How do popular supervised machine learning classification algorithms compare in their implementation and efficacy, for a given dataset?

Rohit Prasad

Abstract

Supervised machine learning is the study of algorithms that build a model based on a set of labelled training data and make predictions on new data. Used in applications where hard-coding algorithms is infeasible, such as optical character recognition and spam filtering, machine learning conceptually "gives computers the ability to learn without being explicitly programmed" (Simon, 2013). Classification algorithms specifically, are used considerably in computer vision, handwriting recognition, document classification and statistical natural language processing. Recently, sentiment analysis of twitter data has become a ubiquitous enabling technology, having applications in political science and market research aplenty. Through linguistic analysis of tweets (including emoticons), essential features can be extracted allowing the tweet to be classified as positive, negative or neutral. (Pak and Paroubek). The purpose of this report is to compare the efficacy and implementation of three binary (2 classes) classification algorithms when implemented on an appropriate data set. Namely, these algorithms are Gaussian Naïve Bayes, Logistic Regression and Linear Discriminant Analysis, where Linear Discriminant Analysis is the most accurate and complex. Gaussian Naïve Bayes is the simplest in computational terms allowing it to execute quickly, but out of the three assessed, it is the least accurate. Logistic Regression is a better compromise between accuracy and time complexity.

1.1 Introduction

The research is focused on *supervised* ML ("machine learning" may be abbreviated as ML in this report) algorithms; there are input variables (x) and an output variable (Y), and the algorithm finds the mapping function (trains a model) that maps x to Y $(Y = f(x))$. Furthermore, to ensure that the algorithms are comparable, they all perform binary classification; the task of assigning *one of two* labels to new data, using a trained model.

After choosing an appropriate dataset, I will research, implement, evaluate and compare 3 classification algorithms, in order of complexity:

- Gaussian Naive Bayes
- Logistic Regression
- Linear Discriminant Analysis

There are two primary sources that I will use to research the algorithms, to facilitate understanding of the algorithm itself. The e-book - "Master Machine Learning Algorithms" by Jason Brownlee (Brownlee, 2015) is the first, introducing me to the basics of machine learning and highlighting the modelling assumptions, whilst also providing exemplar implementations. Dr. Jason Brownlee is "a machine learning specialist" (Jason Brownlee | LinkedIn, no date) with a Ph.D. in artificial intelligence. This e-book, being an unbiased, factual publication written by a qualified author deems it to be an extremely reliable source of information.

The second source is "Machine Learning: An Algorithmic Perspective" by Stephen Marsland (2014) (Marsland 2014), which will provide derivations and detailed explanations of all the algorithms thereby enhancing my comprehension of the algorithm's workings. Prof. Stephen Marsland is a professor of Computer Science whose area of expertise is Artificial intelligence and Image Processing. (Massey and Zealand, 1998). Having authored several journal articles on machine learning, and also several books it can be assumed that his book is an extremely reliable, unbiased source of information for this report.

As a preface to the report, the principle that underlies all three of the algorithms being investigated, must be discussed. ML algorithms learn a function/model f that maps variables/features x to an output variable Y. Initially the form of f is unknown; the model is learnt using the algorithm. To estimate f, we use the data available; this is called *training*. Finally, the trained model f is used to make predictions for new unlabelled data (x that have unknown Y). Each algorithm will have different form of model f , "making different assumptions about the structure of the function" (Brownlee) and different techniques will be used to train it.

1.2 Choosing a Dataset and Features

The choice of dataset will be critical for a successful evaluation of efficacy. There are three criteria that a dataset needs to meet for use in this investigation:

a. Must have output variable that is binary (for 2 class classification.)
b. Must have at least 100 objects (rows) to all for sufficient training of a model to avoid underfitting of data.
c. Must have more than 2 attributes to be correlated with each other.
d. Must be real-world to keep the investigation interesting.

A popular archive of open-source datasets for experimentation is the UCI Machine Learning Repository (UCI machine learning repository, no date) which maintains 360 datasets and has a searchable interface. Filtering by datasets tagged for classification purposes, the “Tennis Major Tournament Match Statistics Data Set” (UCI machine learning repository: Tennis Major Tournament match statistics data set, 2014) looks particularly interesting. With 42 attributes and at least 127 objects in each tournament set, this dataset meets all of the criteria. It contains details of all the matches of the major tournaments in 2013 for both men and women. This gives me ample data to train the models for the algorithms, and sufficient data to make predictions on.

The dataset contains data, for both men and women, concerning 4 major tournaments; US Open, Australian Open, Wimbledon and the French Open. In each set of data there are records for each match that was played in the tournament, stating the players that played, outcome of the game, number of unforced errors for each player, serving rates etc.

After observing the dataset for “Men’s Australian Open” and using prior knowledge of tennis, the features that will be most interesting to correlate, and make predictions for wins and losses will be the *unforced errors* for both players, as that can be a deciding factor in the outcome of a tennis match. Correlating the data for unforced errors of player 1 against unforced errors of player 2 shows some separability between the wins and the losses.

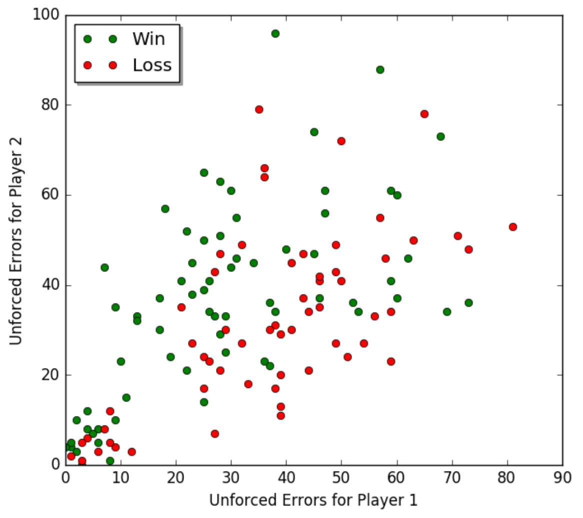
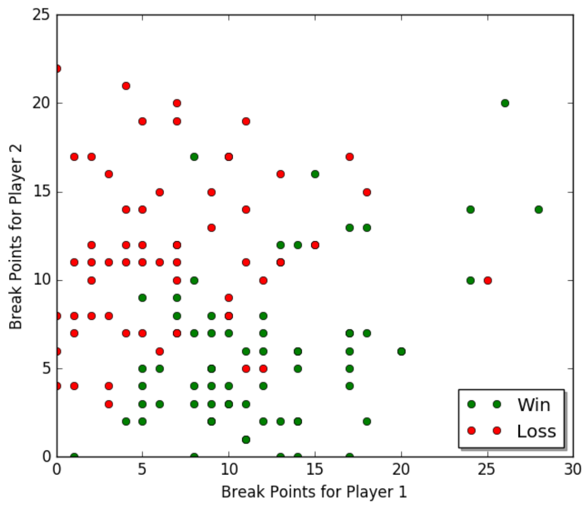


Figure 1 - Unforced Errors

Another pair of attributes that show excellent separability, making them good candidates for ML classification algorithms are the *break points* for player 1 and player 2.



Comparing the *aces* for player 1 and player 2 also

Figure 2 - Break Points

shows a scatter plot with decent separability, making this correlation an excellent candidate for binary classification.

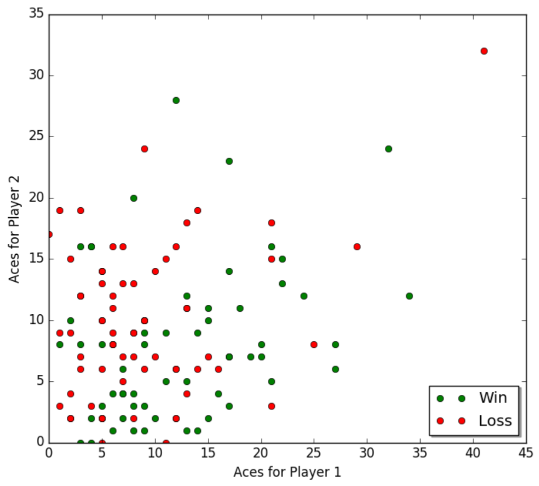


Figure 3 - Aces

Lastly, correlating the winners for player one against the winners for player two also shows excellent separability between the classes.

