

HW7\_Petrov.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

Comment Share Reconnect

+ Code + Text

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра математического моделирования и искусственного интеллекта

▼ ОТЧЕТ ПО КОНТРОЛЬНОЙ РАБОТЕ № 7

Дисциплина: Методы машинного обучения

Студент: Петров Артем Евгеньевич

Группа: НКНбд-01-21

Москва 2024

---

▼ Задание:

1. Загрузите заданный в индивидуальном задании набор данных с изображениями из Tensorflow Datasets с разбиением на обучающую и тестовую выборки. Оставьте в обучающей и тестовой выборках диапазон классов, указанных в индивидуальном задании. Если изображения цветные (с тремя каналами), то перекодируйте их в одноцветные (оттенки серого).
2. Постройте для набора данных график логарифмического правдоподобия профиля в зависимости от числа главных компонент и определите размерность латентного пространства.
3. Создайте и обучите на обучающей выборке автокодировщик архитектуры, указанной в индивидуальном задании, с размерностью скрытого представления, равной размерности латентного пространства, определенной в п.2. Подберите такие параметры, как функции активации, оптимизатор, начальная скорость обучения, размер мини-пакета и др. самостоятельно, обеспечивая обучение нейронных сетей. Визуализируйте несколько исходных и восстановленных автокодировщиком изображений.
4. Оцените качество модели автокодировщика на тестовой выборке по показателю, указанному в индивидуальном задании.
5. Оставьте в наборах изображения первых двух классов диапазона, указанного в индивидуальном задании первыми. Визуализируйте набор данных на плоскости, соответствующей двум первым латентным признакам, отображая точки различных классов разными цветами. Подпишите оси и рисунок, создайте легенду для классов набора данных.
6. Выполните бинарную классификацию изображений по латентным (скрытым) признакам и всем признакам при помощи классификатора метода ближайших соседей (kNN). Оцените бинарный классификатор, указанный в индивидуальном задании, для двух построенных классификаторов.
7. Визуализируйте ROC-кривые для построенных классификаторов на одном рисунке (с легендой) (Указание: используйте метод predict\_proba() класса KNeighborsClassifier).
8. Визуализируйте границы принятия решений классификатора kNN для латентных признаков на плоскости, соответствующей двум первым латентным признакам (для прочих латентных признаков задайте средние/медианные значения).
9. Определите на первоначальной тестовой выборке изображение, имеющее наибольшую ошибку реконструкции. Выведите для этого изображения первоначальное и реконструированное изображения.

---

Вариант 3

1. Набор данных: svhn\_cropped
2. Диапазон классов: 0, 1, 2, 3, 4
3. Архитектура автокодировщика: MLP
4. Показатель качества: среднее квадратичное логарифмическое отклонение (MSLE) для ошибки реконструкции
5. Показатель качества бинарной классификации: F1-мера, равная  $2TP/(2TP+FP+FN)$

---

1. Загрузите заданный в индивидуальном задании набор данных с изображениями из Tensorflow Datasets с разбиением на обучающую и тестовую выборки. Оставьте в обучающей и тестовой выборках диапазон классов, указанных в индивидуальном задании. Если изображения цветные (с тремя каналами), то перекодируйте их в одноцветные (оттенки серого).

▼

1. Набор данных: svhn\_cropped

2. Диапазон классов: 0, 1, 2, 3, 4

```
[ ] import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

[ ] ds_train = tfds.load('svhn_cropped', split = 'train')
```

Downloading and preparing dataset 1.47 GiB (download: 1.47 GiB, generated: Unknown size, total: 1.47 GiB) to /root/tensorflow\_datasets/svhn\_cropped/3.0.0...  
DL Completed... 100% [██████████] 3/3 [02:18<00:00, 60.21s/ url]  
DL Size... 100% [██████████] 1501/1501 [02:18<00:00, 10.04 MiB/s]

```
dataset svhn_cropped downloaded and prepared to /root/tensorflow_datasets/svhn_cropped/3.0.0. Subsequent calls will reuse this data.
```

```
[ ] ds_test = tfds.load('svhn_cropped', split = 'test')

[ ] df_train = tfds.as_dataframe(ds_train)

[ ] df_test = tfds.as_dataframe(ds_test)

[ ] df_train['label'].value_counts()

label
1    13861
2    10585
3     8497
4     7458
5     6882
6     5727
7     5595
8     5045
0     4948
9     4659
Name: count, dtype: int64

[ ] df_train = df_train[df_train['label'] < 5]

[ ] df_train['label'].value_counts()

label
1    13861
2    10585
3     8497
4     7458
0     4948
Name: count, dtype: int64

[ ] df_test = df_test[df_test['label'] < 5]

[ ] df_test['label'].value_counts()

label
1     5099
2     4149
3     2882
4     2523
0     1744
Name: count, dtype: int64

[ ] from PIL import Image, ImageOps

[ ] df_train['image'] = df_train['image'].apply(lambda x: np.array(ImageOps.grayscale((Image.fromarray(x)))) )

<ipython-input-12-25ab07e78023>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy
df_train['image'] = df_train['image'].apply(lambda x: np.array(ImageOps.grayscale((Image.fromarray(x)))) )

[ ] df_train['image'][0]

ndarray (32, 32) show data
 A small square grayscale image showing a house, labeled with a large number 1 below it.

[ ] df_test['image'] = df_test['image'].apply(lambda x: np.array(ImageOps.grayscale((Image.fromarray(x)))) )

<ipython-input-14-39d25263a5ed>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy
df_test['image'] = df_test['image'].apply(lambda x: np.array(ImageOps.grayscale((Image.fromarray(x)))) )

[ ] df_test['image'][2]

ndarray (32, 32) show data
 A small square grayscale image showing a house, labeled with a large number 2 below it.
```

