Week 13 Lab (Neural Network)

COSC 3337 Dr. Rizk

About The Data

Our goal for this lab is to construct a model that can take a certain set of features related to the Titanic and predict whether a person survived or not (0 or 1). Since we're trying to predict a binary categorical variable (1 or 0), logistic regression seems like a good place to start from.

The dataset that we'll be using for this task comes from kaggle.com and contains the following attributes:

- PassengerId
- Survived (0 or 1)
- Pclass: Ticket class (1, 2, or 3 where 3 is the lowest class)
- Name
- Sex
- Age: Age in years
- SibSp: # of siblings / spouses aboard the Titanic
- Parch: # of parents / children aboard the Titanic
- Ticket: Ticket number
- Fare: Passenger fare
- Cabin: Cabin number
- Embarked: Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

Note: This lab will be the same as lab 5 (logistic regression). The only difference is that here we will be using a neural network instad. You're welcome to skip to the creating our neural network section if you're already familiar with this dataset.

Exploratory Data Analysis

Let's begin by importing some necessary libraries that we'll be using to explore the data.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
from matplotlib import rcParams
rcParams['figure.figsize'] = 15, 5
sns.set_style('darkgrid')
```

Our first step is to load the data into a pandas DataFrame

In [3]:
```
titanic_data = pd.read_csv('titanic.csv')
titanic_data.head()
```

Out[3]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

From here, it's always a good step to use ***describe()*** and ***info()*** to get a better sense of the data and see if we have any missing values.

In [4]:
```
titanic_data.describe()
```

Out[4]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

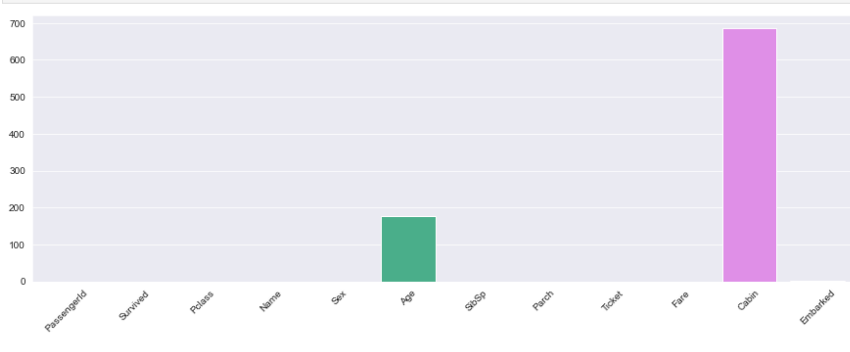
We can see that Age, Cabin, and Embarked contain missing values since this dataset contains 891 entries in total, and Age, Cabin, and Embarked only contain 714 non-null entries, 204 non-null entries, and 889 non-null entries respectively. Thus, we will have to take care of these missing values.

In [5]:
```
titanic_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Note, we can also make a plot of our missing data if we'd prefer to visualize it. Here we use seaborn's barplot ***sns.barplot(x, y)*** and pass our DataFrame's columns as the x axis and the sum of all missing values in each column in the y axis. since Embarked only has 2 missing values, it's very hard to see, but there's a slight raise in the y axis under Embarked.

In [6]:
```
sns.barplot(x=titanic_data.columns, y=titanic_data.isnull().sum().values)
plt.xticks(rotation=45)
plt.show()
```



Tip: If you're ever confused how a chained line of code works in this course, just break it down into multiple steps. For example, say you didn't know how the piece of code above 'y=titanic_data.isnull().sum().values' gives us all of the missing values. Well, let's break it down. ***titanic_data.isnull()*** gives us back the original DataFrame (titanic_data), but with True and False values placed where there is a missing value.

In [7]:
```
titanic_data.isnull()
```

Out[7]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	False	False	False	False	False	False	False	False	False	False	True	False
1	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	True	False
3	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	True	False
...	...	...	...	...	...	...	...	...	...	...	...	...
886	False	False	False	False	False	False	False	False	False	False	True	False
887	False	False	False	False	False	False	False	False	False	False	False	False
888	False	False	False	False	False	True	False	False	False	False	True	False
889	False	False	False	False	False	False	False	False	False	False	False	False
890	False	False	False	False	False	False	False	False	False	False	True	False

891 rows × 12 columns

Then calling ***.sum()*** off of this gives us back a Series telling us how many true (missing values) were in each column. Recall that True is an alias for 1, which is why we can take the sum of True False columns.

