Detecting harzadous near-Earth objects (NEOs) using logistic regression and multi-layer perceptron (MLP) model

Trung Nguyen

October 9, 2022

1 Introduction

There is an infinite number of stars, planets, and objects in our observable universe, and although there is low probability that one might collide into Earth, many of asteroids are closer to Earth than we realize. The analysis of the distribution of near-Earth objects (NEOs) and detection of potentially dangerous asteroids are therefore imperative to avoid its harmful impact to the lives on Earth as well as ensure the continuous orbits of the active satellites.

This project aims to build a reliable supervised model to detect potential harmful objects using their known characteristics. In Section 2 the problem is formulated into a classic machine learning problem based on a chosen data frame. Section 3 will go deep into the preprocessing of data and how the data is trained using both a logistic regression model and a multi-layer perceptron (MLP) model. Finally, the models are tested using k-fold cross validation and the results are analysed and compared in Section 4. The code used is attached as an appendix to this report.

2 Problem formulation

The preprocessed data is obtained from Kaggle [Kag] based on the raw recorded data from NASA Open API: NEO Earth Close Approaches [fNSC]. Information of the NEOs and their features, marked by their ID and name, are used as datapoints. A datapoint would include two parts: the object's features and its label. An object can be labeled as true or false: true if it has a harmful potential and false if it is not at the time the data is recorded. Hence the label can be change to *false = 0*, *true = 1* and this is the binary classification problem. Below is a chart depicting the percentage of harmful NEOs (Figure 1).

A datapoint is consisted of several features, however in this project we will only consider three relevant features: the object's estimated minimum and maximum diameter (in kilometers), and its intrinsic luminosity (the measure of total energy output independent from distance) [Wik]. From these three features we will build two models to predict the label of the data point: that is if the object is hazardous or not according to its characteristics, and the results are compared to determine the optimal model.

3 Methods

3.1 Preparing data

The dataset consists of 90836 datapoints corresponding to the NEOs in consideration. From the original data obtained from Kaggle, we can preprocess the data with extra step by first dropping the ID, name, orbiting body and sentry object features of the datapoints as these features do not have an impact on the level of harmfulness. Relative velocity and miss distance are not chosen as features for the model because the hazard indicator does not show any dependency to these features. Three features are kept for the building of models: minimum diameter and maximum diameter as they are

connected to the gravitational effects, and luminosity as it indicates the energy level of the objects.

Since the original data is imbalanced, that is the label *false* outweighs the label *true* by a factor of 9, the models are subjects to bias: all of the NEOs are predicted as *false* and still have and accuracy of over 0.9. One way to avoid this is to take a random sample from the *false* dataset with a size of a tenth and merge with the *false* dataset. Once the data is relatively balanced, the labels are converted into *false = 0*, *true = 1* and separated from the data.

At this stage the new data frame only have three features, and two of which are diametric. To reduce the redundancy of variables in the model, we create an average diameter feature by calculating the average of the estimated minimum and maximum diameter (equally weighed), substituting the two existing features. The final data used for the project includes two directly related features for the training model (diameter and luminosity) and the binary label indicating if the NEO is hazardous.

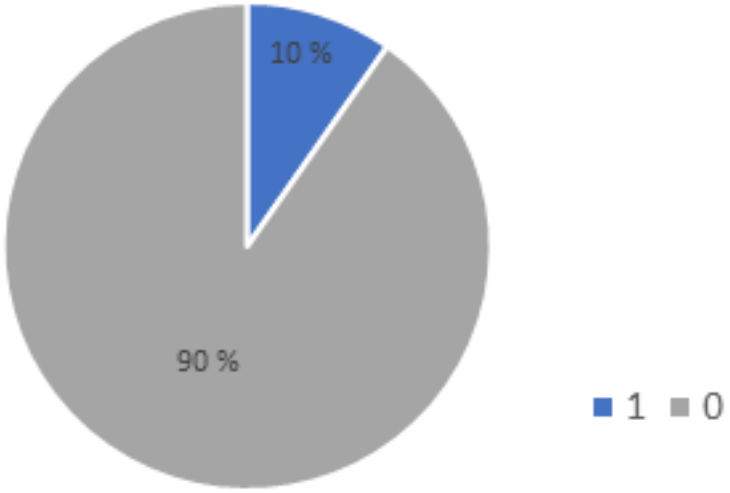


Figure 1: The percentage of the NEOs labels as harzadous

3.2 Model selection

As this is a classic binary classification problem, a practical solution is to apply the linear regression with a logistic loss function. This model applies well to model with labels between 0 and 1, and it can demonstrate the linear relationship between the features and the classification of the label. The logistic loss is chosen as it allowed the use of a ready-made library for logistic regression, as well as avoiding non-convex loss function which leads to possibility of having local minima and not fully optimize the model, a situation which can be caused when we apply the mean-squared error function. Both the model and the loss function are included in *scikit-learn* package [sl] as we import it at the beginning of the project.

