Overview of Machine Learning Techniques as Explored for Application to the GCFP Chip Lyme Data

Prepared by F. Doyle

6/8/2022

Table of Contents

[Introduction 3](#_Toc105579144)

[Python and ML Libraries 3](#_Toc105579145)

[Data Format 4](#_Toc105579146)

[Supervised Machine Learning 5](#_Toc105579147)

[Training and Testing Machine Learning Models 5](#_Toc105579148)

[Feature Selection for Models 5](#_Toc105579149)

[Cross Validation 7](#_Toc105579150)

[Machine Learning Models Examined 8](#_Toc105579151)

[Decision Trees 8](#_Toc105579152)

[Decision tree construction: 9](#_Toc105579153)

[Random Forest Classifier 9](#_Toc105579154)

[Logistic Regression 10](#_Toc105579155)

[Support Vector Machine 11](#_Toc105579156)

# Introduction

This document is meant to introduce a number of concepts and techniques at a relatively high level, in order to provide a “jumping off” point. There are numerous resources online where details may be found. In some cases, specific references to such resources may be noted, if they were deemed of particular utility.

The machine learning techniques investigated for this project have all fallen within the category of “supervised learning”. Data from samples consisting of normalized antigen/antibody signals with corresponding diagnostic classification are split into training and test subsets (within data structures…the full dataset is loaded from a single file). ML models are fit to the training data features (“predictors”) and diagnostic classes (“targets”) and are then used to predict the classes of the test set samples. There are two fundamental types of classifiers explored here, binary (meaning two classes, “positive” or “negative” for our purposes) and multiclass (or multinomial, encompassing more than two distinct classes)

## Python Jupyter Notebooks

The code used was written in Python using Jupyter Notebooks and libraries commonly used for data science and machine learning (NumPy, Pandas, Matplotlib, Scikit-learn, etc..). The notebooks for each of the approaches are saved as individual ***.ipynb*** files that are referenced in their respective sections and are available to anyone that wishes to experiment with them. This document gives very brief overviews of the different models conceptually or provides links to pages that do so. The notebooks show how to actually apply the models in Python and are commented to (hopefully) make understanding the code easier.

Jupyter Notebooks provide a web-based interface for running Python in editable segments and allow for data visualization and analysis in an editable environment. You can define and execute blocks of code as small as a single instruction, or write full functions as well as manipulate, analyze, and graph data in real time. You can think of Jupyter Notebooks as an R&D environment. Once ML models and code have been refined to a point where they are ready for final use, you will likely eventually convert to a deployable application, but in the meantime, this is an easy-to-use development environment.

There are a handful of Python libraries that are needed to perform the analyses that will be described here, the most important (grouped by category) are:

Scientific Computing:

* Pandas – data frame and associated logic
* NumPy – arrays and matrices
* SciPy – functions for advanced math problems

Visualization (in some cases, redundant capabilities but one version may be preferred aesthetically):

* Matplotlib – graphs and plots
* Seaborn – heatmaps, violin plots, etc...

Algorithms:

* Scikit-learn – machine learning models (support vector machines, decision trees, etc.)

An easy approach for setting up this environment is to install Anaconda3 via <https://www.anaconda.com/> which includes Jupyter as well as the libraries needed here for data science and machine learning. Once installed, you should have a launch icon for Jupyter Notebook. When executed, it should open a console window and display a number of lines of output. At the bottom of the output there should be a URL that you can paste into a local web browser (if it hasn’t already been launched by Jupyter) to enter the environment. The files tab will allow you to open one of the **.ipynb** files I’ve provided (after it has been downloaded to your hard drive). I suggest using ‘eda.ipynb’ first, which is the “exploratory data analysis” notebook, pictured in Figure 1. Note, though the individual Jupyter notebook cells organize blocks of code, lower cells may be dependent on previous cells having executed for variable initialization or library imports. If you have just loaded/reloaded a notebook file, and encounter an unexpected error, keep this in mind.

Graphical user interface, text, application

Description automatically generated

Figure – A Jupyter Notebook as a tab in web browser. Note separate code cells (disregard numbering in brackets). The code in the currently active cell (highlighted blue) will be executed by pressing the “Run” button (circled in red) on the taskbar.