## 2. Постройте для набора данных график логарифмического правдоподобия

- ✓ профиля в зависимости от числа главных компонент и определите размерность латентного пространства.

```
[ ] from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error
from scipy.stats import multivariate_normal
import warnings
warnings.simplefilter("ignore", RuntimeWarning)

[ ] def log_likelihood(evals):
    Lmax = len(evals)
    l1 = np.arange(0,0, Lmax)

    for L in range(Lmax):
        group1 = evals[0: L+1]
        group2 = evals[L+1: Lmax]

        mu1 = np.mean(group1)
        mu2 = np.mean(group2)

        sigma = (np.sum((group1 - mu1) ** 2) + np.sum((group2 - mu2) ** 2))/ Lmax
        l1_group1 = np.sum(np.sum(multivariate_normal.logpdf(group1, mu1, sigma)))
```

```

l1_group2 = np.sum(np.sum(multivariate_normal.logpdf(group2, mu2, sigma)))

l1[L] = l1_group1 + l1_group2

return l1

[ ] df_test['image'].values.shape[0], df_test['image'].values[0].shape[1],
→ (16397, 32)

❶ def unstack(vals: np.ndarray):
    X = np.empty(shape = (vals.shape[0], vals[0].shape[0], vals[0].shape[1]))
    for i in range(X.shape[0]):
        X[i] = vals[i]

    return X

[ ] X_train = unstack(df_train['image'].values)
X_test = unstack(df_test['image'].values)

[ ] X_train.shape, X_test.shape
→ ((45349, 32, 32), (16397, 32, 32))

[ ] x_pca = np.reshape(X_train, (X_train.shape[0], (X_train.shape[1] * X_train.shape[2])))

[ ] n_samples, n_features = x_pca.shape
Kmax = min(n_samples, n_features)

pca = PCA(n_components = Kmax)
X_transformed = pca.fit_transform(x_pca)

[ ] evals = pca.explained_variance_
l1 = log_likelihood(evals)

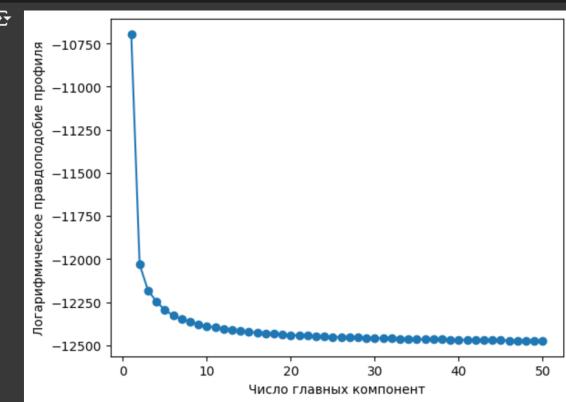
fraction_var = np.cumsum(evals[:50] / np.sum(evals))

[ ] fig, ax = plt.subplots()
xs = np.arange(1, 51)
ys = l1[:50]

plt.xlabel("Число главных компонент")
plt.ylabel("Логарифмическое правдоподобие профиля")

ax.plot(xs, ys, marker = "o")
plt.show()

```



3. Создайте и обучите на обучающей выборке автокодировщик архитектуры, указанной в индивидуальном задании, с размерностью скрытого представления, равной размерности латентного пространства, определенной в п.2. Подберите такие параметры, как функции активации, оптимизатор, начальная скорость обучения, размер мини-пакета и др. самостоятельно, обеспечивая обучение нейронных сетей. Визуализируйте несколько исходных и восстановленных автокодировщиком изображений.

3. Архитектура автокодировщика: MLP

4. Показатель качества: среднее квадратичное логарифмическое отклонение (MSLE) для ошибки реконструкции

```

[ ] X_train_MLP = np.reshape(X_train, (X_train.shape[0], (X_train.shape[1] * X_train.shape[2])))

[ ] X_test_MLP = np.reshape(X_test, (X_test.shape[0], (X_test.shape[1] * X_test.shape[2])))

[ ] X_train_MLP.shape, X_test_MLP.shape
→ ((45349, 1024), (16397, 1024))

[ ] X_train_MLP = X_train_MLP / 255
X_test_MLP = X_test_MLP / 255

[ ] X_train_MLP[0]

