Forest cover type Prediction

(Through cartographic variables only)
Project Documentation

Machine Learning –LM Project 2016-17

Professor : Mrs.P.Velardi

Participant: Panteha Pishehvar

Maricola: 1672553

Panteha.pishehvar@gmail.com

September 2017

SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA
UNIVERSITÀ DI ROMA

Table of Contents

Introduction **1**

Problem Description 3

Data Set & Tools Used 3

Approach & Methodology 4

Data Exploration **5**

Feature Ranking 6

Data Preprocessing **8**

Missing Values 8

Correlations between features 8

Feature Engineering 10

Dimensionality Reduction 10

Models & Methods **12**

Random Forest 12

K Nearest Neighboring 12

SVM 13

Naïve Bayes 13

Parameter Tuning **14**

Random Forest Tuning 14

Evaluation **15**

Prediction **18**

Further steps **18**

SAPIENZA
UNIVERSITÀ DI ROMA

Problem Description

The Forest Cover Type Prediction is a supervised multi-class classification problem addressing the prediction of the forest cover type using cartographic features. Each instance is sampled on 30 x30 m squares of the Roosevelt National Forest in northern Colorado. The forests in these areas have seen few human-caused disturbances; therefore existing forest cover types are not a result of forest management practices but ecological processes so it makes sense try to predict the cover type having given variables of sun shade, soil type and hydrological properties.

Data Set & Tools Used

The data set is composed of cartographical and geological data gathered from over 581,012 30 x 30 meters square cells of undisturbed forest. Each sample consists of 12 measures, broken into 54 distinct input variables. Of these 54, 10 are quantitative measures while the remaining 44 are Boolean values representing soil conditions and area.

The data was presented, without scaling, in a comma-delimited list, the last column of each row being the class designation.

A sample row is presented below.

2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,5

The dominant forest cover type associated to each sample is one of seven specific forest cover types:

1. Spruce-Fir
2. Lodgepole Pine
3. Ponderosa Pine
4. Cottonwood/Willow
5. Aspen
6. Douglas-Fir
7. Krummholz

The csv-format raw data is approximately 74 Megabytes (MB) in size. Any conversion or manipulation of data would be significantly time-consuming: Minor changes on a record by record basis had incredible effects, as they were repeated over 580,000 times, once for each data point (row in the dataset).

I used **Python 2.7** embedded in **Anaconda**(which is a package management tool), documenting my experiments in **Jupyter notebook**, which contains both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc...).

Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis.

Scikit-learn is an open source machine learning library for the Python programming language. It features various classification, regression and clustering algorithms and is designed to interoperate with the Python numerical and scientific libraries **NumPy** and **SciPy**.

Pandas is a software library written for the Python programming language for data manipulation and analysis.

It's a good tool for reading and writing data between in-memory data structures and different formats: CSV and text files, Microsoft Excel and SQL databases.

I use pandas to load data from CSV files in this project.

Occasionally I used **Seaborn** library to better plot the graphs.

Approach and Methodology

I tried to complete the project in several rounds, each time a bit deeper into the tools and possibilities to improve the task: I spent much time in realizing how preprocessing would affect the final results, I came to the end by writing two little scripts to manipulate binary data and combine them into an integer value.

Then I studied the differences between cross validation train_test_split and KFold and eventually used the former one provided that randomization is guaranteed.

```
from sklearn import cross_validation
X_train, X_test, y_train, y_test = cross_validation.train_test_split(X_covertype,y_covertype,test
X_val, X_test, y_val, y_test =cross_validation.train_test_split(X_test, y_test, test_size=0.88)
print X_train.shape
print X_val.shape
print X_test.shape
```

```
(29050, 50)
(66235, 50)
(485727, 50)
```

I chose the classifiers based on a rough idea got from studying different resources upon their proper estimation on multi-classifiers.

At the end sometime dedicated to tuning the parameters.

In the jupyter notebook where ever I did any process I added the explanation of what is going on.