## Data Format

The general data format for the tools presented here and for many data science applications consists of table-based structures known as “data frames” conceptually similar to spreadsheets. The columns of the table represent particular attributes or “features” and individual rows represent samples with entries for each of the designated features. If the data already exists in a spreadsheet, the easiest way to provide a file to work with is to save a sheet (in the format shown in fig 2) as a “.csv” (comma separated value) file which can then easily be loaded directly into a Pandas DataFrame using the Pandas “read\_csv” function.

Graphical user interface, application, table, Excel

Description automatically generated

Figure 2 – Excel sheet of data formatted as needed for easy loading as a DataFrame after export to a .csv file

# Supervised Machine Learning

In brief, supervised (versus unsupervised) machine learning utilizes “labeled” data for training. In our current context, we use samples exhibiting varying levels of Lyme antigen related signals from the diagnostic chip and their already known (e.g., “labeled”) associated diagnostic class (“negative”, “positive early acute”, “positive early convalescent” or “positive late”) to train a classifier model to predict this for future samples where it has not yet been identified. Colloquially speaking, we are teaching the models to recognize diagnostic classes (the ***target***) from signals (***predictors***) by providing examples.

Unsupervised learning looks for patterns (clustering, etc.) in unlabeled data. It is not in the scope of this discussion.

# Training and Testing Machine Learning Models

Any machine learning model is only as good as the data it has been trained with. In general, more data for training is better. Also, data representative of the whole population of potential cases is also better in order to avoid biased models. If you train a model with pictures to differentiate between domesticated cats, dogs and frogs and you only ever train with dog images of St Bernard, Mastiffs, and similar large breeds…the model could conceivably predict that an image of a Chihuahua is a frog. That said, depending on choice of features used as predictors and their weighting, it may correctly predict dog or instead err with cat (frogs aren’t fuzzy, cats are smaller than large breed dogs, etc.).

## Feature Selection for Models

Feature choice is an important consideration. More data, in the form of “more examples” is better. More data, in the form of more complicated samples may just confuse the model with irrelevant features. Correlation with the target is a straightforward example of a feature selection analysis approach. A heatmap of a correlation grid for features in the dataframe is shown in figure 3 and the code to produce it is in the *eda* notebook.

A picture containing text, colorful, colors, different

Description automatically generated

Figure 3 – Heatmap of correlations between features in the dataframe

Additionally, the *eda* notebook has some other data visualization examples (graphs and plots) that are easy to produce and can help to identify relationships between features. Figure 4 shows a portion of a grid of pairwise plots produced between features in the dataframe. Where the same feature intersects with itself on x and y axis in the grid, a histogram is shown. The “Diag” label is used to color the sample and was converted into a numeric form in the “state” feature, which provides yet another way to visualize the information. With minimal additional code, other features can be derived from the existing data and used with this same plot type to look at the data in different ways (e.g., create an ‘outlier’ column and mark as true or false depending on some specified analysis, then use that feature to color the plots. This will allow visualization of how the outlier cases cluster and/or separate out in certain pairwise plots.

Calendar

Description automatically generated

Figure 4 – Portion of a grid of pairwise plots visualizing the full dataset as of 5/17/2022. Coloring is based on the “Diag” classification. New columns can be created to allow alternate visualization. For example, an “outlier” column where non-Negative cases that had a VlsE score less than some cutoff. Using that column for color would allow easier visualization of whether those particular samples are more easily identified as “positives” in other feature plots.

## Cross Validation

A significant problem with many machine learning approaches, particularly with small training sets, is the potential for “**overfitting**” where the resulting model corresponds too closely to the training set to make accurate predictions for more generalized future samples.

The common approach for avoiding overfitting is to reserve a random subset of available data for testing and then use the remaining entries for training (“train/test splitting”). When tuning parameters for particular machine learning approaches (different models have different types and numbers of parameters that may be adjusted), you still risk overfitting based on the test set. The solution to this is to create a third “validation” subset. In this method, the training is done on the “training set”, evaluation and tuning of the model is performed on the ”validation set” and final evaluation is performed on the reserved “test set”. The issue here is that by subdividing available data into three sets, we severely decrease the number of samples that are available for use in learning and risk pushing the model towards some particular bias in aspects of the randomly chosen subsets used for training and validation.

