Probabilistic Methods for Diagnosing Parkinson's Disease in Hand-Drawn Spirals

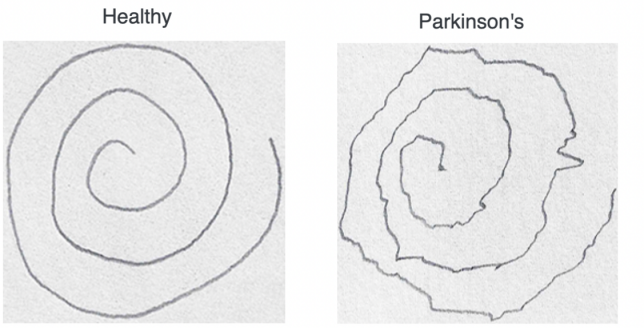
Rohan Sikand

CS 109, Winter 2022

Video link.

1 Introduction

We propose several different methods for identifying Parkinson's disease through drawings of spirals. In essence, this is a classification task: given an image of a hand-drawn spiral, does the patient, who drew the image, have Parkinson's disease?



Parkinson's disease is a disease of the nervous system that affects movement. It often leads to instability in movements and tremors. The diagnosis of Parkinson's disease is ambiguous: there is no specific test. A trained medical doctor makes a diagnosis through a historical analysis of patient symptoms, tests, and other factors. The problem is that a trained medical professional isn't always available–especially in under-resourced, remote areas. Thus, there is a need for an efficient heuristic for the diagnosis of Parkinson's disease. here we propose to study hand-drawn spirals using various concepts from probability theory.

2 Methods

We developed and tested several methods for this classification task–all based on probability theory.

2.1 Data

The data was collected from Kaggle (link) and the structure of the data is as follows:

- Split into `train` and test.
- For both 'train' and 'test', there are two classes of images: `healthy` and `parkinson`.
- There are 36 training samples for each class and 15 test samples for each class.

2.2 Probabilistic modeling of curvature

Here we propose an interesting paradigm: using **curvature as a measure**. Eventually, we will create a distribution from curvature measures but let's first discuss how to measure curvature.

The motivation is that the "curvature" of the drawing from a Parkinson's patient might be a bit more jagged than that from a healthy patient. We will use probability to analyze such curvature but we first need a way to actually measure the curvature of the drawing from the image.

For each image, a segmentation process, via thresholding (by determining which pixel's had grayscale values that were below the value 230) was performed to identify which pixel values had pencil present (i.e. part of the drawing). This yielded a collection of Cartesian (x, y) coordinates that compose a curve that resembles the hand drawing from the image.

Given these (x, y) coordinates for each sample, we can calculate the "curvature" for each curve by using ideas from Calculus (see here and here for more details on this calculation).

Now that we have a measure of curvature for every sample, it is time to use probability theory to predict classifications.

Let X and Y be random variables that represent the measure of curvature for healthy patients and parkinson's patients respectively. From the training data, we approximate these random variables as normals (using unbiased estimations for the parameters). For each sample in the training data, a curvature was measured and appended to one of two new n vectors which we denote as $\mathbf{v} = [v_1, ..., v_n]$ and $\mathbf{w} = [w_1, ..., w_n]$ for healthy patients and parkinson's patients respectively. Thus,

$$X \sim N\left(\mu = \frac{1}{n}\sum_{i=1}^{n} v_i, \sigma^2 = \frac{\sum (v_i - \bar{v})^2}{n-1}\right), Y \sim N\left(\mu = \frac{1}{n}\sum_{i=1}^{n} w_i, \sigma^2 = \frac{\sum (w_i - \bar{w})^2}{n-1}\right)$$

Now that we have our random variable distributions approximated as normals from the training data, we can make predictions on the test data. Our approach for this was as follows: (1) for each test sample, get the curvature measure, denote it as c_i; (2) calculate the probability density from the probability density function for each $X = c_i$ and $Y = c_i$; (3) divide the two probability densities to get a ratio and use that ratio to determine the prediction. We derive this mathematically as follows (in summary, we are interested in finding the ratio of probability densities[1]):

$$\frac{f_x(c_i)}{f_y(c_i)}.$$

From here, we check if $\frac{f_x(C_i)}{f_y(C_i)} \geq 1$. If so, the probability density ratio was above 1:1 and so the class prediction for the sample c_i is the class represented by X: `healthy`. If the ratio was below 1, then the prediction is the class represented by Y: `Parkinson`.

