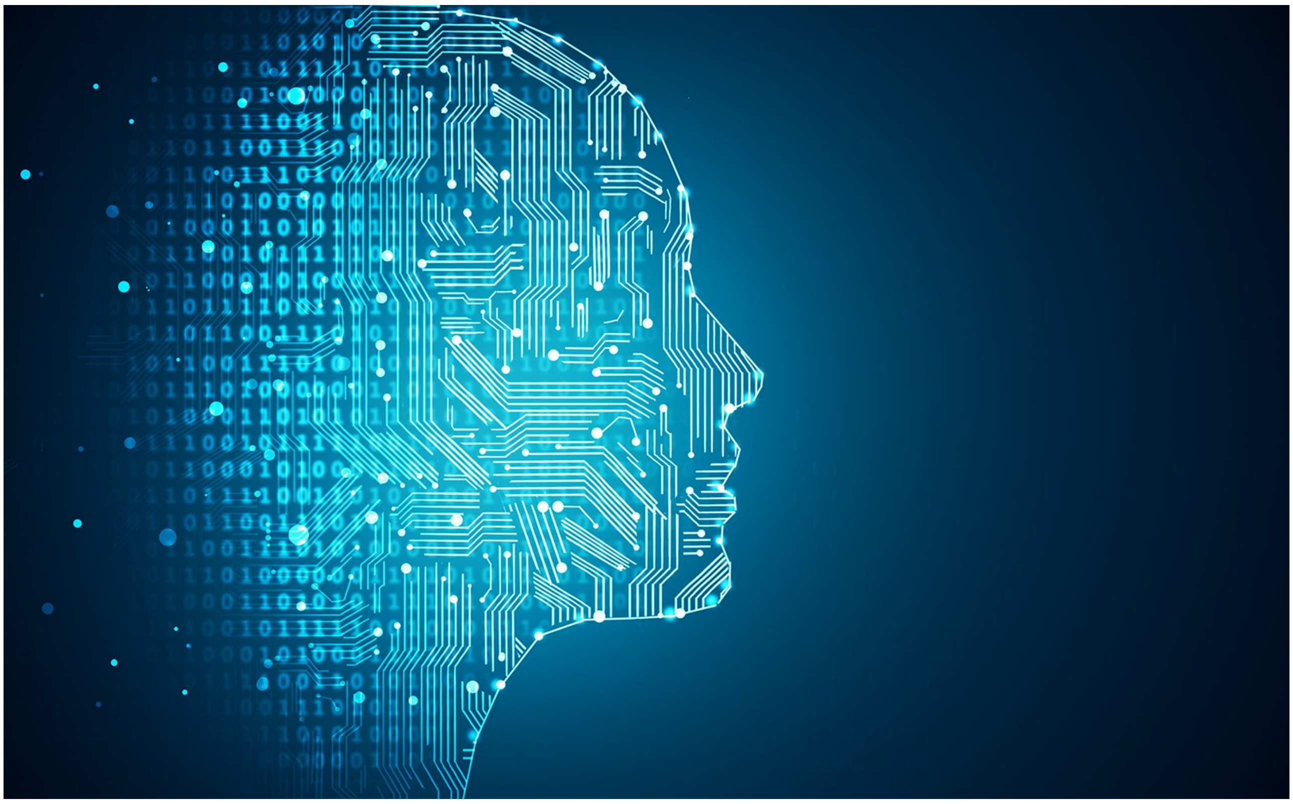
ECEN 4532 - Lab 6: Machine Learning

Andrew Teta

May 1, 2019



Contents

1 Introduction

Researchers in many different fields often collect large quantities of data, with the hope of finding it useful to develop predictions of various phenomena. One very common example in today's age would be advertisement targeting. As users browse the web, certain details about the demographic of people viewing and purchasing products can be collected and stored as a large data set. Machine learning may be a good way to predict what a random user will purchase based on their purchase history or demographic relationship. Many other problems similar to this can be solved using tools like machine learning. Often, they are problems of classification. By classifying users based on their gender, age, region of residency, or other factors, a program or algorithm can predict an outcome for a given activity. The classification involved may be binary or higher order, where a statistic could fall into one of many classification "bins". Other problems may involve continuous predictions.

When discussing the concept of machine learning, it is helpful to have some motivation and notation. Considering the first example of this lab, we have a domain of objects (passengers on the Titanic) which we would like to classify into categories ("survived or "perished"). Often, the objects are too complex to classify directly and so instead, an array of characteristics for each object are defined as *features*. These features are grouped into a *feature vector*. Each feature vector is referred to as an *instance* and the set of all feature vectors, the *instance space*.

We will then take a "training set" of data, which correctly maps the instances to its classification. The training set is incomplete and only contains a sample of all the possible instances, so doing our best, we can "learn" an approximation to the real function mapping features to classifications.

In this lab, we will investigate binary classification using Support Vector Machines, linear regression for predicting a continuous quantity, and multi-class classification using neural networks. The approach here is in the context of data analysis.

2 Background

Three data sets will be used for three different classification techniques, respectively. Specifically, we consider:

- Support Vector Machine (SVM) to predict who survived the sinking of the Titanic. This is an example of a binary classification problem (i.e. "survived" or "perished").
- Linear regression to predict the water resistance of a sailing yacht based on a selection of design parameters. The predicted value will be a real number here, so this is an example of a continuous classification problem.
- Neural networking to recognize hand-written digits. This is a multi-class classification problem as there are 10 classes or "bins".

