which is about **86.41%**.

Because this model consists of several submodels, it is good at learning a large amount of features of the dataset. Besides, the model is very robust because of the existing of many little submodels.

In addition, due to the randomness of its sturcture, which means the submodels are built randomly, the generalization ability of the model is very good and we don't need to do the feature extraction before.

Theoretically, if the number of submodel is small, it is easy to be underfitting, while it is likely to overfit if the number of submodel gets bigger. According to the figure, we can see that the accuracy rises in general as the number of trees grows when it is less than 350. However, when the number of trees gets bigger, the accuracy begins to decrease, due to the overfitting.

## B. Extra Tree

Extra Tree Algorithm is the variant of Random Forest Algorithm. However, there are differences between them.

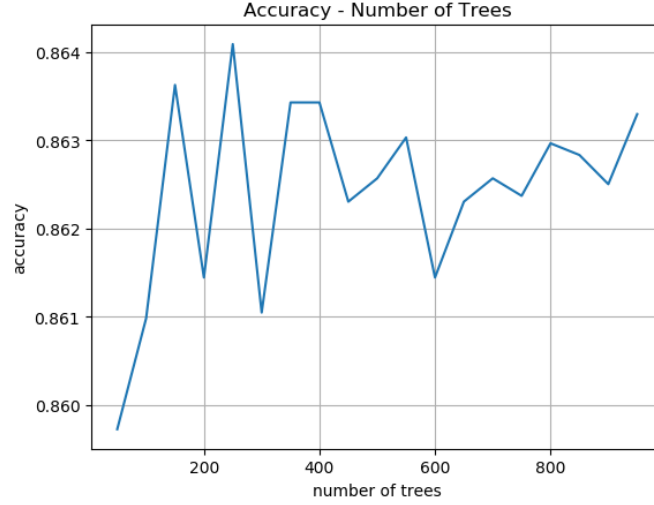


FIG. 2: result of Random Forest

Firstly, the Random Forest Algorithm corresponds to Bagging Model, which means each decision tree corresponds to only a part of the whole dataset. Extra Tree, however, uses all of the dataset to get each submodel(decision tree).

Besides, Extra Tree has greater randomness than Random Forest. Here is the theory of Extra Tree Algorithm: (take binary tree as an example)

When the feature attribute is in form of a category, samples with certain categories are randomly selected as the left branch, and samples with other categories are used as the right branch. When the feature attribute is in form of a value, a random number will be chosen between the minimum value and maximum value. When the value of the sample is greater than the random number, it is regarded as the left branch; when it is less than the the random number, it is regarded as the right branch. In this way, the purpose of randomly distributing samples to two branches under this feature attribute is achieved. Then calculate the bifurcation value at this time (if the feature attribute is in the form of a category, the Gini index can be applied; if the feature attribute is in the form of a value, the mean square error can be applied). Iterate through all the feature attributes in the node, and obtain the fork values of all the feature attributes according to the above method. We choose the form with the largest fork value to achieve the fork of the node. As can be seen from the introduction above, this method is more random than the Random Forest.

Here is the code of Extra Tree:

We import the certain library using *from sklearn.ensemble import ExtraTreesClassifier* syntax. After our analysis, we think the number of trees as the most important factor that effects the performance. Therefore, we set one of the arguments of the function as *numberOfTree*.

In this function:

1. `model = ExtraTreesClassifier(n_estimators = numberOfTree)`

To bulid our model.

2. `model.fit(features, labels)`

To fit our model with input data.

3. `score = accuracy_score(labels_test, predict)`

To get the prediction through the algorithm.

4. `score = accuracy_score(labels_test, predict)`

To compute the accuracy.

```

def extraForest(features, labels, features_test, labels_test, numberOfTree):

    model = ExtraTreesClassifier(n_estimators=numberOfTree)

    model.fit(features, labels)

    predict = model.predict(features_test)

    score = accuracy_score(labels_test, predict)

    data = open("final/extf/extf_" + str(numberOfTree) + ".txt", "a", newline = '')

    print('Tree num:', numberOfTree, '\t', score, file = data)

    data.close()

    if os.path.exists("final/extf/predict_extraforest_" + str(numberOfTree) + ".csv") == False:

        with open("final/extf/predict_extraforest_" + str(numberOfTree) + ".csv", "w", newline = '') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(
                ["Id", "Cover_Type"])
            for i in range(0, len(predict)):
                writer.writerow([15121 + i, predict[i]])

    return score

```

FIG. 3: code of Extra Tree

Also, we use k-fold cross validation to evaluate the performance of the model.

