The dataset is from the Many Labs Replication Project in which 13 effects were replicated across 36 samples and over 6,000 participants. Data from the replications are included, along with demographic variables about the participants and contextual information about the environment in which the replication was conducted. Data were collected in-lab and online through a standardized procedure administered via an online link. The dataset is stored on the Open Science Framework website. These data could be used to further investigate the results of the included 13 effects or to study replication and generalizability more broadly

The sample is comprised of 6,344 participants recruited from 36 different sources including university subject pools, Amazon Mechanical Turk, Project Implicit, and other sources. The aggregate sample has a mean age of 25.98. Participant ethnicity is: 65.1% White, 6.7% Black or African American, 6.5% East Asian, 4.5% South Asian, 17.2% Other or Unknown. Participant gender is 63.6% female, 29.9% male, 6.5% no response.

The Original Many Labs Project: The original Many Labs project attempted to replicate 28 psychological studies, across 60 different labs, trying to determine to what extent the originally studied effect was reproducible. Questions given to subjects touched on a diverse array of topics from nationalism, to the perceptions of numbers, to feelings about art and mathematics.

Table of Contents

Importing the Libraries

In [57]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statistics
import random
from copy import deepcopy, copy
import itertools
from collections import Counter
import math
from graphviz import Digraph, Source, Graph
import scipy
from sklearn.metrics import pairwise_distances
from scipy.stats import multivariate_normal as mvn
```

Loading the Dataset

In [3]:
```
data = pd.read_csv('/Users/adityavyas/Desk/Sem-2/Machine Learning/End
game/ML1/Tab.delimited.Cleaned.dataset.WITH.variable.labels.csv', sep
= '\t', encoding = "ISO-8859-1")
data.head()
```

```
/Users/adityavyas/anaconda/envs/py36/lib/python3.6/site-packages/IPyt
hon/core/interactiveshell.py:3020: DtypeWarning: Columns (17,55,59,6
1,65,68,69,70,83,90,91,92,93,120,121,122,123,126,140,141) have mixed
types. Specify dtype option on import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

Out[3]:

	session_id	session_date	last_update_date	session_last_update_date	referrer	creation_dat
0	2400853	8/28/2013	8/28/13 12:15	8/28/13 12:15	abington	8/28/201
1	2400856	8/28/2013	8/28/13 12:13	8/28/13 12:13	abington	8/28/201
2	2400860	8/28/2013	8/28/13 12:15	8/28/13 12:15	abington	8/28/201
3	2400868	8/28/2013	8/28/13 12:12	8/28/13 12:12	abington	8/28/201
4	2400872	8/28/2013	8/28/13 12:11	8/28/13 12:11	abington	8/28/201

5 rows × 382 columns

Identify Column Classes

In [4]:
```
mixed_columns = ['flagsupplement1', 'flagsupplement2', 'flagsupplement3', 'iatexplicitart1', 'iatexplicitart2',
                'iatexplicitart3', 'iatexplicitart4', 'iatexplicitart5', 'iatexplicitart6', 'iatexplicitmath1',
                'iatexplicitmath2', 'iatexplicitmath3', 'iatexplicitmath4', 'iatexplicitmath5', 'iatexplicitmath6',
                'sysjust1', 'sysjust2', 'sysjust3', 'sysjust4', 'sysjust5', 'sysjust6', 'sysjust7', 'sysjust8',
                'mturk.non.US', 'exprace']

all_NAN_columns = ['task_status', 'task_sequence', 'beginlocaltime']

numeric_columns = ['anchoring3ameter', 'anchoring3bmeter', 'anchoring1akm', 'anchoring1bkm',
                    'gamblerfallacya_sd', 'gamblerfallacyb_sd', 'numparticipants_actual', 'IATfilter',
                    'numparticipants', 'age', 'anchoring1a', 'mturk.total.mini.exps', 'anchoring1b', 'anchoring2a',
                    'anchoring2b', 'anchoring3a', 'anchoring3b', 'anchoring4a', 'anchoring4b', 'd_donotuse',
                    'gamblerfallacya', 'gamblerfallacyb', 'omdimc3rt', 'omdimc3trt', 'order', 'meanlatency',
                    'meanerror', 'block2_meanerror', 'block3_meanerror', 'block5_meanerror', 'block6_meanerror',
                    'lat11', 'lat12', 'lat21', 'lat22', 'sd1', 'sd2', 'd_art1', 'd_art2', 'd_art',
                    'anchoring1', 'anchoring2', 'anchoring3', 'anchoring4', 'Ranchori', 'RAN001',
                    'RAN002', 'RAN003', 'Ranch1', 'Ranch2', 'Ranch3', 'Ranch4', 'gambfalDV',
                    'reciprocityother', 'flagdv', 'Sysjust', 'Imagineddv', 'IATexpart', 'IATexpmath', 'IATexp.overall',
                    'IATEXPfilter']

date_columns = [col for col in data.columns if '_date' in col]

exclude_from_data = [col for col in data.columns if '_url.' in col]

categorical = ["mturk.duplicate", "mturk.exclude.null", "mturk.keep", "mturk.exclude", "totexpmissed", "artwarm",
               "ethnicity", "imaginedexplicit1", "imaginedexplicit2", "imaginedexplicit3", "imaginedexplicit4",
               "major", "mathwarm", "quotea", "quoteb", "sunkcosta", "sunkcostb", "sunkDV", "scalesreca",
               "scalesrecb", "quotearec", "quotebrec", "quote", "totalflagestimations", "totalnoflagtimeestimations",
               "moneyfilter", 'flagdv1', 'flagdv2', 'flagdv3', 'flagdv4', 'flagdv5', 'flagdv6', 'flagdv7', 'flagdv8',
               'priorexposure3', 'priorexposure4', 'priorexposure5', 'priorexposure6', 'priorexposure7',
               'priorexposure9', 'priorexposure10', 'priorexposure11', 'priorexposure12', 'priorexposure13',
               'priorexposure1', 'priorexposure2', 'priorexposure8', 'scalesorder', 'reciprocorder', 'diseaseforder',
               'quoteorder', 'flagprimorder', 'sunkcostorder', 'anchorinorder', 'allowedforder', 'gamblerforder',
```

```
                'moneypriorder', 'imaginedorder', 'iatorder']

to_remove = ["moneyagea", "moneyageb", "moneyethnicitya", "moneyethni
cityb", "moneygendera", "moneygenderb", "text",
             "session_id", "user_id", "previous_session_id", 'expcomm
ents', "study_name", "study_url"]

# Can do feature engineering in nlp_feature
nlp_feature = ['feedback', "imagineddescribe"]
non_string_columns = mixed_columns + all_NAN_columns + numeric_column
s + date_columns + exclude_from_data + nlp_feature + to_remove
string_columns = [col for col in data.columns if col not in non_strin
g_columns]
string_columns = sorted(string_columns)
```

In [5]:
```
len(string_columns) + len(non_string_columns)
```

Out[5]: 382

whole number - age, numparticipants sort task_id columns

Convert ['', ' ' and .] to NaN

In [6]:
```
data = data.replace({'': np.nan, ' ': np.nan, '.': np.nan})
```

Convert Columns to Respective Data Types

1. Numeric Columns

In [7]:
```
for col in numeric_columns:
    data[col] = pd.to_numeric(data[col])
```

2. Date Columns

In [8]:
```
for col in date_columns:
    data[col] = pd.to_datetime(data[col])
```

3. String Columns

In [9]:
```
for col in string_columns:
    data[col] = data[col].astype(str)
```

4. Mixed Columns

Reading the paper [https://osf.io/ebmf8/ (https://osf.io/ebmf8/)](https://osf.io/ebmf8/) to clean the mixed columns

How to encode exprace?

exprace ['6' '10' 'brazilwhite' 'brazilblack' 'brazilbrown' nan 'chinese' 'malay' '8' '7' '5' '9' '2' '3' 'dutch' '1']

In [10]:
```
data["flagsupplement1"] = data["flagsupplement1"].apply(lambda x:
                                                    '11' if x == 'Ver
y much' else
                                                    ('1' if x == 'Not
at all' else x))
data["flagsupplement2"] = data["flagsupplement2"].apply(lambda x:
                                                    '1' if x == 'Demo
crat' else
                                                    ('7' if x == 'Rep
ublican' else x))
data["flagsupplement3"] = data["flagsupplement3"].apply(lambda x:
                                                    '1' if x == 'Libe
ral' else
                                                    ('7' if x == 'Con
servative' else x))

data["iatexplicitart1"] = data["iatexplicitart1"].apply(lambda x:
                                                    '1' if x == 'Very
bad' else
                                                    ('2' if x == 'Mod
erately bad' else x))
data["iatexplicitart2"] = data["iatexplicitart2"].apply(lambda x:
                                                    '1' if x == 'Very
Sad' else
                                                    ('2' if x == 'Mod
erately Sad' else x))
data["iatexplicitart3"] = data["iatexplicitart3"].apply(lambda x:
                                                    '1' if x == 'Very
Ugly' else
                                                    ('2' if x == 'Mod
erately Ugly' else x))
data["iatexplicitart4"] = data["iatexplicitart4"].apply(lambda x:
                                                    '1' if x == 'Very
Disgusting' else
                                                    ('2' if x == 'Mod
erately Disgusting' else x))
data["iatexplicitart5"] = data["iatexplicitart5"].apply(lambda x:
                                                    '1' if x == 'Very
Avoid' else
                                                    ('2' if x == 'Mod
erately Avoid' else x))
data["iatexplicitart6"] = data["iatexplicitart6"].apply(lambda x:
                                                    '1' if x == 'Very
Afraid' else
                                                    ('2' if x == 'Mod
erately Afraid' else x))

data["iatexplicitmath1"] = data["iatexplicitmath1"].apply(lambda x:
                                                    '1' if x == 'Very
bad' else
                                                    ('2' if x == 'Mod
erately bad' else
                                                    ('3' if x == 'Sli
ghtly bad' else x)))
data["iatexplicitmath2"] = data["iatexplicitmath2"].apply(lambda x:
                                                    '1' if x == 'Very
```

```
Sad' else
                                                    ('2' if x == 'Mod
erately Sad' else
                                                    ('3' if x == 'Sli
ghtly Sad' else x)))
data["iatexplicitmath3"] = data["iatexplicitmath3"].apply(lambda x:
                                                    '1' if x == 'Very
Ugly' else
                                                    ('2' if x == 'Mod
erately Ugly' else
                                                    ('3' if x == 'Sli
ghtly Ugly' else x)))
data["iatexplicitmath4"] = data["iatexplicitmath4"].apply(lambda x:
                                                    '1' if x == 'Very
Disgusting' else
                                                    ('2' if x == 'Mod
erately Disgusting' else
                                                    ('3' if x == 'Sli
ghtly Disgusting' else x)))
data["iatexplicitmath5"] = data["iatexplicitmath5"].apply(lambda x:
                                                    '1' if x == 'Very
Avoid' else
                                                    ('2' if x == 'Mod
erately Avoid' else
                                                    ('3' if x == 'Sli
ghtly Avoid' else x)))
data["iatexplicitmath6"] = data["iatexplicitmath6"].apply(lambda x:
                                                    '1' if x == 'Very
Afraid' else
                                                    ('2' if x == 'Mod
erately Afraid' else
                                                    ('3' if x == 'Sli
ghtly Afraid' else x)))

for col in mixed_columns:
    if "sysjust" in col:
        data[col] = data[col].apply(lambda x:
                                    '1' if x == 'Strongly disagree' e
lse
                                    ('7' if x == 'Strongly agree' els
e x))

data["mturk.non.US"] = data["mturk.non.US"].apply(lambda x: '1' if x
== 'non-US IP address' else x)
data["exprace"] = data["exprace"].apply(lambda x: '11' if x == 'brazi
lwhite' else
                                        ('12' if x == 'brazilblack' e
lse
                                        ('13' if x == 'brazilbrown' e
lse
                                        ('14' if x == 'chinese' else
                                        ('15' if x == 'malay' else
                                        ('16' if x == 'dutch' else x
))))))
```

