

Machine learning approaches to the classification problem for Autistic spectrum disorder

Project designed and executed by Sushanth Inampudi

Abstract:

Autistic Spectrum Disorder (ASD) is a neurodevelopment condition associated with significant healthcare costs, and early diagnosis can significantly reduce these. Unfortunately, waiting times for an ASD diagnosis are lengthy and procedures are not cost effective. The economic impact of autism and the increase in the number of ASD cases across the world reveals an urgent need for the development of easily implemented and effective screening methods. Therefore, a time-efficient and accessible ASD screening is imminent to help health professionals and inform individuals whether they should pursue formal clinical diagnosis.

The rapid growth in the number of ASD cases worldwide necessitates datasets related to behaviour traits. However, such datasets are rare making it difficult to perform thorough analyses to improve the efficiency, sensitivity, specificity and predictive accuracy of the ASD screening process. Presently, very limited autism datasets associated with clinical or screening are available and most of them are genetic in nature. Hence, we propose a new dataset related to autism screening of adults that contained 20 features to be utilised for further analysis especially in determining influential autistic traits and improving the classification of ASD cases.

Dataset Link: <https://archive.ics.uci.edu/dataset/426/autism+screening+adult>

Let's begin with preparing our data set.


Step 0: Import Datasets

Start by importing the 'ASD.csv' file into a Pandas dataframe and take a look at it.

```
# Import libraries necessary for this project
import numpy as np
import pandas as pd
from time import time
from IPython.display import display # Allows the use of display() for DataFrames
```

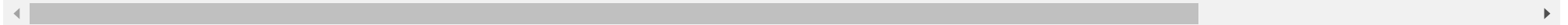
```
# Pretty display for notebooks
%matplotlib inline
```

```
data = pd.read_csv('ASD.csv')
display(data.head(n=5))
```



	id	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score	...	gender	ethnicity	jundice	austim	contry_of_res	used_app_befc
0	1	1	1	1	1	0	0	1	1	0	...	f	White-European	no	no	United States	
1	2	1	1	0	1	0	0	0	1	0	...	m	Latino	no	yes	Brazil	
2	3	1	1	0	1	1	0	1	1	1	...	m	Latino	yes	yes	Spain	
3	4	1	1	0	1	0	0	1	1	0	...	f	White-European	no	yes	United States	
4	5	1	0	0	0	0	0	0	1	0	...	f	?	no	no	Egypt	

5 rows × 22 columns




```
# Total number of records
n_records = len(data.index)

# TODO: Number of records where individual's with ASD
n_asd_yes = len(data[data['Class/ASD'] == 'YES'])

# TODO: Number of records where individual's with no ASD
n_asd_no = len(data[data['Class/ASD'] == 'NO'])

# TODO: Percentage of individuals whose are with ASD
yes_percent = float(n_asd_yes) / n_records *100
# Print the results
print("Total number of records: {}".format(n_records))
print("Individuals diagonised with ASD: {}".format(n_asd_yes))
print("Individuals not diagonised with ASD: {}".format(n_asd_no))
print("Percentage of individuals diagonised with ASD: {:.2f}%".format(yes_percent))
```

 Total number of records: 704
 Individuals diagonised with ASD: 189
 Individuals not diagonised with ASD: 515
 Percentage of individuals diagonised with ASD: 26.85%

Featureset Exploration


This data contains 704 instances, and contains the following attributes:

- **age:** *number* (Age in years).
 - **gender:** *String* [Male/Female].
 - **ethnicity:** *String* (List of common ethnicities in text format).
 - **Born with jaundice:** *Boolean* [yes or no].
 - **Family member with PDD:** *Boolean* [yes or no].
 - **Who is completing the test:** *String* [Parent, self, caregiver, medical staff, clinician ,etc.].
 - **Country of residence:** *String* (List of countries in text format).
 - **Used the screening app before:** *Boolean* [yes or no] (Whether the user has used a screening app)
 - **Screening Method Type:** *Integer* [0,1,2,3] (The type of screening methods chosen based on age category (0=toddler, 1=child, 2= adolescent, 3= adult).
 - **Question 1-10 Answer:** *Binary* [0, 1] (The answer code of the question based on the screening method used).
 - **Screening Score:** *Integer* (The final score obtained based on the scoring algorithm of the screening method used. This was computed in an automated manner).
-

