```
# Load the Drive helper and mount
from google.colab import drive
# This will prompt for authorization.
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
```

```
import numpy as np
import pandas as pd
from prettytable import PrettyTable
import random
from math import floor
import matplotlib.pyplot as plt
```

```
#Standardize numerical Values
def standardize_col(col):
    return (col - col.min()) * 1.0 / (col.max() - col.min())
```

```
#order the columns of the dataframe according to decreasing number of null values, ret
def ordered_cols(df):
  x = df.isna().sum()
  x = x.sort_values()
  x = x.to_frame()
  columns_sorted = x.index.values
  return list(columns_sorted)
```

```
class DecisionTree(object):
    """
    Class to create decision tree model (CART)
    """
    def __init__(self, _max_depth, _min_splits):
        self.max_depth = _max_depth
        self.min_splits = _min_splits

    def fit(self, _feature, _label):
        """
        :param _feature:
        :param _label:
        :return:
        """
        self.feature = _feature
        self.label = _label
        self.train_data = np.column_stack((self.feature,self.label))
        self.build_tree()


    def compute_gini_similarity(self, groups, class_labels):
        """
        compute the gini index for the groups and class labels
        :param groups:
        :param class_labels:
        :return:
        """
        num_sample = sum([len(group) for group in groups])
        gini_score = 0

        for group in groups:
            size = float(len(group))

            if size == 0:
                continue
```

```
            score = 0.0
            for label in class_labels:
                porportion = (group[:,-1] == label).sum() / size
                score += porportion * porportion
            gini_score += (1.0 - score) * (size/num_sample)

        return gini_score

    def terminal_node(self, _group):
        """
        Function set terminal node as the most common class in the group to make predi
        is an helper function used to mark the leaf node in the tree based on the earl
        or actual stop condition which ever is meet early
        :param _group:
        :return:
        """
        class_labels, count = np.unique(_group[:,-1], return_counts= True)
        return class_labels[np.argmax(count)]

    def split(self, index, val, data):
        """
        split features into two groups based on their values
        :param index:
        :param val:
        :param data:
        :return:
        """
        data_left = np.array([]).reshape(0,self.train_data.shape[1])
        data_right = np.array([]).reshape(0, self.train_data.shape[1])

        for row in data:
            if row[index] <= val :
                data_left = np.vstack((data_left,row))

            if row[index] > val:
                data_right = np.vstack((data_right, row))

        return data_left, data_right

    def best_split(self, data):
        """
        find the best split information using the gini score
        :param data:
        :return best_split result dict:
        """
        class_labels = np.unique(data[:,-1])
        best_index = 999
        best_val = 999
        best_score = 999
        best_groups = None

        for idx in range(data.shape[1]-1):
            for row in data:
                groups = self.split(idx, row[idx], data)
                gini_score = self.compute_gini_similarity(groups,class_labels)

                if gini_score < best_score:
                    best_index = idx
                    best_val = row[idx]
                    best_score = gini_score
                    best_groups = groups
        result = {}
        result['index'] = best_index
        result['val'] = best_val
        result['groups'] = best_groups
        return result


    def split_branch(self, node, depth):
        """
```

```
        recursively split the data and
        check for early stop argument based on self.max_depth and self.min_splits
        - check if left or right groups are empty is yess craete terminal node
        - check if we have reached max_depth early stop condition if yes create termin
        - Consider left node, check if the group is too small using min_split conditic
            - if yes create terminal node
            - else continue to build the tree
        - same is done to the right side as well.