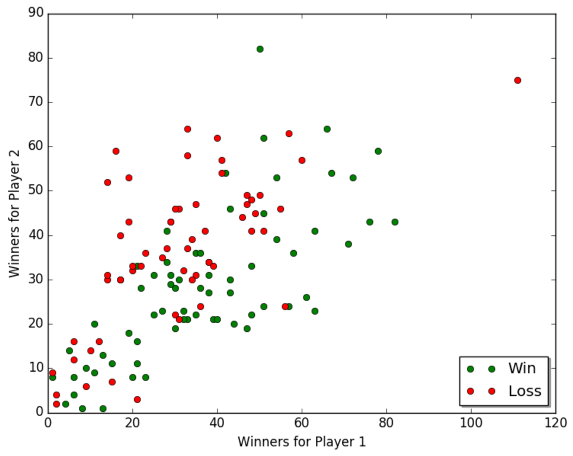


Figure 4 - Winners

When evaluating the algorithms, I will compare the performance of the algorithms when implemented to compare these 4 sets of features:

- Unforced Errors
- Break Points
- Aces
- Winners

The variance in separability of the classes (wins and losses) in all four correlations allows for a fair test of performance of the algorithms. They will be tested on weakly separated classes (unforced errors) and well separated classes (break points).

2.1 Gaussian Naïve Bayes – Research

Before attempting to understand Naïve Bayes, relevant concepts such as conditional probability and Bayes' Rule need exploration.

Conditional Probability

Independent events are those that are unaffected by other events, such as a coin flip. Dependent events can be affected by previous events. For example, from a bag containing 4 green balls and 3 red balls you remove one to find that it is green, and without replacement you remove one more. The probability that the second is green is affected by the result of the first event, making it *dependent*.

The probability of event A can be denoted by: $P(A) = x$, where x is the probability of event A. The probability of event B given A has already occurred is denoted by: $P(B|A) = x$. This is also called the *conditional probability* of B given A. The probability of event A and event B occurring is:

$$P(A\&B) = P(A) \times P(B|A)$$

Conjunctional Probability (1.0)

Bayes' Rule

Bayes' Rule is based on this concept of conditional probability, except it describes the probability of event A happening given that event B has occurred. Mathematically, it can be stated to be (Stuart and Ord, 1994):

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

Bayes' Rule (1.1)

A "classic example to understand Bayes' Rule" (A simple explanation of naive Bayes classification, 2012) is the probability of an arbitrary disease given a test is positive is the probability of the test being positive *given* that there is a disease multiplied by the probability of disease, divided by the probability of testing positive (with or without a disease).

Learning a Naïve Bayes Model for Categorical Data

Bayes' Rule allows you to predict an outcome given a single event has occurred. To predict an outcome, given that *multiple* other events have already occurred is an entirely different question, the mathematics of which is beyond the realms of this report. To resolve the complication, each other event can be assumed to be independent allowing

Bayes' Rule to expressed as the following, when given the occurrence of multiple other events.

$$P(A|B,C\dots) = \frac{P(B|A)\times P(C|A)\dots\times P(A)}{P(B,C)}$$

Naïve Bayes (1.2)

It is due to the assumption that the variables are independent of each other that brings Naïve Bayes its colloquial name "idiot Bayes". (Stuart and Ord, 1994)

Despite being a "strong assumption that is most unlikely in real data" (Brownlee, 2015, page 83), this approach bodes surprisingly accurately in most approaches. To learn a model, the *class probabilities* and *conditional probabilities* need to be calculated. Class probability is the frequency of instances in each class divided by the total number of instances. (Brownlee, 2015, page 83) – essentially the proportion of instances each of the two classes is of the whole set. For example, the probability that an instance belongs in class (c) 1 or class 0 (for binary classification) is:

$$P(c = 1) = \frac{count(c = 1)}{count(c = 0) + count(c = 1)}$$

Class Probabilities (1.3)

The conditional probabilities are the frequency of each attribute value for a given class value, divided by the frequency of instances within that class value. For example, take a $temperature$ (T) attribute with values of hot (H) and $cold$ (C), and the outcome (class) attribute had values of $go\ out$ (G) or $stay\ home$ (S). The conditional probability of each value of $temperature$ for each class will be calculated. The conditional probabilities of H would be ($\wedge\ means\ "and"$):

$$P(T = H|class = G) = \frac{count(T = H \wedge class = G)}{count(class = G)}$$

$$P(T = H|class = S) = \frac{count(T = H \wedge class = S)}{count(class = S)}$$

Conditional Probabilities (1.4)

Once a Naïve Bayes model is learned, a list of probabilities is stored (Brownlee, 2015, page 84), including the class probabilities (probability of each class in the training data set) and the conditional probabilities of each input value given each class value.

<u>Making Predictions with a Naïve Bayes Model</u>

Using the same example as above; if there was a new instance with the temperature hot, to predict whether the outcome is $stay\ home$ or $go\ out$, the formula in figure 1.2 will be used. The probability that the outcome is $go\ out$ given hot is:

$$go\ out = P(T = H|class = G)\times P(class = G)$$
$$stay\ home = P(T = H|class = S)\times P(class = S)$$

$$P(G|T = H) = \frac{go\ out}{go\ out + stay\ home}$$

Prediction Calculation for Naïve Bayes (1.5)

<u>Gaussian Naïve Bayes</u>

The previous example was limited to categorical data only, whereas the data that I will intend to use (discussed in section 1.2) is real valued.

Gaussian Naïve Bayes assumes a Gaussian distribution for the attributes. The Gaussian distribution is described by the bell-shaped curve defined by the probability density function. It is often used in the natural and social sciences to represent real-valued random variables with unknown distributions. (Casella and Berger, 2001). In the context of this algorithm, it assumes that the histogram of the input variables look somewhat like this curve (StatsDirect Limited, 2000):

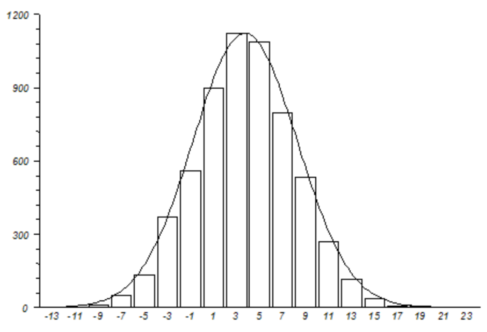


Figure 5 - Normal Distribution

If the histograms of the input variables do not follow a bell curve, the appropriate transformation (normally logarithm or square root) (Data transformation (statistics), 2016) can be applied to change the shape of the curve. To calculate the probability of a certain attribute being a certain value, the "Gaussian probability density function" is used (PDF) (Brownlee). It accepts three arguments, the value the probability is being calculated of, the mean of the data and the standard deviation:

$$pdf(x, mean, sd) = \frac{e^{-\left(\frac{(x-mean)^2}{2\times sd^2}\right)}}{\sqrt{2\pi}\times sd}$$

Gaussian Probability Density Function (1.6)

Adapting the examples in figure 1.5 to real valued attributes for *Gaussian* naïve Bayes.

$$go\ out = P(pdf(temperature)|class = G) \times P(class = G)$$

Gaussian Naïve Bayes Conditional Probability Formula (1.7)

Working with the PDF instead of the categorical probabilities will allow for the use of real valued attributes in Naïve Bayes, concluding the research on this algorithm.

2.2 Gaussian Naive Bayes – Implementation

Data Extraction

The first step in implementing any ML classification algorithm is extracting the specific data required from the data set, and preparing it to train a model. Since I will be comparing the performance of the algorithm on 4 pairs of attributes (4 attributes for player 1 and player 2), it would make sense to generalise the data extraction to any two parameters. The data for the first parameter, second parameter and the result (win/loss) can be each added to an array respectively.

```
parameter1 = 'UFE.1'
parameter2 = 'UFE.2'
par1, par2, result = [], [], []
f = '../datasets/AusOpen-men-2013.csv'
with open(f, 'r') as data:

    reader = csv.reader(data)
    for row in reader: header = row; break

    par1_index = header.index(parameter1)
    par2_index = header.index(parameter2)
    result_index = header.index('Result')

    for row in reader:
        par1.append(row[par1_index])
        par2.append(row[par2_index])
        result.append(row[result_index])
```

Snippet 1 - Data Extraction

There are 126 elements in each array. The model needs to be trained and then tested by making predictions on new data, meaning that data set needs to split into a *training set* and a *testing set*.

```
x1 = [int(x) for x in par1[0:74]]
x2 = [int(x) for x in par2[0:74]]
y = [int(y) for y in result[0:74]]
```

Snippet 2 - Training Data Set

Finally, both x_1 and x_2 need to be split into two arrays each by wins and losses, leaving four arrays:

- win_{x_1}
- win_{x_2}
- $loss_{x_1}$
- $loss_{x_2}$

```
win_x1, win_x2 = [], []
loss_x1, loss_x2 = [], []
for index in range(len(x1)):
    if y[index] == 1:
        win_x1.append(x1[index])
        win_x2.append(x2[index])
    else:
        loss_x1.append(x1[index])
        loss_x2.append(x2[index])
```

Snippet 3 - Sort Features by Result

Required Functions

The first, most important function required is the probability density function (PDF) to estimate the probability of x_1 or x_2 being a certain value given a mean and standard deviation. It is the same as the formula in figure 1.5.

```
def pdf(x, mean, sd):
    return (e**(-((x-mean)**2)/(2*(sd**2))))\
          / (((2*pi)**0.5)*sd)
```

