Project 5 - Application flow identification

Giuseppe Gattulli, Filippo Kubler
Person codes: 10668469 - 10656370

Contents

Raw data visualization/analysis

To begin, our initial focus was on comprehending the structure of the dataset and conducting preprocessing steps to retain only the relevant features. Let's break it down step by step:

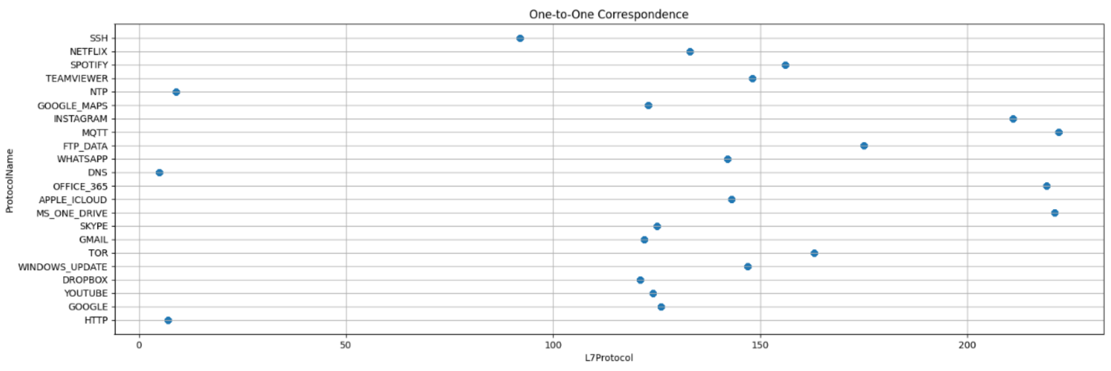
1. Removing irrelevant columns: the code identifies a set of columns that are considered not meaningful for the analysis. These columns, specified in the *columns_to_delete* list, are dropped from the dataframe *df*. These features are considered irrelevant because they relate primarily to specific capture rather than providing valuable insights in distinguishing between different application flows.

2. Handling missing values: the code removes rows and columns with NULL values from *df*. It first drops any rows containing NULL values using the *df.dropna(inplace=True)* function. Then, it drops any columns containing NULL values using *df.dropna(axis='columns')*.

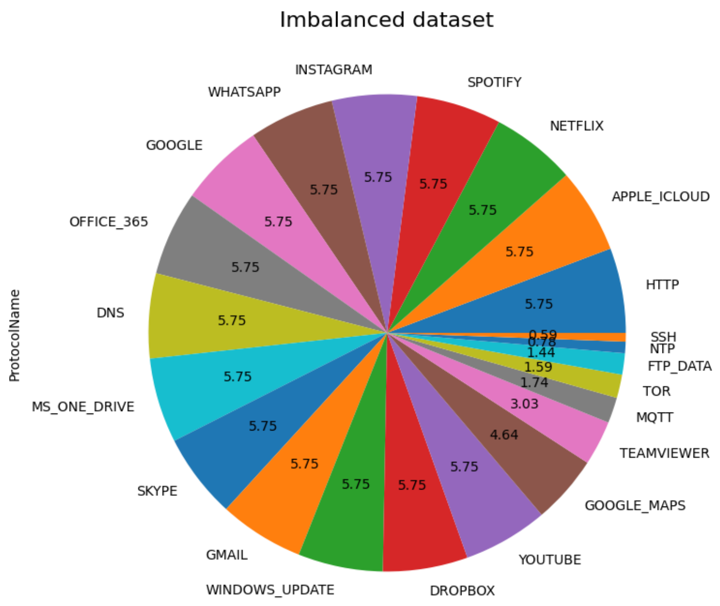
3. Removing columns with all zeroes: the code identifies columns in the dataframe that contain all zeroes and removes them using the *df.loc[:, (df != 0).any(axis=0)]* syntax. This keeps only the columns that have at least one non-zero value. Then, it displays the columns that were removed.

4. Analyzing the dataset: the code performs various analyses on the dataset. It calculates the number of different application flows (22) by finding the unique values in the *'ProtocolName'* column. It also counts the number of samples for each application and determines the maximum number of samples for an application (1000).

5. Visualizing data correspondence: the code creates a scatter plot to show the relationship between *'L7Protocol'* and *'ProtocolName'*. It uses *plt.scatter()* to plot the two columns against each other. By observing this pattern, it's possible to see that there is a one-to-one correspondence between the columns. Hence, in the subsequent analysis, only the '*ProtocolName*' column is used as the label for the machine learning algorithms.



6. Visualizing class imbalance: the code creates a pie chart to visualize the class imbalance in the *'ProtocolName'* column. It counts the occurrences of each unique value using *y.value_counts()* and plots a pie chart with percentage labels. As it's possible to see, 7 applications have less samples than the others.



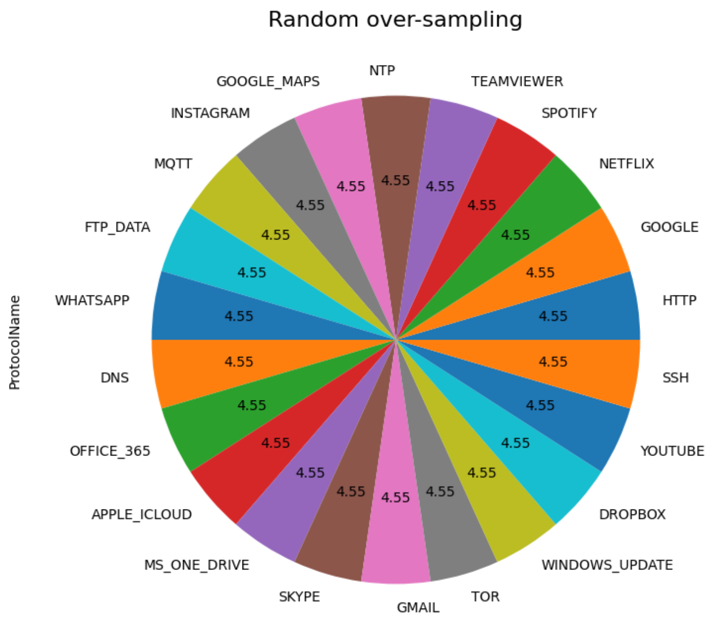
Data preprocessing

Having highlighted the problem of dataset imbalance, the purpose of this section is to evaluate the effectiveness of various over-sampling techniques in achieving a balanced dataset. The following steps are undertaken:

1. ***correct_oversample()*** function: this function is defined to verify if the number of samples for each application is equal to the *max_samples* value, which represents the maximum number of samples among all applications. It counts the number of rows with a specific label and compares it with *max_samples*. If the counts are equal for all applications, it prints a message indicating that all applications have the desired number of samples.

Otherwise, it prints the labels of applications that have a different number of samples and returns 1 to indicate an imbalance.