```

```

array([0.5254902 , 0.52941176, 0.53333333, ..., 0.40784314, 0.30980392,
       0.20392157])

[ ] X_test_MLP[0]

array([0.7372549 , 0.74509804, 0.76470588, ..., 0.81176471, 0.80784314,
       0.80784314])

[ ] def create_autoencoders_simple( feature_layer_dim = 16):
    input_img = tf.keras.layers.Input(shape = (1024), name = 'Input_Layer')

    encoded = tf.keras.layers.Dense(feature_layer_dim, activation = 'relu', name = 'Encoded_Features')(input_img)
    decoded = tf.keras.layers.Dense(1024, activation = 'sigmoid', name = 'Decoded_Input')(encoded)

    autoencoder = tf.keras.Model(input_img, decoded)
    encoder = tf.keras.Model(input_img, encoded)

    encoded_input = tf.keras.layers.Input(shape = (feature_layer_dim, ))
    decoder = autoencoder.layers[-1]
    decoder = tf.keras.Model(encoded_input, decoder(encoded_input))

    return autoencoder, encoder, decoder

[ ] autoencoder16_simple, encoder16_simple, decoder16_simple = create_autoencoders_simple(16)

[ ] autoencoder16_simple.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 0.0001),
                                loss = tf.keras.losses.MeanSquaredError(),
                                metrics = [tf.keras.metrics.MeanSquaredLogarithmicError()]
                               )

[ ] autoencoder16_simple_hist = autoencoder16_simple.fit(
    X_train_MLP,
    X_train_MLP,
    epochs = 30,
    batch_size = 128,
    shuffle = True,
    validation_data = (X_test_MLP, X_test_MLP),
    verbose = 1
)

Epoch 1/30
355/355 [=====] - 4s 9ms/step - loss: 0.0405 - mean_squared_logarithmic_error: 0.0196 - val_loss: 0.0455 - val_mean_squared_logarithmic_error: 0.0219
Epoch 2/30
355/355 [=====] - 3s 8ms/step - loss: 0.0351 - mean_squared_logarithmic_error: 0.0169 - val_loss: 0.0362 - val_mean_squared_logarithmic_error: 0.0176
Epoch 3/30
355/355 [=====] - 3s 7ms/step - loss: 0.0312 - mean_squared_logarithmic_error: 0.0151 - val_loss: 0.0330 - val_mean_squared_logarithmic_error: 0.0160
Epoch 4/30
355/355 [=====] - 4s 12ms/step - loss: 0.0291 - mean_squared_logarithmic_error: 0.0141 - val_loss: 0.0310 - val_mean_squared_logarithmic_error: 0.0150
Epoch 5/30
355/355 [=====] - 3s 7ms/step - loss: 0.0274 - mean_squared_logarithmic_error: 0.0133 - val_loss: 0.0291 - val_mean_squared_logarithmic_error: 0.0140
Epoch 6/30
355/355 [=====] - 3s 8ms/step - loss: 0.0256 - mean_squared_logarithmic_error: 0.0124 - val_loss: 0.0271 - val_mean_squared_logarithmic_error: 0.0131
Epoch 7/30
355/355 [=====] - 3s 8ms/step - loss: 0.0238 - mean_squared_logarithmic_error: 0.0115 - val_loss: 0.0256 - val_mean_squared_logarithmic_error: 0.0124
Epoch 8/30
355/355 [=====] - 3s 9ms/step - loss: 0.0220 - mean_squared_logarithmic_error: 0.0107 - val_loss: 0.0233 - val_mean_squared_logarithmic_error: 0.0113
Epoch 9/30
355/355 [=====] - 4s 10ms/step - loss: 0.0202 - mean_squared_logarithmic_error: 0.0098 - val_loss: 0.0218 - val_mean_squared_logarithmic_error: 0.0104
Epoch 10/30
355/355 [=====] - 3s 8ms/step - loss: 0.0184 - mean_squared_logarithmic_error: 0.0090 - val_loss: 0.0195 - val_mean_squared_logarithmic_error: 0.0095
Epoch 11/30
355/355 [=====] - 3s 8ms/step - loss: 0.0167 - mean_squared_logarithmic_error: 0.0081 - val_loss: 0.0176 - val_mean_squared_logarithmic_error: 0.0085
Epoch 12/30
355/355 [=====] - 3s 8ms/step - loss: 0.0150 - mean_squared_logarithmic_error: 0.0073 - val_loss: 0.0162 - val_mean_squared_logarithmic_error: 0.0077
Epoch 13/30
355/355 [=====] - 5s 13ms/step - loss: 0.0133 - mean_squared_logarithmic_error: 0.0065 - val_loss: 0.0139 - val_mean_squared_logarithmic_error: 0.0069
Epoch 14/30
355/355 [=====] - 3s 7ms/step - loss: 0.0118 - mean_squared_logarithmic_error: 0.0058 - val_loss: 0.0127 - val_mean_squared_logarithmic_error: 0.0061
Epoch 15/30
355/355 [=====] - 3s 7ms/step - loss: 0.0106 - mean_squared_logarithmic_error: 0.0052 - val_loss: 0.0113 - val_mean_squared_logarithmic_error: 0.0055
Epoch 16/30
355/355 [=====] - 3s 8ms/step - loss: 0.0096 - mean_squared_logarithmic_error: 0.0047 - val_loss: 0.0103 - val_mean_squared_logarithmic_error: 0.0050
Epoch 17/30
355/355 [=====] - 4s 10ms/step - loss: 0.0088 - mean_squared_logarithmic_error: 0.0043 - val_loss: 0.0094 - val_mean_squared_logarithmic_error: 0.0046
Epoch 18/30
355/355 [=====] - 4s 10ms/step - loss: 0.0082 - mean_squared_logarithmic_error: 0.0040 - val_loss: 0.0088 - val_mean_squared_logarithmic_error: 0.0043
Epoch 19/30
355/355 [=====] - 3s 8ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0084 - val_mean_squared_logarithmic_error: 0.0040
Epoch 20/30
355/355 [=====] - 3s 8ms/step - loss: 0.0074 - mean_squared_logarithmic_error: 0.0036 - val_loss: 0.0080 - val_mean_squared_logarithmic_error: 0.0038
Epoch 21/30
355/355 [=====] - 3s 8ms/step - loss: 0.0071 - mean_squared_logarithmic_error: 0.0034 - val_loss: 0.0077 - val_mean_squared_logarithmic_error: 0.0037
Epoch 22/30
355/355 [=====] - 4s 12ms/step - loss: 0.0069 - mean_squared_logarithmic_error: 0.0033 - val_loss: 0.0074 - val_mean_squared_logarithmic_error: 0.0035
Epoch 23/30
355/355 [=====] - 3s 8ms/step - loss: 0.0068 - mean_squared_logarithmic_error: 0.0033 - val_loss: 0.0072 - val_mean_squared_logarithmic_error: 0.0035
Epoch 24/30
355/355 [=====] - 3s 8ms/step - loss: 0.0066 - mean_squared_logarithmic_error: 0.0032 - val_loss: 0.0071 - val_mean_squared_logarithmic_error: 0.0034
Epoch 25/30
355/355 [=====] - 3s 8ms/step - loss: 0.0065 - mean_squared_logarithmic_error: 0.0031 - val_loss: 0.0070 - val_mean_squared_logarithmic_error: 0.0033
Epoch 26/30
355/355 [=====] - 4s 11ms/step - loss: 0.0065 - mean_squared_logarithmic_error: 0.0031 - val_loss: 0.0069 - val_mean_squared_logarithmic_error: 0.0033
Epoch 27/30
355/355 [=====] - 3s 9ms/step - loss: 0.0064 - mean_squared_logarithmic_error: 0.0031 - val_loss: 0.0068 - val_mean_squared_logarithmic_error: 0.0032
Epoch 28/30
355/355 [=====] - 3s 7ms/step - loss: 0.0063 - mean_squared_logarithmic_error: 0.0030 - val_loss: 0.0067 - val_mean_squared_logarithmic_error: 0.0032
Epoch 29/30
355/355 [=====] - 3s 8ms/step - loss: 0.0063 - mean_squared_logarithmic_error: 0.0030 - val_loss: 0.0067 - val_mean_squared_logarithmic_error: 0.0032

[ ] encoded_imgs_simple = encoder16_simple.predict(X_test_MLP)
decoded_imgs_simple = decoder16_simple.predict(encoded_imgs_simple)

513/513 [=====] - 1s 2ms/step
513/513 [=====] - 1s 2ms/step

[ ] def image_show(orig, dec, fname = None):
    n = 10
    fig = plt.figure(figsize = (20, 4))

    for i in range(n):
        ax = plt.subplot(2, n, i+1)
        plt.imshow(orig[i].reshape(32, 32))
        plt.gray()

        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

```

```

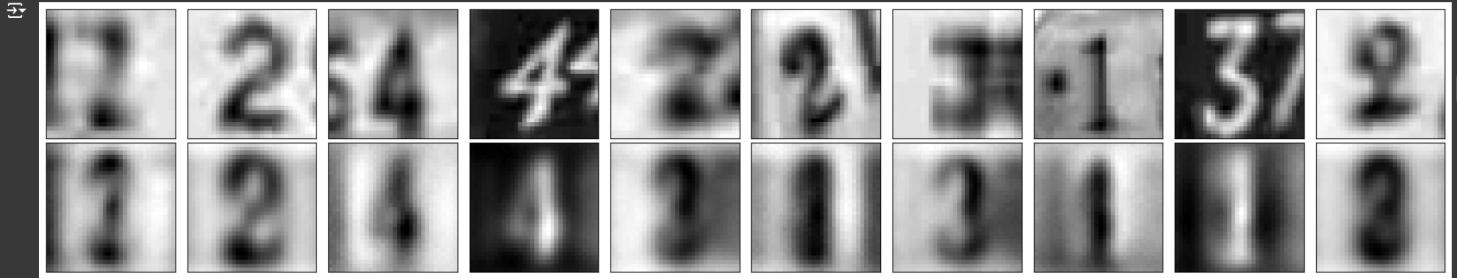
ax = plt.subplot(2, n, i+1+n)
plt.imshow(dec[i].reshape(32, 32))
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.tight_layout()
plt.show()

if fname:
    fig.savefig(fname)