In addition to the traditional logistic regression, we aim to fine-tune the method by extending the hypothesis space from linear to a polynomial of degree 2 and then apply a polynomial logistic regression. In this case, the data is preprocessed in one further step by using the polynomial transformation and preprocessing scale, then the transformed data is then trained and validated by the standard logistic regression and logistic loss function.

A second method is to apply the multi-layer perceptron (MLP) model. The choice of this neural network model (which comprises multiple layers of logistic regression models) [3Bl] is to improve the accuracy of the results by training the data through multiple layers compared to a single logistic regression, in expense of more computational resource and longer runtime. Due to the large size of

the dataset, for this model we use a network of 5 layers and 15 neurons at each layers, adding up to 75 neurons in the network. The loss function used for this model is the loss function for neural networks as it allowed the use of a ready-made library for MLP regression and it also account for the backpropagation of the network.

3.3 Model validation

The model accuracy and be validated by using k-fold cross validation, as the data is imbalanced and sensitive to changes of parameter. Due to the large size of data, we choose $k = 5$ to ensure both runtime and randomness of data generation. Hence, each model is trained 5 times with 13632 datapoints and validated with 3408 datapoints. To maximize the randomness of the sets, k-fold shuffle is enabled with 4 was chosen as a seed, and errors of both training and validation sets are recorded.

4 Results and analysis

As this is a classification problem, the average accuracy scores of the validation set are considered when accessing the accuracy of the models. From the first linear regression model, the average accuracy score is 86.92%, and the accuracy score rise to 86.98% after polynomial transformed and scaled. There is an 0.06% increase in accuracy as the hypothesis space of 2 fits the model better with curves. With little runtime difference taken into account, polynomial transfroming and scaling improves the model to a small extent with no impactful drawback.

The accuracy score of the MLP model is 89.63%, which is notably higher than both the original and transformed logistic regression. However, the running time of this method was also much higher due to the fact that this method implement both multi-layer regression and k-fold cross validation. Therefore, MLP model is chosen as the optimal solution for this problem in terms of accuracy, and polynomial linear regression is chosen when runtime efficiency and computational resources are considered.

5 Conclusion

In this report, two methods was chosen for a binary classification problem, and each of the solution has its own advantages. Linear regression and polynomial linear regression offer a model with a relatively high accuracy score of over 85% with little running time. A MLP model can improve this number by over 2.5%, but the runtime and computer resources required are much higher than the two methods mentioned above. One way to address this problem is to adjust the size of the MLP network: fewer neurons and layers to improve runtime and avoid overfitting, and more neurtons and layers to improve the accuracy of the model.

From the results of this project, multiple possible tracks can be further examined, such as analyzing the inverse relationship between the NEO's diameter and luminosity, validating and testing the model on other cosmic objects, and predicting other features of a NEOs such as its velocity and miss distance from the Earth. Overall, the results of the classification model is acceptable, and both methods can be applied depending on the priority of the case.

References

[3Bl] 3Blue1Brown. But what is a neural network? — chapter 1, deep learning. https://www.youtube.com/watch?v=aircAruvnKk.

[fNSC] Center for NEO Studies (CNEOS). Neo earth close approaches. https://cneos.jpl.nasa.gov/ca/.

[Kag] Kaggle. Nasa - nearest earth objects. https://www.kaggle.com/datasets/sameepvani/nasa-nearest-earth-objects?select=neo_v2.csv.

[sl] scikit learn. Logistic regression (aka logit, maxent) classifier. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

[Wik] Wikipedia. Luminosity. https://en.wikipedia.org/wiki/Luminosity.

Appendix

October 8, 2022

1 Introduction

This part of the report include the Python implementation of the Machine Learning methods discussed above. Parts of the codes are distributed as materials in CS-C3240 Machine Learning course at Aalto University, and the student project *Stellar classification with logistic regression*. The appendix will cover the codes for preprocessing the dataframe, visualizing the features and data, creating the validation dataset and building the models. Below is the importation of the libraries neccessary for the project.