Data Exploration

On the first observation, the fact that four wilderness-area and forty soil_type features are mutually exclusive raised the idea to combine them to make out two features.

In machine learning literature these raw features are called one-hot encoded and I looked to do a converse process to combine them back together and see if this might change the models: is it efficient to do such reduction?

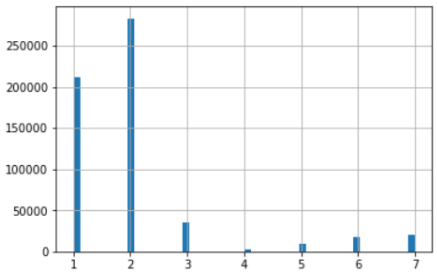


Fig.1

HINT: The histogram of target classes shows that the distribution is not uniform, so we must make sure while splitting the data set for training and testing the samples should be well-shuffled.

- **Feature ranking**

Using feature_importance_ and a typical model on the training split of data, just to get a rough idea which features play a more important role (i.e. a bigger InfoGain)

```
# check out the feature importance on raw dataset
# Need a typical classifer to determine the feature importance according to
import matplotlib.pylab as plt
def importances(estimator, col_array):
     importances = estimator.feature_importances_
     indices = np.argsort(importances)[::-1]
     for f in range(20):
        print("%d. %s (%f)" % (f + 1, col_array.columns[indices[f]], importances[indices[f]]))
     print "\nMean Feature Importance %.6f" %np.mean(importances)
     indices=indices[:10]
     plt.figure()
     plt.bar(range(10), importances[indices], color="bc", align="center")
     plt.xticks(range(10), col_array.columns[indices], fontsize=14, rotation=90)
     plt.xlim([-1, 10])
     plt.show()
```

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_estimators=200, bootstrap=True, oob_score=True)
rfc.fit(X_covertype,y_covertype)
importances(rfc,X_covertype)
```

OUTPUT:

```
Cover Type (Initial RF) Top 20 Important Features

1. Elevation (0.234067)
2. Horizontal_Distance_To_Roadways (0.101709)
3. Horizontal_Distance_To_Fire_Points (0.094316)
4. Horizontal_Distance_To_Hydrology (0.060900)
5. Vertical_Distance_To_Hydrology (0.058190)
6. Aspect (0.055521)
7. Hillshade_Noon (0.051850)
8. Hillshade_3pm (0.050252)
9. Hillshade_9am (0.049478)
10. slope (0.040792)
11. Wilderness_Area4 (0.033868)
12. Soil_Type22 (0.015718)
13. Soil_Type4 (0.011642)
14. Wilderness_Area3 (0.011361)
15. Soil_Type10 (0.011203)
16. Wilderness_Area1 (0.010690)
17. Soil_Type12 (0.010575)
18. Soil_Type39 (0.010408)
19. Soil_Type38 (0.009798)
20. Soil_Type23 (0.009597)