I will also need to calculate an average and standard deviation of the data, which are both standard formulae in statistics. Standard deviation quantifies the amount of variation or dispersion of a set of data values. (Bland and Altman). It is the square root of the variance of the data.

```
def standard_deviation(data):
    squared_data = [x**2 for x in data]
    mean_squared_data = sum(squared_data) / \
                        len(squared_data)
    mean_data = sum(data) / len(data)
    variance = mean_squared_data - (mean_data ** 2)
    return variance**0.5

def avg(data):
    return sum(data) / len(data)
```

Snippet 4 - Standard Deviation and Mean

Training the Gaussian Naïve Bayes Model

The next step is to calculate the mean and standard deviations of each of the four arrays (win_{x_1}, win_{x_2} etc.), for use in the PDF. The functions for mean and standard deviation written in the previous section will be used here, saving each value into a separate variable.

```
win_x1_m = avg(win_x1)
win_x2_m = avg(win_x2)
loss_x1_m = avg(loss_x1)
loss_x2_m = avg(loss_x2)

win_x1_std = standard_deviation(win_x1)
win_x2_std = standard_deviation(win_x2)
loss_x1_std = standard_deviation(loss_x1)
loss_x2_std = standard_deviation(loss_x2)
```

Snippet 5 - Means and Standard Deviations of Data

The final aspects of the Naïve Bayes model are the class probabilities; the probability that any item of data is of a particular class. Their calculation is as straightforward as it sounds – the proportion of all the 126 items that are losses, and the proportion of all the 126 items that are wins.

```
p0 = len(loss_x1) / (len(win_x1) + len(loss_x1))
p1 = len(win_x1) / (len(win_x1) + len(loss_x1))
```

Snippet 6 - Class Probabilities

Transforming the Data to a Gaussian Distribution

Now that the model is ready to be trained, I have to ensure that the data being used follows a *normal distribution,* or is as close to one as possible. To test the model, I will use the Men's Australian Open dataset, and initially compare the unforced errors for player 1 with unforced errors for player 2 (see page 2 figure 1).

Experimenting with the histograms of each attribute for unforced errors shows that the transformation that best normalises the data is raising every value to the same power. Instead of raising every value to the same power once and "hoping" this provides the best accuracy of predictions, I can simply repeat the experiment for hundreds of different powers, and find which one yielded the best accuracy thereby telling me the best transformation.

```
transformation = 0.01
for x in range(150):
    x1 = [int(x)**transformation for x in par1[0:74]]
    x2 = [int(x)**transformation for x in par2[0:74]]
    ...
    transformation+=0.01
```

Snippet 7 - Repeating experiment for different transformations

Making Predictions

The final step is to use the Naïve Bayes model created from the training data to make predictions on the testing data. The testing inputs will need to be separated into different arrays (x_1 and x_2) and so will the result. Then, the probability that a data point is of a certain class (win or $loss$) will be generated using the formula for Naïve Bayes in figure 1.7. The values calculated in snippet 5 will be used in the PDF function. If the probability that the data point is of class 1, then a 1 will be appended to a predictions list, else a 0 will be appended. To find the accuracy of the predictions, this list will be compared one to one with the actual results in y_test.

```
predictions = []
x1_test = [int(x)**transformation for x in par1[75::]]
x2_test = [int(x)**transformation for x in par2[75::]]
y_test = [int(y)**transformation for y in result[75::]]

for item1, item2 in zip(x1_test, x2_test):
    c0 = pdf(item1, loss_x1_m, loss_x1_std)\
         * pdf(item2, loss_x2_m, loss_x2_std)\
         * p0
    c1 = pdf(item1, win_x1_m, win_x1_std)\
         * pdf(item2, win_x2_m, win_x2_std)\
         * p1
    if c1 > c0:
        predictions.append(1)
    else:
        predictions.append(0)
```

Snippet 8 - Making predictions using Naïve Bayes

2.3 Gaussian Naive Bayes – Evaluation

Gathering Accuracy Data

To make full use of the data available, I should repeat the algorithm on all datasets, saving the accuracy value for the dataset in an array.

As mentioned in section 1.2, the four attributes that I will compare for player 1 and player 2 are (in ascending order of class separability):

- Unforced Errors (weak separability)
- Aces (medium separability)
- Winners (good separability)
- Break Points Won (excellent separability)

It is safe to assume that there will be higher accuracy when testing the algorithm on "break points won", as the classes are well separated. It may perform most weakly on unforced errors as the classes are not well separated. There is missing data for unforced errors and winners for US Open Women, but since this will affect all algorithms, it will not be detrimental to the final comparison.

There are some missing values dotted around the dataset for certain attributes, but Naïve Bayes is unaffected by missing data. (Brownlee, 2014)

Finally executing the algorithm on all the datasets yields these results:

Dataset	Prediction Accuracy (%)			
	UFE	ACE	WNR	BPW
Wim-M	73.3	66.7	71.1	88.8
Wim-W	64.6	68.1	72.9	91.7
Aus-M	66.0	64.0	66.0	88.0
Aus-W	78.0	62.5	70.0	86.0
Fre-M	64.0	70.0	68.0	88.0
Fre-W	68.0	64.4	62.0	72.0
US-M	-	64.0	-	78.0
US-W	66.7	63.3	66.7	96.7

Table 1 - Gaussian Naive Bayes Accuracy

As expected, correlating the break points won for player one against player two turned out to be an excellent indicator for the player of the match with an average prediction accuracy of 86.2%. The average accuracy for winners was 68.1%, and for aces it was 65.3%. The final indicator for the winner of a match was the number of unforced errors for both players, at 60.1% average prediction accuracy. After looking at the scatter plot for unforced errors in the Australian Open dataset, it did not look like there was good separability between the classes and the results confirmed this hypothesis. datasets. The algorithm worked, as these accuracies are far better than a random selection for the classes (50%) which is the bare minimum accuracy any ML binary classification algorithm has to achieve! Its efficacy in determining the class of any data point can only be discussed after the evaluation of the next two algorithms, but without consideration of the upcoming algorithms, Gaussian Naïve Bayes was a very good predictor for the winner of a tennis match based only on the number of break points won for both players and a relatively good predictor using only winners.

The complete code for Gaussian Naïve Bayes can be found in appendix item A. The parameters to be compared are chosen at the top of the program, and the results are logged to the console.

3.1 Logistic Regression – Research

Regression

The concept of a *regression problem* in statistics must first be clarified. Take, for example these data points (Marsland):

x	y
0	0
0.52	1.5
1.05	-2.59
1.57	3
2.09	-2.59
2.61	1.5
3.14	0

Table 2 - Arbitrary Data

If one were to identify the y value for $x = 0.44$, the examples in table 2 could not be used directly because $x = 0.44$ is not present. There would need to be some way to *predict* the y value. "If you assume that the values come from sort of function and try to find out what the function is, then you can give the output value y for any value of x" (Marsland 2014, page 9). This is a *regression problem*. The problem is to work out what function to use.

With respect to the problem of classification, it is similar to regression in the sense that it calculates the *decision boundaries* (Two Feature Data, no date) that can be used to separate the classes.

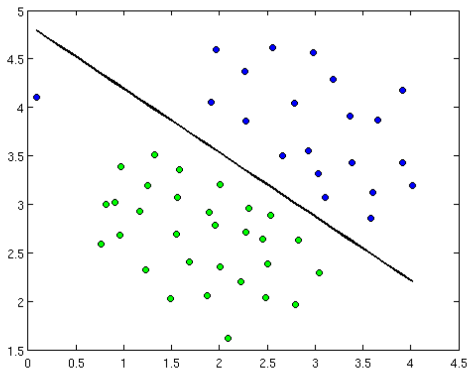


Figure 6 - Decision Boundary (Black line)

If you have the function for the decision boundary, then it will be possible to classify the data with high accuracy.

Logistic Regression

Instead of an algorithm outputting '1' or '0' predictions for the classification of a data point, it would be far more useful if probabilities (like in Naïve Bayes) were outputted. (Logistic Regression:

Carnegie Mellon University) For example, a model that tells us that there is a 51% chance that it snows and it doesn't snow, is better than a model that predicted a 99% chance of snow. Again, similar to Naïve Bayes, these are conditional probabilities, the probability of a data point being a certain class based on the datum: $P(Y = 0 \mid X_1 = x_1, X_2 = x_2)$.

