Supervised Machine Learning with Random Forest in Python

Random Forest is a versatile and powerful ensemble learning method that combines the strengths of multiple decision trees to improve predictive accuracy and robustness. In this segment, we will move beyond the theory and learn how to implement Random Forest in Python.

Here is the step-by-step process to understand how it works before we move on to the practical implementation in Python:

So, step 1 is to pick Random K Data Points from the Training Set. The next step is to build the decision tree associated with these K data points. Next, in step 3, we choose the number N of trees you want to build and repeat steps 1 and 2. Lastly, in step 4, for a new data point, have each N tree predict its category and assign the new data point to the category that wins the majority vote.

Now that we have revisited the theoretical steps, let's see how to implement this algorithm in Python. We will use the same dataset we used in the Decision Tree video: the 'Social Network Ads' dataset. This dataset contains information about 400 customers from a car dealership that has just released a new luxury SUV. The dealership's strategy team wants to understand which customers are most likely to buy the SUV so they can target these customers with ads on social networks. Here's a brief overview of the dataset:

- Each row represents a different customer.
- We have two key features for each customer: age and estimated salary.
- The dependent variable is 'Purchased', indicating whether the customer bought an SUV before. A value of 0 means they did not buy a previously launched SUV, and a value of 1 means they did.

Our goal is to train a classification model to predict whether a customer will likely buy the new SUV based on age and estimated salary. This prediction helps the strategy team target potential buyers with tailored social network ads.

Let’s examine the Python implementation and see how the Random Forest algorithm can help us achieve this!

For the implementation, the initial steps in our Python code remain the same as in the Decision Tree segment, up to the point of feature scaling. We’ll start by importing the dataset, preprocessing the data, and scaling the features. After that, we will build and evaluate the Random Forest model.

I encourage you to pause the video here and write the code to the feature scaling part from the Decision Tree segment.

Okay, let’s continue building our model.

Now that we have our data preprocessed let's move on to the training phase of our Random Forest Classification model using the preprocessed data.

```
from sklearn.ensemble import RandomForestClassifier
```

First, we import the RandomForestClassifier class from scikit-learn's ensemble module. This class provides the tools to create and train a Random Forest model, an ensemble method that combines multiple decision trees to improve the model's performance. Remember, scikit-learn provides a comprehensive toolkit for various machine learning tasks, including Random Forest Classification. We can leverage this library's functionalities to streamline the development process.

If you’re interested in more detailed information about model development and the various parameters we can tune, you can refer to the scikit-learn documentation or search for tutorials and guides online.

Scikit-learn's extensive resources are invaluable for understanding and implementing machine learning models effectively.

Continuing with the model building process, in the next step, we create an instance of the RandomForestClassifier class we just imported.

classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)

This line of code is pivotal in setting up our Random Forest model with specific parameters for training. Here The RandomForestClassifier class from scikit-learn's ensemble module is used to create a Random Forest model with n_estimators=10 specifying the number of decision trees (in this case, 10) for improved accuracy. The criterion='entropy' parameter measures impurity at each split in the trees using information gain to make the most effective splits. Finally, random_state=0 sets a seed for the random number generator, ensuring reproducible results if you run the code multiple times.

Now that our classifier is set up with the specified parameters, we can train it on our preprocessed data:

classifier.fit(X_train, y_train)

This line of code is where the actual training happens. The fit method takes two inputs: X_train, which contains our feature variables (like age and estimated salary), and y_train, which holds the corresponding target variable (whether the customer purchased the SUV or not). By calling classifier.fit(X_train, y_train), we train our Random Forest model on the training dataset, allowing it to learn the underlying patterns that associate the features with the target variable. This trained model can then be used to make predictions on new data. Once our Random Forest model is trained, we can utilize it to make predictions on new data.

Example: predicting whether a 30-year-old customer with an estimated salary of 87,000 will purchase an SUV.

```
print(classifier.predict(sc.transform([[30,87000]])))
```

In this code snippet, we use the trained classifier to predict the purchase decision for a new customer. Before making this prediction, we preprocess the input data by applying the same scaling transformation that was applied to our training data (sc.transform([[30, 87000]])).

This step ensures consistency in the data scale between the training and the new data points.

The predict method of the classifier returns [0], indicating that the model predicts this particular 30-year-old with an estimated salary of 87,000 is not likely to purchase the SUV. To validate this prediction, we can refer to the Social_Network_Ads.csv dataset, which indeed shows that the actual purchase decision for a 30-year-old with an estimated salary of 87,000 is 0 (not purchasing the SUV). This confirms our model's prediction, demonstrating its accuracy in identifying purchasing behaviors based on the learned patterns from the training phase.

