Comparison of Linear Classification Methods For Human Brain Tractography Data

Introduction:

Diffusion tensor imaging (DTI) is a magnetic resonance imaging (MRI) method that produces images of the molecular diffusion processes in brain tissue. The white matter of the brain is composed of bundles of myelinated axons that connect various grey matter areas. Since water tends to diffuse along the length of axons rather than through the myelin barrier, the orientation of fiber bundles in each voxel can be modeled using a tensor. To generate a model for entire fiber tracks connecting brain regions, tractography algorithms are employed to follow the tensors voxel-by-voxel from a "seeded" voxel until the path terminates in the grey matter. This type of imaging is used in pre-neurosurgical planning to map important brain connections that are considered critical to motor and language function.

Modeling brain connections through tractography is an extremely noisy process. There are often more fiber populations in a given voxel than can be modeled accurately, so the voxel-by-voxel tracking will often track paths that are anatomically inaccurate. Often, complex anatomy will redirect a large population of tracking instances. This warps the shape of the modeled brain tracks and misrepresents the underlying anatomy. Trained human intervention is used to augment this processing by segmenting and "cleaning" the tracks based on a learned shape. Automating this process is difficult because streamline shapes are highly variable even within the same bundle of a healthy control. There is also significant inter-subject anatomical variation on top of pathologies, which make reliably classifying streamlines a very difficult task for a human or computer.

Many groups have worked on similar problems, but these projects tend to cluster streamlines instead of defining specific tracks of interest automatically based on the data from a trained neuroradiologist. One group used a tree classifier with the individual points of a streamline as features to segment the entire brain and achieved 80% true positives/all positives identified and 55% true positives/all real positives (Mayer, et al.). Another group developed a method that clusters similar streamlines into bundles based on a supervised learning approach in which a neuroradiologist identifies good and bad bundles, as opposed to individual streamlines (Olivetti et al.).

For this project, I focused on comparing different methods for classification of human brain tractography data. Specifically, each method is used to classify a set of tracts from the left External Capsule into three categories: Inferior Fronto-Occipital Fasciculus (IFOF), Uncinate Fasciculus, and noise.

Data Set

Instead of focusing on the whole brain, I began with the fiber populations that pass through the left External Capsule as a proof of principle. The Inferior Fronto-Occipital Fasciculus (IFOF) and the Uncinate Fasciculus are tracked simultaneously because of their proximity in the External Capsule region. These tracks must be separated and cleaned to produce the known shape of the underlying anatomy.

Our input to the algorithm is a set of tracts from the subject's left External Capsule, obtained from using a probabilistic tractography algorithm on diffusion tensor images (Figure 1). These tracts were also run through a separate algorithm to remove erroneous tracts that contained false loops formed by the tractography algorithm.

Our data set consisted of the fiber populations from the left External Capsule in fifteen different subjects. Two different experts in neuroradiology performed tractography on the brains of these subjects to produce around 2500 streamlines for each subject. The experts then established a ground truth by manually classifying tracks as either IFOF, Uncinate, or noise, using the software TrackVis (Figure 2). In total, the training set consisted of around 45,000 streamlines and the test set consisted of around 5,000 streamlines. It should be noted that the data sets from different patients were not registered to each other so there may be variation due to head angle or brain volume.

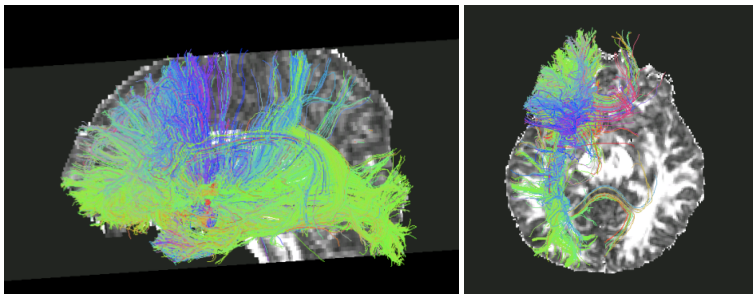


Figure 1: INPUT = Output of Tractography Algorithm tracking from the external capsule. This region contains two distinct fiber populations of interest (the Uncinate Fasciculus and the Inferior Fronto-Occipital Fasciculus

