Final Project: Alzheimer's Prediction & Adversarial Models

Ash

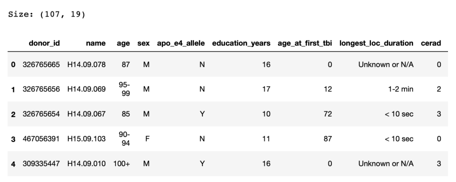
2019/01/15

1 Alzheimer's Disease: Data

Alzheimer's Disease (AD) is a neurological disorder that creates behavioral changes in memory and cognitive functions, most commonly dementia, social-withdrawal and language problems. The disease develops over time as the patient ages and eventually leads to loss of bodily functions, ultimately death, with a life expectancy of only 3-9 years following a positive diagnosis. Furthermore, AD's mechanism has not been fully understood and no effective cure has been developed even though several hypothesis has been proposed. Most recent researches agree that the biochemistry hallmark of AD is the present of amyloid beta placques (a type of peptide) and tau tangles (a type of protein) in the brain's cerebrospinal fluid (Ballard, C. et al., 2011). So, this project aims to analyze biochemistry and behavioral data from the Allen Institute of postmortem samples from aged individuals to explore potential good proxies for diagnosing AD.

1.1 Data Description

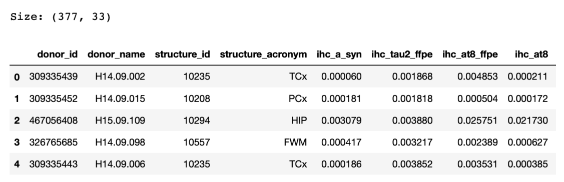
Two datasets from the Allen Institute will be combined and analyzed. The first dataset ("diagnosis.csv") has 107 datapoints, one for each donor, containing information about basics behavioral measures and demographics of the individual ranging from age, sex and race to years of education. All donors are of 70 years old or above, so the conclusions



```
Size: (107, 19)
```

	donor_id	name	age	sex	apo_e4_allele	education_years	age_at_first_tbi	longest_loc_duration	cerad
0	326765665	H14.09.078	87	M	N	16	0	Unknown or N/A	0
1	326765656	H14.09.069	95-99	M	N	17	12	1-2 min	2
2	326765654	H14.09.067	85	M	Y	10	72	< 10 sec	3
3	467056391	H15.09.103	90-94	F	N	11	87	< 10 sec	0
4	309335447	H14.09.010	100+	M	Y	16	0	Unknown or N/A	3

(a) "diagnosis.csv": Behavioral Measures



```
Size: (377, 33)
```

	donor_id	donor_name	structure_id	structure_acronym	ihc_a_syn	ihc_tau2_ffpe	ihc_at8_ffpe	ihc_at8
0	309335439	H14.09.002	10235	TCx	0.000060	0.001868	0.004853	0.000211
1	309335452	H14.09.015	10208	PCx	0.000181	0.001818	0.000504	0.000172
2	467056408	H15.09.109	10294	HIP	0.003079	0.003880	0.025751	0.021730
3	326765685	H14.09.098	10557	FWM	0.000417	0.003217	0.002389	0.000627
4	309335443	H14.09.006	10235	TCx	0.000186	0.003852	0.003531	0.000385

(b) "chemical.csv": Biochemistry Measures

Figure 1: Examples Of Data

of the subsequent analysis can only be generalized into this age group (and thus cannot be used to predict early onsets of AD). A sample of the data is included in Figure 1a.

The second dataset (”chemical.csv”) includes samples from four regions of the brain: the hippocampus (mainly attributed to memory encoding - coded as HIP); the frontal cortex (executive functions - coded as FWM); the parietal cortex (sensory information integration & language processing - coded as PCx) and the temporal cortex (memory encoding & language comprehension - coded as TCx). Each sample contains information ranging from the amount of amyloid beta and tau tangle to arachidonic acid concentration (considered a reliable biomarker for stress). A sample of the data is included in Figure 1b.

1.2 Cleaning

First, we match the donor id between the two csv files to have a full dataset. Then some variables are dropped from analysis:
- `donor_id` and `donor_name`: dropped because they do not affect our analysis. They are just random sequence assigned to each donor for anonymizing purposes.
- `structure_id`: dropped because we use structure acronym for more clarity.
- `age`: as reasoned, only aged donors were presented in this dataset, and the age range were estimated in some samples rather than accurately obtained. Thus, with no intention of using age as a proxy of detecting the disease (AD usually diagnosed for individuals over 65, and we won't use the subsequent analysis to predict early onset AD), we dropped this variable .
- `act_demented`: dropped because acting demented is a diagnosis criteria for the DSM-IV diagnosis manual for AD, so it will serve as a very good proxy for diagnosis of the disease and we use this diagnosis as the target label. Thus, if we include this, it will dominate the results and we won't have good predictors of AD.

Some variables have missing values, both categorical and numerical. So we use scikit-learn MinMaxScaler to scale numerical values to the range of 0 to 1, and binary encoders (OneHot scheme) to encode categorical data. Median are used to substituted for missing numerical data, assuming that the individual in the same age group has similar values of biochemistry stains. As for the labels, we use both the DSM-IV diagnosis, focusing on the binary classification of having AD or not, and the ADRDA diagnosis, focusing on the stage of AD (ADRDA diagnosis has 4 stages: not Alzheimer's, probable Alzheimer's disease, possible Alzheimer's disease and Alzheimer's). The DSM-IV diagnosis will be coded as binary (1 for positive AD diagnosis) while ADRDA diagnosis will be coded as numerical (from 1 to 4, increasing order of certainty for AD).

```python
#Loading relevant libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn

#Loading raw data
raw_1 = pd.read_csv('/Users/ash/Downloads/cs156_final/chemical.csv')
raw_2 = pd.read_csv('/Users/ash/Downloads/cs156_final/diagnosis.csv')

#Function to match donor ID between the two csv files
def match_donor_id(chemical, diagnosis):
    diag_column = diagnosis.columns
    for _ in diag_column[2:]:
        chemical[_] = np.nan
    dumb = len(diag_column) - 2
    count = 0
    for i in range(chemical.shape[0]):
        if chemical['donor_id'].iloc[i] == diagnosis['donor_id'].iloc[count]:
            chemical.iloc[i,-dumb:] = diagnosis.iloc[count,2:]
        elif chemical['donor_id'].iloc[i] != diagnosis['donor_id'].iloc[count]:
            count += 1
            chemical.iloc[i,-dumb:] = diagnosis.iloc[count,2:]
        else:
            print('Error!')
    return chemical

#First sort the files, merge sort seems to be the most stable
raw_1 = raw_1.sort_values(by=['donor_id'], kind='mergesort')
raw_2 = raw_2.sort_values(by=['donor_id'], kind='mergesort')
#Matching donor ID
data = match_donor_id(raw_1, raw_2)

#Dropping unrelevant variables
data = data.drop(['donor_id', 'donor_name', 'structure_id', 'age', 'act_demented'], axis=1)

#Separating data into brain regions for analysis (since 1 person probably only has 1 frontal cortex)
data_HIP = data.loc[data['structure_acronym'] == 'HIP']
data_HIP = data_HIP.drop(['isoprostane_pg_per_mg'], axis=1) #This variable was not recorded in this region
data_FC = data.loc[data['structure_acronym'] == 'FWM']
data_FC = data_FC.drop(['isoprostane_pg_per_mg'], axis=1) #This variable was not recorded in this region
data_PC = data.loc[data['structure_acronym'] == 'PCx']
data_TC = data.loc[data['structure_acronym'] == 'TCx']

#Preprocessing data
from sklearn.preprocessing import MinMaxScaler, LabelEncoder

#Function to impute missing data
def imputer_float(data):
    for _ in data.columns:
        if data[_].dtype == object:
                pass
        #Replacing missing numericals with medians
        elif data[_].dtype == float:
            data[_] = data[_].replace(np.nan, data[_].median())
        else:
            print('Error!')
#      print(data.isnull().sum())
    return data

#Function to preprocess data completely, creating a training X and a label
    y
def preprocessing(data):
    #Dropping categorical missing data
    data = data.dropna(axis=0)
    #Creating data labels
    encoder = LabelEncoder()
    y_1 = pd.get_dummies(data['dsm_iv_clinical_diagnosis'])
    y_1 = y_1.iloc[:,0].reset_index(drop=True)
    y_2 = data['nincds_arda_diagnosis']
    all_dumb = data['dsm_iv_clinical_diagnosis']
    data = data.drop(['dsm_iv_clinical_diagnosis', 'nincds_arda_diagnosis'
    ], axis=1)

    #Scaling and OneHotEncoding data
    object_col = []
    for _ in data.columns:
        if data[_].dtype == object:
            object_col.append(_)
            dumb = pd.get_dummies(data[_], prefix='['+_+']')
            all_dumb = pd.concat((all_dumb, dumb), axis=1)
        elif data[_].dtype == float:
            scaler = MinMaxScaler()
            data[_] = scaler.fit_transform(np.array(data[_]).reshape(-1,1))
        else:
            print('Error!')
    all_dumb = all_dumb.drop(['dsm_iv_clinical_diagnosis'], axis=1)
    data = data.drop(object_col, axis=1)
    X = pd.concat((data, all_dumb), axis=1)
    print('Preprocessing completed')
    return X.reset_index(drop=True), {'dsm': y_1, 'arda':y_2}

#Preprocess data for the hippocampus
data_HIP = imputer_float(data_HIP)
X_hip, y_hip = preprocessing(data_HIP)
#Preprocess data for the parietal cortex
data_PC = imputer_float(data_PC)
X_par, y_par = preprocessing(data_PC)
#Preprocess data for the frontal cortex
data_FC = imputer_float(data_FC)
X_fro, y_fro = preprocessing(data_FC)
#Preprocess data for the temporal cortex
data_TC = imputer_float(data_TC)
X_tem, y_tem = preprocessing(data_TC)
```

After preprocessing, there are around 90 datapoints for each region ready for analysis. The data is small due to the nature of the data collection process: the data were collected postmortem, by willing donor at old age. Admittedly, the small size of dataset will pose challenges to generalization claims, but we will try to address this by quantifying uncertainty in the second model. We will also attempt to generate more data by an adversarial model in section 3.

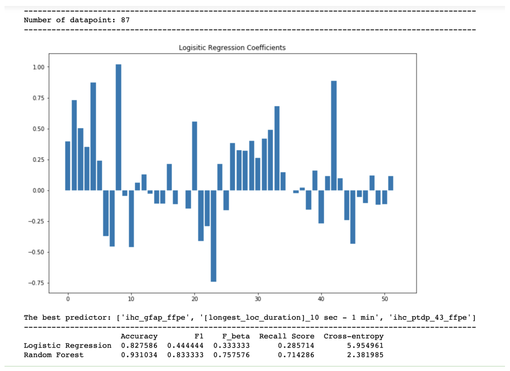
2 Alzheimer's Disease: Classification Models & Uncertainty Estimation

2.1 Classification Model

We will use two main classification models: logistic regression and random forest. Logistic regression was selected because its ability to quantify uncertainty by prediction of probability values rather than absolute categorical classification, which makes sense for predicting a condition: all biochemical and behavioral predictors determine only likelihood of a positive diagnosis (or they serve only as "risk factors"), and a positive diagnosis is made by a human physician judging all risk factors and other neuropsychological tests altogether. Thus, it makes sense to build a model that presents predictions that can be interpreted as "chances of being diagnosed with AD", rather than "whether you will be diagnosed with AD or not". In addition, logistic regression provides a good quantification of contribution for each predictor by its coefficient, thus providing a highly interpretable model. In this case we made no assumptions about the underlying data generating mechanism: logistic regression approximates a maximum likelihood estimation of the decision boundary in high-dimension without any assumptions on the underlying distributions of the variables (in contrary to, say, Linear Discriminant Analysis where the goal is to find the means and covariance matrix of the assumed underlying Gaussians distribution the data follows). Admittedly, logistic regression without any re-parameterization of the variables assuming no interaction between the variables, which is quite unlikely here because the chemical interactions between different substances in the brain. However, we do not possess enough domain-specific knowledge to perform feature engineering (and thus creating interaction terms between current variables) nor do we wish to perform an automatic re-parameterization using SVMs (because scikit-learn implementation uses liblinear, and the amount of variables here can cause performance problem) or neural networks (because we do not have enough data). Another problem arises with logistic regression: overfitting, which will be combated by using and l-2 regularization terms, which also fosters small coefficient values and some feature selection (since the ridge penalize non-zero coefficients).

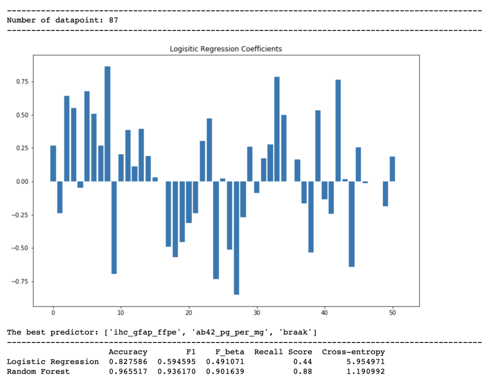
The second classification model, random forest, is chosen because its contrast with logistic regression by being a non-parametric model. It can detect highly non-linear patterns in the data with decision trees, but its can potentially has a more biased estimate because of the repeated sampling of samples from the training set to train individual trees in the forest. However, since we have a small dataset here, random forest allows us to trade off some bias for a better reusability of the data: using this bagging methods, data is repeatedly and randomly sampled with replacement from the small dataset we have to build individual trees before aggregate to a forest to make decisions. This ensemble method is suitable for the small amount of data we have, and also reduces more variance compared to simply bagging logistic regression models (to build a continuous ensemble model) because only a subset of variable are used to train individual trees, making the forest a little bit more resilience to variance.

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import fbeta_score, accuracy_score, roc_auc_score,
    recall_score, f1_score, log_loss

#Function to return most influence variables, from a logistic regression
    model
def most_influence(coefs, names, num=3):
    coefs = np.abs(coefs)
    names = list(names)
    max_coef = []
    #Returning (default) 3 variables with largest coefficients
    for _ in range(num):
        index = np.argmax(coefs)
        max_coef.append(names[index])
        coefs = np.delete(coefs, index)
        names.pop(index)
    return max_coef

#Visualizing results and comparisons between logistic regression and random
     forest
def compare_LG_RF(X_, y_):
    #Using LogisticRegression
    print('------------------------------------------------------------------'
    )
    print('Number of datapoint:', y_['dsm'].shape[0])
    logreg = LogisticRegression(penalty='l2', solver='lbfgs')
    y = y_['dsm']
    logreg.fit(X_, y)
    preds = logreg.predict(X_)
    LG = [accuracy_score(y, preds), f1_score(y, preds), fbeta_score(y,
    preds, beta=2),\
          recall_score(y, preds), log_loss(y, preds)]
    #Using RandomForest, we restricted the maximum depth of the tree to
    decrease chance of overfitting
    forest = RandomForestClassifier(n_estimators=20, criterion='gini',
    max_depth=5)
    forest.fit(X_, y)
    preds = forest.predict(X_)
    RF = [accuracy_score(y, preds), f1_score(y, preds), fbeta_score(y,
    preds, beta=2),\
```



```
------------------------------------------------------------------------------------------
Number of datapoint: 87
------------------------------------------------------------------------------------------
The best predictor: ['ihc_gfap_ffpe', '[longest_loc_duration]_10 sec - 1 min', 'ihc_ptdp_43_ffpe']
------------------------------------------------------------------------------------------
                     Accuracy        F1    F_beta  Recall Score  Cross-entropy
Logistic Regression  0.827586  0.444444  0.333333      0.285714       5.954961
Random Forest        0.931034  0.833333  0.757576      0.714286       2.381985
```

(a) Hippocampus



```
------------------------------------------------------------------------------------------
Number of datapoint: 87
------------------------------------------------------------------------------------------
The best predictor: ['ihc_gfap_ffpe', 'ab42_pg_per_mg', 'braak']
------------------------------------------------------------------------------------------
                     Accuracy        F1    F_beta  Recall Score  Cross-entropy
Logistic Regression  0.827586  0.594595  0.491071          0.44       5.954971
Random Forest        0.965517  0.936170  0.901639          0.88       1.190992
```

(b) Frontal Cortex

Figure 2: Comparison Between Logistic Regression & Random Forest

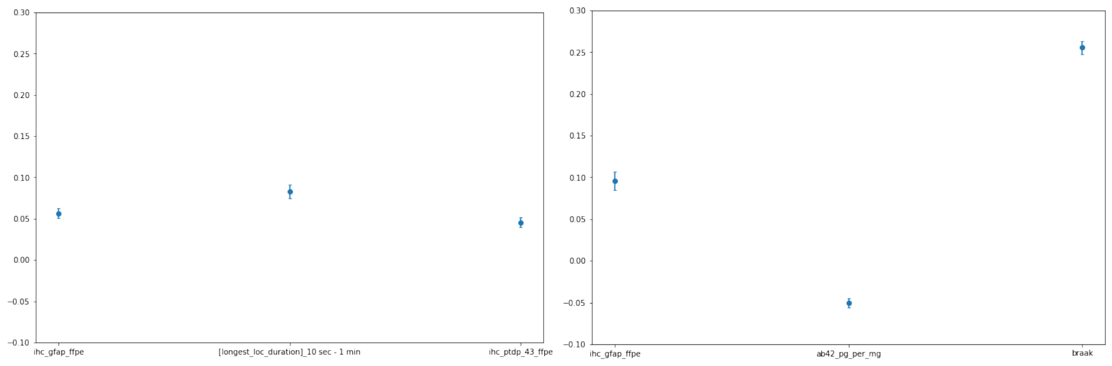
```
            recall_score(y, preds), log_loss(y, preds)]
    df = pd.DataFrame(np.array([LG, RF]), columns=['Accuracy', 'F1', 'F_beta', 'Recall Score', 'Cross-entropy'],\
                      index=['Logistic Regression', 'Random Forest'])
    plt.figure(figsize=(12,8))
    print('------------------------------------------------------------------------------------------')
    plt.bar([_ for _ in range(len(logreg.coef_[0]))],logreg.coef_[0])
    plt.title('Logisitic Regression Coefficients')
    plt.show()
    print('The best predictor:', most_influence(logreg.coef_[0],X_.columns))
    print('------------------------------------------------------------------------------------------')
    print(df)
    return most_influence(logreg.coef_[0],X_.columns)

best_hip = compare_LG_RF(X_hip, y_hip)
best_par = compare_LG_RF(X_par, y_par)

-->Output:

Figure 2
```

In Figure 2 the results are shown for the hippocampus and the frontal cortex (full code and results are in the Jupyter file). The coefficients showed interesting similarities, which make senses since tangles and placques are generally found all over the brains of aged AD patients. There are two interesting points worthy of discussion. Firsts, even though the dominant feature in the decision for both regions is `ihc_gfap_ffpe` (Glial Fibrillary Acidic Protein, a byproduct of amyloid beta placques), there are differences in the second and third dominant predictors. For the hippocampus, the unconscious duration `[longest_loc_duration]_10 sec - 1 min` and phospho-TDP-43 `ihc_ptdp_43_ffpe` (which is a reliable predictor of dementia), while for the frontal cortex it's the amyloid beta concentration directly `ab42_pg_per_mg` and stage of tau tangles `braak`. This is interesting, because it reveals results consistent with a subtype classification of AD that is being pushed to be included in DSV-V: hippocampal sparing, which has more tau tangles in the frontal areas and behaviorally related to more loss of executive control; and limbic predominant, which has more pathology in the hippocampal region, and behaviorally related to amnestic and forgetfulness. The coefficent results point out interesting relation between the subtypes, and probably can be used as preliminary evidences supporting the pushing of this subtype classification since the biochemical markers are found by the model to be quite different between the frontal cortex and the hippocampus, corresponding to the regions' functions. Second, in both model, the metric comparisons indicate at first glance similar accuracies between logistic regression (LG) and random forest (RF), but upon closer inspection we can see that LG has terrible F1 and F-beta (which weights recall more) compared to RF. And in the case of AD, we value recall more since we want to make sure we detect all instances of possible positive diagnosis and can afford some loss in precision. When we compare the raw recall scores, we can clearly see that LG is performing worse here, maybe because the relationship between the variables is non-linear and thus RF detected that better. The cross-entropy score also reflects this, since it's significantly lower for RF.

2.2 Uncertainty Model

Since we have a small dataset, it's desirable to quantify uncertainty in our MLE estimation of the coefficients. Using the stages of AD diagnosis of ADRDA, we can perform Bayesian regression with l2 regularization on the most influential variables extracted from the logistic model. With this model, it's possible to quantify uncertainty in the effect of these variables toward a spectrum of AD diagnosis (ADRDA). One important assumption of this model is that the ADRDA's criteria to diagnose AD is on a continuous rating rather than ordinal but discrete scale. Theoretically, this can be justified by arguing that going from "Not AD" to "Possible AD" to "Probable AD" can exist as a spectrum given that the certainty in the ADRDA diagnosis of a particular physician can form a continuous spectrum. However, in reality, the ADRDA's diagnosis is quite discrte in the sense that physicians check a list of symptoms that are linked to AD upon inspecting the patient and give a rating to the degree of which they believe no cause other than AD can be responsible for the checked symptoms.

```
#Function to encode ADRDA stage diagnosis on a numerical scale
def stage_of_AD(arda):
    stage = []
    #Changing the stage from ordinal category between 1-4 to numerical
    values between 0-1
    #is equivalent to scaling the output (which is strictly not neccessary,
     given that
    #the coefficients are compared with each other by relative magnitude)
    for _ in range(arda.shape[0]):
        if arda.iloc[_] == 'No Dementia':
            stage.append(0.25)
        elif arda.iloc[_] == "Possible Alzheimer'S Disease":
            stage.append(0.50)
        elif arda.iloc[_] == "Probable Alzheimer'S Disease":
            stage.append(0.75)
        elif arda.iloc[_] == "Dementia, Type Unknown":
            stage.append(1.0)
        else:
            print('Error!')
    return np.array(stage)

#Function to perform l2 regularized Bayesian regression on the data
def bayes_on_best(X_, y_, bests, yes_best=False):
    from sklearn.linear_model import BayesianRidge
    y_bayes_hip = stage_of_AD(y_['arda'])
    if yes_best == True:
        X_ = X_[bests]
    bayes = BayesianRidge()
    bayes.fit(X_, y_bayes_hip)
    return bayes

#Function to visualize uncertainty in the Bayesian estimations of
    coefficients
def visualize_coef_uncertainty(model, bests):
    #Using the diagonal values from covariance matrix of the
    #estimated multivariate Gaussian (which stands for the coefficients)
    #we can derive the error bars for each coefficients
    errors = [np.array([model.sigma_[i,i] for i in range(len(model.coef_))
    ]),\
              np.array([model.sigma_[i,i] for i in range(len(model.coef_))
    ])]
    plt.figure(figsize=(12,8))
    plt.errorbar(x=range(len(model.coef_)), y=model.coef_ , yerr=errors ,
     fmt='o' ,capsize=2)
    plt.xticks(range(len(model.coef_)),bests)
    plt.show()

#Hippocampus
bayes_hip = bayes_on_best(X_hip, y_hip, best_hip, yes_best=True)
visualize_coef_uncertainty(bayes_hip, best_hip)

#Parietal Cortex
bayes_par = bayes_on_best(X_par, y_par, best_par, yes_best=True)
visualize_coef_uncertainty(bayes_par, best_par)

#Frontal Cortex
bayes_fro = bayes_on_best(X_fro, y_fro, best_fro, yes_best=True)
visualize_coef_uncertainty(bayes_fro, best_fro)

#Temporal cortex
bayes_tem = bayes_on_best(X_tem, y_tem, best_tem, yes_best=True)
visualize_coef_uncertainty(bayes_tem, best_tem)

--->Output:

Figure 3
```



(a) Hippocampus (b) Frontal Cortex

Figure 3: Bayesian Uncertainty Estimation For The ADRDA Stage Diagnosis

Interestingly the coefficient estimations for each region are different in Figure 3, suggesting that the ADRDA's stage diagnosis employ testing techniques that weight behavioral impacts of each region differently. For instance, ADRDA might uses more executive functions, such as antisocial behaviors, which is the frontal cortex function, as proxies of diagnosis compared to memory functions, which is the hippocampus function. This will explain the lower coefficients value for contributing to the ADRDA diagnosis of the hippocampus. Additionally, it's possible that frontal cortex behavioral measures are more easily dissociated from other conditions than the hippocampus measures (for example, it's probably easier to check whether AD or an epilepsy is responsible for loss of executive controls, compared to to check whether AD or Lewy's Body disease is responsible for loss of memory encoding capacity). Another point worth discussing is the low uncertainty quantification for the coefficients despite little data. This is probably because scikit-learn Bayesian Ridge class treats the covariance matrix (which we used to visualize the error bars) as a random variable estimated using expectation maximization. Expectation maximization involves the assumption that the latent variables are missing at random (which means the latent variables generating mechanism is independent of the observed variables). In this case, we considered the covariance as a latent variable jointly estimated with the regularization term, thus violating that assumption because it's highly likely that the latent generating mechanism is dependent upon the observed variables, and so the regularization term is overestimated and penalized if the covariance entries were too big too harshly.

3 Adversarial Model

In the research context, self-generated data is strictly speaking unusable since we wish to investigate the relationship between the variables of interest organically, not under some assumptions (that we must make in order to generate data). However, it's theoretically interesting to think about the process of generating "fake" data that is in some statistical sense very similar to the small amount of data we have at hand.

One of the state-of-the-art data generation method is generative adversarial networks (GANs), which employs two neural networks trained together with different functions. The core idea is simple: one network will act as a generator, which generates data, and the data is fed to the other network, the discriminator, which discriminates whether the data is generated or authentic. Iterations by iterations, the generator will learn to generate ever more convincing fake data to fool the discriminator, whereas the discriminator will become more sophisticated in recognizing fake data. However, there are many practical considerations in training such a joint system of networks, for example how to make sure the performance of the networks are balanced so that none becomes too good too quickly and diminish the gradient of the other network, not letting the other learn. Here obviously we cannot use GANs in its original form because we have too few datapoints, but we can attempt a similar formulation with simpler generator and discriminator.

Our system contains two components: the generator, which is a kernel density model with Gaussian kernel that can generate samples from a dataset that model the distribution of each variable in that dataset; and a discriminator, which is a logistic regression model trained to differentiate between real and fake data. Initially, we create some fake data with completely uniform distributions for every variables, and feed that to the discriminator for it to learn to differentiate between real and fake data. Next, we use kernel density modeling to create a generator, from a subset of the real data, and use that generator to generate 100 fake samples. Using the base classifier trained above, try to recognize the the 100 fake samples just generated (ideally the classifier will say all 100 are fake). If the classifier makes mistakes (i.e. incorrectly classifies some of the 100 fake samples as "real"), re-label these mistakes as fakes and add them to the original dataset used to train the classifier and train the classifier again with their mistakes. Finally, include the "real" samples to the kernel density model for the generator to know which of its samples fooled the discriminator. The process is then repeated.

```python
#Using the hippocampal data
real = X_hip

#1. generate fake data using completely random values
def generate_fake_data(real, size):
    from scipy.stats import uniform
    fake = []
    for _ in range(real.shape[1]):
        min_, max_ = min(real.iloc[:, _]), max(real.iloc[:, _])
        some_data = uniform(loc=min_, scale=max_).rvs(size=size)
        fake.append(some_data)
    return pd.DataFrame(np.array(fake).T, columns=real.columns)

fake = generate_fake_data(real, 30)

```

```python
base_clf_X = pd.concat((fake, real), axis=0).reset_index(drop=True)
base_clf_y = np.ones(fake.shape[0]+real.shape[0])
base_clf_y[:fake.shape[0]] = 0

#2. train a classifier to realize fake data
clf = LogisticRegression(penalty='l2', solver='lbfgs')
clf.fit(base_clf_X, base_clf_y)
preds = clf.predict(base_clf_X)
print('Base Accuracy:', accuracy_score(base_clf_y, preds))

#3. include 10 real data to kernel model/Gaussian Mixtures --> generate 100 fake data

def generator_density(train, size=100):
    from sklearn.neighbors import KernelDensity
    dense = KernelDensity(bandwidth=100, kernel='gaussian')
    dense.fit(train)
    samples = dense.sample(100)
    return pd.DataFrame(samples, columns=train.columns)

fake_dense = generator_density(real.iloc[:10,:])

#4. Use trained classifier in 2. to learn which one can be "real" --> put the "real" one to 3. and repeat
#5. re-parse "real" one to fake --> train classifier again
def adversarial(real, fake, base_clf_X, base_clf_y, base_clf, generator, step=20):
    gen, dis, exam = [], [], []
    for ash in range(step):
        print('------------------------------------------------------------------------------------------------')
        real = real.iloc[:10,:]
        preds = base_clf.predict(fake)
#          print('Accuracy of discriminator at step {}:'.format(ash+1), accuracy_score(np.zeros(fake.shape[0]),preds))
        index_1 = [preds[_] == 1 for _ in range(preds.shape[0])]
        print('Percent of fake by generator went unnoticed at step {}:'.format(ash+1), sum(index_1)/len(index_1))
        gen.append(sum(index_1)/len(index_1))
        dumb = fake[index_1]
        train_generator = pd.concat((real, dumb), axis=0)
        train_discriminator_X = pd.concat((base_clf_X, dumb), axis=0)
        train_discriminator_y = np.zeros(train_discriminator_X.shape[0])
        train_discriminator_y[:base_clf_y.shape[0]] = base_clf_y
        fake = generator(train_generator)
        base_clf.fit(train_discriminator_X, train_discriminator_y)
        preds = base_clf.predict(train_discriminator_X)
        print('Accuracy of discriminator at step {}:'.format(ash+1), accuracy_score(train_discriminator_y,preds))
        dis.append(accuracy_score(train_discriminator_y,preds))
        print('Number of training example for next step:', dumb.shape[0])
        exam.append(dumb.shape[0])
        base_clf_X = train_discriminator_X
        base_clf_y = train_discriminator_y
    return gen, dis, exam

gen, dis, exam = adversarial(real, fake_dense, base_clf_X, base_clf_y, clf,
     generator_density)
plt.figure(figsize=(12,8))
plt.subplot(2,1,1)
plt.plot(range(len(gen)), gen, color='r', label='Generator Success Rate')
plt.plot(range(len(gen)), dis, color='b', label='Accuracy of Discriminator'
    )
plt.legend()
plt.subplot(2,1,2)
plt.bar(range(len(gen)), exam, color='g', label='Number of Example
    Available For Next Step')
plt.legend()
plt.show()

--->Output:

Figure 4
```

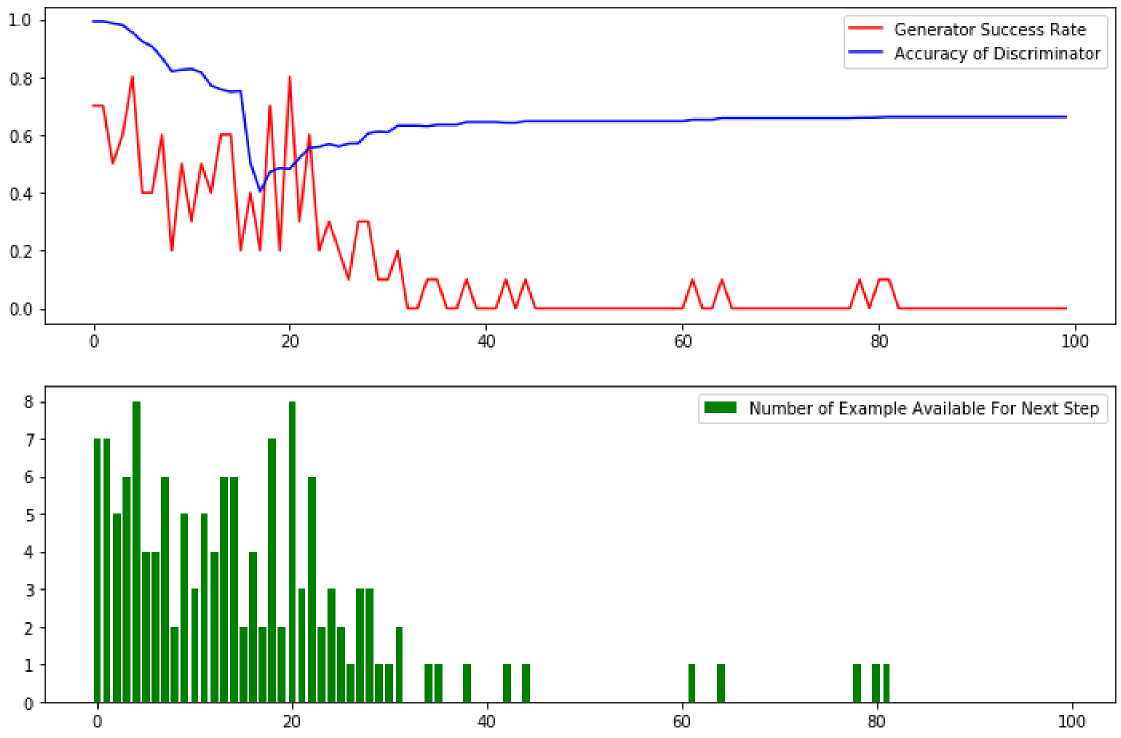


Figure 4: Adversarial Network Performance In 20 Steps

As we can see in Figure 4, the accuracy of the discriminator (how many percent of the data it realized, correctly, as fake) saturated at around step 40. At the same time the generator success rate (how many percent of the 100 samples it generated were able to fool the discriminator) also saturated at 0. This is caused by the problem of unbalanced training: looking at the figure below we can see that the number of example available for training for next step decrease over time at become 0 at step 40, which means the discriminator got so good so quickly that it did not allow enough time for the generator to learn useful information, so from step 41 on there is no training samples for any of the components since the discriminator recognized all fake instances.

References

Data source: Allen Institute - retrieved from: http://aging.brain-map.org/download/index

[1] Ballard, C., Gauthier, S., Corbett, A., Brayne, C., Aarsland, D., Jones, E. (2011). *Alzheimer's disease.* Retrieved from:
https://www.ncbi.nlm.nih.gov/pubmed/21371747

[2] Duyckaerts, C. (2011). *Disentangling Alzheimer's disease.* Retrieved from:
https://www.thelancet.com/journals/laneur/article/PIIS1474-4422(11)70171-5

[3] Kamphuis W., Middeldorp J., Kooijman L., Sluijs J. A., Kooi E. J., Moeton M., Freriks M., Mizee M. R., Hol E. M. (2014). *Glial fibrillary acidic protein isoform expression in plaque related astrogliosis in Alzheimer's disease.* Retrieved from:
https://www.ncbi.nlm.nih.gov/pubmed/24269023