MAP553 - Forest Cover Type Prediction.

Florian LABAYE - Florian Labaye

Oscar ROUX - Oscar Roux

1 - Preprocessing and feature extraction :

The first step in any data analysis and prediction problem is exploring and processing the provided datasets in order to understand their particular properties.

The challenge of data cleaning was fortunately insignificant with no missing values at all. Furthermore, using the “X_test.sum()” function, we checked that in the test data set, all features appeared at least once (especially for the “soil_types”). As it was the case, we concluded that all features were useful.

```
data.isnull().sum()
Id                                    0
Elevation                             0
Aspect                                0
Slope                                 0
Horizontal_Distance_To_Hydrology      0
Vertical_Distance_To_Hydrology        0
Horizontal_Distance_To_Roadways       0
Hillshade_9am                         0
Hillshade_Noon                        0
Hillshade_3pm                         0
Horizontal_Distance_To_Fire_Points    0
Wilderness_Area1                      0
Wilderness_Area2                      0
Wilderness_Area3                      0
Wilderness_Area4                      0
Soil_Type1                            0
Soil_Type2                            0
Soil_Type3                            0
Soil_Type4                            0
Soil_Type5                            0
Soil_Type6                            0
Soil_Type7                            0
Soil_Type8                            0
Soil_Type9                            0
Soil_Type10                           0
Soil_Type11                           0
Soil_Type12                           0
Soil_Type13                           0
Soil_Type14                           0
Soil_Type15                           0
Soil_Type16                           0
Soil_Type17                           0
Soil_Type18                           0
Soil_Type19                           0
Soil_Type20                           0
Soil_Type21                           0
Soil_Type22                           0
Soil_Type23                           0
Soil_Type24                           0
Soil_Type25                           0
Soil_Type26                           0
Soil_Type27                           0
Soil_Type28                           0
Soil_Type29                           0
Soil_Type30                           0
Soil_Type31                           0
Soil_Type32                           0
```

```
Soil_Type33                         0
Soil_Type34                         0
Soil_Type35                         0
Soil_Type36                         0
Soil_Type37                         0
Soil_Type38                         0
Soil_Type39                         0
Soil_Type40                         0
Cover_Type                          0
dtype: int64
```

Later on, we tried exploring the different features. All features of the dataset seemed relevant to use for the prediction of the cover type, so we decided to keep them all when we trained our models.

As we intended to use models based on distance calculations, it was necessary to scale our data before training models. Otherwise, the features with higher values would have taken too much importance, and our models would have been less performant.

We also split our training data set in order to assess the performance of different models and the relevance of their hyperparameters. We chose a test size of 0.2 as it is commonly accepted as the most effective size.

```
features = ['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',
       'Wilderness_Area1', 'Wilderness_Area2', 'Wilderness_Area3',
       'Wilderness_Area4', 'Soil_Type1', 'Soil_Type2', 'Soil_Type3',
       'Soil_Type4', 'Soil_Type5', 'Soil_Type6', 'Soil_Type7', 'Soil_Type8',
       'Soil_Type9', 'Soil_Type10', 'Soil_Type11', 'Soil_Type12',
       'Soil_Type13', 'Soil_Type14', 'Soil_Type15', 'Soil_Type16',
       'Soil_Type17', 'Soil_Type18', 'Soil_Type19', 'Soil_Type20',
       'Soil_Type21', 'Soil_Type22', 'Soil_Type23', 'Soil_Type24',
       'Soil_Type25', 'Soil_Type26', 'Soil_Type27', 'Soil_Type28',
       'Soil_Type29', 'Soil_Type30', 'Soil_Type31', 'Soil_Type32',
       'Soil_Type33', 'Soil_Type34', 'Soil_Type35', 'Soil_Type36',
       'Soil_Type37', 'Soil_Type38', 'Soil_Type39', 'Soil_Type40']
X_train = df[features]
Y_train = df['Cover_Type']
scaler = preprocessing.StandardScaler().fit(X_train.values)
X_scaled = scaler.transform(X_train.values)
X_scaled_train, X_scaled_test, y_train, y_test = train_test_split(X_scaled,Y_train.values,random_state=42, test_size=0.2)
```

The first model we used is the multinomial logistic regression. It is based on logistic regression, which is a rather simple model. By starting with this model, we avoid any “black box” effect, we have access to the coefficients of the regression and we can look at the resulting probabilities for each input.

```
model = linear_model.LogisticRegression(multi_class='multinomial', solver='newton-cg')
model.fit(X_scaled, Y_train.values)

df_test = pd.read_csv('/kaggle/input/map553-2022/test-full.csv')
X_test = df_test[features]
X_test_scaled = scaler.transform(X_test.values)
prediction = model.predict(X_test_scaled)

print(confusion_matrix(y_test, model.predict(X_scaled_test)))
```

```
 Confusion matrix :

       [[308,  62,   0,   0,  21,   2,  34],
       [106, 224,   9,   0, 102,  14,   9],
       [  0,   1, 227,  51,  13, 108,   0],
       [  0,   0,  25, 380,   0,  31,   0],
       [  1,  60,  10,   0, 361,  20,   0],
       [  0,   7,  80,  38,  10, 280,   0],
       [ 39,   0,   0,   0,   4,   0, 387]]

Accuracy on the test data taken from the train data set : 71%
```

We submitted these predictions on Kaggle and **the score we obtained was 58%,** which is a very low score. So, we decided to try models based on partitioning, and then more complex models.