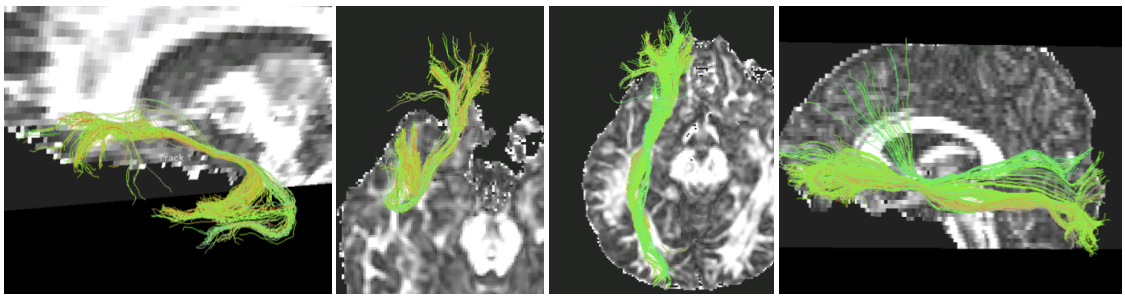


Figure 2: TRAINING DATA = Tracks that have been segmented and cleaned by a human operator. The left two panels are the Uncinate Fascicuis and the right two panels are the Inferior-Fronto-Occipital Faciculus

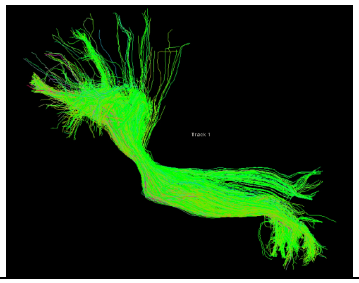
Feature Vectors

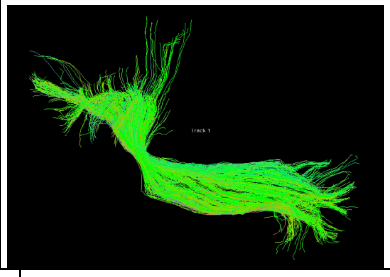
Since the streamlines were originally represented as variable lists of coordinates in three dimensional space, it was necessary to find a standard feature vector to represent this data. A number of different feature vectors were tested in order to find the optimal choice, including the histogram of orientations, contour length, end-to-end length, endpoint coordinates, and midpoint and quartile coordinates. The feature vector ultimately used to represent each streamline included the length of the contour, the end-to-end length, the ratio of contour length and end-to-end length, and the actual coordinates of the endpoints. These values were stored for each individual streamline to form the final data set that acts as input for each of the classification methods.

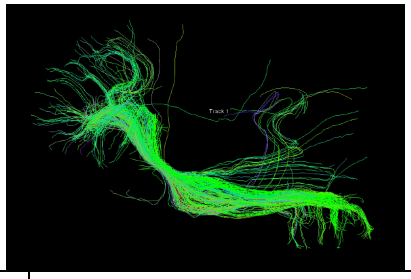
Classification Methods

Built-in SVM Model:
As a baseline, I classified the model using the built in support vector machine classifier from the sklearn python package. This method gave a specificity of around 92%, but the sensitivity was only around 79%. In investigating other classification methods, I hoped to find other methods that would have a better balance of specificity and sensitivity.

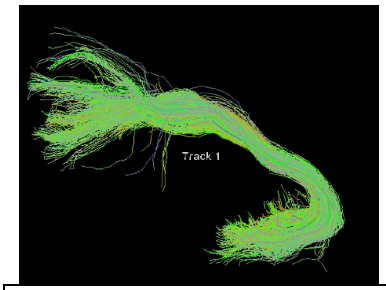
IFOF:


True Positive


False Negative

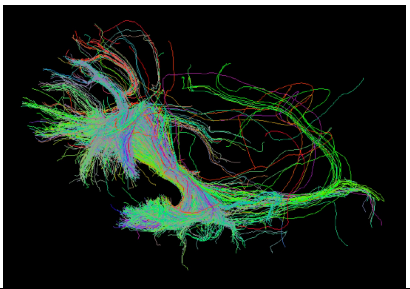

False Positive

Uncinate:



True Positive

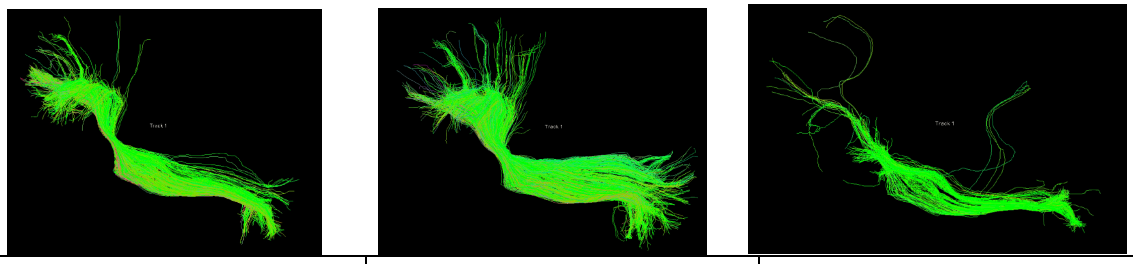
False Negative


False Positive

Gaussian Class Conditional Densities Model:

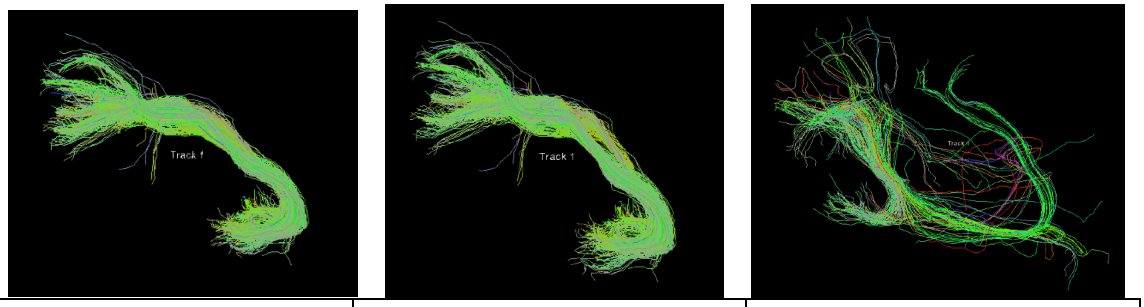
I also classified the model by using a Gaussian class conditional densities model, which assumed Gaussian distributions for each class. Though the overall accuracy was 93% and the specificity was very high at around 99%, the sensitivity was very low at 34%.

IFOF:



True Positive	False Negative	False Positive

Uncinate:

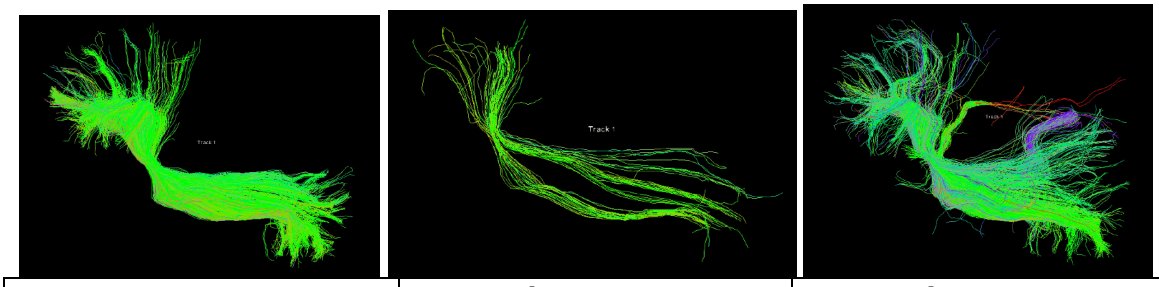


True Positive	False Negative	False Positive

Gaussian Naïve Bayes Model

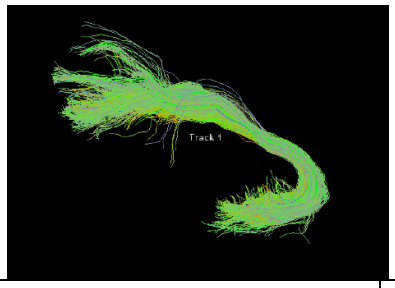
I also classified the model using a Gaussian naïve Bayes model. This version of naïve Bayes allows for continuous data and assumes that the features in each class follow a Gaussian distribution.