[1]:
```
%config Completer.use_jedi = False  # enable code auto-completion

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns  #data visualization library
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error  # evaluation␣
 ↪metrics

from sklearn.model_selection import KFold
from sklearn import preprocessing
from sklearn.preprocessing import PolynomialFeatures # function to generate␣
 ↪polynomial and interaction features
from sklearn.neural_network import MLPRegressor
```

2 Preprocessing and Visualization of The Dataset

In this part we will first import the dataset containing the NEOs and its features to the code. Preprocessing includes dropping irrelevant features and create a new *average diameter* feature subtitute for the other two diameter features. The first 5 datapoints of the new dataframe is printed below.

[2]:
```
# Read in the data stored in the file 'neo_v2.csv'
# Clean the dataframe and modify into necessary features

df = pd.read_csv('neo_v2.csv')
```

```
df.
 ↪drop(columns=['id','name','relative_velocity','miss_distance','orbiting_body','sentry_objec
 ↪ # drop irrelevant columns

df.columns =['minDiameter','maxDiameter','Luminosity','Harzadous'] # rename␣
 ↪columns

# Replace the minimum and maximum size with a new feature: average size
# Drop all datapoints with invalid features
df=df.dropna(axis=0)
data = df.assign(Diameter_avg = (df["minDiameter"] + df["maxDiameter"]) / 2)
data = data.drop(['minDiameter','maxDiameter'],axis=1)
data.head(5)
```

[2]:

	Luminosity	Harzadous	Diameter_avg
0	16.73	False	1.938843
1	20.00	True	0.430073
2	17.83	False	1.168268
3	22.20	False	0.156150
4	20.09	True	0.412613

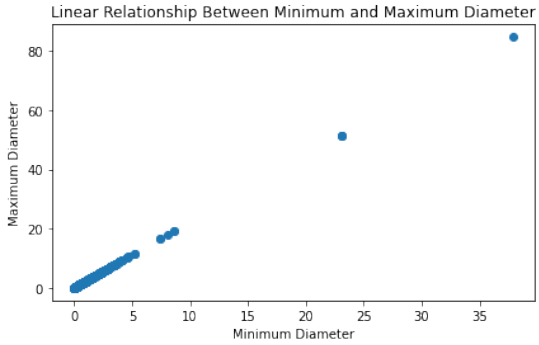
We can justify by visualization that the average diameter is determined correctly as the relationship between minimum and maximum diameter is linear, hence the average is arithmetical. We also have the distribution of NEOs based on their average diameter and intrinsic luminosity below.

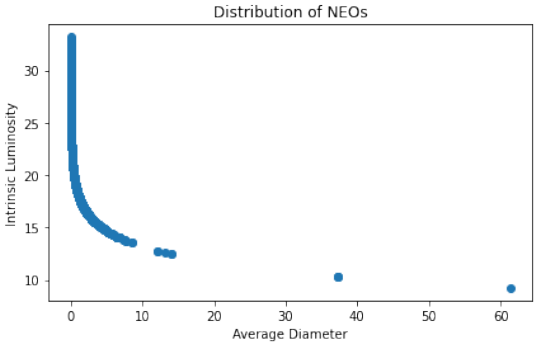
[3]:
```
fig, axes = plt.subplots(1, 2, figsize=(15,4))

axes[0].scatter(df['minDiameter'],df['maxDiameter']);
axes[0].set_xlabel("Minimum Diameter")
axes[0].set_ylabel("Maximum Diameter")
axes[0].set_title("Linear Relationship Between Minimum and Maximum Diameter")

axes[1].scatter(data['Diameter_avg'],data['Luminosity']);
axes[1].set_xlabel("Average Diameter")
axes[1].set_ylabel("Intrinsic Luminosity")
axes[1].set_title("Distribution of NEOs")
```

[3]:
```
Text(0.5, 1.0, 'Distribution of NEOs')
```





Since the dataframe is imbalanced, we create a new dataframe with the ratio of the label approximately 1:1. The next step is to change the Boolean type into integer (True = 1, False = 0) and separate the labels from the feature.

