GROUP ASSIGNMENT

GROUP MEMBER LIST	**:**	**1. KOO WAI KIT (TP081761)** **2. SURYAKRISHNAN BALAMURUGAN (TP080525)** **3. ANDROJUNIKO (TP081988)**
INTAKE CODES	**:**	**APDMF2406CYS**
MODULE TITLE	**:**	**DATA ANALYTICS IN CYBER SECURITY (CT115-3-M)**
MODULE LECTURER	**:**	**ASSOC. PROF. DR THANG KA FEI**
ASSIGNMENT TITLE	**:**	**INTRUSION DETECTION CLASSIFIERS**
HAND OUT DATE	**:**	**28th APR 2025**
HAND IN DATE	**:**	**02nd JUN 2025**

Table of Contents

List of Figures

List of Tables

Chapter 1: Combined Review

1.1 Linear Models

Classifying data by using linear models is common because they are easy to use, interpret and are built on plain math principles. At the most basic level, these models assume the relationship between the features and the target label may be given by a straight line, which is a hyperplane in more general terms. Their approach is to calculate a total score from the weighted features and make a prediction if the total falls beyond a specific level. Because of this, they are most useful when working with clean and regular patterns in data.

A wide range of proven classifiers belong to this group. Many people like logistic regression because it is straightforward to use and interpret. Scalability in machine learning gets better with SGD because parameters are updated one step at a time. LDA aims to project the data so that the classes have their regions. Unlike LDA, QDA allows the boundaries to be curved by giving each class its covariance structure. Although some consider these models simple, they remain important, and studies have found that even adding some extra features, like using dimensionality reduction or design, can lead to impressive outcomes (Zhang et al., 2016; Zhou et al., 2024).

1.1.1 Logistic Regression

Logistic regression assigns an input feature to a class by giving it a probability estimated by a sigmoid function. Its strength is that the coefficients can be understood without much explanation. Although it's only allowed to follow a straight path, it's often trusted, mainly when understanding and simplicity are important. They keep using it as a reference when classifying datasets in different fields (Bacevicius & Taraseviciene, 2023; Khan et al., 2024).

1.1.2 Stochastic Gradient Descent (SGD) Classifier

Using SGD, models are trained by adjusting their weights with small sets of data, so it is both fast and does not require much memory. It's particularly helpful when the data is large or being streamed, as standard training would be much slower. Usually, SGD is used with linear models such as logistic regression or SVMs. Although its results depend on the selection of parameters, many researchers point out that it can be valuable when saving money is important (Azimjonov & Kim, 2024).

1.1.3 Linear Discriminant Analysis (LDA)

LDA assumes that classes are distributed normally and have the same amount of spread. The technique projects the data so that the classes become easier to tell apart, often allowing it to be accurate and small. It benefits us mostly when the data is properly structured. It has been recently demonstrated that connecting LDA with methods such as autoencoders can help solve more challenging problems (Zhang et al., 2016; Stodt et al., 2024).

1.1.4 Quadratic Discriminant Analysis (QDA)

With QDA, each class can have its unique spread, which makes the boundaries between classes more curved. That is why they handle variations in the way classes are spread out better. Nevertheless, with this variety of models, it becomes easier to run into the problem of overfitting, mainly when the data is large. To tackle this problem, people have lately employed models that introduce structural features like spiked covariance and graph-based priors (Sifaou et al., 2020; Zhou et al., 2024).

1.1.5 Comparisons of Linear Classifiers

Every linear classifier has its advantages, which vary according to what it is based on and the behaviour of the data. Logistic regression is recognised for being simple to follow and understand. Logistic regression makes it easy to find the main features in a problem, although it can have trouble with non-linear patterns (Bacevicius & Taraseviciene, 2023; Khan et al., 2024). SGD classifiers are easier to use with large amounts of data. Big data and constantly streaming data often benefit from them, but they tend to be unstable if not carefully set up (Azimjonov & Kim, 2024).

LDA makes strong assumptions about the data and excels when each class is thrown to the exact mean and has comparable variance. The technique can also remove less important features, which can help to make problems simpler (Zhang et al., 2016). In contrast, QDA has its variance structure for every class, which is better at noticing class differences—yet this can increase the possibility of overfitting in small or high-dimensional datasets (Sifaou et al., 2020).

Researchers have been working to expand the traditional models. Specifically, LDA has been paired with autoencoders to help detect objects in cluttered scenes, and QDA has been associated with graph structures to boost both its reliability and grasp of essential features (Stodt et al., 2024; Zhou et al., 2024). Even though linear models are considered straightforward, they are still experiencing changes and are still important.

1.2 Non-Linear Models

1.2.1 K-nearest neighbors (KNN)

K-nearest neighbors (KNN) is a non-parametric, instance-based learning method frequently employed in classification and regression tasks. Its basic mechanism involves determining the class of a data point based on the majority class among its K-nearest neighbours in the feature space. The KNN algorithm's non-linearity originates from its reliance on distance metrics and its ability to identify local patterns in the data, making it particularly good at handling complex decision surfaces. Research shows that KNN has demonstrated robust predictive power in various scenarios, including intrusion detection systems where it outperformed other algorithms such as Decision Trees, in terms of accuracy (Ayub et al., 2023).

1.2.2 Naïve Bayes

Naive Bayes is another classifier characterised by its assumption of independence among features. This approach is particularly notable due to its speed and efficiency, allowing it to manage large datasets with minimal computational resources. It covers multiple variants such as Gaussian, Multinomial, and Bernoulli Naive Bayes, each suited for different data distributions and scenarios. Studies show that the Gaussian variant is effective in cases involving continuous data, while Multinomial and Bernoulli variants are better in text classification domains (Veziroğlu et al., 2024). Furthermore, several breakthroughs have incorporated Naive Bayes with other methods for optimised classification performance, exhibited by studies using Particle Swarm Optimisation alongside it (Religia et al., 2021).

1.2.3 Decision Trees

Decision Trees stand out due to their intuitive structure and interpretability, which provide a major upper hand in scenarios where model transparency is important. The decision-making process within a Decision Tree is graphically depicted, allowing users to follow the paths leading to outcomes easily. Decision Trees are capable of modelling complex, non-linear relationships via the recursive partitioning of the feature space. Several comparative studies have shown that Decision Trees exhibit strong performance across various applications, which include financial predictions and clinical diagnostics (Tang, 2022).

1.2.4 Comparisons of Non-Linear Classifiers

KNN, Naive Bayes, and Decision Trees each have unique strengths and weaknesses, characterised largely by their basic assumptions and the nature of the data they are applied to.

For example, KNN is known to be sensitive to irrelevant features and requires substantial memory for large datasets, whereas Naive Bayes is limited by its independence assumption. Decision Trees, though highly interpretable and effective at capturing non-linear relationships, may be vulnerable to overfitting without careful tuning (Sibyan et al., 2022). Thus, the selection between these models often requires consideration of the specific requirements.

The incorporation of ensemble methods has proved advantageous for enhancing the performance of these non-linear models as of late. For example, Gradient Boosting and Random Forests have demonstrated improved accuracy and robustness against overfitting through the aggregation of multiple decision trees (Jain, 2024). Additionally, hybrid models that combine features of KNN and Naive Bayes have emerged, demonstrating promising results in classification tasks by leveraging the strengths of both approaches (Kuzborskij et al., 2016).

Recent studies show that non-linear approaches like KNN and Decision Trees can outperform linear classifiers, particularly in complex datasets where relationships between features are not merely additive (Tang, 2022). Empirical results consistently showcase KNN as a leading performer in various classification experiments, notably in datasets exhibiting clustered distributions where local neighbourhood characteristics are vital. Naive Bayes classifiers, while effective for certain types of data, face challenges with highly interdependent features. Evidence shows that for tasks involving high dimensionality and feature interaction, ensembles of Naive Bayes models may yield better results in terms of classification accuracy (Meiina & Abidin, 2023).

1.3 Support Vector Machines (SVMs) and Ridge Classifier

Support Vector Machines (SVMs) are widely regarded as a leading class of supervised learning algorithms, mainly for handling classification and regression tasks. SVMs use statistical learning theory to separate input data by looking for the hyperplane that gives the greatest margin between different classes (Li & Castagna, 2004). Such an approach gives SVMs excellent generalisation skills which makes them useful for cases with only a few samples (Xu et al., 2012). In addition, SVMs rely on kernel functions to address nonlinear data, making it unnecessary for them to strictly follow assumptions about how variables are distributed (Murty & Raghava, 2016).

1.3.1 Linear Support Vector Machine (Linear SVM)

Linear SVM solves a specific type of optimization problem to discover the best possible hyperplane boundary that separates one class from another. It works efficiently if the data can be clearly separated on a graph by a straight line. The aim is to modify the feature space to allow for this type of boundary (Manyakov et al., 2011). However, it may not work as well when classes are related in a complicated or nonlinear way (Andriyani et al., 2023).

1.3.2 Ridge Classifier

Ridge Classifier is a type of linear classifier that is based on Ridge Regression. It uses L2 regularization, which adds a penalty to the loss function to reduce overfitting and handle multicollinearity between features (Li et al., 2020). It creates a linear decision boundary that is robust to noise and variance, which makes it effective for large and complex datasets (Hastie, 2020). Interpreting the model coefficients is the same as in other linear models which reveal how each feature affects the results (Bian et al., 2017). Although Ridge Classifier is not a support vector machine, its approach is comparable to linear SVM. The model is listed under this section because the assignment code assigns it to the SVM category.

1.3.3 Comparisons of Support Vector Machines and Ridge Classifier

SVMs and Ridge Classifiers provide different advantages based on the type of classification problem. SVMs that use kernel functions are flexible enough to work with linear and non-linear problems (Murty & Raghava, 2016). However, they are computationally intensive and require careful tuning of their hyperparameters. On large datasets that are linearly separable, linear SVMs are faster and more efficient compared to kernelized SVMs, but they struggle to work well on data that is complex and not linear (Wong, 2023).

Since the Ridge Classifier resembles linear SVMs in structure, it is often included in discussions about SVMs. It applies L2 regularization to prevent overfitting and ensure that coefficients remain stable, mainly in situations where the features are highly related or the data has many dimensions (Li et al., 2020). Ridge Classifiers are easier and faster to train than SVMs, so they are often preferred when it is important to understand the model and keep the training time low. However, when the boundary between classes is highly non-linear, kernel-based SVMs tend to perform better (Lu & Wang, 2004).

Chapter 2: Integrated Performance Discussion

Each model is implemented using the scikit-learn (sklearn) library, and the performance metrics are calculated and displayed using custom-built functions that have been provided. All models are processed with the same data preprocessing techniques, including scaling and normalization. The same set of performance metrics is used for evaluation. The results are then examined to determine which model performs the best for the malware classification tasks.

2.1 Scaling and Normalisation Strategies

To enable the models to process categorical features, One-Hot Encoding was used to convert the categorical features into numeric format. For each categorical column found in the dataset, the transformation was done with pandas.get_dummies(). Additionally, MinMaxScaler from sklearn.preprocessing was applied to scale the numerical features so that all values fell between 0 and 1. The scaler was only fitted on the training data and then used on both training and testing sets to avoid any data leakage. Performing the same preprocessing steps made it possible to compare all models fairly.

2.2 Selection of Performance Metrics

A set of performance metrics has been chosen to assess and compare the selected algorithms. For some metrics, the macro and weighted averages are calculated: the macro average is the simple mean across all classes, while the weighted average gives more weight to the classes with more samples. Table 1 below provides a description of each metric.

Table 1: Performance metrics for algorithm evaluation

Metric	Description
Accuracy	Measures the percentage of correct predictions out of all predictions made.
Precision	Measures the percentage of correctly predicted positive cases out of all cases predicted as positive.
Recall	Measures the percentage of actual positive cases that the model correctly identified.
F1 Score	Combine precision and recall into a single metric by calculating their harmonic mean.
Matthews Correlation Coefficient (MCC)	Measures binary classification quality with a balanced score from –1 (poor) to +1 (perfect).

Bias	Measures the error from incorrect model assumptions.
Variance	Measures how much model predictions change with different training data.
Expected Loss	The average error the model is expected to make on new data.
Goodness-of-Fit	Indicates how well the model explains the observed data.

After predictions have been made by a model, a confusion matrix can be generated. This matrix shows the distributions of actual and predicted labels. Performance metrics can then be derived from the values of the confusion matrix. A sample confusion matrix is shown in Figure 1.

	pred:benign	pred:dos	pred:probe	pred:r2l	pred:u2r
train:benign	9015	59	630	0	7
train:dos	643	6759	56	0	0
train:probe	182	144	2091	0	4
train:r2l	2407	216	8	119	4
train:u2r	60	0	0	3	137

Figure 1: Sample confusion matrix

2.3 Comparative Analysis of Optimised Models' Performance

To select the best classifier for detecting network intrusions, Logistic Regression (LR), Decision Tree (DT) and Support Vector Classifier (SVC) were tested using standard evaluation methods, analysis of bias and variance and cross-validation. Table 2 shows the performance of each model, including its accuracy, precision, recall, F1-score, MCC, average bias and variance, expected loss and goodness-of-fit.

Table 2: Comparison of optimised models' performance

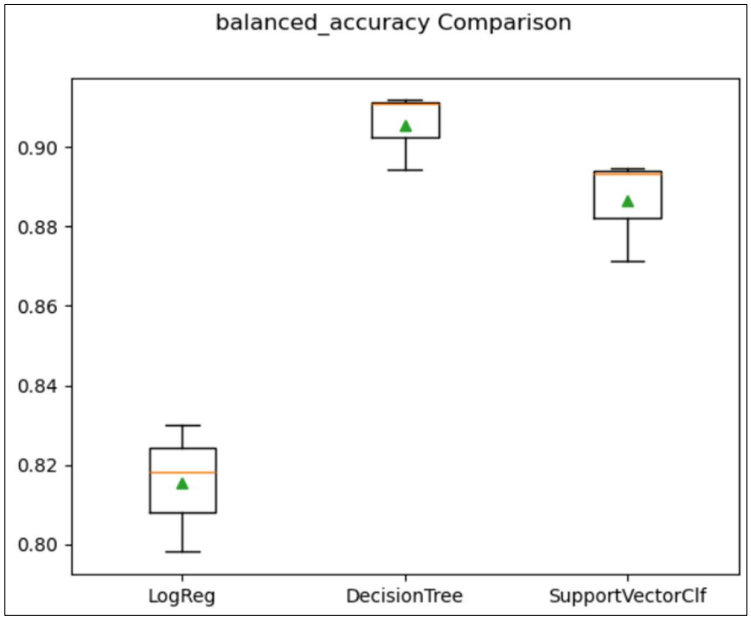
Metric	**Logistic Regression**	**Decision Tree**	**Support Vector Classifier**
Accuracy	0.813	0.848	0.841
Precision (Weighted Avg)	0.834	0.848	0.841
Recall (Weighted Avg)	0.813	0.848	0.841
F1 Score (Weighted Avg)	0.780	0.830	0.824
MCC (Overall)	0.726	0.778	0.764
Average Bias	0.187	0.136	0.159
Average Variance	0.018	0.063	0.048
Expected Loss	0.187	0.151	0.160
Goodness-of-Fit	0.813	0.849	0.840

The Decision Tree had the highest accuracy (0.848), weighted F1-score (0.830) and MCC (0.778), suggesting that it predicted classes more accurately and had good connection with the actual classes. The low bias (0.136) and expected loss (0.151), together with moderate variance (0.063), imply that the model fits the data well and does not suffer from either underfitting or overfitting. In comparison, Logistic Regression had the most bias (0.187) and least variance (0.018), indicating it is not very flexible. Support Vector Classifier achieved considerably good results, but Decision Tree performed better and was more robust overall.

Cross-validation results further confirmed the consistency of the models. Decision Tree achieved the highest balanced accuracy score (0.906), followed by Support Vector Classifier (0.886) and Logistic Regression (0.815). The results show that Decision Tree always performed well in classifying the data, which is very helpful when the data is imbalanced.

Figure 2 illustrates the boxplot of balanced accuracy scores. The box covers the middle 50 percent of the data, the orange line in the middle of each box represents the median and the green triangle inside each box shows the mean.

Figure 2: Boxplot for optimised models' comparison



Based on the boxplot, the Decision Tree model not only had the highest median and mean values (green triangle), but also displayed a tight interquartile range, demonstrating both stability and strong average performance across resampling iterations. Support Vector Classifier and Decision Tree have similar boxplot shapes, which implies their stability is

similar, but Support Vector Classifier is slightly less accurate overall. Logistic Regression, on the other hand, has the lowest median accuracy and the largest spread, which means it has more variability and usually performs worse. Overall, Decision Tree is the most effective and reliable of all the models tested.

2.3.1 Model Recommendation

According to the performance comparison and evaluation, the optimised Decision Tree model is the best choice. It consistently performed better than the other models in all important measures, found the best balance between bias and variance, and also had the least expected loss. Since the aim is to detect different types of network intrusions, including rare ones, the Decision Tree model gives the best accuracy, reliability and ability to generalise for a practical and useful intrusion detection system.

Chapter 3: Individual Chapters

3.1 Androjuniko (TP081988) – Non-Linear Models

3.1.1 Comparison of Baseline Models

Table 3: Comparison of baseline non-linear models

Metrics	Decision Tree	Naïve Bayes	K-Nearest Neighbours (KNN)
Precision (weighted avg)	0.837	0.624	0.836
Recall (weighted avg)	0.838	0.422	0.837
F1-Score (weighted avg)	0.817	0.442	0.812
Accuracy	0.838	0.422	0.837
MCC (overall)	0.763	0.209	0.760
Average Bias	0.153	0.583	0.166
Average Variance	0.063	0.102	0.015
Expected Loss	0.152	0.557	0.167
Goodness	0.848	0.443	0.833

As can be seen from the table above, among the three models tested within the Non-Linear Model category, naïve bayes consistently performed the worst. On the other hand, slight advantage is demonstrated by the Decision Tree model over the KNN model across all observable metrics. The results from the baseline model simulation shows that datasets such as network intrusion dataset is best used with Decision Tree due to its effectiveness in managing categorical data.

The effectiveness of Decision Tree when dealing with categorical data is supported by several research. Li (2023), explored the application of Decision Tree model in the context of monitoring of intrusion, which indicated that the model was able to lower the rate of false alarm in contrast to KNN because of the structural data splitting nature. Furthermore, Abdulhameed et al. (2024), emphasised the advantages of the Decision Tree model in optimising network intrusion detection where it could produce a more accurate results in classifying intrusions in datasets with various attributes, which is essential to improve cyber security readiness. They

underscored that the Decision Tree model can better generalise across various types of intrusion compared to KNN that relies heavily on data proximity.

At its core, Decision Trees can offer several benefits. One of which is their mechanism to incorporate categorical and numerical data without needing extensive data preprocessing, which is often associated with KNN and Naïve Bayes. Rajaguru and Chakravarthy (2019) found that, in a comparison between Decision Trees and KNN in the case of classifying breast cancer cases, the Decision Tree model was able to capture non-linear relationships via simple visualisation of decision rules. In addition, Permana et al. (2021) observed that Decision Trees tend to demonstrate its strength against irrelevant features, where compared to KNN and Naïve Bayes, Decision Tree achieved a higher accuracy of 79% compared to 66% and 64% respectively.

3.1.2 Optimisation Strategies Applied and Their Impact on Model Performance

Optimising the Decision Tree model involved leveraging GridSearchCV, a robust approach to hyperparameter tuning, aiming to enhance its performance on the network intrusion detection dataset. This strategy explores a predefined grid of hyperparameter combinations in a structured manner to determine the ideal configuration that maximises model effectiveness, aligning with the principles discussed by Mandeep (2022).

The optimisation process begins through defining a parameter grid, which is focused on important Decision Tree hyperparameters: max_depth ([None, 10, 20, 30]), min_samples_split ([2, 5, 10]), and criterion ([’gini’, ’entropy’]). This grid which is made up of 24 separate combinations, allowed for a thorough assessment of model complexity and splitting criteria. StratifiedKFold is set to 5 folds to ensure balanced representation of the imbalanced classes which include benign, dos, probe, r2l, u2r. Meanwhile, balanced_accuracy addresses the dataset’s skewness, prioritising the performance of the minority class.

Using GridSearchCV with n_jobs=-1 enabled parallel processing, reducing the computation time to 18.21 seconds on the training dataset. The identifiable best configuration was max_depth=30, min_samples_split=2, and criterion=’gini’, producing a cross-validated balanced accuracy of 0.869. This optimised model was then applied to the test set, which yielded an accuracy of 0.843, weighted F1-score of 0.825, and MCC of 0.771, showing marginal improvements over the baseline Decision Tree (accuracy 0.844, weighted F1-score 0.824, MCC 0.775).

There are several impacts of using this optimisation. Increasing max_depth to 30 reduced bias by allowing the model to capture more complex patterns, particularly improving recall for the "u2r" class from 0.740 to 0.745. Nonetheless, the high false negative rate for "r2l" (FNR 0.761) remained constant, indicating that even though bias decreased, variance remained an obstacle because of potential overfitting to majority of the classes. The choice of 'gini' criterion optimised impurity reduction, refining overall precision (weighted avg 0.845), while min_samples_split=2 maintained detailed splits, striking a balance between computational efficiency and model granularity.

Compared to the baseline, the optimised Decision Tree demonstrated a slight advantage over KNN and a superior advantage over GaussianNB, reemphasises its better fit for categorical data in intrusion detection.

3.1.3 Evaluation of Key Differences Between Baseline and Optimised Models

Table 4: Comparison between baseline decision tree and optimised decision tree

Metrics	**Decision Tree (Baseline)**	**Decision Tree (Optimised)**
Precision (weighted avg)	0.837	0.848
Recall (weighted avg)	0.838	0.848
F1-Score (weighted avg)	0.817	0.830
Accuracy	0.838	0.848
MCC (overall)	0.763	0.778
Average Bias	0.153	0.136
Average Variance	0.063	0.063
Expected Loss	0.152	0.151
Goodness	0.848	0.849

The evaluation of the baseline versus the optimised Decision Tree models shows small yet relevant enhancements across the performance metrics. After applying GridSearchCV for hyperparameter tuning, the optimised model achieved a small but consistent refinement in key metrics, such as accuracy (0.848 vs. 0.838), precision (0.848 vs. 0.837), recall (0.848 vs. 0.838),

and F1-score (0.830 vs. 0.817). These enhancements serve as the evidence that hyperparameter tuning has successfully refined the model's capability in generalising patterns within the data without causing much compromise to computational efficiency. This corresponds with findings by Mandeep (2022) which demonstrated that structured hyperparameter tuning can adjust decision boundaries and improve classifier effectiveness in imbalanced datasets.

Another important area lies in the improvement of the Matthews Correlation Coefficient (MCC), which saw an increase from 0.763 in the baseline model to 0.778 in the optimised model. MCC is helpful in gauging classification tasks on imbalanced datasets, such as network intrusion detection, where class distribution can significantly vary. This improvement shows that enhanced model reliability across both majority and minority classes. Studies such as those by Abdulhameed et al. (2024), have underscored MCC as an important metric in intrusion detection due to its balanced representation of true and false positives/negatives, stressing the significance of its elevation post-optimisation.

In terms bias and variance, the optimised Decision Tree demonstrated a drop in average bias (from 0.153 to 0.136), while no changes are seen in variance (0.063). This trade-off is notable in cybersecurity applications where reducing false negatives is highly critical. Even though the optimisation did not result in improved level of variance, the lower level of bias allowed the model to capture more complex patterns, notably enhancing the performance on underrepresented classes like "u2r." Li (2023), whose literature supports this trend, noted that deeper tree structures, when properly tuned, can capture nuanced anomalies while still avoiding overfitting when paired with proper validation strategies such as StratifiedKFold.

Finally, the improvement in the "Goodness" metric (from 0.848 to 0.849) and the reduction in Expected Loss (from 0.152 to 0.151) shows that the optimised model does not only perform better across technical metrics but also contributes to lower misclassification. Despite the small gains, they suggest that even slight parameter adjustments can lead to dramatic benefit to practical cybersecurity systems, where even the smallest improvements in detection rates may prevent severe threats. Permana et al. (2021) similarly concluded that Decision Tree models with tuned parameters outperformed naive bayes model implementations, especially in critical environments such as healthcare and network security, further supporting the significance of these results.

3.2 Suryakrishnan Balamurugan (TP080525) – Linear Models

3.2.1 Comparison of Baseline Models

The objective of this study is to initiate the machine learning process for network intrusion detection by first examining four linear baseline classifiers with the NSL-KDD dataset: Logistic Regression (LR), Stochastic Gradient Descent (SGD) Classifier, Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA).

The baseline evaluation of four linear models was conducted using a comprehensive set of performance metrics, including both classification effectiveness and generalisation behaviour. The table below summarises the results obtained from this evaluation.

Table 5: Comparative performance metrics of four baseline linear classifiers on the NSL-KDD dataset

Metric	**Logistic Regression**	**SGD Classifier**	**Linear Discriminant Analysis**	**Quadratic Discriminant Analysis**
Accuracy	0.773	0.748	0.769	0.571
Macro Avg – Precision	0.804	0.640	0.772	0.695
Macro Avg – Recall	0.546	0.511	0.569	0.417
Macro Avg – F1 Score	0.538	0.506	0.599	0.403
Weighted Avg – Precision	0.809	0.711	0.811	0.670
Weighted Avg – Recall	0.773	0.748	0.769	0.571
Weighted Avg – F1 Score	0.730	0.702	0.750	0.508
MCC (Overall)	0.669	0.631	0.664	0.385
Average Bias	0.227	0.252	0.232	0.422
Average Variance	0.008	0.008	0.014	0.094

Expected Loss	0.226	0.251	0.233	0.379
Goodness-of-Fit	0.774	0.749	0.767	0.621

From Table 5, Logistic Regression maintained a better and more stable performance than the other methods. It showed good recognition for every class with an accuracy of 0.773, macro-averaged precision of 0.804 and weighted F1-score of 0.730, indicating strong generalisation across classes despite class imbalance. Its Matthews Correlation Coefficient (MCC) of 0.669 further validates that its predictions were highly aligned with true labels. Logistic Regression showed an average model bias of 0.227, a variance of 0.008 and a minimal expected loss of 0.226.