```

```
[ ] image_show(X_test_MLP, decoded_imgs_simple)
```



Как мы видим, несмотря на то, что нейросеть обучилась, она плохо декодирует изображения(см картинки 3, 4, 5, 6, 9)

```

[ ] # def create_autoencoders (feature_layer_dim = 16):
#     input_img = Input(shape = (784,), name = 'Input_Layer')
#     # Слой encoded имеет размерность, равную feature_layer_dim
#     # и содержит закодированные входные данные
#     encoded = Dense(feature_layer_dim, activation = 'relu',
#                     name = 'Encoded_Features')(input_img)
#     decoded = Dense(784, activation = 'sigmoid',
#                     name = 'Decoded_Input')(encoded)

#     autoencoder = Model(input_img, decoded)
#     encoder = Model(input_img, encoded)

#     encoded_input = Input(shape = (feature_layer_dim,))
#     decoder = autoencoder.layers[-1]
#     decoder = Model(encoded_input, decoder(encoded_input))

#     return autoencoder, encoder, decoder

[ ] def create_autoencoders ( feature_layer_dim = 16, resolution = 1024):
    input_img = tf.keras.layers.Input(shape = (resolution,), name = 'Input_Layer')

    encoded = tf.keras.layers.Dense(512, activation = 'relu', name = 'Encoded_Features1')(input_img)
    encoded = tf.keras.layers.Dense(256, activation = 'relu', name = 'Encoded_Features2')(encoded)
    encoded = tf.keras.layers.Dense(128, activation = 'relu', name = 'Encoded_Features3')(encoded)
    encoded = tf.keras.layers.Dense(64, activation = 'relu', name = 'Encoded_Features4')(encoded)
    encoded = tf.keras.layers.Dense(feature_layer_dim, activation = 'relu', name = 'Encoded_Features5')(encoded)

    input_decode = tf.keras.layers.Input(shape = (feature_layer_dim, ), name = "decoder input")
    decoded = tf.keras.layers.Dense(64, activation = 'relu', name = 'Decoded_Features1')(encoded)
    decoded = tf.keras.layers.Dense(128, activation = 'relu', name = 'Decoded_Features2')(decoded)
    decoded = tf.keras.layers.Dense(256, activation = 'relu', name = 'Decoded_Features3')(decoded)
    decoded = tf.keras.layers.Dense(512, activation = 'relu', name = 'Decoded_Features4')(decoded)
    decoded = tf.keras.layers.Dense(resolution, activation = 'sigmoid', name = 'Decoded_Features5')(decoded)

    autoencoder = tf.keras.Model(input_img, decoded)
    encoder = tf.keras.Model(input_img, encoded)

    # encoded_input = tf.keras.layers.Input(shape = (feature_layer_dim, ))
    decoder1 = autoencoder.layers[-5](input_decode)
    decoder1 = autoencoder.layers[-4](decoder1)
    decoder1 = autoencoder.layers[-3](decoder1)
    decoder1 = autoencoder.layers[-2](decoder1)
    decoder1 = autoencoder.layers[-1](decoder1)
    decoder = tf.keras.Model(input_decode, decoder1)

    return autoencoder, encoder, decoder

[ ] autoencoder16, encoder16, decoder16 = create_autoencoders(feature_layer_dim=16, resolution=1024)

[ ] autoencoder16.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 0.0001),
                        loss = tf.keras.losses.MeanSquaredError(),
                        metrics = [tf.keras.metrics.MeanSquaredLogarithmicError()]
                      )

[ ] autoencoder16_hist = autoencoder16.fit(
    X_train_MLP,
    X_train_MLP,
    epochs = 30,
    batch_size = 128,
    shuffle = True,
    validation_data = (X_test_MLP, X_test_MLP),
    verbose = 1
)

Epoch 1/30
355/355 [=====] - 20s 50ms/step - loss: 0.0262 - mean_squared_logarithmic_error: 0.0127 - val_loss: 0.0150 - val_mean_squared_logarithmic_error: 0.0071
Epoch 2/30
355/355 [=====] - 18s 50ms/step - loss: 0.0132 - mean_squared_logarithmic_error: 0.0064 - val_loss: 0.0135 - val_mean_squared_logarithmic_error: 0.0063
Epoch 3/30
355/355 [=====] - 17s 49ms/step - loss: 0.0118 - mean_squared_logarithmic_error: 0.0057 - val_loss: 0.0123 - val_mean_squared_logarithmic_error: 0.0059
Epoch 4/30
355/355 [=====] - 17s 49ms/step - loss: 0.0114 - mean_squared_logarithmic_error: 0.0055 - val_loss: 0.0113 - val_mean_squared_logarithmic_error: 0.0053
Epoch 5/30
355/355 [=====] - 21s 59ms/step - loss: 0.0101 - mean_squared_logarithmic_error: 0.0049 - val_loss: 0.0108 - val_mean_squared_logarithmic_error: 0.0051
Epoch 6/30