IFOF:



True Positive	False Negative	False Positive

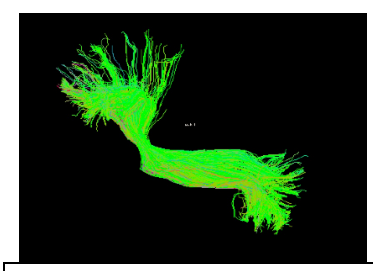
Uncinate:



True Positive	False Negative	False Positive

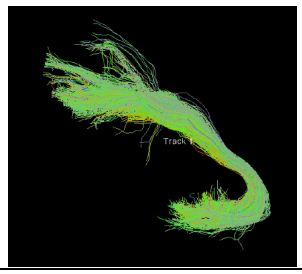
Logistic Regression With Stochastic Gradient Descent:
I also classified the model by using multivariate logistic regression implemented with stochastic gradient descent. This method gave the best overall results. It had the greatest overall accuracy at 94%. In addition, both the sensitivity and the specificity were very high at around 93-94%.

IFOF:



True Positive	False Negative	False Positive

Uncinate:



True Positive	False Negative	False Positive

Results and Discussion

	Overall Accuracy	Sensitivity	Specificity
Built-in SVM	90.79%	78.50%	91.88%
Gaussian Class Conditional Densities	93.64%	34.33%	98.92%
Gaussian Naïve Bayes	87.61%	98.06%	86.68%
Logistic Regression With Stochastic Gradient Descent	93.92%	92.56%	94.05%

In the comparison of different classification methods, it is apparent that there is a tradeoff between sensitivity and specificity when identifying brain tracts. I had initially investigated this problem using support vector machines and was motivated to try other approaches in order to find a better balance between sensitivity and specificity.

The Gaussian Class Conditional Densities model has the best overall accuracy and also the best specificity. However, this may be a poor and risky choice for this application. The algorithm only correctly identifies a small percentage of the streamlines. If a doctor were to use these results to plan a brain surgery, the chance of accidentally cutting into an important motor or language tract would be high.

The Gaussian Naïve Bayes model has the best sensitivity. This would be a very conservative estimate for this application. The algorithm catches nearly all of the important streamlines, but it also includes a lot of noise streamlines as well. The surgeon could be fairly confident that he would be able to avoid the important areas when cutting into the brain. However, this approach is also prohibitive and may unnecessarily prevent the doctor from performing needed surgery in certain areas.

The Logistic Regression model with stochastic gradient descent seems to be the best overall approach for pre-surgical mapping. The sensitivity and specificity are both fairly high, so there is a good balance between the risk of underestimating and overestimating the tract.

References:

Mayer, A, et. al. "A Supervised Framework for the Registration and Segmentation of White Matter Fiber Tracts". IEEE Trans Med Imaging. 2011 Jan;30(1):131-45. doi: 10.1109/TMI.2010.2067222. Epub 2010 Aug 16.

Olivetti, E. et. al. "Fast Clustering for Interactive Tractography Segmentation". Pattern Recognition in Neuroimaging (PRNI), *2013 International Workshop on Pattern Recognition in Neuroimaging*. pp. 42-45 Philadelphia, PA. June 2013