The obvious way that we can calculate the probability that the data point is of a certain class, is by letting the probability be a linear function of x_1 and x_2, since there are 2 attributes to be considered for the chosen dataset:

$$p(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \ (1)$$

Every x value will add or subtract to the probability proportionally. But there is a problem with this, probabilities can only be between 0 and 1, whereas a linear function can assume any value! Logically, to fix this issue, instead of every change of x adding or subtracting to the probability, it can *multiply* the probability proportionally, essentially letting $\log(p(x_1, x_2))$ be a linear function of x:

$$\log(p(x_1, x_2)) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \ (2)$$

However, the issue with this is that logarithms are only "unbounded in one direction" (Logistic Regression: Carnegie Mellon University). Their minimum value is 0, but there is no maximum (which should be 1 for this problem). Finally, the simplest modification of the function would be the logit transformation (Grace-Martin, 2008):

$$\log\left(\frac{p(x_1, x_2)}{1 - p(x_1, x_2)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \ (3)$$

This limits the probability to a minimum of 0 and a maximum of 1. The train of thought is best illustrated graphically (Desmos, 2017):

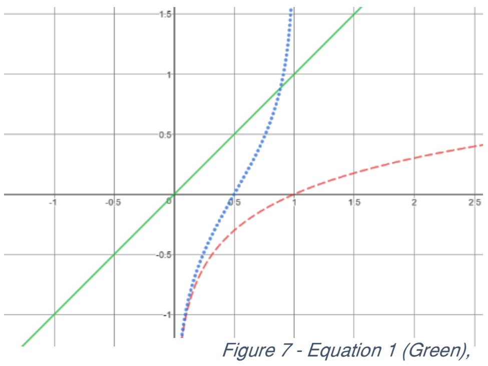


Figure 7 - Equation 1 (Green), Equation 2 (Red), Equation 3 (Blue)

Equation 3 is the model for logistic regression. Rearranging the equation to make $p(x_1, x_2)$ the subject (Logistic Regression: Carnegie Mellon University):

$$p(x_1, x_2) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

Logistic Regression Probability Function (1.8)

<u>Training a Logistic Regression Model</u>

Unlike Gaussian Naïve Bayes where the unique characteristics of the model were means, standard deviations and probabilities for both classes, in logistic regression, the parameters that will uniquely identify the model are β and β_0. These are the coefficients that the ML algorithm will have to learn. To calculate the probability, these values will be plugged into the equation in figure 1.8. To convert the probability into a "crisp prediction" (Brownlee), values above 0.5 are converted to 1 and values below 0.5 are converted to 0.

The coefficients of the logistic regression algorithm are estimated from the training data using a "stochastic gradient descent" (Brownlee) An ideal model with optimal coefficients would output a probability very close to 1, if the data point is in class 1, and a value very close to 0 for the other class. Stochastic gradient descent for logistic regression is a *search* procedure that "seeks values for the coefficients that minimize the error in the probabilities predicted by the model to those in the data" (Brownlee, 2015, page 53).

Stochastic gradient descent for logistic regression is an optimization procedure that "seeks values for the coefficients that minimize a "cost" function. It is used when the parameters cannot be calculated analytically. The derivation of the gradient descent algorithm is beyond this project, but the intuition for gradient descent is to imagine the plot of the cost function to be like a large bowl. "A random position on the surface of the bowl is the cost of the current values of the coefficients" (Brownlee, 2016). The bottom of the bowl is the cost of the best set of coefficients. Different values of coefficients are evaluated for cost, and are then adjusted for a lower cost. With enough repetition the minimum of the cost can be attained, with the optimal coefficients.

The coefficients are initialised at: $\beta_0 = 0$, $\beta_1 = 0$ and $\beta_2 = 0$. The way in which coefficients will be updated, will be using this "delta function" (Brownlee) for stochastic gradient descent:

$$\beta = \beta + \alpha \times (y - p(x_1, x_2)) \times p(x_1, x_2) \times (1 - p(x_1, x_2)) \times x$$

"Delta" Function (1.9)

α is the parameter that controls how much the model learns for each update of x_1 and x_2. For β_0, since there is no corresponding x coefficient, it is multiplied by a $bias$ variable, initialised at 1. If the values of x_1 and x_2 are relatively large, then α would need to be smaller. After all the values of x_1 and x_2 are used to update the function, the entire process is repeated again, but this time initialising the coefficients at their latest value, instead of 0. One repetition of the process is called an "epoch" (Brownlee), and the accuracy of the model will increase over several epochs.

One further change that may need to be made to the training data is to remove outliers; the values on the scatter plot that are anomalous.

3.2 Logistic Regression – Implementation

Data Extraction

The first step is to extract the required features from the data set. However, this part of the algorithm is identical to implementation for Gaussian Naïve Bayes. Refer to Snippet 1 – Data Extraction and Snippet 2 – Training Dataset on page 5 for the code.

Variable Initialisation

The coefficients that uniquely identify the model that is about to be learned are initialised at 0. The learning rate, α, is initialised at 0.01 because all the training data is relatively large in value. Bias is defaulted at 1, and the number of $epochs$ is set to 10; the number of times the model will use the training data to update the coefficients. Training accuracy keeps track of the accuracy of the model after each epoch.

```
B0, B1, B2 = 0, 0, 0
alpha = 0.01
bias = 1
epochs = 10
training_accuracy = []
```

Snippet 9 - Variable Initialisation (Logistic Regression)

Logistic Function

The logistic function from figure 1.8 is shown below.

```
def logistic_function(B0, B1, B2, x1, x2):
    probability = 1 / (1 + (e ** (-(B0 +
                                    (B1 * x1)
                                    + (B2 * x2)
                                    ))))
    return probability
```

Snippet 10 - Logistic Function

Coefficient Optimization

The first definite iteration repeats the entire procedure for the specified number of epochs. The nested definite iteration iterates over the data items. Within this iteration, the prediction is calculated using the logistic function, entering the values for β_0, β_1, β_2, x_1 and x_2.

Next, the prediction is converted to a crisp prediction of 0 or 1, which is then compared to the actual value. If correct, it is logged to a counter variable. Following that, the delta coefficient is calculated using the part of the delta function that is common for coefficients β_0, β_1 and β_2, in figure 1.9. Finally, the coefficients are updated using the rest of the delta function and the accuracy is appended to the training accuracy array.

```
for repeat in range(epochs):
    total_correct = 0
    for index in range(len(x1)):
        prediction = logistic_function(B0, B1, B2,
                                       x1[index],
                                       x2[index])

        if prediction < 0.5: sharp_prediction = 0
        else: sharp_prediction = 1
        if sharp_prediction == y[index]:
            total_correct += 1

        delta_coefficient = (alpha
                             * (y[index] -
                                prediction)
                             * prediction
                             * (1 - prediction))

        #Update Coefficients.
        B0 += delta_coefficient * bias
        B1 += delta_coefficient * x1[index]
        B2 += delta_coefficient * x2[index]

    training_accuracy.append(((total_correct
                              / len(x1)) * 100))
```

Snippet 11 - Logistic Regression Coefficient Optimization

Testing the model

The model is then tested using the learned coefficients on the testing portion of the dataset. Predictions are made using the logistic function, and are converted to crisp predictions thereafter.

```
predictions = []
x1_test = [int(x) for x in par1[split+1::]]
x2_test = [int(x) for x in par2[split+1::]]
y_test = [int(y) for y in result[split+1::]]
for item1, item2 in zip(x1_test, x2_test):
    prediction = logistic_function(B0, B1, B2,
                                   item1,
                                   item2)

    # Calculate Accuracy
    if prediction < 0.5:
        predictions.append(0)
    else:
        predictions.append(1)
```

Snippet 12 - Testing the Logistic Regression Model

The entire process was repeated for all the datasets, and all the features to be tested (break points won, aces, unforced errors and winners). For each dataset, it was repeated for 1000 α values, and the α value that provided the best accuracy was used.

3.3 Logistic Regression – Evaluation

Gathering Accuracy Data

As mentioned in section 1.2, the four attributes that I will compare for player 1 and player 2 are (in ascending order of class separability):

- Unforced Errors (weak separability)
- Aces (medium separability)
- Winners (good separability)
- Break Points Won (excellent separability)

Dataset	Prediction Accuracy (%)			
	UFE	ACE	WNR	BPW
Wim-M	68.9	66.7	77.8	93.3
Wim-W	60.4	63.8	66.7	91.7
Aus-M	72.0	64.0	72.0	88.0
Aus-W	80.0	62.5	60.0	86.0
Fre-M	60.0	68.0	68.0	88.0
Fre-W	68.0	68.0	58.0	76.0
US-M	-	64.0	-	74.0
US-W	70.0	63.3	66.7	83.3

Table 3 - Logistic Regression Accuracy

Again, as expected, the attribute that had the highest average prediction accuracy at 85.0% was break points won. Next best predictor for the winner of a tennis match was surprisingly unforced errors, with an average of 68.4%. The average prediction accuracy for number of winners was 67.0% and for number of aces it was 65.2%. The accuracy attained for unforced errors, aces and winners is a huge improvement over the random selection of a winner (50%), and considering the amount of noise and highly correlated inputs in the dataset the numbers show that logistic regression is also a good predictor for the winner of a tennis match.

The complete code for Logistic Regression can be found in appendix item B. The parameters to be compared are chosen at the top of the program, and the results are logged to the console.