        else
        :param node:
        :param depth:
        :return:
        """
        left_node , right_node = node['groups']
        del(node['groups'])

        if not isinstance(left_node,np.ndarray) or not isinstance(right_node,np.ndarra
            node['left'] = self.terminal_node(left_node + right_node)
            node['right'] = self.terminal_node(left_node + right_node)
            return

        if depth >= self.max_depth:
            node['left'] = self.terminal_node(left_node)
            node['right'] = self.terminal_node(right_node)
            return

        if len(left_node) <= self.min_splits:
            node['left'] = self.terminal_node(left_node)
        else:
            node['left'] = self.best_split(left_node)
            self.split_branch(node['left'],depth + 1)


        if len(right_node) <= self.min_splits:
            node['right'] = self.terminal_node(right_node)
        else:
            node['right'] = self.best_split(right_node)
            self.split_branch(node['right'],depth + 1)

    def build_tree(self):
        """
        build tree recursively with help of split_branch function
         - Create a root node
         - call recursive split_branch to build the complete tree
        :return:
        """
        self.root = self.best_split(self.train_data)
        self.split_branch(self.root, 1)
        return self.root

    def _predict(self, node, row):
        """
        Recursively traverse through the tress to determine the
        class of unseen sample data point during prediction
        :param node:
        :param row:
        :return:
        """
        if row[node['index']] < node['val']:
            if isinstance(node['left'], dict):
                return self._predict(node['left'], row)
            else:
                return node['left']

        else:
            if isinstance(node['right'],dict):
                return self._predict(node['right'],row)
            else:
                return node['right']
```

```
    def predict(self, test_data):
        """
        predict the set of data point
        :param test_data:
        :return:
        """
        self.predicted_label = np.array([])
        for idx in test_data:
            self.predicted_label = np.append(self.predicted_label, self._predict(self.

        return self.predicted_label
```

```
def impute(df):
  #initial imputed values

  Z = np.matrix(df['previous_session_id']).reshape(-1,1)
  indexes_dict = {}

  #Get ordered list of col names except the session id which is always filled
  col_partial = ordered_cols(df.loc[:, df.columns != 'previous_session_id'])
  col_list = col_partial
#   print(col_partial)


  for col in col_partial:
  #   print(col)
    y  = df[col]
  #   print(type(y))
    pos = list(y[y.isnull()].index)
  #   print(pos)
    indexes_dict[col] = pos

    X_train = np.delete(Z, pos, axis=0)
    X_test = Z[pos, :]
    y_train = np.matrix(y.drop(y.index[pos])).reshape(Z.shape[0]-len(pos),1)

    y_final = np.matrix(y).reshape(Z.shape[0],1)

  #   print(y.values.reshape(Z.shape[0],1)[pos[0]])
  #   X_test = Z.loc[pos].values#.reshape(1,-1)
  #   y_train = (y.drop(y.index[pos])).values(columns = 1)
  #   X_train = (Z.drop(Z.index[pos])).values
  #   X_train = (X_train).values#.reshape(-1,1)

    clf = tree.DecisionTreeRegressor()
    clf = clf.fit(X_train, y_train)

    predicted = clf.predict(X_test)
  #   print(predicted)

    for i in range(len(pos)):
      ind = pos[i]
      y_final[ind] = predicted[i]

    Z = np.concatenate((Z,y_final),axis = 1)
  #   break

  #Converge 10 times
  for l in range(10):
    for colm in range(1,Z.shape[1]):
      column_name = col_partial[colm-1]
      pos = indexes_dict[column_name]

      X_train = np.delete(Z, colm, axis=1)
      X_train = np.delete(X_train, pos, axis=0)
      X_test = np.delete(Z[pos, :], colm, axis=1)
      y_train = Z[:, colm]
      y_train = np.delete(y_train, pos, axis=0)
```

```
  #     print(X_train.shape)
  #     print(X_test.shape)
  #     print(y_train.shape)

      clf = tree.DecisionTreeRegressor()
      clf = clf.fit(X_train, y_train)

      predicted = clf.predict(X_test)

      for i in range(len(pos)):
        ind = pos[i]
        Z[ind, colm] = predicted[i]

  col_list = ['previous_session_id'] + col_list
#   print(col_list)
  return pd.DataFrame(Z, columns=col_list)
```

```
!ls 'drive/My Drive'
```

```
data_num=pd.read_csv("/content/drive/My Drive/data_num.csv",encoding='unicode_escape')
num_samples, column_size = data_num.shape
```

```
for col in data_num.columns:
    data_num[col] = standardize_col(data_num[col])
```

```
Z1 = impute(data_num)
```

```
Z1 = impute(data_num)
Z2 = impute(data_num)
Z3 = impute(data_num)
```

```
data_imputed_copy1 = pd.DataFrame(np.nan, index = np.arange(6344), columns = data_num.
# data_imputed_copy
data_imputed_copy2 = pd.DataFrame(np.nan, index = np.arange(6344), columns = data_num.
# data_imputed_copy
data_imputed_copy3 = pd.DataFrame(np.nan, index = np.arange(6344), columns = data_num.