For all cases, we will utilize a set of labeled outcomes. As an example, for hand-written digit recognition, we have 70,000 images of numerals 0 through 9, correctly labeled. Thus, we are practicing "supervised" machine learning by fitting an appropriate function to a data set based on a large number of input/output pairs.

2.1 Support Vector Machines

A Support Vector Machine (SVM) is a method of binary classification through supervised learning. A model is built based on a training set which can assign a new data point to one classification or the other. It is easiest to understand the process and goal of fitting this model with a graphical representation. Figure 1 shows that an SVM aims to find the optimal "line", called a *hyperplane* between data points falling into either classification.

Calling the hyperplane a "line" is inaccurate because really it forms a plane of separation between the feature vectors, represented by red squares and blue circles in fig. 1. This plane lives in multidimensional space. If each support vector has dimension p, then the hyperplane is of dimension $p-1$.

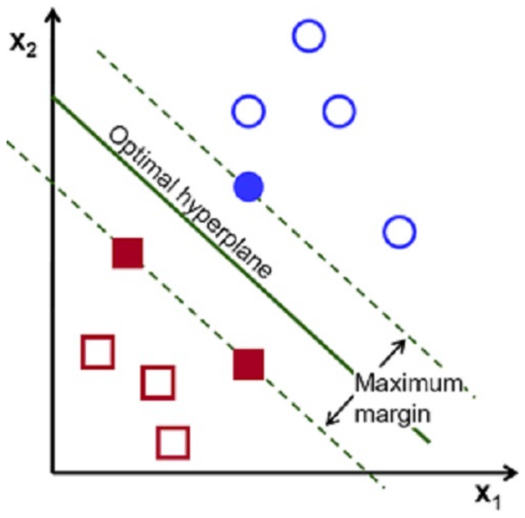


Figure 1: Graphical representation of an SVM hyperplane.
Credit: https://www.aitrends.com/ai-insider/support-vector-machines-svm-ai-self-driving-cars/

As the graphic suggests, another characteristic of SVMs is the margin of the hyperplane. An optimal hyperplane will maximize the margin, effectively finding the largest separation between classifications. Thus, a new data point will fall more decisively on one side of the hyperplane. Furthermore, the vectors defining the margin are called support vectors.

2.2 Linear Regression

Fundamentally, the problem of linear regression seeks to find a "best fit" line passing through a data set, to find a model to which predicted data can be fit. The concept of multiple linear regression deals with more than one independent variable. Linear regression is useful when a data set is "over-determined"–often occurring in scientific or statistical measurement. This would be the case where a solution is desired for only one or two dependent variables, but hundreds or thousands of independent variable samples are collected. Then, rather than a single solution to what has become a linear algebra problem, we now have multiple solutions. Through the following proof, we can show that the "best" solution can be found. We begin with the assumption that we have a set of N measurements, x_i, collected together into a vector,

$$\vec{x} = (x_1, x_2, \ldots, x_N)$$

Recall, the length of $\vec{x}^2$ is its "energy". Additionally, we will refer to a vector of constants, all equal to μ as $\vec{\mu}$. Then, we can define the average energy per vector component of $\vec{x}$ as the error power and call this the *variance*

$$\sigma^2 = \frac{1}{N}|\vec{x} = \vec{\mu}|^2$$

where the vertical bars imply vector length and $\vec{\mu}$ is the mean vector of $\vec{x}$. Finding *standard deviation*, σ, we have:

$$\sigma = \frac{1}{\sqrt{N}}|\vec{x} - \vec{\mu}|$$

Now, considering the case of data point vectors, $\vec{x}$ and $\vec{y}$ (independent/independent variable pairs) we can begin to find a linear regression. First, we remap each vector into unit vector form:

$$\vec{w} = \frac{1}{\sigma_x}(\vec{x} - \vec{\mu_x})$$

$$\vec{z} = \frac{1}{\sigma_y}(\vec{y} - \vec{\mu_y})$$

We now seek a line, defined by the scalar constants a and b, which minimizes the cost (or error power) and write:

$$C(a,b) = |\vec{z} - (a\vec{w} + \vec{b})|^2$$

which can be written as a dot product:

$$\begin{aligned} C(a,b) &= (\vec{z} = (a\vec{w} + \vec{b})) \cdot (\vec{z} - (a\vec{w} + \vec{b})) \\ &= N - 2a\vec{w} \cdot \vec{z} + a^2N + Nb^2. \end{aligned}$$

Then, it is possible to show that $N - 2a\vec{w}\cdot\vec{z} + a^2N$ is a parabola in terms of a and has its minimum at

$$\begin{aligned} a &= \frac{\vec{w}\vec{z}}{N} \\ &= \frac{\vec{w}}{\sqrt{N}} \cdot \frac{\vec{z}}{\sqrt{N}} \\ &= cos\theta \end{aligned}$$

Thus, the value a is the correlation coefficient and is the cosine of the angle between the vectors $\vec{w}$ and $\vec{z}$. For notation consistency with other documentation, we will redefine $a = \rho$ and substitute to arrive at:

$$y = \beta x + (\mu_x - \beta\mu_x), \quad \text{where } \beta = \rho\frac{\sigma_y}{\sigma_x}$$

2.3 Neural Networks

The basic building blocks of a neural network are ”perceptrons” (fig. 2) which compute the dot product of its input vector $\vec{x}$, with its weights, $\vec{w}$. The result is passed through a threshold with respect to a bias, b (fig. 3).

