Jongho Baeck
Class of 2022

Classification Methods and Forensic Applications

Abstract

Diverse machine learning models each seek to answer different types of questions, and a careful implementation of a model can serve as a powerful means to facilitate objective decision-making processes. Today, the application of the models is already prevalent in many industries such as business, entertainment, and social sciences, and it is continuing to expand into sectors that are still governed primarily by human judgements. One of such sectors is forensic science, and Alicia Carriquiry, the Director of the Center for Statistics and Applications in Forensic Evidence (CSAFE), presented a talk in the fall of 2021 at Carleton College, illustrating the application of classification algorithms to discern patterns of markings on fired bullets. This talk is dedicated to exploring the mechanisms of two classification methods, Logistic Regression and Random Forest, and applying them to the forensic data set introduced by Alicia Carriquiry.

Section 1. Introduction

Since the invention of computers, technological developments have given birth to the emergence of new industries around the world. New technological inventions such as the cloud-computing and automated vehicles have already become commercialized, and emerging fields, such as the Metaverse and Augmented Reality (AR), are challenging people's perception of the future to come. While one part of the technological developments has, and is continuing to be, focused on generating new inventions, the other part of it has greatly revolutionized the ways people have been performing certain tasks for a long period of time. An automated machinery in a factory is one obvious example, and it can perform a complex task incomparably faster than a human worker can, with much greater precision and efficiency. The utilization of such computing and technological powers can also be found in many areas beyond the commercial industries, such as in the fields of social sciences and government services, and it is continuing to empower objective and efficient decision-making processes.

On October 12th, 2021, Professor Alicia Carriquiry from the Iowa State University's Statistics department visited Carleton College and described an application of technology in a rather unexpected field of studies: forensic science. Carriquiry, et al. (2019) share the story of Robert Lee Stinson, who was wrongly convicted for a murder case in Milwaukee, Wisconsin in 1984, and consequently served for 23 years in prison until he was later proven innocent. The process through which Stinson was initially charged with murder was rather simple. A group of investigation and forensic experts identified bite marks on the victim's body and concluded that the bite marks must belong to someone who had a missing front tooth. Unluckily, they found that the victim's neighbor Stinson had a missing front tooth and was sentenced to life in prison despite his constant claim of innocence.

Pattern evidence is a subject of forensic science, and it includes analysis of "handwriting, firearms and tool marks, fingerprints, shoe prints, and anything else that comes in the form of an image" (Carriquiry, et al., 2019, p. 29). Through the case of Robert Lee Stinson, the researchers express their concerns on the current usage of pattern evidence in the criminal justice system, and they further cite the words of the 2009 National Academy of Sciences (NAS) report that condemned the pattern evidence "to be rife with subjectivity and lacking in scientific validity" (Carriquiry, et al., 2019, p. 29). They explain that discovering pattern evidence is especially challenging because there are no standardized and reproducible procedures to quantify the degrees of similarities or differences between two samples of evidence collected from crime

scenes. As a result, current forensic practitioners rely on their experience and expertise to visually determine if two samples share enough commonality to conclude if they are likely to be originated from the same source. In case of bite mark analysis, this would be comparing an imagery of bite mark found in a crime scene to that of a suspect's to suggest an expert's interpretation of the similarities found in the two samples.

Finding the current practices of pattern evidence to be problematic, Carriquiry has devoted her time in discovering features from fired bullet imageries that could be used to quantify the similarities with respect to other bullets. As Hare, et al. (2017) explain in detail, seven features have been discovered and applied to machine learning algorithms to test their abilities to discern if pairs of bullets were indeed fired from the same gun. The results have been promising, and this independent project is focused on exploring two classification methods and applying the algorithms to a set of real pair-wise data to test the models' performance.

Following his exoneration, Robert Lee Stinson filed a lawsuit against the city of Milwaukee, and reached a settlement with the city for approximately $7.5 million (The National Registry, 2019). Unfortunately, Stinson is not the only person who is a victim of wrongful conviction, and experts estimate that approximately 1 to 5% of the nearly 2.3 million people incarcerated in the United States are wrongly convicted (Kare, 2021). Such statistics show the enormous amount of financial cost that result from the misjudgments, but do not successfully capture the injustice that the wrongly convicted individuals must suffer. Alicia Carriquiry's research is therefore promising in that she introduces technological applications to criminal investigation processes that can help reduce subjectivity and prevent tragedies of individuals like Robert Lee Stimson.

Section 2. Data

Hare, et al. (2017) report that Carriquiry and her group of researchers provide detailed methodologies that they have developed to describe striation marks on bullets. It should be noted that their work is focused on identifying matches between bullets fired specifically from pistols.

Figure 1 below illustrates a typical anatomy of a pistol. In each firing process, a pull of the trigger separates a bullet from the loaded cartridge, and the bullet travels through the barrel to be fired out of the muzzle. Hare, et al. (2017) explain that the barrel forms a spiral-like structure, composed of *lands* and elevations called *grooves*, to cause a spin and stabilize the trajectory of a fired bullet (see **Figure 2**).

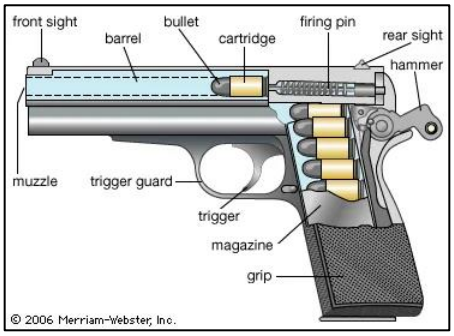


Figure 1. The basic anatomy of a pistol gun (Image source: Britannica)

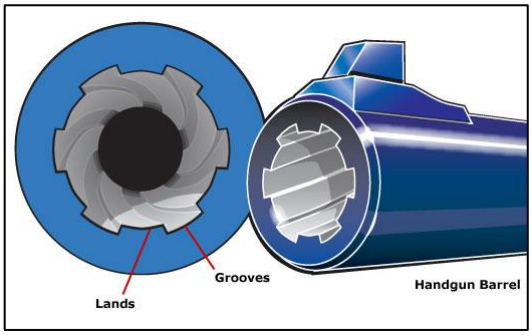


Figure 2. Close examination of the structure of a gun barrel (Image source: Kalkomey)

According to the researchers, the inherent manufacturing defects of a barrel and friction caused by the inner surface of the barrel during the firing process leave *land engraved areas* and *groove engraved areas* on a bullet (see **Figure 3**). Closer examination of these areas reveals so-called striation marks that are "assumed to be unique the barrel" (Hare, et al., 2017, p. 2333). The researchers refer to striations in each engraved areas as an *impression* and perceived it to contain unique striation marks that could be compared to those of other bullets to assess their similarities.

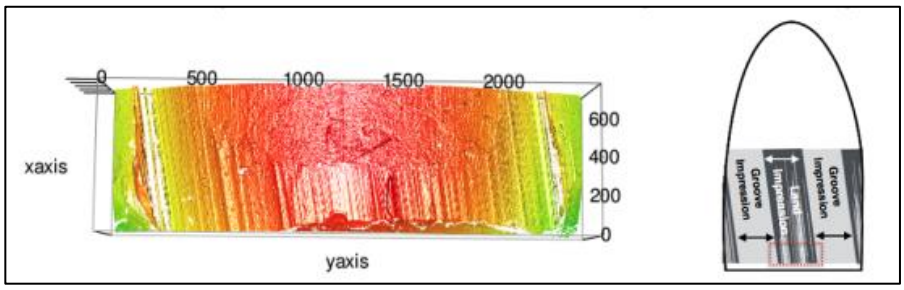


Figure 3. Striations in each land and groove engraved are called impressions, and the image on the left is a 3D representation of a land impression (Image source: Hare et al., [2017]).

For effective comparison of striations between two land impressions, the researchers sought to reduce the dimensionality of the three-dimensional land impressions as shown in **Figure 3** while still capturing relevant information about striations on the surface of a land. To do so, they took a cross-sectional view of each land impressions at a fixed height and outlined the contours at relative locations. The fixed height from which the researchers took cross-sectional view of was chosen toward the bottom of the bullet as they explained that "individual characteristics [impressions] are best expressed at the lower end of the bullet" (Hare, et al., 2017, p. 2339)

Figure 4 shows the cross-sectional view of a land impression with respect to its relative location, and the blue line represents the loess fit that essentially best captures the curve of bullet after it was fired. Once the researchers found the best fit, they were able to generate a residual plot (**Figure 5**) that represents the differences between the actual height of the land and the fit line at relative locations, and this graphical representation of the residuals is referred to as *signatures.*

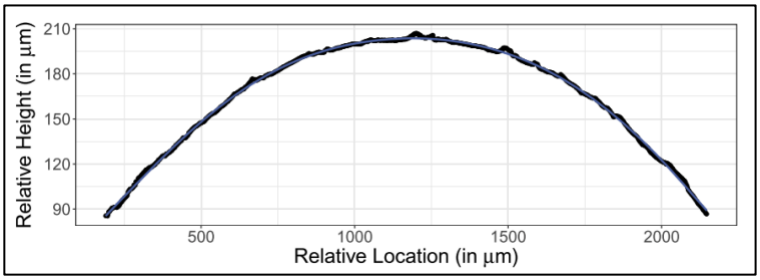


Figure 4. Cross-sectional view of a land impression. The blue line represents the loess fit. (Image source: Hare et al., [2017])

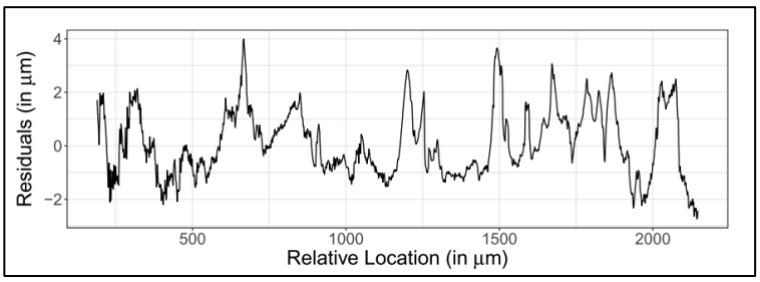


Figure 5. Residuals of land impressions with respect to the loess fit at relative locations.
(Image source: Hare et al., [2017])

Hare, et al. (2017) demonstrate that the advantage of visualizing land impressions in terms of their residuals with respect to the best fit curve is that local maxima and minima can be identified. The researchers refer to a local maximum as a *peak* and a local minimum as a local *valley*. As the researchers explained that striations are unique to the barrel, we can expect the bullets fired from the same barrel (gun) to share the same peaks and valleys at a certain interval of locations (see **Figure 6**). In the image, each box represents signatures of a fired bullet, where the green and red areas indicate local minima and maxima, respectively.