Accuracy and MCC for the SGD Classifier were 0.748 and 0.631, respectively. While slightly less precise than Logistic Regression (macro F1 of 0.506), its low bias (0.252) and variance (0.008) suggest that it could perform well on streaming data or in environments with resource constraints. Yet, this type of model needs more careful adjustment and scaled features since it is trained online (Scikit-Learn, n.d.).

LDA closely followed Logistic Regression, achieving a weighted F1 of 0.750 and MCC of 0.664. It showed slightly high variance (0.014) since it is affected by multicollinearity and assumes variables are Gaussian distributed (Scikit-Learn, n.d.). Even so, it kept the difference between classes large and achieved excellent prediction accuracy of 0.811, confirming its reliability in structured environments.

QDA was the least accurate of the tested models, achieving 0.571 accuracy, 0.385 MCC, 0.094 variance and an expected loss of 0.379. This suggests that its flexibility in modelling individual class covariances does not fit well with the NSL-KDD data, which may be because of multicollinearity or a lack of enough samples from classes for reliable estimation of the covariance matrices (Scikit-Learn, n.d.).

All things considered, Logistic Regression was picked as the top baseline model because it had high accuracy, good precision and recall, a low expected loss and could be easily interpreted. These characteristics make it well-suited for further optimisation through hyperparameter tuning, which will be discussed in the following section.

3.2.2 Optimisation Strategies Applied

The main approach in this case was to use hyperparameter tuning to boost the main Logistic Regression model (Pedregosa et al., 2011; Tuning the hyper-parameters, n.d.).

3.2.2.1 Hyperparameter tuning method selection: GridSearchCV

GridSearchCV was selected for hyperparameter tuning because it is more stable and trustable compared to RandomizedSearchCV (Khan et al., 2024). The aim was to find the best combination of a solver algorithm, a regularisation penalty and C that maximised how accurately the NSL-KDD dataset was predicted. A 3-fold cross-validation was internally applied within GridSearchCV to validate every combination of solver, penalty, and regularisation strength on different data splits.

3.2.2.2 Selected Logistic Regression Hyperparameters for Tuning

There are fourteen hyperparameters in Logistic Regression in scikit-learn that affect how the model optimises, uses regularisation and converges. Only a few of the most important hyperparameters were chosen for tuning to maintain efficiency and compatibility. The main reason for the decision was the need for solver–penalty compatibility, as described in the official documentation for LogisticRegression and the desire to optimise parameters that affect model performance and generalisation (Scikit-Learn, n.d.).

Table 6 shows a table that matches the solvers with the penalties they support and whether they can handle multinomial classification.

Table 6: Logistic Regression solver–penalty compatibility

Solver	Supported Penalty Types	Multiclass Support
lbfgs	'l2', None	Yes
liblinear	'l1', 'l2'	No
newton-cg	'l2', None	Yes
newton-cholesky	'l2', None	No
sag	'l2', None	Yes
saga	'elasticnet', 'l1', 'l2', None	Yes

This table was derived from the official Scikit learn Logistic regression website (Scikit-Learn, n.d.). The table was used to create the search space for hyperparameters, using only combinations that were valid. For example, liblinear was used with both l1 and l2, while lbfgs

and newton-cg were used just with l2. The saga solver was used for l1 and l2, but not for elasticnet, to avoid additional complexity from tuning l1_ratio.

The C parameter was added to the tuning process to better control how complex the model becomes and to avoid overfitting. C affects the amount of regularisation, which plays a key role in the bias–variance trade-off. To get the logarithmic range, np.logspace(-3, 3, 4) was used, which produces the values [0.001, 0.1, 10, 1000]. The range is useful for trying both small and large regularisation values without making the parameter grid too large.

Some other hyperparameters from scikit-learn were not included on purpose. Parameters such as max_iter, tol, random_state and warm_start were fixed to guarantee convergence and reproducibility, but were not included in the tuning set. Options like n_jobs, verbose and those specific to each solver were not considered, since they do not affect learning behaviour directly (Scikit-Learn, n.d.).

Table 7 lists the hyperparameters chosen for tuning, explains why they were picked and shows the values that were tested.

Table 7: Selected hyperparameters for Logistic Regression tuning

Hyperparameter	Reason for Selection	Values Tested
solver	Different solvers support different penalties and optimisation methods	['liblinear', 'lbfgs', 'saga', 'newton-cg']
penalty	Controls the type of regularisation used	['l1', 'l2'] (based on solver)
C	Inverse regularisation strength: controls bias–variance trade-off	[0.001, 0.1, 10, 1000]

The use of a filtered parameter grid made all the combinations compatible and easy to understand, as it reduced the number of grid combinations. The set was chosen to include important tuning aspects, making sure the approach is both detailed and practical. The best parameters were (C=10.0, penalty='l1', solver='saga'). The model was optimised using these parameters.

3.2.3 Evaluation of Key Differences Between Baseline and Optimised Models

This section compares the baseline Logistic Regression model to the optimised version regarding classification performance, bias–variance decomposition and statistical robustness.

3.2.3.1 Performance Comparison

On the NSL-KDD dataset, the baseline Logistic Regression achieved a strikingly good performance, with an overall accuracy of 0.773, an average precision of 0.804 and a weighted F1-score of 0.730. After hyperparameter optimisation was done, the model performed better, showing increases in accuracy to 0.813, which indicates a stronger uplift. The Matthews Correlation Coefficient (MCC), a balanced metric for multi-class classification, also increased from 0.669 to 0.726.

These results are summarised in Table 8:

Table 8: Comparative evaluation of baseline and optimised Logistic Regression models

Metric	Baseline Model	Optimised Model
Accuracy	0.773	0.813
Macro Avg - Precision	0.804	0.861
Macro Avg - Recall	0.546	0.707
Macro Avg – F1 Score	0.538	0.717
Weighted Avg - Precision	0.809	0.834
Weighted Avg - Recall	0.773	0.813
Weighted Avg F1-Score	0.730	0.780
MCC (Overall)	0.669	0.726
Average Bias	0.227	0.187
Average Variance	0.008	0.018
Expected Loss	0.226	0.187
Goodness	0.774	0.813

3.2.3.2 Bias–Variance Trade-Off Discussion

Bias–variance decomposition is an important feature of model evaluation (Gao, 2021). Since bias improved from 0.227 to 0.187, the new model was better able to capture patterns in the training data. Reducing bias correlates with improved recall and F1 scores, and this positive impact is usually stronger for minority classes like r2l and u2r.

Variance rose slightly (to 0.018 from 0.008) as is usual when the model has greater flexibility in what it tests. Even so, it is understandable since the expected loss is now lower (0.226 to 0.187) and the overall score higher (0.774 to 0.813).

3.2.3.3 Class-Wise Detection Capability

Using the optimised model confusion matrix, we noticed that it was more likely to detect instances from the probe and u2r classes, which struggle in imbalanced datasets such as NSL-KDD (Kaur & Gupta, 2024). After optimisation, the macro-averaged recall rose from 0.546 to 0.707, showing that sensitivity for all five attacks improved. For intrusion detection systems, not being able to detect the rare classes may cause major security problems.

Table 9: Class-wise comparison – Baseline vs Optimised Logistic Regression

Attack	Precision (Base)	Precision (Optimised)	Recall (Base)	Recall (Optimised)	F1-Score (Base)	F1-Score (Optimised)	MCC (Base)	MCC (Optimised)
benign	0.678	0.728	0.927	0.926	0.783	0.815	0.598	0.661
dos	0.971	0.944	0.839	0.906	0.900	0.925	0.861	0.889
probe	0.765	0.838	0.856	0.876	0.808	0.856	0.785	0.839
r2l	0.879	0.902	0.029	0.110	0.056	0.197	0.147	0.293
u2r	0.727	0.894	0.080	0.715	0.144	0.794	0.239	0.798

- All metrics improved for r2l and u2r classes, showing better detection of rare classes.
- Overall uplift seen in F1-score and MCC for every attack type.
- Precision slightly dropped for dos, but its recall and MCC improved — meaning better balance.

3.2.3.4 Interpretative Insights

What makes the optimised model effective is that it doesn't underfit or overfit. Though the baseline model was easy to use and ran quickly, it did not perform well with minority classes. Using C: 10.0, penalty: 'l1' and solver: 'saga', the model was able to represent important boundaries well and still perform efficiently. Especially, this design-maintained interpretability with L1 regularisation which supports feature selection and sparsity.

3.2.4 Impact on Model Performance

Accuracy was used as the main scoring method and GridSearchCV applied a 3-fold cross-validation process. Because of cross-validation, the evaluation was not influenced by a single train-test split and could handle new data well. Solver='saga', penalty='l1' and C=10.0 were found to be the best hyperparameters and the cross-validation accuracy was 0.985.

Picking the 'saga' solver and the l1 penalty is very important. Handling big data and applying l1 or l2 regularization are strengths of the saga solver. The l1 penalty makes the model sparse which acts as a form of feature selection. Besides, the high value of C=10.0 suggests that using only a little regularisation was the best choice for this dataset.

As a result of the optimisation process, many performance metrics improved noticeably. The optimised model scored better in macro precision, macro recall and Matthews Correlation Coefficient (MCC) than the baseline model, as discussed earlier. Also, bias–variance decomposition found that the model's bias went down (from 0.227 to 0.187), as did the expected loss (from 0.226 to 0.187). At the same time, variance went up (from 0.008 to 0.018). It means the model learns well from different data and does not overfit too much. Additionally, a final bias–variance decomposition was performed on the optimised model using StratifiedKFold validation to confirm that the generalisation behaviour remained stable.

In short, using the grid-based approach helped the model perform better and it was still easy to interpret and compute. The correct choice and confirmation of parameters helped build a stronger and more useful intrusion detection model.

3.3 Koo Wai Kit (TP081761) – Support Vector Machines and Ridge Classifier

This chapter investigates the performance of three classifiers from the scikit-learn library: SVC, LinearSVC and RidgeClf. The Boosted NSL-KDD training set was used to train all the models, and each model was tested on the pre-processed ppNSL test set.

3.3.1 Model Comparison

This section looks at how the classifiers performed when implemented with scikit-learn using their default hyperparameters. The same set of performance metrics is used to compare and evaluate the models. The performance of each model is summarized in Table 10 below.

Table 10: Performance metrics of SVC, LinearSVC, RidgeClf

Metric	SVC	LinearSVC	RidgeClf
Accuracy	0.804	0.766	0.741
Precision (Macro Avg)	0.860	0.718	0.773
Precision (Weighted Avg)	0.835	0.773	0.751
Recall (Macro Avg)	0.685	0.538	0.500
Recall (Weighted Avg)	0.804	0.766	0.741
F1 Score (Macro Avg)	0.681	0.534	0.493
F1 Score (Weighted Avg)	0.762	0.721	0.692
MCC (Overall)	0.713	0.658	0.622
Average Bias	0.195	0.234	0.259
Average Variance	0.007	0.008	0.007
Expected Loss	0.197	0.235	0.259
Goodness-of-Fit	0.803	0.765	0.741

Based on the results, SVC was the best performing classifier across all metrics. It had the highest accuracy (0.804) and goodness-of-fit (0.803), which means it fits the real class distribution well and can reliably predict outcomes. Both the macro and weighted average precision scores (0.860 and 0.835) are high, showing that the model can still separate classes well, even when class imbalance exists. It also had a higher rate of identifying true positives for all categories, as shown by its recall scores (macro: 0.685, weighted: 0.804) and it achieved the best balance between precision and recall with macro and weighted F1 scores of 0.681 and 0.762, respectively. The strong link between the model's predictions and the true labels is also backed by its Matthews Correlation Coefficient (0.713).

On the other hand, LinearSVC performed moderately, achieving decent accuracy (0.766) and weighted values that were similar to SVC, but it had lower scores on macro-averaged values. The precision and recall values (0.718 and 0.538) suggest that the model struggles to detect minority classes. This is a problem in intrusion detection since the model will have difficulties identifying attacks that are not common. RidgeClf performed worst, as it had the lowest accuracy (0.741), macro recall (0.500) and macro F1 score (0.493), showing that it did not handle the multi-class classification problem well. Additionally, both LinearSVC and RidgeClf

had a higher expected loss (0.235 and 0.259) and average bias (0.234 and 0.259), which means they made more prediction errors compared to SVC.

3.3.1.1 Baseline Model Selected for Optimisation

While LinearSVC is designed for efficiency with large datasets and only uses linear boundaries, SVC supports non-linear kernels and applies a one-vs-one classification strategy, making it more effective at finding complex patterns in the data (Scikit-Learn, n.d.). RidgeClf, on the other hand, treats classification as a regression problem by assigning {-1, 1} to each class, which may affect its ability to distinguish between classes (Scikit-Learn, n.d.).

Hence, SVC was chosen as the baseline model since it outperformed both LinearSVC and RidgeClf in all the evaluation metrics. This model will be further improved by applying optimisation strategy to achieve a better performance in the network intrusion detection task.

3.3.2 Optimisation Strategies Applied and Their Impact on Model Performance

3.3.2.1 Selected Optimisation Strategy

Hyperparameter tuning with cross-validation was selected as the optimisation strategy to improve the baseline model's performance. Hyperparameter tuning was selected for this study since it greatly affects how well the model performs and how well it generalises. It requires adjusting configuration variables like regularisation strength before model training to reduce loss and increase accuracy. Unlike model parameters which are discovered during training, hyperparameters are set by the user to control how the model learns. For this reason, tuning is necessary to get the best results and avoid problems such as underfitting or overfitting (Belcic, 2024). In addition, cross-validation helps by testing each hyperparameter set across several data splits, making sure the chosen model is not overly influenced by just one training set.

Some of the most common hyperparameter tuning methods are grid search, randomised search, Bayesian optimization and Hyperband. Randomised search is quicker for large hyperparameter spaces and Bayesian optimization uses probabilistic sampling to be more efficient, but they may not find the best combinations when the search spaces are small. Hyperband, by contrast, helps speed up tuning by removing configurations that do not perform well early on. However, for this study, grid search is chosen because it checks every possible combination of hyperparameters values from a defined set of values (Belcic, 2024). Since the hyperparameter space is not big and a detailed comparison is required, this method is a good choice for

optimising the SVM model. Thus, hyperparameter tuning with cross-validation will be implemented based on the GridSearchCV model of scikit-learn.

3.3.2.2 Selected SVC hyperparameters for tuning

For SVMs, it is crucial to tune the main hyperparameters like C, kernel, and gamma. The C parameter affects the balance between a smooth classification boundary and accuracy, where a low C helps the model to generalise more, and a high C makes it more precise but can lead to overfitting. The kernel parameter decides how the data is divided, where different kernels can be used to handle specific data types. Moreover, the gamma parameter determines the importance of a single training example. If it is too high, the model may fit the training set too well and become overfitted, but if it is too low, it might not fit the data enough and become underfitted. When these hyperparameters are tuned properly, it leads to a balanced bias-variance tradeoff and helps the model generalise better to new data (Belcic, 2024).

Initially, a larger set of hyperparameters and their values were used for grid search, but due to the significant training time required, only the most impactful ones were kept. Table 11 below outlines the hyperparameters selected for tuning, along with justification on why each parameter is selected and the rationale for testing a particular set of values.

Table 11: SVC hyperparameters selected for tuning

Parameter	Justification	Values to Test & Rationale
C	Controls the balance between having a low error during training and a low error during testing, which helps avoid overfitting.	**0.1, 1, 10** → Using values on a log scale helps examine how sensitive the model is to C (Wainer & Fonseca, 2021).
kernel	Chooses the type of transformation for the input data which impacts how effective the model is at finding complex patterns.	**'poly', 'rbf', 'sigmoid'** → All available kernel options in the library except 'linear' and 'precomputed' should be tested, because 'linear'was already tested in LinearSVC and there is no precomputed kernel (Scikit-Learn, n.d.).
gamma	Controls the influence of each training example which affects	**0.1, 1, 10**

	how smoothly the decision boundary is formed. Only used in 'rbf', 'poly' and 'sigmoid' kernels.	→ Study suggests checking gamma values at different orders of magnitude to observe a wide variety of behaviors (Kharki et al., 2024).

Besides choosing the hyperparameters to tune, additional parameters are set in GridSearchCV to control how GridSearchCV evaluates and validates the models. The scoring parameter is set to 'balanced_accuracy', which is helpful when the data is imbalanced since it computes the average recall for all classes (Scikit-Learn, n.d.). Also, the cv parameter is set to 3, which means that each hyperparameter combination will be evaluated using 3-fold cross-validation. This way, the data is divided into three groups and each group is used once for validation, while the other two are used for training. This ensures that the model does not overfit to a particular subset of the data.

3.3.3 Evaluation of Key Differences Between Baseline and Optimised Models

Once the grid search is completed, the combination of hyperparameter values that achieve the best score will be used to train an optimised model. The best combination of values returned by GridSearchCV is {'C': 1, 'kernel': 'poly', 'gamma': 10}. The default values for these parameters are {'C': 1, 'kernel': 'rbf', 'gamma': 'scale'}. The results of GridSearchCV indicate that the best hyperparameters are not the same as the defaults, since the kernel is now 'poly' and gamma is set to 10. This suggests a preference for a polynomial decision boundary that pays more attention to individual data points. Although C did not change, the new kernel and gamma values indicate that the data requires a more complex model (Yeh & Lu, 2019).

The results of the optimisations are evaluated by comparing the new performance results with those of the baseline model. Table 12 below compares the performance differences between the baseline model and the optimised model.

Table 12: Performance metrics of baseline and optimised SVC model

Metric	**Baseline Model**	**Optimised Model**	**Difference**
Accuracy	0.804	0.841	+0.037
Precision (Macro Avg)	0.860	0.805	-0.055
Precision (Weighted Avg)	0.835	0.841	+0.006
Recall (Macro Avg)	0.685	0.753	+0.068

Recall (Weighted Avg)	0.804	0.841	+0.037
F1 Score (Macro Avg)	0.681	0.753	+0.072
F1 Score (Weighted Avg)	0.762	0.824	+0.062
MCC (Overall)	0.713	0.764	+0.051
Average Bias	0.195	0.159	-0.036
Average Variance	0.007	0.048	+0.041
Expected Loss	0.197	0.160	-0.037
Goodness-of-Fit	0.803	0.840	+0.037

The comparison of the baseline and optimized SVC models shows that nearly all metrics have improved, which proves that hyperparameter tuning works well. Accuracy went up from 0.804 to 0.841, and there were significant improvements in recall and F1 scores for both macro and weighted averages. This means that the optimised model not only performed better for all classes but also handled the minority classes more effectively. The key to this improvement was changing the kernel from the default radial basis function (RBF) kernel to polynomial kernel. Even though the RBF kernel is effective for certain datasets, sometimes it leads to smoother decision boundaries (Izmailov et al., 2013). Alternatively, the polynomial kernel can handle more complex patterns in the data, so it is useful for cases where the boundaries between classes are not easy to define and are highly complex (Yeh & Lu, 2019). As a result, the model was able to distinguish between classes that were overlapping or close to each other, which improved its recall and made its performance more balanced for all classes.

Additionally, when the gamma value was changed from 'scale' to 10, the model became more affected by each training example. This is because ‘scale’ is data-dependent and it is usually a small value. Increasing the gamma helps the model pay more attention to individual training examples, which makes its decision boundaries more precise and flexible (Yeh & Lu, 2019). As a result, the recall and F1 scores increased, mainly for classes that were not well represented or that were hard to tell apart before. However, having such a sensitive model can sometimes lead to overfitting. The slight increase in average variance (from 0.007 to 0.048) suggests that the model is becoming more complex, though it still does not negatively influence its ability to generalise. The fact that weighted precision, recall and F1 score are improving steadily indicates that the model is handling the balance between bias and variance well.

Moreover, the MCC also improved from 0.713 to 0.764, which means the model's predictive ability improved, especially when dealing with multiple classes. The fact that the expected loss decreased (from 0.197 to 0.160) and the goodness-of-fit increased (from 0.803 to 0.840) means the optimised model fits the training data better and should perform better on new data. Although there was a small decrease in macro averaged precision from 0.860 to 0.805, this is a common tradeoff when trying to increase recall, especially for classes that are underrepresented (Prexawanprasut & Banditwattanawong, 2024). The fact that average bias went down from 0.195 to 0.159 suggests the model is learning the data more accurately. Overall, by switching to a polynomial kernel and increasing gamma, the model became better at handling the data's complexity, made fewer mistakes and improved its generalisation, leading to a more balanced and reliable performance.

References

Abdulhameed, A., Alazawi, S., & Hassan, G. (2024). An optimized model for network intrusion detection in the network operating system environment. *Mesopotamian Journal of CyberSecurity, 4*(3), 75-85. https://doi.org/10.58496/mjcs/2024/017

Andriyani, S., Lydia, M., & Efendi, S. (2023). Optimization of support vector machine algorithm using stunting data classification. *Prisma Sains Jurnal Pengkajian Ilmu Dan Pembelajaran Matematika Dan Ipa Ikip Mataram*, 11(1), 164. https://doi.org/10.33394/j-ps.v11i1.6619

Ayub, M. Y., Haider, U., Ali, H., Tashfeen, M. T. A., Shoukat, H., & Basit, A. W. (2023). An intelligent machine learning based intrusion detection system (ids) for smart cities networks. *EAI Endorsed Transactions on Smart Cities, 7*(1), e4. https://doi.org/10.4108/eetsc.v7i1.2825

Azimjonov, J., & Kim, T. (2024). Stochastic gradient descent classifier-based lightweight intrusion detection systems using the efficient feature subsets of datasets. *Expert Systems with Applications, 237*, 121493. https://doi.org/10.1016/j.eswa.2023.121493

Bacevicius, M., & Paulauskaite-Taraseviciene, A. (2023). Machine learning algorithms for raw and unbalanced intrusion detection data in a multi-class classification problem. *Applied Sciences, 13*(12), 7328. https://doi.org/10.3390/app13127328

Belcic, I. (2024, July 23). *What is hyperparameter tuning?*. IBM. https://www.ibm.com/think/topics/hyperparameter-tuning

Bian, W., Ding, S., & Xue, Y. (2017). Fingerprint image super resolution using sparse representation with ridge pattern prior by classification coupled dictionaries. *IET Biometrics*, 6(5), 342-350. https://doi.org/10.1049/iet-bmt.2016.0097

Gao, J. (2021). Bias-variance decomposition of absolute errors for diagnosing regression models of continuous data. Volume 2, Issue 8.

Hastie, T. (2020). Ridge regularizaton: an essential concept in data science. *Technometrics*, *62*(4), 426–433. https://doi.org/10.1080/00401706.2020.1791959

Izmailov, R., Vapnik, V., & Vashist, A. (2013). Multidimensional splines with infinite number of knots as SVM kernels. *International Joint Conference on Neural Network*, 1–7. https://doi.org/10.1109/IJCNN.2013.6706860

Kaur, R., & Gupta, N. (2024). Harnessing Decision Tree-guided Dynamic Oversampling for Intrusion Detection. *Engineering, Technology & Applied Science Research*, 17456-17463.

Khan, I., Khan, J., Bangash, S. H., Ahmad, W., Iftikhar, M. A., & Hameed, K. (2024). Intrusion detection using machine learning and deep learning models on cybersecurity attacks. *VFAST Transactions on Computer Sciences, 11*(2), 95–113. https://doi.org/10.21015/vtse.v12i2.1817

Kharki, A., Mechbouh, J., Wahbi, M., Alaoui, O., Boulaassal, H., Maâtouk, M., & Kharki, O. (2024). Optimizing SVM for argan tree classification using Sentinel-2 data: A case study in the Sous-Massa Region, Morocco. *Revista de Teledetección*. https://doi.org/10.4995/raet.2025.22060

Kuzborskij, I., Carlucci, F. M., & Caputo, B. (2016). When naïve bayes nearest neighbors meet convolutional neural networks. 2016 *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2100-2109*. https://doi.org/10.1109/cvpr.2016.231

Li, D., Ge, Q.-F., Zhang, P.-C., Xing, Y., Yang, Z., & Nai, W. (2020). Ridge regression with high order truncated gradient descent method. *International Conference on Intelligent Human-Machine Systems and Cybernetics*. https://doi.org/10.1109/IHMSC49165.2020.00063

Li, J., & Castagna, J. (2004). Support vector machine (SVM) pattern recognition to AVO classification. *Geophysical Research Letters*, 31(2). https://doi.org/10.1029/2003gl018299

Li, S. (2023). Research on the application of DT and K-NN-based data mining algorithms in network intrusion monitoring. *Third International Conference on Electronics, Electrical and Information Engineering (ICEEIE 2023)*, 112. https://doi.org/10.1117/12.3009151

Lorenzo, N., Rocha, R., Papaioannou, E., Mutz, Y., Tessaro, L., & Nunes, C. (2024). Feasibility of using a cheap colour sensor to detect blends of vegetable oils in avocado oil. *Foods*, 13(4), 572. https://doi.org/10.3390/foods13040572

Lu, S.-X., & Wang, X. (2004). A comparison among four SVM classification methods: LSVM, NLSVM, SSVM and NSVM. *International Conference on Machine Learning and Cybernetics*, *7*, 4277–4282. https://doi.org/10.1109/ICMLC.2004.1384589

Mandeep. (2022, April 5). *GridSearchCV: Key concepts*. Medium. https://medium.com/@Mandeep2002/gridsearchcv-key-concepts-8f98ceb633e4

Manyakov, N., Chumerin, N., Combaz, A., & Hulle, M. (2011). Comparison of classification methods for p300 brain-computer interface on disabled subjects. *Computational Intelligence and Neuroscience*, 2011, 1-12. https://doi.org/10.1155/2011/519868

Meidina, A., & Abidin, Z. (2023). Diagnosis of heart disease using optimized naïve bayes algorithm with particle swarm optimization and gain ratio. *Recursive Journal of Informatics, 1*(2), 47-54. https://doi.org/10.15294/rji.v1i2.67278

Murty, M. N., & Raghava, R. (2016). Kernel-based svm. *SpringerBriefs in Computer Science*, 57–67. https://doi.org/10.1007/978-3-319-41063-0_5

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . P. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2825 - 2830.