Mean Feature Importance 0.018519
```

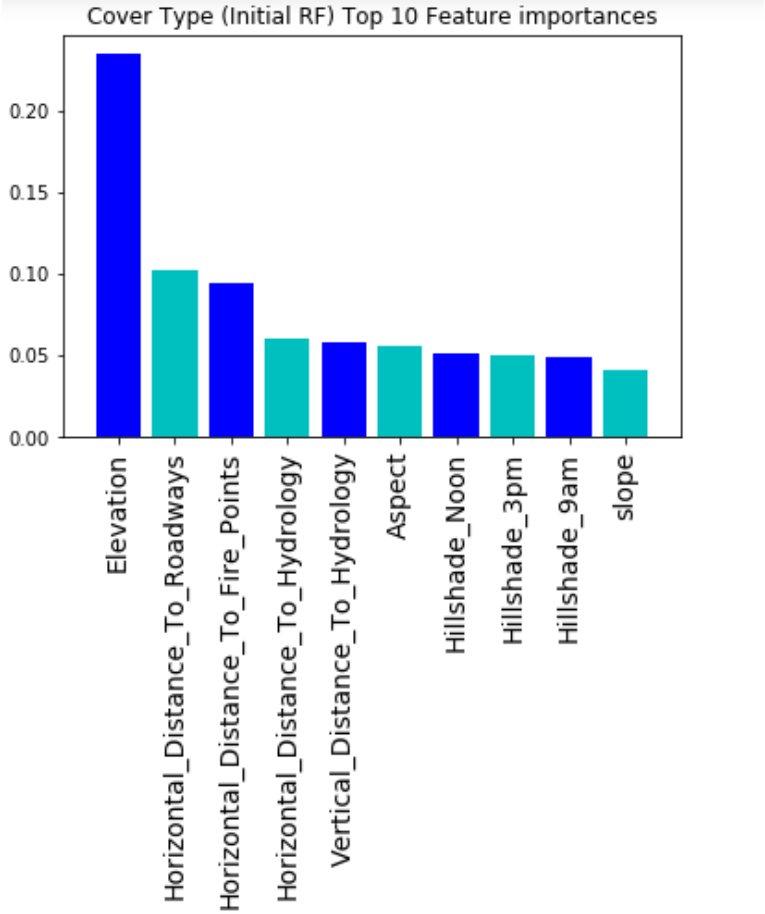


Fig.2

As we can see from histogram of 'Elevation', this feature alone can discriminate target classes 3and 7. Class 4 is really rare.

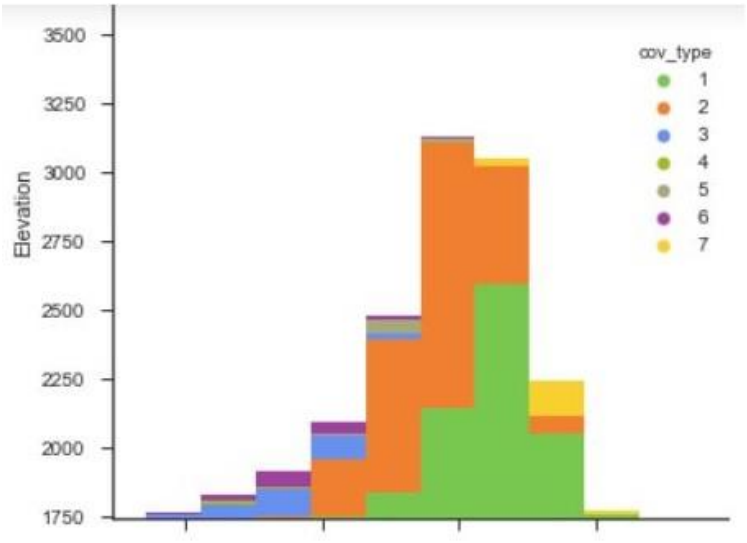


Fig.3

Data Preprocessing

Data preprocessing is a very critical step in any machine learning process, the accuracy of our models is directly dependent on how the raw data is prepared to be conducted to the classifier.

The specific preprocessing needed on the data is decided by properties required for the learning algorithm used to solve the problem.

Here I did some essential preprocessing experiments:

- **Missing values**

scikit-learn estimators assume that all values in an array(i.e. feature vector & target value) are numerical, and that all have and hold meaning; meanwhile many real world datasets contain missing values, often encoded as blanks, NaNs or other placeholders.

A basic strategy to use incomplete datasets is to discard entire rows and/or columns containing missing values.

A better strategy is to impute the missing values, i.e., to infer them from the known part of the data: like replacing the missing values for a feature by the mean value of that feature.

```
# Is there any null value?
X_covertype[X_covertype.isnull().any(axis=1)]
```

	Elevation	Aspect	slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance

0 rows × 54 columns

- **Correlations between features**

First tried to find out any correlation between features, in case of perfect correlation between a pair of features, one is actually redundant and can be dropped out of feature vector. The correlation function applies on continuous input therefore we exempt wilderness area and soil type feature (i.e. categorical features) and consider the very first 10 features.

The correlation function output range is [-1,1] where -1 is perfect negative correlation and 1 is perfect positive correlation.