✓ Preparing the Data

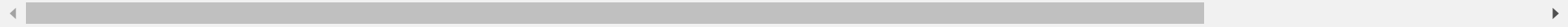
Before data can be used as input for machine learning algorithms, it must be cleaned, formatted, and maybe even restructured — this is typically known as **preprocessing**. Unfortunately, for this dataset, there are many invalid or missing entries(?) we must deal with, moreover, there are some qualities about certain features that must be adjusted. This preprocessing can help tremendously with the outcome and predictive power of nearly all learning algorithms.

```
asd_data = pd.read_csv('ASD.csv', na_values=['?'])
asd_data.head(n=5)
```




	id	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score	...	gender	ethnicity	jundice	austim	contry_of_res	used_app_befc
0	1	1	1	1	1	0	0	1	1	0	...	f	White-European	no	no	United States	
1	2	1	1	0	1	0	0	0	1	0	...	m	Latino	no	yes	Brazil	
2	3	1	1	0	1	1	0	1	1	1	...	m	Latino	yes	yes	Spain	
3	4	1	1	0	1	0	0	1	1	0	...	f	White-European	no	yes	United States	
4	5	1	0	0	0	0	0	0	1	0	...	f	NaN	no	no	Egypt	



5 rows × 22 columns



Here I evaluate whether the data needs cleaning; your model is only as good as the data it's given.


```
asd_data.describe()
```



	id	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score	A10_Score	age	result	
count	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	704.000000	702.000000	704.000000	
mean	352.500000	0.721591	0.453125	0.457386	0.495739	0.498580	0.284091	0.417614	0.649148	0.323864	0.573864	29.698006	4.875000	
std	203.371581	0.448535	0.498152	0.498535	0.500337	0.500353	0.451301	0.493516	0.477576	0.468281	0.494866	16.507465	2.501493	
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	17.000000	0.000000	
25%	176.750000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	21.000000	3.000000	
50%	352.500000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	1.000000	27.000000	4.000000	
75%	528.250000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	35.000000	7.000000	
max	704.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	383.000000	10.000000	

▼ Step 1: Clean Datasets

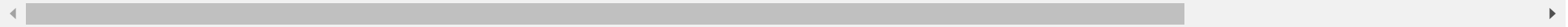
```
asd_data.loc[(asd_data['age'].isnull()) |(asd_data['gender'].isnull()) |(asd_data['ethnicity'].isnull())
|(asd_data['jundice'].isnull())|(asd_data['austim'].isnull()) |(asd_data['contry_of_res'].isnull())
|(asd_data['used_app_before'].isnull())|(asd_data['result'].isnull())|(asd_data['age_desc'].isnull())
|(asd_data['relation'].isnull())]
```



	id	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score	...	gender	ethnicity	jundice	austim	contry_of_res	used_app_t
	4	5	1	0	0	0	0	0	0	1	0	...	f	NaN	no	no	Egypt
	12	13	0	1	1	1	1	1	0	0	1	...	f	NaN	no	no	Bahamas
	13	14	1	0	0	0	0	0	1	1	0	...	m	NaN	no	no	Austria
	14	15	1	0	0	0	0	0	1	1	0	...	f	NaN	no	no	Argentina
	19	20	0	0	0	0	0	0	1	1	0	...	m	NaN	yes	no	United Arab Emirates

	652	653	0	0	0	0	0	0	0	0	0	...	f	NaN	no	no	United States
	658	659	0	0	1	1	0	0	1	0	0	...	m	NaN	no	no	Azerbaijan
	659	660	1	1	1	1	1	1	0	0	1	...	m	NaN	no	no	Pakistan
	666	667	0	0	0	0	0	0	0	1	0	...	m	NaN	no	no	Iraq
	701	702	1	0	1	1	1	0	1	1	0	...	f	NaN	no	no	Russia

95 rows × 22 columns



Since the missing data seems randomly distributed, I go ahead and drop rows with missing data.

```
asd_data.dropna(inplace=True)
asd_data.describe()
```



	id	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score	A10_Score	age	result
count	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000	609.000000
mean	349.725780	0.740558	0.469622	0.481117	0.520525	0.525452	0.307061	0.428571	0.665025	0.341544	0.597701	30.215107	5.077176
std	207.856238	0.438689	0.499487	0.500054	0.499989	0.499762	0.461654	0.495278	0.472370	0.474617	0.490765	17.287470	2.522717
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	17.000000	0.000000
25%	166.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	22.000000	3.000000
50%	329.000000	1.000000	0.000000	0.000000	1.000000	1.000000	0.000000	0.000000	1.000000	0.000000	1.000000	27.000000	5.000000
75%	533.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	35.000000	7.000000
max	704.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	383.000000	10.000000



Let's check out the data types of all our features including the target feature. Moreover, let's count the total number of instances and the target-class distribution.

```
# Reminder of the features:
print(asd_data.dtypes)

# Total number of records in clean dataset
n_records = len(asd_data.index)

# TODO: Number of records where individual's with ASD in the clean dataset
n_asd_yes = len(asd_data[asd_data['Class/ASD'] == 'YES'])

# TODO: Number of records where individual's with no ASD in the clean dataset
n_asd_no = len(asd_data[asd_data['Class/ASD'] == 'NO'])
# Print the results
print("Total number of records: {}".format(n_records))
print("Individuals diagnosed with ASD: {}".format(n_asd_yes))
print("Individuals not diagnosed with ASD: {}".format(n_asd_no))
```