In [8]:
```
titanic_data.isnull().sum()
```

Out[8]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Finally, if you remember from lab 3, calling ***.index*** on this will give us the index labels (left side), and ***.values*** will give us the missing value counts for each column (right side), which is the array that we passed in as y.

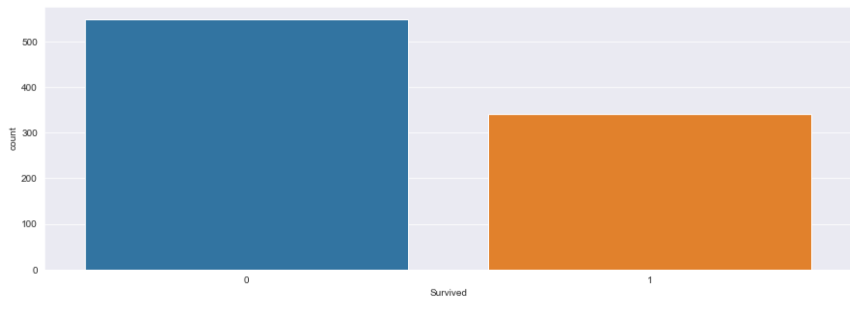
In [9]:
```
titanic_data.isnull().sum().values
```

Out[9]:
```
array([  0,   0,   0,   0,   0, 177,   0,   0,   0,   0, 687,   2])
```

Keep this tip in mind when exploring other people's notebooks on github or kaggle, since you'll soon find out that it's very common on kaggle for people to chain functions together, which can sometimes be hard to understand at first, but much easier to understand once you break it down into smaller chunks.

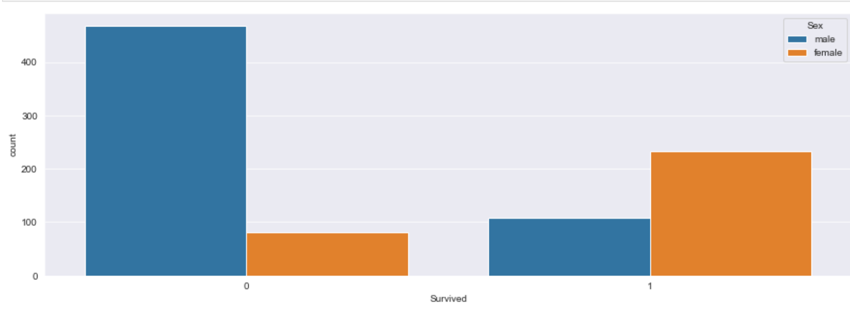
Let's continue on with our data exploration by next seeing how many people survived (1) and did not survive (0) in our dataset. To accomplish this, we can pass any column in our DataFrame into ***sns.countplot(x)***, which will list all of the unique values in that column along the x-axis, and plot the total counts for each unique value along the y-axis. So here we can see that majority of the people in our dataset did not survive (0).

In [10]:
```
sns.countplot(x=titanic_data['Survived'])
plt.show()
```



Did more men or females survive? Recall that ***hue*** parameter seaborn gives us access too. This will let us expand on the previous graph by also telling us how many from each value (0 or 1) were male and female.

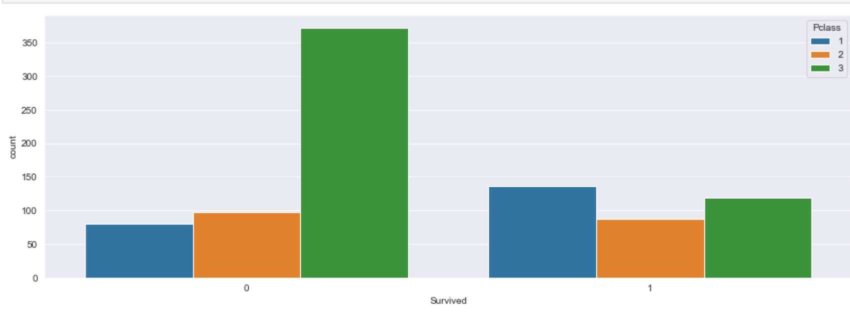
In [11]:
```
sns.countplot(x=titanic_data['Survived'], hue='Sex', data=titanic_data)
plt.show()
```



Interpretation: We can see that from those who did not survive (0), majority of them were male.

How about from ticket class? Was the lower class less likely to survive?