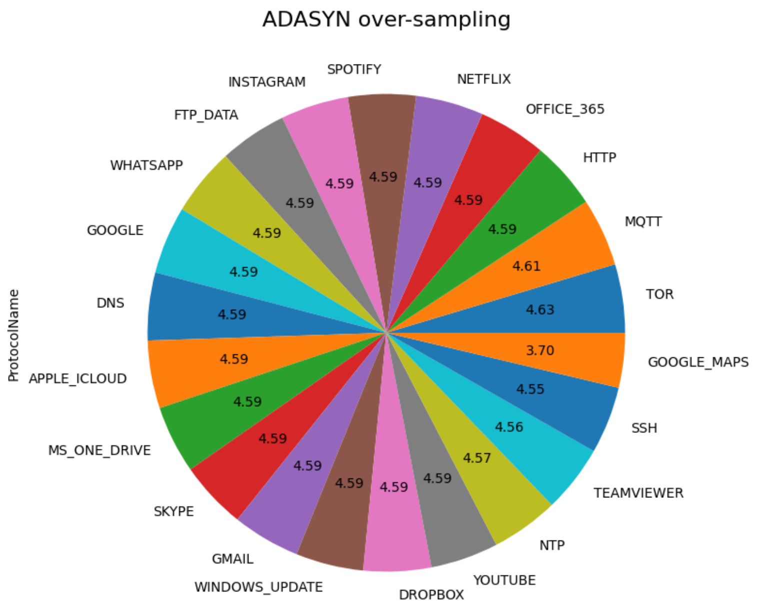
2. **Random Over-sampling**: the code employs the Random Oversampler with *sampling_strategy='not majority'* (which resamples all classes but the majority classes) to randomly over-sample the dataset. It applies the over-sampling to both the feature matrix (*X*) and the target variable (*y*) using *ros.fit_resample(X, y)*. The function *correct_oversample()* is then called to check if the over-sampling was done correctly. If the balance is achieved (*check == 0*), the balanced features and labels are combined into a new dataframe named *df_rand*. Then, a pie chart is created to visualize the balance achieved through random over-sampling. This shows the distribution of application flows in the over-sampled dataset.



3. **SMOTE Over-sampling**: the code applies the Synthetic Minority Over-sampling Technique (SMOTE) using *SMOTE(sampling_strategy='not majority', k_neighbors=5)*. The *k_neighbors* parameter select a number of nearest neighbors used to define the neighborhood of samples to use to generate the synthetic samples. It performs the over-sampling on X and y with smote.fit_resample(X, y). The *correct_oversample()* function acts in the same way as the random over-sampler, the balanced features and labels are stored in the dataframe *df_smote*. With perfect balance, the pie chart turns out to be the same as the previous case, so it's not shown again.
4. **Borderline-SMOTE Over-sampling**: the code applies the Borderline-SMOTE technique using *BorderlineSMOTE(sampling_strategy='not majority', k_neighbors=5)*. The result achieved is the same as the SMOTE case, with the creation of a new balanced dataframe called *df_bsmote*.
5. **Finding the best number of neighbors for ADASYN**: the code defines the *find_best_n()* function to determine the optimal number of neighbors for the Adaptive Synthetic Sampling (ADASYN) technique. The function attempts various numbers of neighbors (*n_neighbors*) and calculates the difference between the proportion of each application and the desired balanced proportion. It aims to find the *n_neighbors* value that yields the most balanced dataset. The best *n_neighbors* value is stored in the *best_n* variable. The

objective of this function is to enable the generation of an ADASYN dataframe that closely resembles the other dataframes obtained through alternative oversampling algorithms. This ensures a fair comparison in the final analysis.

6. **ADASYN Over-sampling**: Using the best *n_neighbors* value obtained in the previous step, the code applies multiple times the ADASYN technique for over-sampling with *ADASYN(sampling_strategy='minority', n_neighbors=best_n)*. In this case, the *sampling_strategy* is '*minority*' which resamples only the minority class, that's why it has to be repeated for every minority class. This approach is adopted because with any other type of *sampling_strategy* the ADASYN algorithm wouldn't work correctly considering the dataframe already balanced. As a result, a new dataframe named *df_adasyn* is created. However, unlike the initial dataframe, *df_adasyn* exhibits improved balance, although it may still retain some degree of imbalance. This observation is supported by the visual representation of the data through the pie chart:



After analyzing the code and its implementation of various oversampling techniques, it is essential to discuss their effectiveness in addressing class imbalance.
Oversampling techniques are employed to alleviate class imbalance, where one or more classes have significantly fewer samples compared to the others. These techniques aim to increase the representation of minority classes samples to achieve a more balanced dataset, which can lead to better model performance and mitigate bias towards the majority classes.

- Random Over-sampling: this technique randomly duplicates samples from the minority classes until it matches the majority classes's sample count. Although straightforward, it may result in overfitting and a higher risk of introducing noise into the dataset.
- SMOTE: it creates synthetic samples by interpolating between neighboring instances of the minority classes. This technique generates artificial samples by considering the feature space, resulting in a more diverse oversampled dataset.
- Borderline-SMOTE: building upon SMOTE, Borderline-SMOTE identifies "borderline" instances, which are close to the decision boundary between classes, and synthesizes samples in their vicinity. This approach aims to improve the quality of synthetic samples and increase the separation between classes

- ADASYN: it focuses on generating synthetic samples by considering the distribution of minority class instances. It generates more samples in regions where the class density is low, effectively adapting to the underlying distribution. ADASYN can be particularly useful when the density of minority class instances varies across the feature space.

It is crucial to note that no oversampling technique is universally superior, as their effectiveness depends on the specific dataset and problem domain. Therefore, it is recommended to evaluate the impact of different oversampling techniques on the model's performance to determine the most suitable approach for a given scenario.

ML models optimization and training

In this section the focus is on the optimization of machine learning models using Random Forest and K-Nearest Neighbors (KNN) algorithms.

Optimizing Random Forest

The code begins by defining the *optimize_RandomForest* function. It takes a dataframe as input, which represents the dataset. Here's an overview of the steps involved:

1. Data preprocessing: the input dataframe is divided into features (*X*) and the target variable (*y*). The features are scaled using *StandardScaler* to ensure they have zero mean and unit variance.
2. Train-test split: the dataset is split into training and testing sets using a 80:20 ratio. The training set will be used to train the model, and the testing set will be used to evaluate its performance.
3. Hyperparameter optimization: the *hyperopt* library is used to perform hyperparameter optimization for the Random Forest model. The *fmin* function is employed to search for the best combination of hyperparameters by maximizing the cross-validation accuracy. The hyperparameters considered are the number of estimators (*n_estimators*), criterion (*crit*), and maximum depth (*maxd*). By changing these hyperparameters, we can assess how varying the number of decision trees, the splitting criterion, and the maximum depth of each tree affects the overall behavior and accuracy of the random forest model.
4. Model training: using the best hyperparameters obtained from the optimization step, a new Random Forest model is trained on the entire training set (*X_train* and *y_train*).
5. Model evaluation: the accuracy of the trained model is evaluated using cross-validation. The best hyperparameters, cross-validation accuracy, and training duration are printed to provide insights into the model's performance.
6. Saving the model: finally, the trained Random Forest model is saved using the pickle library for future use.

Optimizing K-Nearest Neighbors (KNN)

The code then proceeds to optimize the KNN model using a similar approach. The *optimize_KNN* function is defined, which takes a dataframe as input. The main difference between the two optimizations is in the hyperparameter optimization step, in fact in this case the hyperparameters considered are the number of neighbors (*n_neighbors*) and the weight function (*weights*). The number of neighbors determines the number of data points that will be considered when making predictions. By adjusting this hyperparameter, we can control the level of local influence on the predictions. The weight function determines how the neighbors are weighted when making predictions. Changing the weight function allows us to prioritize certain neighbors and potentially improve the accuracy of the model in specific scenarios.

Testing the performance

After completing the training phase for multiple models, it is essential to determine which classifier and oversampling technique yield the best results. In order to make an informed decision, the evaluation process is conducted using two different approaches, each providing valuable insights into the performance of the models.