In [11]:
```
for col in data.columns:
    if "mturk" in col:
        print(col, data[col].unique())
```

```
mturk.non.US [nan '0' '1']
mturk.Submitted.PaymentReq ['nan' 'yes']
mturk.total.mini.exps [nan 11. 10.  9.]
mturk.duplicate ['nan' '0' '1']
mturk.exclude.null ['nan' '0' '1']
mturk.keep ['nan' '1' '0']
mturk.exclude ['nan' '2' '99']
```

In [12]:
```
data["citizenship2"].unique()
```

Out[12]:
```
array(['nan', 'oraz norweskie', 'Polska', 'rumena', 'italiana', 'vene
ta'],
      dtype=object)
```

In [13]:
```
for col in mixed_columns:
    data[col] = data[col].astype(str)
```

In [14]:
```
print(len(string_columns), len(mixed_columns))
string_columns.extend(mixed_columns)
print(len(string_columns))
```

```
181 25
206
```

In [15]:
```
def clean_user_agent(x):
    x = x.split(" ")[1]
    x = x.replace("(", "")
    x = x.replace(";", "")
    x = x.lower()
    if "windows" in x:
        return "windows"
    if "compatible" in x:
        return "compatible"
    if "macintosh" in x:
        return "macintosh"
    if "x11" in x:
        return "x11"
    return x

data["user_agent"] = data["user_agent"].apply(lambda x: clean_user_ag
ent(x))
data["user_agent"].unique()
```

Out[15]:
```
array(['windows', 'compatible', 'macintosh', 'x11', 'linux',
       'masking-agent', 'ipad'], dtype=object)
```

In [16]:
```
def clean_exprunafter2(x):
    if "group" in x:
        return "group"
    if "past" in x:
        return "your past and your future"
    if ("thinking" or "reasoning") in x:
        return "thinking and reasoning"
    if "social" in x:
        return "understanding social situations"
    if ("emotion" or "verbal") in x:
        return "emotion and verbal working memory"
    if ("intentionality" or "inentionality" or "intentionally") in x:
        return "intentionality"
    if "a study on intentionally. takes 5 minutes to complete. read a
scenario and answer questions about the intentions of the actor." in
x:
        return "intentionality"
    if "a study on intentionally. takes 5 minutes to complete. read a
scenario and answer questions about the intentions of the actor" in x
:
        return "intentionality"
    return x

data["exprunafter2"] = data["exprunafter2"].apply(lambda x: x.lower
())
data["exprunafter2"] = data["exprunafter2"].apply(lambda x: clean_exp
runafter2(x))

data["exprunafter2"].unique()
```

Out[16]:
```
array(['nan', 'group', 'linear regression lab',
       'it was not provided to me', 'your past and your future',
       'thinking and reasoning', 'understanding social situations',
'no',
       'emotion and verbal working memory', 'verbal ospan', 'trust ga
me',
       'intentionality', 'a36', 'intentions', 'inentionality'],
      dtype=object)
```

In [17]:
```
def clean_native_lang(x):
    if "creol" in x:
        return "creole"
    if "filipino" in x:
        return "filipino"
    if "cantonese" in x:
        return "cantonese"
    if "taiwanese" in x:
        return "taiwanese"
    # Assuming asian would mean mandarin, as most popular language
    if ("asian" or "chinese" or "manderine" or "mandrain" or "madaria
n" in x) in x:
        return "mandarin"
    if "hindi" in x:
        return "hindi"
    if "spanish" in x:
        return "spanish"
    if "arabic" in x:
        return "arabic"
    if "mi'kmaq" in x:
        return "mikmaq"
    if ("na" or "-" or "not in college" or "marketing" or "fashion" o
r "communication" or "disorders" or "merchandising") in x:
        return "nan"
    if "dual citizen" in x:
        return "english"
    if "english" in x:
        return "english"
    if "serbo-croation" in x:
        return "serbian"
    return x

data["nativelang2"] = data["nativelang2"].apply(lambda x: x.lower())
data["nativelang2"] = data["nativelang2"].apply(lambda x: clean_nativ
e_lang(x))
```

In [18]:
```
all_columns = [numeric_columns, string_columns, date_columns, all_NAN
_columns]
```

There are 182 out of 382 columns which have at least 1 missing value. The above table is sorted in ascending order. Our initial thought process is to start from the columns which have low count of missing values, because those will be relatively easy to impute.

According to the Codebook,

- session_creation_date is redundant as we have create_date
- session_last_update_date is redundant as we have last_update_date

As of now, we believe we can remove columns which have all missing values, because we have no knowledge of how that column is, and what should be filled there (no training examples). This means the following are removed,

- task_sequence
- task_status
- beginlocaltime

"expcomments" has nan, still not coming in missing values. When doing a check with == np.nan, showing False ---- added check

In [19]:
```
data.drop(date_columns, inplace=True, axis=1)
data.drop(exclude_from_data, inplace=True, axis=1)
data.drop(all_NAN_columns, inplace=True, axis=1)
data.drop(to_remove, inplace=True, axis=1)
data.drop(nlp_feature, inplace=True, axis=1)
```