Utilizing this approach on the test data, we predicted 18/30 samples correctly for a total accuracy of 60%. Not terrible. But we can do better.

2.3 Logistic regression

We perform logistic regression on the images directly as samples. Some pre-processing was performed on the data: specifically, each pixel value was made binary (no pencil, 0, or pencil present, 1) based on a threshold value (if the grayscale value of the current pixel was below 230, it was set to 1 and and if greater than 230, it was set to 0). Second, all of the image samples for the training and testing sets were downsampled to reduce the size of the image by a factor of 16. Each image was originally 256×256 and after downsampling, 16×16 for a total of 256 features per input[2].

Logistic regression is essentially a function approximation that maps $\mathbf{x}$ to y. This is the same thing as $g(\mathbf{x}) = \text{argmax}\, \hat{P}(Y = y \mid \mathbf{X})$. Mathematically, each prediction is made based on the trained values for the parameters (θ's) as follows:

$$P(Y = 1 \mid \mathbf{X} = \mathbf{x}) = \sigma\left(\sum_i \theta_i x_i\right)$$

In this case, we denote the event $Y = 1$ as a prediction of the `parkinson` class and $Y = 0$ as a prediction of the `healthy` class where $\mathbf{X}$ is a list of the feature vectors (inputs). This is a binary

[1] This part of the approach is similar to problem set 4, question 7.

[2] The reasoning for doing this is because the number of features per input would have been to large to train locally.

classification problem and so we have:

$$P(Y = 1 \mid \mathbf{X} = \mathbf{x}) = \sigma\left(\theta^T \mathbf{x}\right),$$
$$P(Y = 0 \mid \mathbf{X} = \mathbf{x}) = 1 - \sigma\left(\theta^T \mathbf{x}\right)$$

This defines the forward pass which is determined based on the values for θ, the model's parameters. We learn these parameters by training the algorithm on the training data. To do this, we use the gradient ascent algorithm by updating the weights (θ's) at iteration.

To make things flow mathematically[3], we interpret a class prediction as an indicator random variable $Y \sim \text{Bern}(p)$ where $p = \sigma\left(\theta^T \mathbf{x}\right)$. We need a way to measure the performance of how well we are estimating these parameters. For that, we use the likelihood of the continuous version of the probability density function for a Bernoulli defined as

$$L(\theta) = \prod_{i=1}^{n} \sigma\left(\theta^T \mathbf{x}^{(i)}\right)^{y^{(i)}} \cdot \left[1 - \sigma\left(\theta^T \mathbf{x}^{(i)}\right)\right]^{\left(1-y^{(i)}\right)}$$

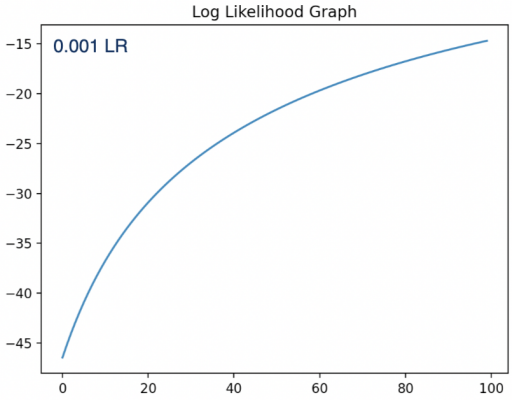
For numerical stability, we convert this to log space and define the log-likelihood function as follows:

$$LL(\theta) = \sum_{i=1}^{n} y^{(i)} \log \sigma\left(\theta^T \mathbf{x}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left[1 - \sigma\left(\theta^T \mathbf{x}^{(i)}\right)\right]$$

Now to train the algorithm using gradient ascent, we iterate over the data a number of times and at each point, update our parameters for the model. Specifically,

$$\theta_j^{\text{new}} = \theta_j^{\text{old}} + \eta \cdot \frac{\partial LL\left(\theta^{\text{old}}\right)}{\partial \theta_j^{\text{old}}} = \theta_j^{\text{old}} + \eta \cdot \sum_{i=1}^{n} \left[y^{(i)} - \sigma\left(\theta^T \mathbf{x}^{(i)}\right)\right] x_j^{(i)}.$$

We trained the model for 100 steps (epochs) with a learning rate $\eta = 0.001$. These hyperparameters were determined through experimentation. We plot the log-likelihood for the model predicting on the training data at each step in the graph below:



We can see that the model's performance was increasing throughout the entirety of training and was slowly converging.