The process of training a neural network optimizes the parameters of the perceptrons in the network. The hidden layers shown in figure 2 can be many layers deep and define the complexity of the network, but eventually narrow the input into a single classification at the output layer.

A good way to think about perceptrons is as if they are the neurons of neural networks. Each perceptron has input weights and a bias value which lead to a calculation of the output value. The output can be an input to many downstream perceptrons and as the network is trained, the perceptron's weights and biases are iteratively adjusted. Therefore, many passes through the training sequence are necessary.

The process of finding the right weights and biases is solved using a *gradient descent* algorithm - searching through a high-dimensional vector space for a solution, where a *back-propagation* algorithm is used to compute the gradient.

Each classification “bin” would be called a class, and the highest value of the output layer of perceptrons indicates the predicted class.

We will not go into any more detail on the specifics of neural network processing in this lab, instead utilizing the `sklearn` `MLPClassifier` library function.

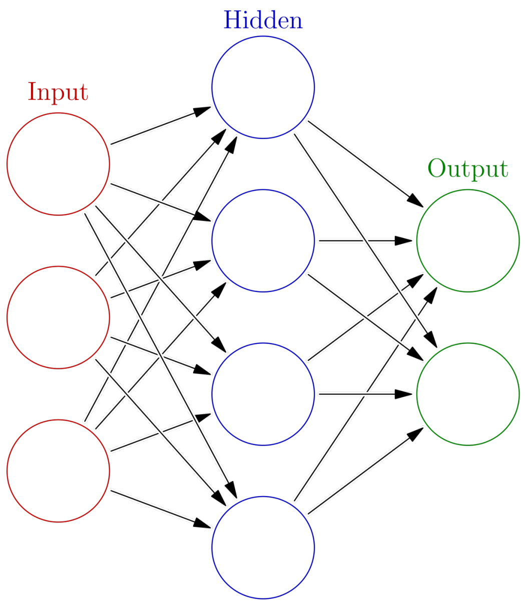


Figure 2: Perceptrons shown on the left.

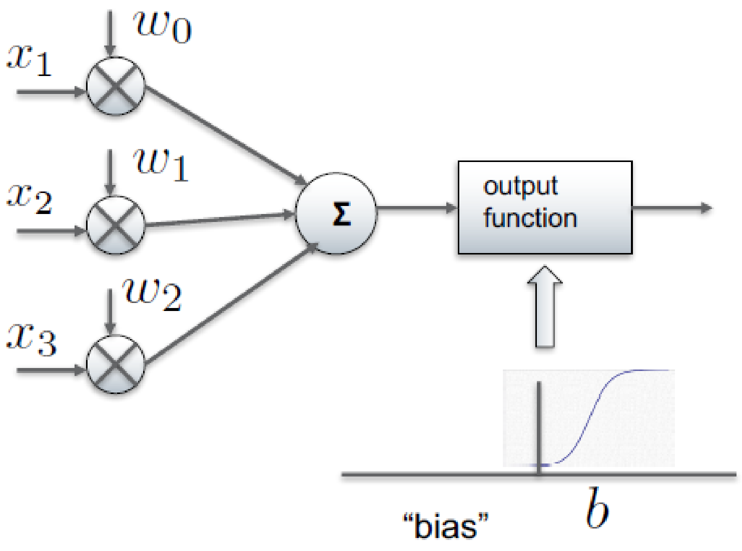


Figure 3: Inside of a perceptron.

3 Procedure

The procedure for this lab is divided into three parts.

1. Binary classification using an SVM
2. Continuous classification using linear regression
3. Multiple classification using a neural network

3.1 SVM

For this portion of the lab, we will be using a training set of data, organized in multiple column fields of a `.csv` file saved to the disk.

The data set we will be using is a collection of information about passengers who were on the Titanic when it sank. The binary classification we wish to predict is whether each passenger "survived" or "perished". For each passenger, the file contains the following columns:

- Class
- Survived
- Name
- Sex
- Age
- Siblings and Parents
- Parents and Children
- Ticket number
- Fare
- Cabin
- Location Embarked
- Boat
- Body Weight
- Destination

Utilizing the Python module, `csv`, we can import this table and convert the fields into a `numpy` matrix of support vectors and a vector of labels. Some entries of this `.csv` table are in text format, others in integer, however all entries are initially imported as character arrays and so must be parsed correctly to prepare the data for processing in an SVM. Thus the data import procedure is:

1. Create `csv` reader object.
2. Iterate through `.csv` file, element by element.
3. Parse individual data fields and build support vectors for each instance
4. Build support vector matrix and label vector

We will not import all columns into our data matrix, but instead use only:

- Class

- Sex
- Age
- Siblings and Parents
- Parents and Children
- Fare
- Location Embarked

and use the "survived" column as our label vector.

Next, we want to analyze this data using the techniques of an SVM. Rather than writing the complex algorithms from scratch, we will utilize a Python module, `SVM` from the package `sklearn`. Then, we feed our support vector matrix and label vector to the `fit()` function to "train" our SVM. Finally, we can run some predictions using the `predict()` function and analyze the performance of our machine. Again, listing the procedure:

1. Declare a `sklearn` SVM object.
2. `fit()` the SVM to our data set, passing the label vector in.
3. `predict()` an outcome for each instance and measure performance.