In [20]:
```
data.shape
```

Out[20]:
```
(6344, 267)
```

Generating Synthetic Data

Variational AutoEncoders

In [125]:
```
np.seterr(all = "warn")
class VariationalAutoencoder():

    SigmoidActivation = "sigmoid"
    ReLUActivation = "relu"
    LinearActivation = "linear"
    LeakyReLUActivation = "lrelu"

    def __init__(self,
                 learning_rate = 0.04,
                 batch_size = 32,
                 num_hidden_layers = None,
                 num_neurons_each_layer = None,
                 z_shape = 4,
                 epochs = 10):
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.epochs = epochs
        self.num_hidden_layers = num_hidden_layers
        self.num_neurons_each_layer = num_neurons_each_layer
        self.z_shape = z_shape

        self.activations_functions = {
            self.SigmoidActivation: self._sigmoid,
            self.LeakyReLUActivation: self._leaky_relu,
            self.ReLUActivation: self._relu,
            self.LinearActivation: self._linear
        }
        self.activations_derivatives = {
            self.SigmoidActivation: self._sigmoid_derivative,
            self.LeakyReLUActivation: self._leaky_relu_derivative,
            self.ReLUActivation: self._relu_derivative,
            self.LinearActivation: self._linear_derivative
        }

        # Activations for Encoder and Decoder
        self.encoder_activations = [self.LeakyReLUActivation] * self.
num_hidden_layers + [self.LinearActivation]
        self.decoder_activations = [self.ReLUActivation] * self.num_h
idden_layers + [self.SigmoidActivation]

        self.num_neurons_each_encoder_layer = self.num_neurons_each_l
ayer
        self.num_neurons_each_decoder_layer = self.num_neurons_each_l
ayer[::-1]


    def _sigmoid(self, x):
        x = np.select([x < 0, x >= 0], [np.exp(x)/(1 + np.exp(x)), 1/
(1 + np.exp(-x))])
        return x

    def _relu(self, x):
        return np.maximum(0, x)

    def _leaky_relu(self, x):
```

```
        return np.maximum(0, x)

    def _linear(self, x):
        return x

    def _sigmoid_derivative(self, x):
        return self._sigmoid(x) * (1 - self._sigmoid(x))

    def _relu_derivative(self, x):
        return (np.ones_like(x) * (x > 0))

    def _leaky_relu_derivative(self, x):
        return

    def _linear_derivative(self, x):
        return np.ones_like(x)

    def _binary_cross_entropy_loss(self, y_hat, y):
        loss = np.sum(-y * np.log(y_hat + 1e-15) - (1 - y) * np.log(1
- y_hat + 1e-15))
        return loss

    def _kl_divergence(self, mu, log_var):
        return -0.5 * np.sum(1 + log_var - np.power(mu, 2) - np.exp(l
og_var))

    def _encoder(self, X):
        encoder_out = []

        for curr_layer in self.encoder_layers:
            encoder_out.append([])

            # Get the activation for this layer and its function
            activation_for_this_layer = self.encoder_activations[curr
_layer]
            activation_function = self.activations_functions[activati
on_for_this_layer]

            if curr_layer == 0:
                previous_layer_output = X
            else:
                previous_layer_output = encoder_out[curr_layer - 1].c
opy()
                previous_layer_output = np.insert(previous_layer_outp
ut, obj = 0, values = 1, axis = 1)

            if curr_layer != self.encoder_layers[-1]:
                encoder_out[curr_layer] = activation_function(previou
s_layer_output @ self.encoder_weights[curr_layer].T)
            else:
                encoder_weights_last_layer = np.transpose(self.encode
r_weights[curr_layer], axes = (0, 2, 1))
                encoder_out[curr_layer] = activation_function(previou
s_layer_output @ encoder_weights_last_layer)

        encoder_out = np.array(encoder_out)
        mu, log_var = encoder_out[-1][0], encoder_out[-1][1]
```

```
        return mu, log_var, encoder_out

    def _decoder(self, z):

        decoder_out = []

        for curr_layer in self.decoder_layers:
            decoder_out.append([])

            # Get the activation for this layer and its function
            activation_for_this_layer = self.decoder_activations[curr
_layer]
            activation_function = self.activations_functions[activati
on_for_this_layer]

            if curr_layer == 0:
                previous_layer_output = z
            else:
                previous_layer_output = decoder_out[curr_layer - 1].c
opy()
            previous_layer_output = np.insert(previous_layer_output,
obj = 0, values = 1, axis = 1)

            decoder_out[curr_layer] = activation_function(previous_la
yer_output @ self.decoder_weights[curr_layer].T)

        xhat_batch = decoder_out[-1]
        return xhat_batch, decoder_out

    def _forward(self, X):

        # Encode
        mu, log_var, encoder_out = self._encoder(X)

        # Reparametrization trick to sample z from gaussian. First sa
mple x from standard normal distribution.
        # Then we use z = mu + sigma*x to get our latent variable.
        self.rand_sample = np.random.standard_normal(size = (self.bat
ch_size, self.z_shape))
        self.sample_z = mu + np.exp(log_var * .5) * self.rand_sample

        # Decode
        xhat_batch, decoder_out = self._decoder(self.sample_z)

        return mu, log_var, xhat_batch, encoder_out, decoder_out

    def _backward_decoder(self, y, decoder_out):

        decoder_output_derivatives = deepcopy(decoder_out)
        decoder_weight_derivatives = deepcopy(self.decoder_weights)

        # We calculate weight derivatives for each data row in the ba
tch and average the
        # derivatives at the end.
        decoder_weight_derivatives = [decoder_weight_derivatives] * s
elf.batch_size
```

```
        # Compute the output derivatives
        layers_reversed = self.decoder_layers[::-1]
        for curr_layer in layers_reversed:
            next_layer = curr_layer + 1

            # For the last layer simply use the formula
            if curr_layer == self.total_decoder_layers - 1:
                decoder_output_derivatives[curr_layer] = -y/(decoder_
out[curr_layer] + 1e-16) + \
                                                     (1 - y) * 1/(1 - 
decoder_out[curr_layer] + 1e-16)
                continue

            # Get the activation derivative function for next layer
            activation_for_next_layer = self.decoder_activations[next
_layer]
            activation_derivative = self.activations_derivatives[acti
vation_for_next_layer]

            # The next layer output derivatives
            next_layer_output_derivatives = decoder_output_derivative
s[next_layer]

            # Calculate the activation derivative. Add a 1 for the bi
as weight
            current_layer_output = decoder_out[curr_layer].copy()
            current_layer_output = np.insert(current_layer_output, ob
j = 0, values = 1, axis = 1)
            next_layer_activation_derivatives = activation_derivative
(current_layer_output @ self.decoder_weights[next_layer].T)

            # Remove the bias from the weights. Bias output derivativ
e is 1.
            next_layer_weights_without_bias = self.decoder_weights[ne
xt_layer][:, 1:]

            # Cycle through the batch of next layer activation deriva
tives
            for batch_index, next_layer_activation_derivative in enum
erate(next_layer_activation_derivatives):
                next_layer_activation_derivative = next_layer_activat
ion_derivative.reshape(-1, 1)

                # Multiply each neuron's activation derivative with i
ts weights. This is the Hadmard product
                second_term = next_layer_activation_derivative * next
_layer_weights_without_bias

                # Sum over all the neurons in the next layer to get t
he output derivative for each
                # neuron in the current layer. This is because each n
euron contributes to all the neurons
                # in the next layer.
                decoder_output_derivatives[curr_layer][batch_index] = 
next_layer_output_derivatives[batch_index] @ second_term

        # Update the weights using the output derivative calculated a
```

```
bove
        for curr_layer in layers_reversed:

            # Get the activation for this layer and its derivative fu
nction
            activation_for_this_layer = self.decoder_activations[curr
_layer]
            activation_derivative = self.activations_derivatives[acti
vation_for_this_layer]

            # If first layer then use the data as the previous layer.
            if curr_layer == 0:
                previous_layer_output = self.sample_z
            else:
                prev_layer = curr_layer - 1
                previous_layer_output = decoder_out[prev_layer].copy
()
            previous_layer_output = np.insert(previous_layer_output,
obj = 0, values = 1, axis = 1)

            # Current layer output derivatives
            curr_layer_output_derivatives = decoder_output_derivative
s[curr_layer]

            # Get current layer's activation derivatives
            curr_layer_activation_derivatives = activation_derivative
(previous_layer_output @ self.decoder_weights[curr_layer].T)
            curr_layer_activation_derivatives = curr_layer_activation
_derivatives

            # Cycle through the batch of next layer activation deriva
tives
            for batch_index, curr_layer_activation_derivative in enum
erate(curr_layer_activation_derivatives):
                curr_layer_activation_derivative = curr_layer_activat
ion_derivative.reshape(-1, 1)

                # For the current layer multiply each neuron's activa
tion derivatives with all previous layer outputs.
                curr_layer_weight_derivatives = curr_layer_output_der
ivatives[batch_index].reshape(-1, 1) * \
                                                curr_layer_activation
_derivative * previous_layer_output[batch_index]
                decoder_weight_derivatives[batch_index][curr_layer] =
curr_layer_weight_derivatives

        # Average the gradients across batch
        decoder_weight_derivatives = np.mean(decoder_weight_derivativ
es, axis = 0)

        return decoder_weight_derivatives, decoder_output_derivatives

    def _calculate_mu_derivative(self, decoder_output_derivatives):

        mu_derivatives = np.zeros((self.batch_size, self.z_shape))

        # Add a bias to z
```

```
        z_with_bias = np.insert(self.sample_z, obj = 0, values = 1, a
xis = 1)

        # Activation derivative function for the first layer of decod
er
        activation_for_decoder_first_layer = self.decoder_activations
[0]
        activation_derivative_func = self.activations_derivatives[act
ivation_for_decoder_first_layer]

        # Activation derivatives for the first layer of decoder.
        decoder_first_layer_activation_derivatives = activation_deriv
ative_func(z_with_bias @ self.decoder_weights[0].T)
        decoder_first_layer_weights_without_bias = self.decoder_weigh
ts[0][:, 1:]

        # Cycle through the batch of next layer's activation derivati
ves
        for batch_index, next_layer_activation_derivative in enumerat
e(decoder_first_layer_activation_derivatives):
            next_layer_activation_derivative = next_layer_activation_
derivative.reshape(-1, 1)
            second_term = next_layer_activation_derivative * decoder_
first_layer_weights_without_bias
            mu_derivatives[batch_index] = decoder_output_derivatives[
0][batch_index] @ second_term

        return mu_derivatives

    def _calculate_log_var_derivative(self, decoder_output_derivative
s, log_var):
        log_var_derivatives = np.zeros((self.batch_size, self.z_shape
))

        # Add a bias to z
        z_with_bias = np.insert(self.sample_z, obj = 0, values = 1, a
xis = 1)

        # Activation derivative function for the first layer of decod
er
        activation_for_decoder_first_layer = self.decoder_activations
[0]
        activation_derivative_func = self.activations_derivatives[act
ivation_for_decoder_first_layer]

        # Activation derivatives for the first layer of decoder.
        decoder_first_layer_activation_derivatives = activation_deriv
ative_func(z_with_bias @ self.decoder_weights[0].T)
        decoder_first_layer_weights_without_bias = self.decoder_weigh
ts[0][:, 1:]

        # Cycle through the batch of next layer's activation derivati
ves
        for batch_index, next_layer_activation_derivative in enumerat
e(decoder_first_layer_activation_derivatives):
            next_layer_activation_derivative = next_layer_activation_
derivative.reshape(-1, 1)
```

```
            second_term = next_layer_activation_derivative * decoder_
first_layer_weights_without_bias
            log_var_derivatives[batch_index] = (decoder_output_deriva
tives[0][batch_index] @ second_term) * \
                                                np.exp(log_var * .5)[batc
h_index] * 0.5 * self.rand_sample[batch_index]

        return log_var_derivatives

    def _backward_encoder_recon_loss(self, encoder_out, decoder_out,
decoder_output_derivatives, log_var):
        encoder_output_derivatives = deepcopy(encoder_out)
        encoder_weight_derivatives = deepcopy(self.encoder_weights)

        # Calculate derivatives of mu and log_var using decoder outpu
ts and derivatives
        mu_derivatives = self._calculate_mu_derivative(decoder_output
_derivatives)
        log_var_derivatives = self._calculate_log_var_derivative(deco
der_output_derivatives, log_var)

        encoder_output_derivatives_recon = deepcopy(encoder_out)
        encoder_weight_derivatives_recon = deepcopy(self.encoder_weig
hts)

        # We calculate weight derivatives for each data row in the ba
tch and average the
        # derivatives at the end.
        encoder_weight_derivatives_recon = [encoder_weight_derivative
s_recon] * self.batch_size
        print(mu_derivatives)
        s
        return

    def _backward_encoder_kl_loss(self):
        return

    def _update_weights(self):
        return

    def _backward(self, xhat_batch, x_batch, encoder_out, decoder_out
, log_var):

        # Calculate decoder gradients. We use the reconstruction loss
to backpropagate through decoder.
        decoder_weight_derivatives, decoder_output_derivatives = self
._backward_decoder(x_batch, decoder_out)

        # Calculate encoder gradients. For encoder, we use both the r
econstruction loss and the
        # KL Divergence loss.
        encoder_weight_derivatives_recon_loss = self._backward_encode
r_recon_loss(encoder_out,

decoder_out,

decoder_output_derivatives,
```

```
log_var)
        encoder_weight_derivatives_kl_loss = self._backward_encoder_k
l_loss()

        # Update weights using Adam
        self._update_weights(decoder_weight_derivatives, encoder_weig
ht_derivatives)

        return

    def _initialise_weights(self, input_shape):

        # Encoder Layers
        self.num_neurons_each_encoder_layer.append(2) # 2 for two out
puts - mu and sigma
        self.total_encoder_layers = self.num_hidden_layers + 1 # +1 f
or the last output layer
        self.encoder_layers = range(self.total_encoder_layers)

        # Decoder Layers
        self.num_neurons_each_decoder_layer.append(input_shape) # Las
t layer of decoder has input shape
        self.total_decoder_layers = self.num_hidden_layers + 1 # +1 f
or the last output layer
        self.decoder_layers = range(self.total_decoder_layers)

        # Empty weight arrays
        self.encoder_weights = []
        self.decoder_weights = []

        # Initialise encoder weights
        for layer in self.encoder_layers:
            self.encoder_weights.append([])

            number_of_neurons_in_this_layer = self.num_neurons_each_e
ncoder_layer[layer]
            if layer == 0:
                fan_in = input_shape
                previous_layer_shape = fan_in
            else:
                fan_in = self.num_neurons_each_encoder_layer[layer -
1]
                previous_layer_shape =  1 + fan_in

            fan_out = number_of_neurons_in_this_layer
            init_bound = np.sqrt(6. / (fan_in + fan_out))
            if layer != self.encoder_layers[-1]:
                self.encoder_weights[layer] = np.random.uniform(low =
-init_bound,
                                                                high
= init_bound,
                                                                size
= (number_of_neurons_in_this_layer,

previous_layer_shape))
            else:
```

```
                # Last layer of encoder outputs mu and sigma whose di
mensions
                # are of shape z_shape.
                self.encoder_weights[layer] = np.random.uniform(low =
-init_bound,
                                                                high
= init_bound,
                                                                size
= (number_of_neurons_in_this_layer,

self.z_shape,

previous_layer_shape))



        # Initialise decoder weights
        for layer in self.decoder_layers:
            self.decoder_weights.append([])

            number_of_neurons_in_this_layer = self.num_neurons_each_d
ecoder_layer[layer]
            if layer == 0:
                # Input to decoder is the latent variable constructed
from
                # gaussian distribution
                fan_in = self.z_shape
            else:
                fan_in =  self.num_neurons_each_layer[layer - 1]

            fan_out = number_of_neurons_in_this_layer
            previous_layer_shape = 1 + fan_in # +1 for the bias
            init_bound = np.sqrt(6. / (fan_in + fan_out))
            self.decoder_weights[layer] = np.random.uniform(low = -in
it_bound,
                                                            high = in
it_bound,
                                                            size = (n
umber_of_neurons_in_this_layer,
                                                                   pr
evious_layer_shape))

        self.encoder_weights = np.array(self.encoder_weights)
        self.decoder_weights = np.array(self.decoder_weights)
        self.old_encoder_weights = deepcopy(self.encoder_weights)
        self.old_decoder_weights = deepcopy(self.decoder_weights)

    def _get_batches(self, X):
        for i in range(0, X.shape[0], self.batch_size):
            yield X[i: i + self.batch_size]

    def fit(self, X):

        # Add a bias column to X
        X_new = np.column_stack((np.ones(len(X)), X))

        # Initialise weights using Glorot Uniform initialiser
```

```
        self._initialise_weights(X_new.shape[1])

        # Get batches
        batches = self._get_batches(X_new)

        iterations = 0
        while iterations <= self.epochs:

            # Train using mini-batch SGD
            for x_batch in batches:

                # Forward pass
                mu, log_var, xhat_batch, encoder_out, decoder_out = s
elf._forward(x_batch)

                # Reconstruction Loss - between decoded output and in
put data
                reconstruction_loss = self._binary_cross_entropy_loss
(xhat_batch, x_batch)

                # Calculate KL Divergence between sampled z (Gaussian
Distribution: N(mu, sigma))
                # and N(0, 1)
                kl_loss = self._kl_divergence(mu, log_var)

                loss = reconstruction_loss + kl_loss
                loss = loss / self.batch_size

                # Backward pass - for every result in the batch
                # calculate gradient and update the weights using Ada
m
                self._backward(xhat_batch, x_batch, encoder_out, deco
der_out, log_var)
```

KMeans

In [ ]:
```
class KMeans():

    def __init__(self, k = 5, max_iters = 100, random_seed = 42):
        self.k = k
        self.max_iters = max_iters

        # Set random seed
        np.random.seed(random_seed)

    def _initialise_centroids(self, X):
        random_indices = np.random.permutation(X.shape[0])
        random_indices = random_indices[:self.k]
        self.centroids = X[random_indices]

    def _euclidien_distance(self, x):
        return np.sum((x - self.centroids)**2, axis = 1)

    def _assign_clusters(self, X):
        cluster_distances = pairwise_distances(X, self.centroids, met
ric = 'euclidean')
        cluster_labels = np.argmin(cluster_distances, axis = 1)
        return cluster_labels

    def _update_centroids(self, X, cluster_labels):
        for cluster in range(self.k):

            # Get all data points of a cluster
            X_cluster = X[cluster_labels == cluster]

            # Update the cluster's centroid
            cluster_mean = np.mean(X_cluster, axis = 0)
            self.centroids[cluster] = cluster_mean

    def fit(self, X):

        # Initialise random centroids
        self._initialise_centroids(X)

        iterations = 0
        while iterations <= self.max_iters:
            iterations += 1

            # Assign clusters to data
            cluster_labels = self._assign_clusters(X)

            # Update centroids
            self._update_centroids(X, cluster_labels)

    def predict(self, X):
        return self._assign_clusters(X)
```