Appendix: Python Code for Classification

```
import nibabel as nib
import numpy as np
from math import exp, log, isnan
from random import randint
import glob
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from streamline_labels import streamline_labels
from sklearn.linear_model import BayesianRidge

def mdot(*args):
    return reduce(np.dot, args)

test_labels = np.loadtxt('test_labels.txt')
training_labels = np.loadtxt('training_labels.txt')
test_feature_vectors = np.loadtxt('test_feature_vectors.txt')
training_feature_vectors = np.loadtxt('training_feature_vectors.txt')

num_examples, num_features = training_feature_vectors.shape
num_test_examples, num_features = test_feature_vectors.shape

# Normalize examples to prevent overflow problems

training_features_mean = np.mean(training_feature_vectors,axis=0)
training_features_stdev = np.std(training_feature_vectors,axis=0)
norm_training_features = training_feature_vectors - training_features_mean[None,:]
norm_training_features = norm_training_features/training_features_stdev[None,:]
norm_test_features = test_feature_vectors - training_features_mean[None,:]
norm_test_features = norm_test_features/training_features_stdev[None,:]

column_of_ones = np.ones((num_examples,1))
norm_aug_training_features = np.hstack((column_of_ones, norm_training_features))
column_of_ones = np.ones((num_test_examples,1))
norm_aug_test_features = np.hstack((column_of_ones, norm_test_features))

mdl = 'LOGISTIC'

# BUILT-IN SVM MODEL
if(mdl == 'SVM'):
    print('SVM_Model')
    model = LinearSVC()
    model.fit(training_feature_vectors, training_labels)
    test_predictions = model.predict(test_feature_vectors)

# GAUSSIAN CLASS-CONDITIONAL DENSITIES
if(mdl == 'GAUSSIAN'):
    noise_features = norm_training_features[training_labels == 0]
    IFOF_features = norm_training_features[training_labels == 1]
    UNC_features = norm_training_features[training_labels == 2]

    # Max Likelihood Mean is sample mean for each of 3 classes
    noise_ml_mean = np.reshape(np.mean(noise_features, axis = 0), (1, num_features))
    IFOF_ml_mean = np.reshape(np.mean(IFOF_features, axis = 0), (1, num_features))
    UNC_ml_mean = np.reshape(np.mean(UNC_features, axis = 0), (1,num_features))

    # Max Likelihood Variance is pooled estimate of variance
    noise_residual = noise_features -noise_ml_mean
    IFOF_residual = IFOF_features - IFOF_ml_mean
    UNC_residual = UNC_features - UNC_ml_mean

    sigma_squared_ml =
(np.sum(np.square(noise_residual),axis=0)+np.sum(np.square(IFOF_residual),axis=0)+np.sum(
np.square(UNC_residual),axis=0))/float(num_examples)
    sigma_matrix_ml = np.diag(sigma_squared_ml)

    for i in xrange(num_features):
        for j in xrange(i+1,num_features):
            sigma_matrix_ml[i,j] =
(np.sum(noise_residual[:,i]*noise_residual[:,j])+np.sum(IFOF_residual[:,i]*IFOF_residual[
```

```
:,j])+np.sum(UNC_residual[:,i]*UNC_residual[:,j]))/float(num_examples)
            sigma_matrix_ml[j,i] = sigma_matrix_ml[i,j]

    # Max Likelihood of prior is proportion

    pi_ml_IFOF = len(IFOF_features)/float(num_examples)
    pi_ml_UNC = len(UNC_features)/float(num_examples)
    pi_ml_noise = len(noise_features)/float(num_examples)

    # Posterior Probability

    Beta_noise_term1 = -mdot(noise_ml_mean, np.linalg.inv(sigma_matrix_ml),
np.transpose(noise_ml_mean)) + log(pi_ml_noise)
    Beta_noise_term2 = np.dot(np.linalg.inv(sigma_matrix_ml),np.transpose(noise_ml_mean))
    Beta_noise = np.vstack((Beta_noise_term1, Beta_noise_term2))