3.1.1 Results

We started by training the SVM on all features. While the outcome was not bad, the process took a very long time, so we considered limiting the training data set to only three features. Defining the percentage of correctly predicted outcomes as p, we found:

$$p = 80.51\%$$

To find the optimal features to use, while still achieving similar prediction performance, we ran the SVM training and prediction process on each feature individually. The results for each feature are shown below.

Feature	Outcome Success (%)
Fare	66.00
Class	67.68
Sex	77.99
Age	66.08
Siblings and Parents	62.33
Parents and Children	64.32
Location Embarked	64.17

Having obtained some insight into which features are most correlated with the outcome, we recombined multiple features into our training set. Selecting the three features with best prediction accuracy, we found highest overall accuracy based on age, sex, and class to be:

$$p = 78.99\%$$

which is very close to the performance we had with all features considered. This is a much better solution, because the processing time was much less.

3.2 Linear Regression

Again, using `sklearn`, this time we will use a module from the `linear_model` package, called `LinearRegression`. And, you guessed it, this module will help us perform linear regression.

Overall, the procedure is very similar to what we did for the SVM.

- Import training data from a `.csv`
  - Organize independent variables in a matrix and labels into a vector
- `fit()` a model to the data, using the label vector
- `predict` the outcome based on the input
- Statistically measure the performance

For this portion of the lab, we will be using linear regression to fit a model to a set of data containing a list of parameters used in the design of a yacht hull. These design parameters are suspected to contribute to the resulting resistance seen by the yacht against the water. The purpose of this model will then be to accurately measure which parameters contribute most heavily to yacht resistance and fit a model which could then be used to optimize a design.

The design parameters considered are:

- Longitudinal Position Center of Buoyancy
- Prismatic Coefficient
- Length-Displacement Ratio
- Beam-Draught Ratio
- Length-Beam Ratio
- Froude Number

And, more formally, the value we wish to predict is the "Residuary resistance per unit weight of displacement."

3.2.1 Measuring Performance

It will be important to compare the performance of two regression model predictors and so we will define the two terms,

- Total Sum of Squares (TSS)
- Residual Sum of Squares (RSS)

Where,

$$TSS = \sum_{i=1}^{M} (y_i - \mu_y)^2$$

and

$$RSS = \sum_{i=1}^{M} (y_i - \hat{y}_i)^2.$$

Then, the "Fraction of Variance Unexplained" is defined as

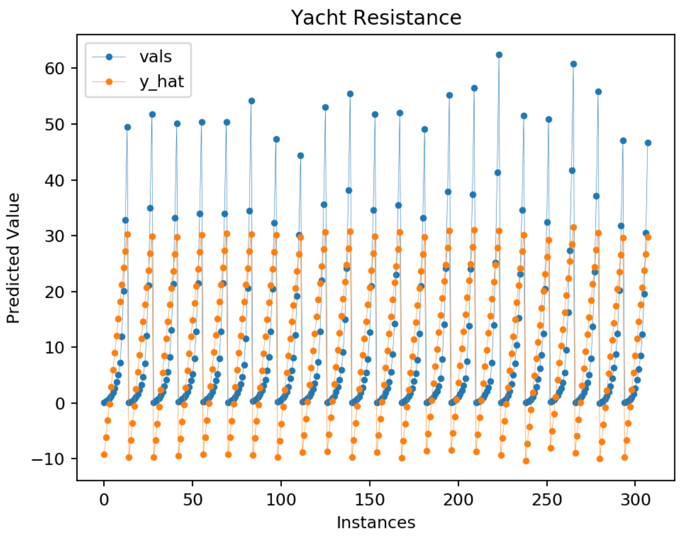


Figure 4: Comparison of labeled outcomes and predicted outcomes based on a linear regression learning method.

$$FVU = \frac{RSS}{TSS}$$

and the "Coefficient of Determination", R^2 as

$$R^2 = 1 - FVU = 1 - \frac{RSS}{TSS}.$$

The value R^2 is essentially the fraction of variance of y (the outcome) that is "explained" by the prediction. As follows, the quantity, $1 - R^2$ is the fraction of variance that is "unexplained" by the prediction. Thus, we should attempt to maximize the value of R^2 to find the best prediction model.

3.2.2 Results

To begin, we `fit()` a linear regression model to all six design parameters over a range of over 300 sample data points for each. Running the prediction over the training set, we found the value of $R^2 = 0.6575$. Figure 4 shows graphically what this outcome looks like. This R^2 value is not great, but there are a couple things to consider. First, not all the design parameters are guaranteed to have a significant effect on the outcome, and so could be introducing unnecessary model parameters. Secondly, and also more importantly, we have just trained this regression model with the assumption of a linear relationship between each of the design parameters and outcome.

Considering first, that some parameters may not be contributing much, we ran the same procedure of fitting, predicting, and calculating R^2 for each parameter set alone. The results are shown in table 3.2.2.