1. Performance metrics

The *metrics* function calculates various performance metrics for the trained Random Forest and K-Nearest Neighbor (KNN) models on different datasets. Here's an overview of the function:

- The function takes a dataframe and a name as input. The dataframe represents the dataset, and the name is used for saving the model files.
- The dataframe is divided into features (*X*) and the target variable (*y*). The features are standardized using *StandardScaler*.
- The function loads the pre-trained Random Forest and KNN models from the JSON files based on the provided name.
- Predictions are made on the test set using both models.
- Various metrics such as mean squared error (MSE), mean absolute error (MAE), accuracy, precision, recall, and F1-score are calculated for both models.
- The calculated metrics are stored in a dictionary (*final_metrics*).
- The function returns the calculated metrics as a dataframe.

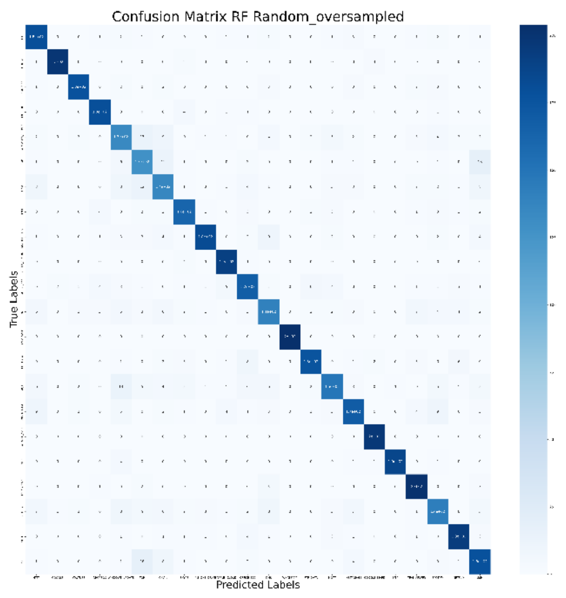
2. Confusion matrix

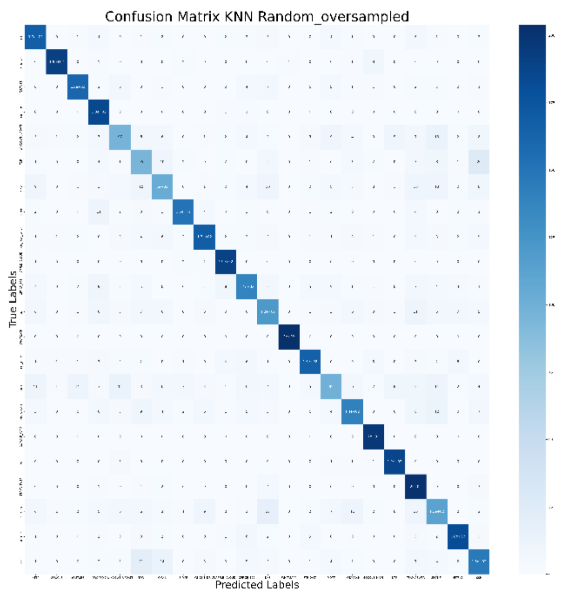
The *conf_matrix* function generates and saves the confusion matrix for the trained Random Forest and KNN models on different datasets. Here's an overview of the function:

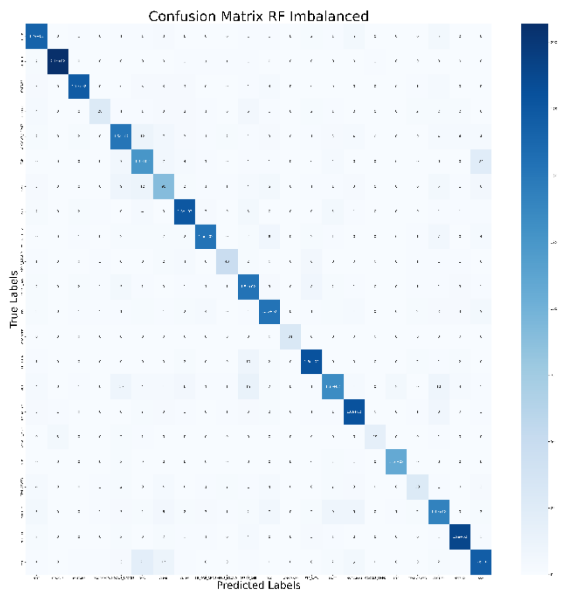
- The function takes a dataframe and a name as input, similar to the *metrics* function.
- The dataframe is divided into features (*X*) and the target variable (*y*). The features are standardized using *StandardScaler*.
- The function loads the pre-trained Random Forest and KNN models from the JSON files based on the provided name.
- Predictions are made on the test set using both models.
- The confusion matrix is calculated for both models using the true labels and predicted labels.
- The confusion matrix is visualized using a heatmap plot and saved as an image file.

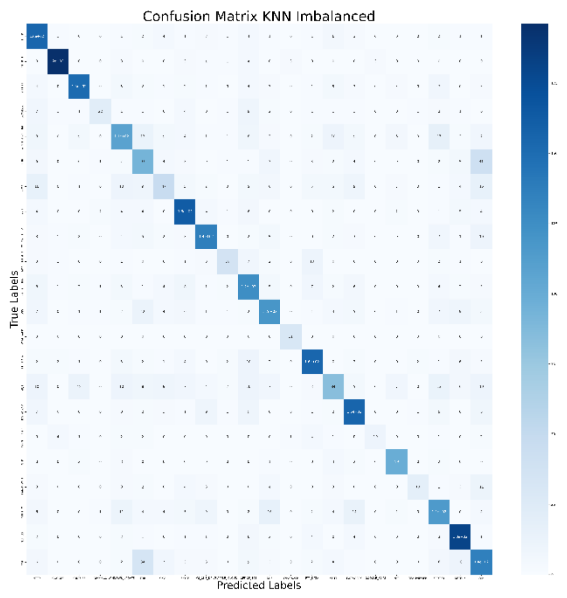
The code then calls both functions to evaluate and visualize the performance of the trained models on different datasets.

ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.815517	0.825429	0.795383	0.806181
	Random	0.863636	0.863483	0.863740	0.862567
	SMOTE	0.841136	0.842138	0.841018	0.840467
	BorderlineSMOTE	0.845682	0.845602	0.845608	0.844457
	ADASYN	0.847053	0.845265	0.844166	0.843783
K Nearest Neighbor	Imbalanced	0.711207	0.715599	0.691307	0.698769
	Random	0.777273	0.773676	0.776920	0.771325
	SMOTE	0.744545	0.740396	0.744072	0.740341
	BorderlineSMOTE	0.755000	0.750225	0.754243	0.749793
	ADASYN	0.739051	0.730418	0.733655	0.729886









The performance metrics indicate that the difference in accuracy between the imbalanced dataset and the balanced datasets is not as significant as anticipated. This observation aligns with the findings from the confusion matrices associated with the different datasets. Upon analyzing the imbalanced dataset and the randomly oversampled dataset, it becomes evident that there is no substantial prediction error for any specific class in either case.