**Cross validation** can help to solve this by taking **k-folds** (divisions) of the training portion of the training/test split, then performing iterative training/validation using k-1 folds for training and the remaining fold for testing/validation, where the fold used for testing changes in each iteration. Parameters that showed best general performance across all iterations can then be used to train the model on the full training set and the final evaluation can be made on the reserved test data. Figure 5 depicts this process.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5 k-fold cross validation - <https://scikit-learn.org/stable/modules/cross_validation.html>

In some of the example notebooks, a tool called GridSearchCV is used. This tool takes a set of values for a number of specified parameters and then invokes them in a combinatorial manner on the model under examination using cross validation and returns the best performing set. This will allow for a somewhat automated future exploration and estimation of the best performing models and parameters as additional data is gathered for the project.

## Performance Measures

There are a number of metrics available to gauge how well a given ML model performs, and each provide particular and subtly different perspectives. The common diagnostic measurements of “sensitivity” and “specificity” are available via “classification reports” and “confusion matrices” that are easily generated by Python code. Sensitivity, (aka “recall” or “true positive rate”) equates to TP/(TP+FN) where TP is “true positive”, and FN is “False Negative”. Specificity (“true negative rate”) equates to TN/(TN+FP) where TN is “true negative”, and FP is “false positive”. “Recall” is reported in the classification reports for each label and represents “sensitivity” in regard to the “Pos” label or “specificity” in regard to the ”Neg” label (see Fig 6). The classification report gives a breakdown of performance metrics by label including “precision” (aka “positive predictive value”), which for “positive” equates to TP/(TP+FP). Also reported is “**accuracy**” (# of correct predictions/# of total predictions). Accuracy is considered a poor metric if there are class imbalances in the data. Additionally, it presents the **f1 score**, which is the harmonic mean of the precision and recall for a given label (2\*[(precision\*recall)/(precision+recall)]):

* A model will obtain a **high F1 score** if both Precision and Recall are high
* A model will obtain a **low F1 score** if both Precision and Recall are low
* A model will obtain a **medium F1 score** if one of Precision and Recall is low and the other is high

Chart

Description automatically generated

Figure Classification report and confusion matrix examples for a binary classifier. In this case the values for sensitivity and specificity are directly given by the classification report as the recall with respect to “Pos” label (Sensitivity, .88) and the recall with respect to “Neg” label (Specificity, .75). These could also be calculated via the numbers in the confusion matrix corresponding to TP,FP,TN and FN.

The confusion matrix is simply a grid of predictions versus actual values. Individual grid tiles or groups (depending on number of classes) can correspond to the TP/FP/TN/FN values. Calculating sensitivity and specificity for the multiclassification is slightly more complicated but can still be derived from the confusion matrix in a manner that ignores the correct/incorrect subclassification of the positive values (see Fig 7).

A picture containing calendar

Description automatically generated

Figure Grouping subclasses while ignoring accuracy to derive TP/FP/TN/FN values.

# Machine Learning Models That Have Been Explored

A number of techniques were applied to the existing dataset. Most have showed some promise. It is too early to say which will provide the best performance, though there are some that have particularly desirable qualities if they can be made to perform well (e.g., logistic regression’s probabilities). Jupyter notebooks for multiclass (attempting to identify positive subclasses of “Early Acute”, etc.) and binary classification (simply “pos” vs “neg”) have been provided for each model.

## Decision Trees

**Related code in dt.ipynb**. Decision trees are perhaps the easiest model to understand and to manually follow a sample through their analysis. They can be thought of as similar to the “differential diagnosis” process. Basically, decision trees consist of a logical flow of test-based choices that can be represented by a tree style graph starting at a root/decision node. Internal nodes correspond to tests and the branches (“edges” in graph parlance) from these nodes correspond to results, based on values of the predictor variables (e.g., “symptoms”, or in this case, antigen related signals). Eventually, the final “leaf nodes” assign a classification (“diagnosis”, or here, one of the four patient state categories associated with the samples).