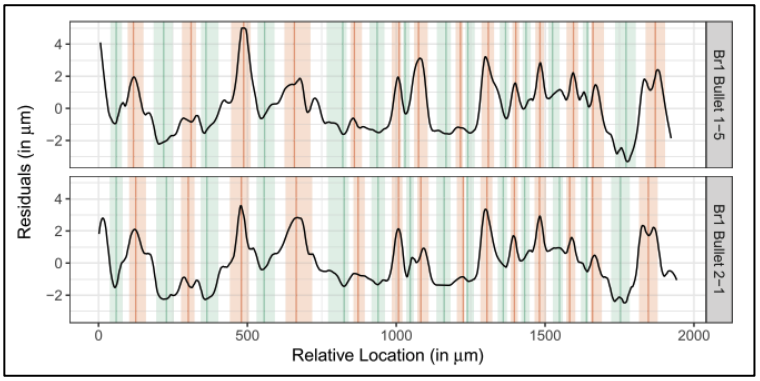


Figure 6. Comparison of signatures from two fired bullets. The two bullets were fired from the same barrel.
(Image source: Hare et al., [2017])

The researchers overlay one signature on top of the other to assess the degrees of similarities between two bullets and extract the following features listed below. Further details on the features can be found at Hare, et al. (pp. 2340 - 2342).

1. The maximal number of consecutive matching striae, referred to as CMS
2. The minimal number of consecutive non-matching striae, referred to as CNMS
3. The number of total matches
4. The number of nonmatches
5. The cross-correlation function (CCF) which measures the correlation of movements between the signatures
6. The average Euclidean vertical distance between the signatures
7. The sum of the average of the matching extrema's absolute height. In other words, if there exists a matching valley at a certain interval, the average of the absolute heights of the valleys is calculated and is summed with those of any other matching extrema.

In preparing this project, a set of pair-wise data was obtained from Professor Adam Loy from Carleton College's Statistics Department. **Table 1** displays a few rows and partial columns

of the preprocessed version of the data set. Each row describes a pair-wise relationship between signatures from two bullets. The first seven columns are the quantitative features developed by Hare, et al. (2017) and the last column *match* is a boolean feature that represents whether the pairs match. The term bullet *match* is used interchangeably to indicate that the two bullets are fired from the same gun barrel.

ccf	d	num_matches	num_mismatches	non_cms	sumpeaks	cms	match
0.267628	1.545342	6	21	10	7.163380	2	0
0.418789	2.772891	5	18	9	8.396331	2	0
0.246006	3.045523	0	18	18	0.000000	0	0
0.331219	1.663244	11	16	6	11.177136	4	0
0.313009	2.394211	8	13	3	11.089648	2	0

Table 1. Display of the pair-wise bullet relationship data set obtained by Carleton College's Statistics Department.

Section 3. Methodology

Identifying whether a set of two bullets match, given the appropriate measures to describe the relationship between the two, is a binary classification problem. In a binary classification problem, the goal is to assign a label to an observation to indicate to which *class* the specific observation most likely belongs to. For example, in a classification problem that seeks to predict if a student will pass a course or not, the only possible set of *class*, or outcome, is that the student either passes or fails the course. An experimenter might consider the student's past grades and attendance, and finally decide whether the student will most likely pass the course or not. Ultimately, we need to build a classifier that recognizes and learns patterns from past observations with known outcomes so that when a new observation is introduced the model, the model can correctly assign labels to it.

Carriquiry, et al. (2019) have used a Random Forest classifier to determine whether two bullets match. Various open-source models can be used to build a classifier, and some of these models include Logistic Regression, Decision Trees, Random Forest, and Support Vector Machine (SVM). The purpose of this independent project was to learn Logistic Regression and Random Forest models, and the algorithms of the models will be discussed in the following sections.

3.1 Decision Tree Model

Before learning the Random Forest model, it is essential to understand the Decision Tree model as a Random Forest is comprised of many Decision Tree classifiers. As its name suggests, a Decision Tree classifier forms a tree-like structure, as shown in **Figure 6**. The tree diagram in the figure illustrates the processes through which a Decision Tree classifier will hypothetically classify whether a customer will purchase a used car (Cravit, 2021).

3.1.1 Structure of a Decision Tree

Chauhan (2022) highlights important characteristics of the Decision Tree structure. First, every time the tree branches out from a *node*, the depth of the tree becomes equally incremented. In the diagram, we observe that the tree has branched out from the *root node*, which is the very top node, for a maximum number of three times which makes the maximum tree depth of three. **Figure 7** illustrates a general structure a Decision Tree model and terminologies of its elements

(Chauhan, 2022). The nodes at which further branching out is discontinued is referred to as the *leaf nodes*, and as it can be noticed from the diagram, not every *leaf node* must necessarily possess the same level of depth. Lastly, each node is split based on a binary condition. It helps to understand that a node contains several observations with features that describe the characteristic of each observation, and that it distributes its contained observations into two distinct *child nodes* depending on the splitting condition. In **Figure 6**, the original data set at the root node is first split based on the color of the car, and data whose values in the *color* column equals red will be distributed to the left child node while the other will be distributed to the right node.

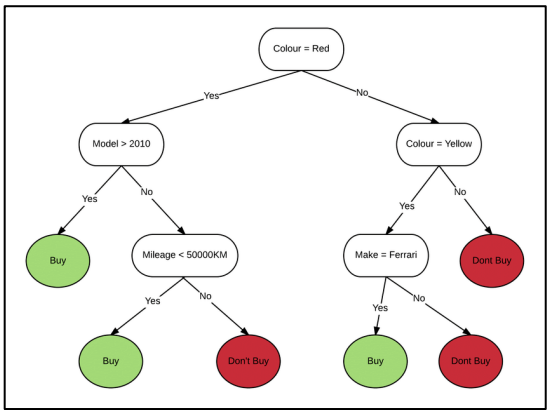


Figure 6. An illustration of a Decision Tree classier (Image source: Cravit [2021])

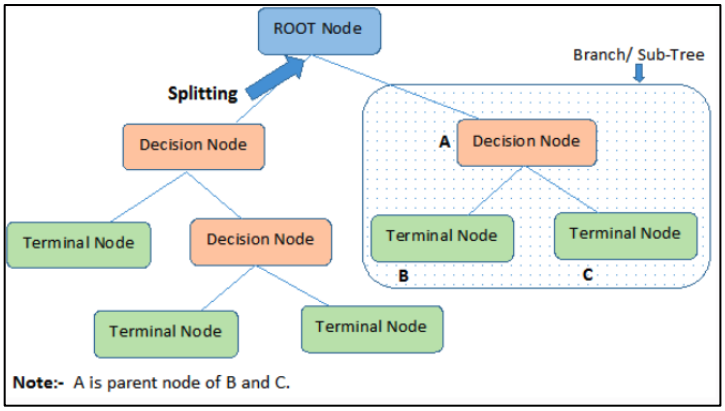


Figure 7. General structure of a Decision Tree model (Image source: Chauhan [2022])

3.1.2 Finding the Optimal Splitting Condition in a Decision Tree Model

There are different measures used to determine how efficiently a node is split. Pedregosa, et al. (2011) explain that the default algorithm used to find the optimal splitting condition in Python's *scikit-learn* library is the Classification and Regression Trees (CART), which utilizes the Gini Index. The Gini Index is a metric that measures the noise and variability in a particular set of data, and its formula is defined below in **Equation 1** (Tahsildar, 2019):

$$Gini\ Index(S) = 1 - \sum_{i \in S} (p_i)^2$$

Equation 1. Gini Index formula

In the formula, *S* refers to the list of binary data that we wish to measure the Gini Index of, and *i* refers to each class identifier within the list. In case of a binary set of data comprised of zeros and ones, *i* is either zero or one, and p_i defines the probability of class *i* observed in *S.* For example, if we are given a binary set of data $S = \{0, 0, 1, 0, 1, 1, 0, 0, 1, 0\}$, we can calculate the Gini Index of the data set as below:

$$Gini\ Index(S) = 1 - \sum_{i \in S}(p_i)^2 = 1 - \left(\left(\frac{6}{10}\right)^2 + \left(\frac{4}{10}\right)^2\right) \approx 0.48$$

It can be inferred from the equation above that the Gini Index reaches a maximum of 0.5 when values within a set of binary data are equally distributed across the classes, and that it reaches a minimum of 0 when all values in the set belong to a single class (Loazia, 2020) (see **Figure 8**).