Here is the result of Extra Tree:

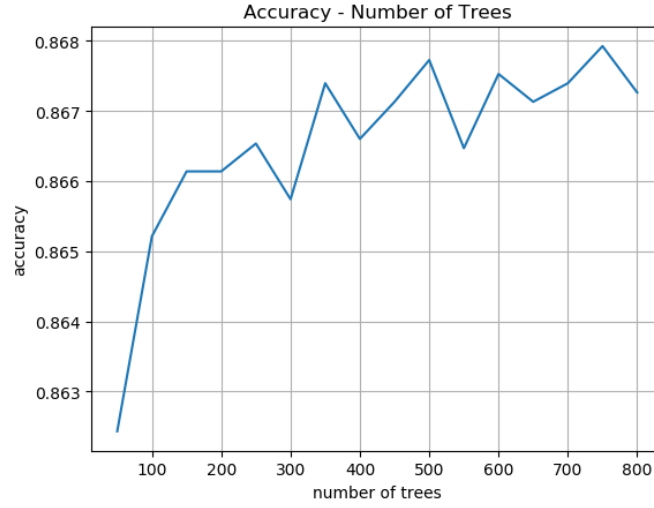


FIG. 4: result of Extra Tree

As we can see, as the number of tree grows, the accuracy also increases. When the number of trees is higher than 400, the average accuracy is around **86.75%**. And the highest accuracy can goes to **86.80%** with the number of trees as 750.

Besides, because the accuracy already meets the requirment, we deside not to keep on increasing the number of trees in order to prevent the overfitting phenomenon.

As the theoretical analysis mentioned above, Extra Tree has greater randomness than Random Forest, so it behaves better if we have a large amount of features.

### C. KNN

We use *sklearn* to implement the Knn Algorithm. The code is shown below:

```
def knn(features, labels, features_test, labels_test, i):
    knn = neighbors.KNeighborsClassifier(n_neighbors=i, metric='_minkowski', p=_2)

    # train
    knn.fit(features, labels)
    predict = knn.predict(features_test)
    return accuracy_score(labels_test, predict)
```

FIG. 5: code of KNN

After analysis, we think the number of neighbors as the most important factor that make a difference. Therefore, the arguments of this function are training data, training lables, testing data, testing labels and number of neighbors. Note that we use k-fold cross validation to spilt the whole dataset(with labels) into training data and testing data.

In this function:

1. *model = KNeighborClassifier(i=number of neighbors)*  
To bulid our model.
2. *model.fit(features, labels)*  
To fit our model with input data.

### 3. *model.predict(features\_test)*

To get the prediction through the algorithm.

We use 5-fold cross validation to get five different results each time of training and compute the average of them to get the final accuracy. After we do the whole training process, we can get the following figure:

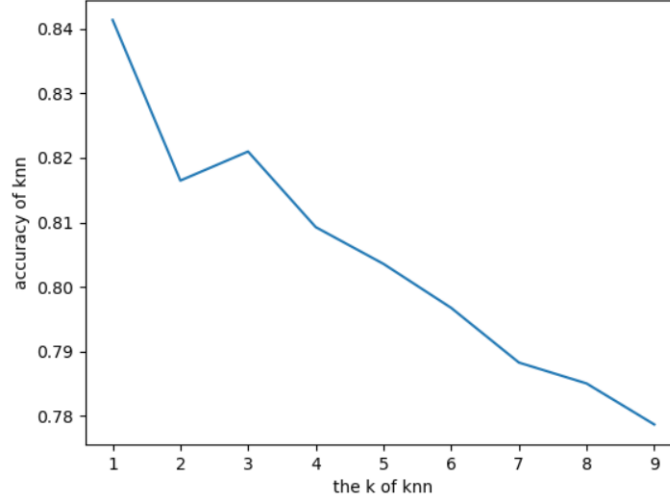


FIG. 6: result of Knn

As we can see from the figure, the Knn Algorithm can get a normal result which better than naive bayes and svm, which best is around **84%**. We can see that  $k=1$  is the most accurate. As the number of  $k$  increases, the accuracy decreases linearly. When  $k=3$ , there is a small peak that is larger than the surrounding value. We can conclude that the algorithm of KNN also achieves high precision. KNN also has the advantage of being insensitive to outliers. However, KNN has high computational complexity and space complexity when calculating multi-classification problems.

#### D. K-fold Cross Validation

In order to evaluate the performance of the algorithms, we use k-fold cross validation and  $k=5$  in our model.