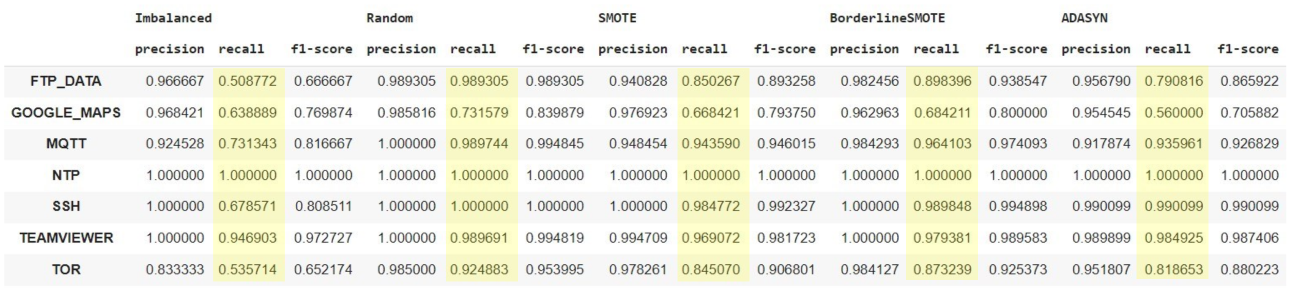
The significant difference lies in the fact that, in the case of imbalance, the number of samples for certain classes is considerably smaller compared to the random oversampled dataset. This disparity has a notable impact on the calculation of accuracy because the error is magnified due to the limited number of samples available for those classes.

3. Per-class classification metrics

Then, we conduct a comparative analysis of per-class classification metrics using various starting dataframes and sampling techniques. The evaluation is performed utilizing a trained random forest model.

- The *per_class_metrics* function takes three arguments: *dataframe* (the input dataframe), *name* (a name for the model), and *to_augment* (a list of labels to consider for filtering).
- The dataframe is divided into features (*X*) and the target variable (*y*). The features are standardized using *StandardScaler*. The pre-trained Random Forest model is loaded from the JSON files based on the provided name. Predictions are made on the test set using the model.
- It filters the predictions and ground truth labels based on the *to_augment* list and it calculates classification metrics (precision, recall, F1-score) per class using *classification_report*, with *output_dict=True* to obtain a dictionary of metrics.
- The accuracy metric is printed, and the classification metrics are stored in a pandas dataframe and returned by the function.

By exclusively considering the original imbalanced labels, our aim is to emphasize the disparity between the initial imbalanced dataframe and the subsequent balanced dataframes generated by different sampling techniques.



	Imbalanced			Random			SMOTE			BorderlineSMOTE			ADASYN		
	precision	recall	f1-score	precision	recall	f1-score	precision	recall	f1-score	precision	recall	f1-score	precision	recall	f1-score
FTP_DATA	0.966667	0.508772	0.666667	0.989305	0.989305	0.989305	0.940828	0.850267	0.893258	0.982456	0.898396	0.938547	0.956790	0.790816	0.865922
GOOGLE_MAPS	0.968421	0.638889	0.769874	0.985816	0.731579	0.839879	0.976923	0.668421	0.793750	0.962963	0.684211	0.800000	0.954545	0.560000	0.705882
MQTT	0.924528	0.731343	0.816667	1.000000	0.989744	0.994845	0.948454	0.943590	0.946015	0.984293	0.964103	0.974093	0.917874	0.935961	0.926829
NTP	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
SSH	1.000000	0.678571	0.808511	1.000000	1.000000	1.000000	1.000000	0.984772	0.992327	1.000000	0.989848	0.994898	0.990099	0.990099	0.990099
TEAMVIEWER	1.000000	0.946903	0.972727	1.000000	0.989691	0.994819	0.994709	0.969072	0.981723	1.000000	0.979381	0.989583	0.989899	0.984925	0.987406
TOR	0.833333	0.535714	0.652174	0.985000	0.924883	0.953995	0.978261	0.845070	0.906801	0.984127	0.873239	0.925373	0.951807	0.818653	0.880223

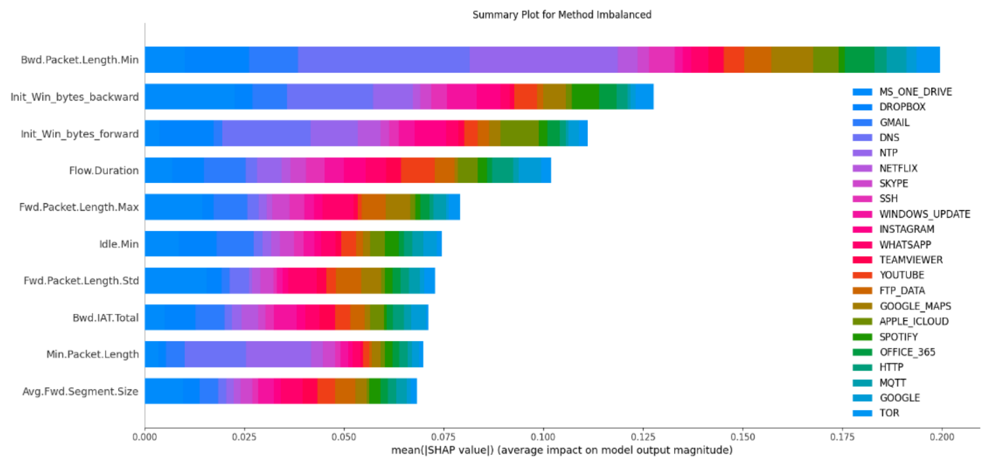
As it’s possible to see the biggest difference between the imbalanced dataframe and the other balanced dataframes is given by the *recall* value and subsequently by the *f1-score*. Recall (also known as sensitivity) is a measure of how well a model can identify positive instances correctly, it calculates the ratio of true positives to the sum of true positives and false negatives. Considering that the *precision*, which is computed as the ratio of true positives to the sum of true positives and false positives, remains relatively consistent, we can infer that oversampling greatly reduces the number of false negatives. As a result, the increase in the f1-score can be attributed to the concurrent increase in recall, as the f1-score is the harmonic mean of precision and recall.