Gaussian Mixture Model

In [124]:
```
class GaussianMixtureModel():

    def __init__(self, k = 5, max_iters = 100, random_seed = 42, reg_
covar = 1e-6, verbose = True):
        self.k = k # number of Gaussians
        self.max_iters = max_iters
        self.reg_covar = reg_covar
        self.verbose = verbose

        # Set random seed
        np.random.seed(random_seed)

    def _initialise_prams(self, X):

        # Get initial clusters using Kmeans
        kmeans = KMeans(k = self.k, max_iters = 500)
        kmeans.fit(X)
        kmeans_preds = kmeans.predict(X)

        N, col_length = X.shape
        mixture_labels = np.unique(kmeans_preds)
        initial_mean = np.zeros((self.k, col_length))
        initial_cov = np.zeros((self.k, col_length, col_length))
        initial_pi = np.zeros(self.k)

        for index, mixture_label in enumerate(mixture_labels):
            mixture_indices = (kmeans_preds == mixture_label)
            Nk = X[mixture_indices].shape[0]

            # Initial pi
            initial_pi[index] = Nk/N

            # Intial mean
            initial_mean[index, :] = np.mean(X[mixture_indices], axis
= 0)

            # Initial covariance
            de_meaned = X[mixture_indices] - initial_mean[index, :]
            initial_cov[index] = np.dot(initial_pi[index] * de_meaned
.T, de_meaned) / Nk
        assert np.sum(initial_pi) == 1
        return initial_pi, initial_mean, initial_cov

    def _compute_loss(self, X):
        N = X.shape[0]
        loss = np.zeros((N, self.k))

        for k in range(self.k):
            dist = mvn(self.mu[k], self.cov[k], allow_singular = True
)
            loss[:, k] = self.gamma[:, k] * (np.log(self.pi[k] + 1e-5
) + \
                                                        dist.logpdf(X) - np
.log(self.gamma[:, k] + 1e-6))
        loss = np.sum(loss)
        return loss
```

```
    def _E(self, X):
        row_length, col_length = X.shape
        self.gamma = np.zeros((row_length, self.k))

        # Calculate gamma
        for k in range(self.k):
            # Regularise the covariance to prevent singular matrix
            self.cov[k].flat[::col_length + 1] += self.reg_covar
            self.gamma[:, k] = self.pi[k] * mvn.pdf(X, self.mu[k, :],
self.cov[k])

        # Normalise gamma
        self.gamma = self.gamma/np.sum(self.gamma, axis = 1, keepdims
= True)

    def _M(self, X):
        N = X.shape[0]
        col_length = X.shape[1]

        Nk = self.gamma.sum(axis = 0)[:, np.newaxis]

        # Update pi
        self.pi = Nk/N

        # Update mu
        self.mu = (self.gamma.T @ X)/Nk

        # Update covariance
        for k in range(self.k):
            x = X - self.mu[k, :] # (N x d)

            gamma_diag = np.diag(self.gamma[:, k])
            x_mu = np.matrix(x)
            gamma_diag = np.matrix(gamma_diag)

            cov_k = x.T * gamma_diag * x
            self.cov[k] = (cov_k) / Nk[k]

    def fit(self, X):

        # Initialise parameters
        self.pi, self.mu, self.cov = self._initialise_prams(X)

        iterations = 0
        while iterations <= self.max_iters:
            iterations += 1

            # Expectation Step
            self._E(X)

            # Maximisation Step
            self._M(X)

            # Get the loss
            loss = self._compute_loss(X)
            if self.verbose:
```

```
                print("Epoch - ", str(iterations), " Loss - ", str(lo
ss))

    def predict_proba(self, X):
        labels = np.zeros((X.shape[0], self.k))
        for k in range(self.k):
            self.cov[k].flat[::X.shape[1] + 1] += self.reg_covar
            labels[:, k] = self.pi[k] * mvn.pdf(X, self.mu[k, :], sel
f.cov[k])

        # Normalise
        labels = labels/np.sum(labels, axis = 1, keepdims = True)
        return labels

    def predict(self, X):
        labels = np.zeros((X.shape[0], self.k))
        for k in range(self.k):
            self.cov[k].flat[::X.shape[1] + 1] += self.reg_covar
            labels[:, k] = self.pi[k] * mvn.pdf(X, self.mu[k, :], sel
f.cov[k])

        # Normalise
        labels = labels/np.sum(labels, axis = 1, keepdims = True)
        labels  = labels.argmax(1)
        return labels

    def sample(self, n_samples = 1):
        n_samples_comp = np.random.multinomial(n_samples, self.pi.res
hape(1, -1)[0])
        X = np.vstack([
                np.random.multivariate_normal(mean, covariance, int(s
ample))
                for (mean, covariance, sample) in zip(
                    self.mu, self.cov, n_samples_comp)])
        y = np.concatenate([np.full(sample, j, dtype = int) for j, sa
mple in enumerate(n_samples_comp)])
        return X, y
```

In [129]:
```
gmm = GaussianMixtureModel(k = 3, max_iters = 20)
gmm.fit(X[:, :20])
```

```
Epoch -  1  Loss -  -91452.36730602988
Epoch -  2  Loss -  -88367.67455596146
Epoch -  3  Loss -  -92206.31206617941

/Users/adityavyas/anaconda/envs/py36/lib/python3.6/site-packages/scip
y/stats/_multivariate.py:522: RuntimeWarning: underflow encountered i
n exp
  out = np.exp(self._logpdf(x, mean, psd.U, psd.log_pdet, psd.rank))

Epoch -  4  Loss -  -92287.40910402693
Epoch -  5  Loss -  -88662.1106432321
Epoch -  6  Loss -  -92738.98532379243
Epoch -  7  Loss -  -92716.39757179572
Epoch -  8  Loss -  -92713.436418836
Epoch -  9  Loss -  -92710.76866866181
Epoch -  10  Loss -  -92708.4769974573
Epoch -  11  Loss -  -92707.75214230109
Epoch -  12  Loss -  -92707.65952135189
Epoch -  13  Loss -  -92707.65210713033
Epoch -  14  Loss -  -92707.65134446723
Epoch -  15  Loss -  -92707.65121799172
Epoch -  16  Loss -  -92707.6511918644
Epoch -  17  Loss -  -92707.65118616608
Epoch -  18  Loss -  -92707.65118491817
Epoch -  19  Loss -  -92707.65118464959
Epoch -  20  Loss -  -92707.65118459439
Epoch -  21  Loss -  -92707.65118458436
```

In [130]:
```
gmm.sample(100)
```

Out[130]:
```
(array([[ 5.48612297,  5.48612294,  0.11789534, ...,  1.57360866,
          0.19713533,  0.68390668],
        [11.8067395 , 11.8067395 ,  0.28027371, ...,  0.17516452,
          0.04393491,  1.22830315],
        [19.03332185, 19.03332175, -0.13324087, ...,  0.52026637,
          0.72715486,  1.42492499],
        ...,
        [21.8943055 , 21.8943055 ,  0.13872084, ...,  0.26560805,
          0.03964917,  0.61686526],
        [27.83727985, 27.83727984, -0.54127388, ...,  0.81630838,
          0.16788616,  1.16213122],
        [26.37108503, 26.37108511,  0.29160193, ...,  1.58113643,
         -0.17888648,  0.03928547]]),
 array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1,
        1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2]))
```

Supervised/Unsupervised Learning

In [21]:
```
class MeanMode():
    def __init__(self, numeric_columns):
        self.numeric_columns = numeric_columns

    def predict(self, feature_name, train):
        if feature_name in self.numeric_columns:
            return np.mean(train)
        else:
            return Counter(train).most_common(1)[0][0]

    def categorical_loss(self, y_true, y_pred):
#         print(np.mean((y_pred == y_true)))
#         print("Predicted, Actual ", y_pred, y_true)
        return np.mean((y_pred == y_true))

    def get_mse(self, y_true, y_pred, feature_name):
        y_true = np.asarray(y_true)
        y_pred = np.asarray(y_pred)
        if feature_name in self.numeric_columns:
            return np.mean((y_true - y_pred)**2)
        else:
            return self.categorical_loss(y_true, y_pred)
```

2. Linear Regression

In [22]:
```
class LinearRegression:
    def __init__(self, weights=None, bias=None):
        self.w = weights
        self.b = bias

    def predict(self, X):
        return (np.dot(X, self.w)) + self.b

    def fit(self, X, y, alpha=0.001, iterations=100):
#         Step 1: Initialize the parameters
        n_samples, n_features = X.shape
        self.w = np.zeros(shape=(n_features, 1))
        self.b = 0
        J = []
        y = y.reshape(-1, 1)
        for i in range(iterations):
#             Step 2: Calculate y_predicted
            y_hat = self.predict(X)
#             Step 3: Compute the cost
            cost = (1/n_samples)*np.sum((y_hat-y)**2)
            J.append(cost)

#             if i % (iterations-1) == 0:
#                 print("Cost at iteration {} is: {}".format(i, cos
t))

#             Step 4: Compute partial derivatives
            dJ_dw = (2/n_samples)*np.dot(X.T, (y_hat-y))
            dJ_db = (2/n_samples)*np.sum((y_hat-y))
#             Step 5: Update the parameters
            self.w = self.w - alpha*dJ_dw
            self.b = self.b - alpha*dJ_db

    def get_mse(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)
```

3. Ridge Regression

In [23]:
```
class RidgeRegression():

    def __init__(self,
                 bias = None,
                 weights = None,
                 lambda_param = 10,
                 fit_intercept = True):
        self.bias = bias
        self.weights = weights
        self.fit_intercept = fit_intercept
        self.lambda_param = lambda_param

    def fit(self, X, y):
        if self.fit_intercept:
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))

        self.all_weights = np.linalg.inv(np.dot(X.T, X) + \
                             self.lambda_param * np.identity(X.shape[1
])).dot(X.T).dot(y)
        self.weights = self.all_weights[1:]
        self.bias = self.all_weights[0]

    def predict(self, X):
        self.weights = self.weights.reshape(1, -1)
        predictions = self.bias + np.dot(self.weights, X.T)
        return predictions[0]

    def get_mse(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)
```

4. Lasso Regression

In [24]:
```
class LassoRegression():
    def __init__(self,
                 bias = None,
                 weights = None,
                 lambda_param = 10,
                 max_iters = 100,
                 fit_intercept = True):
        self.bias = 0
        self.lambda_param = lambda_param
        self.max_iters = max_iters
        self.fit_intercept = fit_intercept

    def _soft_threshold(self, x, lambda_):
        if x > 0.0 and lambda_ < abs(x):
            return x - lambda_
        elif x < 0.0 and lambda_ < abs(x):
            return x + lambda_
        else:
            return 0.0

    def fit(self, X, y):
        if self.fit_intercept:
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))

        row_length, column_length = X.shape

#         print("X, y", X.shape, y.shape) #shapes are fine
        # Define the weights
        self.weights = np.zeros((1, column_length))[0]
#         print("w", self.weights.shape)
        if self.fit_intercept:
            self.weights[0] = np.sum(y - \
                                     np.dot(X[:, 1:], self.weights[1:]))/(
X.shape[0])

#         print("bias", self.weights[0]) # value coming properly
        #Looping until max number of iterations
        for iteration in range(self.max_iters):
            start = 1 if self.fit_intercept else 0

            #Looping through each coordinate
            for j in range(start, column_length):
                tmp_weights = self.weights.copy()
                tmp_weights[j] = 0.0
                r_j = y - np.dot(X, tmp_weights)
                arg1 = np.dot(X[:, j], r_j)
                arg2 = self.lambda_param * X.shape[0]

                self.weights[j] = self._soft_threshold(arg1, arg2)/(X
[:, j]**2).sum()

#                 print(iteration, j, self.weights[j], np.unique(X[:,
j]), np.min(X[:, j]), np.max(X[:, j]))
```

```
                if self.fit_intercept:
                    self.weights[0] = np.sum(y - np.dot(X[:, 1:], sel
f.weights[1:]))/(X.shape[0])

        self.bias = self.weights[0]
        self.weights = self.weights[1:]

    def predict(self, X):
        self.weights = self.weights.reshape(1, -1)
        predictions = self.bias + np.dot(self.weights, X.T)
        return predictions[0]

    def get_mse(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)
```

5. Decision Tree Regressor

In [25]:

```
class Node():
    def __init__(self,
                 data = None,
                 split_variable = None,
                 split_variable_value = None,
                 left = None,
                 right = None,
                 depth = 0,
                 criterion_value = None):
        self.data = data
        self.split_variable = split_variable
        self.split_variable_value = split_variable_value
        self.left = left
        self.right = right
        self.criterion_value = criterion_value
        self.depth = depth
```