```

```

355/355 [=====] - 16s 46ms/step - loss: 0.0100 - mean_squared_logarithmic_error: 0.0048 - val_loss: 0.0107 - val_mean_squared_logarithmic_error: 0.0051
Epoch 7/30
355/355 [=====] - 16s 46ms/step - loss: 0.0099 - mean_squared_logarithmic_error: 0.0047 - val_loss: 0.0104 - val_mean_squared_logarithmic_error: 0.0049
Epoch 8/30
355/355 [=====] - 17s 49ms/step - loss: 0.0094 - mean_squared_logarithmic_error: 0.0045 - val_loss: 0.0098 - val_mean_squared_logarithmic_error: 0.0047
Epoch 9/30
355/355 [=====] - 17s 49ms/step - loss: 0.0092 - mean_squared_logarithmic_error: 0.0044 - val_loss: 0.0098 - val_mean_squared_logarithmic_error: 0.0046
Epoch 10/30
355/355 [=====] - 20s 56ms/step - loss: 0.0092 - mean_squared_logarithmic_error: 0.0044 - val_loss: 0.0098 - val_mean_squared_logarithmic_error: 0.0046
Epoch 11/30
355/355 [=====] - 16s 46ms/step - loss: 0.0091 - mean_squared_logarithmic_error: 0.0044 - val_loss: 0.0097 - val_mean_squared_logarithmic_error: 0.0046
Epoch 12/30
355/355 [=====] - 17s 48ms/step - loss: 0.0091 - mean_squared_logarithmic_error: 0.0044 - val_loss: 0.0096 - val_mean_squared_logarithmic_error: 0.0046
Epoch 13/30
355/355 [=====] - 17s 49ms/step - loss: 0.0089 - mean_squared_logarithmic_error: 0.0043 - val_loss: 0.0092 - val_mean_squared_logarithmic_error: 0.0044
Epoch 14/30
355/355 [=====] - 17s 49ms/step - loss: 0.0083 - mean_squared_logarithmic_error: 0.0040 - val_loss: 0.0085 - val_mean_squared_logarithmic_error: 0.0041
Epoch 15/30
355/355 [=====] - 18s 50ms/step - loss: 0.0080 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0087 - val_mean_squared_logarithmic_error: 0.0040
Epoch 16/30
355/355 [=====] - 17s 48ms/step - loss: 0.0079 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0084 - val_mean_squared_logarithmic_error: 0.0040
Epoch 17/30
355/355 [=====] - 16s 46ms/step - loss: 0.0079 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0085 - val_mean_squared_logarithmic_error: 0.0040
Epoch 18/30
355/355 [=====] - 17s 49ms/step - loss: 0.0079 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0084 - val_mean_squared_logarithmic_error: 0.0040
Epoch 19/30
355/355 [=====] - 18s 50ms/step - loss: 0.0079 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0084 - val_mean_squared_logarithmic_error: 0.0040
Epoch 20/30
355/355 [=====] - 17s 49ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0083 - val_mean_squared_logarithmic_error: 0.0039
Epoch 21/30
355/355 [=====] - 16s 45ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0083 - val_mean_squared_logarithmic_error: 0.0039
Epoch 22/30
355/355 [=====] - 17s 49ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0083 - val_mean_squared_logarithmic_error: 0.0039
Epoch 23/30
355/355 [=====] - 16s 46ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0038 - val_loss: 0.0084 - val_mean_squared_logarithmic_error: 0.0039
Epoch 24/30
355/355 [=====] - 17s 48ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0083 - val_mean_squared_logarithmic_error: 0.0039
Epoch 25/30
355/355 [=====] - 17s 49ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0083 - val_mean_squared_logarithmic_error: 0.0040
Epoch 26/30
355/355 [=====] - 18s 51ms/step - loss: 0.0078 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0082 - val_mean_squared_logarithmic_error: 0.0039
Epoch 27/30
355/355 [=====] - 17s 49ms/step - loss: 0.0077 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0082 - val_mean_squared_logarithmic_error: 0.0039
Epoch 28/30
355/355 [=====] - 17s 48ms/step - loss: 0.0077 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0085 - val_mean_squared_logarithmic_error: 0.0040
Epoch 29/30
355/355 [=====] - 18s 49ms/step - loss: 0.0077 - mean_squared_logarithmic_error: 0.0037 - val_loss: 0.0082 - val_mean_squared_logarithmic_error: 0.0039

```

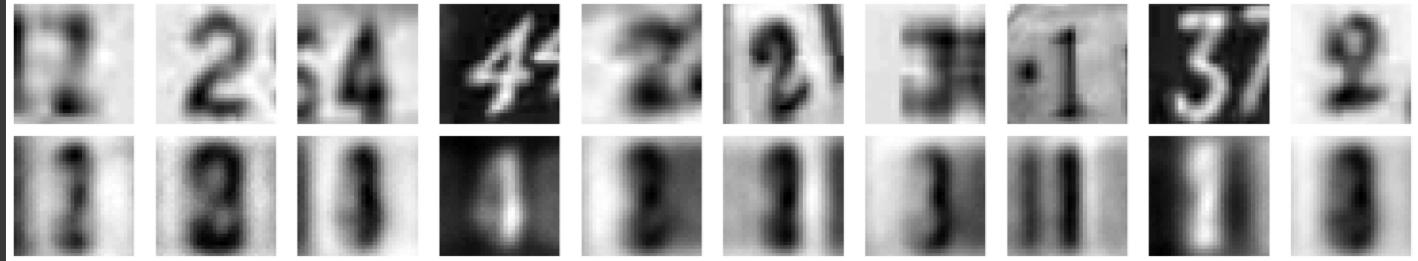
```
[ ] encoded_imgs = encoder16.predict(X_test_MLP)
```

513/513 [=====] - 4s 8ms/step

```
[ ] decoded_imgs = decoder16.predict(encoded_imgs)
```

513/513 [=====] - 2s 3ms/step

```
[ ] image_show(X_test_MLP, decoded_imgs)
```

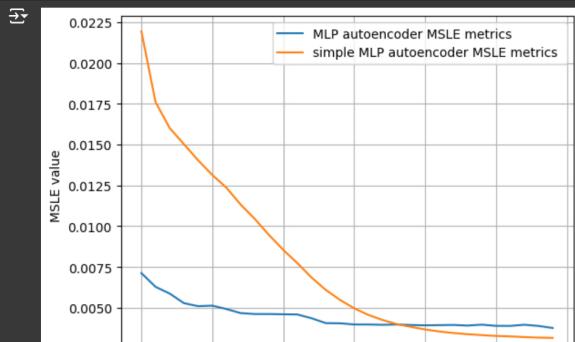


К сожалению, тут тоже не похоже на хорошую декодировку. Однако, двузначные числа она определяет явно лучше.

4. Оцените качество модели автокодировщика на тестовой выборке по показателю, указанному в индивидуальном задании.

Показатель качества – MSLE

```
[ ] plt.plot(autoencoder16_hist.history['val_mean_squared_logarithmic_error'], label='MLP autoencoder MSLE metrics')
# plt.plot(autoencoder16_simple_hist.history['val_mean_squared_logarithmic_error'], label='simple MLP autoencoder MSLE metrics ')
plt.ylim([0, max(history.history['loss'])*0.5])
plt.xlabel('Эпохи обучения')
plt.ylabel('MSLE value')
plt.legend()
plt.grid(True)
```





Однозначно, обе модели обучились примерно одинаково. По наблюдениям, лучше справилась простая MLP-модель. Будем наблюдать далее.

5. Оставьте в наборах изображения первых двух классов диапазона, указанного в индивидуальном задании первыми. Визуализируйте набор данных на плоскости, соответствующей двум первым латентным признакам, отображая точки различных классов различными цветами. Подпишите оси и рисунок, создайте легенду для классов набора данных.