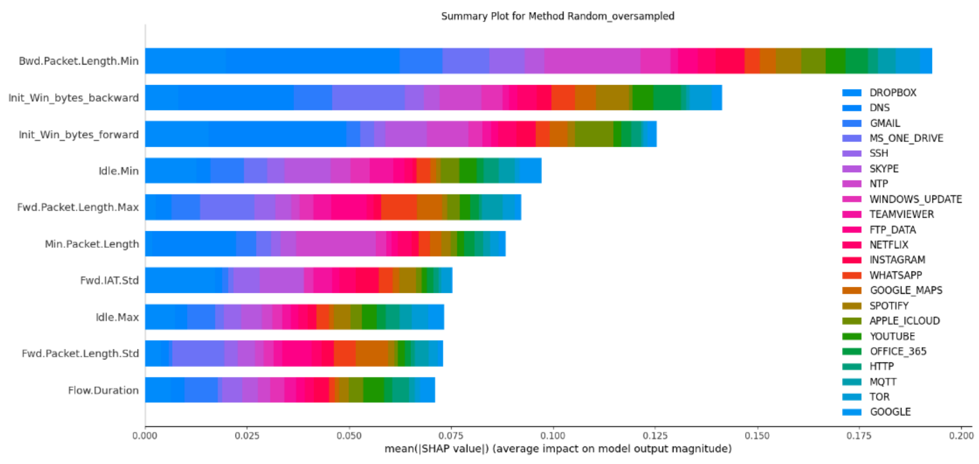
Advanced task

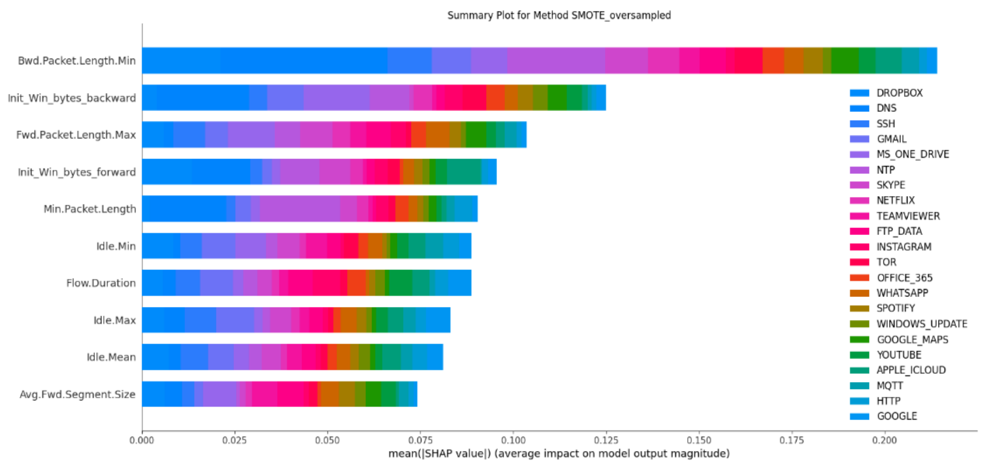
For our advanced task, we have chosen to concentrate on *eXplainable Artificial Intelligence* (XAI). The utilization of machine learning algorithms often leads to them being perceived as black boxes. However, XAI enables us to gain an understanding of how these algorithms generate specific outputs or predictions. It is a crucial area of research that strives to enhance the transparency, interpretability, and accountability of AI systems. The concept of Explainability is very important in our project, because it returns to us the “reasons” why the ML models considered (RF and KNN) arrived at those specific results. In fact, we search for how much each feature contributes to the prediction and how the values affect it. How does it work? For each feature the data inside the column are randomly shuffled, if the accuracy of the model decreases significantly means that the specific feature is important, instead, if changes a little the feature is not so important for predictions.

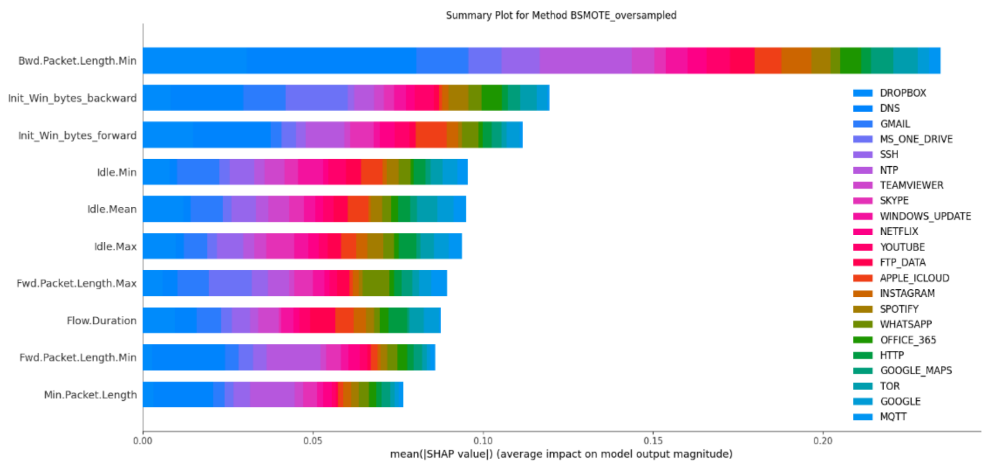
We use SHAP library to implement XAI in the project, it is useful to find the contribution of each feature in the prediction of a class. We decided to focus only on RF Algorithm because applying SHAP to KNN would have taken too much time to execute (code is implemented but not used). With the configuration used the entire dataset is considered, applying SHAP to the RF models takes about 10 hours to compute the shap values (we tried with *shappoints=5000* and execution time was 2 hour and 20 minutes).

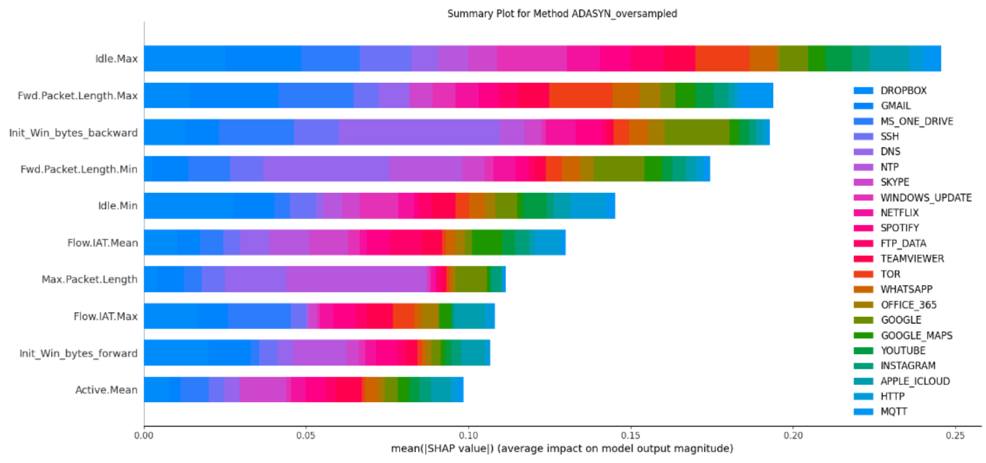
The result of the computation of the shap values for each model (Imbalanced, Random_oversampled, SMOTE_oversampled, BSMOTE_oversampled, ADASYN_oversampled) is shown in the following figures:



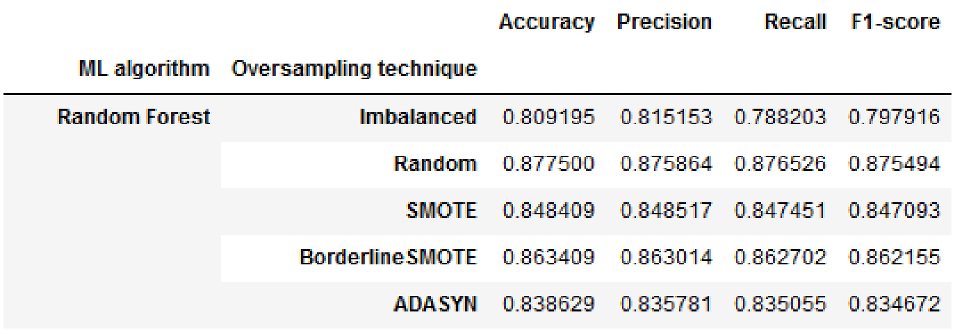

Summary Plot for Method Random_oversampled
Bwd.Packet.Length.Min
Init_Win_bytes_backward
Init_Win_bytes_forward
Idle.Min
Fwd.Packet.Length.Max
Min.Packet.Length
Fwd.IAT.Std
Idle.Max
Fwd.Packet.Length.Std
Flow.Duration
DROPBOX
DNS
GMAIL
MS_ONE_DRIVE
SSH
SKYPE
NTP
WINDOWS_UPDATE
TEAMVIEWER
FTP_DATA
NETFLIX
INSTAGRAM
WHATSAPP
GOOGLE_MAPS
SPOTIFY
APPLE_ICLOUD
YOUTUBE
OFFICE_365
HTTP
MQTT
TOR
GOOGLE
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
mean(|SHAP value|) (average impact on model output magnitude)


Summary Plot for Method SMOTE_oversampled
Bwd.Packet.Length.Min
Init_Win_bytes_backward
Fwd.Packet.Length.Max
Init_Win_bytes_forward
Min.Packet.Length
Idle.Min
Flow.Duration
Idle.Max
Idle.Mean
Avg.Fwd.Segment.Size
DROPBOX
DNS
SSH
GMAIL
MS_ONE_DRIVE
NTP
SKYPE
NETFLIX
TEAMVIEWER
FTP_DATA
INSTAGRAM
TOR
OFFICE_365
WHATSAPP
SPOTIFY
WINDOWS_UPDATE
GOOGLE_MAPS
YOUTUBE
APPLE_ICLOUD
MQTT
HTTP
GOOGLE
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
mean(|SHAP value|) (average impact on model output magnitude)

