

```
In [1]: import requests
import io
import sys
import json
import itertools
from IPython.display import Image
from bs4 import BeautifulSoup
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import plotly.express as px
import sqlite3 as sql
from sklearn.model_selection import train_test_split, cross_val_score
import statsmodels.api as sm
import statsmodels.stats.api as sms
import scipy.stats as stats
from scipy.stats import chisquare
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score, f1_score, precision_recall_curve, roc_curve, auc, mean_squared_error
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from imblearn.over_sampling import RandomOverSampler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

```
In [2]: import warnings
warnings.filterwarnings("ignore")
```

A class of functions that pull data from sources and store the dataframes in a json.

```
In [3]: class Json():
    global json_storage
    d={}
    s=json.dumps(d)
    json_storage = json.loads(s)
    def __init__(self, df_name):
        self.df_name=df_name
    def csv(self, url):
        if url[0]=='r':
            download = requests.get(url).content
            df = pd.read_csv(io.StringIO(download.decode('utf-8')))
            df=pd.DataFrame(df)
            json_storage[self.df_name]=[url,df]
        else:
            df=pd.read_csv(url)
            df=pd.DataFrame(df)
            json_storage[self.df_name]=[url,df]
        return(df)
    def excel(self, e, s):
        df=pd.read_excel(e, s)
        json_storage[self.df_name]=[e,df]
        return(df)
    def web_scrape(self, u, c):
        url = requests.get(u).text
        soup = BeautifulSoup(url,'lxml')
        table = soup.find('table')
```

```
table_rows = table.find_all('tr')
l = []
for tr in table_rows:
    td = tr.find_all('td')
    row = [tr.text for tr in td]
    l.append(row)
df=pd.DataFrame(l, columns=c)
json_storage[self.df_name]=[u,df]
return(df)
def file(self, f):
    file = open(f, "r")
    read=file.read()
    json_storage[self.df_name]=read
    return(read)
```

A class of graphing functions.

```
In [ ]: class graph:
    def choropleth(self, d, location, lm, c, af, s, t):
        fig = px.choropleth(d , locations = location, locationmode = lm, color = c
            ,animation_frame=af, scope=s)
        fig.update_layout(title_text = t)
        fig.show()
    def bar_chart(self, x, y, t, x2, y2):
        fig, ax = plt.subplots(figsize=(50,150))
        width=.25
        ax.barh(x, y, width, color='red')
        for i, v in enumerate(y):
            ax.text(v, i, str(v), color='blue')
        plt.rcParams.update({'font.size': 20})
        plt.title(t, fontsize=40)
        plt.xlabel(x2, fontsize=30)
        plt.ylabel(y2, fontsize=30)
        plt.show()
```

A function that prints summary statistics.

```
In [ ]: def desc(x):
    print(x.name)
    print('Count:', x.count())
    print('Mean:', x.mean())
    print('Standard Deviation:', x.std())
    print('Min:', x.min())
    print("Q1 quantile: ", np.quantile(x, .25))
    print("Q2 quantile: ", np.quantile(x, .5))
    print("Q3 quantile: ", np.quantile(x, .75))
    print('Max:', x.max())
```

A function that creates a dataframe of covid-19 totals and prints the totals.

```
In [ ]: def stats(d, dd1, name, dd2, ddd):
    for i in d:
        p=dd1.loc[dd1[name]==f'{i}']['Population']
        p=int(p)
        c=dd1.loc[dd1[name]==f'{i}']['Total_Cases']
        c=int(c)
        d=dd1.loc[dd1[name]==f'{i}']['Total_Deaths']
        d=int(d)
        r=dd1.loc[dd1[name]==f'{i}']['Total_Recovered']
        r=int(r)
```