```
data=covertype.iloc[:,:10]                    #create a dataframe with only 10 first features
cols=data.columns                             #get the names of all the columns
data_corr = data.corr()                    # Calculates pearson co-efficient for all combinations
threshold = 0.5            # Set the threshold to select only only highly correlated attributes
corr_list = []                            # List of pairs along with correlation above threshold
#Search for the highly correlated pairs
for i in range(0,10): #for 'size' features
    for j in range(i+1,10): #avoid repetition
        if (data_corr.iloc[i,j] >= threshold and data_corr.iloc[i,j] < 1) or
        (data_corr.iloc[i,j] < 0 and data_corr.iloc[i,j] <= -threshold):
            corr_list.append([data_corr.iloc[i,j],i,j]) #store correlation and columns index
#Print correlations and column names
for v,i,j in corr_list:
    print ("%s and %s = %.2f" % (cols[i],cols[j],v))
```

OUTPUT:

```
Aspect and Hillshade_9am = -0.58
Aspect and Hillshade_3pm = 0.65
slope and Hillshade_Noon = -0.53
Horizontal_Distance_To_Hydrology and Vertical_Distance_To_Hydrology = 0.61
Hillshade_9am and Hillshade_3pm = -0.78
Hillshade_Noon and Hillshade_3pm = 0.59
```

Strong correlation is observed between the above pairs of features.

Looking at the list of features, intuitively we guess there must be some correlation between features; like the sunlight/shade depends to aspect (geographical direction) or shade at morning is exclusively mutual with shade in afternoon (only in case aspect is south then sunlight will be all day long otherwise sunlight/shade is mutually exclusive in the morning or in the afternoon)

This represents an opportunity to reduce the feature set through transformations such as PCAAs

Is there any constant features?

If a feature is permanently constant it makes no discrimination so we can simply drop it out.

```
#is there any feature premanently constant?
pd.set_option('display.max_columns', None) # we need to see all the columns
X_covertype.describe()
```

Count is 581012 for each column, so no data point is missing, No constant feature to be deleted meanwhile there are some features with such insignificant role in the discrimination we can eliminate them.

- **Feature Engineering**

Feature engineering is an important pre-process for any machine learning task. I did some reasonable feature engineering and followings are the main choices:

```
#Checking the value count for different soil_types
for i in range(15, X_covertype.shape[1]):
    j = X_covertype.columns[i]
    print (X_covertype[j].value_counts())
```

As observed in the output (referred to Jupyter notebook), some soil types are so rare that we can ignore their presence and shrink the dataset retaining those columns.

- **Dimensionality Reduction**

In general the more uncorrelated the features are, the better the classifier performance is going to be. Given a set of highly correlated features, it may be possible to use PCA techniques to make them as orthogonal as possible to improve classifier performance.

We should note that PCA does not "discard" or "retain" any of our pre-defined. It mixes all of features (by weighted sums) to find orthogonal directions of maximum variance.

Since some correlations were found between various features, we applied a dimensionality reduction technique named Principal Component Analysis on the original dataset and did some of the modeling, eventually:

It was observed that on reducing the dimensions, the accuracy decreases as the new set of features are independent. Thus, I decided to take the entire feature set and based on the experiments did some manual dimensionality reduction as follows:

Firstly I dropped the rare soil type features:

```
X_covertype = X_covertype.drop(['Soil_Type7','Soil_Type8', 'Soil_Type15', Soil_Type36'], axis=1)
print X_covertype.shape
```

Then as already mentioned soil_type and wilderness_areas are one-hot encoded categorical features, so I wrote a piece of code to combine wilderness_areas into one feature {1,2,3,4} and the same approach also for the remaining soil_types :