In [26]:
```
class DecisionTreeRegressor():
    def __init__(self,
                 root = None,
                 criterion = "mse",
                 max_depth = 2,
                 significance = None,
                 significance_threshold = 3.841,
                 min_samples_split = 10):
        self.root = root
        self.criterion = criterion
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.significance = significance
        self.significance_threshold = significance_threshold

        self.split_score_funcs = {'mse': self._calculate_mse_values}

    def _get_mse(self, X):
        if X.empty:
            return 0

        # Calculate the mean square error with respect to the mean
        y = X['Y']
        y_mean = np.mean(y)
        mse = np.mean((y - y_mean)**2)
        return mse

    def _calculate_mse_values(self, X, feature):

        # Calculate unique values of X. For a feature, there are diff
erent
        # values on which that feature can be split
        classes = X[feature].unique()

        # Calculate the gini value for a split on each unique value o
f the feature.
        best_mse_score = np.iinfo(np.int32(10)).max
        best_feature_value = ""
        for unique_value in classes:
            # Split data
            left_split = X[X[feature] <= unique_value]
            right_split = X[X[feature] > unique_value]

            # Get gini scores of left, right nodes
            mse_value_left_split = self._get_mse(left_split)
            mse_value_right_split = self._get_mse(right_split)

            # Combine the 2 scores to get the overall score for the s
plit
            mse_score_of_current_value = (left_split.shape[0]/X.shape
[0]) * mse_value_left_split + \
                                                (right_split.shape[0]/X.sh
ape[0]) * mse_value_right_split
            if mse_score_of_current_value < best_mse_score:
                best_mse_score = mse_score_of_current_value
                best_feature_value = unique_value
```

```
        return best_mse_score, best_feature_value

    def _get_best_split_feature(self, X):
        best_split_score = np.iinfo(np.int32(10)).max
        best_feature = ""
        best_value = None
        columns = X.drop('Y', 1).columns

        for feature in columns:

            # Calculate the best split score and the best value
            # for the current feature.
            split_score, feature_value = self.split_score_funcs[self.
criterion](X, feature)

            # Compare this feature's split score with the current bes
t score
            if split_score < best_split_score:
                best_split_score = split_score
                best_feature = feature
                best_value = feature_value

        return best_feature, best_value, best_split_score

    def _split_data(self, X, X_depth = None):

        # Return if dataframe is empty, depth exceeds maximum depth o
r sample size exceeds
        # minimum sample size required to split.
        if X.empty or len(X['Y'].value_counts()) == 1 or X_depth == s
elf.max_depth \
                                 or X.shape[0] <= self.min_samples_split:
            return None, None, "", "", 0

        # Calculate the best feature to split X
        best_feature, best_value, best_score = self._get_best_split_f
eature(X)

        if best_feature == "":
            return None, None, "", "", 0

        # Create left and right nodes
        X_left = Node(data = X[X[best_feature] <= best_value].drop(be
st_feature, 1),
                      depth = X_depth + 1)
        X_right = Node(data = X[X[best_feature] > best_value].drop(be
st_feature, 1),
                       depth = X_depth + 1)

        return X_left, X_right, best_feature, best_value, best_score

    def _fit(self, X):

        # Handle the initial case
        if not (type(X) == Node):
            X = Node(data = X)
```

```
            self.root = X

        # Get the splits
        X_left, X_right, best_feature, best_value, best_score = self.
_split_data(X.data, X.depth)

        # Assign attributes of node X
        X.left = X_left
        X.right = X_right
        X.split_variable = best_feature
        X.split_variable_value = round(best_value, 3) if type(best_va
lue) != str else best_value
        X.criterion_value = round(best_score, 3)

        # Return if no best variable found to split on.
        # This means you have reached the leaf node.
        if best_feature == "":
            return

        # Recurse for left and right children
        self._fit(X_left)
        self._fit(X_right)

    def fit(self, X, y):

        # Combine the 2 and fit
        X = pd.DataFrame(X)
        X['Y'] = y
        self._fit(X)

    def predict(self, X):
        X = np.asarray(X)
        X = pd.DataFrame(X)

        preds = []
        for index, row in X.iterrows():
            curr_node = self.root
            while(curr_node.left != None and curr_node.right != None
):
                split_variable = curr_node.split_variable
                split_variable_value = curr_node.split_variable_value
                if X.loc[index, split_variable] <= split_variable_val
ue:
                    curr_node = curr_node.left
                else:
                    curr_node = curr_node.right

            # Get prediction
            preds.append(np.mean(curr_node.data['Y'].values))

        return preds

    def display_tree_structure(self):
        tree = Digraph('DecisionTree',
                       filename = 'tree.dot',
                       node_attr = {'shape': 'box'})
        tree.attr(size = '10, 20')
```

```
        root = self.root
        id = 0

        # queue with nodes to process
        nodes = [(None, root, 'root')]
        while nodes:
            parent, node, x = nodes.pop(0)

            # Generate appropriate labels for the nodes
            value_counts_length = len(node.data['Y'].value_counts())
            if node.split_variable != "":
                split_variable = node.split_variable
                split_variable_value = node.split_variable_value
            else:
                split_variable = "None"

            if value_counts_length > 1:
                label = str(split_variable) + '\n' + str(self.criteri
on) + " = " + \
                            str(node.criterion_value)
            else:
                label = "None"

            # Make edges between the nodes
            tree.node(name = str(id),
                      label = label,
                      color = 'black',
                      fillcolor = 'goldenrod2',
                      style = 'filled')

            if parent is not None:
                if x == 'left':
                    tree.edge(parent, str(id), color = 'sienna',
                              style = 'filled', label = '<=' + ' ' +
str(split_variable_value))
                else:
                    tree.edge(parent, str(id), color = 'sienna',
                              style = 'filled', label = '>' + ' ' + s
tr(split_variable_value))

            if node.left is not None:
                nodes.append((str(id), node.left, 'left'))

            if node.right is not None:
                nodes.append((str(id), node.right, 'right'))
            id += 1

        return tree

    def get_error(self, y, y_hat):
        return np.mean((y - y_hat)**2)
```

In [27]:
```
class KNeighbours():
    def __init__(self, k = 5, distance_metric = 'euclid', problem =
"classify"):
        self.k = k
        self.distance_metric = distance_metric
        self.problem = problem
        self.prediction_functions = {'classify': self._top_k_votes,
                                     'regress': self._top_k_mean}
        self.eval_functions = {'classify': self._get_accuracy,
                               'regress': self._get_mse}

    def fit(self, X, y):
        self.X = np.asarray(X)
        self.y = np.asarray(y)

    def _euclidien_distance(self, x):
        return np.sqrt(np.sum((x - self.X)**2, axis = 1))

    def _top_k_mean(self, top_k):
        return np.mean(top_k)

    def _top_k_votes(self, top_k):
        return max(top_k, key = list(top_k).count)

    def _get_accuracy(self, pred, y):
        return np.mean((pred == y))

    def _get_mse(self, pred, y):
        return np.mean((pred - y)**2)

    def predict(self, X):
        preds = list()
        X = np.asarray(X)
        for x in X:
            distances = self._euclidien_distance(x)

            # Zip the distances and y values together
            distances = zip(*(distances, self.y))

            # Sort the distances list by distance values in descendin
g order
            distances = sorted(distances, key = lambda x: x[0])

            # Select top k distances
            top_k = distances[:(self.k)]

            top_k = np.array(top_k)
            top_k = top_k[:, 1]

            # Calculate mean of y values of these top k data items
            pred = self.prediction_functions[self.problem](top_k)
            preds.append(pred)

        return preds

    def evaluate(self, pred, y):
```

```
        eval_func = self.eval_functions[self.problem]
        return eval_func(pred, y)
```

In [28]:
```
class LogisticRegression():

    def __init__(self,
                 weights = None,
                 bias = None,
                 fit_intercept = True,
                 decision_threshold = 0.5,
                 epochs = 50,
                 solver = 'sgd',
                 batch_size = 30,
                 learning_rate = 0.05,
                 tolerance = 1e-13):
        self.weights = weights
        self.bias = bias
        self.fit_intercept = fit_intercept
        self.tolerance = tolerance
        self.decision_threshold = decision_threshold
        self.epochs = epochs
        self.solver = solver
        self.batch_size = batch_size
        self.learning_rate = learning_rate

        self.solver_func = {'newton': self._newton_solver,
                            'sgd': self._sgd_solver}

    def _sigmoid(self, z):
        return 1.0 / (1.0 + np.exp(-z))

    def _log_likelihood(self, X, y):
        P = self._sigmoid(X @ self.weights)
        P = P.reshape(-1, 1)
        log_P = np.log(P + 1e-16)

        P_ = 1 - P
        log_P_ = np.log(P_ + 1e-16)
        log_likelihood = np.sum(y*log_P + (1 - y)*log_P_)
        return log_likelihood

    def _get_true_class_labels(self, labels):
        true_labels = np.array([self.class_range_to_actual_classes[i]
for i in labels])
        return true_labels

    def _get_batches(self, X, y):
        for i in range(0, X.shape[0], self.batch_size):
            yield (X[i: i + self.batch_size], y[i: i + self.batch_siz
e])

    def _convert_y(self, y):
        self.actual_classes = sorted(np.unique(y))
        self.class_range = [0, 1]
        self.class_range_to_actual_classes = dict(zip(*(self.class_ra
nge, self.actual_classes)))
        self.actual_classes_to_class_range = dict(zip(*(self.actual_c
lasses, self.class_range)))
```

```
        y_ = np.array([self.actual_classes_to_class_range[i] for i in
y])
        y_ = y_.reshape(-1, 1)
        return y_

    def _newton_solver(self, X, y):
        log_likelihood = self._log_likelihood(X, y)
        iterations = 0
        delta = np.inf
        while(np.abs(delta) > self.tolerance and iterations < self.ep
ochs):
            iterations += 1

            # Calculate positive class probabilities: p = sigmoid(W*x
+ b)
            z = X @ self.weights
            P = self._sigmoid(z)
            P = P.reshape(-1, 1)

            # First derivative of loss w.r.t weights
            grad = X.T @ (P - y)

            # Hessian of loss w.r.t weights
            P_ = 1 - P
            W = P * P_
            W = W.reshape(1, -1)[0]
            W = np.diag(W)
            hess = X.T @ W @ X

            # Weight update using Newton-Rhapson Method
            self.weights -= np.linalg.inv(hess) @ grad

            # Calculate new log likelihood
            new_log_likelihood = self._log_likelihood(X, y)
            delta = log_likelihood - new_log_likelihood
            log_likelihood = new_log_likelihood

    def _sgd_solver(self, X, y):
        iterations = 0
        while(iterations < self.epochs):
            iterations += 1

            # Get batches
            batches = self._get_batches(X, y)

            # Update weights using Mini batch stochastic gradient des
cent
            for (x_batch, y_batch) in batches:

                # Raw output
                z = x_batch @ self.weights

                # Calculate positive class probabilities: p = sigmoid
(W*x + b)
                P = self._sigmoid(z)

                # First derivative of loss w.r.t weights
```

```
                grad = x_batch.T @ (P - y_batch)

                # Update weights
                self.weights -= self.learning_rate * grad

    def fit(self, X, y):
        X = np.asarray(X)
        y = np.asarray(y)

        if self.fit_intercept:
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))
        row_length, column_length = X.shape

        # Define the weights
        self.weights = np.zeros((column_length, 1))

        # Convert y to {0, 1}
        y = self._convert_y(y)

        # Use the solver
        self.solver_func[self.solver](X, y)

    def predict_proba(self, X):

        if self.fit_intercept:
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))

        z = X @ self.weights
        predicted_probs = self._sigmoid(z)
        return predicted_probs

    def predict(self, X):
        predict_probs = self.predict_proba(X)
        preds = np.where(predict_probs < 0.5, 0, 1).flatten()
        true_preds = self._get_true_class_labels(preds)
        return true_preds

    def get_accuracy(self, y, y_hat):
        return np.mean(y == y_hat)
```