```
t=dd1.loc[dd1[name]==f'{i}']['Total_Tests']
t=int(t)
a=dd1.loc[dd1[name]==f'{i}']['Active_Cases']
a=int(a)
if name=='Country':
    try:
        v=dd2.loc[dd2[name]==f'{i}'].iloc[[-1]]['total_vaccinations']
    except:
        v=0
elif name == 'State':
    try:
        v=dd2.loc[dd2[name]==f'{i}']['Vaccines_Administered']
    except:
        v=0
v=int(v)
try:
    tp=(t/p)*100
    if tp>=100:
        tp=99.99
except:
    tp=0
try:
    ct=(c/t)*100
    if ct>=100:
        ct=99.99
except:
    ct=0
try:
    rc=(r/c)*100
    if rc>=100:
        rc=99.99
except:
    rc=0
try:
    dc=(d/c)*100
    if dc>=100:
        dc=99.99
except:
    dc=0
try:
    vp=(v/p)*100
    if vp>=100:
        vp=99.99
except:
    vp=0
print(f'{i}')
print('Population Total:', p)
print('Tested Total:', t)
print('Tested Percentage of Population:', tp, '%')
print('Cases Total:', c)
print('Active Cases:', a)
print('Confirmed Cases Percentage of Tested:', ct, '%')
print('Recoveries Total:', r)
print('Recoveries Percentage of Confirmed Cases:', rc, '%')
print('Deaths Total:', d)
print('Deaths Percentage of Confirmed Cases:', dc, '%')
print('Vaccinations:', v)
print('Vaccinations of Population:', vp, '%')
print('-----')
ddd.loc[len(ddd.index)] = [f'{i}', p, t, tp, c, a, ct, r, rc, d, dc, v, vp]
```

A function that prints a confusion matrix.

```
In [ ]: def con_mat(y, y_pred):
        print('\nConfusion Matrix')
        print('-----')
        cm=pd.DataFrame(confusion_matrix(y, y_pred))
        print(cm)
        plt.matshow(cm)
        plt.title('Predicted')
        plt.ylabel('Actual')
        plt.colorbar()
        cm
```

A function that prints a classification metrics.

```
In [4]: def Metrics(labels, preds):
        actual_pos = labels == 1
        actual_neg = labels == 0
        tp = (preds == 1) & (actual_pos)
        fp = (preds == 1) & (actual_neg)
        tn = (preds == 0) & (actual_neg)
        fn = (preds == 0) & (actual_pos)
        precision=precision_score(labels, preds)
        recall=recall_score(labels, preds)
        f1=f1_score(labels, preds)
        accuracy=accuracy_score(labels, preds)
        specificity = sum(tn) / (sum(tn) + sum(fn))
        print("Precision Score: {}".format(precision))
        print("Recall Score: {}".format(recall))
        print("F1 Score: {}".format(f1))
        print("Accuracy Score: {}".format(accuracy))
        print("Specificity Score: {}".format(specificity))
        lr_precision, lr_recall, _ = precision_recall_curve(labels, preds)
        plt.plot(lr_recall, lr_precision, marker='o')
        plt.title('Precision Recall Tradeoff')
        plt.xlabel('Recall')
        plt.ylabel('Precision')
```

a function that prints roc auc.

```
In [ ]: def roc(y, y_hat):
        false_positive_rate, true_positive_rate, thresholds = roc_curve(y, y_hat)
        roc_auc = auc(false_positive_rate, true_positive_rate)

        sns.set_style('darkgrid', {'axes.facecolor': '0.9'})

        plt.figure(figsize=(10, 8))
        lw = 2
        plt.plot(false_positive_rate, true_positive_rate, color='darkorange',
                  lw=lw, label='ROC curve')
        plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.yticks([i/20.0 for i in range(21)])
        plt.xticks([i/20.0 for i in range(21)])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Receiver operating characteristic (ROC) Curve')
        plt.legend(loc='lower right')
```

```
print('AUC: {}'.format(auc(false_positive_rate, true_positive_rate)))
plt.show()
```

A function that prints muticlass metrics.

```
In [ ]: def multiclass(model,x,y,y_hat,classes):
        sort=sorted(set(classes))
        for i,l,c in zip(y,y_hat,sort):
            print(f'Class:{c}')
            Metrics(i,l)
            roc(i,l)
            cv_score = cross_val_score(model, x, l, cv=5, scoring='roc_auc')
            mean_cv_score = np.mean(cv_score)
            print(f"Cross Validated ROC AUC score: {mean_cv_score}")
            print('_____')
```

A function that prints time series analysis.