```
#Group one-hot encoded features into one single feature
import numpy as np
def cut_down_col(data_In):
 cols = data_In.columns
 r,c = data_In.shape
 data=[]                              # Create a new dataframe with r rows,
                                      #one column for each newly-encoded category
 new_data = pd.DataFrame(index= np.arange(0,r), columns=['Wild_Area', 'Soil_Types'])
 for i in range(0,r):
    p = 0;
    q = 0;
    for j in range(11,14):            # wild_area range
        if (data_In.iloc[i,j] == 1):
            p = j-10                  # wild_area{1,2,3,4}
            break
        # soil_type range
    for k in range(15,50):      #we already dropped out 4 soil types, so we are left with 36
        if (data_In.iloc[i,k] == 1):
            q = k-14 # soil_type{1,...,40}
            break
    new_data.iloc[i] = [p,q]  # Make an entry in data for each r
 data_out= data_In.iloc[:,:10]     #for i in range(11,50): X_train.drop(cols[i], axis=1)
 data_out=np.concatenate((data_out,new_data.iloc[:,:]),axis=1)
 return(data_out)
```

I wrote above function to create new integer features out of current Booleans; I did it as a function to be able to feed it by splits of original dataset otherwise my computer halted in attempts to convert the whole dataset in one go.

Models & Methods

Once the data cleaning done, training and testing was simply a matter of deciding what ML algorithms to be chosen, which configurations to try and determining how many trials of each configuration should be run. At this point, training and testing went through three phases.
The purpose of this phase was to ascertain which configurations would best classify the data.

- **Random Forest classifier:**

Random forest uses an ensemble method by combining a multitude of decision trees. Random forest uses a bagging method, which averages the predictions of multiple models trained on different samples to reduce the variance and achieve higher accuracy.
For a classification task, an instance from a dataset is classified according to each single tree, the trees will vote for their classification and the class having the most votes will be chosen as the final result.

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_estimators=100, bootstrap=True, oob_score=True)
rfc.fit(X_train,y_train)
y_rfc_model=rfc.predict(X_val)
from sklearn.metrics import accuracy_score
score= accuracy_score(y_val, y_rfc_model)
print score
```

```
0.860783573639
```

- **K nearest neighbors**

The k nearest neighbors is a clustering technique in which we use the Euclidean distance to find the k nearest neighbors to every vector. Then, we find the k nearest neighbors for every test vector during prediction and we use the majority vote to classify the test vector suitably. Extremely high dimensional data may lead to issues with knn as the Euclidean distance becomes almost same for all vectors and hence dimensionality reduction plays an important role in it.
We tried PCA but it was futile. As we reduce the dimensions, the accuracy during validation and testing decreases.

K nearest Neighbor Classifier

```
from sklearn.neighbors import KNeighborsClassifier
knnc = KNeighborsClassifier(n_neighbors=5)
knnc.fit(X_train, y_train)
y_knn_model=knnc.predict(X_val)
score= accuracy_score(y_val, y_knn_model)
print score
```

```
0.829818072016
```

- **SVM**

A support vector machine (SVM) constructs a hyper-plane or set of hyper-planes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. It selects a maximum margin classifier as in general the larger the margin the lower the generalization error of the classifier.
C is called the regular factor. If C is set to be very large, the regularization would be very low, lower C would result in a higher regularization penalty. Thus tweaking the C parameter could give us a better performance in the new data.

RBF Support Vector Machine Classifier

```
from sklearn import svm
svmRBF=svm.SVC(kernel='rbf', gamma=1, C=1)
svmRBF.fit(X_train, y_train)
y_svmRBF_model=svmRBF.predict(X_val)
from sklearn.metrics import accuracy_score
score= accuracy_score(y_val, y_svmRBF_model)
print score
```

```
0.487974635767
```

- **Naive Bayes**

Naive Bayes classifier is a probabilistic classifier based on Bayes' theorem under the assumption of independence between the features. This naïve assumption of independence of features makes it an improper candidate for our project since we already observed a strong correlation between some major features.