Design Parameter	R^2
Longitudinal Position Center of Buoyancy	0.000372
Prismatic Coefficient	0.000816
Length-Displacement Ratio	$8.8e^{-6}$
Beam-Draught Ratio	0.000154
Length-Beam Ratio	$1.05e^{-6}$
Froude Number	0.656

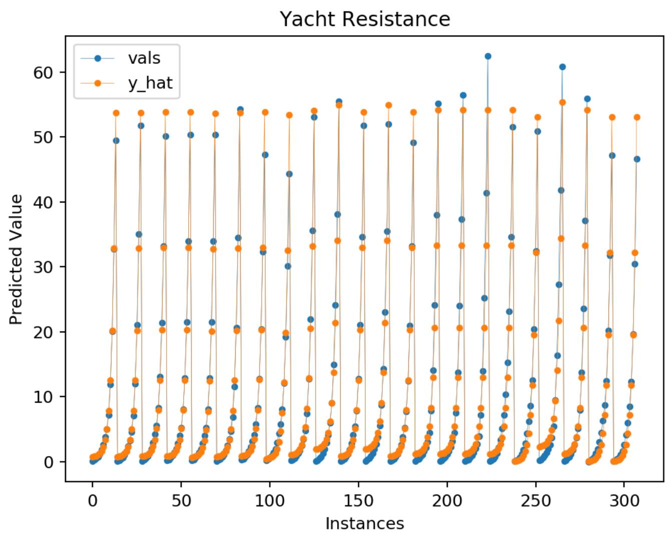


Figure 5: Residuary resistance per unit weight of displacement (labeled vs. predicted).

It is easy to see that most parameters do not contribute much to the variance of the outcome, with the highest contenders being:

1. Froude Number
2. Prismatic Coefficient
3. Longitudinal Position Center of Buoyancy

Taking this into consideration, we ran the procedure again, using only these three parameters. As may have been expected, however, we found $R^2 = 0.6574$, with the remaining significant figures attributed to the other parameters. Since this obviously did not improve our prediction results, we had to look for another cause.

To gain some insight into the relationship between each parameter and the outcome, we plotted all data points for each, as shown in figure 6. It can be seen than the Prismatic coefficient and Longitudinal positional center of buoyancy do not hold a strong relationship. However, the Froude number appears to fit an exponential relationship.

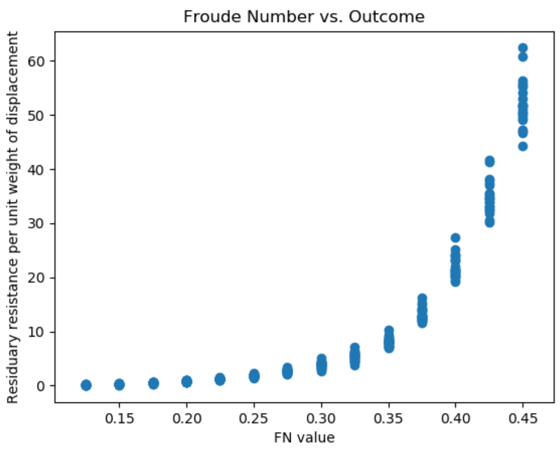
It would be logical to assume a form $y = e^x$ for the Froude number. Taking $y = e^{fn}$, we found $R^2 = 0.6934$. After some trial and error, a quickly deduced "optimal" solution was found for $y = e^{20 \cdot fn}$, resulting in $R^2 = 0.9856$. Then, for ease of calculation, we combined the Prismatic coefficient, Longitudinal positional center of buoyancy, and Froude number vectors into a single matrix, M and calculated

$$R^2 = 0.9870$$

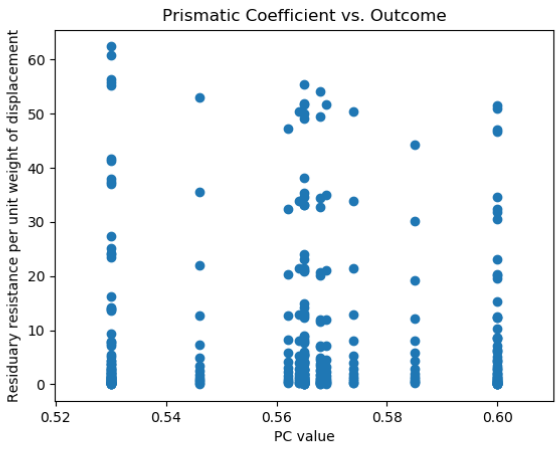
for

$$e^{20 \cdot M}.$$

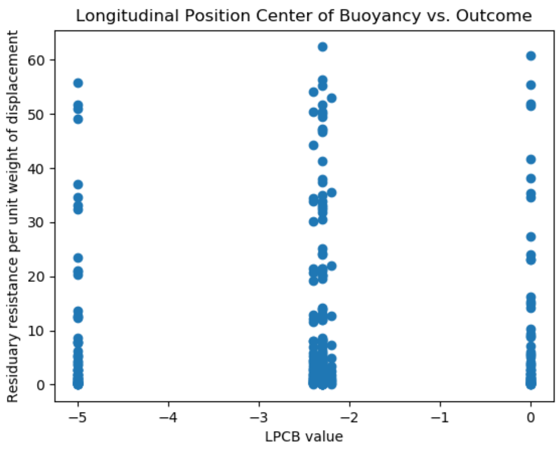
Additionally, the labels and prediction values were plotted together and are shown in figure 5. Not bad.