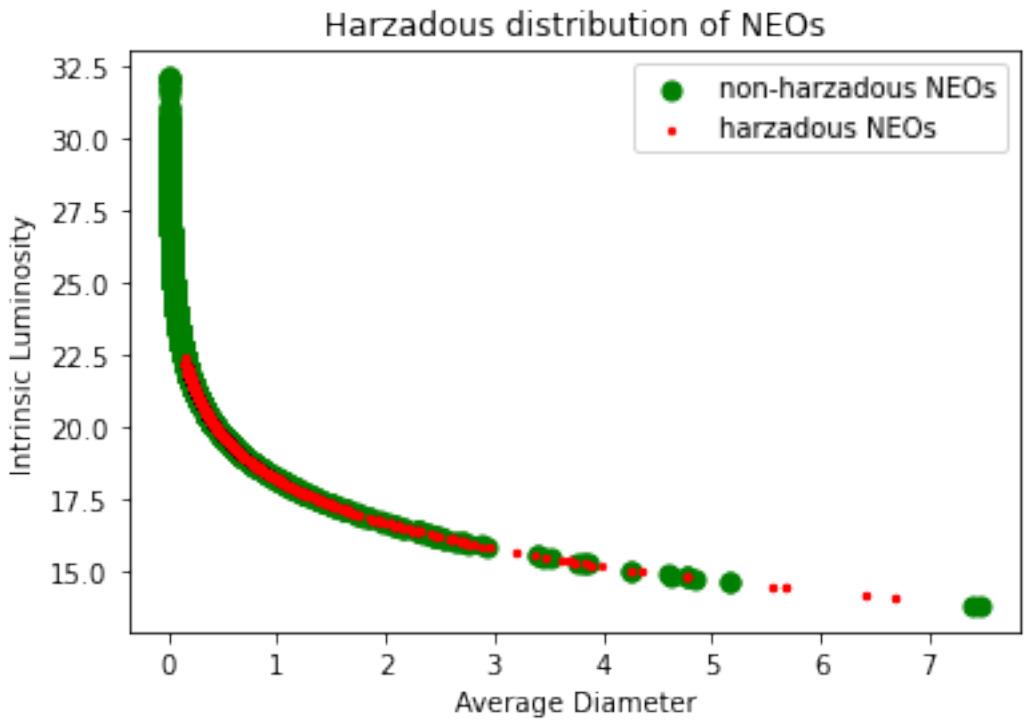
[4]:
```
# New dataframe with a 1:1 ratio of the label
df1 = data[data['Harzadous'] == True]
df0 = data[data['Harzadous'] == False].sample(frac=0.10)
frames = [df1, df0]
data = pd.concat(frames)

# Change the label from Boolean to Int (True = 1, False = 0)
y = data['Harzadous'].astype(int)
y = np.array(y)

# Separate the label and the feature
data.drop(columns=['Harzadous'],inplace=True)
X = data.to_numpy().reshape(17040, -1)
```

From the new dataset we can visualize the distribution of the harzadous NEOs as below.

[5]:
```
# plot results
fig, ax = plt.subplots()
ax.set_xlabel("Average Diameter")
ax.set_ylabel("Intrinsic Luminosity")
ax.set_title("Harzadous distribution of NEOs")
ax.
 ↪scatter(df0['Diameter_avg'],df0['Luminosity'],s=50,c="green",label="non-harzadous␣
 ↪NEOs")
ax.scatter(df1['Diameter_avg'],df1['Luminosity'],s=5,c="r",label="harzadous␣
 ↪NEOs")
ax.legend()
plt.show()
```



2.1 Determination of the Validation Method

In this project, we determine the training and validation set by using k-fold Cross-validation. In regard of the size of the data and running time, $k = 5$ is a reasonable choice with a seed of $seed = 4$.

[6]:
```
#define k-fold
k = 5
shuffle = True
seed = 4
kfold = KFold(n_splits=k, shuffle=shuffle, random_state=seed)
```

3 Implementation of the Machine Learning Methods

3.1 Linear Regression and Polynomial Logistic Regression model

First we consider a classic *Linear Regression model*: the model is built using the training sets, and the accuracy of the validation sets are recorded. We create arrays to record the training and validation errors at each fold, and the mean error and mean accuracy is reported at the end of the model.

[7]:
```
tr_errors1 = [] #k different errors
val_errors1 = []
average_val_accuracy1 = 0
```

```
#go though all folds
for (train_index, val_index) in kfold.split(X):
    #split into training and validation sets
    X_train, y_train, X_val, y_val = X[train_index], y[train_index],␣
 ↪X[val_index], y[val_index]
    #Linear Regression model
    clf = LogisticRegression()
    clf.fit(X_train, y_train)
    y_pred_train = clf.predict(X_train)
    accuracy_train = accuracy_score(y_train, y_pred_train)
    #validation
    y_pred_val = clf.predict(X_val)
    accuracy_val = accuracy_score(y_val, y_pred_val)
    average_val_accuracy1 += accuracy_val
    #calculate the errors
    tr_error = mean_squared_error(y_train, y_pred_train)
    val_error = mean_squared_error(y_val, y_pred_val)
    #save errors
    tr_errors1.append(tr_error)
    val_errors1.append(val_error)
#scores
average_train_error1 = np.mean(tr_errors1)
average_val_error1 = np.mean(val_errors1)
print("Logistic regression model:")
print("Average training error : ", average_train_error1)
print("Average validation error : ", average_val_error1)
print("Average validation accuracy : ", average_val_accuracy1/5)
```

```
Logistic regression model:
Average training error :  0.1307511737089202
Average validation error :  0.13075117370892014
Average validation accuracy :  0.8692488262910798
```

We can fine-tune this method by letting the hypothesis space a polynomials of degree 2 and apply *Polynomial Logistic Regression model* using polynomial features generator and standard scalar. The mean error and mean accuracy is reported at the end of the model, and we can see that the accuracy is improved by 0.06%.