A picture containing text, sign, several

Description automatically generated

Figure 8 Example simple decision tree built from an earlier subset of the Lyme data

Each of the decision nodes in this plot shows the test to be performed on the top line. Additionally:

* **“entropy”** refers to how “mixed” the samples are with regard to the represented classes (0=’pure’, or only a single class type).
* **“samples”** is the number of samples from the training set that are represented in the node prior to test being applied.
* **“value”** is the breakdown of counts by class. Here [“Negative”, “Pos – Early Acute”, “Pos – Early Conv”, “Pos - Late” ]
* **“class”** refers to the class representing the majority of samples at node (node colors relate to this).

As stated, entropy in this regard generally refers to the measure of randomness in the data. At the top, the root node contains all the samples representing all the mixed classes and has the highest entropy. The terminal leaf nodes consist of only single classes and have corresponding entropy of zero. Lower data entropy can be described as higher *information purity*. Reductions in entropy equate to *information gain*.

For a binary classifier:

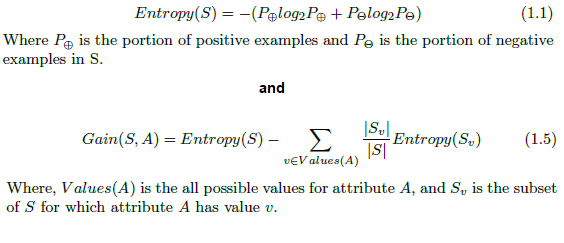


Figure 9 – Entropy calculations for a binary classifier. Note figure taken from <https://medium.com/analytics-vidhya/entropy-calculation-information-gain-decision-tree-learning-771325d16f>)

### Decision tree construction:

The logic to produce decision trees exists within the Python Scikit-learn library and therefore does not need to be reimplemented. However, it is always worth having a general understanding of any tools used, so this is very quick overview of the high-level logic. Decision trees are built with recursive partitioning to classify the data. The basic approach is:

1. For each attribute in dataset
   1. Calculate the significance of that attribute in splitting of data (information gain). That is, which attribute can be used to divide the data into more distinct target class groupings. The determination of values for split tests is part of this process but outside the scope of this description.
2. Split the data based on the value of the best (highest information gain) attribute.
3. For resulting nodes, repeat process starting at step 1.

All available attributes may not be needed to build a tree that achieves full classification of available training samples and maximum allowed depth (number of decisions nodes to traverse for classification) might also not be needed in constructing the tree. Maximum depth of tree may need to be constrained to avoid “overfitting” data to the training sample. Decision trees may be prone to overfitting and creating a model that perfectly categorizes the training data based on peculiarities (i.e., outliers) amongst *predictors* (the variables used to determine target classification) that will not accurately classify more generalized future data. For example, in the tree shown, the final decision node for “negative” or “pos – early acute” uses **higher** values for p66 to classify samples as negative. This seems counterintuitive and may be an artifact due to chip manufacture/processing or some individual’s biology presenting an outlier (though it is worth keeping in mind that machine learning is adept at finding counterintuitive yet factual relationships).

There are not many available parameters to set for decision trees. Entropy vs Gini index is one, and for now we will concentrate on entropy (it is generally easier to understand, and the benefit of Gini seems to reportedly be in processing speed which is unlikely to be an issue until sample size is much larger than currently available).

## Random Forest Classifier

**Related code in rnd\_forest.ipynb**. The Random Forest Classifier relates to Decision Trees. This model will create some number of specified “estimators” (individual decision trees). Each one of these is built by selecting random subsamples from the dataset and using random subsets of features to split the data with. When a sample’s class is predicted by this model, the sample is passed through each of the model’s estimators and the **majority** class result is returned as the prediction. As an ensemble approach based on what are expected to be uncorrelated trees, it has a reduced risk of overfitting to a particular dataset as a whole. See <https://www.ibm.com/cloud/learn/random-forest> for more information.

Chart

Description automatically generated

Figure 10 Diagram of a Random Forest Classifier from <https://medium.com/analytics-vidhya/random-forest-classifier-and-its-hyperparameters-8467bec755f6>