# data_imputed_copy
```

```
for col in data_num.columns:
#   print(data_num[col].isnull())
#   break
  data_imputed_copy1.loc[data_num[col].isnull(), col] = Z1[col]
  data_imputed_copy2.loc[data_num[col].isnull(), col] = Z2[col]
  data_imputed_copy3.loc[data_num[col].isnull(), col] = Z3[col]
```

```
x = PrettyTable()

x.field_names = ["Column", "N (observed samples)", "Mean (observed samples)", "SD (obs
```

```
import statistics
```

```
for col in data_num.columns:
  obs_mean = data_num[col].mean()
```

```
    obs_sd = data_num[col].std()
    obs_min = data_num[col].min()
    obs_max = data_num[col].max()

    obs_num = data_num[col].isna().sum()
    imp_num = num_samples - obs_num

    imp_mean = statistics.mean([data_imputed_copy1[col].mean(),data_imputed_copy2[col].m
    imp_sd = statistics.mean([data_imputed_copy1[col].std(),data_imputed_copy2[col].std(
    imp_min = min([data_imputed_copy1[col].min(),data_imputed_copy2[col].min(),data_impu
    imp_max = max([data_imputed_copy1[col].max(),data_imputed_copy2[col].max(),data_impu

    x.add_row([col, obs_num, obs_mean, obs_sd, obs_min, obs_max, imp_num, imp_mean, imp_


print(x)
```

```
+------------------------+----------------------+-------------------------+----
|         Column         | N (observed samples) | Mean (observed samples) | SD
+------------------------+----------------------+-------------------------+----
|        gambfalDV       |         402          |   0.11557210181262266   |   0
| numparticipants_actual |         6125         |    0.503196347031964    |  0.
|    numparticipants     |         3609         |    0.2761942611875044   |  0.