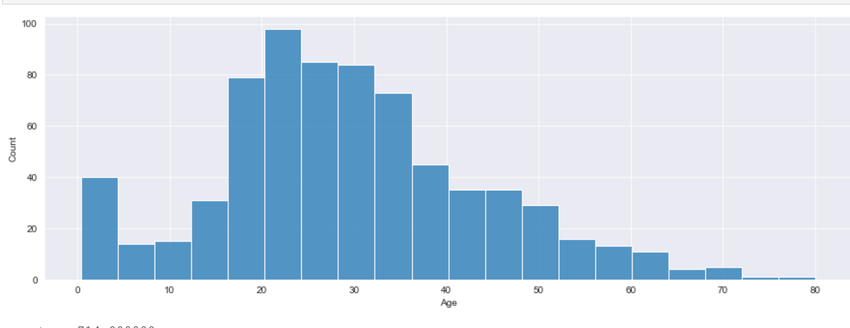
In [12]:
```
sns.countplot(x=titanic_data['Survived'], hue='Pclass', data=titanic_data)
plt.show()
```



Interpretation: We can see that from those who did not survive (0), majority of them were from the lower class, 3.

What did the Titanic age distribution look like?

In [13]:
```
sns.histplot(x=titanic_data['Age'].dropna())
plt.show()
titanic_data['Age'].describe()
```

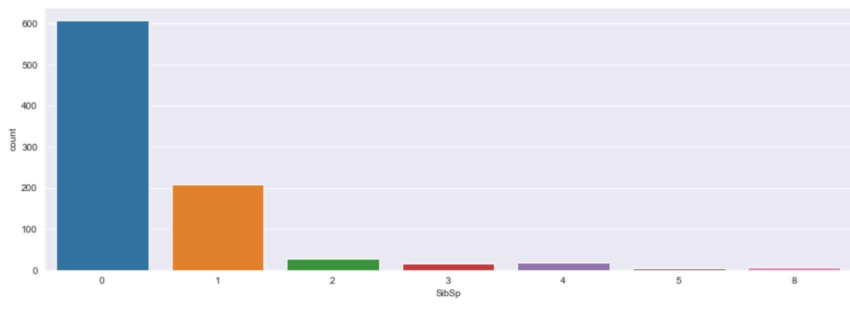


Out[13]:
```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
Name: Age, dtype: float64
```

Interpretation: The average age on the Titanic seems to be ~30, with 75% of people onboard being 38 years of age or younger.

What's the most common number of siblings one had with them on the Titanic?

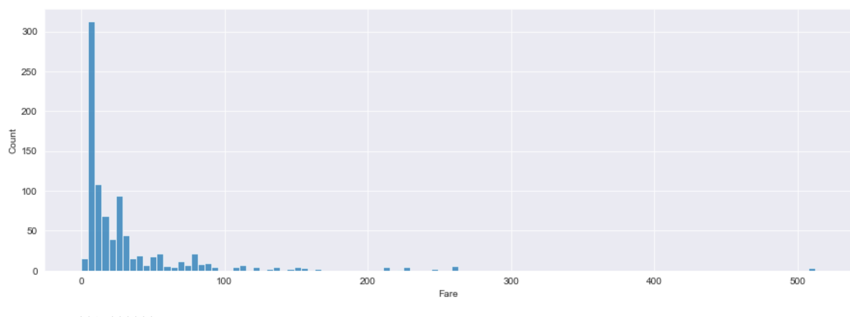
In [14]:
```
sns.countplot(x=titanic_data['SibSp'])
plt.show()
```



Interpretation: Majority of those onboard had 0 siblings/spouses also onboard, with the 2nd most popular being having 1 sibling/spouse onboard (most likely that 1 person onboard was a spouse).

What was the Fare distribution on the Titanic? How much did the average person pay?

In [15]:
```
sns.histplot(x=titanic_data['Fare'])
plt.show()
titanic_data['Fare'].describe()
```