In [29]:
```
class MultiClassLogisticRegression():

    def __init__(self,
                 weights = None,
                 bias = None,
                 fit_intercept = True,
                 epochs = 50,
                 learning_rate = 0.05,
                 batch_size = 50):
        self.weights = weights
        self.learning_rate = learning_rate
        self.bias = bias
        self.fit_intercept = fit_intercept
        self.epochs = epochs
        self.batch_size = batch_size

#     def _softmax(self, z):
#         e_x = np.exp(z)
#         out = e_x / (1 + e_x.sum(axis = 1, keepdims = True))
#         return out
    def _softmax(self, z):

        # We only calculate the softmax probabilities of the first (K
-1) classes
        z_ = z[:, :(z.shape[1] - 1)]
        e_x = np.exp(z_)
        out_k_minus_1 = e_x / (1 + e_x.sum(axis = 1, keepdims = True
))

        # Probability for last K = 1 - p((K - 1))
        out_k = 1 - out_k_minus_1.sum(axis = 1)
        out = np.column_stack((out_k_minus_1, out_k))

        return out

    def _get_true_class_labels(self, P):
        labels = P.argmax(axis = 1)
        labels = np.array([self.class_range_to_actual_classes[i] for
i in labels])
        return labels

    def _calculate_cross_entropy(self, y, log_yhat):
        return -np.sum(y * log_yhat, axis = 1)

    def _convert_to_indicator(self, y):
        y_indicator = np.zeros((y.shape[0], self.num_classes))
        for index, y_value in enumerate(y):
            class_range_mapping = int(self.actual_classes_to_class_ra
nge[y_value])
            y_indicator[index, class_range_mapping] = 1
        return y_indicator

    def _get_batches(self, X, y):
        for i in range(0, X.shape[0], self.batch_size):
            yield (X[i: i + self.batch_size], y[i: i + self.batch_siz
e])
```

```
    def fit(self, X, y):
        X = np.asarray(X)
        y = np.asarray(y)
        if self.fit_intercept:
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))
        row_length, column_length = X.shape

        # Number of unique classes
        self.actual_classes = sorted(np.unique(y))
        self.num_classes = len(self.actual_classes)

        # This will generate a list of [0,1,2,3....]. However, we wan
t to map these class labels
        # to the original class labels in Y
        self.class_range = list(range(self.num_classes))
        self.class_range_to_actual_classes = dict(zip(*(self.class_ra
nge, self.actual_classes)))
        self.actual_classes_to_class_range = dict(zip(*(self.actual_c
lasses, self.class_range)))

        # Convert y to indicator matrix form e.g. If y belongs to cla
ss 3, then y = [0,0,1,0..0]
        y = self._convert_to_indicator(y)

        # Define the weights, shape = (P + 1, K)
        self.weights = np.zeros((column_length, self.num_classes))

        iterations = 0
        while(iterations < self.epochs):
            iterations += 1

            # Get batches
            batches = self._get_batches(X, y)

            # Update weights using Mini batch stochastic gradient des
cent
            for (x_batch, y_batch) in batches:

                # Get raw output
                z = x_batch @ self.weights

                # Calculate class probabilities from raw output, shap
e = (B, K); B = batch size
                P = self._softmax(z)

                # Calculate gradient
                grad = x_batch.T @ (P - y_batch)

                # Update weights
                self.weights -= self.learning_rate * grad

    def predict_proba(self, X):

        if self.fit_intercept:
```

```
            X = np.column_stack((np.ones(len(X)), X))
        else:
            X = np.column_stack((np.zeros(len(X)), X))

        z = X @ self.weights
        predicted_probs = self._softmax(z)
        return predicted_probs

    def predict(self, X):
        predicted_probs = self.predict_proba(X)
        preds = self._get_true_class_labels(predicted_probs)
        return preds

    def get_accuracy(self, y, y_hat):
        return np.mean(y == y_hat)
```

In [30]:
```
class NeuralNetworkRegressor():

    SigmoidActivation = "sigmoid"
    ReLUActivation = "relu"
    LinearActivation = "linear"

    def __init__(self,
                 num_hidden_layers = 1,
                 learning_rate = 0.03,
                 num_neurons_each_layer = [10],
                 num_neurons_last_layer = 1,
                 batch_size = 32,
                 epochs = 10,
                 weights = None):
        self.weights = weights
        self.num_hidden_layers = num_hidden_layers
        self.num_neurons_each_layer = num_neurons_each_layer
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.num_neurons_last_layer = num_neurons_last_layer

        # Sigmoid activation for other layers. Linear activation for
 last layer
        self.activations = [self.ReLUActivation] * self.num_hidden_la
yers + [self.LinearActivation]
        self.activations_functions = {
            self.SigmoidActivation: self._sigmoid,
            self.ReLUActivation: self._relu,
            self.LinearActivation: self._linear
        }
        self.activations_derivatives = {
            self.SigmoidActivation: self._sigmoid_derivative,
            self.ReLUActivation: self._relu_derivative,
            self.LinearActivation: self._linear_derivative
        }

    def _sigmoid(self, x):
        def sigfunc(x):
            if x < 0:
                return 1 - 1 / (1 + math.exp(x))
            else:
                return 1 / (1 + math.exp(-x))
        x_ = np.array([sigfunc(i) for i in x])
        return x_

    def _relu(self, x):
        return np.maximum(0, x)

    def _linear(self, x):
        return x

    def _sigmoid_derivative(self, x):
        return self._sigmoid(x) * (1 - self._sigmoid(x))

    def _relu_derivative(self, x):
```

```
        return (np.ones_like(x) * (x > 0))

    def _linear_derivative(self, x):
        return np.ones_like(x)

    def _mse_loss(self, pred, y):
        return np.mean((pred - y) ** 2)

    def _initialise_weights(self, input_shape):

        self.num_neurons_each_layer.append(self.num_neurons_last_laye
r)
        self.total_layers = self.num_hidden_layers + 1
        self.layers = range(self.total_layers)

        # Initialising a numpy array of
        # shape = (number of hidden layers, number of neurons, number
of weights per neuron) to store weights
        self.weights = []

        # Iterate through the layers
        for layer in self.layers:
            self.weights.append([])

            number_of_neurons_in_this_layer = self.num_neurons_each_l
ayer[layer]
            if layer == 0:
                fan_in = input_shape
                fan_out = number_of_neurons_in_this_layer
                previous_layer_shape = fan_in
            else:
                fan_in = self.num_neurons_each_layer[layer - 1]
                fan_out = number_of_neurons_in_this_layer
                previous_layer_shape =  1 + fan_in

            init_bound = np.sqrt(2. / (fan_in + fan_out))
            self.weights[layer] = np.random.uniform(low = -init_bound
,
                                                    high = init_bound
,
                                                    size = (number_of
_neurons_in_this_layer,
                                                            previous_l
ayer_shape))

        self.weights = np.array(self.weights)
        self.old_weights = deepcopy(self.weights)

    def _update_weights(self):
        avg_batch_weight_derivatives = np.mean(self.batch_weight_deri
vatives, axis = 0)
        self.weights = self.old_weights - self.learning_rate * avg_ba
tch_weight_derivatives
        self.old_weights = deepcopy(self.weights)
        self.batch_weight_derivatives = []

    def _backward(self, x, y, out):
```

```
        # The derivatives array will have the same shape as weights a
rray. - one derivative for each
        # weight
        output_derivatives = deepcopy(out)
        weight_derivatives = deepcopy(self.weights)

        # Compute the output derivatives
        layers_reversed = self.layers[::-1]
        for curr_layer in layers_reversed:
            next_layer = curr_layer + 1

            # For the last layer simply use the formula
            if curr_layer == self.total_layers - 1:
                output_derivatives[curr_layer] = 2*(out[curr_layer] -
y)
                continue

            # Get the activation derivative function for next layer
            activation_for_next_layer = self.activations[next_layer]
            activation_derivative = self.activations_derivatives[acti
vation_for_next_layer]

            # The next layer output derivatives
            next_layer_output_derivatives = output_derivatives[next_l
ayer]

            # Calculate the activation derivative. Add a 1 for the bi
as weight
            current_layer_output = out[curr_layer].copy()
            current_layer_output = np.insert(current_layer_output, ob
j = 0, values = 1)
            next_layer_activation_derivatives = activation_derivative
(self.old_weights[next_layer] @ current_layer_output)
            next_layer_activation_derivatives = next_layer_activation
_derivatives.reshape(-1, 1)

            # Remove the bias from the weights.
            next_layer_weights_without_bias = self.old_weights[next_l
ayer][:, 1:]

            # Multiply each neuron's activation derivative with its w
eights. This is the Hadmard product
            second_term = next_layer_activation_derivatives * next_la
yer_weights_without_bias

            # Sum over all the neurons in the next layer to get the o
utput derivative for each
            # neuron in the current layer. This is because each neuro
n contributes to all the neurons
            # in the next layer.
            output_derivatives[curr_layer] = next_layer_output_deriva
tives @ second_term

        # Update the weights using the output derivative calculated a
bove
        for curr_layer in layers_reversed:
```

```
            # Get the activation for this layer and its derivative fu
nction
            activation_for_this_layer = self.activations[curr_layer]
            activation_derivative = self.activations_derivatives[acti
vation_for_this_layer]

            # If first layer then use the data as the previous layer.
            if curr_layer == 0:
                previous_layer_output = x
            else:
                prev_layer = curr_layer - 1
                previous_layer_output = out[prev_layer].copy()
                previous_layer_output = np.insert(previous_layer_outp
ut, obj = 0, values = 1)

            # Current layer output derivatives
            curr_layer_output_derivatives = output_derivatives[curr_l
ayer].reshape(-1, 1)

            # Get current layer's activation derivatives
            curr_layer_activation_derivatives = activation_derivative
(self.old_weights[curr_layer] @ previous_layer_output)
            curr_layer_activation_derivatives = curr_layer_activation
_derivatives.reshape(-1, 1)

            # For the current layer multiply each neuron's activation
derivatives with all previous layer outputs.
            curr_layer_weight_derivatives = curr_layer_output_derivat
ives * \
                                                curr_layer_activation_der
ivatives * previous_layer_output
            weight_derivatives[curr_layer] = curr_layer_weight_deriva
tives

        # Append the current data point's weight derivatives in the b
atch derivatives array
        self.batch_weight_derivatives.append(weight_derivatives)

    def _forward(self, x):
        out = []
        for curr_layer in self.layers:
            out.append([])

            # Get the activation for this layer and its function
            activation_for_this_layer = self.activations[curr_layer]
            activation_function = self.activations_functions[activati
on_for_this_layer]

            if curr_layer == 0:
                previous_layer_output = x
            else:
                previous_layer_output = out[curr_layer - 1].copy()
                previous_layer_output = np.insert(previous_layer_outp
ut, obj = 0, values = 1)

            out[curr_layer] = activation_function(self.weights[curr_l
```

```
ayer] @ previous_layer_output)

        out = np.array(out)
        return out

    def fit(self, X, y):
        X = np.asarray(X)
        y = np.asarray(y)
        # Add a bias column to X
        X_new = np.column_stack((np.ones(len(X)), X))

        # Initialise the weights of the network
        self._initialise_weights(X_new.shape[1])

        for epoch in range(self.epochs):

            # Initialise arrays to store all weight derivatives of th
e batch
            self.batch_weight_derivatives = []

            # Update weights using mini-batch stochastic gradient des
cent
            for data_index in range(X_new.shape[0]):
                out = self._forward(X_new[data_index])
                self._backward(X_new[data_index], y[data_index], out)

                if (data_index + 1) % self.batch_size == 0:
                    self._update_weights()

            predictions = self.predict(X)
            loss = self._mse_loss(predictions, y)
            print("Epoch = ", str(epoch + 1), " - ", "Loss = ", str(l
oss))

    def predict(self, X):

        # Add a bias column to X
        X_new = np.column_stack((np.ones(len(X)), X))

        preds = []
        for x in X_new:
            pred = self._forward(x)[-1]
            preds.append(pred)

        preds = np.array(preds).flatten()
        return preds
```