Having successfully predicted the outcome for a single customer, it’s essential to assess how well our model performs on a broader set of data. We can evaluate our model's performance on the entire test set to determine its generalization capability and overall effectiveness in making accurate predictions on unseen data.

```
y_pred = classifier.predict(X_test)
```

Here, y_pred is a NumPy array containing the predicted purchase decisions (either 0 for not purchasing or 1 for purchasing) for each customer in the testing set. The testing set, X_test, was not used during the training process, so these predictions provide a clear indication of the model's performance on new, unseen data.

These predictions reflect what our model has learned about which customers are likely to purchase the SUV based on their age and estimated salary.

Next, we will compare these predicted values (y_pred) with the actual outcomes (y_test) to evaluate our model’s accuracy and performance. This comparison will reveal how well the model's predictions align with the real purchase decisions, thus giving us insight into its effectiveness in a real-world context.

```
print(np.concatenate((y_pred.reshape(len(y_pred),1),
y_test.reshape(len(y_test),1)),1))
```

Here by using np.concatenate, we combine these reshaped arrays (y_pred and y_test) side by side. This creates a new array where each row displays the predicted purchase decision next to the actual purchase decision for each customer in the test set. Printing this concatenated array allows us to visually inspect how well the model's predictions align with the real purchase behaviors.

Each row shows the model's predicted purchase decision (y_pred) next to the actual purchase decision (y_test) for a customer in the test set. This allows us to see how well the predictions match reality.

- A pair like [0 0] means the model correctly predicted "no purchase" for the customer.
- A pair like [1 1] indicates the model correctly predicted a "purchase."
- Discrepancies like [1 0] or [0 1] show where predictions differed from actual outcomes.

By visually inspecting this table, we can gauge the model's performance. Ideally, we would observe a high number of matching pairs (where the predicted value in the first column matches the actual value in the second column), which suggests the model is accurately predicting purchase behaviors for new, unseen data.

In cases where the model predicted a purchase (1) but the actual outcome was no purchase (0), or vice versa, it shows areas where the model's predictions could be improved or where the data patterns are inherently more complex and challenging to predict accurately. This comparative analysis provides valuable insights into how well our model generalizes and where it might require further refinement to improve accuracy.

Moving on

To quantify our model's performance further, we will now calculate the confusion matrix and accuracy score. These metrics give a more detailed statistical overview of the model's predictive capabilities:

```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
```

When we run this code, we obtain the following output:

[Show the output]

First we see the confusion matrix. Next the accuracy score of 0.91 (or 91%) shows that the model correctly predicted 91% of the cases, which is similar to the performance of the decision tree model we previously evaluated.

We can also improve the accuracy of our Random Forest model, and can consider one of the several strategies such as increasing the number of n_estimators: This parameter controls the number of trees in the forest. More trees can potentially capture more patterns and provide better predictions, though it also increases computational complexity. Implementing strategies like these can refine the model further and enhance its predictive power. To gain a deeper understanding of our model's decision boundaries and how well it separates the different classes (purchase vs. no purchase) in the training data, we can use data visualization techniques. Let's first visualize the training set results to explore how our Random Forest model categorizes customers based on their age and estimated salary:

```
from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:, 0].max() + 10, step = 0.25), np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() + 1000, step = 0.25))
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Random Forest Classification (Training set)')
X_set, y_set = sc.inverse_transform(X_test), y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:, 0].max() + 10, step = 0.25), np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() + 1000, step = 0.25))
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Random Forest Classification (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

In this test set plot, we observe how the model's predictions hold up against new data. Even though the model is well-trained, we can still see some green points (customers predicted to purchase) falling into red regions (where the model predicts no purchase), and vice versa. This is a common scenario in real-world applications where no model can perfectly classify all data points due to overlapping patterns. While our model achieves an accuracy of 91%, you can experiment with different parameters, like increasing the number of trees (n_estimators), to see if you can further improve its performance. Feel free to tinker with the parameters of the models we've covered so far, and let us know if you can improve the accuracy from the current 91%.

This concludes our exploration of the Random Forest model for predicting customer purchases. Remember, the best model for one dataset may not be the best for another. Use Decision Trees for simpler problems or when you need easy-to-understand results. Opt for Random Forests for more complex problems or when you need higher accuracy and can handle more computation.

It's essential to try different models and parameter settings to find the optimal solution for your specific data.

Now that we have already trained and run our KMeans algorithm, we need to compute the WCSS (Within-Cluster Sum of Squares) value. We do this by first taking our wcss list, which is so far initialized as an empty list. Then, we use the append function to add a new value inside the list. The first value we add is the WCSS value computed for 1 cluster, which is actually the sum of the squared distances between all the observation points and the centroid of the single cluster.

Alright, let's append. Since it's a function, we're adding some parenthesis. The way to get the WCSS value is to call an attribute of the KMeans object called inertia_, which will give us exactly that WCSS value. In order to get that attribute, we just need to call our object first, because when we want to get an attribute, we always have to call it from the object, and then we add a dot and then we call the attribute which is here inertia_.

Therefore,

Google Collab