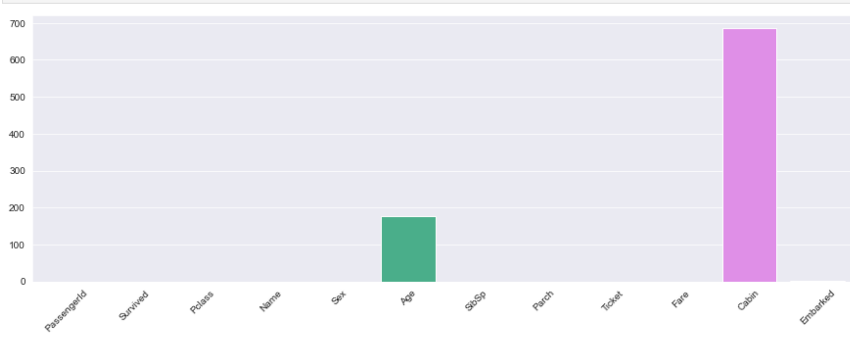
Out[15]:
```
count    891.000000
mean      32.204208
std       49.693429
min        0.000000
25%        7.910400
50%       14.454200
75%       31.000000
max      512.329200
Name: Fare, dtype: float64
```

Interpretation: The average person paid 32.204208, with 75% of people paying 31.000000 or less. One interesting note is that the min is 0. This could mean that there were people unaccounted for who managed to sneak in for free. Or someone who won a free ride or something.

Data Preprocessing

Let's first take care of our missing values. Recall how much data was missing:

In [16]:
```
sns.barplot(x=titanic_data.columns, y=titanic_data.isnull().sum().values)
plt.xticks(rotation=45)
plt.show()
```

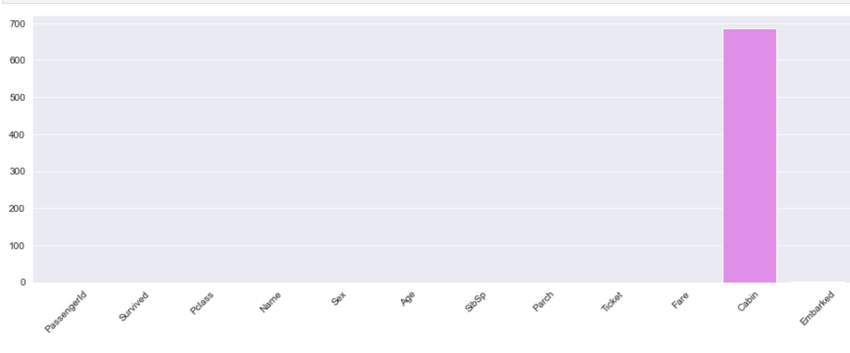


For Age, our best bet would be to impute any missing values with the mean age. We can do this very quickly with pandas ***.apply(func)***. This will apply any function to every value along a column. If you're not familiar with lambda functions, you can create a normal python function that accepts the age and mean_age, and returns the mean age if age is null, or the age itself if it's not null. Then you can supply that function to ***.apply(func)***. So here we're reassigning the titanic_data['Age'] column to titanic_data['Age'] after our function has been applied on it, which will essentially fill any missing age values with the mean age calculated.

In [17]:
```
mean_age = int(titanic_data['Age'].mean())
titanic_data['Age'] = titanic_data['Age'].apply(lambda age : mean_age if pd.isnull(age) else age)
```

If we recreate our missing data plot, we can see that there are no longer any missing Age values.

In [18]:
```
sns.barplot(x=titanic_data.columns, y=titanic_data.isnull().sum().values)
plt.xticks(rotation=45)
plt.show()
```



For Cabin, we have so much data missing (more missing than non-null data) that performing any type of imputation seems like a bad idea since we don't have much original data to work with. For this reason, we will just drop this column. I will go ahead and also drop the 2 missing Embarked rows while we're at it, but you can choose to keep them if you'd like and impute them.

In [19]:
```
titanic_data.drop(labels=['Cabin'], axis=1, inplace=True)
titanic_data.dropna(inplace=True)
```

Recalling ***.info()***, we can see that there are no more missing values in this dataset.

In [20]:
```
titanic_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 889 entries, 0 to 890
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  889 non-null    int64
 1   Survived     889 non-null    int64
 2   Pclass       889 non-null    int64
 3   Name         889 non-null    object
 4   Sex          889 non-null    object
 5   Age          889 non-null    float64
 6   SibSp        889 non-null    int64
 7   Parch        889 non-null    int64
 8   Ticket       889 non-null    object
 9   Fare         889 non-null    float64
 10  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(4)
memory usage: 83.3+ KB
```

Our next step is to handle categorical variables since machine learning algorithms can only understand numbers. The variables to consider are Name, Sex, Ticket, and Embarked. We'll use dummy variables for Sex and Embarked and drop Name and Ticket. You can choose to do some type of feature engineering on Name and Ticket and compare it with our model without these features if you wish.