In [31]:
```
class AdaboostClassifier():

    def __init__(self, n_estimators = 100, weights = None):
        self.n_estimators = n_estimators
        self.weights = weights
        self.alphas = []

    def _convert_y(self, y):
        self.actual_classes = sorted(np.unique(y))
        self.class_range = [-1, 1]
        self.class_range_to_actual_classes = dict(zip(*(self.class_ra
nge, self.actual_classes)))
        self.actual_classes_to_class_range = dict(zip(*(self.actual_c
lasses, self.class_range)))

        y_ = np.array([self.actual_classes_to_class_range[i] for i in
y])
        return y_

    def _get_true_class_labels(self, labels):
        true_labels = np.array([self.class_range_to_actual_classes[i]
for i in labels])
        return true_labels

    def fit(self, X, y):
        X = np.asarray(X)
        y = np.asarray(y)
        # Convert y to {-1, 1}
        y = self._convert_y(y)

        # Initialise weights for all data points
        row_length = X.shape[0]
        self.weights = np.ones((self.n_estimators, row_length))
        self.alphas = np.zeros((self.n_estimators, 1))
        self.estimators = np.empty((self.n_estimators, 1), dtype = ob
ject)

        time_step = 0
        for time_step in range(self.n_estimators):

            # Use a weak classifier to fit on data
            weak_classifier = LogisticRegression(solver = "sgd", epoc
hs = 10)
            weak_classifier.fit(X, y)
            pred = weak_classifier.predict(X)

            # Get weighted error
            weighted_sample_err = (np.sum((pred != y) * self.weights
))/np.sum(self.weights)

            # Alpha for current classifer
            alpha_t = 1/2*np.log(((1 - weighted_sample_err)/weighted_
sample_err) + 1e-16)
            self.alphas[time_step] = alpha_t
            self.estimators[time_step] = weak_classifier
```

```
            # Update weights of next time step for all data points
            if time_step == (self.n_estimators - 1):
                break
            self.weights[time_step + 1, :] = self.weights[time_step,
:] * np.exp(-y * alpha_t * pred)

    def predict(self, X):
        X = np.asarray(X)
        preds = []
        self.estimators = self.estimators.flatten()
        self.alphas = self.alphas.flatten()
        for index in range(self.n_estimators):
            preds.append(self.alphas[index] * self.estimators[index].
predict(X))

        preds = np.sum(preds, 0)
        preds = np.sign(preds)
        true_preds = self._get_true_class_labels(preds)
        return true_preds

    def get_accuracy(self, y, y_hat):
        return np.mean(y == y_hat)
```

In [38]:

```
mean_mode_model = MeanMode(numeric_columns=all_columns[0])
lin_reg = LinearRegression()
ridge = RidgeRegression()
lasso = LassoRegression(max_iters=10)
dt_reg = DecisionTreeRegressor()
knn = KNeighbours()
binary_log = LogisticRegression(epochs=20)
multi_log = MultiClassLogisticRegression(epochs=20)
nn_reg = NeuralNetworkRegressor()
adaboost = AdaboostClassifier()

models = [mean_mode_model, lin_reg, ridge, lasso, dt_reg, knn, binary
_log, multi_log, nn_reg, adaboost]
```

Pipeline

In [36]:
```
# all_columns = [numeric_columns, string_columns, date_columns, all_N
AN_columns]
class Pipeline():
    def __init__(self, data, models, all_columns):
        # Copy of original data
        self.data_orig = data
        self.data = data

        self.all_columns = all_columns
        # 'nan' string to convert to np.nan added
        self.data = self.data.replace({'': np.nan, ' ': np.nan, '.':
np.nan, 'nan': np.nan})
        # self.data["expcomments"] = self.data["expcomments"].replace
({'nan': np.nan})

        # Convert categorical columns to encodings, then automaticall
y np.nan becomes -1, so replaced it
        for col in self.all_columns[1]:
            if col in self.data.columns:
                self.data[col] = self.data[col].astype("category").ca
t.codes
                self.data[col] = self.data[col].replace({-1: np.nan})

        # Model objects
        self.mean_mode = models[0]
        self.lin_reg = models[1]
        self.ridge = models[2]
        self.lasso = models[3]
        self.dt_reg = models[4]
        self.knn = models[5]
        self.b_logistic = models[6]
        self.m_logistic = models[7]
        self.nn_reg = models[8]
        self.adaboost = models[9]

        # "mean_mode", "linear", "ridge", "lasso", "dt_reg", "knn",
 "nn_reg"
        self.regression_models = ["mean_mode", "linear", "ridge", "kn
n"]
        # "mean_mode", "logistic", "knn", "adaboost"
        self.classification_models = ["mean_mode", "logistic", "knn",
"adaboost"]
        self.classification_models_without_ada = ["mean_mode", "logis
tic", "knn"]

    def count_missing(self, data):
        return data.isnull().sum()

    # Calculate the columns which have missing values, seperate into
 two datas, missing and full data
    def missing_value_perc(self):
        missing_value_data = (self.data.isnull().sum()*100/len(self.d
ata)).reset_index()
        missing_value_data.columns = ["feature", "perc"]
        full_value_data = missing_value_data[missing_value_data["per
c"] == 0]
```

```
        missing_value_data = missing_value_data[missing_value_data["p
erc"] > 0]
        missing_value_data = missing_value_data.sort_values(by=['per
c'])
        return missing_value_data, full_value_data

    def missing_value_update_check(self, data):
        missing_value_data = (data.isnull().sum()*100/len(data)).rese
t_index()
        missing_value_data.columns = ["feature", "perc"]
        full_value_data = missing_value_data[missing_value_data["per
c"] == 0]
        missing_value_data = missing_value_data[missing_value_data["p
erc"] > 0]
        missing_value_data = missing_value_data.sort_values(by=['per
c'])
        print(missing_value_data.shape, full_value_data.shape)


    # Drops nan rows, making the data fully complete
    def create_full_data(self, data):
        data_without_nan = data.drop(pd.isnull(data).any(1).nonzero()
[0])
        return data_without_nan

    # Normalizing the data through min max. A dataframe is created to
store the
    # minimum and maximum values per column
    def min_max_scalar(self, data):
        d = {}
        for col in data.columns:
            if col not in d:
                d[col] = [min(data[col]), max(data[col])]

        df_min_max = pd.DataFrame.from_dict(d)
        return df_min_max

    # Applies the min and max values of each column to transform the
 data
    def scale_transform(self, data):
        for col in self.df_min_max.columns:
            min_val = self.df_min_max[col][0]
            max_val = self.df_min_max[col][1]
            if max_val != min_val:
                denom = (max_val - min_val)
            else:
                denom = 0.0001
            data[col] = (data[col] - min_val)/ denom
        return data

    # K-folding the dataset. This function makes the splits
    def k_fold(self, max_index, n_folds=10):
        n = max_index
        idxs = np.arange(n)
        fold_sizes = (n // n_folds) * np.ones(n_folds, dtype=np.int)
        fold_sizes[:n % n_folds] += 1
        current = 0
```

```
        splits = []
        for fold_size in fold_sizes:
            start, stop = current, current + fold_size
            val = idxs[start:stop]
            splits.append(list(val))
            current = stop

        return splits

    # A major function, which works on the splits created above, calc
ulates loss based
    # on train-validation sets, finally, the mean/mode of losses and
 predictions is taken
    def cross_validation(self, train_data, X_test, y_test, model, fea
ture_name):
        preds = []
        losses = []

        max_index = len(train_data[feature_name])
        k_fold_splits = self.k_fold(max_index, n_folds=10)

        if model == "mean_mode":
            for i in range(len(k_fold_splits)):
                k_copy = k_fold_splits.copy()
                del k_copy[i]
                val = k_fold_splits[i]
                train_i = list(itertools.chain.from_iterable(k_copy))
                val_data = train_data.iloc[val,:]
                new_train_data = train_data.iloc[train_i,:]

                X_train = new_train_data.drop(feature_name, axis=1)

                self.df_min_max = self.min_max_scalar(X_train)
                X_train = self.scale_transform(X_train)
                y_train = new_train_data[feature_name]

                X_val = val_data.drop(feature_name, axis=1)
                X_val = self.scale_transform(X_val)
                y_val = val_data[feature_name]

                y_pred = [np.nan]*len(y_val)
                y_pred = pd.DataFrame(y_pred)
                value = self.mean_mode.predict(feature_name, y_val)
                y_pred = y_pred.fillna(value)
                loss = self.mean_mode.get_mse(y_val, y_pred, feature_
name)
                preds.append(value)
                losses.append(loss)

            predicted_mean = np.mean(preds) if type(preds[0]) is floa
t else str(Counter(preds).most_common(1)[0][0])
            y_test = y_test.fillna(predicted_mean)

        else:
            for i in range(len(k_fold_splits)):
                k_copy = k_fold_splits.copy()
                del k_copy[i]
```

```
                val = k_fold_splits[i]
                train_i = list(itertools.chain.from_iterable(k_copy))
                val_data = train_data.iloc[val,:]
                new_train_data = train_data.iloc[train_i,:]

                X_train = new_train_data.drop(feature_name, axis=1)

                self.df_min_max = self.min_max_scalar(X_train)
                X_train = self.scale_transform(X_train)
                y_train = new_train_data[feature_name]

                X_val = val_data.drop(feature_name, axis=1)

                X_val = self.scale_transform(X_val)
                y_val = val_data[feature_name]

                X_train = np.asarray(X_train)
                y_train = np.asarray(y_train)
                X_val = np.asarray(X_val)
                y_val = np.asarray(y_val)

                X_test = self.scale_transform(X_test)
                if model == "linear":
                    self.lin_reg.fit(X_train, y_train, iterations=50)
                    y_pred = self.lin_reg.predict(X_val)
                    y_test = self.lin_reg.predict(X_test)
                    loss = self.lin_reg.get_mse(y_val, y_pred)

                elif model == "ridge":
                    self.ridge.fit(X_train, y_train)
                    y_pred = self.ridge.predict(X_val)
                    y_test = self.ridge.predict(X_test)
                    loss = self.ridge.get_mse(y_val, y_pred)

                elif model == "lasso":
                    self.lasso.fit(X_train, y_train)
                    y_pred = self.lasso.predict(X_val)
                    y_test = self.lasso.predict(X_test)
                    loss = self.lasso.get_mse(y_val, y_pred)

                elif model == "dt_reg":
                    self.dt_reg.fit(X_train, y_train)
                    y_pred = self.dt_reg.predict(X_val)
                    y_test = self.dt_reg.predict(X_test)
                    loss = self.dt_reg.get_error(y_val, y_pred)

                elif model == "knn":
                    if feature_name in self.all_columns[1]:
                        self.knn.problem = "classify"
                    else:
                        self.knn.problem = "regress"
                    self.knn.fit(X_train, y_train)
                    y_pred = self.knn.predict(X_val)
                    y_test = self.knn.predict(X_test)
                    loss = self.knn.evaluate(y_pred, y_val)

                elif model == "logistic":
```

```
                    if len(np.unique(y_train)) > 2:
                        self.m_logistic.fit(X_train, y_train)
                        y_pred = self.m_logistic.predict(X_val)
                        y_test = self.m_logistic.predict(X_test)
                        loss = self.m_logistic.get_accuracy(y_val, y_
pred)
                    else:
                        self.b_logistic.fit(X_train, y_train)
                        y_pred = self.b_logistic.predict(X_val)
                        y_test = self.b_logistic.predict(X_test)
                        loss = self.b_logistic.get_accuracy(y_val, y_
pred)

                elif model == "nn_reg":
                    self.nn_reg.fit(X_train, y_train)
                    y_pred = self.nn_reg.predict(X_val)
                    print(y_pred)
                    y_test = self.nn_reg.predict(X_test)
                    loss = self.nn_reg._mse_loss(y_pred, y_val)

                elif model == "adaboost":
                    self.adaboost.fit(X_train, y_train)
                    y_pred = self.adaboost.predict(X_val)
                    y_test = self.adaboost.predict(X_test)
                    loss = self.adaboost.get_accuracy(y_val, y_pred)

                losses.append(loss)
                preds.append(y_test)

        if feature_name in self.all_columns[1] and model != "mean_mod
e":
            preds = list(np.asarray(preds).flatten())
            y_test_mean = [max(preds, key = list(preds).count)]
        else:
            y_test_mean = np.mean(preds, axis=0)
        scaled_loss = np.mean(losses)
        scaled_loss = 100*abs((scaled_loss - min(losses))/(max(losses
) - min(losses)))
        if model == "mean_mode":
            return y_test, scaled_loss
        return pd.DataFrame(y_test_mean, columns=[feature_name]), sca
led_loss

    def train_test_data(self, feature_name):
        train_data = self.data[self.full_value_cols + [feature_name]]
        test_data = train_data[train_data[feature_name].isnull()]
        train_data = self.create_full_data(train_data)
        return train_data, test_data

    def impute_to_main_data(self, new_data, feature_name, max_train_p
oint, count_test_points):
        # print("Length of new data: ", len(new_data))
        indexes = self.data[feature_name].index[self.data[feature_nam
e].apply(np.isnan)]
        indexes_to_add = [l for l in range(max_train_point, max_train
_point+count_test_points)]
        # print("Indexes of main data: {}, Indexes of new data: {}".f
```

```
ormat(indexes, indexes_to_add))
        for index, index_add in zip(indexes, indexes_to_add):
            # print(index, self.data[feature_name].iloc[index], new_d
ata[feature_name].iloc[index_add])
            self.data[feature_name].iloc[index] = new_data[feature_na
me].iloc[index_add]
            # print(self.count_missing(self.data[feature_name]))

    def pearson_correlation(self, data, main_feature):
        # Inputs are dataframes
        mint = 1e-5
        columns_to_keep = []
        for col in data.columns:
            if col != main_feature:
                r = col, scipy.stats.pearsonr(data[main_feature], dat
a[col])[0]
                if r[1] > 0:
                    columns_to_keep.append(r[0])
        columns_to_keep.append(main_feature)
        data = data[columns_to_keep]
        return data

    def workflow(self):
        missing_value_data, full_value_data = self.missing_value_perc
()
        self.full_value_cols = list(full_value_data['feature'])

#         10, 30, 50, 70, 100
#         0, 10, 30, 50, 70
        subsets = [100]
        lags = [70]
        total_loss = []
        features = []
        self.empty_min_error = []
        self.dict_subset = {}
        for subset,lag in zip(subsets, lags):
            if subset not in self.dict_subset:
                self.dict_subset[subset] = {}
            # Finding columns which have missing value less than a su
bset value
            missing_columns = list(missing_value_data[(missing_value_
data["perc"] <= subset) & (missing_value_data["perc"] > lag)].feature
)
            print("####################################################
######################")
            print("Subset: {}\tNumber of missing value columns: {}".f
ormat(subset, len(missing_columns)))
            print("####################################################
######################")

            for feature_name in missing_columns:
                try:
                    if feature_name in
                    if feature_name not in self.dict_subset[subset]:
                        self.dict_subset[subset][feature_name] = {}

                    features.append(feature_name)
```

```
                    train_data, test_data = self.train_test_data(feat
ure_name)

                    # Feature selection using pearson correlation
#                     train_data = self.pearson_correlation(train_dat
a, feature_name)
#                     test_data = test_data[train_data.columns]

                    print(train_data.shape, test_data.shape)
                    # self.missing_value_update_check(self.data)
                    X_train = train_data.drop(feature_name, axis=1)
                    y_train = train_data[feature_name]
                    multi_class = train_data[feature_name].unique()
                    X_test = test_data.drop(feature_name, axis=1)
                    X_test_copy = X_test.copy()
                    y_test = test_data[feature_name]

                    problem_type = ''
                    if feature_name in self.all_columns[0]:
                        problem_type = "Regression"
                        eval_type = "Loss"
                        models_to_use = self.regression_models
                    if feature_name in self.all_columns[1]:
                        problem_type = "Classification"
                        eval_type = "Accuracy"
                        if len(multi_class) > 2:
                            models_to_use = self.classification_model
s_without_ada
                        else:
                            models_to_use = self.classification_model
s

                    main_loss = np.iinfo(np.int32(10)).max if eval_ty
pe == "Loss" else np.iinfo(np.int32(10)).min
                    best_y_test = None
                    best_model = ''
                    if problem_type not in self.dict_subset[subset][f
eature_name]:
                        self.dict_subset[subset][feature_name][proble
m_type] = {}
                    print("Feature Name: {}, Problem Type: {}, Full c
olumns: {}".format(feature_name, problem_type, len(self.full_value_co
ls)))

                    for model in models_to_use:
                        if model not in self.dict_subset[subset][feat
ure_name][problem_type]:
                            self.dict_subset[subset][feature_name][pr
oblem_type][model] = []

                        y_test, loss = self.cross_validation(train_da
ta, X_test_copy, y_test, model, feature_name)
                        print("Model: {}, {}: {}".format(model, eval_
type, loss))
                        self.dict_subset[subset][feature_name][proble
m_type][model] = loss
```

```
                    if eval_type == "Accuracy":
                        for key, val in self.dict_subset[subset][feat
ure_name][problem_type].items():
                            if val >= main_loss:
                                best_y_test = y_test
                                main_loss = val
                                best_model = key

                    if eval_type == "Loss":
                        for key, val in self.dict_subset[subset][feat
ure_name][problem_type].items():
                            if val <= main_loss:
                                best_y_test = y_test
                                main_loss = val
                                best_model = key

                    print("Best model for {} feature is {} with {} {}
".format(feature_name, best_model, eval_type, main_loss))
                    X_test = X_test.reset_index()
                    test_data = pd.concat([X_test, best_y_test], axis
=1)
                    if self.count_missing(test_data[feature_name]) !=
0:
                        self.empty_min_error.append(feature_name)
                    max_index = len(train_data[feature_name])
                    test_points = len(test_data[feature_name])
                    train_data = train_data.append(test_data, ignore_
index=True)
                    self.impute_to_main_data(train_data, feature_name
, max_index, test_points)
                    self.full_value_cols.append(feature_name)
                    total_loss.append(main_loss)
                except Exception as e:
                    self.empty_min_error.append(feature_name)
                    print("Column: {}, Error: {}".format(feature_name
, e))
```