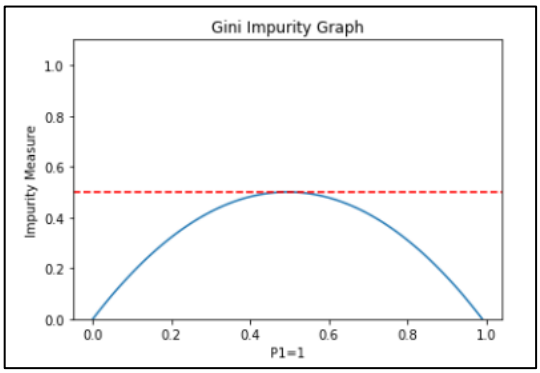


Figure 8. Maximum and minimum Gini Index in a binary data set
(Image source: Loazia [2020])

Using the concept of Gini Index, we can build our intuition on how to find the optimal splitting condition. Recall that when a *node* is split, it creates two *child nodes*. Using the formula discussed above, we can then calculate the weighted average of the Gini Index of the target data set in each of the *child nodes*. This calculation can be expressed as below in **Equation 2** (Karabiber, 2020):

$$Gini\ Index_{condition} = \frac{n_1}{n} Gini(D_1) + \frac{n_2}{n} Gini(D_2)$$

Equation 2. Weighted average of child nodes' Gini Index

This formula essentially summarizes the Gini Index of the target variables in the two child nodes that result from a certain splitting condition. D_1 and D_2 each refer to a child node, where *n* refers to the total number of observations in the parent node prior to split while $n_1$and n_2 refer to the total number of observations in each child node. To illustrate an example, **Table 2** displays a hypothetical data set that includes students' pass or fail record a particular statistics course, and the feature variables include whether each student is a senior and a statistics major. Applying **Table 2** data to **Equation 2** above, for a splitting condition of "Statistic Major = Y," we get a result of approximately 0.167.

Student	Senior	Statistics Major	Pass/Fail
1	N	Y	1
2	N	N	0
3	Y	N	0
4	N	N	0
5	Y	Y	1
6	Y	Y	1
7	N	Y	1
8	N	Y	0
9	Y	N	0
10	Y	Y	1

D1

Student	Senior	Statistics Major	Pass/Fail
1	N	Y	1
5	Y	Y	1
6	Y	Y	1
7	N	Y	1
8	N	Y	0
10	Y	Y	1

D2

Student	Senior	Statistics Major	Pass/Fail
2	N	N	0
3	Y	N	0
4	N	N	0
9	Y	N	0

Table 2. Hypothetical data set that illustrates students' pass/fail record in a particular statistics course

$$Gini\ Index_{statics\ major\ =\ Y} = \frac{6}{10} Gini(D_1) + \frac{4}{10} Gini(D_2)$$

$$= \frac{6}{10}\left(1 - \sum_{i \in D_1} (p_i)^2\right) + \frac{4}{10}\left(1 - \sum_{i \in D_2} (p_i)^2\right)$$

$$= \frac{6}{10}\left(1 - \left(\left(\frac{5}{6}\right)^2 + \left(\frac{1}{6}\right)^2\right)\right) + \frac{4}{10}\left(1 - \left(\left(\frac{0}{4}\right)^2 + \left(\frac{4}{4}\right)^2\right)\right) \approx 0.167$$

Figure 9. Calculation of weighted Gini Index using data from Table 2.

Considering that the Gini Index ranges from a minimum of 0 to a maximum of 0.5, a Gini Index of 0.167 indicates that the above splitting condition has resulted in child nodes whose values of the target variable (*Pass/Fail* column) are relatively homogeneous. In fact, **Table 2** shows that splitting the *root node* with respect to the condition created one child node (D_1) where 5 out of a total of 6 values in the *Pass/Fail* column were identical, and the other node (D_2) where all the values in the column were identical. The goal of the CART algorithm is to test split a data set with respect to every possible option of splitting conditions from the feature variables, and to find the one that maximizes the homogeneity in the resulting nodes.

3.1.3 Building a Tree and its Limitations

Chauhan (2022) further explain that a Decision Tree model is recursive, meaning that once the optimal splitting condition is discovered at the *root node* level, the model creates two *child nodes* at the tree depth of 1. Of course, a model with a single depth isn't enough to find the patterns among a set of observations, so the model seeks to find additional conditions to further split the *child nodes* that resulted from the initial split. In this case, the *n* from **Equation 2** should be the total number of observations found at the node from where the tree is trying to branch out.

A tree may branch out as far as all the values in its *child nodes* are homogeneous and therefore it isn't necessary to further split the nodes. However, this will likely indicate that the tree has branched out excessively, trying to learn every possible pattern within the training data set to successfully categorize each observation. The term *overfitting* is commonly used to describe such phenomenon where a model is overly trained to label every training data as correctly as possible. When a test data set is applied to an overfit model to predict the outcome of unseen observations, the model "can be very non-bust" because "a small change in the data can cause a large change in the final estimated tree" (James, et al., 2021, p. 340).

Luckily, there are ways to prevent a Decision Tree model from becoming too complex. One technique is to tune the hyperparameters that pre-defines the structure of the tree model before it fits to the training data set. A few examples of such parameters and their descriptions are listed below, and please refer to Python's scikit-learn documentation for in-depth explanations (Pedregosa, et al., 2011):

- **max_depth**: Sets the maximum level of depth of a tree
- **min_samples_split**: Sets threshold for splitable nodes. Too small nodes may be restricted to be further split
- **min_samples_leaf**: Sets the minimum size for a node to be concluded as a leaf node

Another way to minimize overfitting issue is to use *bootstrap aggregation, or bagging,* which is built on idea that "averaging a set of observations reduces variance." (James, et al., 2021, p. 340) The Random Forest model is an example of using multiple bagged trees to make predictions, and the details will be explained in the next section.

3.2 Random Forest Model

James, et al. (2021) explain that even if a model is constructed with high variance and complexity by overfitting the training data set, averaging predictions from multiple overfit models can help improve predictability. This explains the aggregation part of a bagging method, and the ideal way to do so would be to collect a large amount of data, create multiple training data sets, and train each model with different training sets. In real experimental setting, however, there is rarely an access to such large training the data set. A way to overcome this issue would be to create multiple bootstrapped resamples of the training data set as many times as it is desired. Once such bootstrapped resamples are created, it is now possible to utilize the *bagging* method to build a Random Forest model.

3.2.1 Structure of a Random Forest Model

Recall from **Section 3.2** that understanding the Decision Tree model is key to visualizing how a Random Forest Model functions. A Random Forest model is comprised of bagged Decision Trees, where each tree is fit with respect to different bootstrapped resamples of the training data set. An additional modification that is applied in building the bagged trees is that during each stage of splitting the nodes, only a subset of predictor variables is considered. Typically, if there are a total of N predictor variables in the training data set, randomly selected $\sqrt{N}$ variables will be used to optimize the splitting condition as discussed in **Section 3.1.2.** This explains why this model is named the "Random" Forest model.

James, et al. (2021) offer an intuition for such characteristic of the model. They illustrate a scenario where every bagged tree in a Random Forest model is allowed to consider all predictor variables for a split and there exists a variable that is exceptionally good at classifying the observations. The authors then state that every bagged tree will most likely choose the variable as their first splitting criterion, which will result in high correlation between the bagged trees. They argue that "averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities" (James, et al., 2021, p. 344). Hence, they explain that limiting the tree to only consider a subset of the variables is a way to build each tree distinctive to another.

After each bagged trees in a Random Forest model makes predictions for an observation, there are a few ways to aggregate the results from the individual trees. The authors explain that

the Random Forest model can take a majority vote of the results from its bagged trees. If there exist 100 bagged trees in a Random Forest model, and 90 predicted positive and the other 10 predicted negative for an observation, then the model will conclude that it is positive (90 is greater than 10) by the majority voting rule. In contrast, Python's scikit-learn documentation offers a different way of aggregating the results. According to the documentation, the "mean predicted class probabilities of the trees" is calculated, where "the class probability of a single tree is the fraction of samples of the same class in a leaf" (Pedregosa, et al., 2011). In other words, each bagged tree finds the leaf node at which an observation falls under, calculates the class probability within the leaf node, and the Random Forest model averages each class probabilities across the trees to decide to which class the observation seems most likely to belong to.

3.2.2 Hyperparameters of the Random Forest Model

Because a Random Forest model is comprised of bagged Decision Trees, it inherits many of the hyperparameters of a Decision Tree model. Some of the hyperparameters of a Random Forest in Python's scikit-learn package include the ones already mentioned in **Section 3.1.3** such as *max_depth, min_samples_split,* and *min_samples_leaf.* One of the other hyperparameters that play an important role in the performance of a Random Forest model is *n_estimators*, which defines how many bagged trees to generate to in a Random Forest. Python's Random Forest Classifier also allows users to manipulate the number of predictor variables that each stage of split considers. According to the documentation, the default is $\sqrt{N}$, where *N* refers to the total number of predictor variables there exist in the data set. There are alternatives to use $log_2(N)$, or simply consider all predictor variables, but the latter will not be used as cautioned by James, et al. (2021).

3.2 Logistic Regression

The Decision Tree and Random Forest models are just two examples of classification models. This section is dedicated to understanding another classification model called Logistic Regression, which is different from the Random Forest in that it fits an equation to describe the relationship between values of predictor variables and the probability of an outcome.

3.2.1 The Logistic Function

In explaining Logistic Regression, many textbooks and other resources compare the model to Linear Regression. The main difference between the two is that Linear Regression seeks to predict a quantitative variable while Logistic Regression models the probability that an observation falls into a particular class of categorical variables. **Figure 11** compares fitting a Linear Regression (left) and Logistic Regression (right) to a set of data that stores individuals' monthly *income* balance and their loan default history. This is a simulated data and plot from James, et al. (2021), where the orange dots represent observed data where 0 indicates a non-default case while 1 indicates a default case.