Permana, A., Ainiyah, K., & Holle, K. (2021). Analisis perbandingan algoritma decision tree, knn, dan naive bayes untuk prediksi kesuksesan start-up. *Jiska (Jurnal Informatika Sunan Kalijaga), 6*(3), 178-188. https://doi.org/10.14421/jiska.2021.6.3.178-188

Prexawanprasut, T., & Banditwattanawong, T. (2024). Improving minority class recall through a novel cluster-based oversampling technique. *Informatics (Basel)*, *11*(2), 35. https://doi.org/10.3390/informatics11020035

Rajaguru, H., & Chakravarthy, S. (2019). Analysis of decision tree and k-nearest neighbor algorithm in the classification of breast cancer. *Asian Pacific Journal of Cancer Prevention, 20*(12), 3777-3781. https://doi.org/10.31557/apjcp.2019.20.12.3777

Razaque, A., Frej, M., Almiani, M., Alotaibi, M., & Alotaibi, B. (2021). Improved support vector machine enabled radial basis function and linear variants for remote sensing image classification. *Sensors, 21*(13), 4431. https://doi.org/10.3390/s21134431

Religia, Y., Pranoto, G. T., & Suwancita, I. M. (2021). Analysis of the use of particle swarm optimization on naïve bayes for classification of credit bank applications. *JISA(Jurnal Informatika Dan Sains), 4*(2), 133-137. https://doi.org/10.31326/jisa.v4i2.946

Scikit-Learn. (n.d.). *Balanced_accuracy_score*. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html#sklearn.metrics.balanced_accuracy_score

Scikit-Learn. (n.d.). *LinearDiscriminantAnalysis*. https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html

Scikit-Learn. (n.d.). *LogisticRegression*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Scikit-Learn. (n.d.). *QuadraticDiscriminantAnalysis*. https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis.html

Scikit-Learn. (n.d.). *RidgeClassifier*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html

Scikit-Learn. (n.d.). *SGDClassifier*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

Scikit-Learn. (n.d.). *SVC*. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Sibyan, H., Švajlenka, J., Hermawan, H., Faqih, N., & Arrizqi, A. N. (2022). Thermal comfort prediction accuracy with machine learning between regression analysis and naïve bayes classifier. *Sustainability, 14*(23), 15663. https://doi.org/10.3390/su142315663

Sifaou, H., Kammoun, A., & Alouini, M.-S. (2020). High-dimensional quadratic discriminant analysis under spiked covariance model. *arXiv preprint arXiv:2006.14325*. https://arxiv.org/abs/2006.14325

Stodt, F., Theoleyre, F., & Reich, C. (2024). Advancing network survivability and reliability: Integrating XAI-enhanced autoencoders and LDA for effective detection of unknown attacks. In *2024 20th International Conference on the Design of Reliable Communication Networks (DRCN)* (pp. 1–6). IEEE. https://doi.org/10.1109/DRCN60692.2024.10539141

Tang, S. (2022). Applications of machine learning in the industry of healthcare. *Highlights in Science, Engineering and Technology, 1*, 87-96. https://doi.org/10.54097/hset.v1i.432

Tuning the hyper-parameters. (n.d.). Retrieved from scikit learn: https://scikit-learn.org/stable/modules/grid_search.html

Veziroğlu, M., Veziroğlu, E., & Bucak, İ. Ö. (2024). Performance comparison between naive bayes and machine learning algorithms for news classification. *Bayesian Inference - Recent Trends*. https://doi.org/10.5772/intechopen.1002778

Wainer, J., & Fonseca, P. (2021). How to tune the RBF SVM hyperparameters? An empirical evaluation of 18 search algorithms. *Artificial Intelligence Review, 54*(6), 4771–4797. doi:10.1007/s10462-021-10011-5

Wong, K. K. L. (2023). *Support vector machine*. 149–176. https://doi.org/10.1002/9781394217519.ch8

Xu, C., Zhang, H., & Peng, D. (2012). Study of fault diagnosis based on svm for turbine generator unit. *2012 8th International Conference on Natural Computation*, 110–113. https://doi.org/10.1109/icnc.2012.6234698

Yeh, L.-C., & Lu, C.-C. (2019). *Parameter optimization of polynomial kernel SVM from miniCV* (pp. 496–507). Springer, Cham. https://doi.org/10.1007/978-3-030-37599-7_41

Zhang, H., Zhu, S., Zhao, J., Gao, M., Shou, Z., & Liang, Y. (2016). Anomaly detection in industrial control networks using hybrid LDA-autoencoder based models. In *Proceedings of the International Conference on Artificial Intelligence and Engineering Applications* (pp. 53–58). Atlantis Press. https://doi.org/10.2991/aiea-16.2016.11

Zhou, X., Chen, W., & Li, Y. (2024). netQDA: Local network-guided high-dimensional quadratic discriminant analysis. *Mathematics, 12*(23), 3823. https://doi.org/10.3390/math12233823

Appendix A (Androjuniko)

Essential ML process for Intrusion Detection

python	3.7.13	scikit-learn	1.0.2
numpy	1.19.5	pandas	1.3.5

Import the main libraries

[1]:
```
import numpy
import pandas

from time import time

import os
data_path = '../datasets/NSL_KDD'
```

import the local library

[2]:
```
# add parent folder path where lib folder is
import sys
if ".." not in sys.path:import sys; sys.path.insert(0, '..')
```

[3]:
```
from mylib import show_labels_dist, show_metrics, bias_var_metrics
```

Import the Dataset

[4]:
```
# Using boosted Train and preprocessed Test

data_file = os.path.join(data_path, 'NSL_boosted-2.csv')
train_df = pandas.read_csv(data_file)
print('Train Dataset: {} rows, {} columns'.format(train_df.shape[0], train_df.shape[1]))

data_file = os.path.join(data_path, 'NSL_ppTest.csv')
test_df = pandas.read_csv(data_file)
print('Test Dataset: {} rows, {} columns'.format(test_df.shape[0], test_df.shape[1]))
```

```
Train Dataset: 63280 rows, 43 columns
Test Dataset: 22544 rows, 43 columns
```

Data Preparation and EDA (unique to this dataset)

- *Check column names of numeric attributes*

[5]:
```
trnn = train_df.select_dtypes(include=['float64','int64']).columns
tstn = test_df.select_dtypes(include=['float64','int64']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Numeric features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Numeric features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Numeric features in the train_set that are not in the test_set: None
Numeric features in the test_set that are not in the train_set: None
```

- *Check column names of categorical attributes*

[6]:
```
trnn = train_df.select_dtypes(include=['object']).columns
tstn = test_df.select_dtypes(include=['object']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Categorical features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Categorical features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n\t',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Categorical features in the train_set that are not in the test_set: None
Categorical features in the test_set that are not in the train_set: None
```

- *Check for missing values*

[7]:
```
cnt=0
print('Missing Values - Train Set')
for col in train_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(train_df[col])
    if (len(nnul)!=len(train_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

cnt=0
print('Missing Values - Test Set')
for col in test_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(test_df[col])
    if (len(nnul)!=len(test_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

# address missing values here
```

```
Missing Values - Train Set
Total 0 features with null values
Missing Values - Test Set
Total 0 features with null values
```

- *Quick visual check of unique values, deal with unique identifiers*

[8]:
```
# Identify columns with only one value
# or with number of unique values == number of rows
n_eq_one = []
n_eq_all = []

print('Unique value count: Train (',train_df.shape[0],'rows ) ~ Test(',test_df.shape[0],'rows )')
for col in train_df.columns:
    lctrn = len(train_df[col].unique())
    lctst = len(test_df[col].unique())
    print(col, ' ::> ', lctrn, ' ~ ', lctst)
    if (lctrn == 1) and (lctrn == lctst):
        n_eq_one.append(train_df[col].name)
    if lctrn == train_df.shape[0]:
        n_eq_all.append(train_df[col].name)
```

```
Unique value count: Train ( 63280 rows ) ~ Test( 22544 rows )
duration  ::>  1657  ~  624
protocol_type  ::>  3  ~  3
service  ::>  70  ~  64
flag  ::>  11  ~  11
src_bytes  ::>  2553  ~  1149
dst_bytes  ::>  6633  ~  3650
land  ::>  2  ~  2
wrong_fragment  ::>  3  ~  3
urgent  ::>  3  ~  4
hot  ::>  24  ~  16
num_failed_logins  ::>  6  ~  5
logged_in  ::>  2  ~  2
num_compromised  ::>  49  ~  23
root_shell  ::>  2  ~  2
su_attempted  ::>  3  ~  3
num_root  ::>  44  ~  20
num_file_creations  ::>  28  ~  9
num_shells  ::>  3  ~  4
num_access_files  ::>  9  ~  5
num_outbound_cmds  ::>  1  ~  1
is_host_login  ::>  2  ~  2
is_guest_login  ::>  2  ~  2
count  ::>  511  ~  495
srv_count  ::>  489  ~  457
serror_rate  ::>  87  ~  88
srv_serror_rate  ::>  75  ~  82
rerror_rate  ::>  81  ~  90
srv_rerror_rate  ::>  63  ~  93
same_srv_rate  ::>  101  ~  75
diff_srv_rate  ::>  88  ~  99
srv_diff_host_rate  ::>  61  ~  84
dst_host_count  ::>  256  ~  256
dst_host_srv_count  ::>  256  ~  256
dst_host_same_srv_rate  ::>  101  ~  101
dst_host_diff_srv_rate  ::>  101  ~  101
dst_host_same_src_port_rate  ::>  101  ~  101
dst_host_srv_diff_host_rate  ::>  67  ~  58
dst_host_serror_rate  ::>  101  ~  99
dst_host_srv_serror_rate  ::>  100  ~  101
dst_host_rerror_rate  ::>  101  ~  101
dst_host_srv_rerror_rate  ::>  101  ~  100
label  ::>  40  ~  38
atakcat  ::>  5  ~  5
```

[9]:
```
# Drop columns with only one value
if len(n_eq_one) > 0:
    print('Dropping single-valued features')
    print(n_eq_one)
    train_df.drop(n_eq_one, axis=1, inplace=True)
    test_df.drop(n_eq_one, axis=1, inplace=True)

# Drop or bin columns with number of unique values == number of rows
if len(n_eq_all) > 0:
    print('Dropping unique identifiers')
    print(n_eq_all)
    train_df.drop(n_eq_all, axis=1, inplace=True)
    test_df.drop(n_eq_all, axis=1, inplace=True)

# continue with featue selection / feature engineering
```

```
Dropping single-valued features
['num_outbound_cmds']
```

- *Check categorical feature values:*
  differences will be resolved by one-hot encoding the combined test and train sets

[10]:
```
trnn = train_df.select_dtypes(include=['object']).columns
for col in trnn:
    tr = train_df[col].unique()
    ts = test_df[col].unique()
    trd = numpy.setdiff1d(tr, ts)
    tsd = numpy.setdiff1d(ts, tr)

    print(col,'::> ')
    print("\tUnique text values in the train_set that are not in the test_set: ",end='')
    if len(trd) > 0:
        print('\n\t',trd)
    else:
        print('None')

    print("\tUnique text values in the test_set that are not in the train_set: ",end='')
    if len(tsd) > 0:
        print('\n\t',tsd)
    else:
        print('None')
```

```
protocol_type ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
service ::>
        Unique text values in the train_set that are not in the test_set:
         ['aol' 'harvest' 'http_2784' 'http_8001' 'red_i' 'urh_i']
        Unique text values in the test_set that are not in the train_set: None
flag ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
label ::>
        Unique text values in the train_set that are not in the test_set:
         ['spy' 'warezclient']
        Unique text values in the test_set that are not in the train_set: None
atakcat ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
```

- *Combine for processing classification target and text features*

[11]:
```
combined_df = pandas.concat([train_df, test_df])
print('Combined Dataset: {} rows, {} columns'.format(
    combined_df.shape[0], combined_df.shape[1]))
```

```
Combined Dataset: 85824 rows, 42 columns
```

- *Classification Target feature:* two columns of labels are available
  - Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
  - Multiclass: Use the category labels (atakcat)

[12]:
```
combined_df['label'].value_counts()
```

[12]:
```
label
normal             43383
neptune            25264
satan               2552
smurf               1988
ipsweep             1941
portsweep           1623
guess_passwd        1257
mscan               1046
warezmaster          954
back                 837
nmap                 820
apache2              774
processtable         719
teardrop             458
warezclient          445
snmpguess            364
saint                350
mailbomb             322
snmpgetattack        196
httptunnel           146
pod                  142
buffer_overflow       35
named                 25
ps                    22
multihop              21
sendmail              21
xterm                 20
rootkit               18
land                  16
xlock                 14
xsnoop                 8
ftp_write              7
phf                    6
loadmodule             6
imap                   6
perl                   5
sqlattack              4
worm                   4
udpstorm               4
spy                    1
Name: count, dtype: int64
```

[13]:
```
combined_df['atakcat'].value_counts()
```

[13]:
```
atakcat
benign    43383
dos       30524
probe      8332
r2l        3329
u2r         256
Name: count, dtype: int64
```

[14]:
```
# Set the classification target
twoclass = False     # True or False
```

[15]:
```
if twoclass:
# Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
# new single column data structure is a [series]
    labels_df = combined_df['label'].copy()
    labels_df[labels_df != 'normal'] = 'attack'
else:
# Multiclass: Use the category labels (atakcat)
# new single column data structure is a [[dataframe]]
# rename the column and convert to a series for later
    labels_df = combined_df[['atakcat']].copy()
    labels_df.rename(columns={'atakcat':'label'}, inplace=True)
    labels_df = labels_df.squeeze('columns')

# drop target features
combined_df.drop(['label'], axis=1, inplace=True)
combined_df.drop(['atakcat'], axis=1, inplace=True)
```

- *One-Hot Encoding the categorical (text) features*

[16]:
```
# put the names into a python list - for pandas.get_dummies()
categori = combined_df.select_dtypes(include=['object']).columns
category_cols = categori.tolist()
print(category_cols)
```

```
['protocol_type', 'service', 'flag']
```

[17]:
```
# generate a sorted list of unique values of categorical features
# we will get a new column for each one with get_dummies()

for col in categori:
    ul = numpy.sort(list(combined_df[col].unique()))
    print(col, ' ::> ', ul)
    print()
```

```
protocol_type  ::>  ['icmp' 'tcp' 'udp']

service  ::>  ['IRC' 'X11' 'Z39_50' 'aol' 'auth' 'bgp' 'courier' 'csnet_ns' 'ctf'
 'daytime' 'discard' 'domain' 'domain_u' 'echo' 'eco_i' 'ecr_i' 'efs'
 'exec' 'finger' 'ftp' 'ftp_data' 'gopher' 'harvest' 'hostnames' 'http'
 'http_2784' 'http_443' 'http_8001' 'imap4' 'iso_tsap' 'klogin' 'kshell'
 'ldap' 'link' 'login' 'mtp' 'name' 'netbios_dgm' 'netbios_ns'
 'netbios_ssn' 'netstat' 'nnsp' 'nntp' 'ntp_u' 'other' 'pm_dump' 'pop_2'
 'pop_3' 'printer' 'private' 'red_i' 'remote_job' 'rje' 'shell' 'smtp'
 'sql_net' 'ssh' 'sunrpc' 'supdup' 'systat' 'telnet' 'tftp_u' 'tim_i'
 'time' 'urh_i' 'urp_i' 'uucp' 'uucp_path' 'vmnet' 'whois']

flag  ::>  ['OTH' 'REJ' 'RSTO' 'RSTOS0' 'RSTR' 'S0' 'S1' 'S2' 'S3' 'SF' 'SH']
```

[18]:
```
# Apply to the list of Categorical columns (text fields)
features_df = pandas.get_dummies(combined_df, columns=category_cols)
features_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 85824 entries, 0 to 22543
Columns: 121 entries, duration to flag_SH
dtypes: bool(84), float64(15), int64(22)
memory usage: 31.8 MB
```

[19]:
```
# generate a list of numeric columns for scaling - After test // train split
numeri = combined_df.select_dtypes(include=['float64','int64']).columns
print(numeri.to_list())
```

```
['duration', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell', 'su_attempted', 'num_root', 'num_file_
creations', 'num_shells', 'num_access_files', 'is_host_login', 'is_guest_login', 'count', 'srv_count', 'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate',
'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_r
ate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate', 'dst_host_srv_rerror_rate']
```

Create Test // Train Datasets

> Normally we split the dataset into train 70 % // test 30 % like this
> `from sklearn.model_selection import train_test_split`
> `X_train, X_test, y_train, y_test =`
> `train_test_split(features_df, labels_df,`
> `test_size=0.3, stratify=labels_df, random_state=42)`

[20]:
```
# Restore the train // test split: slice 1 Dataframe into 2
features_train = features_df.iloc[:len(train_df),:].copy()    # X_train
features_train.reset_index(inplace=True, drop=True)
# pandas has a lot of rules about returning a 'view' vs. a copy from slice
# so we force it to create a new dataframe [avoiding SettingWithCopy Warning]
features_test = features_df.iloc[len(train_df):,:].copy()     # X_test
features_test.reset_index(inplace=True, drop=True)

# Restore the train // test split: slice 1 Series into 2
labels_train = labels_df[:len(train_df)]              # y_train
labels_train.reset_index(inplace=True, drop=True)

labels_test = labels_df[len(train_df):]               # y_test
labels_test.reset_index(inplace=True, drop=True)
```

Scaling comes *after* test // train split

[21]:
```
# scaling the Numeric columns
# StandardScaler range: -1 to 1, MinMaxScaler range: zero to 1

# from skLearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

# skLearn docs say
#   "Don't cheat - fit only on training data, then transform both"
#   fit() expects 2D array: reshape(-1, 1) for single col or (1, -1) single row

for i in numeri:
    arr = numpy.array(features_train[i])
    scale = MinMaxScaler().fit(arr.reshape(-1, 1))
    features_train[i] = scale.transform(arr.reshape(len(arr),1))

    arr = numpy.array(features_test[i])
    features_test[i] = scale.transform(arr.reshape(len(arr),1))
```

Classifier Selection

[22]:
```
# prepare list
models = []

##  --  Linear  --  ##
#from sklearn.linear_model import LogisticRegression
#models.append (("LogReg",LogisticRegression()))
#from sklearn.linear_model import SGDClassifier
#models.append (("StoGradDes",SGDClassifier()))
#from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
#models.append(("LinearDA", LinearDiscriminantAnalysis()))
#from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
#models.append(("QuadraticDA", QuadraticDiscriminantAnalysis()))

##  --  Support Vector  --  ##
#from sklearn.svm import SVC
#models.append(("SupportVectorClf", SVC()))
#from sklearn.svm import LinearSVC
#models.append(("LinearSVC", LinearSVC()))
#from sklearn.linear_model import RidgeClassifier
#models.append (("RidgeClf",RidgeClassifier()))

##  --  Non-Linear  --  ##
from sklearn.tree import DecisionTreeClassifier
models.append (("DecisionTree",DecisionTreeClassifier()))
from sklearn.naive_bayes import GaussianNB
models.append (("GaussianNB",GaussianNB()))
from sklearn.neighbors import KNeighborsClassifier
models.append(("K-NNeighbors", KNeighborsClassifier(weights = 'distance')))

##  --  Ensemble: bagging  --  ##
#from sklearn.ensemble import RandomForestClassifier
#models.append(("RandomForest", RandomForestClassifier()))
##  --  Ensemble: boosting  --  ##
#from sklearn.ensemble import AdaBoostClassifier
#models.append(("AdaBoost", AdaBoostClassifier()))
#from sklearn.ensemble import GradientBoostingClassifier
#models.append(("GradientBoost", GradientBoostingClassifier()))

##  --  NeuralNet (simplest)  --  ##
#from sklearn.linear_model import Perceptron
#models.append (("SingleLayerPtron",Perceptron()))
#from sklearn.neural_network import MLPClassifier
#models.append(("MultiLayerPtron", MLPClassifier()))

print(models)
```

```
[('DecisionTree', DecisionTreeClassifier()), ('GaussianNB', GaussianNB()), ('K-NNeighbors', KNeighborsClassifier(weights='distance'))]
```

[23]:
```
# dataset names
X_train = features_train
y_train = labels_train
X_test = features_test
y_test = labels_test
labels_col = 'label'
# library names
pd = pandas
np = numpy
```

Target Label Distributions (standard block)

[24]:
```
# from our local library
show_labels_dist(X_train,X_test,y_train,y_test)
```

```
features_train: 63280 rows, 121 columns
features_test:  22544 rows, 121 columns

labels_train: 63280 rows, 1 column
labels_test:  22544 rows, 1 column

Frequency and Distribution of labels
        label  %_train  label  %_test
label
benign  33672    53.21   9711   43.08
dos     23066    36.45   7458   33.08
probe    5911     9.34   2421   10.74
r2l       575     0.91   2754   12.22
u2r        56     0.09    200    0.89
```

Fit and Predict (standard block)

```
[25]: # evaluate each model in turn
      results = []

      print('macro average: unweighted mean per label')
      print('weighted average: support-weighted mean per label')
      print('MCC: correlation between prediction and ground truth')
      print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

      for name, clf in models:
          trs = time()
          print('Confusion Matrix:', name)

          clf.fit(X_train, y_train)
          ygx = clf.predict(X_test)
          results.append((name, ygx))

          tre = time() - trs
          print ("Run Time {} seconds".format(round(tre,2)) + '\n')

      # Easy way to ensure that the confusion matrix rows and columns
      #   are labeled exactly as the classifier has coded the classes
      #   [[note the _ at the end of clf.classes_ ]]

          show_metrics(y_test, ygx, clf.classes_)   # from our local library
          print('\nParameters: ', clf.get_params(), '\n\n')
```

```
macro average: unweighted mean per label
weighted average: support-weighted mean per label
MCC: correlation between prediction and ground truth
     (+1 perfect, 0 random prediction, -1 inverse)


Confusion Matrix: DecisionTree
Run Time 0.86 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8773       109         603       212        14
train:dos             258      7156          42         2         0
train:probe            93        69        2245        10         4
train:r2l            1849         1         107       570       227
train:u2r              44         0           5         2       149

~~~~
      benign :  FPR = 0.175   FNR = 0.097
         dos :  FPR = 0.012   FNR = 0.040
       probe :  FPR = 0.038   FNR = 0.073
         r2l :  FPR = 0.011   FNR = 0.793
         u2r :  FPR = 0.011   FNR = 0.255

   macro avg :  FPR = 0.049   FNR = 0.252
weighted avg :  FPR = 0.040   FNR = 0.162

~~~~
              precision    recall  f1-score   support

      benign      0.796     0.903     0.846      9711
         dos      0.976     0.960     0.967      7458
       probe      0.748     0.927     0.828      2421
         r2l      0.716     0.207     0.321      2754
         u2r      0.378     0.745     0.502       200

    accuracy                          0.838     22544
   macro avg      0.723     0.748     0.693     22544
weighted avg      0.837     0.838     0.817     22544

~~~~
MCC: Overall :  0.763
      benign :  0.722
         dos :  0.952
       probe :  0.811
         r2l :  0.347
         u2r :  0.525

Parameters:  {'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split':
2, 'min_weight_fraction_leaf': 0.0, 'random_state': None, 'splitter': 'best'}


Confusion Matrix: GaussianNB
Run Time 0.38 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         4792        58           9      4544       308
train:dos            3138      3017          14      1240        49
train:probe          1193       255         145       516       312
train:r2l             770         0           0      1363       621
train:u2r               3         0           0        10       187

~~~~
      benign :  FPR = 0.398   FNR = 0.507
         dos :  FPR = 0.021   FNR = 0.595
       probe :  FPR = 0.001   FNR = 0.940
         r2l :  FPR = 0.319   FNR = 0.505
         u2r :  FPR = 0.058   FNR = 0.065

   macro avg :  FPR = 0.159   FNR = 0.522
weighted avg :  FPR = 0.145   FNR = 0.578

~~~~
              precision    recall  f1-score   support

      benign      0.484     0.493     0.489      9711
         dos      0.906     0.405     0.559      7458
       probe      0.863     0.060     0.112      2421
         r2l      0.178     0.495     0.261      2754
         u2r      0.127     0.935     0.223       200

    accuracy                          0.422     22544
   macro avg      0.512     0.478     0.329     22544
weighted avg      0.624     0.422     0.442     22544

~~~~
MCC: Overall :  0.209
      benign :  0.096
         dos :  0.509
       probe :  0.211
         r2l :  0.122
         u2r :  0.332

Parameters:  {'priors': None, 'var_smoothing': 1e-09}
```

```
Confusion Matrix: K-NNeighbors
Run Time 1.49 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8877        65         619       146         4
train:dos             243      7103          32         0         0
train:probe           212        72        2128         1         8
train:r2l            2001       227           5       518         3
train:u2r              35         0           0         2       163

~~~~
      benign :  FPR = 0.194   FNR = 0.086
         dos :  FPR = 0.024   FNR = 0.037
       probe :  FPR = 0.033   FNR = 0.121
         r2l :  FPR = 0.008   FNR = 0.812
         u2r :  FPR = 0.001   FNR = 0.185

   macro avg :  FPR = 0.052   FNR = 0.248
weighted avg :  FPR = 0.041   FNR = 0.163

~~~~
              precision    recall  f1-score   support

      benign      0.781     0.914     0.842      9711
         dos      0.952     0.963     0.957      7458
       probe      0.764     0.879     0.818      2421
         r2l      0.777     0.188     0.303      2754
         u2r      0.916     0.815     0.862       200

    accuracy                          0.837     22544
   macro avg      0.838     0.752     0.757     22544
weighted avg      0.836     0.837     0.812     22544

~~~~
MCC: Overall :  0.760
      benign :  0.713
         dos :  0.936
       probe :  0.796
         r2l :  0.349
         u2r :  0.863

Parameters:  {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params': None, 'n_jobs': None, 'n_neighbors': 5, 'p': 2, 'weights': 'distance'}
```

Bias - Variance Decomposition (standard block)

[26]:
```
# from our local library
# reduce (cross-validation) folds for faster results
folds = 20
for name, clf in models:
    print('Bias // Variance Decomposition:', name)
    bias_var_metrics(X_train,X_test,y_train,y_test,clf,folds)
```

```
Bias // Variance Decomposition: DecisionTree
   Average bias: 0.153
   Average variance: 0.063
   Average expected loss: 0.152  "Goodness": 0.848