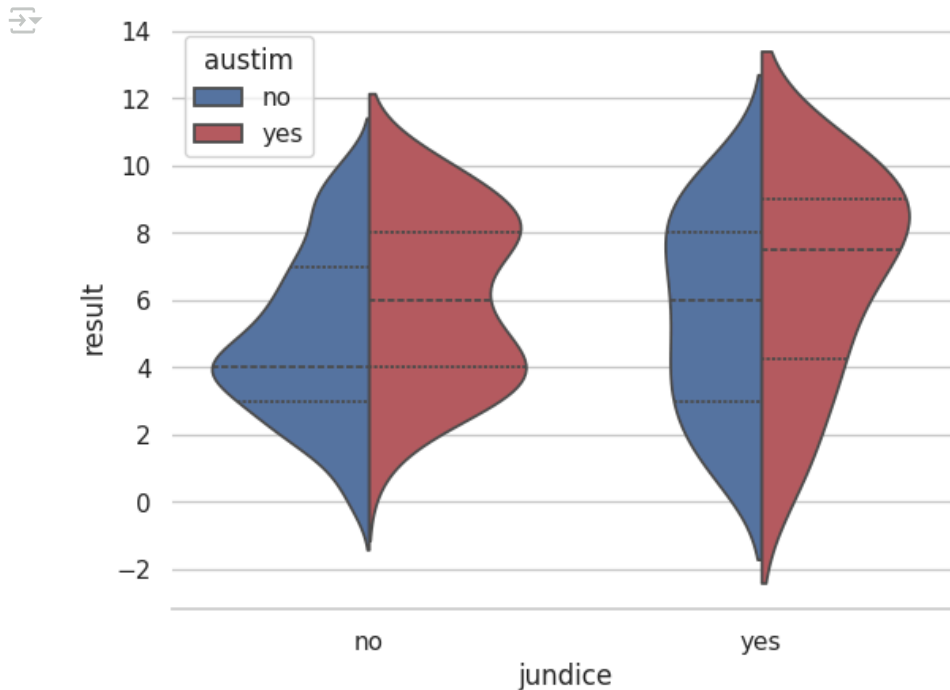
```
id          int64
A1_Score    int64
A2_Score    int64
A3_Score    int64
A4_Score    int64
A5_Score    int64
A6_Score    int64
A7_Score    int64
A8_Score    int64
```

```
A9_Score          int64
A10_Score         int64
age               float64
gender            object
ethnicity         object
jundice           object
austim            object
contry_of_res     object
used_app_before   object
result            int64
age_desc          object
relation          object
Class/ASD         object
dtype: object
Total number of records: 609
Individuals diagonised with ASD: 180
Individuals not diagonised with ASD: 429
```

✓ Step 2: A quick visualization with *Seaborn*

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style="whitegrid", color_codes=True)

# Draw a nested violinplot and split the violins for easier comparison
sns.violinplot(x="jundice", y="result", hue="austim", data=asd_data, split=True,
               inner="quart", palette={'yes': "r", 'no': "b"})
sns.despine(left=True)
```



Next I'll need to convert the Pandas dataframes into numpy arrays that can be used by scikit_learn. Let's create an array that extracts only the feature data we want to work with and another array that contains the classes (class/ASD).

```
# Split the data into features and target label
asd_raw = asd_data['Class/ASD']
features_raw = asd_data[['age', 'gender', 'ethnicity', 'jundice', 'austim', 'contry_of_res', 'result',
                        'relation', 'A1_Score', 'A2_Score', 'A3_Score', 'A4_Score', 'A5_Score', 'A6_Score', 'A7_Score', 'A8_Score',
                        'A9_Score', 'A10_Score']]
```

Some of our models require the input data to be normalized, so I proceed to normalize the attribute data. Here, I use `preprocessing.MinMaxScaler()`.