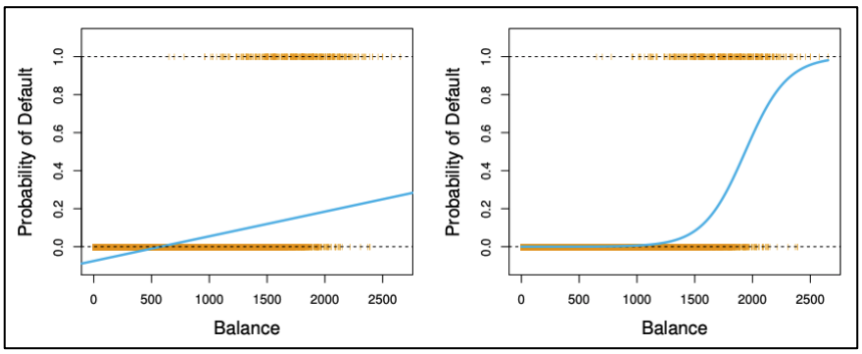


Figure 11. Comparison of Linear Regression and Logistic Regression (Image source from James, et al. [2022])

By fitting a Linear Regression, as in the left panel, to a set of binary data, James, et al. (2021) introduces the limitation of the model in its interpretability of the outcome. Since we want to model the probability that an individual will likely default on loan, our *Y* value must range from zero to one. As it can be noticed from the left panel in **Figure 11,** some *Y* values are estimated to be below zero and negative which are "not sensible"[1] probabilities.

The authors then introduce the logistic function, a S-shaped line that "outputs [values] between 0 and 1 for all values of *X* [predictor variables]" (James, et al., 2021, p. 134). An example of the logistic function can be seen on the right panel of **Figure 11**, and it is defined below where β_0 and β_1 are the coefficients.

$$p(Y = 1 \mid X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

Equation 3. The logistic function expressed by James, et al., (2021)

Note that the authors only consider a single predictor variable to model the probability of default. In general, data sets contain more than one predictor variable, and **Equation 3** can be modified to accommodate multiple predictor variables as below in **Equation 4**. For the simplicity of explanation and expression of equations, we will assume a single predictor variable scenario in alignment with the authors.

$$p(Y = 1 \mid X_1, X_2, \ldots, X_n) = \frac{e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n}}{1 + e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n}}$$

Equation 4. General form of the logistic function (James, et al., [2021])

Unlike Linear Regression, the relationship between *X* and the value of *Y* is not linear. To assess and interpret the effect of any change in values of *X*, they rearrange **Equation 3** and introduce a *logit* (also called *log-odds*) function.

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X$$

Equation 5. A logit found by rearranging the logistic function (James et al., [2021])

The authors states that "the logistic regression model has a logit that is linear in *X*," meaning that changes in *X* have linear effects on log-odd, not necessarily on the probability of an outcome.

[1] Statistical Learning

3.2.2 Finding Coefficients by Maximum Likelihood Estimation

Finding the estimates of the coefficients β_0 and $\beta_1$allows us to best fit the S-curved logistic function with respect to observations, as visualized in the right panel of **Figure 11**. Chihara, et al. (2019) define the likelihood function of a binary Logistic Regression by

$$L(\beta_0, \beta_1) = \prod_{i=1}^{n} {p_i}^{Y_i}(1 - p_i)^{1-Y_i}$$

Equation 6. The likelihood function of Logistic Regression (Chihara, et al., [2019])

In the equation above, p_i refers to the probability that an observation *i* belongs to class 1 as we have discussed in **Equation 3** above. As James, et al. (2021) explain, the way to maximize the likelihood function is to find coefficients β_0 and β_1 such that $p(Y_i = 1 \mid X)$ is maximized for observations whose actual outcome is 1, and $1 - p(Y_i = 1 \mid X)$ is maximized for observations whose actual outcome is 0. The goal is to find the coefficient estimates that best predicts the probability of each observation as close as its actual value.

Chihara, et al. (2019) illustrate to find the maximum of the likelihood function, we need to take the natural logarithm of the likelihood, take partial derivatives of the function with respect to each coefficient, and estimate the coefficients by setting each partial derivative equal to zero. However, the authors note that there is "no closed-form solution" in finding the best estimates of the coefficients and that we must resort to softwares to estimate coefficients.

3.3 Implementation of Random Forest and Logistic Regression to Bullet Data Set

After understanding the algorithms, Random Forest and Logistic Regression were applied to the pair-wise data set that was mentioned in **Section 2**. The original data set that was obtained by Carleton College's Statistics department looked as below (see **Figure 12**).

	b2	b1	ccf	D	num.matches	num.mismatches	non_cms	sumpeaks	cms	data	resID	id.x	id.y	match
0	Br1 Bullet 1-1	Ukn Bullet C-2	0.296164	1.810042	8	22	5	7.146495	2	data-new-all-25-25/bullet1.RData	128	4	1	False
1	Br1 Bullet 1-1	Ukn Bullet S-5	0.263017	2.331692	4	19	11	4.847843	2	data-new-all-25-25/bullet1.RData	185	17	1	False
2	Br1 Bullet 1-1	Ukn Bullet D-1	0.266002	1.492308	10	17	9	7.358286	6	data-new-all-25-25/bullet1.RData	133	23	1	False
3	Br1 Bullet 1-1	Ukn Bullet L-4	0.303057	1.975364	2	20	7	1.429595	1	data-new-all-25-25/bullet1.RData	166	46	1	False
4	Br1 Bullet 1-1	Ukn Bullet H-6	0.226535	2.716547	2	19	14	1.528081	2	data-new-all-25-25/bullet1.RData	156	8	1	False

Figure 12. Original pair-wise bullet comparison acquired by Carleton College's Statistics department

There was a total of 43,890 observations in the data set. The *b1* and *b2* columns represent the two bullets that the researchers were interested in comparing. Closer look at the columns reveal that the source of the bullets was not known all the time. This resembles the real crime scene in that the role of forensic experts is to identify an unknown bullet that was found in the scene. For such reason, a natural training set was formed by selecting rows that had both known values of *b1* and *b2*. As a result, the training and test data set were divided such that each comprised of 14,280 and 29,610 observations, respectively.

After dividing the training and test data sets, columns *b1* and *b2* were removed because in the test data set, they are assumed to be unknown and to provide no information about the bullets. In addition, some miscellaneous columns such as *data*, *resID*, *id.x,* and *id.y* were removed, and the only columns that were used to build classification models were the seven

characteristics described by Hare, et al. (2017) and the *match* column that indicates whether two bullets match.

3.3.1 Exploratory Data Analysis

Note that exploratory data analysis was conducted using the training data set. We have learned from Hare, et al. (2017) that the seven features were extracted from an overlay of two signatures. This hints that we may expect some correlations between the seven predictor variables because the information was retrieved from the same source, although each observation may be independent of another. Data analysis reveals that there exist correlations between a few independent variables (see **Figure 13**). Assuming a correlational value greater than 0.7 is significant, *num_matches* column seems to be the column that is most frequently correlated with the other columns. It has correlational value of approximately 0.78 with *sumpeaks* column, and 0.83 with *cms* column.

Logistic Regression is known to assume that there is no multicollinearity among the predictor variables (UCLA, 2022). In other words, it is essential that there are no strong correlations among the variables. This is contrary to what we've discovered in **Figure 13**, and as the assumption suggests, we should utilize a subset of non-correlated features to predict the outcome.

The reason why Carriquiry, et al. (2019) extracted the features is that they sought to build a Random Forest model that take all the features into consideration to make predictions. Because the purpose of this project was not replicate their work, but to design the two classification methods and compare their performances, it seemed reasonable to disregard *num_matches* column to satisfy one of the model's assumptions and use the same set of predictor variables for both models.

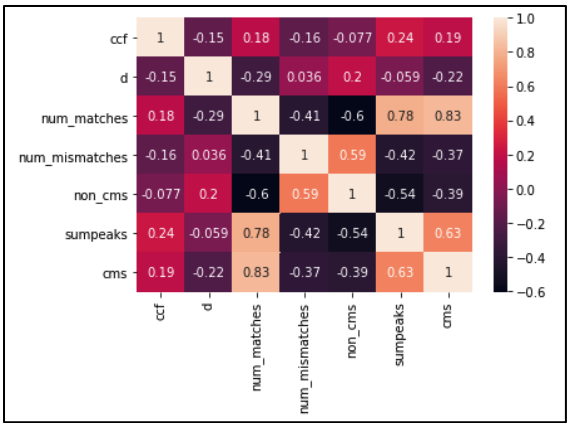


Figure 13. Correlation between predictor variables

Further exploratory data analysis was conducted to compare the distribution of each predictor variables with respect to each class (match/mismatch). Visualization in **Figure 14** reveals that the average values of *cms, ccf,* and *sumpeaks* columns is higher for the matched bullets while those of *num_mismatch, non_cms,* and *d* columns are higher for mismatched bullets. This makes intuitive sense because we would expect the number maximal number of CMS (*cms*), the cross-correlation function value (*ccf*), and the sum of the average absolute heights (*sumpeaks*) for matched signatures to be higher. As Carriquiry, et al. (2019) suggest, however, we observe that distributions for each variable overlap between classes, and there is no single variable that can clearly define whether two bullets match. We can expect, however, that

the models can learn the patterns that is not discernible from simple distribution comparisons and be able to correctly label our test observations.