4.1 Linear Discriminant Analysis – Research

Intuitive Explanation

Credited to R.A. Fisher, LDA is most commonly used as a “dimensionality reduction technique”; the goal is to project a whole dataset into a “lower dimensional space” in which there is good class separability. Let there be two input variables and one output variable, like the case with the Tennis Major Tournaments dataset. The fact there are two input variables means that the data is 2 dimensional, and we want to project the data into a single dimension i.e. a line which shows good separability in the classes. It is best explained graphically.

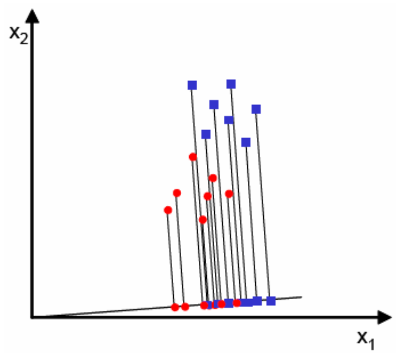


Figure 8 - LDA Projection (Bad Class Separability)

All the data is projected onto the black line in figure 8, but the line chosen does not allow for good separability of the classes. The reds and blues mix up nearer the middle.

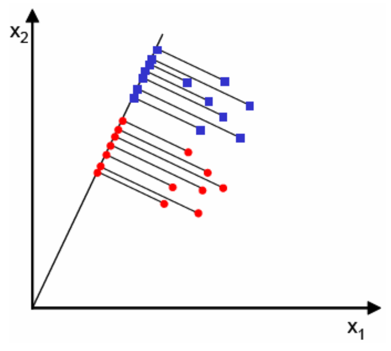


Figure 9 - LDA Projection (Good Class Separability)

When projected on to the line in figure 9, however, shows excellent separability between the classes. In this way, LDA is excellent in reducing the complexity of the problem from two dimensions to a single dimension. The mathematics of the algorithm is beyond this project, but to aid the implementation, the process in which the model is trained, and predictions are made can be explained.

Feature Derivation

There are two methods that can be used to implement linear discriminant analysis; *feature selection* and *feature derivation.* Feature selection involves iterating through the available features and discerning whether or not they're actually useful. (Marsland 221). The method that I will use will be feature derivation, which "derives new features from the old ones …by applying transforms to the dataset that change the axis of the graph by moving and rotating them." (Marsland 221) Essentially, this means the coordinate system of the data is rotated and moved to the perspective of the line onto which data is projected.

Training an LDA Model

A matrix X with 2 columns is populated with the feature data (x_1 and x_2). Each row stands for one object, and each column stands for one feature. A second matrix Y with a single column populated with the output data ($wins$ and $losses$). Two more matrices, X_0 and X_1 are populated with the feature data for their respective classes. The means μ_0 and μ_1 are calculated for matrices X_0 and X_1 respectively, and the mean μ is calculated for matrix X.

Next, a value called the *covariance* (Σ) is calculated for matrices X_0 and X_1. The *variance* of a set of numbers measures how spread out the values are (sum of squared distances between each element of the set and the mean of the set) (Marsland 174). This looks at the variation of *one* variable with respect to its mean. The covariance "looks at how two variables vary together" (Marsland 174). For example, the two variables increase and decrease at the same time, then the covariance is positive. The covariance is used to look at the correlation between all pairs of variables within a set of data (Marsland 175). The covariance of each pair is computed and inserted into a covariance matrix. The formula for covariance of X_1 and X_2:

$$\sum_j (X_j - \mu)(X_j - \mu)^T$$

Covariance (2.0)

The covariance matrices C_0 and C_1 are calculated for matrices X_0 and X_1 respectively. The covariance matrices tell us about the scatter of the data within a dataset – "the amount of spread there is within the data" (Marsland 223). The scatter is found by multiplying the covariance matrices C_0 and C_1 by p_0 and p_1 respectively. p_0 is the class probability for class 0 and p_1 is the class probability for class 1. The formula for class probability can be found in figure 1.3. Adding all the values gives a measure of the *within class scatter* (S_w) of the dataset. (Marsland 223).

$$S_w = \sum_{classes} \sum_{j \in c} p_c (x_j - \mu_c)(x_j - \mu_c)^T$$

Within Class Scatter (2.1)

For this dataset, S_w is the scatter for class 1 plus the scatter for class 0.

$$S_w = S_0 + S_1$$

If the data is easy to separated, then the within class scatter should be small meaning each class is tightly clustered together.

Next, the *between class scatter* (S_B) is calculated. This is an indication of how separable the classes are (distance between the classes.) For this reason, it only looks at the difference between the means.

$$S_B = \sum_{classes\ c} (\mu_c - \mu)(\mu_c - \mu)^T$$

Between Class Scatter (2.2)

If the classes are easy to separate, $\frac{S_B}{S_w}$ should be as large as possible. (Marsland 224).

To compute the LDA projection, this equation is solved for λ:

$$S_w^{-1} S_B Y = \lambda Y$$

Finding the Projection Vector (2.3)

This is an eigenvalue problem, and is solved by finding the eigenvalues and eigenvectors of $S_w^{-1} S_B$. Its solution will include two eigenvectors, but the one with the higher eigenvalue will be the projection vector. This eigenvector can be plotted onto a graph.

Making Predictions using LDA

To make predictions, one extra matrix is required – the *pooled within group covariance matrix*. (Teknomo, 2016). The formula for this is:

$$C_{pg}(r,s) = \frac{1}{n}\sum_{i=1}^{2} n_i C_i(r,s)$$

Pooled Within Group Covariance Matrix (2.4)

Any object k $(x_1 \quad x_2)$ is assigned to class i so that it has a maximum f_i (**Error! Bookmark not defined.**), where f_i is the discriminant function.

$$f_i = \mu_i C^{-1} x_k^T - \frac{1}{2}\mu_i C^{-1}\mu_i^T + \log(p_i)$$

Discriminant Function (2.5)

The discriminant value will be calculated for both values for i i.e. classes (class 0 and class 1). The class with the higher value will be the prediction.

4.2 Linear Discriminant Analysis – Implementation

Data Extraction

The data extraction snippet is similar, but not the same as Gaussian Naïve Bayes and Logistic Regression, due to the extensive use of matrices. Both the features x_1 and x_2 are inserted into the same array, instead of separate arrays.

```
with open(filename, 'r') as data:

    reader = csv.reader(data)
    for row in reader: header = row; break
    par1_index = header.index(parameter1)
    par2_index = header.index(parameter2)
    result_index = header.index('Result')

    for row in reader:
        if row[par1_index] != ''
       and row[par2_index] != '':
            dataset_x.append([int(row[par1_index]),
                              int(row[par2_index])])
            dataset_y.append(int(row[result_index]))
```

Snippet 13 - Data Extraction LDA

Next, the data set must be spliced to create the dataset used for training. 60% of the dataset will be used, the index of which, will be calculated and stored in $split$. After the training dataset is formed, a power transformation is applied to make the data more normally distributed (the procedure is repeated for 500 different powers and the best accuracy is picked). The regular python array is converted to a *numpy* (NumPy – NumPy, no date) array. Numpy is used because the library makes matrix operations such as dot product, inverse, addition etc. far simpler to execute. The training data set is also separated by wins and losses.

```
split = ceil(len(dataset_x) * 0.6) - 1
training_x =[[i[0]**transformation,
              i[1]**transformation]
             for i in dataset_x[0:split]]
training_y = dataset_y[0:split]
x = np.array(training_x)
y = np.array(training_y)
x1 = []
x2 = []
for num, yi in enumerate(y):
    if yi == 0:
        x1.append(x[num])
    else:
        x2.append(x[num])
```

Snippet 14 - Training Data Extraction and Numpy Conversion

The next step is to calculate the means of the matrices for wins, losses and the entire training dataset. Numpy has mean function that can be used for this. (Numpy.mean – NumPy v1.12 manual, 2008)

```
u1 = np.mean(x1, axis=0)
u2 = np.mean(x2, axis=0)
u = np.mean(x, axis=0)
```

Snippet 15 - Means of Feature Data

To standardize the data, the means are subtracted from the feature data x_1 and x_2.

```
x01 = x1 - u
x02 = x2 - u
```

Snippet 16 - Standardized Feature Data

The next step is the calculation for the covariance matrices using the formula in figure 2.0 on page 11.

```
c1 = np.dot(x01.transpose(), x01) / x01.shape[0]
c2 = np.dot(x02.transpose(), x02) / x02.shape[0]
```