2. Диапазон классов: 0, 1, 2, 3, 4

```
[ ] df_train['label'].value_counts()

label
1    13861
2    10585
3     8497
4     7458
0     4948
Name: count, dtype: int64

[ ] X_train = unstack((df_train[df_train['label'] < 2])['image'].values)

[ ] X_train = np.reshape(X_train, (X_train.shape[0], (X_train.shape[1]*X_train.shape[2])))

[ ] X_train.shape
→ (18809, 1024)

[ ] X_test = unstack((df_test[df_test['label'] < 2])['image'].values)

[ ] X_test = np.reshape(X_test, (X_test.shape[0], (X_test.shape[1]*X_test.shape[2])))

[ ] X_test.shape
→ (6843, 1024)

[ ] n_samples, n_features = X_train.shape
Kmax = min(n_samples, n_features)

pca = PCA(n_components = Kmax)
X_transformed = pca.fit_transform(X_train)

evals = pca.explained_variance_
ll = log_likelihood(evals)
fraction_var = np.cumsum(evals[:50]/np.sum(evals))

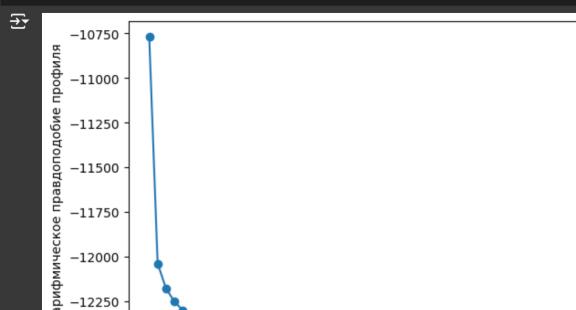
[ ] ll.shape
→ (1024,)

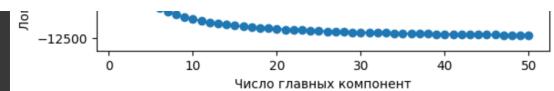
[ ] fraction_var
→ array([-0.59093572,  0.66025521,  0.72066377,  0.75865057,  0.78235859,
       0.80200925,  0.82066198,  0.8358721 ,  0.84933979,  0.85988334,
       0.868657 ,  0.87530839,  0.88107701,  0.88638968,  0.89118928,
       0.8956638 ,  0.89978352,  0.98371993,  0.98739715,  0.91043884,
       0.9135173 ,  0.91639875,  0.91924307,  0.92204991,  0.92468351,
       0.92711381,  0.92942799,  0.9316658 ,  0.93381269,  0.9359568 ,
       0.93883835,  0.93998474,  0.94184121,  0.94359987,  0.94528572,
       0.94691186,  0.94844187,  0.94992529,  0.95137922,  0.95275596,
       0.95487969,  0.95536552,  0.95659754,  0.95777969 ,  0.95892962,
       0.96002821,  0.96110001,  0.96214817,  0.96315171,  0.96414199])
```

```
[ ] fig, ax = plt.subplots()
xs = np.arange(1, 51)
ys = ll[:50]

plt.xlabel("Число главных компонент")
plt.ylabel("Логарифмическое правдоподобие профиля")

ax.plot(xs, ys, marker = 'o')
plt.show()
```





[ ] 11[:50]

```
array([-10768.08991578, -12039.50334162, -12179.06662938, -12253.05844643,
-12301.30713181, -12332.93758767, -12354.63268964, -12371.58720108,
-12384.76427033, -12395.96117638, -12405.39765969,
-12420.70728414, -12426.77563336, -12432.07232496,
-12440.83912381, -12444.50187769, -12447.82495746,
-12453.55436921, -12456.04337979, -12458.30900981,
-12462.29807447, -12464.08350443, -12465.74289386,
-12468.72758347, -12470.05947958, -12471.32582795,
-12473.63364858, -12474.69390032, -12475.69729028,
-12477.55232387, -12478.41123502, -12479.22677812,
-12480.00573687,
-12480.74918174, -12481.45869466, -12482.13777761,
-12482.78718788,
-12483.41113068, -12484.00928679, -12484.58289927, -12485.13335524,
-12485.66334138, -12486.17231608])
```

[ ] X\_transformed.shape

(18809, 1024)

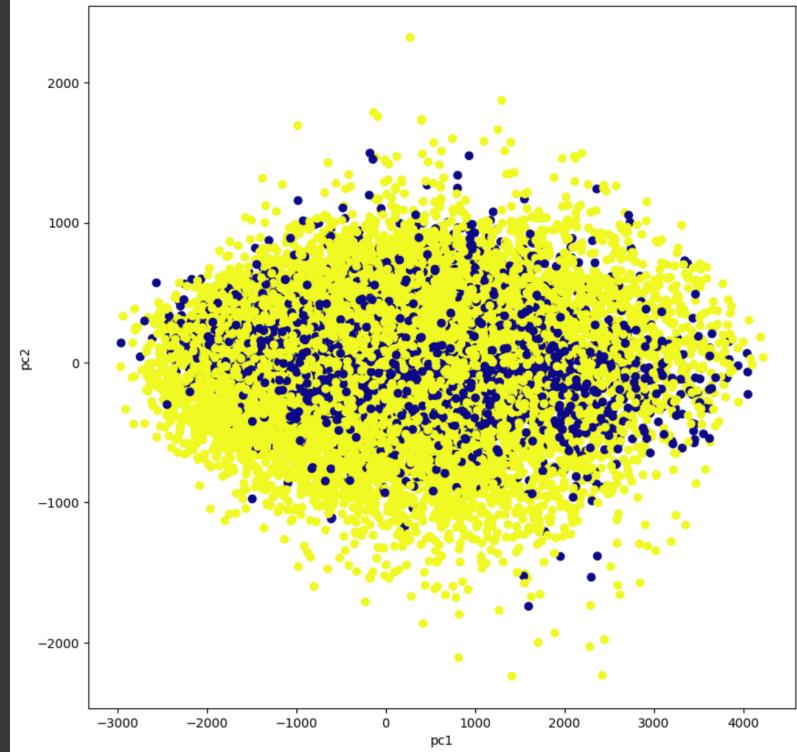
[ ] evals

```
array([1.54367864e+06, 1.81080615e+05, 1.57802969e+05, ...,
1.51190958e-01, 1.48884625e-01, 1.40332863e-01])
```

▼ Желтый – первый класс. Синий – второй

```
[ ] plt.figure(figsize = (10, 10))
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], c = (df_train[df_train['label'] < 2])['label'].values, cmap = 'plasma') # c = df_train['label'].values,
plt.xlabel('pc1')
plt.ylabel('pc2')
```

▼ Text(0, 0.5, 'pc2')



Как мы видим, видной корреляции нет, что было еще понятно на момент визуализации логарифмического правдоподобия.