```
wcss.append(kmeans.inertia_)
```

This completes one iteration of the loop. As the loop progresses, we'll train the KMeans model with different i values (different numbers of clusters) and keep appending the WCSS for each iteration. This will allow us to plot the elbow method graph in the upcoming steps!

Now, we're ready to plot a simple curve that will visualize the WCSS values for different numbers of clusters from 1 to 10. This curve will depict the WCSS on the y-axis, and the number of clusters used will be on the x-axis.

Therefore,

Google Collab

```
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

plt.plot function from the matplotlib library which is abbreviated as plt is used for creating plots.

range(1, 11) creates a sequence of numbers from 1 to 10 which represents the different numbers of clusters used in the loop.

wcss refers to the list we've been populating with WCSS values throughout the loop iterations.

Next the plt.title adds a title to our graph, labeling it "The Elbow Method".

plt.xlabel('Number of clusters') labels the x-axis of the graph, indicating it represents the number of clusters used. And

plt.ylabel('WCSS') labels the y-axis of the graph, indicating it represents the WCSS values.

Lastly plt.show() displays the graph we just created.

Now let’s run this code cell and plot the elbow method graph.

[Show the graph]

So, after seeing the elbow method graph, how would we choose the optimal number of clusters?

Here we need to identify the "elbow point," which is the point where the rate of decrease in WCSS slows down significantly. This point indicates the optimal number of clusters for our data.

So, after seeing the elbow method graph, how do we choose the optimal number of clusters?

The elbow method helps us visually identify a good stopping point for the number of clusters. Ideally, the curve should have a distinct "elbow" shape. The optimal number of clusters is often chosen at the point where the curve starts to bend sharply. This indicates that adding more clusters after this point results in diminishing returns, meaning the WCSS reduction is less significant for each additional cluster.

In our example, the curve seems to be flattening out around 5 clusters. Based on the elbow, this suggests that 5 will be the optimal number of clusters for our data.

Now we re-train the KMeans model with the optimal number of clusters.

This involves initializing a new KMeans object with the optimal number of clusters which is five and fitting it to your dataset.

Therefore,

Google Collab

kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)

The rest remains the same as done previously, and just the n_clusters parameter is initialized to 5 instead of using the loop variable i. This ensures that the model is trained with the optimal number of clusters identified through the elbow method. We keep the same way of starting the sorting process (init='k-means++') and the random order we started with (random_state=42) for consistency.

Next, we use the trained KMeans model to predict which cluster each data point belongs to. This is done by calling the fit_predict method on the KMeans object with our dataset X.

Therefore,

Google Collab

y_kmeans = kmeans.fit_predict(X)

This line of code performs two crucial actions in a single step:

First .fit() method trains the KMeans model based on our actual data (X), making sure each cluster has similar data points with similar characteristics. And

After training, .predict(X) assigns a label to each data point. So, for every data point in our dataset, the model assigns a label (or cluster number) indicating which group it belongs to.

The variable y_kmeans stores these cluster labels for each data point.

So, in summary, the trained KMeans model (kmeans) predicts which cluster each data point in dataset X belongs to, and stores the predicted cluster labels in the array y_kmeans.

Now, let's visualize the clusters identified by the KMeans algorithm. Since we have the cluster labels (y_kmeans) for each data point, we can create a scatter plot. This plot will represent each data point as a dot, and we'll use color to differentiate between the clusters based on their labels. On the X-axis, we'll have the annual income, which is our first feature, and on the Y-axis, the spending score.

To accomplish this, we'll create several scatter plots, one for each cluster. This means we'll plot the data points belonging to cluster 1, then cluster 2, and so on, up to cluster 5.

We'll achieve this by utilizing the scatter function from the matplotlib.pyplot module.

Therefore,

Google Collab

plt.scatter()

Since we are using the scatter function to plot each of the five clusters, we'll call this scatter function five times with different inputs. So, now we're starting with cluster number 1.

Google Collab

plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')

The First Argument X[y_kmeans == 0, 0] selects the X-axis coordinates for the data points we want to plot. Remember X here is the matrix of two columns containing the annual income in the first column and spending score in the second column.

[y_kmeans == 0, 0] filters the data points based on their cluster labels (y_kmeans). We're using a condition == 0 to specifically select only those points where the label is 0 (which corresponds to cluster 1).

Next this, 0 tells the function to pick only the first feature which is the annual income from the selected data points. So, it picks the X-coordinate (first feature) for each point in cluster 1.

The Second Argument X[y_kmeans == 0, 1] selects the Y-axis coordinates for the data points we want to plot, similar to the first argument.