Snippet 17 - Covariance Matrices

Using the formulae in figures 2.1 and 2.2, the within class and between class scatter can be found (cw and cb). To solve the equation in figure 2.3, the inverse of within class scatter also needs to be calculated ($cw_{inverse}$). Finally, the eigenvectors and eigenvalues are calculated using the linear algebra module within numpy, by passing the scalar product of inverse within class scatter and between class scatter into the function. The eigenvalues are stored in w and the eigenvectors are stored in v. The slope of the projection line is calculated from the eigenvector corresponding to the maximum eigenvalue.

```
cw = c1 + c2
cw_inverse = np.linalg.inv(cw)
cb = np.dot((u1-u2),(u1-u2).transpose())
cw_cb = np.dot(cw_inverse, cb)
w, v = np.linalg.eig(cw_cb)

eigenvector = v[w.tolist().index((max(w)))
slope = eigenvector[1] / eigenvector[0]
```

Snippet 18 - Finding eigenvectors and eigenvalues

The next step is to calculate the class probabilities, which as simple as finding the ratio of the length of the array for each class compared to the total dataset. The pooled within group covariance matrix is calculated using the formula in figure 2.4, and its inverse is taken, as this is the matrix required in the discriminant function in figure 2.5.

```
w1, w2 = len(x01) / len(x), len(x02) / len(x)
C = [[i1[0] * w1
      + i2[0] * w2,
      i1[1] * w1
      + i2[1] * w2] for i1, i2 in zip(c1, c2)]
C = np.array(C)
C_inverse = np.linalg.inv(C)
p = [w1, w2]
```

Snippet 19 - Further matrices for use in discriminant function

The final step is to make the predictions using the other 40% of the dataset. The testing dataset is spliced from the original data, and saved into two numpy arrays for x and y respectively. The discriminant function in figure 2.5 is implemented in steps for each class, separated by the line break.

If the discriminant value for class 0 is higher than class 1 then a 0 is predicted, else a 1 is predicted. The predictions are appended to a predictions array, which will then be compared with the original dataset for accuracy.

```
predictions = []
testing_x = np.array([[i[0]**transformation,
                       i[1]**transformation]
                      for i in dataset_x[split+1::]])
testing_y = np.array(dataset_y[split+1::])
for item in testing_x:
    v1_f1 = np.dot(u1, C_inverse)
    v1_f1 = np.dot(v1_f1, item.transpose())
    v2_f1 = 0.5 * np.dot(u1, C_inverse)
    v2_f1 = np.dot(v2_f1, u1.transpose())
    v3_f1 = log(p[0])
    f1 = v1_f1 - v2_f1 + v3_f1

    v1_f2 = np.dot(u2, C_inverse)
    v1_f2 = np.dot(v1_f2, item.transpose())
    v2_f2 = 0.5 * np.dot(u2, C_inverse)
    v2_f2 = np.dot(v2_f2, u2.transpose())
    v3_f2 = log(p[1])
    f2 = v1_f2 - v2_f2 + v3_f2
    if f1 > f2:
        predictions.append(0)
    else:
        predictions.append(1)
```

Snippet 20 - Making Predictions using the LDA model

The procedure is repeated for all the features (break points won, aces etc.) and then it is repeated for all the datasets. For each feature, it is also repeated for 500 different transforms applied to the data, to increase the resemblance to the normal distribution. The optimal transform is picked each time.

4.2 Linear Discriminant Analysis – Evaluation

Gathering Accuracy Data

As mentioned in section 1.2, the four attributes that I will compare for player 1 and player 2 are (in ascending order of class separability):

- Unforced Errors (weak separability)
- Aces (medium separability)
- Winners (good separability)
- Break Points Won (excellent separability)

Since linear discriminant appears to be more complex and sophisticated than the other two algorithms, one can assume that the accuracy results may be higher. The overall trend may still show break points won to have the highest accuracy, and aces to have the lowest.

	Prediction Accuracy (%)			
Dataset	UFE	ACE	WNR	BPW
Wim-M	71.1	73.3	82.2	93.3
Wim-W	60.6	68.5	67.0	94.5
Aus-M	66.7	69.2	70.1	84.7
Aus-W	70.8	65.1	66.7	81.5
Fre-M	70.2	68.6	67.8	82.0
Fre-W	68.6	69.6	68.9	85.1
US-M	-	65.4	-	82.4
US-W	69.6	63.3	68.4	84.8

Table 4 - LDA Prediction Accuracy

As expected, the class prediction accuracy for break points won is the highest on average at 86.1%. LDA performed excellently with the "winners" attribute, with an average prediction accuracy of 70%; a value that neither Logistic Regression or Gaussian Naïve Bayes could attain for the same attribute. It met accuracy expectations for "aces" and "unforced errors" with average prediction accuracies of 67.9% and 68.3% respectively.

With relatively high prediction accuracies for all the tested attributes, it is safe to claim that the linear discriminant analysis model created is an excellent predictor of the outcome of a tennis match, when passed two sufficiently separated features and at least 80 data items.

The complete code for Linear Discriminant Analysis can be found in appendix item B. The parameters to be compared are chosen at the top of the program, and the results are logged to the console.

5.0 Evaluation of Algorithm Efficacy and Implementation

Every algorithm was able to greatly improve upon the accuracy attained by a random selection of class (50%), meaning that they were implemented correctly at the very least. Efficacy is defined as the "capability for producing a desired result or effect", (The definition of efficacy, no date) and the quantitative measure of the desired result (to be able to predict the winner of a tennis match based on two attributes), is the accuracy of the procedure. The accuracy data for all three algorithms, derived from tables 1, 3 and 4 is summarised below (It must be noted that the accuracies calculated *only* are valid for the algorithm's implementation on the Tennis Major Tournaments Dataset and cannot be interpreted as characteristics of the procedure itself.):

	Average prediction accuracy (%)			
Algorithm	UFE	ACE	WNR	BPW
N. Bayes	60.1	65.3	68.1	86.2
Log. Reg	68.4	65.2	67.0	85.0
LDA	68.3	67.9	70.0	86.1

Table 5 - Overall Algorithm Accuracy

Ranking each algorithm by overall average prediction accuracy, Linear Discriminant Analysis has the highest average at 73.1%, Logistic Regression is just below at 71.4% and Gaussian Naïve Bayes is third with an overall average prediction accuracy of 69.9%. With respect to the amount of data used, and the number of repeats done for each feature, for each dataset, these accuracy discrepancies are small. However, when performances for specific features are observed, then there are larger disparities in performance.

For example, for unforced errors, Gaussian Naïve Bayes could only achieve an accuracy of 60.1%; a 10% improvement upon random selection, whereas both Logistic Regression and Linear Discriminant Analysis achieved accuracies of 68% or thereabouts. An 8.4% difference in prediction accuracy shows that there is a fundamental difference in implementation between Naïve Bayes and Logistic Regression or Linear Discriminant Analysis. Naïve Bayes uses conditional probability extensively to estimate classes. The probability density function (figure 1.6) uses only the mean and standard deviation of the class feature data to estimate a class probability. For a weakly separated feature like unforced errors, where the mean and standard deviation can be relatively similar for certain datasets, this method can result in many predictive flaws. Furthermore, use of the PDF explicitly assumes the feature data is Gaussian, whereas it may actually not resemble a normal distribution completely. Logistic Regression, however, estimates the probability function by adjusting a set of coefficients for every single data point, rather than just the means and standard deviations. Similarly, LDA uses several other statistics such as within class, between class scatter and also the covariance. When combined effectively, they can be far better than using just the mean and standard deviation. However, Gaussian Naïve Bayes excelled when the class were very well separated; break points won. It achieved an average prediction accuracy of 86.2% (table 5), which proves the aforementioned point; when classes are well separated, the mean and standard deviation are disparate allowing for accurate predictions. In this case, within-class and between class scatter hold relatively less importance.

Linear Discriminant Analysis broke the 70% accuracy barrier for predicting the winner of game based only winners. When thought about literally, this figure in my opinion is commendable. However, there was a trade-off. The runtime of the algorithm was far longer than Gaussian Naïve Bayes and a little longer than Logistic Regression at around 35-40s per execution. For a relatively small dataset of less than 1000 items, the time complexity of the algorithm is insignificant, but once scaled up there will exponential repercussions. The reason for this disparity in run time may be due to the repeated matrix calculations in the procedure (inverse, eigenvalue/eigenvector calculations). Logistic Regression averaged 32-35s per execution, due to the repetition of the training and predicting for hundreds of learning rates (α), but this can be reduced by lessening the number of epochs (number of times the coefficient optimization procedure is executed). Gaussian Naïve Bayes was the quickest, by far averaging 8-10s despite the searching of hundreds of incrementally different transformation values to find the optimal one.

A factor that had a huge effect on the accuracy of all the algorithms was the ratio of the split between training data and testing data. The ratio used was 60% of the dataset was reserved for training the dataset, and the other 40% was used for testing purposes. As this was kept constant, it had minimal effect on the *relative* algorithm performance but did affect the raw values. However, it must be noted that algorithms like Gaussian Naïve Bayes and Linear Discriminant Analysis still do perform well with sparse data (little training data), whereas Logistic Regression requires more training data since it relies on an optimization procedures.