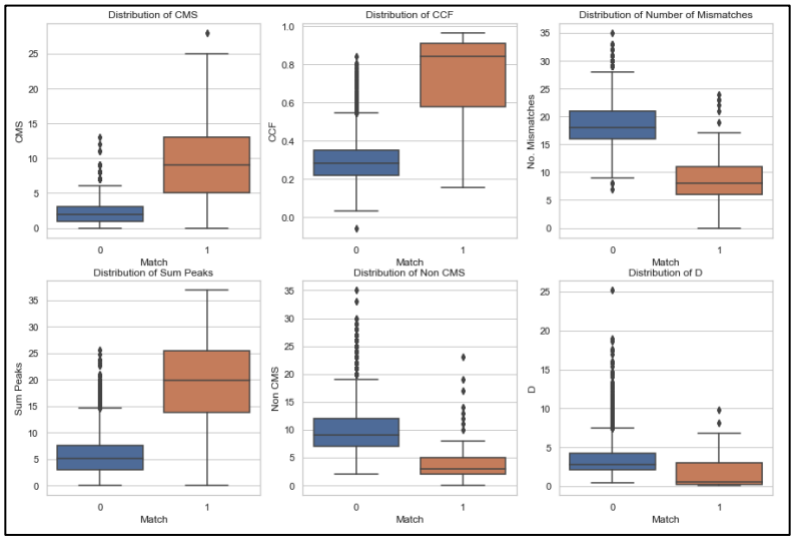


Figure 14. Side-by-side boxplots of the distribution of the predictor variables

3.3.2 Evaluation Metric

There exist various metrics to evaluate the performance of a binary classification model. Typically, a confusion matrix, which compares the predictions of observations to the actual values of the observation, is used to derive these metrics (shown in **Figure 15**). From True Positive, False Positives, False Negatives, and True Negatives, the following metrics can be calculated (Analytics Vidhya, 2021):

- Accuracy = (True Positive + True Negatives) / Total number of observations
- Precision = True Positive / (True Positive + False Positive)
- Recall = True Positive / (True Positive + False Negative)

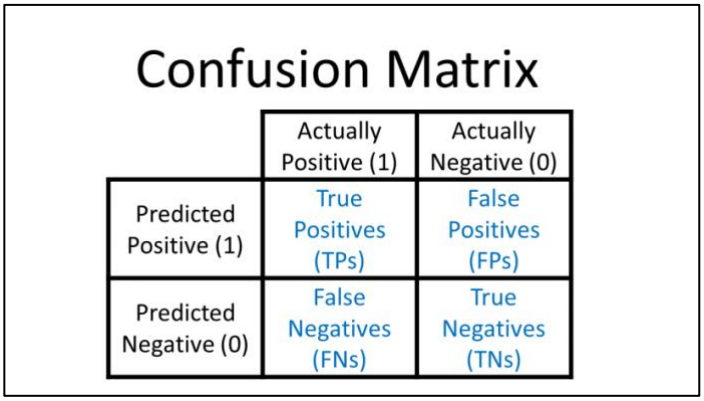


Figure 15. Confusion matrix configuration (Image source: Analytics Vidhya, 2021)

At an initial glance, accuracy seems to be the most appropriate measure because it measures the fraction of correctly labeled observations out of the total number of observations. However, one issue with using accuracy as the metric to assess predictions in our question of interest is that the data set is heavily imbalanced. From a total of 14,280 observations in the

training data set, only 120 observations (about 0.8%) are matched cases. Similarly, from a total of 29,610 observations in the test data set, only 432 observations (about 1.5%) are matched cases. If we were to label every observation in the test data set as non-match, then we will still be able to result with an accuracy of approximately 98.5%. This score would be a distorted measure of the model performance, as it fails to correctly identify any actual matches.

Precision captures the fraction of actual positives out of predicted positives. In the context of criminal investigation, we want to maximize this metric because if forensic scientists predict that a set of bullets are matching, then the two bullets must indeed be fired from the same gun barrel to prevent any false convictions. We also want to maximize recall score because we want to correctly predict as many matching cases as possible from the total matches. Failure to identify match cases will be a huge cost to the society as there may be secondary crimes committed from the culprits who benefit from poor investigation.

The other metric that takes both precision and recall into consideration is called F1 score. F1 score is defined as below, and it is a "harmonic mean of precision and recall" (Analytics Vidhya, 2021).

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Equation 7. F1 score formula (Analytics Vidhya [2021])

For this project, maximizing F1 score seemed most reasonable because the data set is heavily imbalanced and misjudgments, both wrongful conviction and freeing potential criminals, is a high cost to the society.

3.3.3 Modeling

The goal of this project was to create a Logistic Regression model and a Random Forest model. In the process of developing the Random Forest model, a Decision Tree model was also developed to compare its performance with that of a Random Forest model.

Python's scikit-learn package offers a very useful tool for hyperparameter optimization called GridSearchCV and the documentation offers detailed information (Pedregosa, et al. [2011])). It allows users to simultaneously input varying values of different hyperparameters and tests each possible combination of hyperparameters. In doing so, it also conducts cross-validation and returns the cross-validation scores of the different combinations of hyperparameters that it tested. As mentioned in the earlier section, GridSearchCV was constructed to evaluate F1 score, and cross-validation was conducted via stratified 5-folds method because the training data set was imbalanced, and we needed to make sure that each validation training set contained match cases.

Because the models were developed using a local computer with limited computing power, intense hyperparameter optimization was not feasible. As a result, the hyperparameters that were optimized in the Random Forest model consisted of *n_estimators, max_depth, min_samples_split,* and *min_samples_split*, which were discussed in **Section 3.2.2.** No hyperparameter optimization was conducted for the Logistic Regression model.

After the hyperparameters were optimized, the model's performance on the test data set was evaluated. The code that was used to conduct this project can be found in the **Code Appendix** section.

Section 4. Results

A Decision Tree model was designed as a reference to compare its performance to that of a Random Forest. Two Random Forest models were designed: one model was only tuned with respect to *n_estimators* (referred to as Random Forest V1 in **Figure 16**) and the other was designed using additional hyperparameters such as *max_depth*, *min_samples_leaf*, and *min_samples_leaf* that characterize the Decision Tree components of a Random Forest (referred to as Random Forest V2 in **Figure 16**). Finally, a simple Logistic Regression, without any hyperparameter, was designed. **Figure 16** outlines the characteristics of each model, best cross-validation scores and hyperparameters with respect to F1 score in the training data set, and performance scores in the test data set.

				Training Performance			Test Performance		
Model	**Hyperparameters**	**Hyperparameter Values**	**Best Parameters**	**Best CV Precision**	**Best CV Recall**	**Best CV F1**	**Precision**	**Recall**	**F1**
Decision Tree	max_depth min_samples_leaf min_samples_split	1 to 9 1 to 9 1 to 9	9 8 2	0.97	0.65	0.77	0.92	0.61	0.74
Random Forest V1	n_estimators	1 to 300, interval of 5	135	1.00	0.73	0.84	1.00	0.62	0.77
Random Forest V2	n_estimators max_depth min_samples_leaf min_samples_split	1 to 200, interval of 25 1 to 9 1 to 9 1 to 9	175 4 1 2	1.00	0.73	0.84	1.00	0.57	0.73
Logistic Regression	-	-	-	1.00	0.62	0.76	1.00	0.55	0.71

Figure 16. Comparison of Model Performance

The results illustrate the advantage of using bagged Decision Trees to generate a Random Forest model. The two Random Forest models outperform the Decision Tree model in all cross-validated metrics in the training set and better predicts the test data set. Interestingly, the two Random Forest models performed nearly the same in the training data set, both resulting in F1 scores of 0.84. F1 score can be interpreted in terms of precision and recall scores, and at F1 score of 0.84, both Random Forest models perfect its precision. In other words, the models predicted no false positives, free from falsely classifying non-matching bullets as matching. Although the models' F1 score decreased in classifying the test data set, they still maintained 100% precision scores. Interestingly, the Radom Forest model that was tuned only with respect to *n_estimators* (V1) performed better in the test data set compared to the other model (V2) that was tuned with additional hyperparameters. Specifically, the former performed better than the latter in terms of recall score by 5 percentage points, indicating that the former was able to capture more true positives than the latter without jeopardizing its precision. It appears that tuning Random Forest with such additional hyperparameters had no positive effect on the model performance and therefore, the first version of the Random Forest seems to be a better model.

This model also appears to perform better than the Logistic Regression model with respect to all performance metrics. In fact, the Logistic Regression resulted in the lowest F1 score of 0.71 in the test data set because it scored a very low recall score of 0.55. Such recall score indicates that the model was able to predict only half of the actual matched cases as match. In the context of criminal investigation, implementation of the model wouldn't be ideal because it seems to free high suspects very easily.

Section 5. Conclusion

This independent project was intended to learn the mechanisms of Logistic Regression and Random Forest models. Understanding the details of the classifiers allowed implementation

of the models to a real-world data set and a total of four models were created and compared based on their performances.

Even though the first version of the Random Forest model performed the best, it can be further improved. We observe that GridSearchCV found *n_estimators* of 135 to be the optimal hyperparameter for the first version of the Random Forest. This suggests that increasing the number of bagged trees in the forest have not necessarily enhanced the model's performance in cross-validation. We have also witnessed above that introducing additional hyperparameters decreased the model performance. For such reason, the only reasonable option to enhance its performance is to train the model with more balanced data set. The training data set was heavily imbalanced, having only 1% of matched cases.

The training data set can be manipulated for each class to have the equal number of observations. However, there are only 120 matched cases in the training data set and training a classifier with only 240 observations does not seem reasonable, especially given that there are a total of 29,610 test data observations that the model must classify. It would be interesting to monitor changes in the current model's performance with such data sets.