|        exprace         |         3399         |    0.3910809281267705   |  0.
|          age           |          16          |    0.1588179519595533   |  0.
|         sunkDV         |          14          |    0.8191943127962086   |   0
|       anchoring1       |         982          |    0.4310679644399094   |   0
|       anchoring2       |         1060         |   0.38262886964664994   |   0
|       anchoring3       |         717          |   0.48220178718826956   |   0
|       anchoring4       |         735          |    0.3084419368301509   |   0
|        Ranchori        |         982          |    0.5002806886227604   |   0
|         RAN001         |         1060         |    0.500807140822334    |   0
|         RAN002         |         717          |    0.5010246814577267   |   0
|         RAN003         |         735          |   0.49986572374899374   |   0
|         Ranch1         |         982          |    0.5002806886227604   |   0
|         Ranch2         |         1060         |    0.500807140822334    |   0
|         Ranch3         |         717          |    0.5010246814577267   |   0
```

```
data = x.get_string()

with open('/content/drive/My Drive/test.txt', 'w') as f:
    f.write(data)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-ac597eafae40> in <module>()
----> 1 data = x.get_string()
      2 
      3 with open('/content/drive/My Drive/test.txt', 'w') as f:
      4     f.write(data)

NameError: name 'x' is not defined
```

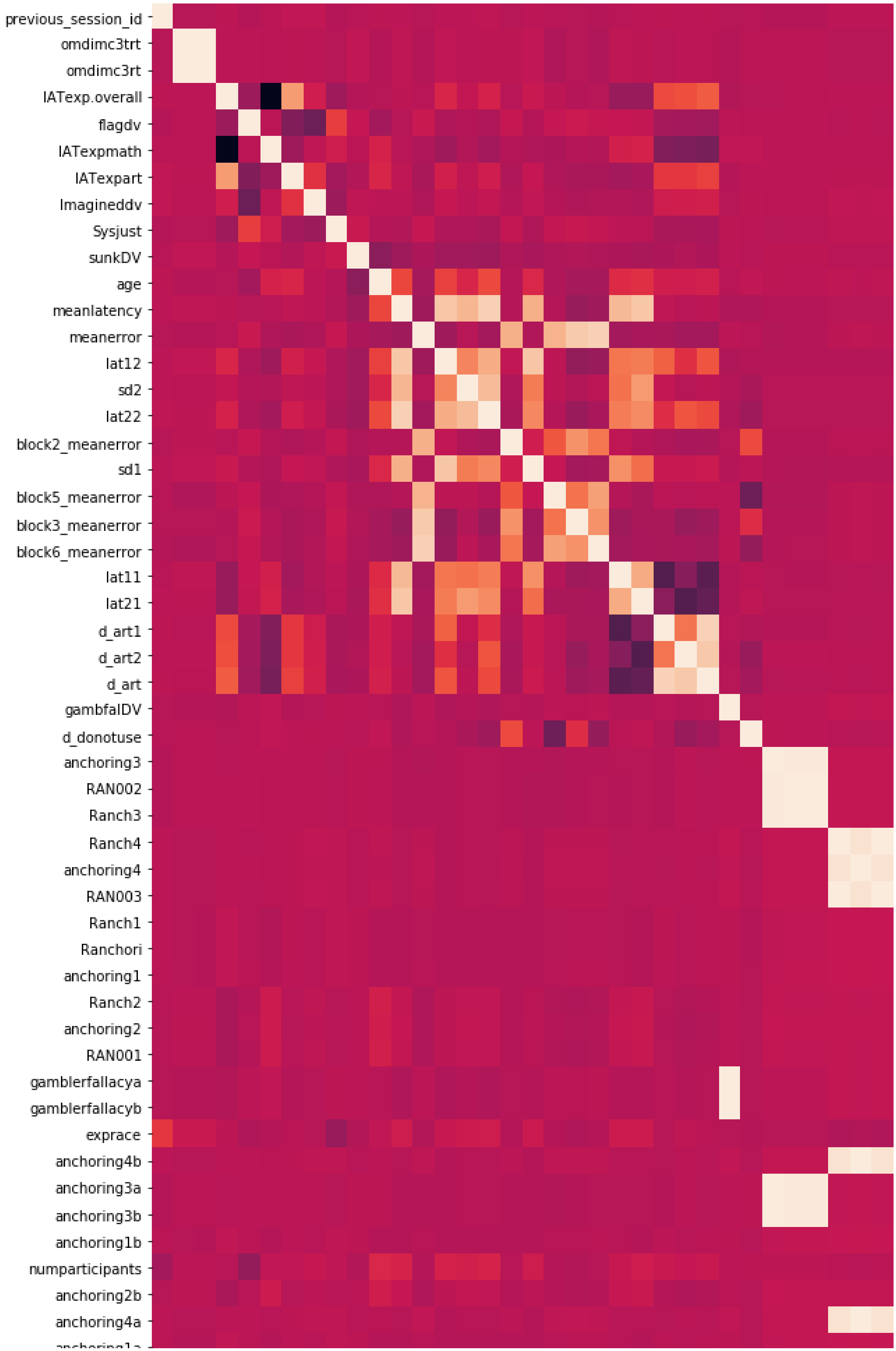
SEARCH STACK OVERFLOW

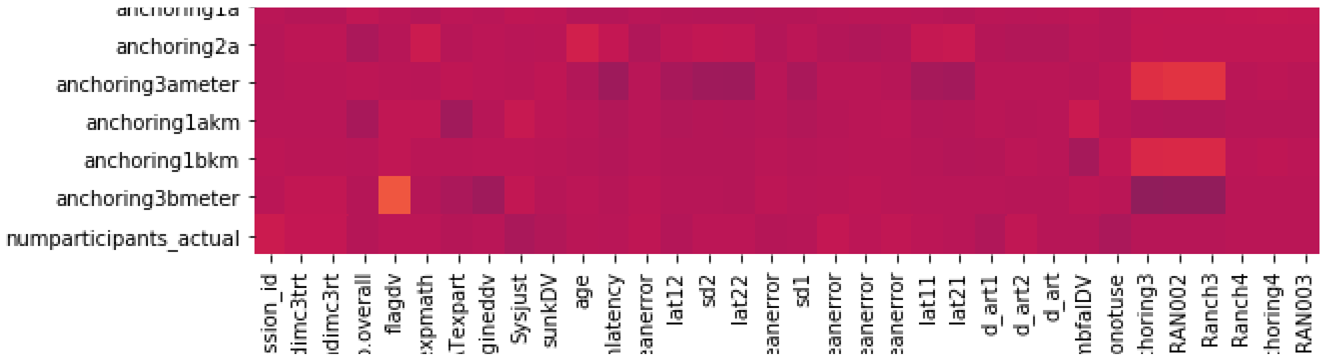
```
|     gamblerfallacyb     |         3379         |    0.0927946445334509   |  0.
