```
import pandas as pd
import numpy as np
import datetime as dt
import plotly.offline as py
import plotly.graph_objs as go
import math as m

import matplotlib.pyplot as plt
data = pd.read_csv("data_tibau.csv", names=['time', 'velocity', 'direction'])
data
data['time'] = pd.to_datetime(data['time'], format='%Y-%m-%d %H:%M:%S')
data['year'] = data['time'].dt.strftime("%Y") ## Extracting year to get mean velocity
data
data[['velocity', 'year']].groupby(by='year').mean()
mean_velocity = data[['velocity', 'year']].groupby(by='year').mean()
mean_velocity
vel = list(data['velocity'])
scaled_velocity = list()

vel_min = data['velocity'].min()
vel_max = data['velocity'].max()

for i in vel:
   scaled_velocity.append((i - vel_min)/(vel_max - vel_min))

data['velocity'] = scaled_velocity ## Extracting max and min velocity, this way we preprocess the
data for predictions. Later we'll reescale the predicted data

mean_velocity = 4.90
rm = 0.15 ## dead radius
ro = 1.18 ## Estimated air density
P = 830 ## desired potency
cp = 0.4 ## standard 0.35 - 0.45
visc_ar = 1.82*pow(10,-5) ## air viscosity
n = 0.90 ## gearbox efficiency
Pi = m.pi
mi = 7 ## tip-air speed
B = 3 ## blades
Cl = 1.70 ## S1223 Profile
Cd = 0.0163 ## drag
sections = 15 ## sections
R = m.sqrt(pow(rm,2)+P/(cp*n*0.5*ro*Pi*pow(mean_velocity,3))) #------------------------------ raio do
rotor
w = mi*mean_velocity/R #--------------------------------------------------------------velocidade angular
```

```
R,w
radius = []
ratio = []
re = [] ##  Reynolds

for i in range(sections):
    r_section = R*rm + (i*((R - R*rm)/(sections-1))) ## Formula to avoid going to the tip
    ratio_section = r_section/R
    radius.append(r_section)
    ratio.append(ratio_section)
    reynolds = ro*mean_velocity*radius[i]/visc_ar
    re.append(reynolds) ## Calculate Reynolds, and the radius of the section

radius[sections-1] = radius[sections-2]+(radius[sections-1]-radius[sections-2])*0.1
## raio igual da na posicao final da erro e quanto mais proximo da penultima, melhor o
resultado

radius, ratio, re
chord_schmitz = []

for i in range(sections):
    ang = mi*radius[i]
    atan = R / ( ang )
    sen_2 = (1/3) * m.atan( atan )

    section_chord = (16*Pi*radius[i]) / (B*Cl)  * pow(m.sin( sen_2 ),2)
    chord_schmitz.append(section_chord)

chord_schmitz

## Used to calculate the chord at each section
for j in range(sections):

    error_a = 1 ## Start relative error
    error_a_ = 1
    a = 0.001 # Start a and a'
    a_ = 0.001
    i = 0 ## Count
    ac = 0.2 ## ac correction from Schmitz

    while (error_a > 0.001) and (error_a_ > 0.001):

        phi = m.atan(((1-a)/(1+a_))*(mean_velocity/(radius[j]*w))) ## Relative attack angle
```

```
        Cx = Cl*(m.sin(phi)) - Cd*(m.cos(phi)) ##  Tangential coef.
        Cy = Cl*(m.cos(phi)) + Cd*(m.sin(phi)) ## Normal coef.


        f = (B/2) * (R-radius[j])/(radius[j]*m.sin(phi))
        F = (2/Pi) * (m.acos(m.exp(-f)))
        sol = (chord_schmitz[j]*B)/(2*Pi*radius[j]) ## Terms to calculate the new a and a' from the
Blade Element Method

        if a > 0.2: ## Correction for values of a greater than 0.2
          K = (4*F*pow(m.sin(phi),2))/(sol*Cy)
          a_new = 0.5 * (2 + K*(1 - 2*ac) - m.sqrt( pow((K*(1-2*ac)+2),2) + 4*(K*pow(ac,2) -1 ) ))
        else:
          a_new = 1 / (((4*F*pow(m.sin(phi),2))/(sol*Cy)) + 1)

        a_new_ = 1 / (( (4*F*m.sin(phi)*m.cos(phi)) / (sol*Cx)) - 1)

        error_a = m.fabs(((a_new - a)/a))
        error_a_ = m.fabs(((a_new_ - a_)/a_)) ## Calculate new error

        a = a_new
        a_ = a_new_ ## Substitute values

        i = i + 1

    print("Section ", j, "Chord: ", np.round(chord_schmitz[j],6), ", Phi:",
np.round(m.degrees(phi),2))
data ## Checking the data
velocity_train = np.array(data['velocity'][0:9948])
velocity_train.shape
velocity_test = np.array(data['velocity'][9941:])
velocity_test.shape  ## Divided the data from before and after may, 2021
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense  ## Importing relevant libraries for DL
def split_data(sequence, n_steps):
    X, Y = list(), list()
    for i in range(len(sequence)):
        initial = i
        end = i + n_steps
        if end > len(sequence)-1:
          break
```

```
        x_seq = sequence[initial:end]
        y_seq = sequence[end]
        X.append(x_seq)
        Y.append(y_seq)
    return np.array(X), np.array(Y)

## Creating a sequence of data, this will feed the neural network. n_steps is equal to the number of samples we want to
## use to predict a new value. In this example, we'll try to predict only one value based on 7 previous values. In a future work, we'll try more than one.