Recall that a dummy variable is a variable that takes the value 0 or 1 to indicate the absence or presence of some category. Pandas has a convenient function ***pd.get_dummies(data, columns)*** that will automatically assign dummy variables for us. For example, if we include Sex in columns, it will create 2 new columns (sex_male, sex_female) and place a 1 for the one that's true, and 0 in the other. So if a specific observation is female, we will place a 1 in sex_female and 0 in sex_male. One important note is that you should always add an additional ***drop_first=True*** parameter when using get_dummies. This will drop one of the columns created in the dummy process, since keeping all of them will result in multicollinearity.

In [21]:
```
titanic_data = pd.get_dummies(data=titanic_data, columns=['Sex', 'Embarked'], drop_first=True)
titanic_data.drop(labels=['Name','Ticket'], axis=1, inplace=True)
```

Now that our data is in the correct form, we're ready to begin building our model.

In [22]:
```
titanic_data.head()
```

Out[22]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare	Sex_male	Embarked_Q	Embarked_S
0	1	0	3	22.0	1	0	7.2500	1	0	1
1	2	1	1	38.0	1	0	71.2833	0	0	0
2	3	1	3	26.0	0	0	7.9250	0	0	1
3	4	1	1	35.0	1	0	53.1000	0	0	1
4	5	0	3	35.0	0	0	8.0500	1	0	1

Creating our Neural Network Model

We're now ready to begin creating and training our model. We first need to split our data into training and testing sets. This can be done using sklearn's ***train_test_split(X, y, test_size)*** function. This function takes in your features (X), the target variable (y), and the test_size you'd like (Generally a test size of around 0.3 is good enough). It will then return a tuple of X_train, X_test, y_train, y_test sets for us. We will train our model on the training set and then use the test set to evaluate the model.

In [23]:
```
from sklearn.model_selection import train_test_split

X = titanic_data[['PassengerId', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare','Sex_male', 'Embarked_Q',
                  'Embarked_S']]
y = titanic_data['Survived']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

We'll now import sklearn's Multi Layer Perceptron (MLP) model and begin training it using the ***fit(train_data, train_data_labels)*** method. In a nutshell, fitting is equal to training. Then, after it is trained, the model can be used to make predictions, usually with a ***predict(test_data)*** method call.

In [27]:
```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(max_iter=500, activation='relu')
mlp.fit(X_train,y_train)
```

Out[27]:
```
MLPClassifier(max_iter=500)
```

Model Evaluation

Now that we've finished training, we can make predictions off of the test data and evaluate our model's performance using the corresponding test data labels.

In [28]:
```
predictions = mlp.predict(X_test)
```

Since we're now dealing with classification, we'll import sklearn's ***classification_report*** and ***confusion_matrix*** to evaluate our model. Both of these take the true values and predictions as parameters.

In [29]:
```
from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(y_test,predictions))
print(classification_report(y_test,predictions))
```

```
[[143  25]
 [ 41  58]]
              precision    recall  f1-score   support

           0       0.78      0.85      0.81       168
           1       0.70      0.59      0.64        99

    accuracy                           0.75       267
   macro avg       0.74      0.72      0.72       267
weighted avg       0.75      0.75      0.75       267
```

Not bad, but we could do better. Let's try to adjusting the default parameters.

In [46]:
```
mlp2 = MLPClassifier(max_iter=3000, activation='logistic', hidden_layer_sizes=(1000,900))
mlp2.fit(X_train,y_train)

predictions2 = mlp2.predict(X_test)

print(confusion_matrix(y_test,predictions2))
print(classification_report(y_test,predictions2))
```

```
[[136  32]
 [ 23  76]]
              precision    recall  f1-score   support

           0       0.86      0.81      0.83       168
           1       0.70      0.77      0.73        99

    accuracy                           0.79       267
   macro avg       0.78      0.79      0.78       267
weighted avg       0.80      0.79      0.80       267
```

Getting better 🙂, see if you can modify the data or neural network parameters to improve on this. Also, try using a dataset where logistic or linear models give poor results and see if a neural network can outperform it.

Generally, you can increase neural network performace by increasing the epochs and hidden layer size. Note, you shouldn't use a neural network if something can be accomplished with a 'simpler' model like linear or logistic regression. Neural networks are best suited for complex data. In these cases, you'll see much stronger performance between something much simpler like a linear or logistic function. Also, I highly recommend that you check out Tensorflow or PyTorch if you're interested in neural network and deep learning, since sklearn is not as well suited for these types of models.