After the model was trained by following the above approach, we predicted on the test samples. The logistic regression approach was able to achieve an **accuracy score of** 86.7%! Quite the improvement!

[3]The following logistic regression derivations were mainly adapted from the CS 109 Course Reader.

3 Conclusion

We have shown here that it is possible to achieve a relatively high accuracy in "diagnosing" (as a heuristic) patients with Parkinson's by using different probabilistic methods. We started with the assumption that the curvature of the spirals can be approximated with normal random variables. Aiming to improve performance, we then applied a different approach using logistic regression on the images themselves which resulted in an overall increase in accuracy.

Future work would involve expanding these methods (particularly the method described in section 2.2) to work for other datasets with a limited number of samples (as we had here). We believe that utilizing our own intuition (e.g. recognizing the difference in curvature between spirals), combined with an informed probabilistic framework is a step forward in making machine learning work with less data.

Appendix: Code

Segmenting drawing from image using thresholding:

```
def segment(image):
    """
    takes in SimpleImage object and returns a
    list of coordinates where the pixel intensity
    is considered "pencil present" (i.e. grayscale
    value is < 230). In essence, this function
    returns the coordinates of the spiral.
    """

    # downsample image first to increase efficiency
    downsampled_size = SimpleImage.blank(width = (image.width), height =
    ↪  (image.height))
    image.make_as_big_as(downsampled_size)

    coords = []  # order matters!
    for pixel in image:
        grayscale = (pixel.red + pixel.green + pixel.blue)/3
        if grayscale < 230:
            coords.append([pixel.x, pixel.y])
    return coords
```

Function for computing curvature measure (adapted from here):

```
import numpy as np
import matplotlib.pyplot as plt


def measure_curvature(coordinates):
    """
    Function that takes in a set of x,y coordinates and returns
    the magnitude of the curvature of the curve formed from
    the coordinate pairs.
    Parameters:
        - coordinates: 2d list where each element is len 2 representing
        the x,y coordinate pair.
    Returns:
        - curvature: magnitude of curvature.

    Source: https://www.delftstack.com/howto/numpy/curvature-formula-numpy/
    """


    coordinates = np.array(coordinates)

    # plt.plot(coordinates[:,0], coordinates[:,1])

    x_t = np.gradient(coordinates[:, 0])
    y_t = np.gradient(coordinates[:, 1])

    vel = np.array([ [x_t[i], y_t[i]] for i in range(x_t.size)])

    speed = np.sqrt(x_t * x_t + y_t * y_t)
    tangent = np.array([1/speed] * 2).transpose() * vel
```

```
    ss_t = np.gradient(speed)
    xx_t = np.gradient(x_t)
    yy_t = np.gradient(y_t)

    curvature_vec = np.abs(xx_t * y_t - x_t * yy_t) / (x_t * x_t + y_t * y_t)**1.5


    curvature_measure = np.linalg.norm(curvature_vec)

    return curvature_measure
```