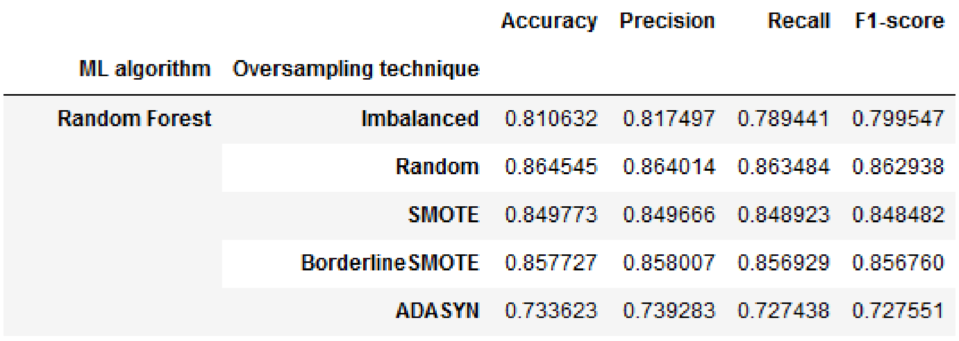

Summary Plot for Method BSMOTE_oversampled
Bwd.Packet.Length.Min
Init_Win_bytes_backward
Init_Win_bytes_forward
Idle.Min
Idle.Mean
Idle.Max
Fwd.Packet.Length.Max
Flow.Duration
Fwd.Packet.Length.Min
Min.Packet.Length
DROPBOX
DNS
GMAIL
MS_ONE_DRIVE
SSH
NTP
TEAMVIEWER
SKYPE
WINDOWS_UPDATE
NETFLIX
YOUTUBE
FTP_DATA
APPLE_ICLOUD
INSTAGRAM
SPOTIFY
WHATSAPP
OFFICE_365
HTTP
GOOGLE_MAPS
TOR
GOOGLE
MQTT
0.00
0.05
0.10
0.15
0.20
mean(|SHAP value|) (average impact on model output magnitude)


Summary Plot for Method ADASYN_oversampled
Idle.Max
Fwd.Packet.Length.Max
Init_Win_bytes_backward
Fwd.Packet.Length.Min
Idle.Min
Flow.IAT.Mean
Max.Packet.Length
Flow.IAT.Max
Init_Win_bytes_forward
Active.Mean
DROPBOX
GMAIL
MS_ONE_DRIVE
SSH
DNS
NTP
SKYPE
WINDOWS_UPDATE
NETFLIX
SPOTIFY
FTP_DATA
TEAMVIEWER
TOR
WHATSAPP
OFFICE_365
GOOGLE
GOOGLE_MAPS
YOUTUBE
INSTAGRAM
APPLE_ICLOUD
HTTP
MQTT
0.00
0.05
0.10
0.15
0.20
0.25
mean(|SHAP value|) (average impact on model output magnitude)

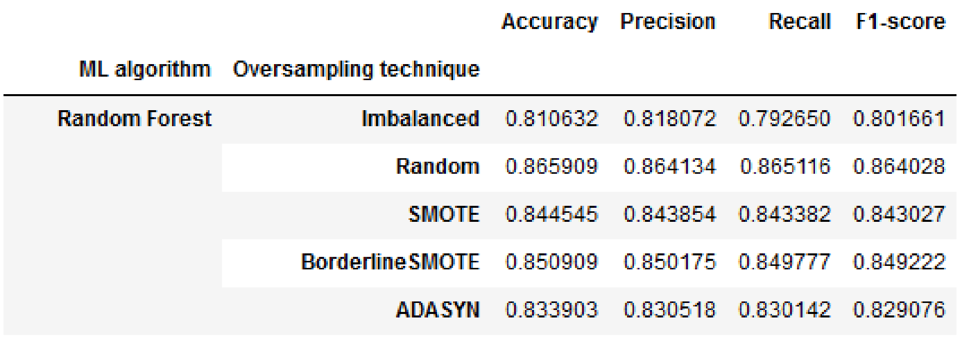
We noticed that the same features appeared in the plots (*Bwd.Packet.Length.Min, Fwd.Packet.Length.Max, Init_Win_bytes_backward, Init_Win_bytes_forward*, ecc...). So, taking, for each method, the 10 best features we wanted to study if the RF model increases performance or not. The results of the procedure are shown in this order: Imbalanced, Random Oversampling, SMOTE, BSMOTE, ADASYN.



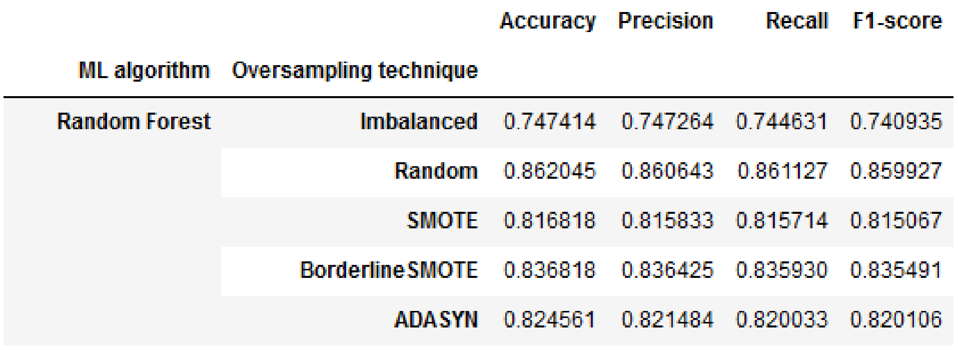
ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.809195	0.815153	0.788203	0.797916
	Random	0.877500	0.875864	0.876526	0.875494
	SMOTE	0.848409	0.848517	0.847451	0.847093
	BorderlineSMOTE	0.863409	0.863014	0.862702	0.862155
	ADASYN	0.838629	0.835781	0.835055	0.834672



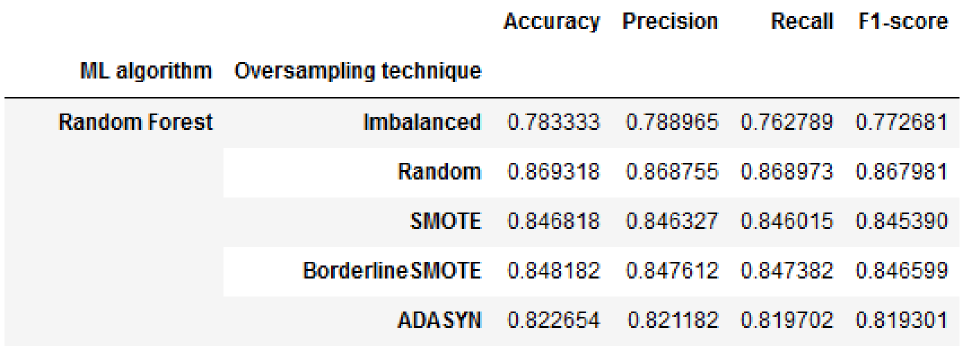
ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.810632	0.817497	0.789441	0.799547
	Random	0.864545	0.864014	0.863484	0.862938
	SMOTE	0.849773	0.849666	0.848923	0.848482
	BorderlineSMOTE	0.857727	0.858007	0.856929	0.856760
	ADASYN	0.733623	0.739283	0.727438	0.727551



ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.810632	0.818072	0.792650	0.801661
	Random	0.865909	0.864134	0.865116	0.864028
	SMOTE	0.844545	0.843854	0.843382	0.843027
	BorderlineSMOTE	0.850909	0.850175	0.849777	0.849222
	ADASYN	0.833903	0.830518	0.830142	0.829076



ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.747414	0.747264	0.744631	0.740935
	Random	0.862045	0.860643	0.861127	0.859927
	SMOTE	0.816818	0.815833	0.815714	0.815067
	BorderlineSMOTE	0.836818	0.836425	0.835930	0.835491
	ADASYN	0.824561	0.821484	0.820033	0.820106



ML algorithm	Oversampling technique	Accuracy	Precision	Recall	F1-score
Random Forest	Imbalanced	0.783333	0.788965	0.762789	0.772681
	Random	0.869318	0.868755	0.868973	0.867981
	SMOTE	0.846818	0.846327	0.846015	0.845390
	BorderlineSMOTE	0.848182	0.847612	0.847382	0.846599
	ADASYN	0.822654	0.821182	0.819702	0.819301

As we can see from the tabs, comparing the original metrics (the ones using the whole dataset provided), we saw that the metrics changed a little bit.

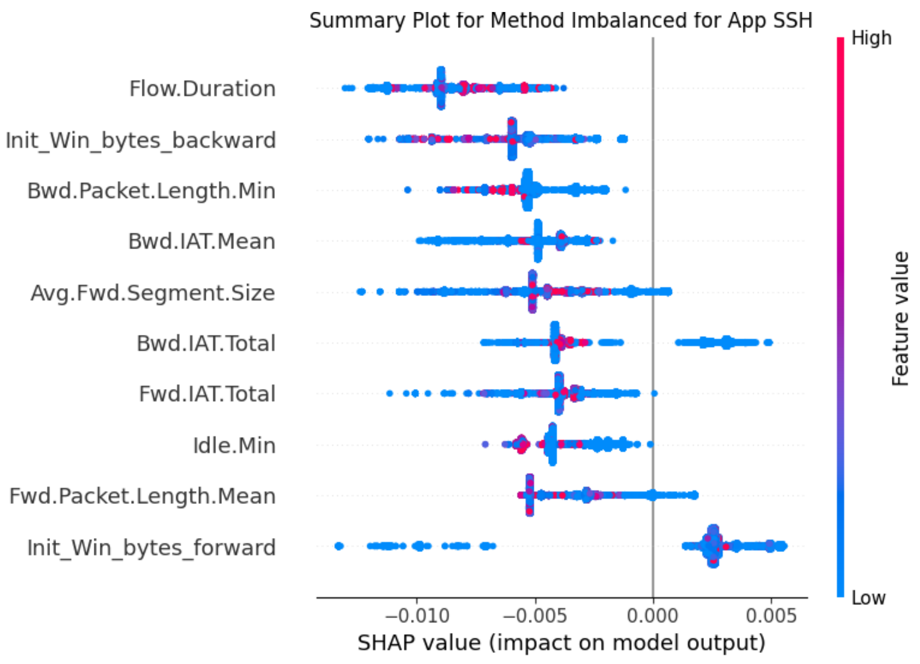
All the metrics considered (Accuracy, Precision, Recall and F1-score) see their values worsening, this happened mostly because of the important reduction of the features taken into account for the prediction, we drop from 67 features to 10. But not as much as expected. If we focus on Accuracy, we can see that it decreases a little with respect to the original values. This is a positive result since we can use only a few features to train models (reducing execution times), instead of the entire dataset, to obtain a very similar performance.

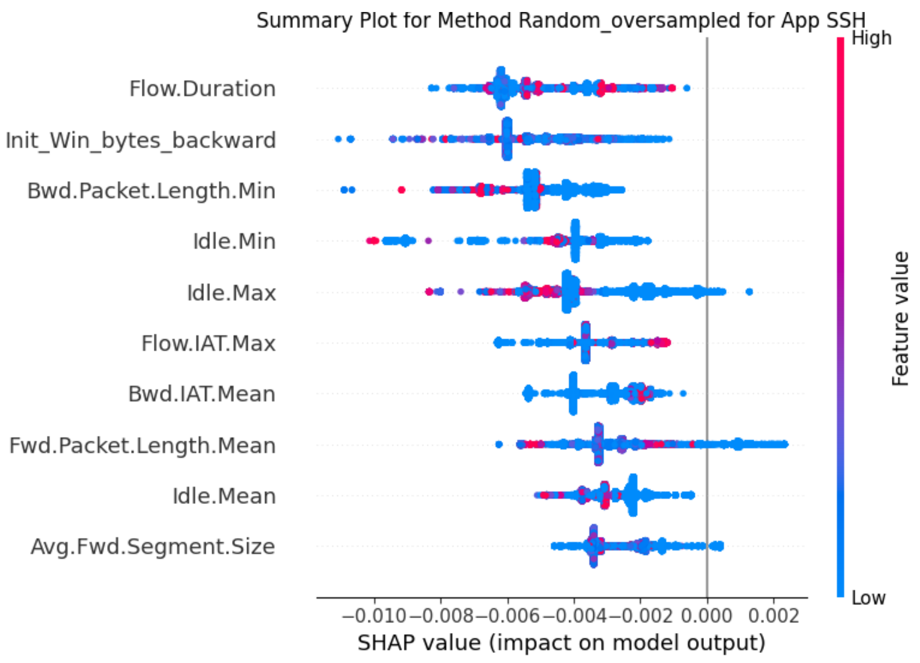
Now we want to highlight how SHAP selects which features are considered important for predicting specific classes when applied on the five different datasets (Imbalanced and the four methods of oversampling used). We chose the class ‘SSH’ for the comparison, because in the original dataset it was the least represented. The order of the figures is the same as before: Imbalanced, Random Oversampling, SMOTE, BSMOTE, ADASYN.

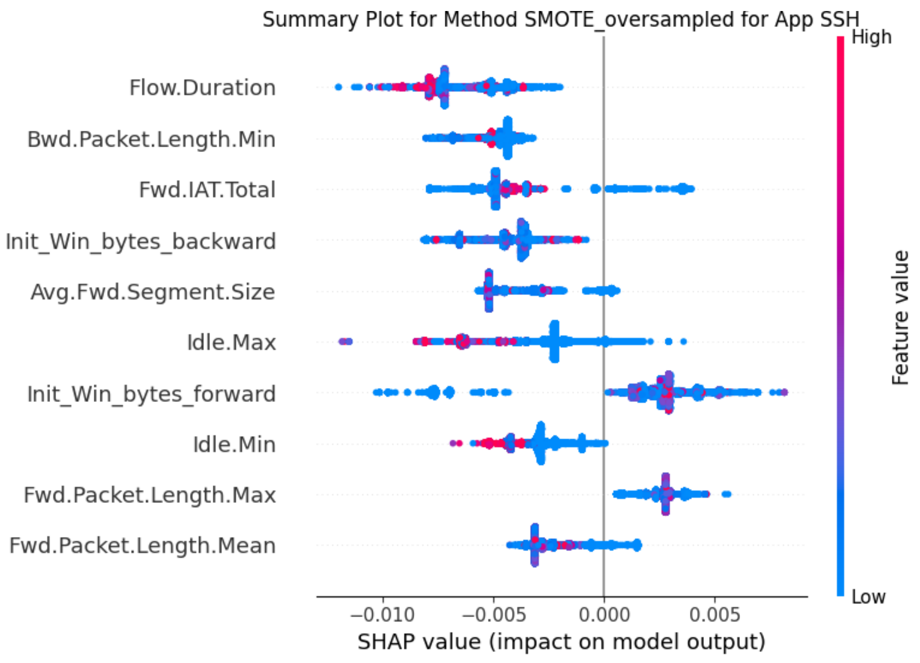
By looking at them we see that, for different methods, SHAP selected different features. This is possible because each method oversamples the dataset in different ways, so some features may get more importance with respect to other methods. We want to recall the meaning of the points in the plots: red points are the ones that push to predict to the class, while blue points push to not predict for that class.