    Beta_IFOF_term1 = -mdot(IFOF_ml_mean, np.linalg.inv(sigma_matrix_ml),
np.transpose(IFOF_ml_mean)) + log(pi_ml_IFOF)
    Beta_IFOF_term2 = np.dot(np.linalg.inv(sigma_matrix_ml),np.transpose(IFOF_ml_mean))
    Beta_IFOF = np.vstack((Beta_IFOF_term1, Beta_IFOF_term2))

    Beta_UNC_term1 = -mdot(UNC_ml_mean, np.linalg.inv(sigma_matrix_ml),
np.transpose(UNC_ml_mean)) + log(pi_ml_UNC)
    Beta_UNC_term2 = np.dot(np.linalg.inv(sigma_matrix_ml),np.transpose(UNC_ml_mean))
    Beta_UNC = np.vstack((Beta_UNC_term1, Beta_UNC_term2))

    IFOF_term =
np.exp(np.dot(np.transpose(Beta_IFOF),np.transpose(norm_aug_test_features)))
    UNC_term =
np.exp(np.dot(np.transpose(Beta_UNC),np.transpose(norm_aug_test_features)))
    noise_term =
np.exp(np.dot(np.transpose(Beta_noise),np.transpose(norm_aug_test_features)))

    posterior_probability = np.zeros((num_test_examples, 3))
    posterior_probability[:,0] = np.divide(noise_term, IFOF_term+UNC_term+noise_term)
    posterior_probability[:,1] = np.divide(IFOF_term, IFOF_term+UNC_term+noise_term)
    posterior_probability[:,2] = np.divide(UNC_term, IFOF_term+UNC_term+noise_term)

    test_predictions = np.argmax(posterior_probability, axis = 1)

# Logistic Regression with gradient descent
if(mdl == 'LOGISTIC'):
    theta_IFOF = np.zeros((num_features+1,1))
    theta_UNC = np.zeros((num_features+1,1))
    theta_noise = np.zeros((num_features+1,1))
    stepsize = .1
    count = 0
    for k in xrange(num_examples):
        i = randint(0,num_examples-1)

        eta_IFOF_i = np.dot(np.transpose(theta_IFOF), norm_aug_training_features[i,:])
        eta_IFOF_i = eta_IFOF_i[0]
        eta_UNC_i = np.dot(np.transpose(theta_UNC), norm_aug_training_features[i,:])
        eta_UNC_i = eta_UNC_i[0]
        eta_noise_i = np.dot(np.transpose(theta_noise), norm_aug_training_features[i,:])
        eta_noise_i = eta_noise_i[0]

        mu_IFOF_i =
np.exp(eta_IFOF_i)/(np.exp(eta_IFOF_i)+np.exp(eta_UNC_i)+np.exp(eta_noise_i))
        mu_UNC_i =
np.exp(eta_UNC_i)/(np.exp(eta_IFOF_i)+np.exp(eta_UNC_i)+np.exp(eta_noise_i))
        mu_noise_i =
np.exp(eta_noise_i)/(np.exp(eta_IFOF_i)+np.exp(eta_UNC_i)+np.exp(eta_noise_i))
        print(mu_IFOF_i)

        if(isnan(mu_IFOF_i) or isnan(mu_UNC_i) or isnan(mu_noise_i)):
            count = count+1
            print(k)

        else:
            noise_label_i = int(training_labels[i] == 0)
            IFOF_label_i = int(training_labels[i] == 1)
            UNC_label_i = int(training_labels[i] == 2)
```

```
            step_IFOF_i = (IFOF_label_i-mu_IFOF_i)*norm_aug_training_features[i,:]
            step_IFOF_i = np.reshape(step_IFOF_i, (num_features+1,1))
            step_UNC_i = (UNC_label_i-mu_UNC_i)*norm_aug_training_features[i,:]
            step_UNC_i = np.reshape(step_UNC_i, (num_features+1,1))
            step_noise_i = (noise_label_i-mu_noise_i)*norm_aug_training_features[i,:]
            step_noise_i = np.reshape(step_noise_i, (num_features+1,1))

            theta_IFOF = theta_IFOF + stepsize*step_IFOF_i
            theta_UNC = theta_UNC + stepsize*step_UNC_i
            theta_noise = theta_noise + stepsize*step_noise_i

    theta = np.hstack((theta_noise, theta_IFOF, theta_UNC))
    posterior_probability = np.dot(norm_aug_test_features, theta)
    test_predictions = np.argmax(posterior_probability, axis = 1)

if(mdl == 'NAIVE_BAYES'):

    noise_features = training_feature_vectors[training_labels == 0]
    IFOF_features = training_feature_vectors[training_labels == 1]
    UNC_features = training_feature_vectors[training_labels == 2]