```
In [ ]: def forecast(dd, w, start, end, s, l, datel, date2, title):
        date3=dd.index[0]
        roll_mean = dd.rolling(window=w, center=False).mean()
        roll_std = dd.rolling(window=w, center=False).std()
        fig = plt.figure(figsize=(12,7))
        plt.plot(dd, color='blue', label='Original')
        plt.plot(roll_mean, color='red', label='Rolling Mean')
        plt.plot(roll_std, color='black', label = 'Rolling Std')
        plt.legend(loc='best')
        plt.title(f'{title} Trend')
        plt.show(block=False)

        dfctest = adfuller(dd)
        dfoutput = pd.Series(dfctest[0:4], index=['Test Statistic', 'p-value', '#Lags Used', 'Number of Observations Used'])
        for key,value in dfctest[4].items():
            dfoutput['Critical Value (%s)'%key] = value
        print (f'{title} Dickey-Fuller test results: \n')
        print(dfoutput)

        fig, ax = plt.subplots(figsize=(16,3))
        plot_acf(dd, ax=ax, lags=1)
        plt.title(f'{title} autocorrelation')
        plt.show()

        fig, ax = plt.subplots(figsize=(16,3))
        plot_pacf(dd, ax=ax, lags=1)
        plt.title(f'{title} partial autocorrelation')
        plt.show()

        print(f'{title} AIC Scores:')
        # Define the p, d and q parameters to take any value between 0 and 2
        p = d = q = range(0, 2)

        # Generate all different combinations of p, d and q triplets
        pdq = list(itertools.product(p, d, q))

        # Generate all different combinations of seasonal p, d and q triplets
        pdqs = [(x[0], x[1], x[2], 7) for x in list(itertools.product(p, d, q))]
        # Run a grid with pdq and seasonal pdq parameters calculated above and get the best AIC value
        ans = []
        for comb in pdq:
            for combs in pdqs:
```

```

try:
    mod = sm.tsa.statespace.SARIMAX(dd,
                                    order=comb,
                                    seasonal_order=combs,
                                    enforce_stationarity=False,
                                    enforce_invertibility=False)

    output = mod.fit()
    ans.append([comb, combs, output.aic])
    print('ARIMA {} x {}12 : AIC Calculated ={}'.format(comb, combs, output.aic))
except:
    continue
# Find the parameters with minimal AIC value
ans_df = pd.DataFrame(ans, columns=['pdq', 'pdqs', 'aic'])
values=ans_df.loc[ans_df['aic'].idxmin()]
# Plug the optimal parameter values into a new SARIMAX model
ARIMA_MODEL = sm.tsa.statespace.SARIMAX(dd,
                                         order=values['pdq'],
                                         seasonal_order=values['pdqs'],
                                         enforce_stationarity=False,
                                         enforce_invertibility=False)

# Fit the model and print results
output = ARIMA_MODEL.fit()

print(f'{title} ARIMA:', output.summary())

output.plot_diagnostics(figsize=(15, 18))
plt.show()

# Get predictions and calculate confidence intervals
pred = output.get_prediction(start=pd.to_datetime(date3), dynamic=False)
pred_conf = pred.conf_int()

# Plot real vs predicted values along with confidence interval

# Plot observed values
ax = dd.plot(label='observed', figsize=(15, 18))

# Plot predicted values
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=0.9)

# Plot the range for confidence intervals
ax.fill_between(pred_conf.index,
                pred_conf.iloc[:, 0],
                pred_conf.iloc[:, 1], color='g', alpha=0.5)

# Set axes labels
plt.title(f'{title} actual and predicted values with confidence interval')
ax.set_xlabel('Date')
ax.set_ylabel(title)
plt.legend()

plt.show()
# Get the real and predicted values

forecasted = pred.predicted_mean
truth = dd
# Compute the mean square error
rmse=np.sqrt(mean_squared_error(truth, forecasted))

```

```
print(f'{title} Root Mean Squared Error:{round(rmse, 2)}')
```

```
f=output.forecast(steps=s)
prediction = output.get_forecast(steps=s)
pred_conf_f = prediction.conf_int()
print(f'{title} Forecast:')
print(f)
start[title]=dd.iloc[-1]
end[title]=f.iloc[-1]
```

```
# Plot future predictions with confidence intervals
ax = dd.plot(label='observed', figsize=(20, 15))
prediction.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_conf_f.index,
                pred_conf_f.iloc[:, 0],
                pred_conf_f.iloc[:, 1], color='k', alpha=0.25)
ax.set_xlabel('Date')
ax.set_ylabel(title)
plt.title(f'{title} trend and future predictions with confidence interval')
```

```
plt.legend()
plt.show()
```

A function that returns percentage change.

```
In [ ]: def pecentage_change(x1,x2):
        c=((x2-x1)/x1)*100
        return(f'Pecentage Change: {c}%')
```