```

```
import seaborn as sns
%matplotlib inline
```

```
corr = Z1.corr()
```

```
fig, ax = plt.subplots(figsize=(20,20))
sns.heatmap(corr,
        xticklabels=corr.columns,
        yticklabels=corr.columns, ax=ax)
plt.show()
fig.savefig('/content/drive/My Drive/correlation.jpg')
plt.close(fig)
```


previous_session_id
omdimc3trt
omdimc3rt
IATexp.overall
flagdv
IATexpmath
IATexpart
Imagineddv
Sysjust
sunkDV
age
meanlatency
meanerror
lat12
sd2
lat22
block2_meanerror
sd1
block5_meanerror
block3_meanerror
block6_meanerror
lat11
lat21
d_art1
d_art2
d_art
gambfalDV
d_donotuse
anchoring3
RAN002
Ranch3
Ranch4
anchoring4
RAN003
Ranch1
Ranchori
anchoring1
Ranch2
anchoring2
RAN001
gamblerfallacya
gamblerfallacyb
exprace
anchoring4b
anchoring3a
anchoring3b
anchoring1b
numparticipants
anchoring2b
anchoring4a

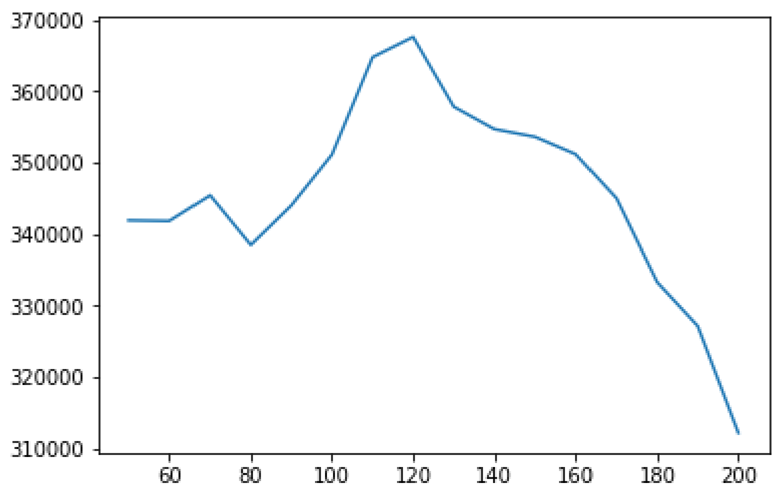



```
from sklearn.decomposition import PCA
pca = PCA(0.99, whiten=True)
data = pca.fit_transform(Z1)
data.shape
```

```
(6344, 27)
```

```
from sklearn.mixture import GaussianMixture
n_components = np.arange(50, 210, 10)
models = [GaussianMixture(n, covariance_type='full', random_state=0)
          for n in n_components]
aics = [model.fit(data).aic(data) for model in models]
plt.plot(n_components, aics);
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/mixture/base.py:273: Convergence
  % (init + 1), ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/mixture/base.py:273: Convergence
  % (init + 1), ConvergenceWarning)
```




```
gmm = GaussianMixture(80, covariance_type='full', random_state=0)
gmm.fit(Z1)
print(gmm.converged_)
```

```
True
```

```
data_new = gmm.sample()
data_new
```

```
(array([[-9.00800538e-02,  1.53337746e-02,  1.35316292e-02,
          7.98449569e-01,  3.32688144e-01,  3.01751874e-01,
          9.33357486e-01,  6.79805138e-01,  3.71515284e-01,
          9.81673048e-01,  1.05429232e-01,  2.43059296e-01,
          1.59419898e-02,  2.49703722e-01,  2.17511233e-01,
          2.39705274e-01,  1.14229825e-01,  4.23969239e-01,
          1.12015721e-01,  1.01366111e-02, -6.86125218e-02,
          4.14941094e-01,  1.57727871e-01,  4.51329923e-01,
          5.39976548e-01,  5.20315488e-01,  9.26194344e-02,
          4.46635941e-01,  5.79684188e-01,  5.49872596e-01,
          5.48433270e-01,  2.89071114e-01,  6.74206657e-02,
          2.91091594e-01,  5.96157922e-01,  5.93973964e-01,
          4.97163108e-01,  6.87147993e-01,  5.44079275e-01,
          6.84544227e-01,  9.30936905e-02,  9.50361072e-02,
          9.53314489e-01,  6.60599655e-02,  5.88596626e-01,
          5.73847839e-01,  4.92085247e-01,  2.96867999e-01,
          5.41865223e-01,  6.68823881e-02,  5.07064023e-01,
```

```
data_new = pca.inverse_transform(data_new)
```