    prior_IFOF = len(IFOF_features)/float(num_examples)
    prior_UNC = len(UNC_features)/float(num_examples)
    prior_noise = len(noise_features)/float(num_examples)

    # Find mean and sigma squared for each of 3 classes
    noise_mean = np.reshape(np.mean(noise_features, axis = 0), (1, num_features))
    IFOF_mean = np.reshape(np.mean(IFOF_features, axis = 0), (1, num_features))
    UNC_mean = np.reshape(np.mean(UNC_features, axis = 0), (1,num_features))

    noise_sigma_squared = np.reshape(np.var(noise_features, axis = 0), (1, num_features))
    IFOF_sigma_squared = np.reshape(np.var(IFOF_features, axis = 0), (1, num_features))
    UNC_sigma_squared = np.reshape(np.var(UNC_features, axis = 0), (1, num_features))

    pi = 3.14159

    # Calculate likelihoods
    noise_exponent = -np.divide(np.square(test_feature_vectors -
noise_mean),2*noise_sigma_squared)
    noise_multiplier = 1/(np.sqrt(2*pi*noise_sigma_squared))
    noise_likelihood = np.multiply(noise_multiplier, np.exp(noise_exponent))

    IFOF_exponent = -np.divide(np.square(test_feature_vectors -
IFOF_mean),2*IFOF_sigma_squared)
    IFOF_multiplier = 1/(np.sqrt(2*pi*IFOF_sigma_squared))
    IFOF_likelihood = np.multiply(IFOF_multiplier, np.exp(IFOF_exponent))

    UNC_exponent = -np.divide(np.square(test_feature_vectors -
UNC_mean),2*UNC_sigma_squared)
    UNC_multiplier = 1/(np.sqrt(2*pi*UNC_sigma_squared))
    UNC_likelihood = np.multiply(UNC_multiplier, np.exp(UNC_exponent))

    # Calculate posterior probabilities
    noise_posterior_probability = prior_noise*np.prod(noise_likelihood,axis=1)
    IFOF_posterior_probability = prior_IFOF*np.prod(IFOF_likelihood,axis=1)
    UNC_posterior_probability = prior_UNC*np.prod(UNC_likelihood,axis=1)

    posterior_probability = np.transpose(np.vstack((noise_posterior_probability,
IFOF_posterior_probability, UNC_posterior_probability)))
    test_predictions = np.argmax(posterior_probability, axis = 1)

# Check Accuracy
number_correct = np.sum(test_predictions == test_labels)
total_number = test_labels.size
accuracy = float(number_correct)/float(total_number)
print('Total Number of Streamlines ='+str(total_number))
print('Number of Correctly Classified Streamlines ='+str(number_correct))
print('Overall Accuracy = '+str(accuracy))

number_IFOF_correct = np.sum(np.logical_and((test_labels == 1),(test_predictions == 1)))
number_IFOF_total = np.sum(test_labels == 1)
if(number_IFOF_total == 0):
```

```
    IFOF_accuracy = 0
else:
    IFOF_accuracy = float(number_IFOF_correct)/float(number_IFOF_total)
print('Total Number of IFOF Streamlines ='+str(number_IFOF_total))
print('Number of Correctly Classified IFOF Streamlines ='+str(number_IFOF_correct))
print('IFOF Accuracy ='+str(IFOF_accuracy))

number_UNC_correct = np.sum(np.logical_and((test_labels == 2),(test_predictions == 2)))
number_UNC_total = np.sum(test_labels == 2)
if(number_UNC_total == 0):
    UNC_accuracy = 0
else:
    UNC_accuracy = float(number_UNC_correct)/float(number_UNC_total)
print('Total Number of UNC Streamlines ='+str(number_UNC_total))
print('Number of Correctly Classified UNCStreamlines ='+str(number_UNC_correct))
print('UNC Accuracy ='+str(UNC_accuracy))
number_noise_correct = np.sum(np.logical_and((test_labels == 0),(test_predictions == 0)))
number_noise_total = np.sum(test_labels == 0)
if(number_noise_total == 0):
    noise_accuracy = 0
else:
    noise_accuracy = float(number_noise_correct)/float(number_noise_total)
print('Total Number of Noise Streamlines ='+str(number_noise_total))