n_steps = 7
X1, Y1 = split_data(velocity_train, n_steps)
from sklearn.model_selection import train_test_split as tts
X_train, X_val, Y_train, Y_val = tts(X1, Y1, test_size=0.10) ## This will be our validation data, as we already have our test data
n_features = 1
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], n_features)) ## n_features is the number of values we'll predict
X_val = X_val.reshape((X_val.shape[0], X_val.shape[1], n_features)) ## Reshape the data

model = keras.Sequential([keras.layers.Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)),
                          keras.layers.Dense(1)])  ## Creation of the LSTM layer and one single output
model.compile(optimizer='adam',
              loss='mse') ## Loss function
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(min_delta=0.00005, patience=50) ## This will make sure our model is improving, and if not, the fit will stop
history = model.fit(x=X_train,
                    epochs=200,
                     y=Y_train,
                     validation_data=(X_val,Y_val),
                     verbose=2,
                     callbacks=[early_stopping])
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1) ## Saving fit history for plotting
fig, ax = plt.subplots(figsize=(15,9))

ax.plot(epochs, loss, label='Training MSE')
ax.plot(epochs,val_loss, label='Validation MSE')
ax.set_title("Training and Validation Accuracy")
```

```
ax.legend()
def predict_data(sequence, n_steps):

    x_real, y_real = split_data(sequence, 7)

    predicted, real = list(), list()

    count = 0

    for i, j in zip(x_real,y_real):
        i = i.reshape(1,7,1)
        prediction = model.predict(i)[0][0] ## This way we extract the value
        print("Real: ", j, "Predicted: ", prediction)
        real.append(j)
        predicted.append(prediction)

    return predicted, real

## Creating a function to receive data, and then predict it. It does the same of the split data, it
just adds the prediction part
predicted, real = predict_data(velocity_test, n_steps) ## Predict data in the test dat (May)
re_vel = list(predicted)
re_vel_real = list(real)

rescaled_velocity = list()
rescaled_real_velocity = list()

for i,j in zip(re_vel, re_vel_real):
    rescaled_velocity.append(i * (vel_max - vel_min) + vel_min)
    rescaled_real_velocity.append(j * (vel_max - vel_min) + vel_min)

    ## Reescaling the data, based on our max and min velocity
May = pd.DataFrame({"time":np.array(data['time'][9948:]), "real":rescaled_real_velocity,
"predictions":rescaled_velocity})
May

## Creating a May dataframe to compare results
import plotly
import plotly.graph_objs as go
import plotly.offline as py
trace = [go.Scatter(x=May.time,
                    y=May.real,
                    name='Real'),
```

```
        go.Scatter(x=May.time,
                   y=May.predictions,
                   name='Predicted')]

layout = go.Layout(title="Prediction x Real values of Wind Speed")

fig = go.Figure(data=trace, layout=layout)

py.iplot(fig)
power_predict = list()
power_real = list()

for i,j in zip(May['predictions'], May['real']):
   if i > 3:
      power_predict.append(0.20*0.5*ro*pow(i,3)*pow((R-rm),2)*Pi)
      power_real.append(0.20*0.5*ro*pow(j,3)*pow((R-rm),2)*Pi)
      ## rm is the "dead radius" where the turbines pratically doesn't "generate" energy
      ## 0.20 takes into account the betz limit and efficiency for 3 blades and gearbox
   else:
      power_predict.append(0)
      power_real.append(0)

print("The total energy generated for this wind turbines is estimated as: ",
np.sum(power_predict),
     "W. The real production would be: ", np.sum(power_real)," W")
```