Bias // Variance Decomposition: GaussianNB
   Average bias: 0.583
   Average variance: 0.102
   Average expected loss: 0.557  "Goodness": 0.443

Bias // Variance Decomposition: K-NNeighbors
   Average bias: 0.166
   Average variance: 0.015
   Average expected loss: 0.167  "Goodness": 0.833
```

Hyperparameter Tuning General pattern:

baseline model

> 1. Classifier selection
> 2. Fit and Predict
> 3. Bias-Variance Tradeoff

optimised model

> 4. Select strategy and hyperparameters
> 5. Plug in the best parameter values
> 6. Fit and Predict
> 7. Bias-Variance Tradeoff

---

[28]:
```
# for graphs
import matplotlib.pyplot as plt
```

[29]:
```
# Each parameter increases time exponentially
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from time import time

# ---- Specific to each classifier ---- #
clf = models[0][1]  # Use Decision Tree (first model in skML-complete.ipynb)

# hyperparameter: values to test (minimum 2x2 Grid)
value_grid = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'criterion': ['gini', 'entropy']
}

# ---- Cross Validation ---- #
folds = 5  # Increased for better robustness
scorer = 'balanced_accuracy'  # Suitable for imbalanced data

# Start the timer
trs = time()

print('GridSearchCV:', folds, 'folds, timer started')
print(f'{clf} with scoring = {scorer}')

grid_search = GridSearchCV(estimator=clf, param_grid=value_grid,
                           scoring=scorer,
                           cv=StratifiedKFold(n_splits=folds, shuffle=True, random_state=50),
                           verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Print details
means = grid_search.cv_results_['mean_test_score']
stds = grid_search.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, grid_search.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))

# Print best parameters and score
print("\tBest parameters: {}".format(grid_search.best_params_))
print("\tBest CV score: {:.3f}".format(grid_search.best_score_))

# Update the model with best parameters
models[0][1].set_params(**grid_search.best_params_)

tre = time() - trs
print("Run Time {} seconds".format(round(tre, 2)) + '\n')
```

```
GridSearchCV: 5 folds, timer started
DecisionTreeClassifier() with scoring = balanced_accuracy
Fitting 5 folds for each of 24 candidates, totalling 120 fits
0.868 (+/-0.058) for {'criterion': 'gini', 'max_depth': None, 'min_samples_split': 2}
0.865 (+/-0.071) for {'criterion': 'gini', 'max_depth': None, 'min_samples_split': 5}
0.860 (+/-0.067) for {'criterion': 'gini', 'max_depth': None, 'min_samples_split': 10}
0.785 (+/-0.035) for {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 2}
0.784 (+/-0.035) for {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 5}
0.784 (+/-0.035) for {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 10}
0.849 (+/-0.062) for {'criterion': 'gini', 'max_depth': 20, 'min_samples_split': 2}
0.850 (+/-0.076) for {'criterion': 'gini', 'max_depth': 20, 'min_samples_split': 5}
0.844 (+/-0.073) for {'criterion': 'gini', 'max_depth': 20, 'min_samples_split': 10}
0.863 (+/-0.055) for {'criterion': 'gini', 'max_depth': 30, 'min_samples_split': 2}
0.859 (+/-0.069) for {'criterion': 'gini', 'max_depth': 30, 'min_samples_split': 5}
0.858 (+/-0.065) for {'criterion': 'gini', 'max_depth': 30, 'min_samples_split': 10}
0.847 (+/-0.049) for {'criterion': 'entropy', 'max_depth': None, 'min_samples_split': 2}
0.857 (+/-0.034) for {'criterion': 'entropy', 'max_depth': None, 'min_samples_split': 5}
0.850 (+/-0.054) for {'criterion': 'entropy', 'max_depth': None, 'min_samples_split': 10}
0.797 (+/-0.072) for {'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 2}
0.796 (+/-0.057) for {'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 5}
0.800 (+/-0.070) for {'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 10}
0.856 (+/-0.063) for {'criterion': 'entropy', 'max_depth': 20, 'min_samples_split': 2}
0.845 (+/-0.042) for {'criterion': 'entropy', 'max_depth': 20, 'min_samples_split': 5}
0.830 (+/-0.073) for {'criterion': 'entropy', 'max_depth': 20, 'min_samples_split': 10}
0.845 (+/-0.049) for {'criterion': 'entropy', 'max_depth': 30, 'min_samples_split': 2}
0.846 (+/-0.072) for {'criterion': 'entropy', 'max_depth': 30, 'min_samples_split': 5}
0.835 (+/-0.074) for {'criterion': 'entropy', 'max_depth': 30, 'min_samples_split': 10}
        Best parameters: {'criterion': 'gini', 'max_depth': None, 'min_samples_split': 2}
        Best CV score: 0.868
Run Time 11.82 seconds
```

[31]:
```
# Grid_Search returns a dict of best parameters
models[0][1].set_params(**grid_search.best_params_)  # Apply to Decision Tree

print(models[0][0], ': Best Values ')
print(models[0][1])
```

```
DecisionTree : Best Values
DecisionTreeClassifier()
```

[32]:
```
# Evaluate only the Decision Tree model after hyperparameter tuning
results = []

print('macro average: unweighted mean per label')
print('weighted average: support-weighted mean per label')
print('MCC: correlation between prediction and ground truth')
print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

# Only evaluate the Decision Tree (first model in the models list)
name, clf = models[0]  # Decision Tree with tuned parameters
trs = time()
print('Confusion Matrix:', name)

clf.fit(X_train, y_train)
ygx = clf.predict(X_test)
results.append((name, ygx))

tre = time() - trs
print("Run Time {} seconds".format(round(tre, 2)) + '\n')

# Display metrics using the local library function
show_metrics(y_test, ygx, clf.classes_)  # from mylib
print('Parameters: ', clf.get_params(), '\n\n')
```

```
macro average: unweighted mean per label
weighted average: support-weighted mean per label
MCC: correlation between prediction and ground truth
     (+1 perfect, 0 random prediction, -1 inverse)
```

```
Confusion Matrix: DecisionTree
Run Time 0.92 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8776       109         609       208         9
train:dos             121      7287          42         8         0
train:probe           106        57        2245        10         3
train:r2l            1629         1         231       656       237
train:u2r              40         0           5         6       149

~~~~
      benign :  FPR = 0.148   FNR = 0.096
         dos :  FPR = 0.011   FNR = 0.023
       probe :  FPR = 0.044   FNR = 0.073
         r2l :  FPR = 0.012   FNR = 0.762
         u2r :  FPR = 0.011   FNR = 0.255

   macro avg :  FPR = 0.045   FNR = 0.242
weighted avg :  FPR = 0.038   FNR = 0.152

~~~~
              precision    recall  f1-score   support

      benign      0.822     0.904     0.861      9711
         dos      0.978     0.977     0.977      7458
       probe      0.717     0.927     0.809      2421
         r2l      0.739     0.238     0.360      2754
         u2r      0.374     0.745     0.498       200

    accuracy                          0.848     22544
   macro avg      0.726     0.758     0.701     22544
weighted avg      0.848     0.848     0.830     22544

~~~~
MCC: Overall :  0.778
      benign :  0.750
         dos :  0.966
       probe :  0.791
         r2l :  0.381
         u2r :  0.523
Parameters:  {'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split':
2, 'min_weight_fraction_leaf': 0.0, 'random_state': None, 'splitter': 'best'}
```

[33]:
```python
from mlxtend.evaluate import bias_variance_decomp
from sklearn.preprocessing import LabelEncoder
import numpy as np
import warnings
from time import time

# Encode target labels to integers
ytrain = LabelEncoder().fit_transform(labels_train)
ytest = LabelEncoder().fit_transform(labels_test)

# Convert DataFrames to NumPy arrays
X_train_np = features_train.values
X_test_np = features_test.values

# Use the best-tuned model from GridSearchCV
clf = grid_search.best_estimator_

# Bias-Variance Decomposition
folds = 12
trs = time()
print('Bias-Variance Decomposition: Optimized Decision Tree\\n\\tTimer started')

avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
    clf, X_train_np, ytrain, X_test_np, ytest,
    loss='0-1_loss', num_rounds=folds, random_seed=150
)

# Calculate goodness
goodness = 1 - avg_expected_loss

tre = time() - trs
print("\\nResults:")
print("   Bias:     %.3f" % avg_bias)
print("   Variance: %.3f" % avg_var)
print("   Expected Loss: %.3f" % avg_expected_loss)
print("   Goodness: %.3f" % goodness)  # Add goodness to the output
```

```
Bias-Variance Decomposition: Optimized Decision Tree\n\tTimer started
\nResults:
   Bias:     0.136
   Variance: 0.063
   Expected Loss: 0.151
   Goodness: 0.849
\nRun Time: 25.68 seconds
```

Appendix B (Suryakrishnan Balamurugan)

> **Essential ML process for Intrusion Detection**
>
python	3.7.13	scikit-learn	1.0.2
numpy	1.19.5	pandas	1.3.5

SURYAKRISHNAN BALAMURUGAN - TP080525

Import the main libraries

[1]:
```
import numpy
import pandas

from time import time

import os
data_path = '../datasets/NSL_KDD'
```

import the local library

[2]:
```
# add parent folder path where lib folder is
import sys
if ".." not in sys.path:import sys; sys.path.insert(0, '..')
```

[3]:
```
from mylib import show_labels_dist, show_metrics, bias_var_metrics
```

Import the Dataset

[4]:
```
# Using boosted Train and preprocessed Test

data_file = os.path.join(data_path, 'NSL_boosted-2.csv')
train_df = pandas.read_csv(data_file)
print('Train Dataset: {} rows, {} columns'.format(train_df.shape[0], train_df.shape[1]))

data_file = os.path.join(data_path, 'NSL_ppTest.csv')
test_df = pandas.read_csv(data_file)
print('Test Dataset: {} rows, {} columns'.format(test_df.shape[0], test_df.shape[1]))
```

```
Train Dataset: 63280 rows, 43 columns
Test Dataset: 22544 rows, 43 columns
```

Data Preparation and EDA (unique to this dataset)

- *Check column names of numeric attributes*

[5]:
```
trnn = train_df.select_dtypes(include=['float64','int64']).columns
tstn = test_df.select_dtypes(include=['float64','int64']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Numeric features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Numeric features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Numeric features in the train_set that are not in the test_set: None
Numeric features in the test_set that are not in the train_set: None
```

- *Check column names of categorical attributes*

[6]:
```
trnn = train_df.select_dtypes(include=['object']).columns
tstn = test_df.select_dtypes(include=['object']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Categorical features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Categorical features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n\t',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Categorical features in the train_set that are not in the test_set: None
Categorical features in the test_set that are not in the train_set: None
```

- *Check for missing values*

[7]:
```
cnt=0
print('Missing Values - Train Set')
for col in train_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(train_df[col])
    if (len(nnul)!=len(train_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

cnt=0
print('Missing Values - Test Set')
for col in test_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(test_df[col])
    if (len(nnul)!=len(test_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

# address missing values here
```

```
Missing Values - Train Set
Total 0 features with null values
Missing Values - Test Set
Total 0 features with null values
```

- *Quick visual check of unique values, deal with unique identifiers*

- *Quick visual check of unique values, deal with unique identifiers*

[8]:
```
# Identify columns with only one value
# or with number of unique values == number of rows
n_eq_one = []
n_eq_all = []

print('Unique value count: Train (',train_df.shape[0],'rows ) ~ Test(',test_df.shape[0],'rows )')
for col in train_df.columns:
    lctrn = len(train_df[col].unique())
    lctst = len(test_df[col].unique())
    print(col, ' ::> ', lctrn, ' ~ ', lctst)
    if (lctrn == 1) and (lctrn == lctst):
        n_eq_one.append(train_df[col].name)
    if lctrn == train_df.shape[0]:
        n_eq_all.append(train_df[col].name)
```

```
Unique value count: Train ( 63280 rows ) ~ Test( 22544 rows )
duration  ::>  1657  ~  624
protocol_type  ::>  3  ~  3
service  ::>  70  ~  64
flag  ::>  11  ~  11
```

```
src_bytes  ::>  2553  ~  1149
dst_bytes  ::>  6633  ~  3650
land  ::>  2  ~  2
wrong_fragment  ::>  3  ~  3
urgent  ::>  3  ~  4
hot  ::>  24  ~  16
num_failed_logins  ::>  6  ~  5
logged_in  ::>  2  ~  2
num_compromised  ::>  49  ~  23
root_shell  ::>  2  ~  2
su_attempted  ::>  3  ~  3
num_root  ::>  44  ~  20
num_file_creations  ::>  28  ~  9
num_shells  ::>  3  ~  4
num_access_files  ::>  9  ~  5
num_outbound_cmds  ::>  1  ~  1
is_host_login  ::>  2  ~  2
is_guest_login  ::>  2  ~  2
count  ::>  511  ~  495
srv_count  ::>  489  ~  457
serror_rate  ::>  87  ~  88
srv_serror_rate  ::>  75  ~  82
rerror_rate  ::>  81  ~  90
srv_rerror_rate  ::>  63  ~  93
same_srv_rate  ::>  101  ~  75
diff_srv_rate  ::>  88  ~  99
srv_diff_host_rate  ::>  61  ~  84
dst_host_count  ::>  256  ~  256
dst_host_srv_count  ::>  256  ~  256
dst_host_same_srv_rate  ::>  101  ~  101
dst_host_diff_srv_rate  ::>  101  ~  101
dst_host_same_src_port_rate  ::>  101  ~  101
dst_host_srv_diff_host_rate  ::>  67  ~  58
dst_host_serror_rate  ::>  101  ~  99
dst_host_srv_serror_rate  ::>  100  ~  101
dst_host_rerror_rate  ::>  101  ~  101
dst_host_srv_rerror_rate  ::>  101  ~  100
label  ::>  40  ~  38
atakcat  ::>  5  ~  5
```

[9]:
```
# Drop columns with only one value
if len(n_eq_one) > 0:
    print('Dropping single-valued features')
    print(n_eq_one)
    train_df.drop(n_eq_one, axis=1, inplace=True)
    test_df.drop(n_eq_one, axis=1, inplace=True)

# Drop or bin columns with number of unique values == number of rows
if len(n_eq_all) > 0:
    print('Dropping unique identifiers')
    print(n_eq_all)
    train_df.drop(n_eq_all, axis=1, inplace=True)
    test_df.drop(n_eq_all, axis=1, inplace=True)

# continue with featue selection / feature engineering
```

```
Dropping single-valued features
['num_outbound_cmds']
```

- *Check categorical feature values:*
  differences will be resolved by one-hot encoding the combined test and train sets

[10]:
```
trnn = train_df.select_dtypes(include=['object']).columns
for col in trnn:
    tr = train_df[col].unique()
    ts = test_df[col].unique()
    trd = numpy.setdiff1d(tr, ts)
    tsd = numpy.setdiff1d(ts, tr)

    print(col,'::> ')
    print("\tUnique text values in the train_set that are not in the test_set: ",end='')
    if len(trd) > 0:
        print('\n\t',trd)
    else:
        print('None')

    print("\tUnique text values in the test_set that are not in the train_set: ",end='')
    if len(tsd) > 0:
        print('\n\t',tsd)
```

```
    else:
        print('None')
```

```
protocol_type ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
service ::>
        Unique text values in the train_set that are not in the test_set:
         ['aol' 'harvest' 'http_2784' 'http_8001' 'red_i' 'urh_i']
        Unique text values in the test_set that are not in the train_set: None
flag ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
label ::>
        Unique text values in the train_set that are not in the test_set:
         ['spy' 'warezclient']
        Unique text values in the test_set that are not in the train_set: None
atakcat ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
```

- *Combine for processing classification target and text features*

[11]:
```
combined_df = pandas.concat([train_df, test_df])
print('Combined Dataset: {} rows, {} columns'.format(
    combined_df.shape[0], combined_df.shape[1]))
```

```
Combined Dataset: 85824 rows, 42 columns
```

- *Classification Target feature:* two columns of labels are available
  - Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
  - Multiclass: Use the category labels (atakcat)

[12]:
```
combined_df['label'].value_counts()
```

[12]:
```
normal             43383
neptune            25264
satan               2552


smurf               1988
ipsweep             1941
portsweep           1623
guess_passwd        1257
mscan               1046
warezmaster          954
back                 837
nmap                 820
apache2              774
processtable         719
teardrop             458
warezclient          445
snmpguess            364
saint                350
mailbomb             322
snmpgetattack        196
httptunnel           146
pod                  142
buffer_overflow       35
named                 25
ps                    22
multihop              21
sendmail              21
xterm                 20
rootkit               18
land                  16
xlock                 14
xsnoop                 8
ftp_write              7
phf                    6
loadmodule             6
imap                   6
perl                   5
sqlattack              4
worm                   4
udpstorm               4
spy                    1
Name: label, dtype: int64
```

[13]:
```
combined_df['atakcat'].value_counts()
```

[13]:
```
benign    43383
dos       30524
probe      8332
r2l        3329
u2r         256
Name: atakcat, dtype: int64
```

[14]:
```
# Set the classification target
twoclass = False     # True or False
```

[15]:
```
if twoclass:
# Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
# new single column data structure is a [series]
    labels_df = combined_df['label'].copy()
    labels_df[labels_df != 'normal'] = 'attack'
else:
# Multiclass: Use the category labels (atakcat)
# new single column data structure is a [[dataframe]]
# rename the column and convert to a series for later
    labels_df = combined_df[['atakcat']].copy()
    labels_df.rename(columns={'atakcat':'label'}, inplace=True)
    labels_df = labels_df.squeeze('columns')

# drop target features
combined_df.drop(['label'], axis=1, inplace=True)
combined_df.drop(['atakcat'], axis=1, inplace=True)
```

- *One-Hot Encoding the categorical (text) features*

[16]:
```
# put the names into a python list - for pandas.get_dummies()
categori = combined_df.select_dtypes(include=['object']).columns
category_cols = categori.tolist()
print(category_cols)
```

```
['protocol_type', 'service', 'flag']
```

[17]:
```
# generate a sorted list of unique values of categorical features
# we will get a new column for each one with get_dummies()

for col in categori:
    ul = numpy.sort(list(combined_df[col].unique()))
    print(col, ' ::> ', ul)
    print()
```

```
protocol_type  ::>  ['icmp' 'tcp' 'udp']

service  ::>  ['IRC' 'X11' 'Z39_50' 'aol' 'auth' 'bgp' 'courier' 'csnet_ns' 'ctf'
 'daytime' 'discard' 'domain' 'domain_u' 'echo' 'eco_i' 'ecr_i' 'efs'
 'exec' 'finger' 'ftp' 'ftp_data' 'gopher' 'harvest' 'hostnames' 'http'
 'http_2784' 'http_443' 'http_8001' 'imap4' 'iso_tsap' 'klogin' 'kshell'
 'ldap' 'link' 'login' 'mtp' 'name' 'netbios_dgm' 'netbios_ns'
 'netbios_ssn' 'netstat' 'nnsp' 'nntp' 'ntp_u' 'other' 'pm_dump' 'pop_2'
 'pop_3' 'printer' 'private' 'red_i' 'remote_job' 'rje' 'shell' 'smtp'
 'sql_net' 'ssh' 'sunrpc' 'supdup' 'systat' 'telnet' 'tftp_u' 'tim_i'
 'time' 'urh_i' 'urp_i' 'uucp' 'uucp_path' 'vmnet' 'whois']

flag  ::>  ['OTH' 'REJ' 'RSTO' 'RSTOS0' 'RSTR' 'S0' 'S1' 'S2' 'S3' 'SF' 'SH']
```

[18]:
```
# Apply to the list of Categorical columns (text fields)
features_df = pandas.get_dummies(combined_df, columns=category_cols)
features_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 85824 entries, 0 to 22543
Columns: 121 entries, duration to flag_SH
dtypes: float64(15), int64(22), uint8(84)
memory usage: 31.8 MB
```

[19]:
```
# generate a list of numeric columns for scaling - After test // train split
numeri = combined_df.select_dtypes(include=['float64','int64']).columns
print(numeri.to_list())
```

```
['duration', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins', 'logged_in', 'num_comprom
ised', 'root_shell', 'su_attempted', 'num_root', 'num_file_creations', 'num_shells', 'num_access_files', 'is_host_login', 'is_gu
est_login', 'count', 'srv_count', 'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate', 'diff_srv
_rate', 'srv_diff_host_rate', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_h
ost_same_src_port_rate', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rat
e', 'dst_host_srv_rerror_rate']
```

Create Test // Train Datasets

> Normally we split the dataset into train 70 % // test 30 % like this
> `from sklearn.model_selection import train_test_split`
> `X_train, X_test, y_train, y_test =`
> `train_test_split(features_df, labels_df,`
> `test_size=0.3, stratify=labels_df, random_state=42)`

[20]:
```
# Restore the train // test split: slice 1 Dataframe into 2
features_train = features_df.iloc[:len(train_df),:].copy()    # X_train
features_train.reset_index(inplace=True, drop=True)
# pandas has a lot of rules about returning a 'view' vs. a copy from slice
# so we force it to create a new dataframe [avoiding SettingWithCopy Warning]
features_test = features_df.iloc[len(train_df):,:].copy()     # X_test
features_test.reset_index(inplace=True, drop=True)

# Restore the train // test split: slice 1 Series into 2
labels_train = labels_df[:len(train_df)]                      # y_train
labels_train.reset_index(inplace=True, drop=True)

labels_test = labels_df[len(train_df):]                       # y_test
labels_test.reset_index(inplace=True, drop=True)
```

Next are standard steps for all datasets: *scaling, classifiers, results*

Scaling comes *after* test // train split

[21]:
```
# scaling the Numeric columns
# StandardScaler range: -1 to 1, MinMaxScaler range: zero to 1

# from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

# sklearn docs say
#   "Don't cheat - fit only on training data, then transform both"
#   fit() expects 2D array: reshape(-1, 1) for single col or (1, -1) single row

for i in numeri:
    arr = numpy.array(features_train[i])
    scale = MinMaxScaler().fit(arr.reshape(-1, 1))
    features_train[i] = scale.transform(arr.reshape(len(arr),1))

    arr = numpy.array(features_test[i])
    features_test[i] = scale.transform(arr.reshape(len(arr),1))
```

Classifier Selection

[22]:
```
# prepare list
models = []

##  --  Linear  --  ##
from sklearn.linear_model import LogisticRegression
models.append (("LogReg",LogisticRegression()))
from sklearn.linear_model import SGDClassifier
models.append (("StocGradDes",SGDClassifier()))
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
models.append(("LinearDA", LinearDiscriminantAnalysis()))
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
models.append(("QuadraticDA", QuadraticDiscriminantAnalysis()))

##  --  Support Vector  --  ##
#from sklearn.svm import SVC
#models.append(("SupportVectorClf", SVC()))
#from sklearn.svm import LinearSVC
#models.append(("LinearSVC", LinearSVC()))
#from sklearn.linear_model import RidgeClassifier
#models.append (("RidgeClf",RidgeClassifier()))

##  --  Non-linear  --  ##
#from sklearn.tree import DecisionTreeClassifier
#models.append (("DecisionTree",DecisionTreeClassifier()))
#from sklearn.naive_bayes import GaussianNB
#models.append (("GaussianNB",GaussianNB()))
#from sklearn.neighbors import KNeighborsClassifier
#models.append(("K-NNeighbors", KNeighborsClassifier()))

##  --  Ensemble: bagging  --  ##
#from sklearn.ensemble import RandomForestClassifier
#models.append(("RandomForest", RandomForestClassifier()))
##  --  Ensemble: boosting  --  ##
#from sklearn.ensemble import AdaBoostClassifier
#models.append(("AdaBoost", AdaBoostClassifier()))
#from sklearn.ensemble import GradientBoostingClassifier
#models.append(("GradientBoost", GradientBoostingClassifier()))
```

```
##  --  NeuralNet (simplest)  --  ##
#from sklearn.linear_model import Perceptron
#models.append (("SingleLayerPtron",Perceptron()))
#from sklearn.neural_network import MLPClassifier
#models.append(("MultiLayerPtron", MLPClassifier()))

print(models)
```

```
[('LogReg', LogisticRegression()), ('StocGradDes', SGDClassifier()), ('LinearDA', LinearDiscriminantAnalysis()), ('QuadraticDA',
QuadraticDiscriminantAnalysis())]
```

compatibility block for pasting in from sample code

[23]:
```
# dataset names
X_train = features_train
y_train = labels_train
X_test = features_test
y_test = labels_test
labels_col = 'label'
# library names
pd = pandas
np = numpy
```

Target Label Distributions (standard block)

[24]:
```
# from our local library
show_labels_dist(X_train,X_test,y_train,y_test)
```

```
features_train: 63280 rows, 121 columns
features_test:  22544 rows, 121 columns

labels_train: 63280 rows, 1 column
labels_test:  22544 rows, 1 column

Frequency and Distribution of labels
        label  %_train  label  %_test
benign  33672    53.21   9711   43.08
dos     23066    36.45   7458   33.08
probe    5911     9.34   2421   10.74
r2l       575     0.91   2754   12.22
u2r        56     0.09    200    0.89
```

... this is a good place for the ClassBalance visualizer ...