File: `extractimgs.py` which is a script that iterates through the image files, extracts the pixels, thresholds them, and stores all of them into a list.

```
from simpleimage import SimpleImage
from glob import glob
from tqdm import tqdm
import pickle

DOWNSAMPLE = 16

train_prefix = 'spiral/training/'
test_prefix = 'spiral/testing/'

healthy_trainpath = train_prefix + 'healthy'
parkinson_trainpath = train_prefix + 'parkinson'
healthy_trainfiles = glob(healthy_trainpath + '/*.png')
parkinson_trainfiles = glob(parkinson_trainpath + '/*.png')

healthy_testpath = test_prefix + 'healthy'
parkinson_testpath = test_prefix + 'parkinson'
healthy_testfiles = glob(healthy_testpath + '/*.png')
parkinson_testfiles = glob(parkinson_testpath + '/*.png')


# cumulate training healthy data samples
train_healthy = []
for sample in tqdm(healthy_trainfiles):
    curr_image = SimpleImage(sample)
    downsampled_size = SimpleImage.blank(width = (curr_image.width // DOWNSAMPLE),
    ↪  height = (curr_image.height // DOWNSAMPLE))
    curr_image.make_as_big_as(downsampled_size)
    pixels = []
    for pixel in curr_image:
        grayscale = (pixel.red + pixel.green + pixel.blue)/3
        if grayscale < 230:
            grayscale = 1
        else:
            grayscale = 0
        pixels.append(grayscale)
    pixels.append(0)  # label
    train_healthy.append(pixels)




# cumulate training healthy data samples
train_parkinsons = []
```

```
for sample in tqdm(parkinson_trainfiles):
    curr_image = SimpleImage(sample)
    downsampled_size = SimpleImage.blank(width = (curr_image.width // DOWNSAMPLE),
    ↪  height = (curr_image.height // DOWNSAMPLE))
    curr_image.make_as_big_as(downsampled_size)
    pixels = []
    for pixel in curr_image:
        grayscale = (pixel.red + pixel.green + pixel.blue)/3
        if grayscale < 230:
            grayscale = 1
        else:
            grayscale = 0
        pixels.append(grayscale)
    pixels.append(1)  # label
    train_parkinsons.append(pixels)



# cumulate test healthy data samples
test_healthy = []
for sample in tqdm(healthy_testfiles):
    curr_image = SimpleImage(sample)
    downsampled_size = SimpleImage.blank(width = (curr_image.width // DOWNSAMPLE),
    ↪  height = (curr_image.height // DOWNSAMPLE))
    curr_image.make_as_big_as(downsampled_size)
    pixels = []
    for pixel in curr_image:
        grayscale = (pixel.red + pixel.green + pixel.blue)/3
        if grayscale < 230:
            grayscale = 1
        else:
            grayscale = 0
        pixels.append(grayscale)
    pixels.append(0)  # label
    test_healthy.append(pixels)


# cumulate test healthy data samples
test_parkinsons = []
for sample in tqdm(parkinson_testfiles):
    curr_image = SimpleImage(sample)
    downsampled_size = SimpleImage.blank(width = (curr_image.width // DOWNSAMPLE),
    ↪  height = (curr_image.height // DOWNSAMPLE))
    curr_image.make_as_big_as(downsampled_size)
    pixels = []
    for pixel in curr_image:
        grayscale = (pixel.red + pixel.green + pixel.blue)/3
        if grayscale < 230:
            grayscale = 1
        else:
            grayscale = 0
        pixels.append(grayscale)
    pixels.append(1)  # label
    test_parkinsons.append(pixels)
```

```
train_set = train_healthy + train_parkinsons
test_set = test_healthy + test_parkinsons


train_out_file = open("image_train_16.pkl", "wb")
pickle.dump(train_set, train_out_file)
train_out_file.close()

test_out_file = open("image_test_16.pkl", "wb")
pickle.dump(test_set, test_out_file)
test_out_file.close()
```

Probability density ratio approach described in section 2.2:

```
import pickle
import scipy
import math
from scipy import stats
import numpy as np

# load the data
healthy_file_train = open('spiral_curve_distro_train_healthy.pkl', 'rb')
healthy_distro_train = pickle.load(healthy_file_train)

parkinson_file_train = open('spiral_curve_distro_train_parkinson.pkl', 'rb')
parkinson_distro_train = pickle.load(parkinson_file_train)

healthy_file_test = open('spiral_curve_distro_test_healthy.pkl', 'rb')
healthy_distro_test = pickle.load(healthy_file_test)

parkinson_file_test = open('spiral_curve_distro_test_parkinson.pkl', 'rb')
parkinson_distro_test = pickle.load(parkinson_file_test)


# probs!
healthy_mean = np.mean(healthy_distro_train)
healthy_variance = np.var(healthy_distro_train)
parkinson_mean = np.mean(parkinson_distro_train)
parkinson_variance = np.var(parkinson_distro_train)

count = 0
for i in range(len(healthy_distro_test)):
    healthy_density = stats.norm.pdf(healthy_distro_test[i], healthy_mean,
    ↪  math.sqrt(healthy_variance))
    parkinson_density = stats.norm.pdf(healthy_distro_test[i], parkinson_mean,
    ↪  math.sqrt(parkinson_variance))

    ratio = healthy_density/parkinson_density
    if ratio >= 1:
        count += 1


# now for parkinson distro test
counter = 0
for i in range(len(parkinson_distro_test)):
    healthy_density = stats.norm.pdf(parkinson_distro_test[i], healthy_mean,
    ↪  math.sqrt(healthy_variance))
    parkinson_density = stats.norm.pdf(parkinson_distro_test[i], parkinson_mean,
    ↪  math.sqrt(parkinson_variance))
```

```
    ratio = parkinson_density/healthy_density
    if ratio >= 1:
        counter += 1


accuracy = (count + counter)/(len(healthy_distro_test) + len(parkinson_distro_test))
print("accuracy: ", accuracy)
```