In conclusion, the disparity in performance of the algorithms was little in terms of the raw average accuracy predictions, but significant with respect to the run time of the algorithm. The raw performance assessment above only applies to this dataset, and is not representative of the procedure overall. However, the relative performance assessment can be representative of the procedures overall, but limited to small scale datasets.

Bibliography

A simple explanation of naive Bayes classification (2012) Available at: http://stackoverflow.com/a/20556654/4201858 (Accessed: 5 January 2017).

Bland, J.M. and Altman, D.G. (1996) ‘Statistics notes: Measurement error’, *BMJ*, 313(7059), p. 2. doi: 10.1136/bmj.313.7059.744.

Brownlee, J. (2014) *Better naive Bayes: 12 tips to get the most from the naive Bayes algorithm*. Available at: http://machinelearningmastery.com/better-naive-bayes/ (Accessed: 7 January 2017).

Brownlee, J. (2015) *Master Machine Learning Algorithms*. 1.6 edn. Jason Brownlee.

Brownlee, J. (2016) *Gradient descent for machine learning*. Available at: http://machinelearningmastery.com/gradient-descent-for-machine-learning/ (Accessed: 9 January 2017).

Carnegie Mellon University (no date) *Logistic Regression: Carnegie Mellon University*. Available at: https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf (Accessed: 8 January 2017).

Casella, G. and Berger, R.L. (2001) *Statistical inference*. 2nd edn. United States: Duxbury/Thomson Learning.

Data transformation (statistics) (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Data_transformation_(statistics) (Accessed: 21 February 2017).

Desmos (2017) *Beautiful, free math*. Available at: https://www.desmos.com/ (Accessed: 9 January 2017).

Grace-Martin, K. (2008) *What is a Logit function and why use logistic regression?* Available at: http://www.theanalysisfactor.com/what-is-logit-function/ (Accessed: 8 January 2017).

Jason Brownlee | LinkedIn (no date) Available at: https://au.linkedin.com/in/jasonbrownlee (Accessed: 4 January 2017).

Marsland, S. (2014) *Machine learning: An algorithmic perspective, second edition*. 02nd edn. Boca Raton, FL, United States: Crc Press.

Massey and Zealand, N. (1998) *Prof Stephen Marsland - professor - Massey university*. Available at: http://www.massey.ac.nz/massey/expertise/profile.cfm?stref=895830 (Accessed: 4 January 2017).

NumPy — NumPy (no date) Available at: http://www.numpy.org/ (Accessed: 10 January 2017).

Numpy.mean — NumPy v1.12 manual (2008) Available at: https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html (Accessed: 10 January 2017).

Pak, A. and Paroubek, P. (2010) 'Twitter as a corpus for sentiment analysis and opinion mining', , p. 1.

Simon, P. (2013) *Too big to ignore: The business case for big data*. United States: Wiley, John & Sons.

StatsDirect Limited (2000a) *Normal distribution and standard normal (Gaussian)*. Available at: http://www.statsdirect.com/help/distributions/normal.htm (Accessed: 6 January 2017).

StatsDirect Limited (2000b) *Normal distribution and standard normal (Gaussian)*. Available at: http://www.statsdirect.com/help/distributions/normal.htm (Accessed: 6 January 2017).

Stuart, A. and Ord, K.J. (1994) *Kendall's advanced theory of statistics, distribution theory (volume 1)*. 6th edn. New York, NY, United States: Wiley-Blackwell.

Teknomo, K. (2016) *Linear Discriminant analysis (LDA) numerical example*. Available at: http://people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html (Accessed: 10 January 2017).

The definition of efficacy (no date) in Available at: http://www.dictionary.com/browse/efficacy (Accessed: 21 February 2017).

Two Feature Data (no date) Available at: http://openclassroom.stanford.edu/MainFolder/courses/MachineLearning/exercises/ex7materials/twofeaturedata.png (Accessed: 8 January 2017).

UCI machine learning repository (no date) Available at: http://archive.ics.uci.edu/ml/ (Accessed: 4 January 2017).

UCI machine learning repository: Tennis Major Tournament match statistics data set (2014) Available at: https://archive.ics.uci.edu/ml/datasets/Tennis+Major+Tournament+Match+Statistics (Accessed: 4 January 2017).

Appendix A

```
import csv
from math import e, pi, ceil, log
import matplotlib.pyplot as plt


def pdf(x, mean, sd):
    return (e ** (-((x - mean) ** 2) / (2 * (sd ** 2))))/ (((2 * pi) ** 0.5) * sd)

def standard_deviation(data):
    squared_data = [x ** 2 for x in data]
    mean_squared_data = sum(squared_data) / len(squared_data)
    mean_data = sum(data) / len(data)
    variance = mean_squared_data - (mean_data ** 2)
    return variance ** 0.5

def avg(data):
    return sum(data) / len(data)


datasets = ['../datasets/Wimbledon-men-2013.csv',
            '../datasets/Wimbledon-women-2013.csv',
            '../datasets/AusOpen-men-2013.csv',
            '../datasets/AusOpen-women-2013.csv',
            '../datasets/FrenchOpen-men-2013.csv',
            '../datasets/FrenchOpen-women-2013.csv',
            '../datasets/USOpen-men-2013.csv',
            '../datasets/USOpen-women-2013.csv']

dataset_accuracy = []
for filename in datasets:
    parameter1 = 'BPW.1'
    parameter2 = 'BPW.2'
    par1, par2, result = [], [], []
    with open(filename, 'r') as data:

        reader = csv.reader(data)  # Create CSV Reader
        for row in reader: header = row; break  # Get headings of table.

        par1_index = header.index(parameter1)  # Get column number of Parameter 1
        par2_index = header.index(parameter2)  # Get column number of Parameter 2
        result_index = header.index('Result')  # Get column number of Result

        for row in reader:
            if row[par1_index] != '' and row[par2_index] != '':
                par1.append(row[par1_index])  # Add Parameter 1 data to x1 array.
                par2.append(row[par2_index])  # Add Parameter 2 data to x2 array.
                result.append(row[result_index])  # Add Result data to y array.
Extracting sample data from data set.
    if len(par1) == 0 or len(par2) == 0:
        continue
    accuracy = []
    transformation = 0.01
    for x in range(150):
        split = ceil(len(par1) * 0.6) - 1
        x1 = [int(x) ** transformation for x in par1[0:split]]
        x2 = [int(x) ** transformation for x in par2[0:split]]
        y = [int(y) for y in result[0:split]]
        win_x1, win_x2 = [], []
        loss_x1, loss_x2 = [], []
        for index in range(len(x1)):  # Filter out wins and losses.
            if y[index] == 1:
                win_x1.append(x1[index])
                win_x2.append(x2[index])
            else:
                loss_x1.append(x1[index])
                loss_x2.append(x2[index])  # Filtering out wins and losses.
        win_x1_m = avg(win_x1)
        win_x2_m = avg(win_x2)
        loss_x1_m = avg(loss_x1)
        loss_x2_m = avg(loss_x2)

        win_x1_std = standard_deviation(win_x1)
        win_x2_std = standard_deviation(win_x2)
        loss_x1_std = standard_deviation(loss_x1)
        loss_x2_std = standard_deviation(loss_x2)
        p0 = len(loss_x1) / (len(win_x1) + len(loss_x1))
        p1 = len(win_x1) / (len(win_x1) + len(loss_x1))

        predictions = []
        x1_test = [int(x) ** transformation for x in par1[split + 1::]]
        x2_test = [int(x) ** transformation for x in par2[split + 1::]]
        y_test = [int(y) ** transformation for y in result[split + 1::]]

        for item1, item2 in zip(x1_test, x2_test):
            c0 = pdf(item1, loss_x1_m, loss_x1_std) \
                 * pdf(item2, loss_x2_m, loss_x2_std) \
                 * p0
            c1 = pdf(item1, win_x1_m, win_x1_std) \
                 * pdf(item2, win_x2_m, win_x2_std) \
                 * p1
            if c1 > c0:
                predictions.append(1)
            else:
                predictions.append(0)
        correct = 0
        for i, j in zip(predictions, y_test):
            if i == j:
                correct += 1
        accuracy.append(correct / len(predictions))

        transformation += 0.01

    dataset_accuracy.append([filename, max(accuracy)])
    fig, ax = plt.subplots()
    fig.suptitle("{} vs {} for {}".format(parameter1, parameter2, filename[12::]))
    ax.plot(win_x1, win_x2, 'ro')
    ax.plot(loss_x1, loss_x2, 'bo')
    plt.xlabel(parameter1)
    plt.ylabel(parameter2)

accuracy_average = []
for acc in dataset_accuracy:
    print("{} : {}".format(acc[0], acc[1]))
    accuracy_average.append(acc[1])

average = sum(accuracy_average) / len(accuracy_average)
print(average)
# plt.show()
```

Appendix B

```
import csv
from math import e, ceil

datasets = ['../datasets/Wimbledon-men-2013.csv',
            '../datasets/Wimbledon-women-2013.csv',
            '../datasets/AusOpen-men-2013.csv',
            '../datasets/AusOpen-women-2013.csv',
            '../datasets/FrenchOpen-men-2013.csv',
            '../datasets/FrenchOpen-women-2013.csv',
            '../datasets/USOpen-men-2013.csv',
            '../datasets/USOpen-women-2013.csv']


def logistic_function(B0, B1, B2, x1, x2):
    probability = 1 / (1 + (e ** (-(B0 + (B1 * x1) + (B2 * x2)))))  # Calculate Prediction
using Logistic Function
    return probability


dataset_accuracy = []
for filename in datasets:
    parameter1 = 'BPW.1'
    parameter2 = 'BPW.2'
    par1, par2, result = [], [], []
    with open(filename, 'r') as data:

        reader = csv.reader(data)  # Create CSV Reader
        for row in reader: header = row; break  # Get headings of table.

        par1_index = header.index(parameter1)  # Get column number of Parameter 1
        par2_index = header.index(parameter2)  # Get column number of Parameter 2
        result_index = header.index('Result')  # Get column number of Result

        for row in reader:
            if row[par1_index] != '' and row[par2_index] != '':
                par1.append(row[par1_index])  # Add Parameter 1 data to x1 array.
                par2.append(row[par2_index])  # Add Parameter 2 data to x2 array.
                result.append(row[result_index])  # Add Result data to y array. Extracting
sample data from data set.
    if len(par1) == 0 or len(par2) == 0:
        continue
    training_accuracy = []
    prediction_accuracy = []
    for x in range(1000):
        split = ceil(len(par1) * 0.6) - 1
        x1 = [int(x) for x in par1[0:split]]  # Apply transformation
        x2 = [int(x) for x in par2[0:split]]  # Apply transformation
        y = [int(y) for y in result[0:split]]

        B0, B1, B2 = 0, 0, 0
        alpha = 0.001
        bias = 1
        epochs = 10
        training_accuracy = []
        for repeat in range(epochs):
            total_correct = 0
            for index in range(len(x1)):
                prediction = logistic_function(B0, B1, B2, x1[index], x2[index])

                if prediction < 0.5:
                    sharp_prediction = 0
                else:
                    sharp_prediction = 1
                if sharp_prediction == y[index]:
                    total_correct += 1

                delta_coefficient =(alpha*(y[index]-prediction)*prediction*(1-prediction))

                # Update Coefficients.
                B0 += delta_coefficient * bias
                B1 += delta_coefficient * x1[index]
                B2 += delta_coefficient * x2[index]
            training_accuracy.append(((total_correct / len(x1)) * 100))

        predictions = []
        x1_test = [int(x) for x in par1[split + 1::]]
        x2_test = [int(x) for x in par2[split + 1::]]
        y_test = [int(y) for y in result[split + 1::]]
        for item1, item2 in zip(x1_test, x2_test):
            prediction = logistic_function(B0, B1, B2, item1, item2)
            # Calculate Accuracy
            if prediction < 0.5:
                predictions.append(0)
            else:
                predictions.append(1)
        correct = 0
        for i, j in zip(predictions, y_test):
            if i == j:
                correct += 1
        prediction_accuracy.append(correct / len(predictions))
        alpha += 0.0005
    dataset_accuracy.append([filename, max(prediction_accuracy)])

accuracy_average = []
for acc in dataset_accuracy:
    print("{} : {}".format(acc[0], acc[1]))
    accuracy_average.append(acc[1])

average = sum(accuracy_average) / len(accuracy_average)
print("\n Average accuracy: {}".format(average))
```

Appendix C

```
import csv
from math import log, ceil
import numpy as np
import matplotlib.pyplot as plt

g = 2  # Number of groups in y.

parameter1 = 'UFE.1'
parameter2 = 'UFE.2'
dataset_x = []
dataset_y = []
datasets = ['../datasets/Wimbledon-men-2013.csv',
            '../datasets/Wimbledon-women-2013.csv',
            '../datasets/AusOpen-men-2013.csv',
            '../datasets/AusOpen-women-2013.csv',
            '../datasets/FrenchOpen-men-2013.csv',
            '../datasets/FrenchOpen-women-2013.csv',
            '../datasets/USOpen-men-2013.csv',
            '../datasets/USOpen-women-2013.csv']


dataset_accuracy = []
for filename in datasets:
    with open(filename, 'r') as data:

        reader = csv.reader(data) # Create CSV Reader
        for row in reader: header = row; break # Get headings of table.

        par1_index = header.index(parameter1) # Get column number of Parameter 1
        par2_index = header.index(parameter2) # Get column number of Parameter 2
        result_index = header.index('Result') # Get column number of Result

        for row in reader:
            if row[par1_index] != '' and row[par2_index] != '':
                dataset_x.append([int(row[par1_index]), int(row[par2_index])]) #Add
Parameter 1 data to x1 array.
                dataset_y.append(int(row[result_index])) #Add Result data to y
array. Extracting sample data from data
                # set.

    if len(dataset_x) == 0 or len(dataset_y) == 0:
        continue

    accuracy = []
    transformation = 0.1
    for repeat in range(500):
        split = ceil(len(dataset_x) * 0.6) - 1
        training_x =[[i[0]**transformation, i[1]**transformation] for i in
dataset_x[0:split]]
        training_y = dataset_y[0:split]
        x = np.array(training_x)
        y = np.array(training_y)
        x1 = []
        x2 = []
        for num, yi in enumerate(y):
            if yi == 0:
                x1.append(x[num]) # 1 = class 0 ; 2 =  class 1
            else:
                x2.append(x[num])

        u1 = np.mean(x1, axis=0)
        u2 = np.mean(x2, axis=0)
        u = np.mean(x, axis=0)

        x01 = x1 - u
        x02 = x2 - u
        c1 = np.dot(x01.transpose(), x01) / x01.shape[0]
        c2 = np.dot(x02.transpose(), x02) / x02.shape[0]
        # Finding discriminant lines
        cw = c1 + c2
        cw_inverse = np.linalg.inv(cw)
        cb = np.dot((u1-u2),(u1-u2).transpose())
        cw_cb = np.dot(cw_inverse, cb)
        w, v = np.linalg.eig(cw_cb)

        eigenvector = v[w.tolist().index((max(w)))]
        slope = eigenvector[1] / eigenvector[0]

        w1, w2 = len(x01) / len(x), len(x02) / len(x)
        C = [[i1[0] * w1
              + i2[0] * w2,
              i1[1] * w1
              + i2[1] * w2] for i1, i2 in zip(c1, c2)]
        C = np.array(C)
        C_inverse = np.linalg.inv(C)
        p = [w1, w2]

        predictions = []
        testing_x = np.array([[i[0]**transformation, i[1]**transformation] for i in da-
taset_x[split+1::]])
        testing_y = np.array(dataset_y[split+1::])
        for item in testing_x:
            v1_f1 = np.dot(u1, C_inverse)
            v1_f1 = np.dot(v1_f1, item.transpose())
            v2_f1 = 0.5 * np.dot(u1, C_inverse)
            v2_f1 = np.dot(v2_f1, u1.transpose())
            v3_f1 = log(p[0])
            f1 = v1_f1 - v2_f1 + v3_f1

            v1_f2 = np.dot(u2, C_inverse)
            v1_f2 = np.dot(v1_f2, item.transpose())
            v2_f2 = 0.5 * np.dot(u2, C_inverse)
            v2_f2 = np.dot(v2_f2, u2.transpose())
            v3_f2 = log(p[1])
            f2 = v1_f2 - v2_f2 + v3_f2
            if f1 > f2:
                predictions.append(0)
            else:
                predictions.append(1)

        correct = 0
        for item1, item2 in zip(predictions, testing_y):
            if item1 == item2:
                correct += 1

        accuracy.append(correct / len(predictions))
        transformation += 0.001
    dataset_accuracy.append([filename, max(accuracy)])
    fig, ax = plt.subplots()

    # Discriminant lines
    x_0 = -5
    x_1 = 100
    y_0 = 100
    y_1 = slope*(x_1-x_0)
    ax.scatter([x_0, x_1], [y_0, y_1], marker='^', s=20, c='r')
    ax.plot([x_0, x_1], [y_0, y_1], c='g')

    # Data
    ax.plot([z[0] for z in x1], [z[1] for z in x1], 'ro')
    ax.plot([z[0] for z in x2], [z[1] for z in x2], 'bo')
accuracy_average = []
for acc in dataset_accuracy:
    print("{} : {}".format(acc[0], acc[1]))
    accuracy_average.append(acc[1])

average = sum(accuracy_average) / len(accuracy_average)
print("\n Average accuracy: {}".format(average))
```

Appendix D

Source Evaluations

A simple explanation of naive Bayes classification (2012) Available at: http://stackoverflow.com/a/20556654/4201858 (Accessed: 5 January 2017).

- The source