[8]:
```
tr_errors2 = [] #k different errors
val_errors2 = []
average_val_accuracy2 = 0

for (train_index, val_index) in kfold.split(X):
    #split into training and validation sets
    X_train, y_train, X_val, y_val = X[train_index], y[train_index],␣
 ↪X[val_index], y[val_index]
    #Polynomial Logistic Regression model
```

```
    log_regr = LogisticRegression()
    poly = PolynomialFeatures(degree=2)    # generate polynomial features
    X_train_poly = poly.fit_transform(X_train)    # fit the raw features
    scaler = preprocessing.StandardScaler().fit(X_train_poly)
    X_train_poly = scaler.transform(X_train_poly) #scale the transformed␣
↪features
    log_regr.fit(X_train_poly, y_train)    # apply logistics regression to␣
↪these new features and labels
    y_pred_train = log_regr.predict(X_train_poly)
    accuracy_train = accuracy_score(y_train, y_pred_train)

    #validation
    X_val_poly = poly.fit_transform(X_val)
    scaler = preprocessing.StandardScaler().fit(X_val_poly)
    X_val_poly = scaler.transform(X_val_poly)
    y_pred_val = log_regr.predict(X_val_poly)
    accuracy_val = accuracy_score(y_val, y_pred_val)
    average_val_accuracy2 += accuracy_val
    #calculate the errors
    tr_error = mean_squared_error(y_train, y_pred_train)
    val_error = mean_squared_error(y_val, y_pred_val)
    #save errors
    tr_errors1.append(tr_error)
    val_errors1.append(val_error)

    tr_errors2.append(tr_error)
    val_errors2.append(val_error)

average_train_error2 = np.mean(tr_errors2)
average_val_error2 = np.mean(val_errors2)
print("Polynomial logistic regression model:")
print("Average training error : ", average_train_error2)
print("Average validation error : ", average_val_error2)
print("Average validation accuracy : ", average_val_accuracy2/5)
```

```
Polynomial logistic regression model:
Average training error :  0.12607100938967136
Average validation error :  0.13022300469483566
Average validation accuracy :  0.8697769953051644
```

3.2 Multilayer Perceptron (MLP) model

In this method, we apply a *Multilayer Perceptron (MLP) model* using 5 hidden layers and 15 neurons in each of the layer. The mean error is reported at the end of the model, and the mean accuracy is calculated implicitly from the errors.

[9]:
```
tr_errors3 = []
val_errors3 = []

num_layers = 5    # number of hidden layers
num_neurons = 15  # number of neurons in each layer
hidden_layer_sizes = tuple([num_neurons]*num_layers) # size (num of neurons) of␣
 ↪each layer stacked in a tuple

for (train_index, val_index) in kfold.split(X):
    #split into training and validation sets
    X_train, y_train, X_val, y_val = X[train_index], y[train_index],␣
 ↪X[val_index], y[val_index]

    mlp_regr = MLPRegressor(hidden_layer_sizes,max_iter=1000,random_state=42)
    mlp_regr.fit(X_train,y_train)

    ## evaluate the trained MLP on both training set and validation set
    y_pred_train = mlp_regr.predict(X_train)    # predict on the training set
    tr_error = mean_squared_error(y_train, y_pred_train)    # calculate the␣
 ↪training error
    y_pred_val = mlp_regr.predict(X_val) # predict values for the validation␣
 ↪data
    val_error = mean_squared_error(y_val, y_pred_val) # calculate the␣
 ↪validation error

    tr_errors3.append(tr_error)
    val_errors3.append(val_error)

average_train_error3 = np.mean(tr_errors3)
average_val_error3 = np.mean(val_errors3)
print("MLP model:")
print("Average training error : ", average_train_error3)
print("Average validation error : ", average_val_error3)
print("Average validation accuracy : ", 1 - average_val_error3)
```

```
MLP model:
Average training error :  0.10378611819115668
Average validation error :  0.1037426888592881
Average validation accuracy :  0.8962573111407119
```

Despite longer runtime, we can see an 2.65% increase in the accuracy in the validation set compared to *Polynomial Logistic Regression model*.