Logistic regression program:

```
"""
File: logreg.py
-----------------
Code for logistic regression.
Adapted from my solution to pset 6, question 3.
"""

import numpy as np  # for data loading utilities
import sys  # command line args for file paths
import math  # for exponent
from tqdm import tqdm  # so I don't get bored waiting (progress bar)
import pickle
from matplotlib import pyplot as plt

# constants (hyperparameters)
LEARNING_RATE = 0.001
STEPS = 100


def sigmoid(x):
    """
    Returns output of Sigmoid function
    given x as an input.
    """

    return 1/(1 + math.exp(-x))


def forward_pass(sample, params):
    """
    Takes in feature vector, x, and
    returns probability that it belongs
    to class 1.
    """

    # model is simply list of params (weights)

    # step 1: dot product
    feature_vector = np.array(sample)  # includes bias variable
    parameters = np.array(params)
    energy = np.dot(parameters, feature_vector)

    # step 2: pass through squashing sigmoid
    prob = sigmoid(energy)

    return prob
```

```
def predict(sample, model):
    """
    Determines class based on if probability is > 0.5.
    """

    prob = forward_pass(sample, model)

    prediction = 0

    if prob >= 0.5:
        prediction = 1

    return prediction, prob


def train(train_set):
    """
    Takes in a training dataset and returns
    a list of trained parameters (i.e. the model).
    """

    # feature vector length (equal to num. params including
    # bias term)
    m = len(train_set[0]) - 1  # - 1 to not include label

    # intialize model (parameters)
    model = [0] * m

    likelihoods = []

    # update model params (gradient ascent training)
    for i in tqdm(range(STEPS)):
        # current gradient
        gradient = [0] * m
        for feature_vector in train_set:
            label = feature_vector[-1]  # y^i
            feature_vector = feature_vector[:m]  # x^i

            for j in range(len(feature_vector)):
                error = label - forward_pass(feature_vector, model)
                gradient[j] += error * feature_vector[j]

        for idx in range(len(model)):
            model[idx] += LEARNING_RATE * gradient[idx]

        likelihoods.append(log_likelihood(train_set, model))

    plt.plot(likelihoods)
    plt.title("Log Likelihood Graph")
    plt.show()

    return model



def test(model, test_set):
```

```
    """
    Takes in testing data set and the model
    and returns accuracy of models preds on
    test set.
    """

    correct = 0
    for vector in test_set:
        m = len(vector) - 1
        feature = vector[:m]
        label = vector[-1]

        prediction, prob = predict(feature, model)

        if prediction == label:
            correct += 1

    accuracy = correct/len(test_set)

    return accuracy



def log_likelihood(train_set, model):

    m = len(train_set[0]) - 1  # - 1 to not include label

    params = model
    summation = 0
    for vector in train_set:
        m = len(vector) - 1
        feature = vector[:m]
        label = vector[-1]

        part_one = label * math.log(forward_pass(feature, params))

        part_two = (1-label) * math.log(1 - forward_pass(feature, params))

        summation = summation + part_one + part_two

    return summation



def main():
    """
    Runs the program.
    """

    in_train_file = open('image_train_16.pkl', 'rb')
    train_set = pickle.load(in_train_file)

    in_test_file = open('image_test_16.pkl', 'rb')
    test_set = pickle.load(in_test_file)


    # add the bias terms here for both train and test sets
```

```
    new_train = []
    for elem in tqdm(train_set):
        # confirm each sample is of (16 x 16) + 1 length
        if len(elem) == 257:
            new_train.append(np.insert(elem, 0, 1))

    new_test = []
    for elem in tqdm(test_set):
        # confirm each sample is of (16 x 16) + 1 length
        if len(elem) == 257:
            new_test.append(np.insert(elem, 0, 1))

    training_data = np.array(new_train)
    testing_data = np.array(new_test)


    # get params
    model = train(training_data)

    # get test accuracy
    accuracy = test(model, testing_data)

    print("Accuracy: ", accuracy)


if __name__ == '__main__':
    main()
```

Reproduce results: how to run code

If you'd like to reproduce my results, first download the zip file containing the code and serialized data files (with all of the pre-processing done). Then, you can run the program `probdensity.py` for the method in section 2.2 and run `logistic.py` to run the logistic regression method from section 2.3.