6. Выполните бинарную классификацию изображений по латентным (скрытым) признакам и всем признакам при помощи классификатора метода ближайших соседей (kNN). Оцените бинарный классификатор, указанный в индивидуальном задании, для двух построенных классификаторов.

5. Показатель качества бинарной классификации:

F1-мера, равная  $2TP/(2TP+FP+FN)$

```
[ ] y_train = (df_train[df_train['label'] < 2])['label'].values
```

[ ] y\_train.shape

(18809,)

```
[ ] y_test = (df_test[df_test['label'] < 2])['label'].values
y_test.shape
→ (6843,)

[ ] from sklearn import datasets
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
import seaborn as sns

[ ] encoded_train_imgs = encoder16.predict(X_train)
→ 588/588 [=====] - 2s 3ms/step

[ ] encoded_train_imgs.shape
→ (18809, 16)

[ ] def plot_confusion_matrix(data, labels, fname):
    sns.set(color_codes=True)
    plt.figure(1, figsize=(9,6))

    sns.set(font_scale = 1.3)
    ax = sns.heatmap(data, annot = True, cmap = 'Blues',
                      cbar_kws = {'label': 'Шкала'}, fmt = 'd')

    ax.set_xticklabels(labels, fontsize = 16)
    ax.set_yticklabels(labels, fontsize = 16)

    ax.set_xlabel("Прогнозируемые метки", fontsize = 16)
    ax.set_ylabel("Истинные метки", fontsize = 16)

    plt.show()
    plt.close()
```

▼ По латентным признакам:

```
[ ] encoded_test_imgs = encoder16.predict(X_test)
→ 214/214 [=====] - 1s 3ms/step

[ ] from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 7).fit(encoded_train_imgs, y_train)

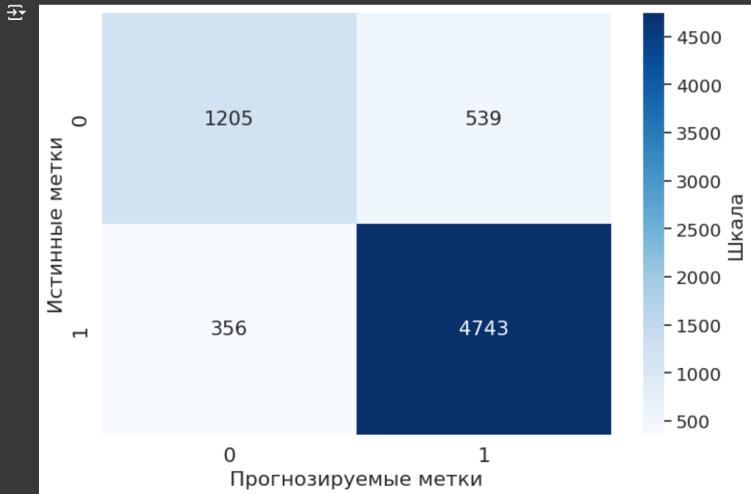
accuracy = knn.score(encoded_test_imgs, y_test)
print(accuracy)

→ 0.869209411077013

[ ] knn_predictions = knn.predict(encoded_test_imgs)

cm = confusion_matrix(y_test, knn_predictions)

plot_confusion_matrix(cm, [0, 1], "idk")
```



▼ Со всеми метками:

```
[ ] knn = KNeighborsClassifier(n_neighbors = 7).fit(X_train, y_train)

accuracy = knn.score(X_test, y_test)
print(accuracy)

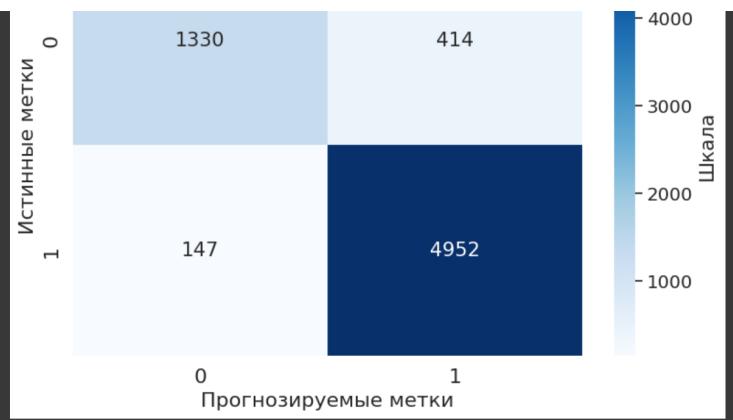
→ 0.9180184129767646

[ ] knn_predictions = knn.predict(X_test)

cm = confusion_matrix(y_test, knn_predictions)

plot_confusion_matrix(cm, [0, 1], "idk")
```