Naive Bayes classifer

Although I do not expect to get a good result from this classifier, since the features are expected to be Guassian

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_gnb_model = gnb.fit(X_train,y_train).predict(X_val)
score= accuracy_score(y_val, y_gnb_model)
print score
```

```
0.615731863818
```

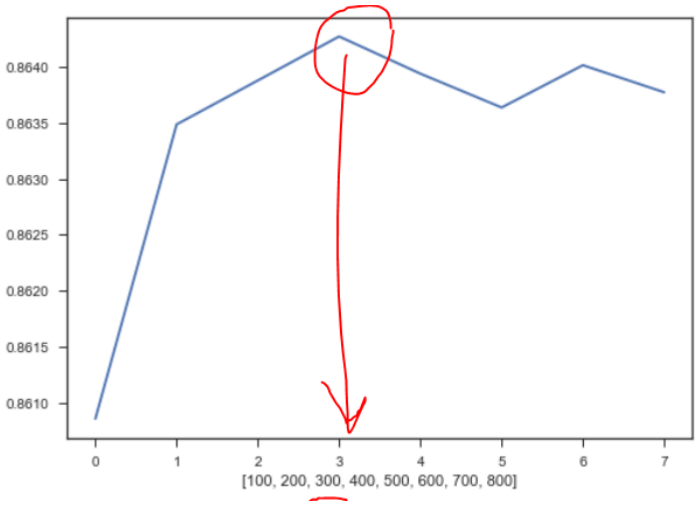
Parameter Tuning

Hyper-parameters are parameters that are not directly learnt within estimators which mean they are passed as arguments to the constructor of the estimator classes.
A traditional method of performing parameter tuning is grid search, which is simply an exhaustive searching over manually specified parameter values for a model. Grid search will evaluate a model for each combination of parameter values specified in a grid.
The goal of running this grid is to find the ranges that the optimum values of the parameters lie in.
For example, to tune the parameters of a Random Forest model, two parameters need to be tuned: number of trees and number of randomly selected features used to look for the best split. The two parameters are denoted as n_estimators and max_features. I tried to manually apply this method to some of the classifiers I modeled:

```
import seaborn as sns
k_range=[100,200,300,400,500,600,700,800]
rfc_scores=[]
for k in k_range:
  rfc = RandomForestClassifier(n_estimators=k, bootstrap=True, oob_score=True)
  rfc.fit(X_train,y_train)
  y_rfc_model=rfc.predict(X_val)
  rfcscore=accuracy_score(y_val, y_rfc_model)
  rfc_scores.append(rfcscore)
plt.plot(rfc_scores)
plt.xlabel(k_range)
plt.show()
```

I didn't engage the parameter max_features since we have only 12 and it is ok to use all of them.

As the output the plot shows the best value for number of estimators:



The other parameter tuning and their plots can be referred to in Jupyter inotebook attached as an appendix.

Evaluation

- **Precision & Recall**

In binary classification, the precision for a class is the number of true positives divided by the total number of elements labeled as belonging to the positive class:

$$P=TP/(TP+FP)$$

Recall in this context is defined as the number of true positives divided by the total number of elements that actually belong to the positive class:

$$R=TP/(TP+FN)$$

A precision score of 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C (but says nothing about the number of items from class C that were not labeled correctly) whereas a recall of 1.0 means that every item from class C was labeled as belonging to class C (but says nothing about how many other items were incorrectly also labeled as belonging to class C).
Often, there is an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other.
Usually, precision and recall scores are not discussed in isolation. Instead, either values for one measure are compared for a fixed level at the other measure or both are combined into a single measure.
F-measure is a combination of precision and recall, which is the weighted harmonic mean of precision and recall. (Also called F1 measure since R and P are equally weighted.)

$$F=2.P.R/(P+R)$$

The **sklearn.metrics** module implements several loss, score, and utility functions to measure classification performance.

Some metrics are essentially defined for binary classification tasks (e.g. f1_score).

In extending a binary metric to multiclass problems, the data is treated as a collection of binary problems, one for each class. There are then a number of ways to average binary metric calculations across the set of classes, each of which may be useful in some scenario. How we select the *average* parameter determines the presentation of the measure, for example:

On our RandomForestClassifier (i.e. rfc_model) the four measures are as follows given that we set *average='weighted'* which obligates the metric method to calculate metrics for each class, and find their average, weighted by support (the number of true instances for each class); I assume that these results are more reliable because the distribution of our classes is not uniform and this setting considers the class imbalance.