Fit and Predict (standard block)

[25]:
```
# evaluate each model in turn
results = []

print('macro average: unweighted mean per label')
print('weighted average: support-weighted mean per label')
print('MCC: correlation between prediction and ground truth')
print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

for name, clf in models:
    trs = time()
    print('Confusion Matrix:', name)

    clf.fit(X_train, y_train)
    ygx = clf.predict(X_test)
    results.append((name, ygx))
```

```
    tre = time() - trs
    print ("Run Time {} seconds".format(round(tre,2)) + '\n')

# Easy way to ensure that the confusion matrix rows and columns
#   are labeled exactly as the classifier has coded the classes
#   [[note the _ at the end of clf.classes_ ]]

    show_metrics(y_test, ygx, clf.classes_)   # from our local library
    print('\nParameters: ', clf.get_params(), '\n\n')
```

```
macro average: unweighted mean per label
weighted average: support-weighted mean per label
MCC: correlation between prediction and ground truth
     (+1 perfect, 0 random prediction, -1 inverse)

Confusion Matrix: LogReg
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs fa
iled to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
Run Time 25.18 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         9004        96         608         1         2
train:dos            1176      6257          25         0         0
train:probe           257        85        2073         6         0
train:r2l            2664         4           2        80         4
train:u2r             177         1           2         4        16

~~~~
      benign :  FPR = 0.333   FNR = 0.073
         dos :  FPR = 0.012   FNR = 0.161
       probe :  FPR = 0.032   FNR = 0.144
         r2l :  FPR = 0.001   FNR = 0.971
         u2r :  FPR = 0.000   FNR = 0.920

   macro avg :  FPR = 0.076   FNR = 0.454
weighted avg :  FPR = 0.057   FNR = 0.227

~~~~
              precision    recall  f1-score   support

      benign      0.678     0.927     0.783      9711
         dos      0.971     0.839     0.900      7458
       probe      0.765     0.856     0.808      2421
         r2l      0.879     0.029     0.056      2754
         u2r      0.727     0.080     0.144       200

    accuracy                          0.773     22544
   macro avg      0.804     0.546     0.538     22544
weighted avg      0.809     0.773     0.730     22544

~~~~
MCC: Overall :  0.669
      benign :  0.598
         dos :  0.861
       probe :  0.785
         r2l :  0.147
         u2r :  0.239

Parameters:  {'C': 1.0, 'class_weight': None, 'dual': False, 'fit_intercept': True, 'intercept_scaling': 1, 'l1_ratio': None, 'm
ax_iter': 100, 'multi_class': 'auto', 'n_jobs': None, 'penalty': 'l2', 'random_state': None, 'solver': 'lbfgs', 'tol': 0.0001,
'verbose': 0, 'warm_start': False}


Confusion Matrix: StocGradDes
Run Time 1.8 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8998        86         621         0         6
train:dos            1346      6091          21         0         0
train:probe           486       160        1760        13         2
train:r2l            2743         2           3         4         2
train:u2r             182         0           0         1        17

~~~~
      benign :  FPR = 0.371   FNR = 0.073
         dos :  FPR = 0.016   FNR = 0.183
       probe :  FPR = 0.032   FNR = 0.273
         r2l :  FPR = 0.001   FNR = 0.999
         u2r :  FPR = 0.000   FNR = 0.915

   macro avg :  FPR = 0.084   FNR = 0.489
weighted avg :  FPR = 0.063   FNR = 0.252
```

```
~~~~
              precision    recall  f1-score   support

      benign      0.654     0.927     0.767      9711
         dos      0.961     0.817     0.883      7458
       probe      0.732     0.727     0.729      2421
         r2l      0.222     0.001     0.003      2754
         u2r      0.630     0.085     0.150       200

    accuracy                          0.748     22544
   macro avg      0.640     0.511     0.506     22544
weighted avg      0.711     0.748     0.702     22544

~~~~
MCC: Overall :  0.631
      benign :  0.564
         dos :  0.838
       probe :  0.697
         r2l :  0.009
         u2r :  0.229

Parameters:  {'alpha': 0.0001, 'average': False, 'class_weight': None, 'early_stopping': False, 'epsilon': 0.1, 'eta0': 0.0, 'fi
t_intercept': True, 'l1_ratio': 0.15, 'learning_rate': 'optimal', 'loss': 'hinge', 'max_iter': 1000, 'n_iter_no_change': 5, 'n_j
obs': None, 'penalty': 'l2', 'power_t': 0.5, 'random_state': None, 'shuffle': True, 'tol': 0.001, 'validation_fraction': 0.1, 'v
erbose': 0, 'warm_start': False}


Confusion Matrix: LinearDA
Run Time 5.57 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         9308        85         280        22        16
train:dos            1327      5607         524         0         0
train:probe           497       176        1748         0         0
train:r2l            2079         0          16       649        10
train:u2r             155         0           0        10        35

~~~~
      benign :  FPR = 0.316   FNR = 0.041
         dos :  FPR = 0.017   FNR = 0.248


       probe :  FPR = 0.041   FNR = 0.278
         r2l :  FPR = 0.002   FNR = 0.764
         u2r :  FPR = 0.001   FNR = 0.825

   macro avg :  FPR = 0.075   FNR = 0.431
weighted avg :  FPR = 0.058   FNR = 0.231

~~~~
              precision    recall  f1-score   support

      benign      0.696     0.959     0.807      9711
         dos      0.956     0.752     0.842      7458
       probe      0.681     0.722     0.701      2421
         r2l      0.953     0.236     0.378      2754
         u2r      0.574     0.175     0.268       200

    accuracy                          0.769     22544
   macro avg      0.772     0.569     0.599     22544
weighted avg      0.811     0.769     0.750     22544

~~~~
MCC: Overall :  0.664
      benign :  0.647
         dos :  0.788
       probe :  0.664
         r2l :  0.448
         u2r :  0.314

Parameters:  {'covariance_estimator': None, 'n_components': None, 'priors': None, 'shrinkage': None, 'solver': 'svd', 'store_cov
ariance': False, 'tol': 0.0001}


Confusion Matrix: QuadraticDA
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
Run Time 5.81 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8968        52          22       669         0
```

```
train:dos            4860      1491         250       857         0
train:probe          1542        98         418       363         0
train:r2l             739         0          42      1972         1
train:u2r             135         0           0        50        15

~~~~
      benign :  FPR = 0.567   FNR = 0.077
         dos :  FPR = 0.010   FNR = 0.800
       probe :  FPR = 0.016   FNR = 0.827
          r2l :  FPR = 0.098   FNR = 0.284
          u2r :  FPR = 0.000   FNR = 0.925

   macro avg :  FPR = 0.138   FNR = 0.583
weighted avg :  FPR = 0.107   FNR = 0.429

~~~~
              precision    recall  f1-score   support

      benign      0.552     0.923     0.691      9711
         dos      0.909     0.200     0.328      7458
       probe      0.571     0.173     0.265      2421
          r2l      0.504     0.716     0.592      2754
          u2r      0.938     0.075     0.139       200

    accuracy                          0.571     22544
   macro avg      0.695     0.417     0.403     22544
weighted avg      0.670     0.571     0.508     22544

~~~~
MCC: Overall :  0.385
      benign :  0.393
         dos :  0.344
       probe :  0.274
          r2l :  0.535
          u2r :  0.264

Parameters:  {'priors': None, 'reg_param': 0.0, 'store_covariance': False, 'tol': 0.0001}
```

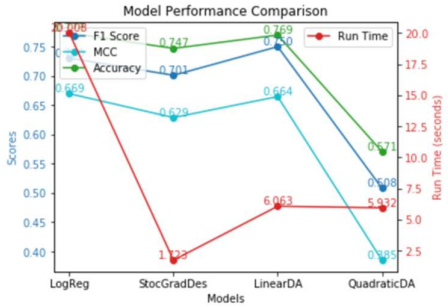
[26]:

```python
import matplotlib.pyplot as plt
import numpy as np
from time import time
from sklearn.metrics import f1_score, matthews_corrcoef, confusion_matrix, accuracy_score

# Assuming show_metrics returns the metrics we need and not just print them
def show_metrics(y_true, y_pred, classes):
    metrics = {}
    metrics['F1 Score'] = f1_score(y_true, y_pred, average='weighted')
    metrics['MCC'] = matthews_corrcoef(y_true, y_pred)
    metrics['Accuracy'] = accuracy_score(y_true, y_pred)
    return metrics

models = [('LogReg', LogisticRegression()),
          ('StocGradDes', SGDClassifier()),
          ('LinearDA', LinearDiscriminantAnalysis()),
          ('QuadraticDA', QuadraticDiscriminantAnalysis())]

results = []

print('macro average: unweighted mean per label')
print('weighted average: support-weighted mean per label')
print('MCC: correlation between prediction and ground truth')
print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

for name, clf in models:
    trs = time()
    print('Confusion Matrix:', name)
    clf.fit(X_train, y_train)
    ygx = clf.predict(X_test)
    tre = time() - trs
    print("Run Time {} seconds".format(round(tre, 2)) + '\n')
    metrics = show_metrics(y_test, ygx, clf.classes_)
    results.append((name, metrics, tre))

# Extract data for plotting
model_names = [result[0] for result in results]
f1_scores = [result[1]['F1 Score'] for result in results]
mcc_scores = [result[1]['MCC'] for result in results]
accuracy_scores = [result[1]['Accuracy'] for result in results]
```

```
run_times = [result[2] for result in results]

# Plotting the results
fig, ax1 = plt.subplots()

ax1.set_xlabel('Models')
ax1.set_ylabel('Scores', color='tab:blue')
ax1.plot(model_names, f1_scores, label='F1 Score', color='tab:blue', marker='o')
ax1.plot(model_names, mcc_scores, label='MCC', color='tab:cyan', marker='o')
ax1.plot(model_names, accuracy_scores, label='Accuracy', color='tab:green', marker='o')
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.legend(loc='upper left')

# Add text annotations for scores
for i, (model, f1, mcc, acc) in enumerate(zip(model_names, f1_scores, mcc_scores, accuracy_scores)):
    ax1.text(i, f1, f'{f1:.3f}', ha='center', va='bottom', color='tab:blue')
    ax1.text(i, mcc, f'{mcc:.3f}', ha='center', va='bottom', color='tab:cyan')
    ax1.text(i, acc, f'{acc:.3f}', ha='center', va='bottom', color='tab:green')

ax2 = ax1.twinx()
ax2.set_ylabel('Run Time (seconds)', color='tab:red')
ax2.plot(model_names, run_times, label='Run Time', color='tab:red', marker='o')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.legend(loc='upper right')

# Add text annotations for run times
for i, rt in enumerate(run_times):
    ax2.text(i, rt, f'{rt:.3f}', ha='center', va='bottom', color='tab:red')

fig.tight_layout()
plt.title('Model Performance Comparison')
plt.show()
```

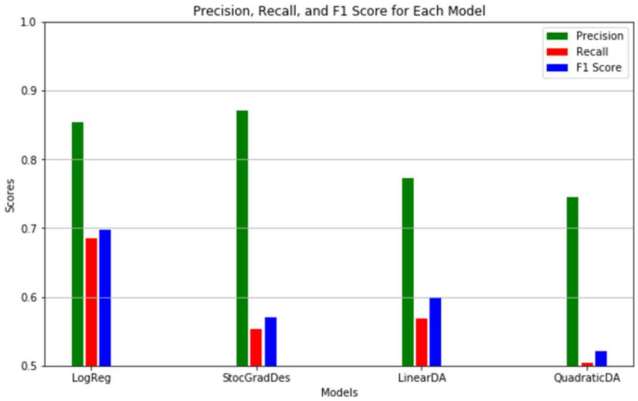
```
macro average: unweighted mean per label
weighted average: support-weighted mean per label
MCC: correlation between prediction and ground truth
     (+1 perfect, 0 random prediction, -1 inverse)
```

```
Confusion Matrix: LogReg
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs fa
iled to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
Run Time 20.01 seconds

Confusion Matrix: StocGradDes
Run Time 1.72 seconds

Confusion Matrix: LinearDA
Run Time 6.06 seconds

Confusion Matrix: QuadraticDA
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
Run Time 5.93 seconds
```



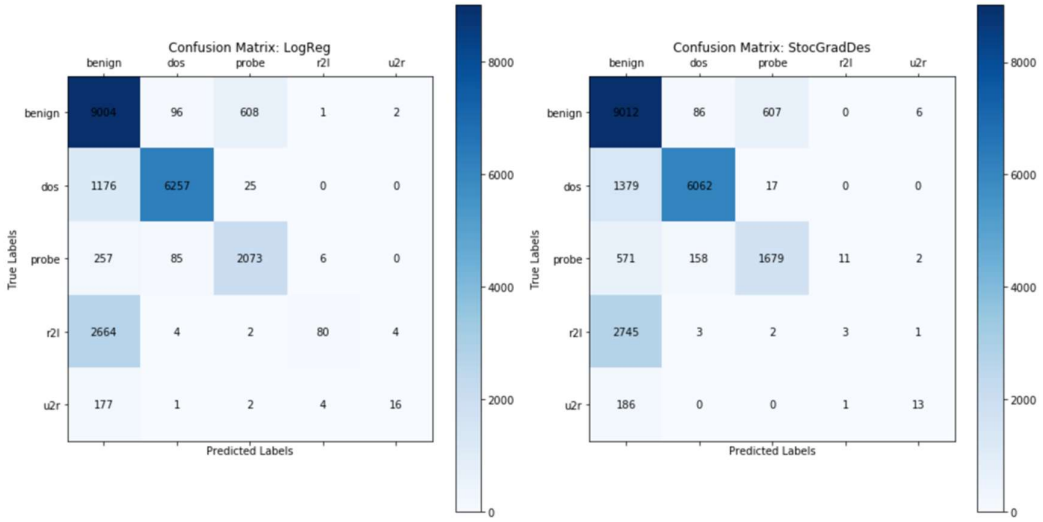
[28]:
```
import numpy as np
import matplotlib.pyplot as plt
import warnings
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnalysis
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

models = [
    ("LogReg", LogisticRegression(max_iter=2000, solver='lbfgs')),
    ("StocGradDes", SGDClassifier()),
    ("LinearDA", LinearDiscriminantAnalysis()),
    ("QuadraticDA", QuadraticDiscriminantAnalysis())
]

precision_results = []
recall_results = []
f1_results = []

for name, clf in models:
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        clf.fit(X_train_scaled, y_train)
        y_pred = clf.predict(X_test_scaled)

        precision = precision_score(y_test, y_pred, average='macro', zero_division=0)
        recall = recall_score(y_test, y_pred, average='macro', zero_division=0)
        f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)

        precision_results.append(precision)
        recall_results.append(recall)
        f1_results.append(f1)

labels = [name for name, _ in models]
x = np.arange(len(labels))
bar_width = 0.07
```

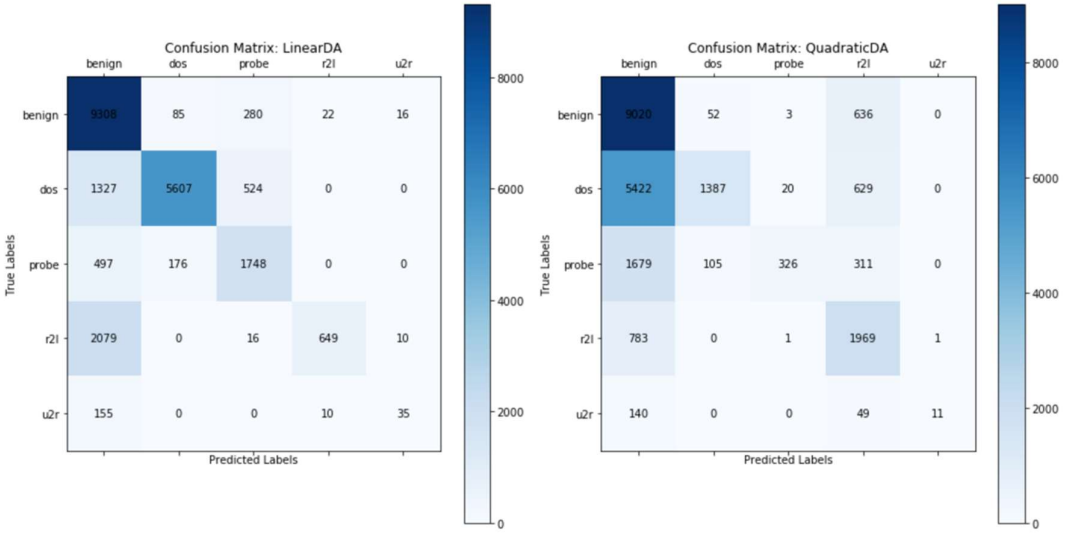
```
spacing = 0.015

plt.figure(figsize=(10, 6))

plt.bar(x - bar_width - spacing, precision_results, width=bar_width, label='Precision', color='green')
plt.bar(x, recall_results, width=bar_width, label='Recall', color='red')
plt.bar(x + bar_width + spacing, f1_results, width=bar_width, label='F1 Score', color='blue')

plt.xlabel('Models')
plt.ylabel('Scores')
plt.title('Precision, Recall, and F1 Score for Each Model')
plt.xticks(x, labels)
plt.legend()
plt.grid(axis='y')
plt.ylim(0.5, 1.0)
plt.show()
```



[29]:
```
import matplotlib.pyplot as plt
import numpy as np

# Confusion matrices for each model
conf_matrices = {
    'LogReg': np.array([
        [9004, 96, 608, 1, 2],
        [1176, 6257, 25, 0, 0],
        [257, 85, 2073, 6, 0],
        [2664, 4, 2, 80, 4],
        [177, 1, 2, 4, 16]
    ]),
    'StocGradDes': np.array([
        [9012, 86, 607, 0, 6],
        [1379, 6062, 17, 0, 0],
        [571, 158, 1679, 11, 2],
        [2745, 3, 2, 3, 1],
        [186, 0, 0, 1, 13]
    ]),
    'LinearDA': np.array([
        [9308, 85, 280, 22, 16],
        [1327, 5607, 524, 0, 0],
        [497, 176, 1748, 0, 0],
        [2079, 0, 16, 649, 10],
        [155, 0, 0, 10, 35]
    ]),
    'QuadraticDA': np.array([
        [9020, 52, 3, 636, 0],
        [5422, 1387, 20, 629, 0],
        [1679, 105, 326, 311, 0],
        [783, 0, 1, 1969, 1],
        [140, 0, 0, 49, 11]
    ])
}

# Labels for the axes
labels = ['benign', 'dos', 'probe', 'r2l', 'u2r']

# Create a 2x2 subplot layout
fig, axes = plt.subplots(2, 2, figsize=(14, 14))
```

```
for i, (model, conf_matrix) in enumerate(conf_matrices.items()):
    ax = axes[i // 2, i % 2]
    cax = ax.matshow(conf_matrix, cmap='Blues')
    fig.colorbar(cax, ax=ax)

    ax.set_title(f'Confusion Matrix: {model}')
    ax.set_xlabel('Predicted Labels')
    ax.set_ylabel('True Labels')

    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)

    # Add annotations inside each cell
    for (row, col), val in np.ndenumerate(conf_matrix):
        ax.text(col, row, f'{val}', ha='center', va='center', color='black')

plt.tight_layout()
plt.show()
```





Bias - Variance Decomposition (standard block)

[30]:
```
# from our local library
# reduce (cross-validation) folds for faster results
folds = 20
for name, clf in models:
    print('Bias // Variance Decomposition:', name)
    bias_var_metrics(X_train,X_test,y_train,y_test,clf,folds)
```

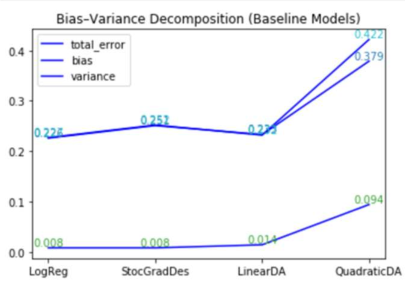
```
Bias // Variance Decomposition: LogReg
   Average bias: 0.227
   Average variance: 0.008
   Average expected loss: 0.226  "Goodness": 0.774

Bias // Variance Decomposition: StocGradDes
   Average bias: 0.252
   Average variance: 0.008
   Average expected loss: 0.251  "Goodness": 0.749

Bias // Variance Decomposition: LinearDA
   Average bias: 0.232
   Average variance: 0.014
   Average expected loss: 0.233  "Goodness": 0.767

Bias // Variance Decomposition: QuadraticDA
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
```

```
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are co
llinear
  warnings.warn("Variables are collinear")
   Average bias: 0.422
   Average variance: 0.094
   Average expected loss: 0.379  "Goodness": 0.621
```

[31]:

```
import matplotlib.pyplot as plt
import numpy as np

# Example data from your results
err = [0.226, 0.251, 0.233, 0.379]     # expected loss
bias = [0.227, 0.252, 0.232, 0.422]
var = [0.008, 0.008, 0.014, 0.094]
model_names = ['LogReg', 'StocGradDes', 'LinearDA', 'QuadraticDA']
rx = np.arange(len(model_names))

# Plot
fig, ax = plt.subplots()

ax.plot(rx, err, 'b', label='total_error')
ax.plot(rx, bias, 'b', label='bias')
ax.plot(rx, var, 'b', label='variance')

ax.legend()
ax.set_xticks(rx)
ax.set_xticklabels(model_names)
ax.set_title('Bias-Variance Decomposition (Baseline Models)')
```

```
# Annotate scores
for i, (e, b, v) in enumerate(zip(err, bias, var)):
    ax.text(i, e, f"{e:.3f}", ha='center', va='bottom', color='tab:blue')
    ax.text(i, b, f"{b:.3f}", ha='center', va='bottom', color='tab:cyan')
    ax.text(i, v, f"{v:.3f}", ha='center', va='bottom', color='tab:green')

plt.show()
```



[ ]:

Type Markdown and LaTeX: α^2

Baseline Model

> Select this block - Go to the Run menu - Run all Above
> Then paste in blocks below from the other examples
> and run them one at a time

Hyperparameter Tuning General pattern:
baseline model

> 1. Classifier selection
> 2. Fit and Predict
> 3. Bias-Variance Tradeoff

optimised model

> 4. Select strategy and hyperparameters
> 5. Plug in the best parameter values
> 6. Fit and Predict
> 7. Bias-Variance Tradeoff

[32]:
```
# Classifier Selection - for these examples
from sklearn.linear_model import LogisticRegression
models = []  # Initialize the list
# Append a tuple with the model name and instance
models.append(("LogReg", LogisticRegression()))
print(models[0][0])  # Should print "LogReg"
print(models[0][1])  # Should print the LogisticRegression model instance
```

```
LogReg
LogisticRegression()
```

[33]:
```
# dataset names
X_train = features_train
y_train = labels_train
X_test = features_test
y_test = labels_test
labels_col = 'label'
# library names
pd = pandas
np = numpy
```

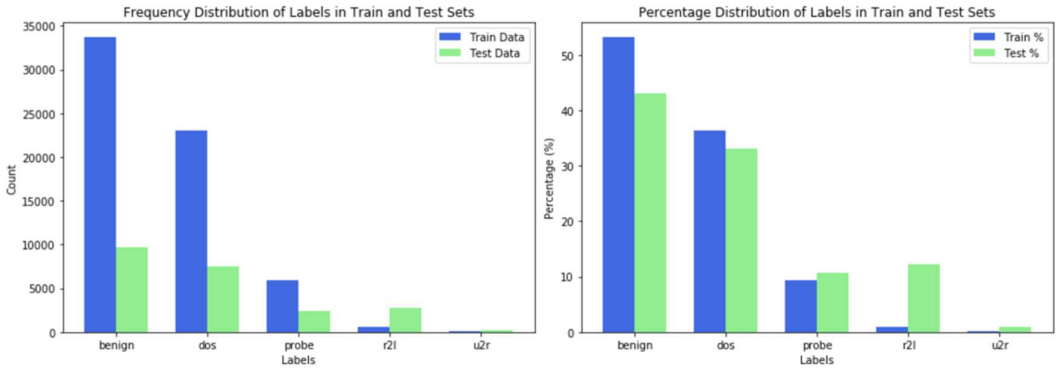
[34]:
```
# from our local library
show_labels_dist(X_train, X_test, y_train, y_test)
```

```
features_train: 63280 rows, 121 columns
features_test:  22544 rows, 121 columns

labels_train: 63280 rows, 1 column
labels_test:  22544 rows, 1 column

Frequency and Distribution of labels
        label  %_train  label  %_test
benign  33672    53.21   9711   43.08
dos     23066    36.45   7458   33.08
probe    5911     9.34   2421   10.74
r2l       575     0.91   2754   12.22
u2r        56     0.09    200    0.89
```

Visual Representation of Frequency and Distribution of Labels in Train and Test sets

[35]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Data for visualization
labels = ["benign", "dos", "probe", "r2l", "u2r"]
train_counts = [33672, 23066, 5911, 575, 56]
test_counts = [9711, 7458, 2421, 2754, 200]
train_percentages = [53.21, 36.45, 9.34, 0.91, 0.09]
test_percentages = [43.08, 33.08, 10.74, 12.22, 0.89]

# Create a figure with subplots
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Bar chart for absolute counts
x = np.arange(len(labels))
width = 0.35
axes[0].bar(x - width/2, train_counts, width, label="Train Data", color="royalblue")
axes[0].bar(x + width/2, test_counts, width, label="Test Data", color="lightgreen")
```

```
axes[0].set_xlabel("Labels")
axes[0].set_ylabel("Count")
axes[0].set_title("Frequency Distribution of Labels in Train and Test Sets")
axes[0].set_xticks(x)
axes[0].set_xticklabels(labels)
axes[0].legend()

# Bar chart for percentage distribution
axes[1].bar(x - width/2, train_percentages, width, label="Train %", color="royalblue")
axes[1].bar(x + width/2, test_percentages, width, label="Test %", color="lightgreen")

axes[1].set_xlabel("Labels")
axes[1].set_ylabel("Percentage (%)")
axes[1].set_title("Percentage Distribution of Labels in Train and Test Sets")
axes[1].set_xticks(x)
axes[1].set_xticklabels(labels)
axes[1].legend()

# Adjust layout and display the plots
plt.tight_layout()
plt.show()
```



[ ]:

Optimisation model select - Hyperparameter tuning for Logistic Regression

[36]:
```
# Default scorer for classification is sklearn.metrics.accuracy_score
# For imbalanced classification, the accuracy score is often uninformative
# For the list of options see
# https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

# average for each label weighted by support
#        (number of true instances for each label)
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import f1_score

eval_metric = ['wtd.avg.accuracy',
               'balanced_accuracy',
               'wtd.avg.f1_score',
               'f1_weighted']
```

```
# for graphs
import matplotlib.pyplot as plt
```

[ ]:

Parameter Testing

Strategy - Parameter grid search

[37]:
```
# Each parameter increases time exponentially!
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
import numpy as np
from time import time

# ---- Specific to each classifier! ---- #
clf = LogisticRegression(max_iter=5000)

# hyperparameter: values to test (minimum 2x2 Grid)
value_grid = [
    {'solver': ['liblinear'], 'penalty': ['l1', 'l2'], 'C': np.logspace(-3, 3, 4)},
    {'solver': ['lbfgs'], 'penalty': ['l2'], 'C': np.logspace(-3, 3, 4)},
    {'solver': ['saga'], 'penalty': ['l1', 'l2'], 'C': np.logspace(-3, 3, 4)},
    {'solver': ['newton-cg'], 'penalty': ['l2'], 'C': np.logspace(-3, 3, 4)},
]