```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
numerical = ['age', 'result']

features_minmax_transform = pd.DataFrame(data = features_raw)
features_minmax_transform[numerical] = scaler.fit_transform(features_raw[numerical])
```



```
features_minmax_transform
# Show an example of a record with scaling applied
display(features_minmax_transform.head(n = 5))
```



	age	gender	ethnicity	jundice	austim	contry_of_res	result	relation	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_S
0	0.024590	f	White-European	no	no	United States	0.6	Self	1	1	1	1	0	0	1	1	
1	0.019126	m	Latino	no	yes	Brazil	0.5	Self	1	1	0	1	0	0	0	1	
2	0.027322	m	Latino	yes	yes	Spain	0.8	Parent	1	1	0	1	1	0	1	1	
3	0.049180	f	White-European	no	yes	United States	0.6	Self	1	1	0	1	0	0	1	1	
5	0.051913	m	Others	yes	no	United States	0.9	Self	1	1	1	1	1	0	1	1	

One-Hot-Coding

From the table in **Clean Data Sets** above, we can see there are several features for each record that are non-numeric such as Country_of_residence, ethnicity etc. Typically, learning algorithms expect input to be numeric, which requires that non-numeric features (called *categorical variables*) be converted. One popular way to convert categorical variables is by using the **one-hot encoding** scheme. One-hot encoding creates a *"dummy"* variable for each possible category of each non-numeric feature. For example, assume someFeature has three possible entries: A, B, or C. We then encode this feature into someFeature_A, someFeature_B and someFeature_C.

someFeature				
		someFeature_A	someFeature_B	someFeature_C
0	B	0	1	0
1	C	0	0	1
2	A	1	0	0

Additionally, as with the non-numeric features, I need to convert the non-numeric target label, 'Class/ASD' to numerical values for the learning algorithm to work. Since there are only two possible categories for this label ("YES" and "NO" to Class/ASD), I can avoid using one-hot encoding and simply encode these two categories as 0 and 1, respectively. In code cell below, I will implement the following:

- Use `pandas.get_dummies()` to perform one-hot encoding on the 'features_minmax_transform' data.
- Convert the target label 'asd_raw' to numerical entries.
 - Set records with "NO" to 0 and records with "YES" to 1.

```
#One-hot encode the 'features_minmax_transform' data using pandas.get_dummies()
features_final = pd.get_dummies(features_minmax_transform)
display(features_final.head(5))
```

```
# Encode the 'all_classes_raw' data to numerical values
asd_classes = asd_raw.apply(lambda x: 1 if x == 'YES' else 0)
```

```
# Print the number of features after one-hot encoding
encoded = list(features_final.columns)
print("{} total features after one-hot encoding.".format(len(encoded)))
```

```
# Uncomment the following line to see the encoded feature names
print(encoded)
```



	age	result	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	...	contry_of_res_United Arab Emirates	contry_of_res_United Kingdom	contry_of_res_Ur Si
0	0.024590	0.6	1	1	1	1	0	0	1	1	...	False	False	
1	0.019126	0.5	1	1	0	1	0	0	0	1	...	False	False	
2	0.027322	0.8	1	1	0	1	1	0	1	1	...	False	False	
3	0.049180	0.6	1	1	0	1	0	0	1	1	...	False	False	
5	0.051913	0.9	1	1	1	1	1	0	1	1	...	False	False	

5 rows × 94 columns

94 total features after one-hot encoding.

['age', 'result', 'A1_Score', 'A2_Score', 'A3_Score', 'A4_Score', 'A5_Score', 'A6_Score', 'A7_Score', 'A8_Score', 'A9_Score', 'A10_Score', 'gender_f', 'gender_m', 'gender_o', 'gender_u', 'gender_y', 'gender_z', 'gender_0', 'gender_1', 'gender_2', 'gender_3', 'gender_4', 'gender_5', 'gender_6', 'gender_7', 'gender_8', 'gender_9', 'gender_10', 'gender_11', 'gender_12', 'gender_13', 'gender_14', 'gender_15', 'gender_16', 'gender_17', 'gender_18', 'gender_19', 'gender_20', 'gender_21', 'gender_22', 'gender_23', 'gender_24', 'gender_25', 'gender_26', 'gender_27', 'gender_28', 'gender_29', 'gender_30', 'gender_31', 'gender_32', 'gender_33', 'gender_34', 'gender_35', 'gender_36', 'gender_37', 'gender_38', 'gender_39', 'gender_40', 'gender_41', 'gender_42', 'gender_43', 'gender_44', 'gender_45', 'gender_46', 'gender_47', 'gender_48', 'gender_49', 'gender_50', 'gender_51', 'gender_52', 'gender_53', 'gender_54', 'gender_55', 'gender_56', 'gender_57', 'gender_58', 'gender_59', 'gender_60', 'gender_61', 'gender_62', 'gender_63', 'gender_64', 'gender_65', 'gender_66', 'gender_67', 'gender_68', 'gender_69', 'gender_70', 'gender_71', 'gender_72', 'gender_73', 'gender_74', 'gender_75', 'gender_76', 'gender_77', 'gender_78', 'gender_79', 'gender_80', 'gender_81', 'gender_82', 'gender_83', 'gender_84', 'gender_85', 'gender_86', 'gender_87', 'gender_88', 'gender_89', 'gender_90', 'gender_91', 'gender_92', 'gender_93', 'gender_94']

✓ Shuffle and Split Data

Now all *categorical variables* have been converted into numerical features, and all numerical features have been normalized. As always, I will now split the data (both features and their labels) into training and test sets. 80% of the data will be used for training and 20% for testing.

```
from sklearn.model_selection import train_test_split
```

```
np.random.seed(1234)
```

```
X_train, X_test, y_train, y_test = train_test_split(features_final, asd_classes, train_size=0.80, random_state=1)
```

```
# Show the results of the split
print("Training set has {} samples.".format(X_train.shape[0]))
```

```
print("Testing set has {} samples.".format(X_test.shape[0]))
#asd_data
```

↗ Training set has 487 samples.
Testing set has 122 samples.

Step 3: Models

Supervised Learning Models:

- (1) Decision Trees
- (2) Random Forest
- (3) K-Nearest Neighbors (KNeighbors)
- (4) Gaussian Naive Bayes (GaussianNB)
- (5) Logistic Regression

▼ (1) Decision Trees

Start with creating a DecisionTreeClassifier and fit it to the training data.

```
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier

dectree = DecisionTreeClassifier(random_state=1)

# Train the classifier on the training set
dectree.fit(X_train, y_train)
```

↗

▼ DecisionTreeClassifier ⓘ ?

DecisionTreeClassifier(random_state=1)

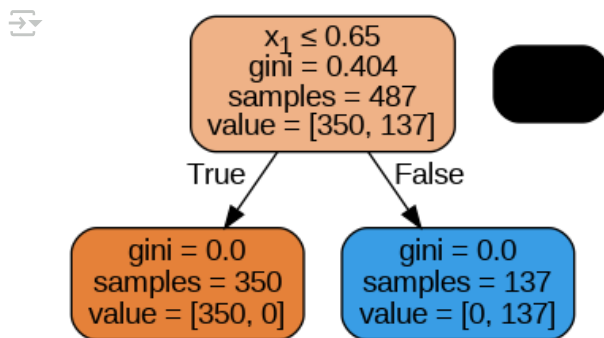
▼ Depiction of Decision Tree algorithm

```
import pydotplus

dot_data = tree.export_graphviz(dectree,
                                out_file=None,
                                filled=True,
                                rounded=True,
                                special_characters=True)

graph = pydotplus.graph_from_dot_data(dot_data)

from IPython.display import Image
Image(graph.create_png())
```



✓ Evaluating Model Performance

Metrics

Note: Recap of accuracy, precision, recall

Accuracy measures how often the classifier makes the correct prediction. It's the ratio of the number of correct predictions to the total number of predictions (the number of test data points).

Precision tells us what proportion of messages we classified as spam, actually were spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all positives(all words classified as spam, irrespective of whether that was the correct classificatio), in other words it is the ratio of

$$\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall(sensitivity) tells us what proportion of messages that actually were spam were classified by us as spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all the words that were actually spam, in other words it is the ratio of

```
[True Positives/(True Positives + False Negatives)]
```

For classification problems that are skewed in their classification distributions like in our case where we have

- a total of 609 records with
- 180 individuals diagnosed with ASD and
- 429 individuals not diagnosed with ASD

Accuracy by itself is not a very good metric. Thus, in this case precision and recall come in very handy. These two metrics can be combined to get the F1 score, which is weighted average(harmonic mean) of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score(we take the harmonic mean as we are dealing with ratios).

```
# make class predictions for the testing set
y_pred_class = dectree.predict(X_test)
```

```
# print the first 25 true and predicted responses
print('True:', y_test.values[0:25])
print('False:', y_pred_class[0:25])
```

```
True: [1 0 0 1 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0]
False: [1 0 0 1 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0]
```

```
from sklearn import metrics
# IMPORTANT: first argument is true values, second argument is predicted values
# this produces a 2x2 numpy array (matrix)
#print(metrics.confusion_matrix(y_test, y_pred_class))
```

```
# save confusion matrix and slice into four pieces
confusion = metrics.confusion_matrix(y_test, y_pred_class)
print(confusion)
#[row, column]
TP = confusion[1, 1]
TN = confusion[0, 0]
FP = confusion[0, 1]
FN = confusion[1, 0]
```

```
[[79  0]
 [ 0 43]]
```

▼ Metrics computed from a confusion matrix

Classification Accuracy: Overall, how often is the classifier correct?

```
# use float to perform true division, not integer division
print((TP + TN) / float(TP + TN + FP + FN))
```

⇒ 1.0

Classification Error: Overall, how often is the classifier incorrect?

```
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print(classification_error)
```

⇒ 0.0

Sensitivity: When the actual value is positive, how often is the prediction correct?

```
sensitivity = TP / float(FN + TP)

print(sensitivity)
print(metrics.recall_score(y_test, y_pred_class))
```

⇒ 1.0
1.0

Specificity: When the actual value is negative, how often is the prediction correct?

```
specificity = TN / (TN + FP)

print(specificity)
```

⇒ 1.0

False Positive Rate: When the actual value is negative, how often is the prediction incorrect?

```
false_positive_rate = FP / float(TN + FP)

print(false_positive_rate)
#print(1 - specificity)
```

⇒ 0.0

Precision: When a positive value is predicted, how often is the prediction correct?

```
precision = TP / float(TP + FP)

#print(precision)
print(metrics.precision_score(y_test, y_pred_class))
```

⇒ 1.0

✓ Visualizing the classification prediction:

```
# print the first 10 predicted responses
# 1D array (vector) of binary values (0, 1)
dectree.predict(X_test)[0:10]
```

⇒ array([1, 0, 0, 1, 1, 0, 1, 1, 0, 0])

```
# print the first 10 predicted probabilities of class membership
dectree.predict_proba(X_test)[0:10]
```

⇒ array([[0., 1.],
[1., 0.],
[1., 0.],
[0., 1.],
[0., 1.],
[1., 0.],
[0., 1.],
[0., 1.],
[1., 0.],
[1., 0.]])

```
# store the predicted probabilities for class 1
y_pred_prob = dectree.predict_proba(X_test)[:, 1]
```

```
# allow plots to appear in the notebook
```

```
import matplotlib.pyplot as plt
```

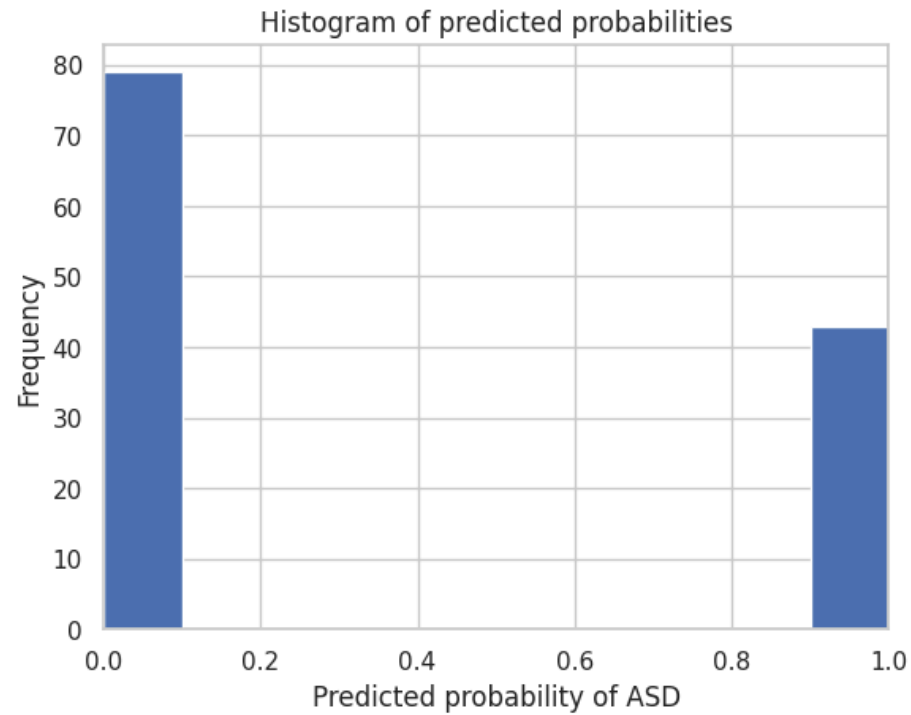
```
# adjust the font size
plt.rcParams['font.size'] = 12
```

```
# histogram of predicted probabilities
```

```
# 8 bins
plt.hist(y_pred_prob, bins=10)
```

```
# x-axis limit from 0 to 1
plt.xlim(0,1)
plt.title('Histogram of predicted probabilities')
plt.xlabel('Predicted probability of ASD')
plt.ylabel('Frequency')
```

 Text(0, 0.5, 'Frequency')



✓ Cross-validation:

Now instead of a single train/test split, I use K-Fold cross validation to get a better measure of your model's accuracy (K=10).

```
from sklearn.model_selection import cross_val_score

dectree = DecisionTreeClassifier(random_state=1)

cv_scores = cross_val_score(dectree, features_final, asd_classes, cv=10)

cv_scores.mean()
```


 1.0

✓ (2) Random Forest

Now I apply a **RandomForestClassifier** instead to see whether it performs better.

```
from sklearn.ensemble import RandomForestClassifier

ranfor = RandomForestClassifier(n_estimators=5, random_state=1)
cv_scores = cross_val_score(ranfor, features_final, asd_classes, cv=10)
cv_scores.mean()
```

 0.9933333333333334

✓ (3) K-Nearest-Neighbors (KNN)

Next, I explore the K-Nearest-Neighbors algorithm with a starting value of K=10. Recall that K is an example of a hyperparameter - a parameter on the model itself which may need to be tuned for best results on your particular data set.

```
from sklearn import neighbors

knn = neighbors.KNeighborsClassifier(n_neighbors=10)
cv_scores = cross_val_score(knn, features_final, asd_classes, cv=10)

cv_scores.mean()
```

 0.9474590163934427

Choosing K is tricky, so I can't discard KNN until we've tried different values of K. Hence we write a for loop to run KNN with K values ranging from 10 to 50 and see if K makes a substantial difference.

```
for n in range(10, 50):
    knn = neighbors.KNeighborsClassifier(n_neighbors=n)
    cv_scores = cross_val_score(knn, features_final, asd_classes, cv=10)
    print (n, cv_scores.mean())
```

 10 0.9474590163934427
11 0.9507377049180328

```
12 0.9507377049180328
13 0.9556830601092896
14 0.9507650273224044
15 0.944207650273224
16 0.9507650273224044
17 0.9507377049180328
18 0.9523770491803278
19 0.955655737704918
20 0.9523770491803278
21 0.9523770491803278
22 0.9474590163934424
23 0.9490983606557375
24 0.9507377049180326
25 0.9507377049180328
26 0.9523770491803278
27 0.9523770491803278
28 0.9507377049180326
29 0.9507377049180328
30 0.9523770491803278
31 0.9474863387978143
32 0.9491256830601094
33 0.9474863387978143
34 0.9507650273224044
35 0.9491256830601094
36 0.9524043715846995
37 0.9524043715846995
38 0.9540710382513661
39 0.9524316939890708
40 0.9540710382513659
41 0.9524316939890708
42 0.9507923497267757
43 0.9507923497267757
44 0.9507923497267757
45 0.9507923497267757
46 0.9524316939890708
47 0.9524316939890708
48 0.9557103825136611
49 0.9524316939890708
```

✓ (4) Naive Bayes

Now I try naive_bayes.MultinomialNB classifier and ask how does its accuracy stack up.

```
from sklearn.naive_bayes import MultinomialNB

#scaler = preprocessing.MinMaxScaler()
#all_features_minmax = scaler.fit_transform(all_features)
```

```
nb = MultinomialNB()
cv_scores = cross_val_score(nb, features_final, asd_classes, cv=10)

cv_scores.mean()

0.885
```

✓ (5) Logistic Regression

We've tried all these fancy techniques, but fundamentally this is just a binary classification problem. Try Logistic Regression, which is a simple way to tackling this sort of thing.

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()
cv_scores = cross_val_score(logreg, features_final, asd_classes, cv=10)
cv_scores.mean()

0.9934426229508198
```

✓ Step 4: Building a MLP model architecture

Building a model here using sequential model architecture best known as **Multi Layer Perceptron (MLP)**.

```
# Imports
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation

np.random.seed(42)

# Building the model architecture with one layer of length 4

model = Sequential()
model.add(Dense(8, activation='relu', input_dim= 94))
model.add(Dropout(0.2))
```

```
model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
```

```
model.summary()
```

```
usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Seq
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
odel: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 8)	760
dropout (Dropout)	(None, 8)	0
dense_1 (Dense)	(None, 1)	9

```
Total params: 769 (3.00 KB)
Trainable params: 769 (3.00 KB)
Non-trainable params: 0 (0.00 B)
```

```
# Compiling the model using categorical_crossentropy loss, and rmsprop optimizer.
```

```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

```
# Running and evaluating the model
```

```
hist = model.fit(X_train, y_train,
                 batch_size=16,
                 epochs=100,
                 validation_data=(X_test, y_test),
                 verbose=2)
```

```
Epoch 1/100
31/31 - 1s - 40ms/step - accuracy: 0.6817 - loss: 0.6819 - val_accuracy: 0.6475 - val_loss: 0.6776
Epoch 2/100
31/31 - 0s - 7ms/step - accuracy: 0.7187 - loss: 0.6601 - val_accuracy: 0.6475 - val_loss: 0.6605
Epoch 3/100
31/31 - 0s - 10ms/step - accuracy: 0.7187 - loss: 0.6305 - val_accuracy: 0.6475 - val_loss: 0.6410
Epoch 4/100
31/31 - 0s - 6ms/step - accuracy: 0.7187 - loss: 0.6055 - val_accuracy: 0.6475 - val_loss: 0.6213
Epoch 5/100
31/31 - 0s - 11ms/step - accuracy: 0.7187 - loss: 0.5722 - val_accuracy: 0.6475 - val_loss: 0.5969
Epoch 6/100
31/31 - 0s - 8ms/step - accuracy: 0.7187 - loss: 0.5417 - val_accuracy: 0.6475 - val_loss: 0.5696
Epoch 7/100
31/31 - 0s - 7ms/step - accuracy: 0.7187 - loss: 0.5077 - val_accuracy: 0.6475 - val_loss: 0.5383
Epoch 8/100
31/31 - 0s - 10ms/step - accuracy: 0.7269 - loss: 0.4826 - val_accuracy: 0.6475 - val_loss: 0.5095
```

```
Epoch 9/100
31/31 - 0s - 10ms/step - accuracy: 0.7515 - loss: 0.4556 - val_accuracy: 0.6967 - val_loss: 0.4807
Epoch 10/100
31/31 - 0s - 7ms/step - accuracy: 0.7762 - loss: 0.4330 - val_accuracy: 0.7295 - val_loss: 0.4501
Epoch 11/100
31/31 - 0s - 6ms/step - accuracy: 0.8131 - loss: 0.3977 - val_accuracy: 0.7787 - val_loss: 0.4224
Epoch 12/100
31/31 - 0s - 4ms/step - accuracy: 0.8501 - loss: 0.3828 - val_accuracy: 0.8033 - val_loss: 0.3969
Epoch 13/100
31/31 - 0s - 6ms/step - accuracy: 0.8563 - loss: 0.3528 - val_accuracy: 0.8607 - val_loss: 0.3686
Epoch 14/100
31/31 - 0s - 4ms/step - accuracy: 0.8932 - loss: 0.3245 - val_accuracy: 0.8689 - val_loss: 0.3453
Epoch 15/100
31/31 - 0s - 5ms/step - accuracy: 0.8624 - loss: 0.3132 - val_accuracy: 0.9262 - val_loss: 0.3193
Epoch 16/100
31/31 - 0s - 11ms/step - accuracy: 0.8912 - loss: 0.2941 - val_accuracy: 0.9262 - val_loss: 0.3008
Epoch 17/100
31/31 - 0s - 5ms/step - accuracy: 0.9014 - loss: 0.2783 - val_accuracy: 0.9098 - val_loss: 0.2870
Epoch 18/100
31/31 - 0s - 5ms/step - accuracy: 0.9035 - loss: 0.2496 - val_accuracy: 0.9098 - val_loss: 0.2646
Epoch 19/100
31/31 - 0s - 5ms/step - accuracy: 0.8850 - loss: 0.2616 - val_accuracy: 0.9098 - val_loss: 0.2518
Epoch 20/100
31/31 - 0s - 9ms/step - accuracy: 0.9076 - loss: 0.2400 - val_accuracy: 0.9098 - val_loss: 0.2388
Epoch 21/100
31/31 - 0s - 5ms/step - accuracy: 0.8912 - loss: 0.2324 - val_accuracy: 0.9180 - val_loss: 0.2271
Epoch 22/100
31/31 - 0s - 5ms/step - accuracy: 0.9179 - loss: 0.2090 - val_accuracy: 0.9180 - val_loss: 0.2160
Epoch 23/100
31/31 - 0s - 5ms/step - accuracy: 0.8768 - loss: 0.2184 - val_accuracy: 0.9262 - val_loss: 0.2078
Epoch 24/100
31/31 - 0s - 5ms/step - accuracy: 0.8891 - loss: 0.2162 - val_accuracy: 0.9180 - val_loss: 0.2027
Epoch 25/100
31/31 - 0s - 5ms/step - accuracy: 0.8912 - loss: 0.2180 - val_accuracy: 0.9344 - val_loss: 0.1948
Epoch 26/100
31/31 - 0s - 5ms/step - accuracy: 0.9014 - loss: 0.2118 - val_accuracy: 0.9344 - val_loss: 0.1889
Epoch 27/100
31/31 - 0s - 5ms/step - accuracy: 0.8850 - loss: 0.1960 - val_accuracy: 0.9426 - val_loss: 0.1819
Epoch 28/100
31/31 - 0s - 5ms/step - accuracy: 0.8768 - loss: 0.2056 - val_accuracy: 0.9344 - val_loss: 0.1795
Epoch 29/100
31/31 - 0s - 5ms/step - accuracy: 0.8891 - loss: 0.1902 - val_accuracy: 0.9426 - val_loss: 0.1722
```

▼ Evaluating the model

This will give you the accuracy of the model.