## Logistic Regression

**Related code in mn\_logistic\_regression.ipynb and binary\_logistic\_regression.ipynb**.Chart, line chart

Description automatically generated Similar to linear regression, logistic regression estimates a set of weight parameters as a vector q (“theta”) whose dot product with the “feature set” vector (‘X’) yields the predictive value. In linear regression the formula for the line is written as = qT·X. For a binary classifier (e.g., two categories), one category is set as y=0 and the other as y=1. A formula for a straight line would be capable of producing values beyond these limits and values that do not accurately reflect probabilities within them. The s (“sigmoid“) function corrects for this and can be written as = s(qT·X) where s(z) = 1/(1+e-z). , in this instance, is the probability (according to the model) of the correct classification being the category represented at y=1. 1- is therefore the probability of the correct classification being the category represented at y=0. Figure 9 depicts the linear regression versus binary logistic regression graphs. Multiclass classification (meaning more than ***two*** classes) with logistic regression is more complicated than binary classification and uses the “softmax” function, a description of which is outside the scope of this document. However, the general concept and output of the model is similar in that a category is predicted, and the probabilities associated with each possible category are provided.

Figure 11 – Linear versus logistic models from <https://www.saedsayad.com/logistic_regression.htm>

## Support Vector Machine (SVM)

**Related code in svm\_gridsearch.ipynb and binary\_svm\_gridsearch.** A thoroughly detailed explanation of support vector machines is beyond my current knowledgebase. The gist is that data which is not easily separable for classification in a particular dimensional space can be raised dimensionally via linear algebra techniques in a manner that makes separating it more straightforward. This process is sometimes referred to as “kernelling”. Figure 12 shows an extremely simplified illustration of this concept where a 1-dimensional dataset is raised to two dimensions and can then be separated in a linear fashion. SVMs often work in high dimensional spaces, and the further raising of dimensionality is done via some form of dot product projection and instead of a 1d line, the dividing construct is an n-dimensional plane. The term hyperplane may be used to refer to the dividing construct even in the 1d case.

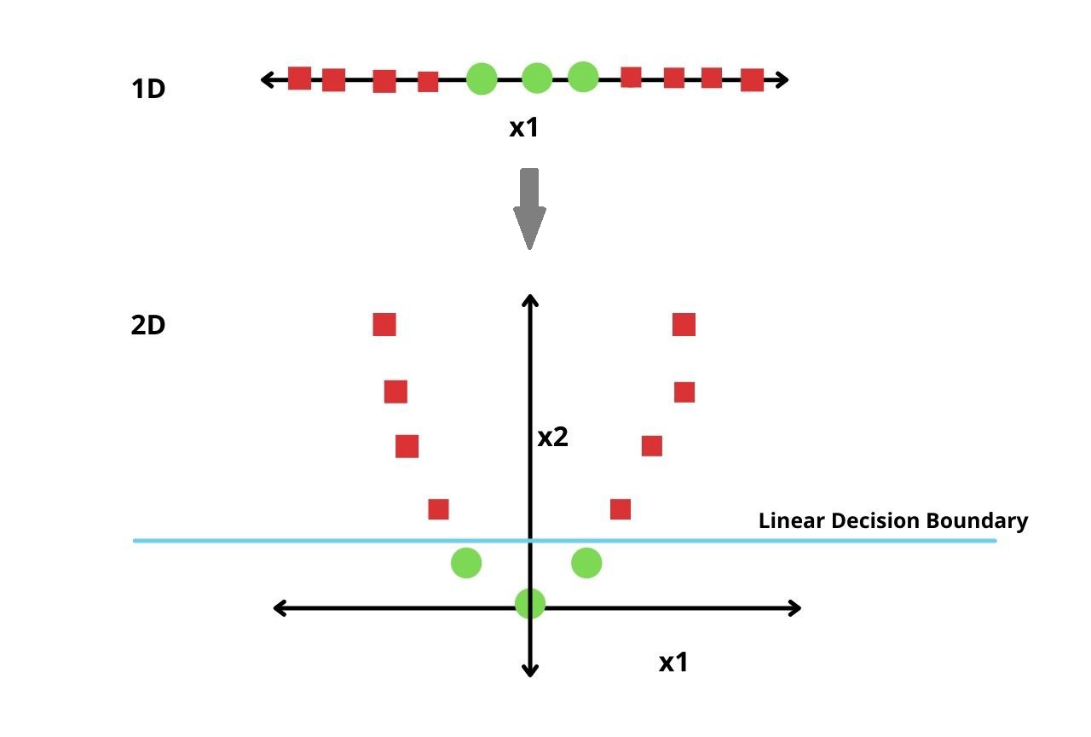


Figure 12 – Applying a function to increase dimensionality of data allows class (green circle vs red square) separation via a linear boundary