```
In [33]: from sklearn.metrics import precision_recall_fscore_support
         precision_recall_fscore_support(y_val, y_rfc_model, average='weighted')

Out[33]: (0.86398258745315537, 0.86366724541405604, 0.86132816232109166, None)
```

The outcome indicates that my RandomForestclassifier has P=0.8639, R=0.8636, Fmeasure=0.8613. (The value shown for Support in NA since we are in multiclass extension of this method meanwhile if I set *average=None*, the support for each class will be shown.)

As another experiment let's assign *average=None*, which obligates the metric to calculate measures separately for each class:

```
In [34]: from sklearn.metrics import precision_recall_fscore_support
         precision_recall_fscore_support(y_val, y_rfc_model, average=None)

Out[34]: (array([ 0.87799715,  0.85819527,  0.82657405,  0.81960784,  0.85009862,
                  0.82515131,  0.9114609 ]),
          array([ 0.84073139,  0.91437347,  0.88359281,  0.6656051 ,  0.40355805,
                  0.62252664,  0.80530204]),
          array([ 0.85896027,  0.88539414,  0.8541329 ,  0.73462214,  0.54730159,
                  0.70965876,  0.85509922]),
          array([24173, 32233,  4175,   314,  1068,  1971,  2301], dtype=int64))
```

P
R
F

Here we can present a better outlook on how well RFC model distinguishes between classes:

Class4 Recall is relatively poor. Why?

My hypothesis is that since class4 is so rare, most probably associated soil_type features were among those we deleted; therefore the learning capacity of classifier is affected. Meanwhile from the values of Precision and Recall we realize that we have difficulty distinguishing Class4 from others (R=0.40) means there have been many instances truly class4 that we missed out)

(FN>>FP)

Precision of Class7 is maximum, why?

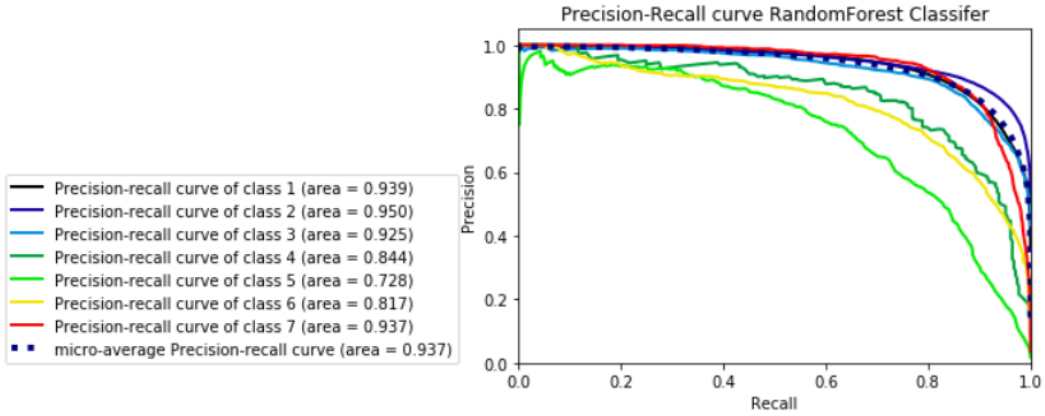
It coincides with Fig.3 page 7: Class7' instances can be well-separated only by the feature of Elevation. (Of course other features are also used, but this 'Elevation' feature makes it precise to pick out positive instances for Class7)

I plotted Precision-recall curves since they are efficient in highlighting differences between models for highly imbalanced data sets (as in our case).

In order to compare different models in imbalanced settings, area under the PR curve will likely exhibit larger differences than area under the ROC curve.