2-From decision tree to Gradient Boosting, method optimization

Decision tree :

A test with default parameters and an unlimited depth gives us interesting results. By training it on the training part of the training set and then scoring it on the test part of the training set, we obtain a score of 0.71.

```
model = tree.DecisionTreeClassifier(max_depth=7)
model.fit(X_scaled, Y_train.values)
model.fit(X_scaled_train, y_train)

model.score(X_scaled_test,y_test) = 71%
```

When training it again on the whole training set and submitting it, the score lowers however to 55%.

The problem with decision trees is that they are unstable. Indeed, the structure of the tree may completely change when a small change in the data occurs. To get closer to optimality, we decided to test the Random Forest model.

Random Forest :

- In regards to the results obtained by the decision tree, even after hyperparameters tuning, the next logical step is using a random forest, in order to take care of our overfitting issue.

```
model = RandomForestClassifier(max_depth=7, random_state=42)
model.fit(X_scaled_train, y_train)
print(model.score(X_scaled_test,y_test))
```

This first naïve try with default parameters and a depth of 7 gives us promising results. By training it on the training part of the training set and then scoring it on the test part of the training set, we obtain a score of 0.74. When training it again on the whole training set and submitting it, the score lowers however to 0.57. Given that it is our best result so far, we decided to improve this model using hyperparameters tuning.

We decided to play on multiple parameters in order to be as efficient as possible, by trying to find the best combination of *n_estimators, min_samples_split, min_samples_leaf, max_features, max_depth* and *bootstrap*.

After some aborted attempts using Gridsearch that were taking too long given the number of parameters and different values to test for each one, we decided to use a *randomized search,* that takes less time to give a result, even if it less effective than a gridsearch.

```
param_grid = {'bootstrap': [True, False],
 'max_depth': [3, 5, 7, 9,11,15,20,40,None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': np.arange(50,300,15)}
model = RSCV(RandomForestClassifier(), param_grid, n_iter =150).fit(X_scaled_train, y_train)
```

```
params= model.best_params_
model = model.best_estimator_
```

Due to a maximum value of n_estimators of 1000 initially taken, the random search was still taking too long so we decided to limit the maximum number of trees to 300.

After executing the random search multiple times for 150 iterations, we identified the best parameters, and after some more tests done manually, we obtained the best results for {*n_estimators=500, min_samples_split=2,min_samples_leaf=1, max_features= 'auto', max_depth=None, bootstrap=False}*

By training it on the training part of the training set and then scoring it on the test part of the training set, we obtain a score of 0.88. When training it again on the whole training set and submitting it, the score lowers however to 0.7.

Gradient Boosting :

Given the improvement obtained by going from a simple tree to a random forest, we decided to get a even more complex method in order to obtain better results. We therefore decided to implement a gradient boosting method.

```
model = GradientBoostingClassifier(n_estimators=300, learning_rate=1,
max_depth=1, random_state=42).fit(X_scaled_train, y_train)
model.score(X_scaled_test, y_test)
```

This first test with default parameters gave us a result of 0.69, which is surprisingly not as promising as the random forest first test.

For parameter tuning, we started reading the documentation GradientBoostingClassifier, searching on the internet for some combinations used for similar problems and decided to first do a gridsearch on only the learning rate, as it seemed to have the most impact on the results. The best result was for a learning rate of 0.4, which gave a result of 0.78, still not enough to compete with the random forest.

We therefore tried using an hyperparameters tuning function from *sklearn* library.

```
param_grid={'n_estimators' : [100,200, 300, 400, 500], 'max_depth' : [2, 3, 5, 10, 15], 'learning_rate' : [1, 0.5,0.1,0.05,0.01]}
model = RSCV(GradientBoostingClassifier(),param_grid, n_iter=50).fit(X_scaled_train,y_train)
params= model.best_params_
model = model.best_estimator_
print (params)
```

The best parameters given by the tuning are `{'n_estimators': 200, 'max_depth': 15, 'learning_rate': 0.5}`

This gives us a score of 0.86 on the test data and 0.74 on the submission, still not a result as accurate as the random forest.

AdaBoost :

In order to find a better alternative for XGBoost, we decided to try an AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier
model = AdaBoostClassifier(n_estimators=10, random_state=42)
model.fit(X_scaled_train, y_train)
print(model.score(X_scaled_test,y_test))
```

We obtain the best result for n_estimators=10 with a score of 0.51 on the test part of the training set, with higher values over-fitting the model and reducing the score. This result oven after tuning n_estimator seemed too low to compete with our random forest with more in depth parameters tuning, so we decided to give up on this method.

Bibliography :

- scikit-learn.org
- Hyperparameter Tuning the Random Forest in Python | by Will Koehrsen | Towards Data Science
- *Intelligence artificielle vulgarisée,* Aurélien Vannieuwenhuyze, Editions ENI, 2019