# ----  Cross Validation  ---- #
folds = 3  # number of rounds

# choose a metric (as defined earlier in your notebook)
scorer = 'accuracy'

# ---- Run Grid Search ---- #
trs = time()

print('GridSearchCV:', folds, 'folds, timer started')
print('%s with scoring = %s' % (clf, scorer))
```

```
grid_search = GridSearchCV(estimator=clf, param_grid=value_grid,
                           scoring=scorer, cv=folds, verbose=1, n_jobs=-1)
grid_search.fit(features_train, labels_train)

# Print all mean scores
means = grid_search.cv_results_['mean_test_score']
stds = grid_search.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, grid_search.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))

# Best results
print("\tBest parameters: {}".format(grid_search.best_params_))
print("\tBest CV score: {:.3f}".format(grid_search.best_score_))

tre = time() - trs
print("Run Time {} seconds".format(round(tre, 2)) + '\n')
```

```
GridSearchCV: 3 folds, timer started
LogisticRegression(max_iter=5000) with scoring = accuracy
Fitting 3 folds for each of 24 candidates, totalling 72 fits
0.860 (+/-0.001) for {'C': 0.001, 'penalty': 'l1', 'solver': 'liblinear'}
0.925 (+/-0.009) for {'C': 0.001, 'penalty': 'l2', 'solver': 'liblinear'}
0.968 (+/-0.006) for {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
0.967 (+/-0.006) for {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.978 (+/-0.008) for {'C': 10.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.975 (+/-0.006) for {'C': 10.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.980 (+/-0.008) for {'C': 1000.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.978 (+/-0.008) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.946 (+/-0.005) for {'C': 0.001, 'penalty': 'l2', 'solver': 'lbfgs'}
0.970 (+/-0.004) for {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.979 (+/-0.009) for {'C': 10.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.985 (+/-0.008) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.868 (+/-0.002) for {'C': 0.001, 'penalty': 'l1', 'solver': 'saga'}
0.946 (+/-0.005) for {'C': 0.001, 'penalty': 'l2', 'solver': 'saga'}
0.970 (+/-0.004) for {'C': 0.1, 'penalty': 'l1', 'solver': 'saga'}
0.970 (+/-0.004) for {'C': 0.1, 'penalty': 'l2', 'solver': 'saga'}
0.985 (+/-0.008) for {'C': 10.0, 'penalty': 'l1', 'solver': 'saga'}
0.979 (+/-0.009) for {'C': 10.0, 'penalty': 'l2', 'solver': 'saga'}
0.985 (+/-0.008) for {'C': 1000.0, 'penalty': 'l1', 'solver': 'saga'}
0.985 (+/-0.008) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'saga'}



0.946 (+/-0.005) for {'C': 0.001, 'penalty': 'l2', 'solver': 'newton-cg'}
0.970 (+/-0.004) for {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.979 (+/-0.009) for {'C': 10.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.985 (+/-0.008) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'newton-cg'}
        Best parameters: {'C': 10.0, 'penalty': 'l1', 'solver': 'saga'}
        Best CV score: 0.985
Run Time 2342.18 seconds
```

[38]:
```
# Grid_Search returns a dict of best paraeters
models[0][1].set_params(**grid_search.best_params_)

print(models[0][0],': Best Values ')
print(models[0][1])
```

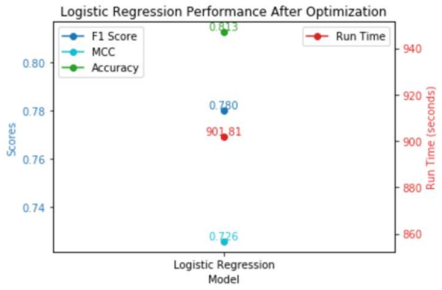
```
LogReg : Best Values
LogisticRegression(C=10.0, penalty='l1', solver='saga')
```

[40]:
```
# Evaluate only the best-tuned Logistic Regression model
clf = grid_search.best_estimator_  # Output from GridSearchCV

results = []

print('macro average: unweighted mean per label')
print('weighted average: support-weighted mean per label')
print('MCC: correlation between prediction and ground truth')
print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

from time import time
trs = time()

print('Confusion Matrix: Logistic Regression (Best Model)')
clf.fit(features_train, labels_train)
ygx = clf.predict(features_test)
results.append(("Logistic Regression", ygx))

tre = time() - trs
print("Run Time {} seconds".format(round(tre, 2)) + '\n')

# from your local mylib
show_metrics(labels_test, ygx, clf.classes_)
print('\nParameters: ', clf.get_params(), '\n\n')
```

```
macro average: unweighted mean per label
weighted average: support-weighted mean per label
MCC: correlation between prediction and ground truth
     (+1 perfect, 0 random prediction, -1 inverse)


Confusion Matrix: Logistic Regression (Best Model)
/Users/suryakrishnanb/miniconda3/lib/python3.8/site-packages/sklearn/linear_model/_sag.py:350: ConvergenceWarning: The max_iter
was reached which means the coef_ did not converge
  warnings.warn(
Run Time 905.43 seconds

             pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         8995       308         394         8         6
train:dos             692      6756          10         0         0
train:probe           198        82        2120        16         5
train:r2l            2436         3           5       304         6
train:u2r              42         4           2         9       143

~~~~
      benign :  FPR = 0.262   FNR = 0.074
         dos :  FPR = 0.026   FNR = 0.094
       probe :  FPR = 0.020   FNR = 0.124
         r2l :  FPR = 0.002   FNR = 0.890
         u2r :  FPR = 0.001   FNR = 0.285

   macro avg :  FPR = 0.062   FNR = 0.293
weighted avg :  FPR = 0.047   FNR = 0.187

~~~~
              precision    recall  f1-score   support

      benign      0.728     0.926     0.815      9711
         dos      0.944     0.906     0.925      7458
       probe      0.838     0.876     0.856      2421
         r2l      0.902     0.110     0.197      2754
         u2r      0.894     0.715     0.794       200

    accuracy                          0.813     22544
   macro avg      0.861     0.707     0.717     22544
weighted avg      0.834     0.813     0.780     22544

~~~~
MCC: Overall :  0.726
      benign :  0.661
         dos :  0.889


       probe :  0.839
         r2l :  0.293
         u2r :  0.798

 Parameters:  {'C': 10.0, 'class_weight': None, 'dual': False, 'fit_intercept': True, 'intercept_scaling': 1, 'l1_ratio': None,
 'max_iter': 5000, 'multi_class': 'auto', 'n_jobs': None, 'penalty': 'l1', 'random_state': None, 'solver': 'saga', 'tol': 0.0001,
 'verbose': 0, 'warm_start': False}
```

[41]:
```
from sklearn.metrics import f1_score, accuracy_score, matthews_corrcoef
import matplotlib.pyplot as plt

# Prepare metrics based on previous predictions
model_names = ["Logistic Regression"]
rx = range(len(model_names))

f1_scores = [f1_score(labels_test, ygx, average='weighted')]
mcc_scores = [matthews_corrcoef(labels_test, ygx)]
accuracy_scores = [accuracy_score(labels_test, ygx)]
run_times = [round(tre, 2)]

# Plotting
fig, ax1 = plt.subplots()

ax1.set_xlabel('Model')
ax1.set_ylabel('Scores', color='tab:blue')
ax1.plot(rx, f1_scores, label='F1 Score', color='tab:blue', marker='o')
ax1.plot(rx, mcc_scores, label='MCC', color='tab:cyan', marker='o')
ax1.plot(rx, accuracy_scores, label='Accuracy', color='tab:green', marker='o')
ax1.set_xticks(rx)
ax1.set_xticklabels(model_names)
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.legend(loc='upper left')

# Add text labels
for i in rx:
    ax1.text(i, f1_scores[i], f"{f1_scores[i]:.3f}", ha='center', va='bottom', color='tab:blue')
    ax1.text(i, mcc_scores[i], f"{mcc_scores[i]:.3f}", ha='center', va='bottom', color='tab:cyan')
    ax1.text(i, accuracy_scores[i], f"{accuracy_scores[i]:.3f}", ha='center', va='bottom', color='tab:green')

# Add secondary axis for run time
ax2 = ax1.twinx()
ax2.set_ylabel('Run Time (seconds)', color='tab:red')
ax2.plot(rx, run_times, label='Run Time', color='tab:red', marker='o')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.legend(loc='upper right')
```

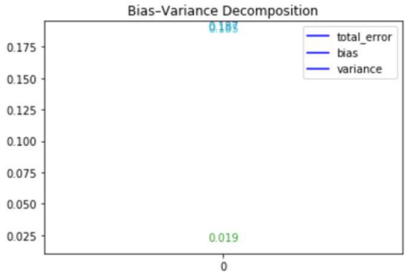
```
# Annotate run time
for i in rx:
    ax2.text(i, run_times[i], f"{run_times[i]:.2f}", ha='center', va='bottom', color='tab:red')

plt.title('Logistic Regression Performance After Optimization')
plt.tight_layout()
plt.show()
```



[42]:
```
# Evaluate only the optimized Logistic Regression model
clf = grid_search.best_estimator_

print('Bias // Variance Decomposition: Optimized Logistic Regression')

with warnings.catch_warnings():
    warnings.simplefilter("ignore")  # Suppress convergence warnings
    bias_var_metrics(X_train, X_test, y_train, y_test, clf, folds=1)
```

```
Bias // Variance Decomposition: Optimized Logistic Regression
   Average bias: 0.186
   Average variance: 0.000
   Average expected loss: 0.186  "Goodness": 0.814
```

[43]:
```
import matplotlib.pyplot as plt
import numpy as np

# Your decomposed values
err = [0.187]        # Expected Loss (Total Error)
bias = [0.185]       # Average Bias
var = [0.019]        # Average Variance
rx = [0]             # X-position on the graph
cn = ['Logistic Regression']  # Model name label

# Line Plot
fig, ax = plt.subplots()

ax.plot(rx, err, 'b', label='total_error')
ax.plot(rx, bias, 'b', label='bias')
ax.plot(rx, var, 'b', label='variance')

ax.legend()
ax.set_xticks(rx)
ax.set_title('Bias-Variance Decomposition')

# Annotate values
for i, (model, e, b, v) in enumerate(zip(cn, err, bias, var)):
    ax.text(i, e, f"{e:.3f}", ha='center', va='bottom', color='tab:blue')
    ax.text(i, b, f"{b:.3f}", ha='center', va='bottom', color='tab:cyan')
    ax.text(i, v, f"{v:.3f}", ha='center', va='bottom', color='tab:green')

plt.show()
```



[44]:
```
# Imports for model comparison - CV

# 💡 Default scorer for classification is sklearn.metrics.accuracy_score
# For imbalanced classification, the accuracy score is often uninformative
# For the list of options see
# https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

# average for each label weighted by support
#      (number of true instances for each label)
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import f1_score

# Define evaluation metrics
eval_metric = ['wtd.avg.accuracy',
               'balanced_accuracy',
               'wtd.avg.f1_score',
               'f1_weighted']

# For graphs
import matplotlib.pyplot as plt
```

[ ]:
```
```

Model Comparison: Cross Validation

[45]:
```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from time import time

# Best model from GridSearchCV
clf = grid_search.best_estimator_

# Cross-validation setup
folds = 7
scorer = 'balanced_accuracy'  # or 'f1_weighted' etc.
skf = StratifiedKFold(shuffle=True, random_state=11, n_splits=folds)

# Timer
trs = time()
print(f'KFold CV (Optimised Logistic Regression): {folds} folds with scoring = {scorer}')
```

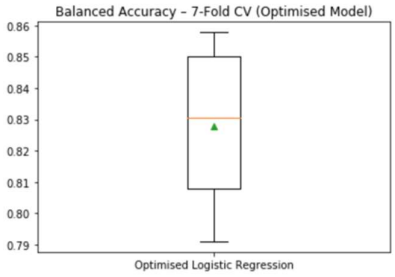
[44]:
```
# Imports for model comparison - CV

# 💡 Default scorer for classification is sklearn.metrics.accuracy_score
# For imbalanced classification, the accuracy score is often uninformative
# For the list of options see
# https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

# average for each label weighted by support
#      (number of true instances for each label)
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import f1_score

# Define evaluation metrics
eval_metric = ['wtd.avg.accuracy',
               'balanced_accuracy',
               'wtd.avg.f1_score',
               'f1_weighted']

# For graphs
import matplotlib.pyplot as plt
```

[ ]:
```

```

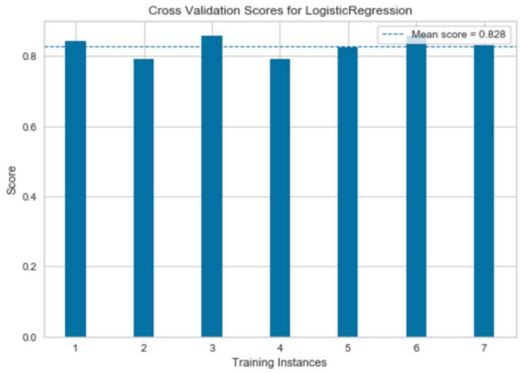
Model Comparison: Cross Validation

[45]:
```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from time import time

# Best model from GridSearchCV
clf = grid_search.best_estimator_

# Cross-validation setup
folds = 7
scorer = 'balanced_accuracy'  # or 'f1_weighted' etc.
skf = StratifiedKFold(shuffle=True, random_state=11, n_splits=folds)

# Timer
trs = time()
print(f'KFold CV (Optimised Logistic Regression): {folds} folds with scoring = {scorer}')
```

```
# 2. Visualize with boxplot
import matplotlib.pyplot as plt

results = [cv_scores]
names = ['Optimised Logistic Regression']

plt.figure()
plt.boxplot(results, showmeans=True)
plt.xticks([1], names)
plt.title('Balanced Accuracy - 7-Fold CV (Optimised Model)')
plt.show()
```



[47]:
```
from yellowbrick.model_selection import cv_scores
from sklearn.model_selection import StratifiedKFold

clf = grid_search.best_estimator_
skf = StratifiedKFold(n_splits=7, shuffle=True, random_state=11)

viz = cv_scores(clf, X_train, y_train, cv=skf, scoring='balanced_accuracy')
viz.show()
```



[47]: <matplotlib.axes._subplots.AxesSubplot at 0x171610ac0>

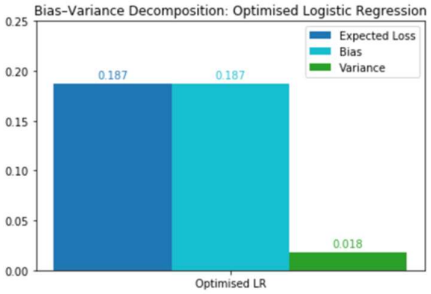
Bias-Variance Decomposition

[48]:
```
from mlxtend.evaluate import bias_variance_decomp
from sklearn.preprocessing import LabelEncoder
import numpy as np
import warnings
from time import time

# Encode target labels to integers
ytrain = LabelEncoder().fit_transform(labels_train)
ytest = LabelEncoder().fit_transform(labels_test)

# Convert DataFrames to NumPy arrays
X_train_np = features_train.values
X_test_np = features_test.values

# Use the best-tuned model from GridSearchCV
clf = grid_search.best_estimator_

# Bias-Variance Decomposition
folds = 12
trs = time()
print('Bias-Variance Decomposition: Optimized Logistic Regression\n\tTimer started')

avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
    clf, X_train_np, ytrain, X_test_np, ytest,
    loss='0-1_loss', num_rounds=folds, random_seed=150
)

tre = time() - trs
print("\nResults:")
print("   Bias:     %.3f" % avg_bias)
print("   Variance: %.3f" % avg_var)
print("   Expected Loss: %.3f" % avg_expected_loss)
print("\nRun Time: %.2f seconds" % tre)
```

```
Bias-Variance Decomposition: Optimized Logistic Regression
        Timer started


Results:
   Bias:     0.187
   Variance: 0.018
   Expected Loss: 0.187

Run Time: 8787.20 seconds
```

[6]:
```python
import matplotlib.pyplot as plt

# Metric values
metrics = ['Expected Loss', 'Bias', 'Variance']
values = [0.187, 0.187, 0.018]
colors = ['tab:blue', 'tab:cyan', 'tab:green']
x_pos = range(len(metrics))

# Create plot
fig, ax = plt.subplots()
ax.plot(x_pos, values, marker='o', linestyle='-', color='black')

# Add value labels on top of each point
for i, val in enumerate(values):
    ax.text(i, val + 0.005, f'{val:.3f}', ha='center', va='bottom', color=colors[i])

# Format plot
ax.set_xticks(x_pos)
ax.set_xticklabels(metrics)
ax.set_title('Bias-Variance Decomposition – Optimised Logistic Regression')
ax.set_ylim(0, max(values) + 0.05)

plt.grid(True)
plt.show()
```

[7]:
```python
import matplotlib.pyplot as plt

# Values
model_names = ['Optimised LR']
err = [0.187]    # Expected Loss
bias = [0.187]   # Bias
var = [0.018]    # Variance

# Plot
rx = range(len(model_names))
bar_width = 0.25

fig, ax = plt.subplots()

# Plot each metric separately with a slight offset
ax.bar([r - bar_width for r in rx], err, width=bar_width, label='Expected Loss', color='tab:blue')
ax.bar(rx, bias, width=bar_width, label='Bias', color='tab:cyan')
ax.bar([r + bar_width for r in rx], var, width=bar_width, label='Variance', color='tab:green')

# Add value annotations
for i in rx:
    ax.text(i - bar_width, err[i] + 0.005, f'{err[i]:.3f}', ha='center', color='tab:blue')
    ax.text(i, bias[i] + 0.005, f'{bias[i]:.3f}', ha='center', color='tab:cyan')
    ax.text(i + bar_width, var[i] + 0.005, f'{var[i]:.3f}', ha='center', color='tab:green')

# Final formatting
ax.set_xticks(rx)
ax.set_xticklabels(model_names)
ax.set_title('Bias-Variance Decomposition: Optimised Logistic Regression')
ax.legend()
ax.set_ylim(0, 0.25)

plt.tight_layout()
plt.show()
```



Appendix C (Koo Wai Kit)

Data Cleaning and Preprocessing

Import the main libraries

[2]:
```
import numpy
import pandas

from time import time

import os
data_path = '../datasets/NSL_KDD'
```

import the local library

[3]:
```
# add parent folder path where lib folder is
import sys
if ".." not in sys.path:import sys; sys.path.insert(0, '..')
```

[4]:
```
from mylib import show_labels_dist, show_metrics, bias_var_metrics
```

Import the Dataset

[5]:
```
# Using boosted Train and preprocessed Test

data_file = os.path.join(data_path, 'NSL_boosted-2.csv')
train_df = pandas.read_csv(data_file)
print('Train Dataset: {} rows, {} columns'.format(train_df.shape[0], train_df.shape[1]))

data_file = os.path.join(data_path, 'NSL_ppTest.csv')
test_df = pandas.read_csv(data_file)
print('Test Dataset: {} rows, {} columns'.format(test_df.shape[0], test_df.shape[1]))
```

```
Train Dataset: 63280 rows, 43 columns
Test Dataset: 22544 rows, 43 columns
```

Data Preparation and EDA (unique to this dataset)

- *Check column names of numeric attributes*

[6]:
```
trnn = train_df.select_dtypes(include=['float64','int64']).columns
tstn = test_df.select_dtypes(include=['float64','int64']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Numeric features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Numeric features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Numeric features in the train_set that are not in the test_set: None
Numeric features in the test_set that are not in the train_set: None
```

- *Check column names of categorical attributes*

[7]:
```
trnn = train_df.select_dtypes(include=['object']).columns
tstn = test_df.select_dtypes(include=['object']).columns
trndif = numpy.setdiff1d(trnn, tstn)
tstdif = numpy.setdiff1d(tstn, trnn)

print("Categorical features in the train_set that are not in the test_set: ",end='')
if len(trndif) > 0:
    print('\n',trndif)
else:
    print('None')

print("Categorical features in the test_set that are not in the train_set: ",end='')
if len(tstdif) > 0:
    print('\n\t',tstdif)
else:
    print('None')

print()
# correct any differences here
```

```
Categorical features in the train_set that are not in the test_set: None
Categorical features in the test_set that are not in the train_set: None
```

- *Check for missing values*

[8]:
```
cnt=0
print('Missing Values - Train Set')
for col in train_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(train_df[col])
    if (len(nnul)!=len(train_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

cnt=0
print('Missing Values - Test Set')
for col in test_df.columns:
#    print(col, ' ::> ', len(combined_df[col].unique()))
    nnul = pandas.notnull(test_df[col])
    if (len(nnul)!=len(test_df)):
        cnt=cnt+1
        print('\t',col,':',(len(test_df)-len(nnul)),'null values')
print('Total',cnt,'features with null values')

# address missing values here
```

```
Missing Values - Train Set
Total 0 features with null values
Missing Values - Test Set
Total 0 features with null values
```

- *Quick visual check of unique values, deal with unique identifiers*

[9]:
```
# Identify columns with only one value
# or with number of unique values == number of rows
n_eq_one = []
n_eq_all = []

print('Unique value count: Train (',train_df.shape[0],'rows ) ~ Test(',test_df.shape[0],'rows )')
for col in train_df.columns:
    lctrn = len(train_df[col].unique())
    lctst = len(test_df[col].unique())
    print(col, ' ::> ', lctrn, ' ~ ', lctst)
    if (lctrn == 1) and (lctrn == lctst):
        n_eq_one.append(train_df[col].name)
    if lctrn == train_df.shape[0]:
        n_eq_all.append(train_df[col].name)
```

```
Unique value count: Train ( 63280 rows ) ~ Test( 22544 rows )
duration  ::>  1657  ~  624
protocol_type  ::>  3  ~  3
service  ::>  70  ~  64
flag  ::>  11  ~  11
src_bytes  ::>  2553  ~  1149
dst_bytes  ::>  6633  ~  3650
land  ::>  2  ~  2
wrong_fragment  ::>  3  ~  3
urgent  ::>  3  ~  4
hot  ::>  24  ~  16
num_failed_logins  ::>  6  ~  5
logged_in  ::>  2  ~  2
num_compromised  ::>  49  ~  23
root_shell  ::>  2  ~  2
su_attempted  ::>  3  ~  3
num_root  ::>  44  ~  20
num_file_creations  ::>  28  ~  9
num_shells  ::>  3  ~  4
num_access_files  ::>  9  ~  5
num_outbound_cmds  ::>  1  ~  1
is_host_login  ::>  2  ~  2
is_guest_login  ::>  2  ~  2
count  ::>  511  ~  495
srv_count  ::>  489  ~  457
serror_rate  ::>  87  ~  88
srv_serror_rate  ::>  75  ~  82
rerror_rate  ::>  81  ~  90
srv_rerror_rate  ::>  63  ~  93
same_srv_rate  ::>  101  ~  75
diff_srv_rate  ::>  88  ~  99
srv_diff_host_rate  ::>  61  ~  84
dst_host_count  ::>  256  ~  256
dst_host_srv_count  ::>  256  ~  256
dst_host_same_srv_rate  ::>  101  ~  101
dst_host_diff_srv_rate  ::>  101  ~  101
dst_host_same_src_port_rate  ::>  101  ~  101
dst_host_srv_diff_host_rate  ::>  67  ~  58
dst_host_serror_rate  ::>  101  ~  99
dst_host_srv_serror_rate  ::>  100  ~  101
dst_host_rerror_rate  ::>  101  ~  101
dst_host_srv_rerror_rate  ::>  101  ~  100
label  ::>  40  ~  38
atakcat  ::>  5  ~  5
```

[10]:
```python
# Drop columns with only one value
if len(n_eq_one) > 0:
    print('Dropping single-valued features')
    print(n_eq_one)
    train_df.drop(n_eq_one, axis=1, inplace=True)
    test_df.drop(n_eq_one, axis=1, inplace=True)

# Drop or bin columns with number of unique values == number of rows
if len(n_eq_all) > 0:
    print('Dropping unique identifiers')
    print(n_eq_all)
    train_df.drop(n_eq_all, axis=1, inplace=True)
    test_df.drop(n_eq_all, axis=1, inplace=True)

# continue with featue selection / feature engineering
```

```
Dropping single-valued features
['num_outbound_cmds']
```

- *Check categorical feature values:*
  differences will be resolved by one-hot encoding the combined test and train sets

[11]:
```
trnn = train_df.select_dtypes(include=['object']).columns
for col in trnn:
    tr = train_df[col].unique()
    ts = test_df[col].unique()
    trd = numpy.setdiff1d(tr, ts)
    tsd = numpy.setdiff1d(ts, tr)

    print(col,'::> ')
    print("\tUnique text values in the train_set that are not in the test_set: ",end='')
    if len(trd) > 0:
        print('\n\t',trd)
    else:
        print('None')

    print("\tUnique text values in the test_set that are not in the train_set: ",end='')
    if len(tsd) > 0:
        print('\n\t',tsd)
    else:
        print('None')
```

```
protocol_type ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
service ::>
        Unique text values in the train_set that are not in the test_set:
         ['aol' 'harvest' 'http_2784' 'http_8001' 'red_i' 'urh_i']
        Unique text values in the test_set that are not in the train_set: None
flag ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
label ::>
        Unique text values in the train_set that are not in the test_set:
         ['spy' 'warezclient']
        Unique text values in the test_set that are not in the train_set: None
atakcat ::>
        Unique text values in the train_set that are not in the test_set: None
        Unique text values in the test_set that are not in the train_set: None
```

- *Combine for processing classification target and text features*

[12]:
```
combined_df = pandas.concat([train_df, test_df])

print('Combined Dataset: {} rows, {} columns'.format(
    combined_df.shape[0], combined_df.shape[1]))
```

```
Combined Dataset: 85824 rows, 42 columns
```