Here is the code of k-fold cross validation:

```
features, labels = load_csv_data('all/train.csv')

features_test = load_csv_test('all/test.csv')

print('extraForest\n')

ss = ShuffleSplit(n_splits=5, random_state=random.randint(1, 100), test_size=0.2)

score = []
num = []

for numTree in range(50, 850, 50):
    num.append(numTree)

    score_per_lis = []

    for train_index, test_index in ss.split(features):
        train_X, train_y = features[train_index], labels[train_index]
        test_X, test_y = features[test_index], labels[test_index]

        score_per_lis.append(extraForest(train_X, train_y, test_X, test_y, numTree))

    score_per = np.mean(score_per_lis)

    score.append(score_per)

plot(num, score)
```

FIG. 7: code of k-fold cross validation

Because we want to find the best hyper-parameter(number of trees), we design the two-loop structure. In the outter loop, we change the number of trees each time of training, and for each value of *numTree*, we design the inner loop to implement the k-fold cross validation. Besides, we split the data randomly through the syntax *ss = ShuffleSplit(n\_splits=5, random\_state=random.randint(1, 100), test\_size=0.2)*.

Above are the implementation of Random Forest, Extra Tree, KNN and k-fold Cross Validation. All of the three algorithms can get the accuracy greater than 80% and the best result is around 86.80%.

In the section of Discussions, we will also introduce the implementation of other algorithms, including SVM, Logistic Regression and Naive Bayes.



### III. DISCUSSIONS

We also implemented other algorithms, including Naive Bayes, SVM and Logistic Regression.

#### Logistic Regression :

We implemented the Logistic Regression Algorithm using *sklearn*. We import the library through *from sklearn.linear\_model import LogisticRegression* and also use k-fold cross validation to evaluate the performance of this algorithm.

The code is shown as follows:

```
def testLogisticRegression(features, labels, features_test):  
  
    rf2 = LogisticRegression()  
  
    # train  
    rf2.fit(features, labels)  
  
    # predict  
    predict = rf2.predict(features_test)  
  
    # write csv  
    with open("all/predict.csv", "w", newline='') as csvfile:  
        writer = csv.writer(csvfile)  
        writer.writerow(  
            ["Id", "Cover_Type"]  
        )  
        for i in range(0, len(predict)):  
            writer.writerow([(15121 + i), predict[i]])
```

FIG. 8: code of Logistic Regression

The result is shown as follows:

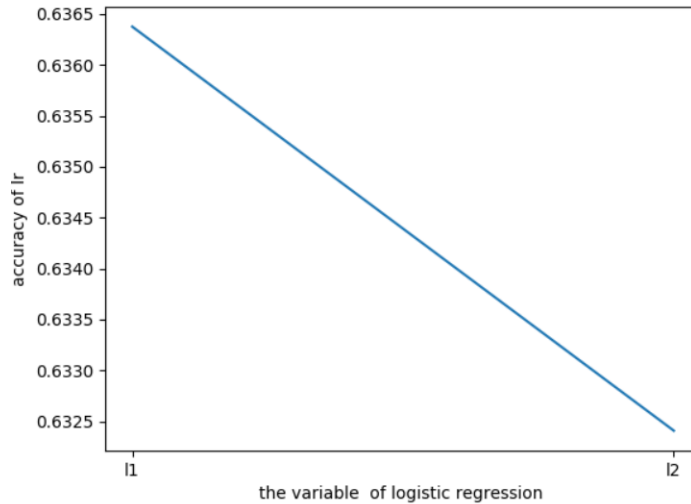


FIG. 9: result of Logistic Regression

As we can see from the figure, no matter how we choose the value of the hyper-parameters of the algorithm, the best result is around 63.64%, which is not satisfactory.

### Naive Bayes (including data preprocessing):

In order to learn the features more precisely, we preprocess the data before running the algorithm. We divide the whole features into **continuous features** and **discrete features**. As for continuous features,

```
# Load data
def load_csv_data(filename):
    file = pd.read_csv(filename)

    data = file[['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology', 'Vertical_Distance_To_Hydrology',
                'Horizontal_Distance_To_Roadways', 'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm', 'Horizontal_Distance_To_Fire_Points']]
    labels = file['Cover_Type']
    data = np.array(data)
    labels = np.array(labels)
    return data, labels

def load_csv_data1(filename):
    file = pd.read_csv(filename)

    data = file[['Wilderness_Area', 'Soil_Type']]
    labels = file['Cover_Type']
    data = np.array(data)
    labels = np.array(labels)
    return data, labels
```

FIG. 10: code of Naive Bayes

we import the *naive\_bayes.GaussianNB* library while *naive\_bayes.MultinomialNB* is imported for learning discrete features. After splitting the features and running the algorithm, we can get the following results:

```
def knn1(features, features1, labels, features_test, features_test1, labels_test, i):
    gnb = GaussianNB()
    mnnb = MultinomialNB(alpha=i/10)
    gnb.fit(features, labels)
    mnnb.fit(features1, labels)
    predict1 = gnb.predict_proba(features_test)
    predict2 = mnnb.predict_proba(features_test1)
    predict = np.argmax(np.multiply(predict1, predict2), axis=1)+1
    return accuracy_score(labels_test, predict)
```