(a)



(b)



(c)

Figure 6: The relationship between input and output of the three highest scoring parameters.

$$\begin{array}{c|cccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 876 & 0 & 4 & 2 & 1 & 8 & 7 & 1314 & 2 & 0 \\
1 & 0 & 975 & 3 & 3 & 1 & 5 & 3 & 2 & 7 & 1 \\
2 & 11 & 2 & 883 & 12 & 14 & 2 & 10 & 12 & 40 & 4 \\
3 & 1 & 1 & 27 & 894 & 0 & 33 & 4 & 14 & 21 & 5 \\
4 & 1 & 1 & 3 & 1 & 773 & 0 & 12 & 2 & 6 & 25 \\
5 & 3 & 1 & 4 & 6 & 6 & 279 & 6 & 1 & 6 & 1 \\
6 & 11 & 3 & 6 & 0 & 17 & 15 & 818 & 1 & 5 & 0 \\
7 & 5 & 14 & 20 & 6 & 8 & 1 & 0 & 914 & 3 & 29 \\
8 & 7 & 5 & 7 & 17 & 9 & 29 & 11 & 6 & 725 & 9 \\
9 & 14 & 6 & 1 & 9 & 42 & 9 & 1 & 12 & 1 & 863
\end{array}$$

Figure 7: Confusion matrix for a neural network with a single hidden layer of size 10.

3.3 Neural Network

For this portion of the lab, we will again use data stored on the disk to train an algorithm to predict an outcome. However, there are a few differences. First, we will be using raw binary data, sorting through it and passing it to an `sklearn` neural network function for fitting and predicting.

This time, we begin with 70,000 images of hand-written numeric characters, all of which are labeled correctly. We will consider 60,000 of them as part of a training set and the remaining 10,000 as a test set.

We will use the training set of data points and labels to `fit()` a neural network model. Then, we will `predict()` the outcome using the test set of data points.

We will then `score()` the result to find the percentage of correct predictions. As a more thorough method of quantifying classification accuracy, we will compute a *confusion matrix*. In our case, the confusion matrix will measure the correct classification rate of each digit.

For a given classification experiment, we will construct a 10×10 matrix whose rows represent the true digit, and whose columns are the digits as classified by the neural network. Thus, the entry $R(i, j)$ of the matrix is the number of digits of type i classified as type j.

3.3.1 Results

For this portion of the lab, all 70,000 images were encoded already in matrix format. Loading an "X" matrix and "y" vector using `pickle` and taking the values of X to be $X = X/255$, we next needed to sort the data into a more manageable format.

We wanted to construct both a training and test data set, each associated with labels. It was also helpful to have the data organized with digits in order, so we looped through the matrices, constructing $X\_training(10, 6000, 784)$, $X\_test(10, 1000, 784)$, $y\_training(10, 6000, 1)$, and $y\_test(10, 1000, 1)$ where the ten rows correspond to digits (sorted 0-9 ascending) and the columns are image indices. The third dimension (784) is related to 28×28 pixel raw image data. This format was chosen to both sort the data and make it easy to slice a selection of digits out.

We reshaped the X data into 2D matrices and flattened the y vectors, so as to conform to the 2D input expected by the `sklearn MLPClassifier` function. This function has a number of parameters consisting of the number of perceptrons in each hidden layer, how many iterations to perform during training, and some other learning parameters and returns a neural network object. We called `fit(X_training, y_training)` to obtain a model of our data, and then `predict(X_test)` to test our algorithm.

Using a single hidden layer of size 10 with a maximum of 25 iterations to train a model, we found the percentage of correct predictions to be 80.00%. The confusion matrix for this model is shown in figure 7.

Notice, for each character, there is another character that is also predicted relatively often.

Correct character	Classified as	% of the time
0	7	131.4
1	8	0.7
2	8	4.0
3	5	3.3
4	9	3.23
5	8	0.6
6	4	1.7
7	9	2.9
8	5	2.9
9	4	4.2

The extremely erroneous value for 0 shows that this model was poorly trained.

We tried running a few more tests with a single hidden layer of sizes 20, 30, and 50 and a maximum iteration count of 50 (table 3.3.1).

Hidden layer size	Percent correct
20	81.51
30	81.82
50	82.09

Additionally, we ran a prediction with 2 hidden layers of size 50 (table 3.3.1).

Hidden layer size	Percent correct
50	83.29

The results of this experiment show that increased hidden layers and iterations both help the neural network gain accuracy.

4 Conclusion

This lab was a very cool introduction to machine learning. I had no prior experience, so this opened my eyes to a whole new concept. I can think of so many ways this could be applied to real problems - even the yacht example was really interesting. The most impressive part has to be the Python `sklearn` module because it does so much of the heavy lifting. I can imagine knowing how to effectively use the parameters in `MMLPClassifier` is really important when working with a neural network to implement real solutions or validate results.