In X[y_kmeans == 0, 1] here, 1 tells the function to pick the second feature which is the spending score from the selected data points in cluster 1. So, it picks the Y-coordinate for each point in cluster 1.

Next s = 100 sets the size of the dots representing each data point. Here, s = 100 makes the dots a bit larger for better visibility. You can adjust this value to control the dot size.

Then c = 'red' argument sets the color of the dots for this scatter plot. We're using c = 'red' to represent cluster 1 with red dots. You can choose different colors for other clusters.

Next label = 'Cluster 1' assigns a label to the cluster, which will be used later in the legend (a small box showing the meaning of each color). Here, label = 'Cluster 1' provides a clear identification for the red dots representing cluster 1.

Alright, now let’s do the same for the other clusters. We'll just copy this line of code and paste it right below to plot the second cluster.

Google Collab

Copy paste above line of code and then make the changes as mentioned.

And now, to plot the second cluster, we just have to change:

y_kmeans == 0 to y_kmeans == 1 to select the data points belonging to cluster 2.

Change the color c to differentiate it from the first cluster, let's say 'blue'.

And Update the label to 'Cluster 2' instead of 'Cluster 1' to reflect the change in cluster number.

Google Collab

```
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
```

now let’s do the same for cluster 3, 4 and 5 as follows.

Google Collab

```
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
```

Now that we've created scatter plots for each individual cluster, let's take a look at the centroids. Remember, centroids are the representative points for each cluster, like the "hearts" or "centers of gravity" that define the core of each group.

To visualize these centroids, we'll use the scatter function again, but this time, we'll provide the coordinates of the centroids themselves. These coordinates are stored in the kmeans.cluster_centers_ attribute of the KMeans model we trained. We'll do this by adding the following line of code:

Google Collab

```
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s = 300, c = 'yellow', label = 'Centroids')
```

here kmeans refers to the KMeans model we've already trained. It likely holds information about the clustering process.

cluster_centers_ is an attribute (a property) of the kmeans model. In the context of KMeans clustering, this attribute specifically stores the coordinates of the centroids.

Next [:, 0] selects all rows (":") and the first column (0) from the "cluster_centers_" attribute. This gives us the x-coordinates of the centroids.

Similar to the previous [:, 1], select all rows and the second column (1), which gives us the y-coordinates of the centroids.

s = 300 sets the size of the marker for each centroid to 300 points. This will make them visually distinct from other data points.

c = 'yellow' sets the color of the centroid markers to yellow. And lastly

label = 'Centroids' assigns a label "Centroids" to this data series, which will be used in the legend if one is created.

In summary, this code snippet extracts the centroids' coordinates from the trained KMeans model and uses them to create a scatter plot with yellow markers sized at 300 points, labeled as "Centroids". This helps visualize where the centroids lie in the data space.

.

Then, we add finishing touches to the plot

[GC]

```
plt.title('Clusters of customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()
```

where

plt.title('Clusters of customers'): Sets the title of the plot as 'Clusters of customers'.

plt.xlabel('Annual Income (k$)'): Labels the x-axis as 'Annual Income (k$)' to indicate the data represented on the x-axis.

plt.ylabel('Spending Score (1-100)'): Labels the y-axis as 'Spending Score (1-100)' to indicate the data represented on the y-axis.

plt.legend(): Adds a legend to the plot to distinguish between clusters and centroids.

plt.show(): Displays the plot.

Now let’s execute this code cell in order to visualize the clusters.

So, with these additions, we can clearly observe the five clusters of customers. Now, let's delve into their analysis. Our goal here is to cluster these customers effectively to gain insights into their behavior. By understanding these clusters, we aim to optimize our business strategies by tailoring offers to each group. Alright, let's proceed.

Let's start by examining each cluster. Cluster 4 comprises customers with low annual income and minimal spending at the mall. Moving on to cluster 5, we find customers with high annual income but low spending. Next, cluster 1 consists of customers with low annual income but high spending scores. Cluster 3, on the other hand, includes high-income customers who spend generously at the mall. Lastly, cluster 2 represents an average group, with customers earning a moderate income and spending moderately at the mall.

Now, armed with this understanding, we can implement targeted marketing strategies to cater to the unique preferences of each cluster. For instance, we can offer discounts and promotions to cluster 1 to encourage their high spending behavior, while focusing on introducing premium products or services to cluster 3 to capitalize on their high-income status. Similarly, we can devise specific incentives for clusters 4 and 5 to increase their spending levels. By tailoring our marketing efforts accordingly, we can maximize engagement and drive profitability across all customer segments.

And this is just a starting point, and further analysis might reveal additional insights within each cluster. By understanding these customer segments, we can craft targeted strategies to strengthen relationships, optimize marketing efforts, and ultimately drive business growth while upholding our moral responsibility to our customers and society.

And that’s all in this part. Thankyou