As illustrated throughout the paper, building a statistical model is a challenging task. While Carriquiry, et al. (2019) share their accomplishments in transforming evidence found at crime scenes to be analyzed digitally and quantitatively, they admit that there also remain obstacles in real-life application of classification modeling in criminal investigation processes. Even if there is plenty of data, a model must be tested repeatedly to ensure that it can function in real-life because there may be high costs associated with the model's misjudgments. Nevertheless, Alicia Carriquiry work is impressive in that her work has potential to minimize human errors and help those who may be wrongly convicted.

REFERENCES

Analytics Vidhya. (2021). *Metrics to Evaluate Your Classification Model to Take the Right Decisions.* https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/

Carriquiry, Hofmann, H., Tai, X. H., & VanderPlas, S. (2019). Machine learning in forensic applications. *Significance (Oxford, England)*, *16*(2), 29–35. https://doi.org/10.1111/j.1740-9713.2019.01252.x

Chauhan, N. (2022, February 9). Decision Tree Algorithm, Explained. *KDNuggets*. https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html

Chihara, & Hesterberg, T. (2019). Mathematical statistics with resampling and R (Second edition.). Wiley.

Cravit, R. (2021, August 03). What is Decision Tree and How to Make One. *Venngage*. https://venngage.com/blog/what-is-a-decision-tree/

Haavik, E. (2021, May 31). How Many Innocent People Are in US Prisons, and Why Can't We Find Them? *Kare11*. https://www.kare11.com/article/syndication/podcasts/record-of-wrong/how-many-innocent-people-are-in-us-prisons/89-cc7d1412-0eec-48af-9168-319f96f887dd

Hare, Hofmann, H., & Carriquiry, A. (2017). Automatic Matching of Bullet Land Impressions. The Annals of Applied Statistics, 11(4), 2332–2356. https://doi.org/10.1214/17-AOAS1080

James, Witten, D., Hastie, T., & Tibshirani, R. (2021). An Introduction to statistical learning : with applications in R (2nd ed.). Springer.

Kalkomey. *Rifling in the Handgun Bore*. https://www.handgunsafetycourse.com/handgun/studyGuide/Rifling-in-the-Handgun-Bore/601099_700077859/

Karabiber, F. (2020, October 30). Gini Impurity. *Learn Data Sci.* https://www.learndatasci.com/author/FatihKarabiber

Loazia, S. (2020, March 20). Gini Impurity Measure – A Simple Explanation Using Python. *Towards Data Science*. https://towardsdatascience.com/gini-impurity-measure-dbd3878ead33#:~:text=The%20Gini%20impurity%20measure%20is,root%20node%2C%20and%20subsequent%20splits.&text=To%20put%20it%20into%20context,the%20data%20into%20smaller%20groups.

The National Registry of Exonerations. (2019, July 11). *Robert Lee Stinson*. https://www.law.umich.edu/special/exoneration/Pages/casedetail.aspx?caseid=3666

Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel, O., … others. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830.
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Rattenbury, R. C. (2015, February 18). Semiautomatic Pistol. *Encyclopedia Britannica*. https://www.britannica.com/technology/semiautomatic-pistol

Tahsildar, S. (2019, April 18)Gini Index: Decision Tree, Formula, and Coefficient. *Quantinsti.* https://blog.quantinsti.com/gini-index/

UCLA: Statistical Consulting Group. (2022). *Logistic Regression Diagnostics.* https://stats.oarc.ucla.edu/stata/webbooks/logistic/chapter3/lesson-3-logistic-regression-diagnostics/

Python_Code_Markdown

February 28, 2022

[12]:
```
#load required libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪StratifiedKFold, cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, recall_score, precision_score,␣
 ↪f1_score, make_scorer
import warnings
#filter warning signs within code blocks
warnings.filterwarnings('ignore')
#customize output display on the notebook
pd.options.display.max_columns = None
pd.set_option('display.max_colwidth', None)

## resources referenced
#https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.
 ↪GridSearchCV.html
#https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.
 ↪RandomForestClassifier.html
#https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.
 ↪LogisticRegression.html
#https://scikit-learn.org/stable/modules/generated/sklearn.tree.
 ↪DecisionTreeClassifier.html
#http://seaborn.pydata.org/introduction.html
```

[2]:
```
#create a function that binarizes the target variable
def label_target(x):
    """
    input: a list of values
    output: return 1 if True, 0 if False
    """
    if x == True:
        return 1
```

```
    elif x == False:
        return 0
```

1 1. Data Preprocessing

- Import data set
- Preprocess data set to drop unnecessary columns and rename column headers
- Split training and test data sets

[3]:
```
#import data set
df_raw = pd.read_csv("bullet_stats.csv")
df_raw.head()
```

[3]:

	b2	b1	ccf	D	num.matches	num.mismatches	non_cms	sumpeaks	cms	data	resID	id.x	id.y	match
0	Br1 Bullet 1-1	Ukn Bullet C-2	0.296164	1.810042	8	22	5	7.146495	2	data-new-all-25-25/bullet1.RData	128	4	1	False
1	Br1 Bullet 1-1	Ukn Bullet S-5	0.263017	2.331692	4	19	11	4.847843	2	data-new-all-25-25/bullet1.RData	185	17	1	False
2	Br1 Bullet 1-1	Ukn Bullet D-1	0.266002	1.492308	10	17	9	7.358286	6	data-new-all-25-25/bullet1.RData	133	23	1	False
3	Br1 Bullet 1-1	Ukn Bullet L-4	0.303057	1.975364	2	20	7	1.429595	1	data-new-all-25-25/bullet1.RData	166	46	1	False
4	Br1 Bullet 1-1	Ukn Bullet H-6	0.226535	2.716547	2	19	14	1.528081	2	data-new-all-25-25/bullet1.RData	156	8	1	False

[4]:
```
#binarize target variable
df_raw['match'] = df_raw['match'].apply(lambda x: label_target(x))
#determine if b1 or is known
df_raw['b1_known'] = df_raw['b1'].apply(lambda x: 0 if x[:3].lower() == "ukn"␣
 ↪else 1)
df_raw['b2_known'] = df_raw['b2'].apply(lambda x: 0 if x[:3].lower() == "ukn"␣
 ↪else 1)
#determine if both bullets are known, as they will be used as training data set
df_raw['known_sum'] = df_raw['b1_known'] + df_raw['b2_known']
df_raw['both_known'] = df_raw['known_sum'].apply(lambda x: 1 if x == 2 else 0)
#select columns which will be used to split training and test data set
df_raw = df_raw[['b2', 'b1', 'b2_known', 'b1_known', 'both_known', 'ccf', 'D',␣
 ↪'num.matches', 'num.mismatches', 'non_cms', 'sumpeaks', 'cms', 'match']]
```

```
df_raw.columns = ['b2', 'b1', 'b2_known', 'b1_known', 'both_known', 'ccf', 'd',␣
↪'num_matches', 'num_mismatches', 'non_cms', 'sumpeaks', 'cms', 'match']
df_raw.head()
```

[4]:

	b2	b1	b2_known	b1_known	both_known	ccf	\
0	Br1 Bullet 1-1	Ukn Bullet C-2	1	0	0	0.296164	
1	Br1 Bullet 1-1	Ukn Bullet S-5	1	0	0	0.263017	
2	Br1 Bullet 1-1	Ukn Bullet D-1	1	0	0	0.266002	
3	Br1 Bullet 1-1	Ukn Bullet L-4	1	0	0	0.303057	
4	Br1 Bullet 1-1	Ukn Bullet H-6	1	0	0	0.226535	

	d	num_matches	num_mismatches	non_cms	sumpeaks	cms	match
0	1.810042	8	22	5	7.146495	2	0
1	2.331692	4	19	11	4.847843	2	0
2	1.492308	10	17	9	7.358286	6	0
3	1.975364	2	20	7	1.429595	1	0
4	2.716547	2	19	14	1.528081	2	0

[5]:
```
#split training and test data sets
df_train = df_raw[df_raw['both_known'] == 1]
df_test = df_raw[df_raw['both_known'] == 0]
#select only the columns which will be used for modeling
df_train = df_train[['ccf', 'd', 'num_matches', 'num_mismatches', 'non_cms',␣
↪'sumpeaks', 'cms', 'match']]
df_test = df_test[['ccf', 'd', 'num_matches', 'num_mismatches', 'non_cms',␣
↪'sumpeaks', 'cms', 'match']]
```

[6]:
```
#print shape of training and test data set
print(df_train.shape)
print(df_test.shape)
```

```
(14280, 8)
(29610, 8)
```

2 2. Exploratory Data Analysis

- EDA was conducted using only the training data set
- Since Logistic Regression assumes independence among predictor variables, correlation between the variables were calculated
- High correlated column was removed in both the training and test data sets

[8]:
```
# find if there is missing values
df_train.isnull().sum()
```

[8]:
```
ccf             0
d               0
num_matches     0
```

```
num_mismatches    0
non_cms           0
sumpeaks          0
cms               0
match             0
dtype: int64
```

[9]:
```
# distribution of match cases in training data sets
df_train.groupby(['match'])['match'].count()
```