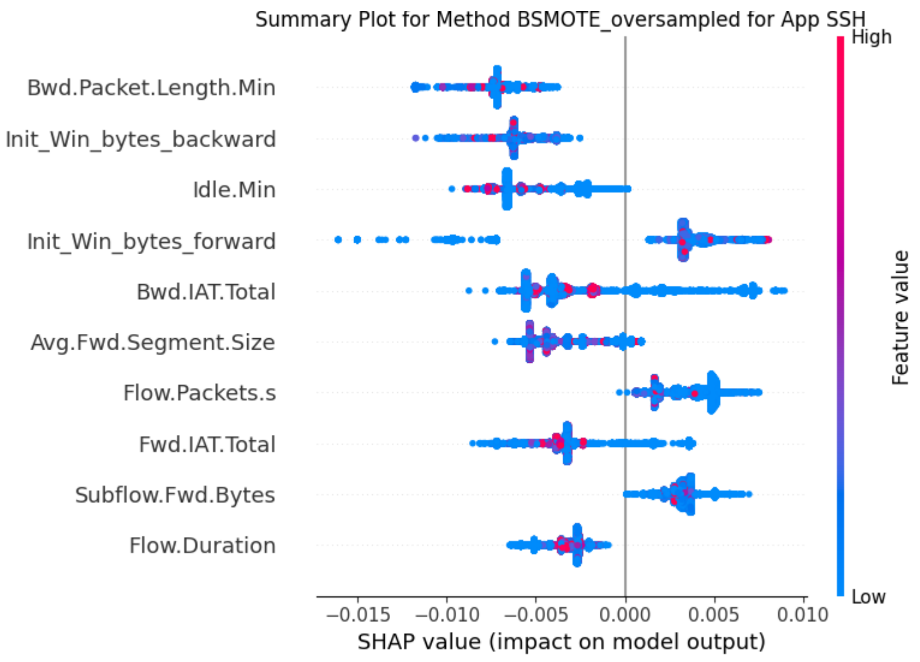
Focusing on the features selected, there are two of them that are in common with four of the methods used: *Bwd.Packet.Length.Min* and *Init_Win_bytes_backward*. These features are most of the time at the top of the lists, this means that they are very important features used in prediction. The only exception is ADASYN, which has different features in its list.

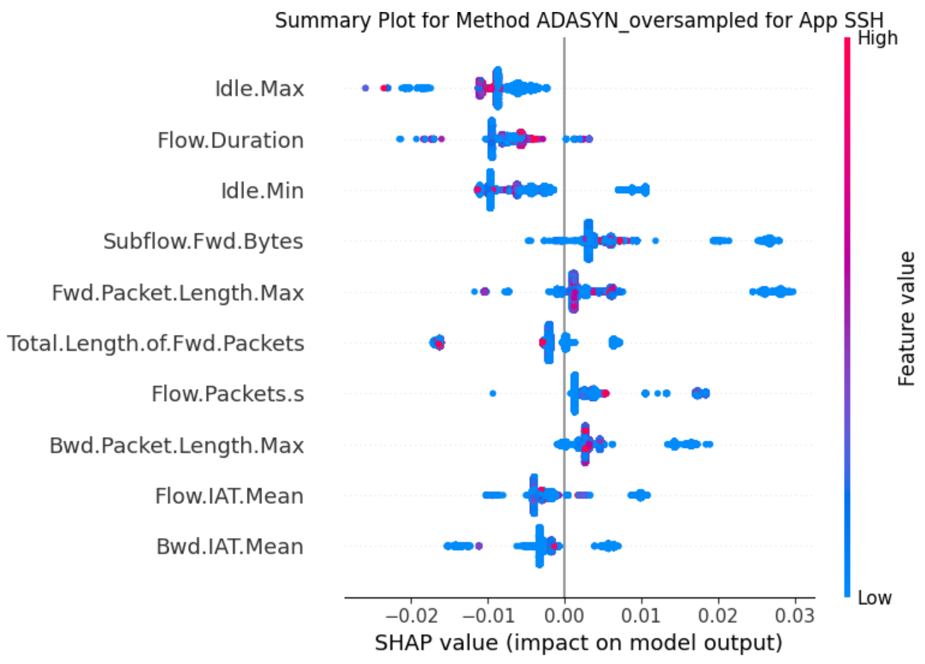
Another important feature that we can see from the plots is *Flow.Duration*. This feature is in all the plots shown below and have similar distribution of points. This means that the oversampling methods could reproduce in a good way the samples of this class in order to have a good prediction of it.


Summary Plot for Method Imbalanced for App SSH
Flow.Duration
Init_Win_bytes_backward
Bwd.Packet.Length.Min
Bwd.IAT.Mean
Avg.Fwd.Segment.Size
Bwd.IAT.Total
Fwd.IAT.Total
Idle.Min
Fwd.Packet.Length.Mean
Init_Win_bytes_forward
−0.010
−0.005
0.000
0.005
SHAP value (impact on model output)
High
Low
Feature value


Summary Plot for Method Random_oversampled for App SSH
Flow.Duration
Init_Win_bytes_backward
Bwd.Packet.Length.Min
Idle.Min
Idle.Max
Flow.IAT.Max
Bwd.IAT.Mean
Fwd.Packet.Length.Mean
Idle.Mean
Avg.Fwd.Segment.Size
−0.010
−0.008
−0.006
−0.004
−0.002
0.000
0.002
SHAP value (impact on model output)
High
Low
Feature value


Summary Plot for Method SMOTE_oversampled for App SSH
Flow.Duration
Bwd.Packet.Length.Min
Fwd.IAT.Total
Init_Win_bytes_backward
Avg.Fwd.Segment.Size
Idle.Max
Init_Win_bytes_forward
Idle.Min
Fwd.Packet.Length.Max
Fwd.Packet.Length.Mean
−0.010
−0.005
0.000
0.005
SHAP value (impact on model output)
High
Low
Feature value


Summary Plot for Method BSMOTE_oversampled for App SSH
Bwd.Packet.Length.Min
Init_Win_bytes_backward
Idle.Min
Init_Win_bytes_forward
Bwd.IAT.Total
Avg.Fwd.Segment.Size
Flow.Packets.s
Fwd.IAT.Total
Subflow.Fwd.Bytes
Flow.Duration
−0.015
−0.010
−0.005
0.000
0.005
0.010
SHAP value (impact on model output)
High
Low
Feature value



Summary Plot for Method ADASYN_oversampled for App SSH
Idle.Max
Flow.Duration
Idle.Min
Subflow.Fwd.Bytes
Fwd.Packet.Length.Max
Total.Length.of.Fwd.Packets
Flow.Packets.s
Bwd.Packet.Length.Max
Flow.IAT.Mean
Bwd.IAT.Mean
−0.02
−0.01
0.00
0.01
0.02
0.03
SHAP value (impact on model output)
High
Low
Feature value