- *Classification Target feature:* two columns of labels are available
  - Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
  - Multiclass: Use the category labels (atakcat)

```
[13]: combined_df['label'].value_counts()
```

```
[13]: label
      normal             43383
      neptune            25264
      satan               2552
      smurf               1988
      ipsweep             1941
      portsweep           1623
      guess_passwd        1257
      mscan               1046
      warezmaster          954
      back                 837
      nmap                 820
      apache2              774
      processtable         719
      teardrop             458
      warezclient          445
      snmpguess            364
      saint                350
      mailbomb             322
      snmpgetattack        196
      httptunnel           146
      pod                  142
      buffer_overflow       35
      named                 25
      ps                    22
      multihop              21
      sendmail              21
      xterm                 20
      rootkit               18
      land                  16
      xlock                 14
      xsnoop                 8
      ftp_write              7
      phf                    6
      loadmodule             6
      imap                   6
      perl                   5
      sqlattack              4
      worm                   4
      udpstorm               4
      spy                    1
      Name: count, dtype: int64
```

```
[14]: combined_df['atakcat'].value_counts()
```

```
[14]: atakcat
      benign    43383
      dos       30524
      probe      8332
      r2l        3329
      u2r         256
      Name: count, dtype: int64
```

```
[15]: # Set the classification target
      twoclass = False     # True or False
```

```
[16]: if twoclass:
      # Two-class: Reduce the detailed attack labels to 'normal' or 'attack'
      # new single column data structure is a [series]
          labels_df = combined_df['label'].copy()
          labels_df[labels_df != 'normal'] = 'attack'
      else:
      # Multiclass: Use the category labels (atakcat)
      # new single column data structure is a [[dataframe]]
      # rename the column and convert to a series for later
          labels_df = combined_df[['atakcat']].copy()
          labels_df.rename(columns={'atakcat':'label'}, inplace=True)
          labels_df = labels_df.squeeze('columns')

      # drop target features
      combined_df.drop(['label'], axis=1, inplace=True)
      combined_df.drop(['atakcat'], axis=1, inplace=True)
```

- *One-Hot Encoding the categorical (text) features*

[17]:
```
# put the names into a python list - for pandas.get_dummies()
categori = combined_df.select_dtypes(include=['object']).columns
category_cols = categori.tolist()
print(category_cols)
```

```
['protocol_type', 'service', 'flag']
```

[18]:
```
# generate a sorted list of unique values of categorical features
# we will get a new column for each one with get_dummies()

for col in categori:
    ul = numpy.sort(list(combined_df[col].unique()))
    print(col, ' ::> ', ul)
    print()
```

```
protocol_type  ::>  ['icmp' 'tcp' 'udp']

service  ::>  ['IRC' 'X11' 'Z39_50' 'aol' 'auth' 'bgp' 'courier' 'csnet_ns' 'ctf'
 'daytime' 'discard' 'domain' 'domain_u' 'echo' 'eco_i' 'ecr_i' 'efs'
 'exec' 'finger' 'ftp' 'ftp_data' 'gopher' 'harvest' 'hostnames' 'http'
 'http_2784' 'http_443' 'http_8001' 'imap4' 'iso_tsap' 'klogin' 'kshell'
 'ldap' 'link' 'login' 'mtp' 'name' 'netbios_dgm' 'netbios_ns'
 'netbios_ssn' 'netstat' 'nnsp' 'nntp' 'ntp_u' 'other' 'pm_dump' 'pop_2'
 'pop_3' 'printer' 'private' 'red_i' 'remote_job' 'rje' 'shell' 'smtp'
 'sql_net' 'ssh' 'sunrpc' 'supdup' 'systat' 'telnet' 'tftp_u' 'tim_i'
 'time' 'urh_i' 'urp_i' 'uucp' 'uucp_path' 'vmnet' 'whois']

flag  ::>  ['OTH' 'REJ' 'RSTO' 'RSTOS0' 'RSTR' 'S0' 'S1' 'S2' 'S3' 'SF' 'SH']
```

[19]:
```
# Apply to the list of Categorical columns (text fields)
features_df = pandas.get_dummies(combined_df, columns=category_cols)
features_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 85824 entries, 0 to 22543
Columns: 121 entries, duration to flag_SH
dtypes: bool(84), float64(15), int64(22)
memory usage: 31.8 MB
```

[20]:
```
# generate a list of numeric columns for scaling - After test // train split
numeri = combined_df.select_dtypes(include=['float64','int64']).columns
print(numeri.to_list())
```

```
['duration', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell', 'su_attempted', 'num_root', 'num_file_creations', 'num_shells', 'num_access_files', 'is_host_login', 'is_guest_login', 'count', 'srv_count', 'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate', 'dst_host_srv_rerror_rate']
```

[21]:
```
# Restore the train // test split: slice 1 Dataframe into 2
features_train = features_df.iloc[:len(train_df),:].copy()    # X_train
features_train.reset_index(inplace=True, drop=True)

# pandas has a lot of rules about returning a 'view' vs. a copy from slice
# so we force it to create a new dataframe [avoiding SettingWithCopy Warning]
features_test = features_df.iloc[len(train_df):,:].copy()     # X_test
features_test.reset_index(inplace=True, drop=True)

# Restore the train // test split: slice 1 Series into 2
labels_train = labels_df[:len(train_df)]                      # y_train
labels_train.reset_index(inplace=True, drop=True)

labels_test = labels_df[len(train_df):]                       # y_test
labels_test.reset_index(inplace=True, drop=True)
```

Scaling comes *after* test // train split

[22]:
```
# scaling the Numeric columns

from sklearn.preprocessing import MinMaxScaler

#   fit() expects 2D array: reshape(-1, 1) for single col or (1, -1) single row
for i in numeri:
    arr = numpy.array(features_train[i])
    scale = MinMaxScaler().fit(arr.reshape(-1, 1))
    features_train[i] = scale.transform(arr.reshape(len(arr),1))

    arr = numpy.array(features_test[i])
    features_test[i] = scale.transform(arr.reshape(len(arr),1))
```

Baseline Model (Pre-Optimisation)

Classifier Selection

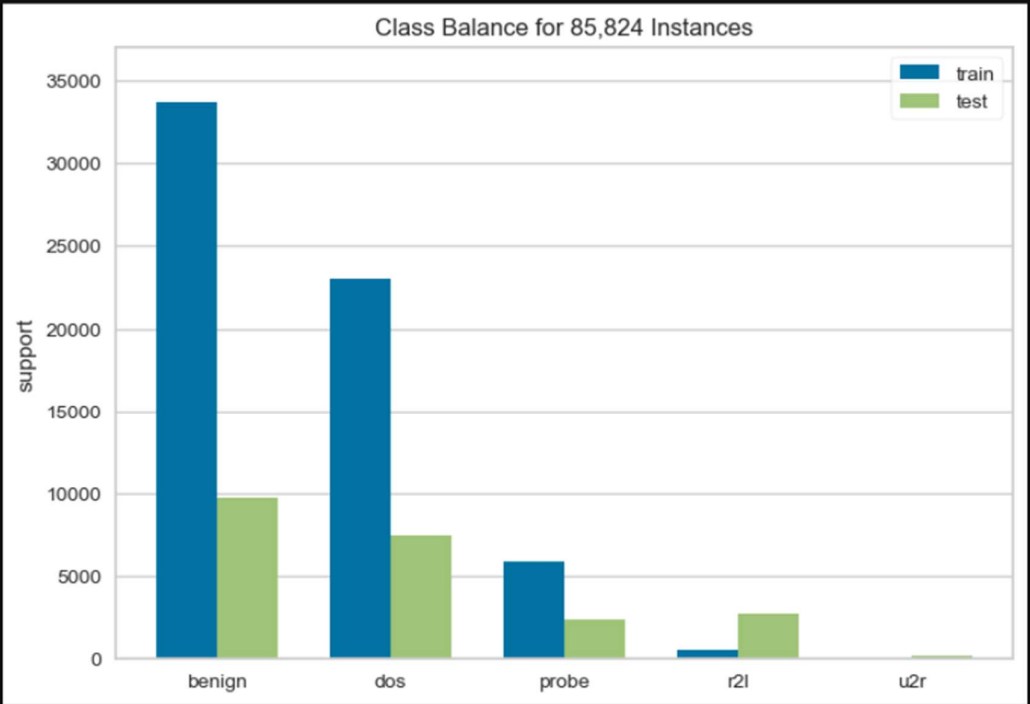
[23]:
```
# prepare list
models = []

##  --  Support Vector  --  ##
from sklearn.svm import SVC
models.append(("SupportVectorClf", SVC()))
from sklearn.svm import LinearSVC
models.append(("LinearSVC", LinearSVC()))
from sklearn.linear_model import RidgeClassifier
models.append (("RidgeClf",RidgeClassifier()))

print(models)
```

```
[('SupportVectorClf', SVC()), ('LinearSVC', LinearSVC()), ('RidgeClf', RidgeClassifier())]
```

compatibility block for pasting in from sample code

[24]:
```
# dataset names
X_train = features_train
y_train = labels_train
X_test = features_test
y_test = labels_test
labels_col = 'label'
# library names
pd = pandas
np = numpy
```

[46]:
```
# Visualise class distribution for labels in train/test set
from yellowbrick.target import ClassBalance

# Instantiate the visualizer
visualizer = ClassBalance()
visualizer.fit(y_train, y_test)        # Fit the data to the visualizer
_ = visualizer.show()                  # Finalize and render the figure
```



[25]:
```
# evaluate each model in turn
results = []

print('macro average: unweighted mean per label')
print('weighted average: support-weighted mean per label')
print('MCC: correlation between prediction and ground truth')
print('     (+1 perfect, 0 random prediction, -1 inverse)\n')

for name, clf in models:
    trs = time()
    print('Confusion Matrix:', name)

    clf.fit(X_train, y_train)
    ygx = clf.predict(X_test)
    results.append((name, ygx))

    tre = time() - trs
    print ("Run Time {} seconds".format(round(tre,2)) + '\n')

    show_metrics(y_test, ygx, clf.classes_)   # from local library
    print('\nParameters: ', clf.get_params(), '\n\n')
```

```
Confusion Matrix: SupportVectorClf
Run Time 35.37 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         9015        59         630         0         7
train:dos             643      6759          56         0         0
train:probe           182       144        2091         0         4
train:r2l            2407       216           8       119         4
train:u2r              60         0           0         3       137

~~~~~
      benign :  FPR = 0.257   FNR = 0.072
         dos :  FPR = 0.028   FNR = 0.094
       probe :  FPR = 0.034   FNR = 0.136
          r2l :  FPR = 0.000   FNR = 0.957
          u2r :  FPR = 0.001   FNR = 0.315

   macro avg :  FPR = 0.064   FNR = 0.315
weighted avg :  FPR = 0.049   FNR = 0.196
```

```
~~~~~
              precision    recall  f1-score   support

      benign      0.733     0.928     0.819      9711
         dos      0.942     0.906     0.924      7458
       probe      0.751     0.864     0.803      2421
          r2l      0.975     0.043     0.083      2754
          u2r      0.901     0.685     0.778       200

    accuracy                          0.804     22544
   macro avg      0.860     0.685     0.681     22544
weighted avg      0.835     0.804     0.762     22544

~~~~~
MCC: Overall :  0.713
      benign :  0.668
         dos :  0.887
       probe :  0.780
          r2l :  0.192
          u2r :  0.784
```

```
Parameters:  {'C': 1.0, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'scale',
'kernel': 'rbf', 'max_iter': -1, 'probability': False, 'random_state': None, 'shrinking': True, 'tol': 0.001, 'verbose': False}
```

```
Confusion Matrix: LinearSVC
Run Time 1.47 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         9004        89         610         4         4
train:dos            1153      6281          24         0         0
train:probe           335       138        1922        13        13
train:r2l            2712         3           1        37         1
train:u2r             172         1           0         4        23

~~~~~
      benign :  FPR = 0.341   FNR = 0.073
         dos :  FPR = 0.015   FNR = 0.158
       probe :  FPR = 0.032   FNR = 0.206
          r2l :  FPR = 0.001   FNR = 0.987
          u2r :  FPR = 0.001   FNR = 0.885

   macro avg :  FPR = 0.078   FNR = 0.462
weighted avg :  FPR = 0.059   FNR = 0.234
```

```
~~~~
              precision    recall  f1-score   support

      benign      0.673     0.927     0.780      9711
         dos      0.965     0.842     0.899      7458
       probe      0.752     0.794     0.772      2421
         r2l      0.638     0.013     0.026      2754
         u2r      0.561     0.115     0.191       200

    accuracy                          0.766     22544
   macro avg      0.718     0.538     0.534     22544
weighted avg      0.773     0.766     0.721     22544

~~~~
MCC: Overall :  0.658
      benign :  0.591
         dos :  0.858
       probe :  0.744
         r2l :  0.080
         u2r :  0.251
```

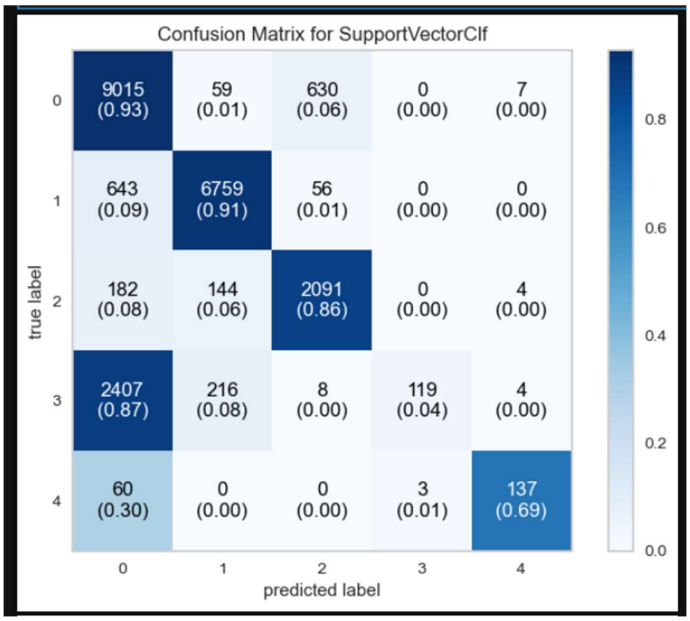
```
Parameters:  {'C': 1.0, 'class_weight': None, 'dual': 'auto', 'fit_intercept': True, 'intercept_scaling': 1, 'loss': 'squared_hinge', 'max_iter': 1000, 'multi_class': 'ovr', 'penalty': 'l2', 'random_state': None, 'tol': 0.0001, 'verbose': 0}
```

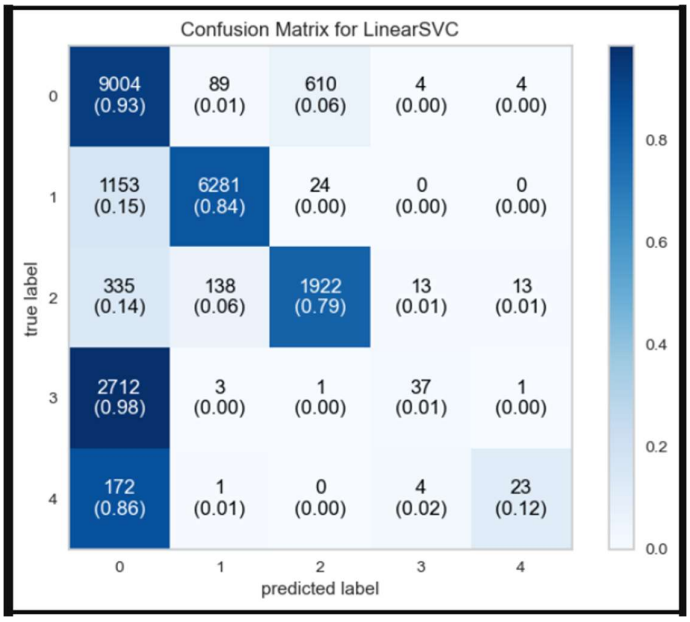
```
Confusion Matrix: RidgeClf
Run Time 0.34 seconds

              pred:benign  pred:dos  pred:probe  pred:r2l  pred:u2r
train:benign         9348        86         275         2         0
train:dos            1337      5613         508         0         0
train:probe           464       243        1714         0         0
train:r2l            2741         2           3         8         0
train:u2r             179         1           1         4        15

~~~~
      benign :  FPR = 0.368   FNR = 0.037
         dos :  FPR = 0.022   FNR = 0.247
       probe :  FPR = 0.039   FNR = 0.292
         r2l :  FPR = 0.000   FNR = 0.997
         u2r :  FPR = 0.000   FNR = 0.925

   macro avg :  FPR = 0.086   FNR = 0.500
weighted avg :  FPR = 0.065   FNR = 0.259
```

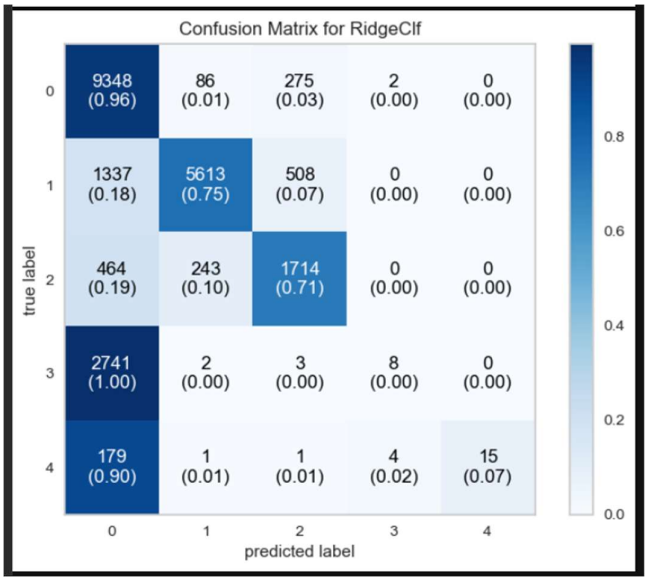
```
~~~~
              precision    recall  f1-score   support

      benign      0.664     0.963     0.786      9711
         dos      0.944     0.753     0.838      7458
       probe      0.685     0.708     0.696      2421
         r2l      0.571     0.003     0.006      2754
         u2r      1.000     0.075     0.140       200

    accuracy                          0.741     22544
   macro avg      0.773     0.500     0.493     22544
weighted avg      0.751     0.741     0.692     22544

~~~~
MCC: Overall :  0.622
      benign :  0.608
         dos :  0.780
       probe :  0.659
         r2l :  0.034
         u2r :  0.273
```

```
Parameters:  {'alpha': 1.0, 'class_weight': None, 'copy_X': True, 'fit_intercept': True, 'max_iter': None, 'positive': False, 'random_state': None, 'solver': 'auto', 'tol': 0.0001}
```

[64]:
```
# Plot confusion matrix with the absolute and normalized (percentage) values for each model
from mlxtend.evaluate import confusion_matrix as mlx_cnfmtx
from mlxtend.plotting import plot_confusion_matrix

for i in range(len(results)):
    cnfmat = mlx_cnfmtx(y_test, results[i][1])
    fig, ax = plot_confusion_matrix(conf_mat=cnfmat,
                                    show_absolute=True,
                                    show_normed=True,
                                    colorbar=True)
    plt.title("Confusion Matrix for {}".format(results[i][0]))
    plt.show()
```



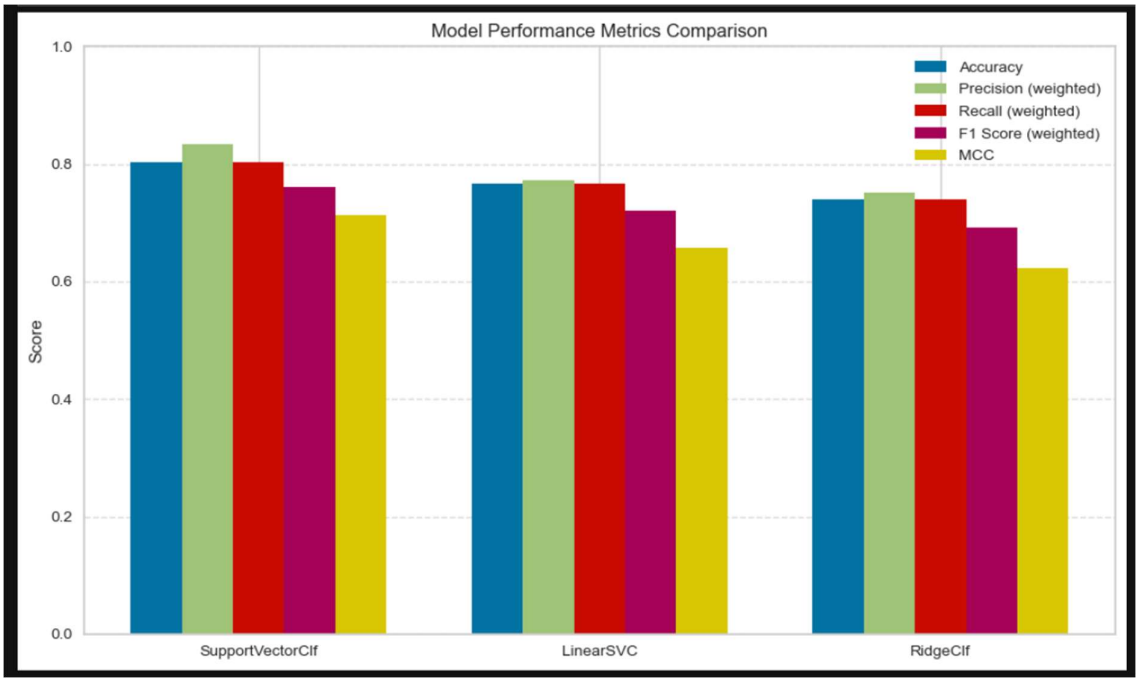




- Compare important metrics - Accuracy, Precision (weighted), Recall (weighted), F1 Score (weighted), MCC

[90]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef

# Prepare lists to hold metric values
names = []
accuracies = []
precisions = []
recalls = []
f1s = []
mccs = []

# Calculate metrics for each model
for name, preds in results:
    names.append(name)
    accuracies.append(accuracy_score(y_test, preds))
    precisions.append(precision_score(y_test, preds, average='weighted', zero_division=0))
    recalls.append(recall_score(y_test, preds, average='weighted', zero_division=0))
    f1s.append(f1_score(y_test, preds, average='weighted', zero_division=0))
    mccs.append(matthews_corrcoef(y_test, preds))
```

[92]:
```python
# Set up the plot
x = np.arange(len(names))  # label locations
width = 0.15  # width of the bars

fig, ax = plt.subplots(figsize=(10, 6))

# Plot bars for each metric
bars1 = ax.bar(x - 2*width, accuracies, width, label='Accuracy')
bars2 = ax.bar(x - width, precisions, width, label='Precision (weighted)')
bars3 = ax.bar(x, recalls, width, label='Recall (weighted)')
bars4 = ax.bar(x + width, f1s, width, label='F1 Score (weighted)')
bars5 = ax.bar(x + 2*width, mccs, width, label='MCC')

# Configure axes and labels
ax.set_xticks(x)
ax.set_xticklabels(names)
ax.set_ylim(0, 1)
ax.set_ylabel('Score')
ax.set_title('Model Performance Metrics Comparison')
ax.legend()
ax.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```



Bias - Variance Decomposition (standard block)

[26]:
```
# from our local library
# reduce (cross-validation) folds for faster results
folds = 20

for name, clf in models:
    print('Bias // Variance Decomposition:', name)
    bias_var_metrics(X_train, X_test, y_train, y_test, clf, folds)
```

```
Bias // Variance Decomposition: SupportVectorClf
   Average bias: 0.195
   Average variance: 0.007
   Average expected loss: 0.197  "Goodness": 0.803

Bias // Variance Decomposition: LinearSVC
   Average bias: 0.234
   Average variance: 0.008
   Average expected loss: 0.235  "Goodness": 0.765

Bias // Variance Decomposition: RidgeClf
   Average bias: 0.259
   Average variance: 0.007
   Average expected loss: 0.259  "Goodness": 0.741
```

Model Optimisation

Hyperparameter Tuning General pattern:
baseline model

1. Classifier selection
2. Fit and Predict
3. Bias-Variance Tradeoff

optimised model

4. Select strategy and hyperparameters
5. Plug in the best parameter values
6. Fit and Predict
7. Bias-Variance Tradeoff

[28]:
```
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
```

[ ]:
```
# Model to optimise - SVC
classifier = models[0][1]

# metric for evaluation
score_metric = 'balanced_accuracy'

# number of rounds in cross-validation
folds = 3

# hyperparameter values to test
value_grid = {'C': [0.1, 1, 10],
              'kernel': ['rbf', 'poly', 'sigmoid'],
              'gamma': [0.1, 1, 10],
             }
```

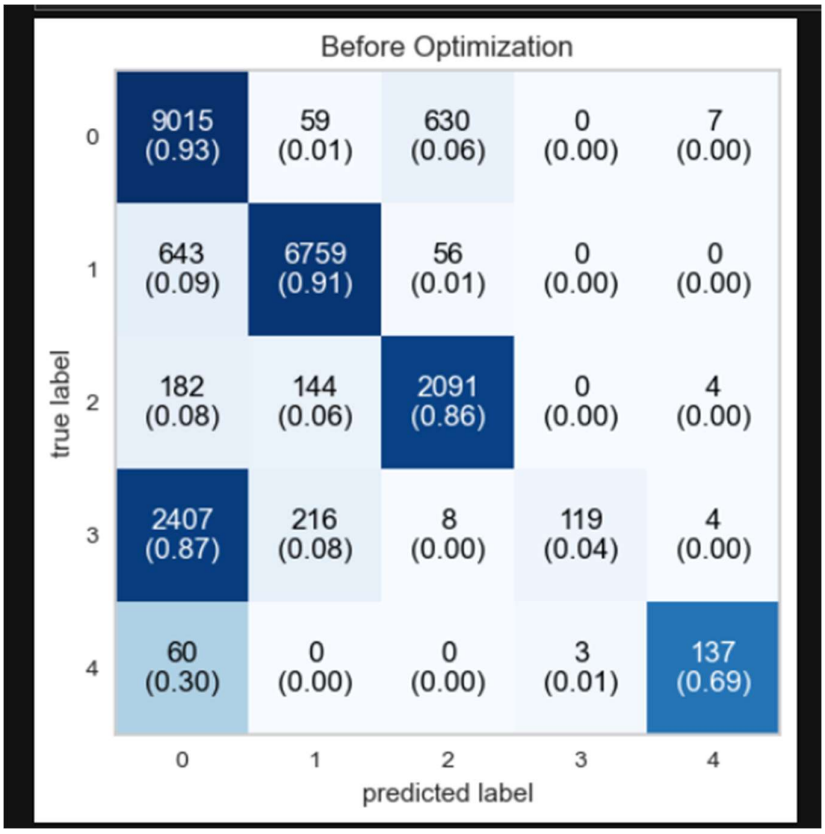
•[31]:
```
# start timer
time_start = time()