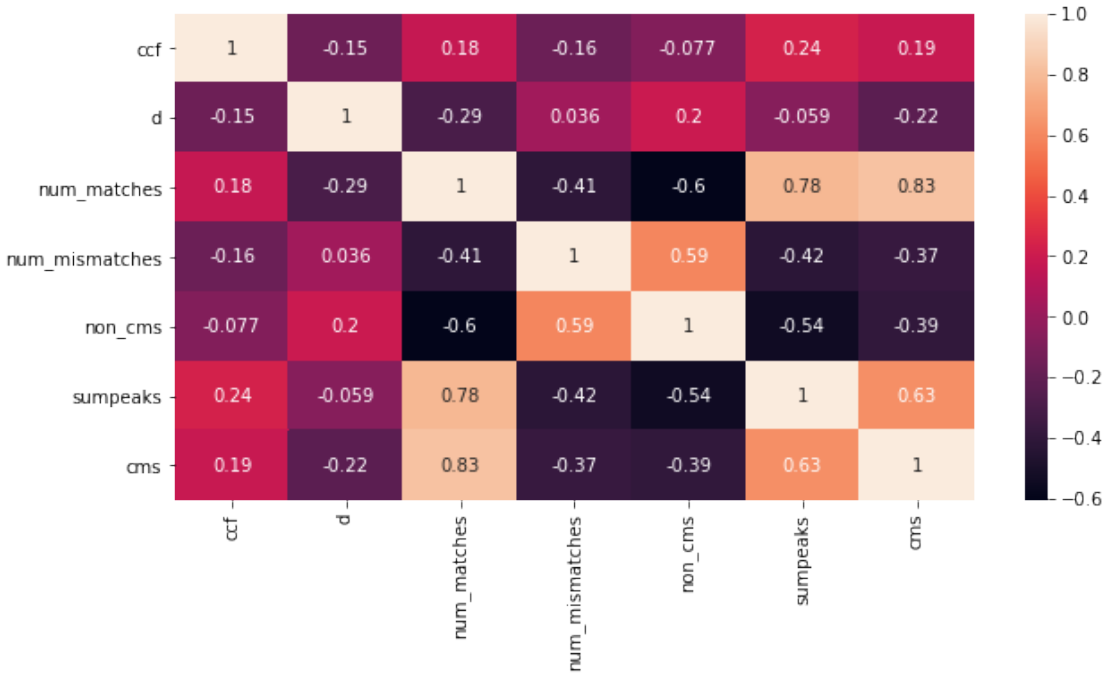
[9]:
```
match
0    14160
1      120
Name: match, dtype: int64
```

[10]:
```
# distribution of match cases in test data sets
df_train.groupby(['match'])['match'].count()
```

[10]:
```
match
0    14160
1      120
Name: match, dtype: int64
```

[15]:
```
#adjust size
plt.rcParams["figure.figsize"] = (10,5)
#create data set without the target variable for correlation test
corr_values = df_train[['ccf', 'd', 'num_matches', 'num_mismatches', 'non_cms',␣
 ↪'sumpeaks', 'cms']].corr()
#plot correlation heatmap. Documentation referenced https://seaborn.pydata.org/
 ↪generated/seaborn.heatmap.html
sns.heatmap(corr_values, annot = True, xticklabels = corr_values.columns,␣
 ↪yticklabels = corr_values.columns)
```

[15]:
```
<AxesSubplot:>
```

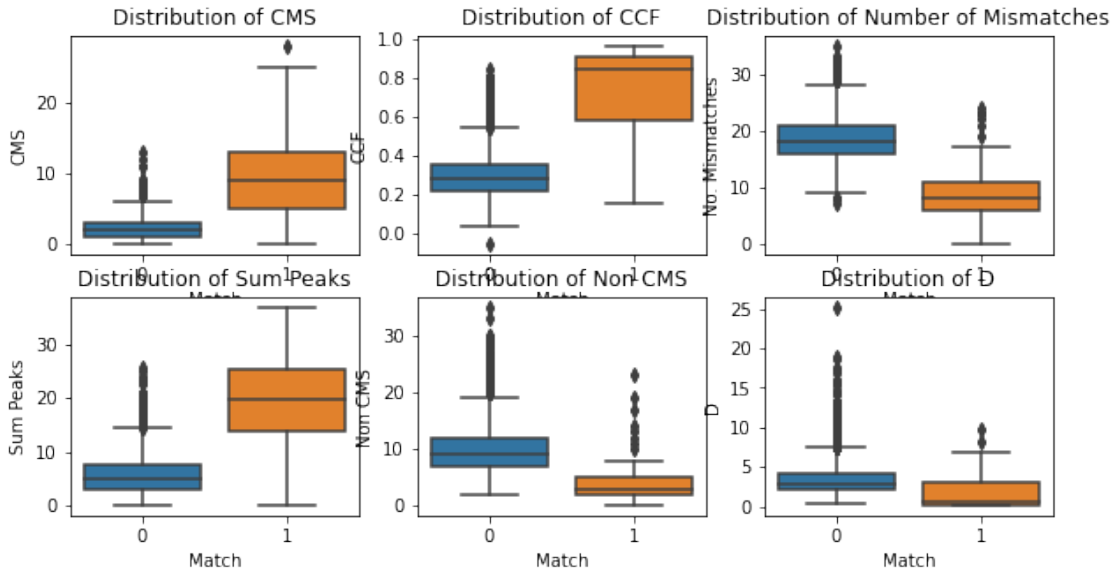


[16]:
```
#remove highly correlated column from the training and test data sets
df_train = df_train[['ccf', 'd', 'num_mismatches', 'non_cms', 'sumpeaks',␣
↪'cms', 'match']]
df_test = df_test[['ccf', 'd', 'num_mismatches', 'non_cms', 'sumpeaks', 'cms',␣
↪'match']]
```

[17]:
```
#create a subplot to visualize variable distributions with respect to each class
#documentation referenced for subplots: https://matplotlib.org/stable/api/
↪_as_gen/matplotlib.pyplot.subplots.html
fig, axs = plt.subplots(nrows=2, ncols=3)
sns.boxplot(x="match", y="cms", data=df_train, ax=axs[0][0]).
↪set(title="Distribution of CMS", xlabel="Match", ylabel="CMS") #cms
sns.boxplot(x="match", y="ccf", data=df_train, ax=axs[0][1]).
↪set(title="Distribution of CCF", xlabel="Match", ylabel="CCF") #ccf
sns.boxplot(x="match", y="num_mismatches", data=df_train, ax=axs[0][2]).
↪set(title="Distribution of Number of Mismatches", xlabel="Match", ylabel="No.
↪ Mismatches") #num_mismatches
sns.boxplot(x="match", y="sumpeaks", data=df_train, ax=axs[1][0]).
↪set(title="Distribution of Sum Peaks", xlabel="Match", ylabel="Sum Peaks")␣
↪#sumpeaks
sns.boxplot(x="match", y="non_cms", data=df_train, ax=axs[1][1]).
↪set(title="Distribution of Non CMS", xlabel="Match", ylabel="Non CMS")␣
↪#non_cms
```

```
sns.boxplot(x="match", y="d", data=df_train, ax=axs[1][2]).
 ↪set(title="Distribution of D", xlabel="Match", ylabel="D") #d
```

[17]: [Text(0.5, 1.0, 'Distribution of D'), Text(0.5, 0, 'Match'), Text(0, 0.5, 'D')]



3 3. Model Preparation

- Generate features and target data sets for both training and test data sets

[18]:
```
#divide feature and target values
x_train = df_train[['ccf', 'd', 'num_mismatches', 'non_cms', 'sumpeaks', 'cms']]
y_train = df_train[['match']]
x_test = df_test[['ccf', 'd', 'num_mismatches', 'non_cms', 'sumpeaks', 'cms']]
y_test = df_test[['match']]
```

4 4. Modeling

4.1 4.1 Decision Tree

- GridSearchCV was used to optimize hyperparameters (max_depth, min_samples_leaf, min_samples_split)
- Cross-validation was conducted with respect to F1 score

[19]:
```
# inputs for hyperparameters
params = {
    'max_depth': [i for i in range(0, 11) if i != 0],
    'min_samples_leaf': [i for i in range(0, 10) if i != 0],
```

```
    'min_samples_split': [i for i in range(0, 10) if i != 0]
}
# scores to record
scoring = {'accuracy': 'accuracy',
           'precision': 'precision',
           'recall': 'recall',
           'f1': 'f1'}
# generate a decesion tree model
decision_tree_classifier_1 = DecisionTreeClassifier(random_state=10)
# conduct GridSearchCV
dt_obj_1 = GridSearchCV(decision_tree_classifier_1,
                        #input test hyperparameters
                        param_grid = params,
                        scoring = scoring,
                        #after finding the best hyperparameter in terms of F1,␣
 ↪fit the model
                        refit = 'f1',
                        #stratified cross-validation since data is imbalanced
                        cv = StratifiedKFold(n_splits=5),
                        n_jobs = 6)
# fit the model
dt_obj_1.fit(x_train, y_train)
```

[19]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=5, random_state=None, shuffle=False),
             estimator=DecisionTreeClassifier(random_state=10), n_jobs=6,
             param_grid={'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                         'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                         'min_samples_split': [1, 2, 3, 4, 5, 6, 7, 8, 9]},
             refit='f1',
             scoring={'accuracy': 'accuracy', 'f1': 'f1',
                      'precision': 'precision', 'recall': 'recall'})
```

[44]:
```
# view results in descending order of f1 score
dt_obj_1_result = pd.DataFrame(dt_obj_1.cv_results_)[['params',␣
 ↪'mean_test_accuracy', 'mean_test_recall', 'mean_test_precision',␣
 ↪'mean_test_f1']].sort_values(by='mean_test_f1', ascending=False)
dt_obj_1_result.head()
```

[44]:

	params	mean_test_accuracy	mean_test_recall	mean_test_precision	mean_test_f1
717	{'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 7}	0.996849	0.65	0.964615	0.773366
716	{'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 6}				
715	{'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 5}				
714	{'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 4}				
713	{'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 3}				