5 Appendix: Code

```python
import numpy as np
from sklearn import svm
from sklearn.linear_model import LinearRegression
import input_titanic_data as titIn
import csv
import matplotlib.pyplot as plt
from scipy.io import arff
import pickle
from sklearn.neural_network import MLPClassifier
import lab6_funcs as lf

def predict(data, labels, svc):
    # fit SVM object to data set with labeled outcomes
    svc.fit(data, labels)
    # run predictions on training set
    p_val = svc.predict(data)
    # calculate percentage of success of predicted outcomes
    p_success = (1 - p_val[labels != p_val].size / labels.size) * 100
    return p_val, p_success

def linRegPredict(data, labels):
    # fit linear regression model to data set with labeled outcomes
    linReg = LinearRegression().fit(data, labels)
    # run predictions on training set
    p_vals = linReg.predict(data)
    # compute errors
    mu = np.mean(labels)
    tss = np.sum((labels - mu)**2)
    rss = np.sum((labels - p_vals)**2)
    ess = np.sum((p_vals - mu)**2)
    r_squared = 1 - (rss / tss)
    return p_vals, r_squared

def importYacht(filename):
    fd_r = open(filename, 'r')
    datareader = csv.reader(fd_r, dialect='excel')
    X = np.zeros( (0,6), np.float)
    y = np.zeros(0, np.float)

    # Read the labels from the file
    a = next(datareader)

    # extract data from csv into feature matrix and prediction vector
    for ctr, line in enumerate(datareader):
        # --------- X --------- #
        # col_1 = longitudinal position center of buoyancy
        # col_2 = prismatic coefficient
        # col_3 = length-displacement ratio
        # col_4 = beam-draught ratio
        # col_5 = length-beam ratio
        # col_6 = Froude number

        # --------- y --------- #
        # residuary resistance per unit weight of displacement
        X = np.vstack( [X, np.array(line[0:-1], dtype=np.float64)] )
        y = np.hstack( [y, np.array(line[-1:], dtype=np.float64)] )

    return X, y

def loadNNData():
    # load data
    print('loading NN data...')
    X = pickle.load(open('mnist_X_uint8.sav', 'rb'))
    y = pickle.load(open('mnist_y_uint8.sav', 'rb'))
    print('done loading NN data')
    X = X/255
    return X, y

def sortNNData(data, labels):
    X_training = np.zeros((10, 6000, 784))
    X_test = np.zeros((10, 1000, 784))
    y_training = np.zeros((10, 6000, 1))
    y_test = np.zeros((10, 1000, 1))

    # loop over digits
    print('sorting NNData...')
    for digit in range(10):
        imageIndex = 0
        for label in range(len(labels)):
            # find all elements equal to digit (in training set)
            if labels[label] == digit and imageIndex < 6000:
                X_training[digit, imageIndex, :] = data[label]
                y_training[digit, imageIndex] = digit
                imageIndex += 1
            # find all elements equal to digit (in test set)
            elif labels[label] == digit and (imageIndex >= 6000 and imageIndex < 7000):
                X_test[digit, imageIndex - 6000, :] = data[label]
                y_test[digit, imageIndex - 6000] = digit
                imageIndex += 1
    print('done sorting NNData')
    return X_training, X_test, y_training, y_test

def confusionMatrix(y, yhat):
    confMatrix = np.zeros((10,10))
    for i in range(len(y)):
        confMatrix[ (int)(y[i]), (int)(yhat[i]) ] += 1
    return confMatrix


# extract raw data for NN processing
NNData = loadNNData()
NNData_sorted = sortNNData(NNData[0], NNData[1])

X_training = np.reshape(NNData_sorted[0], (-1,784))
X_test = np.reshape(NNData_sorted[1], (-1,784))
y_training = NNData_sorted[2].flatten()
y_test = NNData_sorted[3].flatten()

# Define and train NN model and make predictions
mlp = MLPClassifier(hidden_layer_sizes=10, max_iter=25, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1)
mlp.fit(X_training, y_training)
yhat = mlp.predict(X_test)
cm = confusionMatrix(y_test, yhat)
print(lf.bmatrix(cm.astype(np.int)))

# Define a few more classifier objects for comparison
mlp1_20 = MLPClassifier(hidden_layer_sizes=20, max_iter=50, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1)

mlp1_20.fit(X_training, y_training)
yhat1_20 = mlp.predict(X_test)
print(f'mlp1_20 score = {mlp1_20.score(X_test, y_test)}')

mlp1_30 = MLPClassifier(hidden_layer_sizes=30, max_iter=50, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1)

mlp1_30.fit(X_training, y_training)
yhat1_30 = mlp.predict(X_test)
print(f'mlp1_30 score = {mlp1_30.score(X_test, y_test)}')
```

```python

mlp1_50 = MLPClassifier(hidden_layer_sizes=50, max_iter=50, alpha=1e-4,
                        solver='sgd', verbose=10, tol=1e-4, random_state=1)

mlp1_50.fit(X_training, y_training)
yhat1_50 = mlp.predict(X_test)
print(f'mlp1_50 score = {mlp1_50.score(X_test, y_test)}')

# try with two hidden layers
mlp2_50 = MLPClassifier(hidden_layer_sizes=(50, 50), max_iter=50, alpha=1e-4,