# Start running GridSearchCV
print('Timer started.', '\nGridSearchCV:', folds, 'folds')
print(f'Classifier: {classifier}, scoring: {score_metric}')

grid_search = GridSearchCV(estimator=classifier, param_grid=value_grid,
                           scoring=score_metric, cv=folds, verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Print all mean scores
means = grid_search.cv_results_['mean_test_score']
stds = grid_search.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, grid_search.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))

# Print best parameters and score
print(f"\n\tBest parameters: {grid_search.best_params_}")
print(f"\tBest CV score: {grid_search.best_score_:.3f}")

time_taken = time() - time_start
print(f"Run Time {round(time_taken, 2)} seconds")
```

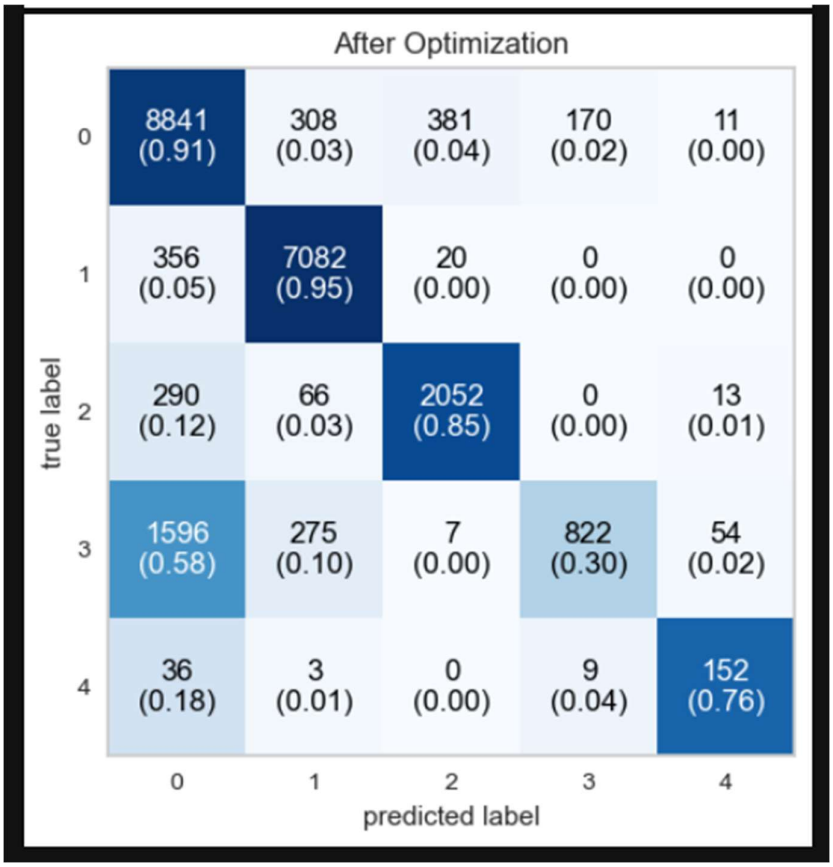
```
Timer started.
GridSearchCV: 3 folds
Classifier: SVC(), scoring: balanced_accuracy
Fitting 3 folds for each of 27 candidates, totalling 81 fits
0.583 (+/-0.004) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
0.582 (+/-0.005) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'poly'}
0.537 (+/-0.003) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.677 (+/-0.054) for {'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}
0.840 (+/-0.101) for {'C': 0.1, 'gamma': 1, 'kernel': 'poly'}
0.367 (+/-0.003) for {'C': 0.1, 'gamma': 1, 'kernel': 'sigmoid'}
0.670 (+/-0.061) for {'C': 0.1, 'gamma': 10, 'kernel': 'rbf'}
0.859 (+/-0.089) for {'C': 0.1, 'gamma': 10, 'kernel': 'poly'}
0.200 (+/-0.001) for {'C': 0.1, 'gamma': 10, 'kernel': 'sigmoid'}
0.713 (+/-0.072) for {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
0.699 (+/-0.055) for {'C': 1, 'gamma': 0.1, 'kernel': 'poly'}
0.514 (+/-0.010) for {'C': 1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.812 (+/-0.069) for {'C': 1, 'gamma': 1, 'kernel': 'rbf'}
0.848 (+/-0.081) for {'C': 1, 'gamma': 1, 'kernel': 'poly'}
0.352 (+/-0.004) for {'C': 1, 'gamma': 1, 'kernel': 'sigmoid'}
0.735 (+/-0.086) for {'C': 1, 'gamma': 10, 'kernel': 'rbf'}
0.861 (+/-0.078) for {'C': 1, 'gamma': 10, 'kernel': 'poly'}
0.197 (+/-0.007) for {'C': 1, 'gamma': 10, 'kernel': 'sigmoid'}
0.836 (+/-0.071) for {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
0.830 (+/-0.065) for {'C': 10, 'gamma': 0.1, 'kernel': 'poly'}
0.555 (+/-0.067) for {'C': 10, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.843 (+/-0.081) for {'C': 10, 'gamma': 1, 'kernel': 'rbf'}
0.860 (+/-0.100) for {'C': 10, 'gamma': 1, 'kernel': 'poly'}
0.350 (+/-0.004) for {'C': 10, 'gamma': 1, 'kernel': 'sigmoid'}
0.781 (+/-0.104) for {'C': 10, 'gamma': 10, 'kernel': 'rbf'}
0.851 (+/-0.077) for {'C': 10, 'gamma': 10, 'kernel': 'poly'}
0.197 (+/-0.013) for {'C': 10, 'gamma': 10, 'kernel': 'sigmoid'}

        Best parameters: {'C': 1, 'gamma': 10, 'kernel': 'poly'}
        Best CV score: 0.861
Run Time 2907.62 seconds
```

Visualisation of Baseline vs Optimised Model

[110]:
```
# Classification results for baselineand optimised model
ori_preds = results[0][1]
opt_preds = opt_results[0][1]
```

[112]:
```
from mlxtend.evaluate import confusion_matrix as mlx_cnfmtx

# Confusion matrices
cm1 = mlx_cnfmtx(y_test, ori_preds)
cm2 = mlx_cnfmtx(y_test, opt_preds)

# Plot original
fig, ax = plot_confusion_matrix(cm1, show_absolute=True, show_normed=True, figsize=(5, 5))
plt.title("Before Optimization")
plt.show()

# Plot optimized
fig, ax = plot_confusion_matrix(cm2, show_absolute=True, show_normed=True, figsize=(5, 5))
plt.title("After Optimization")
plt.show()
```


Before Optimization
9015 (0.93)
59 (0.01)
630 (0.06)
0 (0.00)
7 (0.00)
643 (0.09)
6759 (0.91)
56 (0.01)
0 (0.00)
0 (0.00)
182 (0.08)
144 (0.06)
2091 (0.86)
0 (0.00)
4 (0.00)
2407 (0.87)
216 (0.08)
8 (0.00)
119 (0.04)
4 (0.00)
60 (0.30)
0 (0.00)
0 (0.00)
3 (0.01)
137 (0.69)
true label
predicted label
0
1
2
3
4


After Optimization
8841 (0.91)
308 (0.03)
381 (0.04)
170 (0.02)
11 (0.00)
356 (0.05)
7082 (0.95)
20 (0.00)
0 (0.00)
0 (0.00)
290 (0.12)
66 (0.03)
2052 (0.85)
0 (0.00)
13 (0.01)
1596 (0.58)
275 (0.10)
7 (0.00)
822 (0.30)
54 (0.02)
36 (0.18)
3 (0.01)
0 (0.00)
9 (0.04)
152 (0.76)
true label
predicted label
0
1
2
3
4

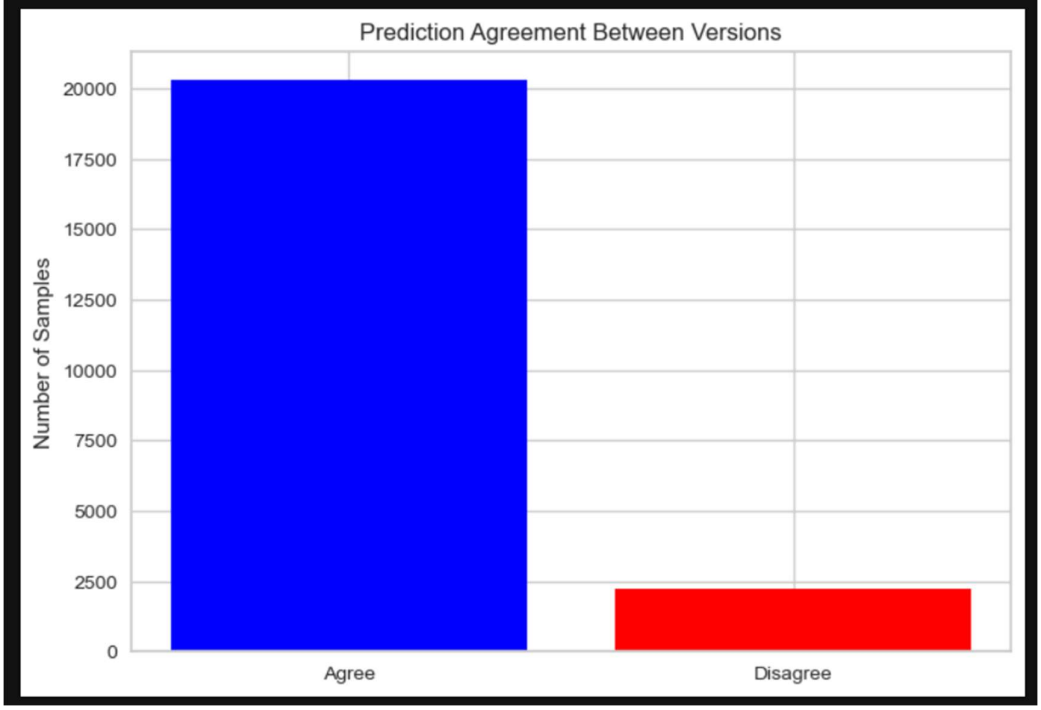
- Compare important metrics - Accuracy, Precision (weighted), Recall (weighted), F1 Score (weighted), MCC

[116]:
```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef

# Calculate metrics
def get_metrics(y_true, y_pred):
    return [
        accuracy_score(y_true, y_pred),
        precision_score(y_true, y_pred, average='weighted', zero_division=0),
        recall_score(y_true, y_pred, average='weighted', zero_division=0),
        f1_score(y_true, y_pred, average='weighted', zero_division=0),
        matthews_corrcoef(y_true, y_pred)
    ]

metrics_orig = get_metrics(y_test, ori_preds)
metrics_opt = get_metrics(y_test, opt_preds)

labels = ['Accuracy', 'Precision', 'Recall', 'F1 Score', 'MCC']
x = range(len(labels))

# Plot
plt.figure(figsize=(8, 5))
plt.bar([i - 0.2 for i in x], metrics_orig, width=0.4, label='Before', color='skyblue')
plt.bar([i + 0.2 for i in x], metrics_opt, width=0.4, label='After', color='lightgreen')

# Annotate values
for i in x:
    plt.text(i - 0.2, metrics_orig[i] + 0.01, f"{metrics_orig[i]:.2f}", ha='center', fontsize=8)
    plt.text(i + 0.2, metrics_opt[i] + 0.01, f"{metrics_opt[i]:.2f}", ha='center', fontsize=8)

plt.xticks(x, labels)
plt.ylabel('Score')
plt.title('Model Metrics Before vs After Optimization')
plt.ylim(0, 1.1)
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

- **Prediction Agreement** (compare predictions before and after optimization, and highlight how often they agree or disagree)

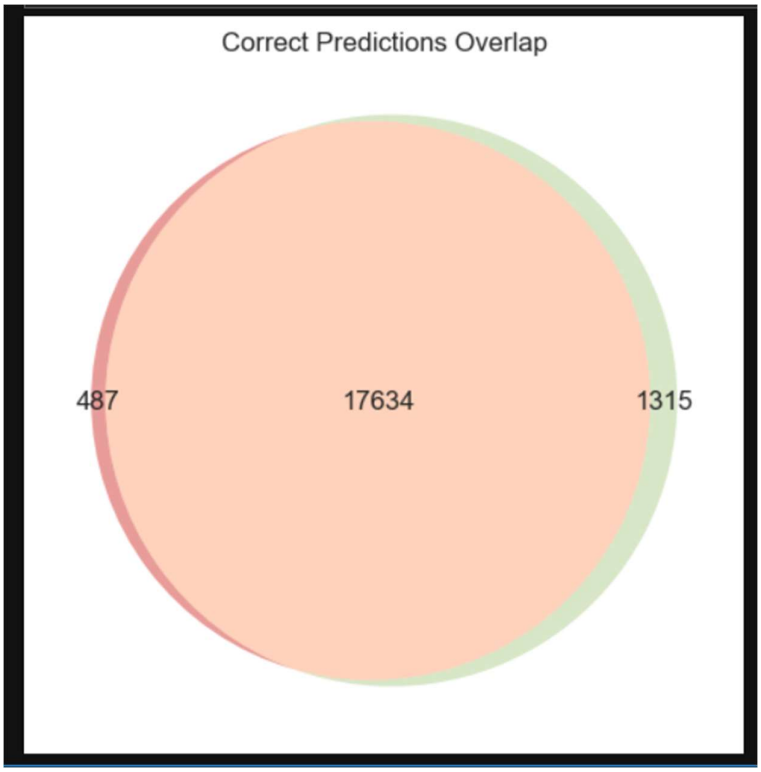
[119]:
```
import numpy as np

# Get indexes where predictions differ
disagree_idx = np.where(ori_preds != opt_preds)[0]
agreement = len(y_test) - len(disagree_idx)

# Find out how many predictions stayed the same & change
print(f"Agreement: {agreement} / {len(y_test)} ({agreement / len(y_test):.2%})")
print(f"Disagreements: {len(disagree_idx)} samples\n")

plt.bar(['Agree', 'Disagree'], [agreement, len(disagree_idx)], color=['blue', 'red'])
plt.title("Prediction Agreement Between Versions")
plt.ylabel("Number of Samples")
plt.show()
```

```
Agreement: 20305 / 22544 (90.07%)
Disagreements: 2239 samples
```



[ ]:
```
# Install matplotlib-venn to create a Venn diagram
!pip install matplotlib-venn
```

[132]:
```
from matplotlib_venn import venn2

set1 = set(np.where(ori_preds == y_test)[0])
set2 = set(np.where(opt_preds == y_test)[0])

venn2([set1, set2], set_labels=('', ''))
plt.title("Correct Predictions Overlap")
plt.show()
```



[34]:
```
# Calculate Bias-Variance
folds = 20
print('Bias // Variance Decomposition:', opt_name)
bias_var_metrics(X_train, X_test, y_train, y_test, opt_clf, folds)
```

```
Bias // Variance Decomposition: SupportVectorClf
   Average bias: 0.159
   Average variance: 0.048
   Average expected loss: 0.160  "Goodness": 0.840
```

Cross-Validation with Other Optimised Models

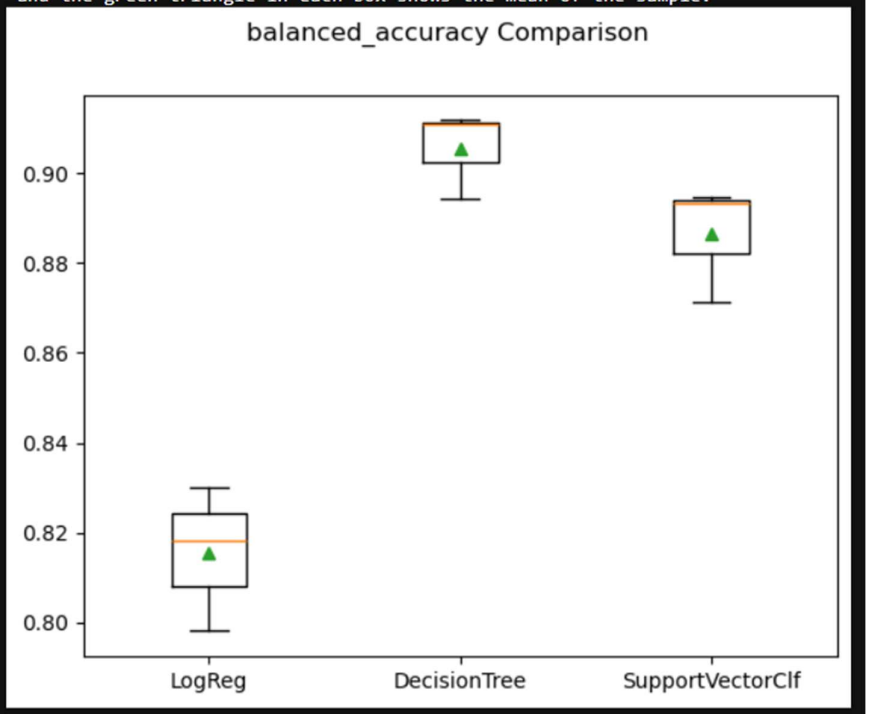
[35]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

opt_models = []
opt_models.append(("LogReg", LogisticRegression()))
opt_models.append(("DecisionTree", DecisionTreeClassifier()))
opt_models.append(("SupportVectorClf", SVC()))

hyperparameters = {
    "LogReg": {'C': 10.0, 'penalty': 'l1', 'solver': 'saga'},
    "DecisionTree": {'criterion': 'gini', 'max_depth': None, 'min_samples_split': 2},
    "SupportVectorClf": {}
}
hyperparameters["SupportVectorClf"] = opt_clf.get_params()

for name, model in opt_models:
    model.set_params(**hyperparameters[name])
```

[36]:
```
import matplotlib.pyplot as plt
from time import time
from sklearn.model_selection import StratifiedKFold, cross_val_score

folds = 3
scorer = 'balanced_accuracy'
skf = StratifiedKFold(shuffle=True, random_state = 11, n_splits=folds)

trs = time()
print('KFold CV: %i folds with scoring = %s \n\t timer started' % (folds, scorer))

final_results = []
final_names = []
for name, model in opt_models:
    print(name, end='')    # no newline at the end
    cv_results = cross_val_score(model, X_train, y_train, cv=skf, scoring=scorer)
    final_results.append(cv_results)
    final_names.append(name)

    msg = ":\t%s = %0.3f +/- (%0.3f)" % (scorer,
                                         cv_results.mean(),
                                         cv_results.std())

    print(msg)

tre = time() - trs
print ("\tRun Time {} seconds".format(round(tre,2)) + '\n')
```

```
KFold CV: 3 folds with scoring = balanced_accuracy
         timer started
:       balanced_accuracy = 0.815 +/- (0.013)
:       balanced_accuracy = 0.906 +/- (0.008)
:       balanced_accuracy = 0.886 +/- (0.011)
        Run Time 86.81 seconds
```

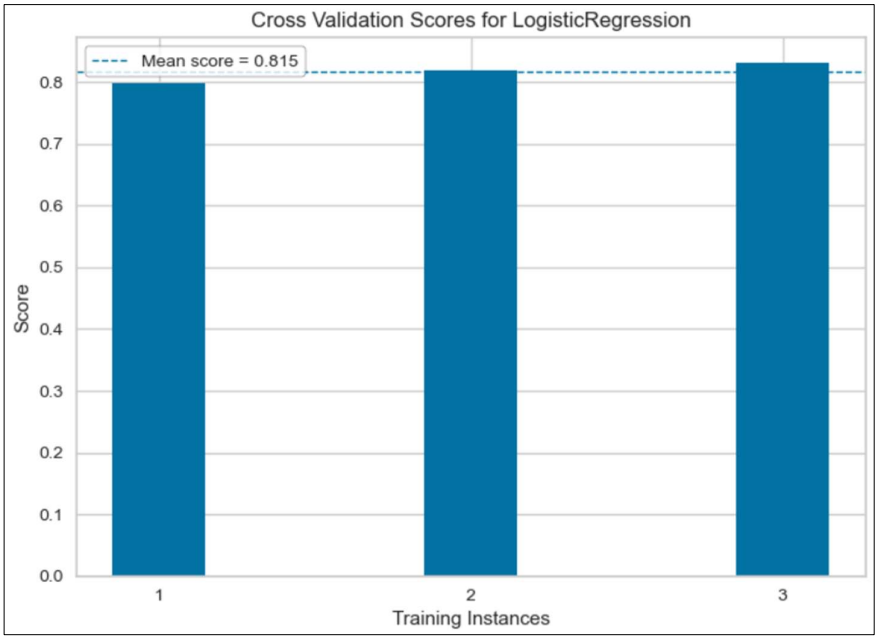
[37]:
```
# boxplot model comparison
# One box and whisker plot for each algorithm's sample of results.

title = scorer + ' Comparison'
print("The box shows the middle 50 percent of the data,\n"
      "the orange line in the middle of each box shows the median of the sample,\n"
      "and the green triangle in each box shows the mean of the sample.")

fig = plt.figure()
fig.suptitle(title)
ax = fig.add_subplot(111)
plt.boxplot(final_results, showmeans=True)
ax.set_xticklabels(final_names)
plt.show()
print(final_names)
```

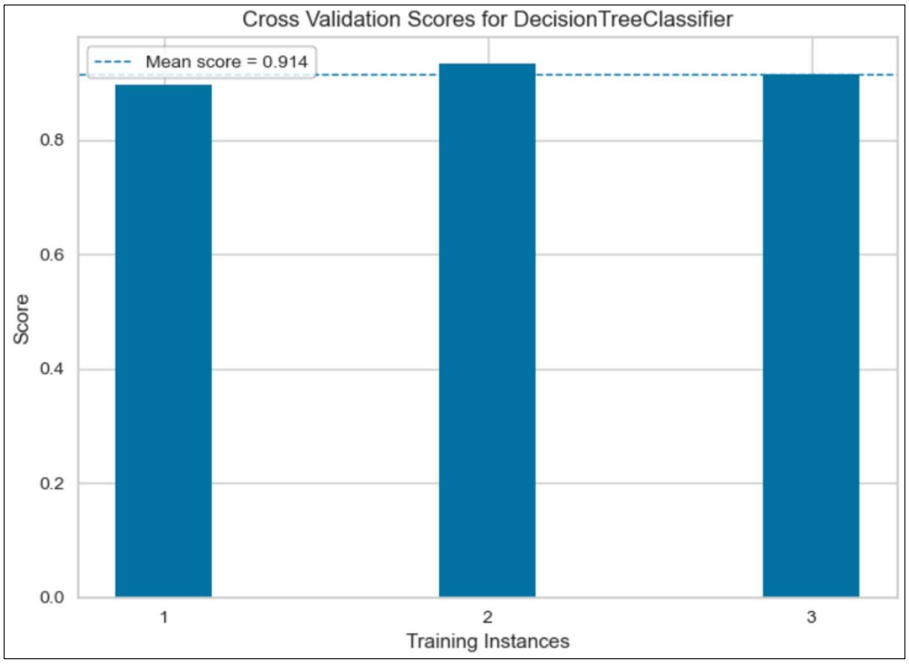
```
The box shows the middle 50 percent of the data,
the orange line in the middle of each box shows the median of the sample,
and the green triangle in each box shows the mean of the sample.
```

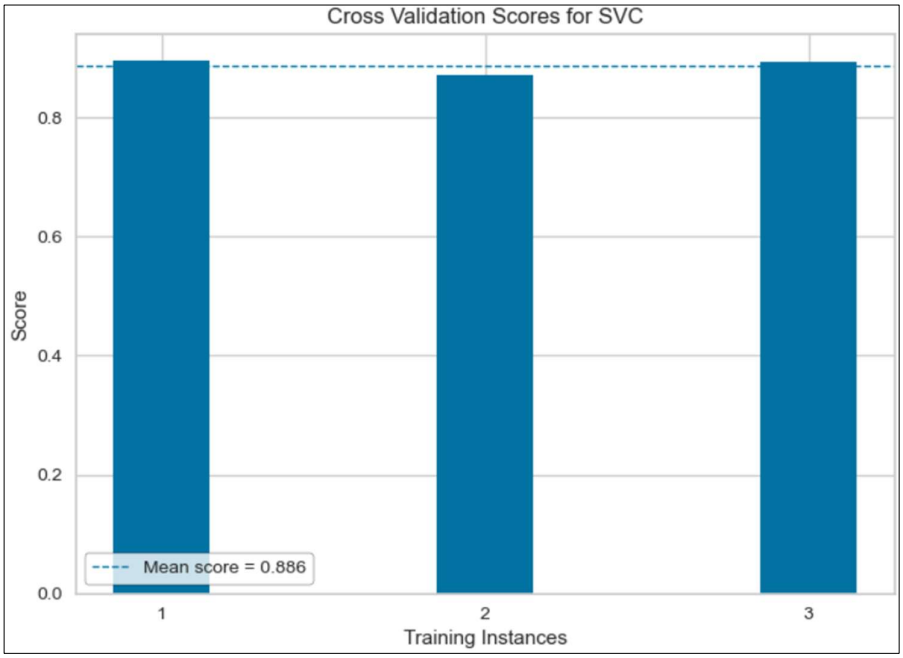



```
['LogReg', 'DecisionTree', 'SupportVectorClf']
```

```
[39]: # individual model per-fold results
from yellowbrick.model_selection import cv_scores

for name, model in opt_models:
    viz = cv_scores(model, X_train, y_train, cv=skf, scoring=scorer)
    viz.set_title(f"{name} - {scorer} per fold")
    viz.show()
```




Cross Validation Scores for DecisionTreeClassifier
Mean score = 0.914
Score
0.0
0.2
0.4
0.6
0.8
1
2
3
Training Instances


Cross Validation Scores for SVC
Mean score = 0.886
Score
0.0
0.2
0.4
0.6
0.8
1
2
3
Training Instances