```
In [34]: import matplotlib.pyplot as plt
         import scikitplot as skplt
         y_rfc_model_1 = rfc.predict_proba(X_val)
         skplt.metrics.plot_precision_recall_curve(y_val, y_rfc_model_1)
         plt.legend(loc=(-1, 0), prop=dict(size=10))
         plt.show()
```

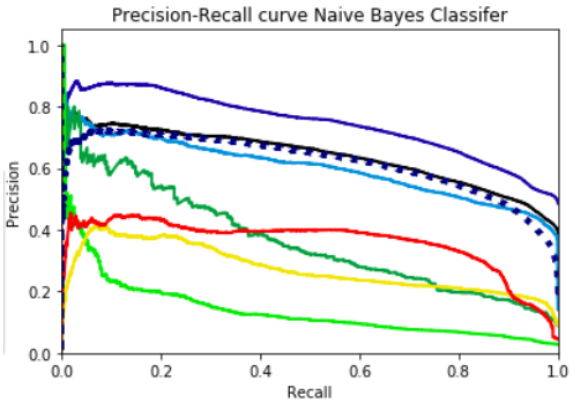
For each classifier I plotted the PR curves for which details refer to the Jupyter notebook, here I try to analyze the observations by the comparison between different models:

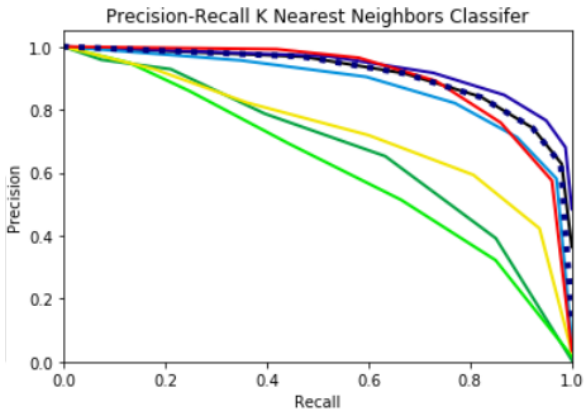


Typically, Precision and Recall are inversely related (ie. as Precision increases, recall falls and vice-versa.) This trade-off between precision and recall can be observed using the precision-recall curve, and an appropriate balance between the two obtained.

Even a hasty look on the three plots proves that RandomForestClassifer works better even on tricky Class4 and Class5.

Among all the models I elaborated, KKN and RandomForest have performed a more robust.





Prediction

Now with the remaining split of dataset (about 400,000 samples) and applying the tuned parameters (k=300 for the Random Forest Classifier) we go through prediction phase which means fitting the tuned classifiers on the (X_test,y_test)

```
rfc = RandomForestClassifier(n_estimators=300, bootstrap=True, oob_score=True)
rfc.fit(X_train,y_train)
y_rfc_model=rfc.predict(X_test)
score= accuracy_score(y_test, y_rfc_Predict)
print score
```

```
0.863235932942
```

Conclusion: in this project we divided the dataset into three splits; trained and valuated the models on these shuffled separated blocks of dataset, tuned the parameters of the classifiers and finally did the prediction on the third partition of data with accuracy 0.86

Further information and explanation can be addressed in the appendix I.

Further steps

The scope of what can be investigated on this problem is so vast that one can dig deep into many other possible machine learning approaches: I am personally psyched to comprehend some facts:

1-Try to realize why SVM proved no satisfactory results although always is suggested as a good candidate classifier in multiclass problems.

I played around with parameter tuning for SVM but never got a proper enhancement in the performance.

2-Investigation of learning curves, in fact how big the training split can be enough given that the distribution is not uniform?

I started out with 15000 samples for training split but then doubled the size, keeping in mind the proper ratio between training and test (33% vs 66%)