                        solver='sgd', verbose=10, tol=1e-4, random_state=1)

mlp2_50.fit(X_training, y_training)
yhat2_50 = mlp2_50.predict(X_test)
print(f'mlp2_50 score = {mlp2_50.score(X_test, y_test)}')

# import data for yacht design into 2D table
yacht_data, vals = importYacht('yacht_data.csv')

# compute regression model for entire data set
yhat, r2 = linRegPredict(yacht_data, vals)

yhat_lpcb, r2_lpcb = linRegPredict(np.reshape(yacht_data[:,0], (-1,1)), vals)
yhat_pc, r2_pc = linRegPredict(np.reshape(yacht_data[:,1], (-1,1)), vals)
yhat_ldr, r2_ldr = linRegPredict(np.reshape(yacht_data[:,2], (-1,1)), vals)
yhat_bdr, r2_bdr = linRegPredict(np.reshape(yacht_data[:,3], (-1,1)), vals)
yhat_lbr, r2_lbr = linRegPredict(np.reshape(yacht_data[:,4], (-1,1)), vals)
yhat_fn, r2_fn = linRegPredict(np.reshape(yacht_data[:,5], (-1,1)), vals)
print(f'r2 = \n {r2_lpcb}\n{r2_pc}\n{r2_ldr}\n{r2_bdr}\n{r2_lbr}\n{r2_fn}\n')

# best predictors:
# fn    (col 5)
# pc    (col 1)
# lpcb  (col 0)

best_data = np.delete(yacht_data, [2,3,4], axis=1)
yhat_best_linear, r2_best_linear = linRegPredict(best_data, vals)
print(np.shape(np.delete(yacht_data, [2,3,4], axis=1)))

plt.figure()
plt.scatter(np.reshape(yacht_data[:,5], (-1,1)), vals)
plt.title('Froude Number vs. Outcome')
plt.xlabel('FN value')
plt.ylabel('Residuary resistance per unit weight of displacement')
plt.savefig('figures/fn.png')

plt.figure()
plt.scatter(np.reshape(yacht_data[:,0], (-1,1)), vals)
plt.title('Longitudinal Position Center of Buoyancy vs. Outcome')
plt.xlabel('LPCB value')
plt.ylabel('Residuary resistance per unit weight of displacement')
plt.savefig('figures/lpcb.png')

plt.figure()
plt.scatter(np.reshape(yacht_data[:,1], (-1,1)), vals)
plt.title('Prismatic Coefficient vs. Outcome')
plt.xlabel('PC value')
plt.ylabel('Residuary resistance per unit weight of displacement')
plt.savefig('figures/pc.png')

# try a few other models
fn_exp = np.exp(20 * np.reshape(yacht_data[:,5], (-1,1)))
yhat_fn_exp, r2_fn_exp = linRegPredict(np.reshape(fn_exp, (-1,1)), vals)
print(f'exp(20*fn) r2 = {r2_fn_exp}')

fn_exp = np.exp(np.reshape(yacht_data[:,5], (-1,1)))
```

```python
yhat_fn_exp, r2_fn_exp = linRegPredict(np.reshape(fn_exp, (-1,1)), vals)
print(f'exp(fn) r2 = {r2_fn_exp}')

#lpcb_data = yacht_data[:,0]
#pc_data = yacht_data[:,1]
comb_data1 = np.exp(20 * np.delete(yacht_data, [2,3,4], axis=1))
yhat_comb, r2_comb = linRegPredict(comb_data1, vals)
print(f'combined r2 = {r2_comb}')


plt.figure(dpi=170)
plt.plot(range(len(vals)), vals, linewidth=0.25, label='vals', marker='.')
plt.plot(range(len(yhat)), yhat_comb, linewidth=0.25, label='y_hat', marker='.')
plt.legend()
plt.xlabel('Instances')
plt.ylabel('Predicted Value')
plt.title('Yacht Resistance')
plt.savefig('figures/yacht_best.png')

print('yacht...')

# create SVM module
clf = svm.SVC(gamma='scale', kernel='linear')
# extract all titanic training data into data matrix and labels vector
titanic_data, survived = titIn.get_titanic_all('titanic_tsmod.csv')

# train and run predictions on all data
pred_val, percent_success = predict(titanic_data, survived, clf)
print(f'percent correct (all) = {percent_success}')

# extract fare data column and run predictions
fareData = np.reshape(titanic_data[:,-4], (-1,1))
pred_val, percent_success = predict(fareData, survived, clf)
print(f'percent correct (fare) = {percent_success}')

# extract class data columns and run predictions
classData = titanic_data[:,0:3]
pred_val, percent_success = predict(classData, survived, clf)
print(f'percent correct (class) = {percent_success}')

# extract sex data column and run predictions
sexData = np.reshape(titanic_data[:,3], (-1,1))
pred_val, percent_success = predict(sexData, survived, clf)
print(f'percent correct (sex) = {percent_success}')

# extract age data columns and run predictions
ageData = titanic_data[:,4:85]
pred_val, percent_success = predict(ageData, survived, clf)
print(f'percent correct (age) = {percent_success}')

# extract siblings/parents data columns and run predictions
sibspData = titanic_data[:,85:94]
pred_val, percent_success = predict(sibspData, survived, clf)
print(f'percent correct (sibsp) = {percent_success}')

# extract parents/children data columns and run predictions
parchData = titanic_data[:,94:104]
pred_val, percent_success = predict(parchData, survived, clf)
print(f'percent correct (parch) = {percent_success}')

# extract parents/children data columns and run predictions
embarkedData = titanic_data[:,105:108]
pred_val, percent_success = predict(embarkedData, survived, clf)
print(f'percent correct (embarked) = {percent_success}')

# extract highest prediction value data columns and run predictions
```

```python
combData = np.hstack((ageData, classData, sexData))
pred_val, percent_success = predict(combData, survived, clf)
print(f'percent correct (age, sex, class) = {percent_success}')



print('done')
```