Chart, scatter chart

Description automatically generated

Figure 13 A more realistic, yet still relatively simple raise in dimensionality. From <https://medium.com/@zachary.bedell/support-vector-machines-explained-73f4ec363f13>

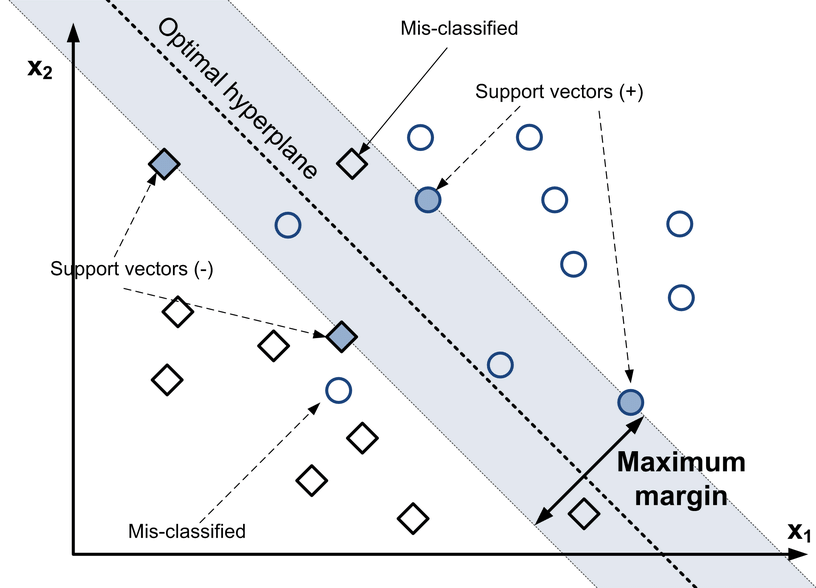


Figure 14 Some of the core concepts of an SVM. “The optimal hyperplane separates two classes of instances: square and circle. The dark-filled instances are called Support Vectors.“ …from <https://www.researchgate.net/figure/Concept-of-Support-Vector-Machines-The-optimal-hyperplane-separates-two-classes-of_fig1_315784930>

## Multi-Layer Perceptron

**Related code is in mlp.ipynb and binary\_mlp.ipynb.** The multilayer perceptron (MLP) is a feedforward artificial neural network (ANN). A description of neural networks is outside the scope of this document. The following are links to reasonable overviews of concepts:

ANNs:

<https://www.ibm.com/cloud/learn/neural-networks>

<https://developer.ibm.com/articles/l-neural/>

MLPs:

<https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141>

<https://wiki.pathmind.com/multilayer-perceptron>

<https://www.dtreg.com/solution/multilayer-perceptron-neural-networks>

# Performance Summaries As of 6/9/2022

Each of the following models’ hyperparameters were optimized with GridSearchCV and then run iteratively **100 times** on random test/train splits. The average accuracy, sensitivity and specificity are shown below. While individual run may vary based on the particular samples used in training vs testing, the average across so many runs should provide an idea as to the performance you can expect from the model on future data after training on the full set.

Note, for multi-classifier models, it is possible to have less than perfect accuracy but achieve 1.0 for both Sensitivity and Specificity as any positive label prediction is counted as a true positive if the correct value is a positive label, regardless of subclassification.

**Logistic Regression - Multinomial Classifier:**

Average Accuracy: 0.736

Average Sensitivity: 0.853

Average Specificity: 0.941

**Logistic Regression - Binary Classifier:**

Average Accuracy: 0.911

Average Sensitivity: 0.927

Average Specificity: 0.891

**Multilayer Perceptron – Binary Classifier:**

Average Accuracy: 0.914

Average Sensitivity: 0.896

Average Specificity: 0.936

**Multilayer Perceptron – Binary Classifier:**

Average Accuracy: 0.737

Average Sensitivity: 0.877

Average Specificity: 0.939

**Support Vector Machine – Binary Classifier:**

Average Accuracy: 0.906

Average Sensitivity: 0.872

Average Specificity: 0.943

**Support Vector Machine – Multinomial Classifier:**

Average Accuracy: 0.684

Average Sensitivity: 0.778

Average Specificity: 0.979