In [39]:
```
pipeline = Pipeline(data, models, all_columns)
```

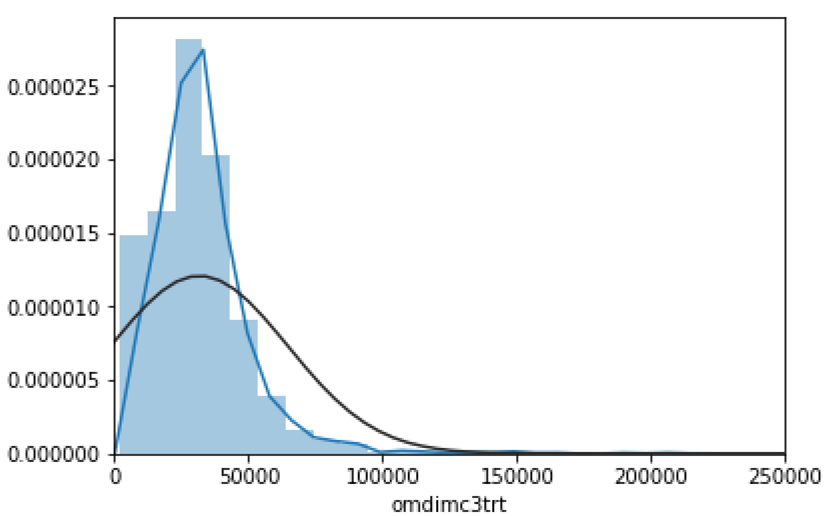
In [ ]:
```
pipeline.workflow()
```

In [ ]:
```
import csv

l = []
for subset, feature in pipeline.dict_subset.items():
    for feature, problem in feature.items():
        for problem, model in problem.items():
            for model, eval_ in model.items():
                l.append([subset, feature, problem, model, eval_])

final_csv = ["subset", "feature", "problem", "model", "eval"]

try:
    with open('results_1_without_pearson_subset_100','w') as csv_file
:
        writer = csv.writer(csv_file, delimiter=',')
        writer.writerow(final_csv)
        for line in l:
            writer.writerow(line)
except IOError:
    print("I/O error")
```

In [7]:
```
sns.distplot(data_without_nan['omdimc3trt'], bins=100, fit=norm)
plt.xlim(0, 250000)
# , data_without_nan['omdimc3rt'].astype('int32'), hue=data_without_n
an['omdimc3'])
```

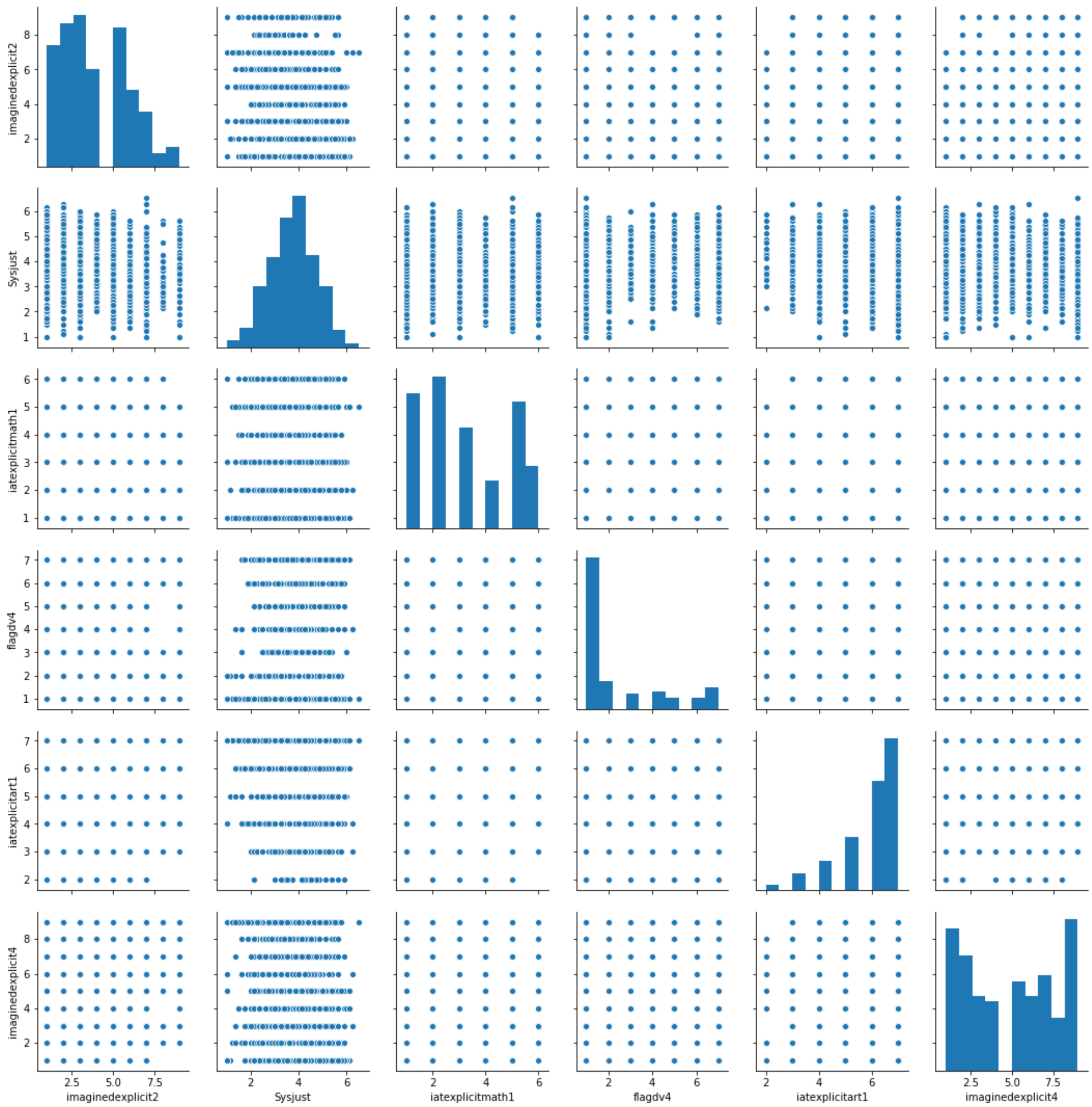
```
/home/vedantc6/anaconda3/lib/python3.7/site-packages/scipy/stats/stat
s.py:1713: FutureWarning: Using a non-tuple sequence for multidimensi
onal indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[se
q]`. In the future this will be interpreted as an array index, `arr[n
p.array(seq)]`, which will result either in an error or a different r
esult.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

Out[7]: (0, 250000)



In [8]:
```
sns.pairplot(data_without_nan.iloc[:,10:20])
```

Out[8]: <seaborn.axisgrid.PairGrid at 0x7f57f9088c50>



In [13]:
```python
corr = data.corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(30, 20))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})

plt.savefig('correlation.jpg')
```