FIG. 11: code of Naive Bayes

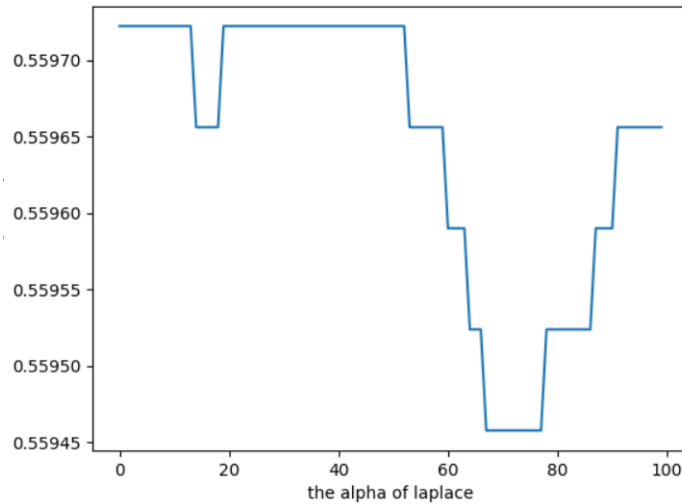


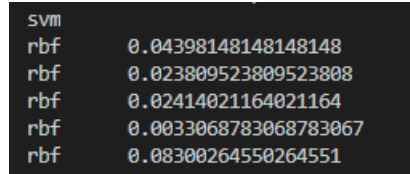
FIG. 12: result of Naive Bayes

As we can see, the best result we get is even less than 55.98%, which is not satisfactory. Also, as the

hyper-parameter  $\alpha$  increases, the result becomes even worse.

**SVM :**

We implemented SVM using *sklearn*. Here we take use 'rbf' as the kernel. The result is shown as follows:



svm	
rbf	0.04398148148148148
rbf	0.023809523809523808
rbf	0.02414021164021164
rbf	0.0033068783068783067
rbf	0.08300264550264551

FIG. 13: result of SVM

As we can see, the accuracy is very low, which is even less than 10%. Besides, the training time is very long. Therefore, this algorithm should not be considered in this problem.

#### IV. COMPARISON BETWEEN DIFFERENT ALGORITHMS

The best result of each algorithm is shown in the table below:

TABLE I: The best accuracy of each algorithm

Random Forest	Extra Tree	KNN	Naive Bayes	Logistic Regression	SVM
86.41%	86.80%	84%	55.97%	63.64%	8.3%

As we can see, the most appropriate algorithm in this problem is **Extra Tree**. Also, we can see that **Random Forest** and **KNN** also work very well.

Due to the randomness of Random Forest and Extra Tree(which is its variant), these two algorithms behave very well when the number of features is large. Also, because of the randomness, it is not likely for them to overfit.(As for the Random Forest, when the number of trees grows, the result can converge.)

As for KNN: Because the sample we input has discrete features, KNN can work well. Also, the time complexity of KNN is  $O(n)$ , which is satisfactory in a way. However, if the number of single samples is not very large, it is likely to misclassify, which is the most possible reason that the accuracy cannot be very high, from our own perspective.

As for Logistic Regression: If two samples which are relevant are inputted into the model, it is likely to misclassify. Because our dataset is large, this phenomenon is likely to occur. We think this is the reason that Logistic Regression behaves not very satisfactorily.

As for Naive Bayes: Naive Bayes model assumes that attributes are independent of each other, which is not always the case. In our dataset, different features are not independent from each other strictly, so the assumption of Naive Bayes does not work, which results in the bad behavior of this model in this problem.

As for SVM: Because it needs to compute the  $n$ -th order matrix( $n$  is the number of features), the training time will be very long if the dataset is large. What's more, SVM behaves better on binary classification problem rather than multi-classification problem. We think this is the reason that SVM is not appropriate in this problem.

#### V. CONCLUSIONS

In this lab, we implement several machine learning algorithms, including Random Forest, Extra Tree, KNN, Naive Bayes, Logistic Regression and SVM to do the classification task on the dataset which is from the forests located in Roosevelt National Forest of northern Colorado and provided by US Geological Survey and USFS.

After using k-fold cross validation process to evaluate the performance of different algorithms, we find that **Random Forest, Extra Tree and KNN** can meet the requirement(with accuracy greater than 80%). Also, when using Extra Tree Algorithm and choose appropriate value of hyper-parameters, we can get the highest accuracy which is around **86.80%**. Besides, we have written the prediction result into a csv file. We also analyze the reason of the result each algorithm gets and do the comparison process among them.

All in all, we successfully implement the algorithms above and do some analysis as well. This lab also enables us to get more familiar with *sklearn*.