```
716            0.996849             0.65             0.964615      0.773366
715            0.996849             0.65             0.964615      0.773366
714            0.996849             0.65             0.964615      0.773366
713            0.996849             0.65             0.964615      0.773366
```

[45]:
```
#find best parameter
dt_obj_1_result[dt_obj_1_result['params'] == dt_obj_1.best_params_]
```

[45]:
```
                                                    params  \
712  {'max_depth': 9, 'min_samples_leaf': 8, 'min_samples_split': 2}

     mean_test_accuracy  mean_test_recall  mean_test_precision  mean_test_f1
712            0.996849              0.65             0.964615      0.773366
```

[46]:
```
#make predictions
y_pred_dt = dt_obj_1.predict(x_test)
#view performance
print("accuracy: " + str(accuracy_score(y_test, y_pred_dt)))
print("precision: " + str(precision_score(y_test, y_pred_dt)))
print("recall: " + str(recall_score(y_test, y_pred_dt)))
print("f1_score: " + str(f1_score(y_test, y_pred_dt)))
```

```
accuracy: 0.9935832489023978
precision: 0.9230769230769231
recall: 0.6111111111111112
f1_score: 0.7353760445682452
```

4.2 4.2 Random Forest - Trial 1

- Test run Random Forest using just n_estimators hyperparameter

[26]:
```
# inputs for hyperparameters
params = {
    'n_estimators': [i for i in range(0, 301, 5) if i != 0],
}
# define scores to record
scoring = {'accuracy': 'accuracy',
           'precision': 'precision',
           'recall': 'recall',
           'f1': 'f1'}
# generate a random forest model
random_forest_classifier_1 = RandomForestClassifier(random_state=10)
# conduct GridSearchCV
grid_obj_1 = GridSearchCV(random_forest_classifier_1,
                          #input test hyperparameters
                          param_grid = params,
                          scoring = scoring,
```

```
                          #after finding the best hyperparameter in terms of␣
 ↪F1, fit the model
                          refit = 'f1',
                          #stratified cross-validation since data is imbalanced
                          cv = StratifiedKFold(n_splits=5),
                          n_jobs = 3)
#fit the model
grid_obj_1.fit(x_train, y_train)
```

[26]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=5, random_state=None, shuffle=False),
             estimator=RandomForestClassifier(random_state=10), n_jobs=3,
             param_grid={'n_estimators': [5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
                                          55, 60, 65, 70, 75, 80, 85, 90, 95,
                                          100, 105, 110, 115, 120, 125, 130,
                                          135, 140, 145, 150, …]},
             refit='f1',
             scoring={'accuracy': 'accuracy', 'f1': 'f1',
                      'precision': 'precision', 'recall': 'recall'})
```

[47]:
```
# view results in descending order of f1 score
grid_obj_1_result = pd.DataFrame(grid_obj_1.cv_results_)[['params',␣
 ↪'mean_test_accuracy', 'mean_test_recall', 'mean_test_precision',␣
 ↪'mean_test_f1']].sort_values(by='mean_test_f1', ascending=False)
grid_obj_1_result.head()
```

[47]:

	params	mean_test_accuracy	mean_test_recall	mean_test_precision	mean_test_f1
30	{'n_estimators': 155}	0.997759	0.733333	1.0	0.838923
45	{'n_estimators': 230}	0.997759	0.733333	1.0	0.838923
33	{'n_estimators': 170}	0.997759	0.733333	1.0	0.838923
34	{'n_estimators': 175}	0.997759	0.733333	1.0	0.838923
35	{'n_estimators': 180}	0.997759	0.733333	1.0	0.838923

[48]:
```
# find best scores
grid_obj_1_result[grid_obj_1_result['params'] == grid_obj_1.best_params_]
```

[48]:

	params	mean_test_accuracy	mean_test_recall	mean_test_precision	mean_test_f1
26	{'n_estimators': 135}	0.997759	0.733333	1.0	0.838923

[49]:
```
#make predictions
y_pred_1 = grid_obj_1.predict(x_test)
#view performance
print("accuracy: " + str(accuracy_score(y_test, y_pred_1)))
print("precision: " + str(precision_score(y_test, y_pred_1)))
print("recall: " + str(recall_score(y_test, y_pred_1)))
print("f1_score: " + str(f1_score(y_test, y_pred_1)))
```

```
accuracy: 0.9944613306315434
precision: 1.0
recall: 0.6203703703703703
f1_score: 0.7657142857142858
```

4.3 4.2 Random Forest - Trial 2

- Enhance optimization using more hyperparameters (n_estimators, max_depth, min_samples_leaf, min_samples_split

[31]:
```
# inputs for hyperparameters
params = {
    'n_estimators': [i for i in range(0, 201, 25) if i != 0],
    'max_depth': [i for i in range(0, 11) if i != 0],
    'min_samples_leaf': [i for i in range(0, 10) if i != 0],
    'min_samples_split': [i for i in range(0, 10) if i != 0]
}
# define scores to record
scoring = {'accuracy': 'accuracy',
           'precision': 'precision',
           'recall': 'recall',
           'f1': 'f1'}
# generate a second random forest model
random_forest_classifier_2 = RandomForestClassifier(random_state=10)
# conduct GridSearchCV
grid_obj_2 = GridSearchCV(random_forest_classifier_2,
                          #input test hyperparameters
                          param_grid = params,
                          scoring = scoring,
                          refit = 'f1',
                          #stratified cross-validation since data is imbalanced
                          cv = StratifiedKFold(n_splits=5),
                          n_jobs = 6)
grid_obj_2.fit(x_train, y_train)
```

[31]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=5, random_state=None, shuffle=False),
             estimator=RandomForestClassifier(random_state=10), n_jobs=6,
             param_grid={'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                         'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                         'min_samples_split': [1, 2, 3, 4, 5, 6, 7, 8, 9],
```

```
                        'n_estimators': [25, 50, 75, 100, 125, 150, 175, 200]},
              refit='f1',
              scoring={'accuracy': 'accuracy', 'f1': 'f1',
                       'precision': 'precision', 'recall': 'recall'})
```

[50]:
```
# view results in descending order of f1 score
grid_obj_2_result = pd.DataFrame(grid_obj_2.cv_results_)[['params',␣
↪'mean_test_accuracy', 'mean_test_recall', 'mean_test_precision',␣
↪'mean_test_f1']].sort_values(by='mean_test_f1', ascending=False)
grid_obj_2_result.head()
```

[50]:
```
                                                 params  \
4748  {'max_depth': 8, 'min_samples_leaf': 3, 'min_samples_split': 9,
'n_estimators': 125}
5366  {'max_depth': 9, 'min_samples_leaf': 3, 'min_samples_split': 5,
'n_estimators': 175}
4011  {'max_depth': 7, 'min_samples_leaf': 2, 'min_samples_split': 7,
'n_estimators': 100}
4012  {'max_depth': 7, 'min_samples_leaf': 2, 'min_samples_split': 7,
'n_estimators': 125}
4013  {'max_depth': 7, 'min_samples_leaf': 2, 'min_samples_split': 7,
'n_estimators': 150}

      mean_test_accuracy  mean_test_recall  mean_test_precision  mean_test_f1
4748            0.997759          0.733333                  1.0      0.838923
5366            0.997759          0.733333                  1.0      0.838923
4011            0.997759          0.733333                  1.0      0.838923
4012            0.997759          0.733333                  1.0      0.838923
4013            0.997759          0.733333                  1.0      0.838923
```

[51]:
```
# view best parameter
grid_obj_2_result[grid_obj_2_result['params'] == grid_obj_2.best_params_]
```

[51]:
```
                                                 params  \
1958  {'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 175}

      mean_test_accuracy  mean_test_recall  mean_test_precision  mean_test_f1
1958            0.997759          0.733333                  1.0      0.838923
```

[53]:
```
#predict outcomes
y_pred_2 = grid_obj_2.predict(x_test)
#view performance
print("accuracy: " + str(accuracy_score(y_test, y_pred_2)))
print("precision: " + str(precision_score(y_test, y_pred_2)))
print("recall: " + str(recall_score(y_test, y_pred_2)))
print("f1_score: " + str(f1_score(y_test, y_pred_2)))
```

```
accuracy: 0.9937858831475853
precision: 1.0
recall: 0.5740740740740741
f1_score: 0.7294117647058824
```

4.4 4.3 Logistic Regression

- Train a logistic regression without hyperparameter tuning

[57]:
```
# generate a model
logi_model = linear_model.LogisticRegression(random_state=10,␣
↪fit_intercept=True)
# scores to cross-validate
score_list = ['accuracy', 'precision', 'recall', 'f1']
score_dict = {}
# for loop to iterate and record each scores
for score_method in score_list:
    # conduct cross validation
    score_results = cross_val_score(logi_model, x_train, y_train, cv =␣
↪StratifiedKFold(n_splits=5), scoring=score_method)
    # store scores
    mean_score = score_results.mean()
    score_dict[score_method] = mean_score
```

[58]:
```
# view scores
score_dict
```

[58]:
```
{'accuracy': 0.9967787114845938,
 'precision': 1.0,
 'recall': 0.6166666666666666,
 'f1': 0.7576470588235296}
```

[59]:
```
# fit model
logi_model.fit(x_train, y_train)
```

[59]:
```
LogisticRegression(random_state=10)
```

[60]:
```
# predict outcomes
y_pred_log = logi_model.predict(x_test)
#view performance
print("accuracy: " + str(accuracy_score(y_test, y_pred_log)))
print("precision: " + str(precision_score(y_test, y_pred_log)))
print("recall: " + str(recall_score(y_test, y_pred_log)))
print("f1_score: " + str(f1_score(y_test, y_pred_log)))
```

```
accuracy: 0.9933806146572104
precision: 1.0
recall: 0.5462962962962963
f1_score: 0.7065868263473054
```

[ ]: