

Project 2: Classification of Drugs by Training on Molecular Data

CHEE 426(G) Spring 2024

Textbook Chapter 3 & 4.

This notebook assumes you start from the Clean Data. This is when your install of "dmol-book" does not work.

The basis of this project is a published work by Data Scientist **Andrew White**: <https://dmol.pub/ml/classification.html#complete-model> (<https://dmol.pub/ml/classification.html#complete-model>).

Check the webpage of this original work (above) for more details.

Problem Statement

This Project 2 covers the development of a Classifier to predict the Approval (or Non-approval) by FDA (Food and Drug Administration) of Drugs by training the model on molecular data. It involves concepts and techniques from the Chapter 3 and Chapter 4 of the textbook.

This is also a project where you will compete with each other via **Kaggle** to develop the best Classifier for this project. Hence, this project has its Kaggle competition page where you will submit your notebook and your predictions to a Test Set that only I know the correct labels (true Y). Therefore, your Kaggle performance from this project will be counted as a seatwork - this is separate from your project grade even though we use the same problem for **Project** and seatwork **Competition** via Kaggle.

What Is Expected of You Given We Have this Template Notebook?

Having this template notebook for the project, your task then is to improve the model. Here are possible ways to improve the model according to the discussion of our textbook author:

- (1) Use only a subset of the X variables.
- (2) Transform some of the X variables.
- (3) Use models other than Logistic Regression.

Referring to (3), you may also use techniques not covered by the textbook (note that the textbook is focused on few project examples - few examples but deep analysis and good demo) and there are many algorithms out there other than Logistic Regression to develop a Classifier.

Where Do You Go from Here?

Use this template notebook as a **Base Case** then include your improved model development **by extending the notebook** (add starting at the end of this template notebook). **Requirement: Use Evaluation Metrics for Classification** (Chapter 4) to numerically show the improved performance of your own model compared to the Base Case model.

Data


The dataset will be downloaded from a GitHub repository via a Pandas function. The dataset was prepared by the MoleculeNet group (Zhenqin Wu, et al. Moleculenet: a benchmark for molecular machine learning. Chemical science, 9(2):513–530, 2018.).

It is a collection of molecules that succeeded or failed in clinical trials.

The development of a new drug can cost well over a \$1 billion, so any way to predict if a molecule will fail during clinical trials is highly valuable.

The labels (Y) will be the **FDA_Approved** column which is a 1 or 0 indicating FDA approval status. This is an example of **binary classification**.

Import (or Install if needed) Python Modules

```
In [1]:  # Uncomment the installation code below if you have not installed the module 'dmol-book' that has the codename 'dmol-book'  
# Keep the comment if you already installed the module in your environment. This is so the install will not happen again  
  
!pip install dmol-book==1.0.0  
  
# Note: The "!pip" above is a code to call an install via pip from within a jupyter notebook
```

```
In [2]: # If any of the modules below results to import error, make sure to install the module
# into the environment you are using to run the notebook.
#
# If the module is not available in Anaconda Navigator, highly likely it is in pip repository. Hence, install
# the module via terminal using 'pip install <package name>'. Remember to launch the terminal from
# the Anaconda Navigator Environment, and not from the pure windows terminal. The conda environment must be your e
# A sign that the you are in the conda environment is a show of the environment name as the left-most (first part) o
# in the terminal.
#
# Some of the installs may take a while as these packages (rdkit, jax, mordred) are large.
# You can also try the '!pip' from within the notebook like the dml-book above.

#import rdkit, rdkit.Chem, rdkit.Chem.Draw
#import jax.numpy as jnp
#import mordred, mordred.descriptors
#import jax
#import dml
```

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
```

Import Data

```
In [3]: # from zipfile import ZipFile
# from io import BytesIO
# from urllib.request import urlopen

## from web version
# url = 'https://deepchemdata.s3-us-west-1.amazonaws.com/datasets/clintox.csv.gz'
# file = urlopen(url).read()
# file = BytesIO(file)
# document = ZipFile(file)
# toxdata = pd.read_csv(document.open('clintox.csv'))

## Local version
#toxdata = pd.read_csv(
#    "https://github.com/whitead/dmol-book/raw/main/data/clintox.csv.gz"
#)
#
#toxdata.head() # this shows the top 5 rows of the data.
#
# Also check the GitHub repo of the data as shown in the URL inside "" above for more details.
```

Raw Data and SMILES

The molecular structure under the heading 'smiles' follows the standardized SMILES format of reporting molecular structure. Read more about this SMILES if you want more details: https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system (https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system).

The SMILES data, however, will be needed to compute the descriptors (X variables) as shown in the following Molecular Descriptors section.

Molecular Descriptors

This time, our data does not come with pre-computed descriptors. We only have the SMILES string, which is a way of writing a molecule using letters and numbers (a string). We can use rdkit to convert the SMILES string into a molecule, and then we can use a package called Mordred (cite)moriwaki2018mordred to compute a set of descriptors for each molecule. This package will compute around 1500 descriptors (1500 X-variables) for each molecule.

We'll start by converting our molecules into rdkit objects and building a calculator to compute the descriptors.

```
In [4]: % ## make object that can compute descriptors
        #calc = mordred.Calculator(mordred.descriptors, ignore_3D=True)
        ## make subsample from pandas df
        #molecules = [rdkit.Chem.MolFromSmiles(smi) for smi in toxdata.smiles]
        #
        ## view one molecule to make sure things look good.
        #molecules[0]
```

Some of our molecules failed to be converted. We'll have to remove them. We need to remember which ones were deleted too, since we need to remove the failed molecules from the labels.

```
In [5]: % # the invalid molecules were None, so we'll just
        # use the fact the None is False in Python
        #valid_mol_idx = [bool(m) for m in molecules]
        #valid_mols = [m for m in molecules if m]
```

```
In [6]: % ## Just checking how many valid samples are there in the dataset:
        #len(valid_mols)
```

```
In [7]: % ## Now generating the descriptors (X vars) from the valid molecules
        ## This will run for some time (seconds to minutes) as the mordred calc extracts the properties of each molecule.
        #
        #
        #features_raw = calc.pandas(valid_mols)
```

```
In [8]: % ## Check how the table looks:
        #features_raw
```

The modeling we are doing must use only the numeric descriptors. Examples of non-numeric are the first 2 columns as shown by the error in numpy. Note: Numpy works with numeric data only. Remember that a categorical data can be converted to numeric via One-Hot Encoding, but we will not do that now - we just work with whatever numeric data we have in the dataset.

```
In [9]: % ## Check data types
        #
        #features_raw.dtypes
```

```
In [10]: % ## Note: there are more than 1000 descriptors/columns, so sifting through each column to 'see' data type is not pra
        #
        ## First, observe that some numeric columns have been automatically labeled by pandas as object. This is not correc
        ## Hence, we force Pandas to convert all possible columns into numeric.
        #
        #features = features_raw.astype(float) # Those data that cannot be converted to float will be NaN
        #
        ## Note: Pay attention to the change of name of the Numpy array. This is a technique in working with notebook - cre
        ##
        ## run this code if you want to see the summary of datatypes:
        ## features.dtypes
        ## features.head()
        #
        ## We then remove columns with NaN entries as shown in the following cells resulting to lower than 1000 descriptors
```

Now we just need to stitch everything back together so that our labels are consistent and standardize our features.

```
In [11]: % #Labels = toxdata[valid_mol_idx].FDA_APPROVED
        #features -= features.mean()
        #features /= features.std()
        #
        ## we have some nans in features, likely because std was 0
        #features.dropna(inplace=True, axis=1)
        #
        #print(f"We have {len(features.columns)} features per molecule")
```

So, the dataset is now ready: Y = labels, and X = features.

That was quite a lengthy pre-processing of data. Sometimes, PRE-PROCESSING is a bottleneck in an ML task. Some Data Scientist sole job is preparing data, which requires a deeper understanding of data structures and transformation techniques. This is a big job that there is a job specialization called Data Engineering in world of Data Science.

Import the Clean Data

```
In [12]: import pandas
data = pandas.read_csv('projec2_data_clean.csv')
```

```
In [13]: data.head()
```

Out[13]:

	Unnamed: 0	nAcid	nBase	nAromAtom	nAromBond	nAtom	nHeavyAtom	nSpiro	nBridgehead	nHetero	...	SRW10	TSRW
0	0	-0.478046	-0.652093	-1.203569	-1.193150	-0.055375	-0.142306	-0.165193	-0.190777	0.445574	...	-0.716841	-0.5085
1	1	-0.478046	-0.652093	-1.203569	-1.193150	-1.034032	-0.902185	-0.165193	-0.190777	-0.232175	...	-0.364046	-1.0829
2	2	1.849282	-0.652093	-1.203569	-1.193150	-0.907754	-0.775538	-0.165193	-0.190777	0.106700	...	-0.669475	-1.0157
3	3	-0.478046	2.323543	-0.300663	-0.318022	-0.371071	-0.268953	-0.165193	-0.190777	-0.401612	...	-0.129900	-0.5301
4	4	-0.478046	3.315422	0.602242	0.557106	0.007764	0.047664	-0.165193	-0.190777	-0.232175	...	-0.139229	-0.2851

5 rows × 483 columns



```
In [14]: # remove the 'Unnamed' column. That is just a data index in the excel workbook.
df = data.loc[:, ~data.columns.str.contains('^Unnamed')]
```

```
In [15]: df.head()
```

Out[15]:

	nAcid	nBase	nAromAtom	nAromBond	nAtom	nHeavyAtom	nSpiro	nBridgehead	nHetero	nH	...	SRW10	TSRW
0	-0.478046	-0.652093	-1.203569	-1.193150	-0.055375	-0.142306	-0.165193	-0.190777	0.445574	0.029177	...	-0.716841	-0.5085
1	-0.478046	-0.652093	-1.203569	-1.193150	-1.034032	-0.902185	-0.165193	-0.190777	-0.232175	-1.099389	...	-0.364046	-1.0829
2	1.849282	-0.652093	-1.203569	-1.193150	-0.907754	-0.775538	-0.165193	-0.190777	0.106700	-0.980593	...	-0.669475	-1.0157
3	-0.478046	2.323543	-0.300663	-0.318022	-0.371071	-0.268953	-0.165193	-0.190777	-0.401612	-0.386610	...	-0.129900	-0.5301
4	-0.478046	3.315422	0.602242	0.557106	0.007764	0.047664	-0.165193	-0.190777	-0.232175	0.088576	...	-0.139229	-0.2851

5 rows × 482 columns



```
In [16]: features = df.drop(['FDA_APPROVED'], axis=1)
```

```
In [17]: # X-vars
features.head()
```

Out[17]:

	nAcid	nBase	nAromAtom	nAromBond	nAtom	nHeavyAtom	nSpiro	nBridgehead	nHetero	nH	...	SRW09	SRW10
0	-0.478046	-0.652093	-1.203569	-1.193150	-0.055375	-0.142306	-0.165193	-0.190777	0.445574	0.029177	...	-0.96614	-0.716841
1	-0.478046	-0.652093	-1.203569	-1.193150	-1.034032	-0.902185	-0.165193	-0.190777	-0.232175	-1.099389	...	-0.96614	-0.364046
2	1.849282	-0.652093	-1.203569	-1.193150	-0.907754	-0.775538	-0.165193	-0.190777	0.106700	-0.980593	...	-0.96614	-0.669475
3	-0.478046	2.323543	-0.300663	-0.318022	-0.371071	-0.268953	-0.165193	-0.190777	-0.401612	-0.386610	...	-0.96614	-0.129900
4	-0.478046	3.315422	0.602242	0.557106	0.007764	0.047664	-0.165193	-0.190777	-0.232175	0.088576	...	-0.96614	-0.139229

5 rows × 481 columns



```
In [18]: # Y-var
labels = df['FDA_APPROVED']
```

```
In [19]: labels.head()
```

Out[19]:

```
0    1.0
1    1.0
2    1.0
3    1.0
4    1.0
Name: FDA_APPROVED, dtype: float64
```

```
In [20]: # X: Check for null: NA, NaN, etc.
print(f'Total working samples X = {len(features)} \n')

print(f'NULL values in X: \n{features.isnull().sum()} \n')
```

Total working samples X = 1476

NULL values in X:

nAcid	0
nBase	0
nAromAtom	0
nAromBond	0
nAtom	0
..	
WPath	0
WPol	0
Zagreb1	0
Zagreb2	0
mZagreb2	0

Length: 481, dtype: int64

Exploratory data analysis

```
In [21]: # Y: Check for null: NA, NaN, etc.
print(f'Total working samples Y = {len(labels)} \n')
print(f'NULL values in Y: {labels.isnull().sum()} \n')

# Let us check our Y var: FDA-approved drug - "Yes"=1, "No"=0.
print(f'Y levels counts: \n{labels.value_counts()}')
```

Total working samples Y = 1476

NULL values in Y: 0

Y levels counts:

FDA_APPROVED	
1.0	1382
0.0	94

Name: count, dtype: int64

Notice that there are way more FDA-approved drugs in our dataset than the drugs that failed FDA approval. This is a limitation inherent to the field of Health Science (not Data Science). Only companies very confident with the success of their drugs will submit their drugs for FDA evaluation. Nonetheless, we, the Data Scientists, must work with the data that we have at hand. So, let us continue.

```
In [22]: # To facilitate the correct splitting of the dataset into Testing, Trainig, and Validation sets,
# Let us combine the features(X) dataframe and the Labels(Y) dataframe and call it df.

# concatenating X and Y along columns
df = pd.concat([features, labels], axis=1) # axis=1 means column-wise; axis=0 means rowwise.

df = df.dropna()
# Remember to use 'df' dataframe in the succeeding steps.
```

```
In [23]: # Checking dataframe first 5 rows
df.head()
```

Out[23]:

	nAcid	nBase	nAromAtom	nAromBond	nAtom	nHeavyAtom	nSpiro	nBridgehead	nHetero	nH	...	SRW10	TSRW10
0	-0.478046	-0.652093	-1.203569	-1.193150	-0.055375	-0.142306	-0.165193	-0.190777	0.445574	0.029177	...	-0.716841	-0.508561
1	-0.478046	-0.652093	-1.203569	-1.193150	-1.034032	-0.902185	-0.165193	-0.190777	-0.232175	-1.099389	...	-0.364046	-1.082961
2	1.849282	-0.652093	-1.203569	-1.193150	-0.907754	-0.775538	-0.165193	-0.190777	0.106700	-0.980593	...	-0.669475	-1.015711
3	-0.478046	2.323543	-0.300663	-0.318022	-0.371071	-0.268953	-0.165193	-0.190777	-0.401612	-0.386610	...	-0.129900	-0.530111
4	-0.478046	3.315422	0.602242	0.557106	0.007764	0.047664	-0.165193	-0.190777	-0.232175	0.088576	...	-0.139229	-0.285111

5 rows × 482 columns



Splitting Dataset

```
In [24]: > from sklearn.model_selection import train_test_split
```

```
In [25]: > df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=1)
```

```
In [26]: > df_train, df_val = train_test_split(df_train_full, test_size=0.33, random_state=11)
```

```
In [27]: > y_train = df_train.FDA_APPROVED  
y_val = df_val.FDA_APPROVED  
y_test = df_test.FDA_APPROVED
```

```
In [28]: > del df_train['FDA_APPROVED']  
del df_val['FDA_APPROVED']  
del df_test['FDA_APPROVED']  
  
# Since there is no categorical var in our X feature data, no need to do One-Hot Encoding. Hence:  
  
X_train = df_train  
X_val = df_val  
X_test = df_test
```

```
In [29]: > y_train.dtype
```

```
Out[29]: dtype('float64')
```

```
In [30]: > # Checking 0 and 1 in the y_test. Make sure they are 0's.  
y_test.value_counts()
```

```
Out[30]: FDA_APPROVED  
1.0      274  
0.0       22  
Name: count, dtype: int64
```

```
In [ ]: >
```

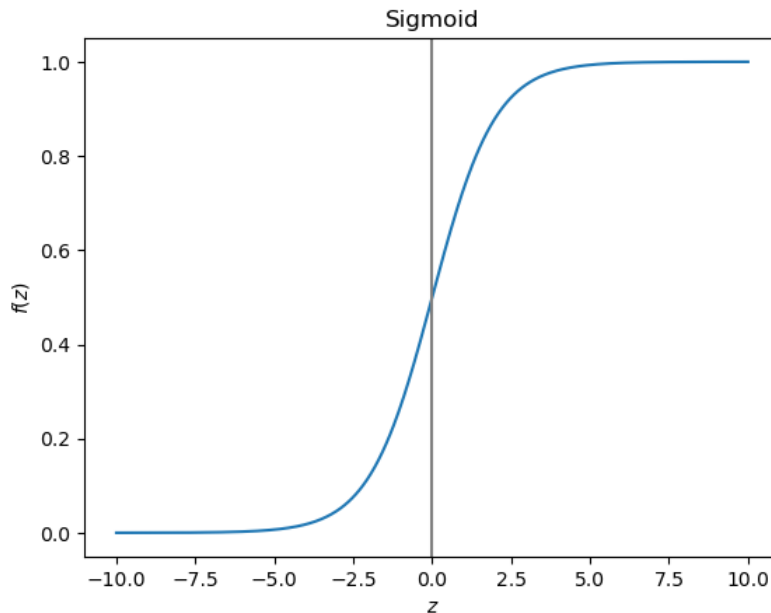
Model Training

Since we ought to develop a classifier that will predict the success of getting approved by FDA (1 = FDA-approved, and 0 = Not FDA-approved), we have to implement a classification ML. We use our Logistic Regression model for two output levels (binary output).

```
In [31]: # Recall the logistic regression base-function  $f(z)=\text{SIGMOID}(z)$ :
```

```
from myst_nb import glue

z = np.linspace(-10, 10, 500)
y = 1 / (1 + np.exp(-z))
plt.plot(z, y)
plt.xlabel(r"$z$")
plt.ylabel(r"$f(z)$")
plt.axvline(0, color="gray")
plt.title("Sigmoid")
glue("sigmoid", plt.gcf(), display=False)
```



where $f(z)$ is the **sigmoid** function. The variable z is the linear model wherein we estimate the model parameters w such that: $z = wX = w_0 + \sum_{i=1}^n w_i X_i$. Recall the dot-product: wX . The molecular descriptors are the model inputs X_i , which are the data in the dataframe **features** we created above. The sigmoid has a domain of $(-\infty, \infty)$ and outputs a probability $(0, 1)$. The input to the sigmoid can be viewed as log-odds, called **logits** for short. Odds are ratios of probability -- odds of 1 means the probability of the class 1 is 0.5 and class 0 is 0.5. Odds of 2 means the probability of class 1 is 0.67 and class 0 is 0.33. Log-odds is the natural logarithm of that, so that log-odds of 0 means the odds are 1 and the output probability should be 0.5. One definition of the sigmoid is

$$f(z) = \frac{1}{1 + e^{-z}}$$
$$z = wX = w_0 + \sum_{i=1}^n w_i X_i$$

however in practice there are some complexities to implementing sigmoids to make sure they're numerically stable. This type of binary classifier is sometimes called **logistic regression** because we're regressing logits.

In essence, all we've done is replacing the inequality of the perceptron with a smooth differentiable version. Just like previously, a positive number indicated class 1 (FDA approved) but now it's a continuum of numbers from 0.5 to 1.0. This is **soft** classification -- we give probabilities of class membership instead of hard assignment. However, our loss function now needs to be modified as well.

Instead of writing our own code to perform logistic regression, we will use the pre-built function in Scikit-Learn as shown by the textbook author (Chapter 3 and 4): **from sklearn.linear_model import LogisticRegression**.

So, please review/study/consult the detailed step-by-step implementation and explanation of the code in the Chapter 3 and 4 notebook and videos.

Training logistic regression

```
In [32]: from sklearn.linear_model import LogisticRegression
```

```
In [33]: model = LogisticRegression(solver='liblinear', random_state=1)
model.fit(X_train, y_train)
```

```
Out[33]: LogisticRegression
LogisticRegression(random_state=1, solver='liblinear')
```

https://scikit-learn.org/1.4/modules/generated/sklearn.linear_model.LogisticRegression.html

```
In [34]: model.predict_proba(X_val)
```

```
Out[34]: array([[2.37791341e-02, 9.76220866e-01],
 [1.67183433e-02, 9.83281657e-01],
 [2.94478882e-02, 9.70552112e-01],
 [8.59114273e-03, 9.91408857e-01],
 [4.78167242e-02, 9.52183276e-01],
 [2.98236347e-01, 7.01763653e-01],
 [2.51266817e-03, 9.97487332e-01],
 [6.45621440e-04, 9.99354379e-01],
 [1.78596430e-03, 9.98214036e-01],
 [2.33961940e-01, 7.66038060e-01],
 [6.39750452e-01, 3.60249548e-01],
 [7.83470309e-02, 9.21652969e-01],
 [2.77041815e-02, 9.72295819e-01],
 [7.00915226e-04, 9.99299085e-01],
 [2.74190537e-03, 9.97258095e-01],
 [9.34025428e-01, 6.59745716e-02],
 [2.34834893e-02, 9.76516511e-01],
 [8.04957089e-03, 9.91950429e-01],
 [8.39292684e-03, 9.91607073e-01],
 [2.10000000e-01, 0.00000000e+00]])
```

```
In [35]: y_pred = model.predict_proba(X_val)[:, 1]
```



```
In [36]: ▶ # These are all the predicted probabilities in the entire validation subset.  
y_pred
```

```
Out[36]: array([9.76220866e-01, 9.83281657e-01, 9.70552112e-01, 9.91408857e-01,
9.52183276e-01, 7.01763653e-01, 9.97487332e-01, 9.99354379e-01,
9.98214036e-01, 7.66038060e-01, 3.60249548e-01, 9.21652969e-01,
9.72295819e-01, 9.99299085e-01, 9.97258095e-01, 6.59745716e-02,
9.76516511e-01, 9.91950429e-01, 9.91607073e-01, 8.81737185e-01,
9.96041144e-01, 4.55793535e-01, 9.17436215e-01, 8.84478879e-01,
9.95846061e-01, 9.87897844e-01, 9.99651517e-01, 9.95714114e-01,
7.42850188e-01, 9.77081128e-01, 9.53188628e-01, 9.97813390e-01,
9.21031345e-01, 9.78874614e-01, 9.99020750e-01, 9.90051158e-01,
9.66104425e-01, 9.10337494e-01, 9.95679086e-01, 7.61824217e-01,
9.82247240e-01, 9.49316611e-01, 9.25395697e-01, 8.67022369e-01,
9.80851200e-01, 9.75482658e-01, 8.37616864e-01, 4.35827308e-01,
9.99998889e-01, 9.85121314e-01, 9.52275789e-01, 8.19901345e-01,
7.97499142e-01, 7.35956784e-01, 9.67637168e-01, 9.90779571e-01,
9.99985601e-01, 9.56492660e-01, 9.99972412e-01, 9.63742437e-01,
9.94128379e-01, 9.99951960e-01, 9.99332924e-01, 6.09718026e-01,
8.80938136e-01, 9.47941514e-01, 9.93934368e-01, 9.91890255e-01,
2.60681267e-01, 9.99972642e-01, 9.48555756e-01, 9.62793604e-01,
9.95426516e-01, 9.93936201e-01, 9.9933268e-01, 8.46331384e-01,
9.15037286e-01, 8.98552951e-01, 9.40647197e-01, 9.35821348e-01,
9.95895816e-01, 9.97243173e-01, 9.48194906e-01, 9.85241648e-01,
9.97996080e-01, 9.49000044e-01, 9.48494399e-01, 9.99538542e-01,
6.79740247e-01, 9.28761846e-01, 9.78090662e-01, 9.87940619e-01,
9.91607505e-01, 9.07373704e-01, 9.91452802e-01, 9.52957158e-01,
9.99684260e-01, 6.26754081e-01, 9.99736963e-01, 9.91249568e-01,
9.86908745e-01, 9.62525540e-01, 9.97028056e-01, 9.97324603e-01,
9.71258165e-01, 9.84282984e-01, 7.64478950e-03, 9.06186720e-01,
8.81886684e-01, 8.85018870e-01, 9.96040047e-01, 9.99316309e-01,
9.88224163e-01, 9.93688581e-01, 9.94323427e-01, 9.96608680e-01,
9.27003585e-01, 9.80716777e-01, 9.86238908e-01, 9.99264012e-01,
9.15228820e-01, 8.40686622e-01, 9.98288804e-01, 9.99901194e-01,
9.43855513e-01, 7.76226688e-01, 9.98792143e-01, 9.93463154e-01,
9.99732310e-01, 9.92429766e-01, 9.77521915e-01, 9.73222803e-01,
9.87691403e-01, 6.73510328e-01, 9.81551895e-01, 9.94698832e-01,
9.31960502e-01, 9.78812270e-01, 9.32207138e-01, 7.77206665e-01,
9.33656393e-01, 9.98969533e-01, 9.65629519e-01, 6.47803300e-01,
9.23561629e-01, 9.78983545e-01, 9.83896407e-01, 9.36947048e-01,
9.27006483e-01, 9.60482675e-01, 9.95410551e-01, 9.97410558e-01,
9.27075087e-01, 9.88940316e-01, 9.62568160e-01, 9.81840211e-01,
9.89352558e-01, 8.94518018e-01, 9.94217871e-01, 9.98761402e-01,
9.95143567e-01, 7.83146326e-01, 9.50401195e-01, 1.39053660e-07,
9.13169793e-01, 8.55526736e-01, 9.63351582e-01, 9.99093024e-01,
9.46281806e-01, 9.90748788e-01, 9.86096684e-01, 9.86815561e-01,
9.54925850e-01, 5.55315579e-04, 9.11317666e-01, 9.99863571e-01,
9.98973644e-01, 9.05527028e-01, 9.97389926e-01, 9.97064216e-01,
9.76866489e-01, 9.37631736e-01, 9.61234217e-01, 8.82046542e-01,
8.88256075e-01, 9.89865595e-01, 8.67645634e-01, 9.97750013e-01,
9.47468969e-01, 9.88812531e-01, 9.85947069e-01, 9.95221660e-01,
9.72006775e-01, 9.99838029e-01, 9.65698112e-01, 3.74804386e-02,
8.63567694e-01, 9.98124109e-01, 9.89427554e-01, 9.92407959e-01,
6.77614531e-01, 9.99986564e-01, 9.99996561e-01, 9.99999918e-01,
9.72896144e-01, 8.88317430e-01, 8.65191263e-01, 8.43224698e-01,
9.98682162e-01, 9.93028047e-01, 9.39058728e-01, 9.94895469e-01,
9.85418858e-01, 9.98408524e-01, 9.98854210e-01, 7.87630794e-01,
9.26764883e-01, 9.49577307e-01, 7.79446088e-01, 9.89253118e-01,
9.02150410e-01, 9.21422759e-01, 9.37217229e-01, 9.99264238e-01,
9.96403446e-01, 9.85486247e-01, 9.85763363e-01, 9.99815870e-01,
9.71779267e-01, 9.98129301e-01, 7.57271479e-03, 9.87256433e-01,
9.78877918e-01, 9.98749899e-01, 9.51480439e-01, 9.66327953e-01,
9.46900855e-01, 9.94428242e-01, 9.12601561e-01, 9.60267146e-01,
9.43975788e-01, 9.63672713e-01, 3.59820829e-01, 9.84778073e-01,
9.99996898e-01, 9.43206822e-01, 9.90653851e-01, 7.95782943e-01,
9.82224359e-01, 9.40925916e-01, 9.47915964e-01, 9.96949061e-01,
6.03036189e-01, 9.99987578e-01, 9.75674080e-01, 9.79994092e-01,
9.71886191e-01, 9.53354113e-01, 9.84962213e-01, 9.66193373e-01,
9.88024062e-01, 9.90190470e-01, 8.85537907e-01, 9.94384346e-01,
9.90729482e-01, 9.75303669e-01, 9.96296211e-01, 9.83516651e-01,
9.77506411e-01, 6.91400921e-01, 9.86332212e-01, 9.66684711e-01,
8.70585903e-01, 9.75864168e-01, 8.44090317e-01, 9.55534761e-01,
9.99999648e-01, 9.97461987e-01, 9.94917352e-01, 9.15563357e-01,
9.60428983e-01, 9.90175290e-01, 9.99214115e-01, 9.46135400e-01,
9.99992939e-01, 9.98920274e-01, 8.70550956e-01, 8.10141868e-01,
9.57833407e-01, 3.72954768e-01, 9.86290759e-01, 9.81269542e-01,
9.99565609e-01, 9.99502036e-01, 9.90383215e-01, 9.77096633e-01,
9.56288344e-01, 7.67661048e-01, 7.06399178e-01, 8.43652996e-01,
9.97691899e-01, 8.91908951e-01, 9.90500310e-01, 9.34218565e-01,
9.87071868e-01, 6.08503649e-01, 8.20013396e-01, 9.97922611e-01,
9.24175800e-01, 9.62744394e-01, 9.95327035e-01, 9.19814269e-01,
9.93296250e-01, 9.99528527e-01, 9.98849443e-01, 2.65899640e-01,
8.71923346e-01, 9.39061684e-01, 7.10779229e-01, 9.58827814e-01,
9.93465427e-01, 9.97945546e-01, 9.80668964e-01, 9.97766498e-01,
6.31636498e-01, 9.83988798e-01, 9.29055209e-01, 9.97214831e-01,
```

```

9.23706876e-01, 9.95704180e-01, 9.92414489e-01, 9.69629408e-01,
9.10713938e-01, 9.67213458e-01, 9.84267799e-01, 9.99999957e-01,
9.98478173e-01, 9.95905719e-01, 8.97968552e-01, 9.93214687e-01,
9.76575194e-01, 9.93139343e-01, 9.09695237e-01, 3.72954768e-01,
9.99926298e-01, 6.70933418e-01, 9.93225344e-01, 8.56692424e-01,
9.99747865e-01, 9.57347194e-01, 9.96289021e-01, 9.64792260e-01,
9.58203312e-01, 9.78150234e-01, 9.50001067e-01, 8.80967220e-01,
9.99780266e-01, 9.86383074e-01, 9.16409014e-01, 9.97710778e-01,
9.60877887e-01, 9.73973053e-01, 9.58963425e-01, 9.71938883e-01,
9.60376306e-01, 9.98408524e-01, 9.96384251e-01, 9.98371035e-01,
9.89640691e-01, 9.99582270e-01, 9.35839677e-01, 8.69719435e-01,
6.70159582e-06, 8.51687440e-01, 9.87292829e-01, 9.55416211e-01,
9.79621792e-01, 7.80619549e-01, 9.99968572e-01, 9.86149047e-01,
7.08037082e-01, 9.64258721e-01, 9.89958939e-01, 9.20645369e-01,
9.99511269e-01, 9.90265169e-01, 9.96704392e-01, 9.64335265e-01,
9.98687863e-01, 9.99827070e-01])

```

```

In [37]: # Check the average probability values of predicted FDA-approved drugs
FDA_approved = y_pred > 0.5
(y_val == FDA_approved).mean()

```

Out[37]: 0.9205128205128205

Using the model

Let us check the performance of the model by comparing model predictions with the Test set of the data. This data subset was not seen the model during training.

```

In [38]: y_pred_test = model.predict_proba(X_test)

```

```

In [39]: # Comparing with collected data:

i = 55 # vary i to check various data-points in the test subset

y_pred_i = y_pred_test[i][1]
print(f'Predicted Probability: {y_pred_i} \nPredicted FDA to Approve Drug: {y_pred_i > 0.5} \n')

y_test_i = y_test.values[i]
print(f'Actual Empirical Probability: {y_test_i} \nFDA Approved Drug: {y_test_i > 0.5} ')

```

```

Predicted Probability: 0.7591315722108296
Predicted FDA to Approve Drug: True

```

```

Actual Empirical Probability: 1.0
FDA Approved Drug: True

```

For your Project #2, extend this notebook by performing model selection, metrics of classification, even hyperparameter grid-search, etc.