print('Number of Correctly Classified Noise Streamlines ='+str(number_noise_correct))
print('Noise Accuracy ='+str(noise_accuracy))

print('Sensitivity (# IFOF,Uncinate Identified correctly/Total # IFOF,Uncinate) =')
print((float(number_IFOF_correct)+float(number_UNC_correct))/(float(number_IFOF_total)+fl
oat(number_UNC_total)))
print('Specificity (# Noise Identified correctly/Total # Noise) =')
print(float(number_noise_correct)/float(number_noise_total))

# Write Output Tracks Found To File
label_status = True
test_set = '/data/henry6/track_reliab_project/controls/cs280_scripts/test_set'
(test_streamlines, test_label) = streamline_labels(test_set, label_status)
trk,hdr =
nib.trackvis.read('/data/henry6/track_reliab_project/controls/cs280_scripts/training_set/
JA/Kesshi/good_IFOF_L_K.trk')

correct_IFOFbool = [np.logical_and([test_predictions == 1],[test_labels == 1])]
correct_IFOFind = (np.nonzero(correct_IFOFbool))[2]
correct_IFOF = []
for item in correct_IFOFind:
    correct_IFOF.append(test_streamlines[item])

incorrect_IFOFbool = [np.logical_and([test_predictions != 1],[test_labels == 1])]
incorrect_IFOFind = (np.nonzero(incorrect_IFOFbool))[2]
incorrect_IFOF = []
for item in incorrect_IFOFind:
    incorrect_IFOF.append(test_streamlines[item])

correct_UNCbool = [np.logical_and([test_predictions == 2],[test_labels == 2])]
correct_UNCind = (np.nonzero(correct_UNCbool))[2]
correct_UNC = []
for item in correct_UNCind:
    correct_UNC.append(test_streamlines[item])

incorrect_UNCbool = [np.logical_and([test_predictions != 2],[test_labels == 2])]
incorrect_UNCind = (np.nonzero(incorrect_UNCbool))[2]
incorrect_UNC = []
for item in incorrect_UNCind:
    incorrect_UNC.append(test_streamlines[item])

noiseas_IFOFbool = [np.logical_and([test_predictions == 1],[test_labels == 0])]
noiseas_IFOFind = (np.nonzero(noiseas_IFOFbool))[2]
noiseas_IFOF = []
for item in noiseas_IFOFind:
    noiseas_IFOF.append(test_streamlines[item])

noiseas_UNCbool = [np.logical_and([test_predictions == 2],[test_labels == 0])]
noiseas_UNCind = (np.nonzero(noiseas_UNCbool))[2]
noiseas_UNC = []
```

```
for item in noiseas_UNCind:
    noiseas_UNC.append(test_streamlines[item])

correct_noisebool = [np.logical_and([test_predictions == 0],[test_labels == 0])]
correct_noiseind = (np.nonzero(correct_noisebool))[2]
correct_noise = []
for item in correct_noiseind:
    correct_noise.append(test_streamlines[item])

correct_noise = ((item,None, None) for item in correct_noise)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/correct_noise.trk',correct_noise,hdr)

correct_IFOF = ((item,None, None) for item in correct_IFOF)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/correct_IFOF.trk',correct_IFOF,hdr)

incorrect_IFOF = ((item,None, None) for item in incorrect_IFOF)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/incorrect_IFOF.trk',incorrect_IFOF,hdr)

correct_UNC = ((item,None, None) for item in correct_UNC)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/correct_UNC.trk',correct_UNC,hdr)

incorrect_UNC = ((item,None, None) for item in incorrect_UNC)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/incorrect_UNC.trk',incorrect_UNC,hdr)

noiseas_IFOF = ((item,None, None) for item in noiseas_IFOF)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/noiseas_IFOF.trk',noiseas_IFOF,hdr)

noiseas_UNC = ((item,None, None) for item in noiseas_UNC)
nib.trackvis.write('/data/henry6/track_reliab_project/controls/cs280_scripts/esha_cs281a_
project/'+mdl+'/noiseas_UNC.trk',noiseas_UNC,hdr)
```