7. Визуализируйте ROC-кривые для построенных классификаторов на одном рисунке (с легендой) (Указание: используйте метод predict\_proba() класса KNeighborsClassifier).

```
[ ] def TN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 0))

[ ] def FP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 1))

[ ] def FN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 0))

[ ] def TP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 1))

❶ def tpr_score(y_true, y_predict):
    tp = TP(y_true, y_predict)
    fn = FN(y_true, y_predict)
    try:
        return tp / (tp+fn)
    except:
        return 0.0

[ ] def fpr_score(y_true, y_predict):
    fp = FP(y_true, y_predict)
    tn = TN(y_true, y_predict)
    try:
        return fp/(fp+tn)
    except:
        return 0.0

[ ] def true_false_positive(threshold_vector, y_test):
    true_positive = np.equal(threshold_vector, 1) & np.equal(y_test, 1)
    true_negative = np.equal(threshold_vector, 0) & np.equal(y_test, 0)
    false_positive = np.equal(threshold_vector, 1) & np.equal(y_test, 0)
    false_negative = np.equal(threshold_vector, 0) & np.equal(y_test, 1)

    tpr = true_positive.sum() / (true_positive.sum() + false_negative.sum())
    fpr = false_positive.sum() / (false_positive.sum() + true_negative.sum())

    return tpr, fpr

[ ] def roc_from_scratch(probabilities, y_test, partitions = 100):
    roc = np.array([[]])

    for i in range(partitions + 1):
        threshold_vector = np.greater_equal(probabilities, i / partitions).astype(int)
        tpr, fpr = true_false_positive(threshold_vector, y_test)
        roc = np.append(roc, [fpr, tpr])

    return roc.reshape(-1, 2)

[ ] prediction = knn.predict_proba(X_test)
prediction.shape
⇒ (6843, 2)

[ ] prediction[0]
⇒ array([0.14285714, 0.85714286])

[ ] y_test.shape
⇒ (6843,)

[ ] y_test[0]
⇒ 1

[ ] def to_one_hot_binary(arr: np.ndarray):
    res = np.array([[]])
    for i in range(arr.shape[0]):
```

```

if arr[i] == 1:
    res = np.append(res, [0, 1])
else:
    res = np.append(res, [1, 0])

return res.reshape(-1, 2)

[ ] y_roc = to_one_hot_binary(y_test)

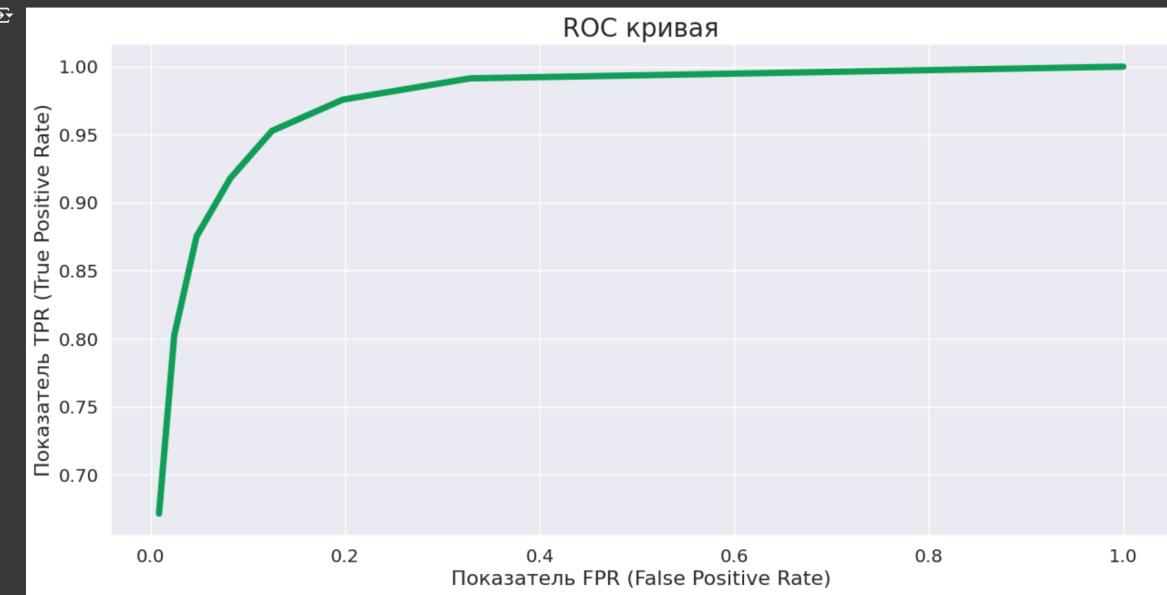
[ ] y_roc.shape
显示 (6843, 2)

[ ] y_roc[0]
显示 array([0., 1.])

[ ] plt.figure(figsize=(15,7))

ROC = roc_from_scratch(prediction.reshape(-1),y_roc.reshape(-1),partitions=500)
#plt.scatter(ROC[:,0],ROC[:,1],color='#0F9D58',s=100)
plt.plot(ROC[:,0],ROC[:,1],color='#0F9D58',lw=5)
plt.title('ROC кривая',fontsize=20)
plt.xlabel('Показатель FPR (False Positive Rate)',fontsize=16)
plt.ylabel('Показатель TPR (True Positive Rate)',fontsize=16);

```



8. Визуализируйте границы принятия решений классификатора kNN для латентных признаков на плоскости, соответствующей двум первым латентным признакам (для прочих латентных признаков задайте средние/медианные значения).

```

[ ] def plot_decision_boundary(model, X, y):
    # Найдем диапазоны изменения по осям и построим сетку
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))
    # Набор данных для прогнозирования
    X_in = np.c_[xx.ravel(), yy.ravel()]
    # Прогноз при помощи обученной модели
    y_pred = model.predict(X_in)
    # Проверка мультиклассовости
    if len(y_pred[1]) > 1:
        # мультиклассовая классификация
        # изменяем форму прогноза для визуализации
        y_pred = np.argmax(y_pred, axis=1).reshape(xx.shape)
    else:
        # бинарная классификация
        y_pred = np.round(y_pred).reshape(xx.shape)
    # Рисуем границу решения
    plt.contourf(xx, yy, y_pred, cmap=plt.cm.RdYlBu, alpha=0.7)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

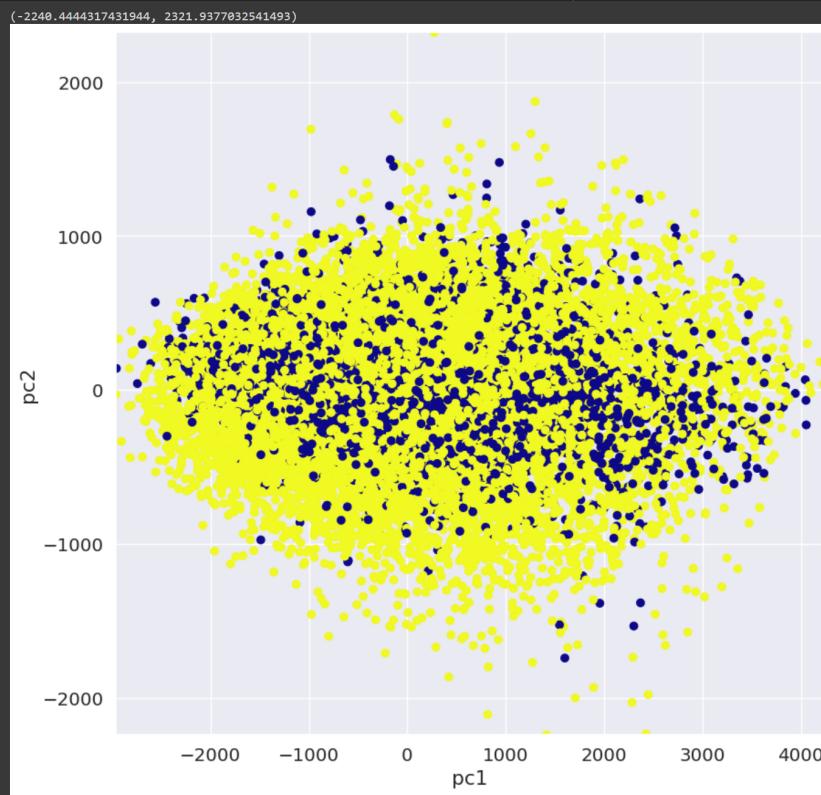
[ ] encoded_test_imgs = encoder16.predict(X_test)
显示 214/214 [=====] - 1s 6ms/step

[ ] knn = KNeighborsClassifier(n_neighbors = 7).fit(encoded_train_imgs, y_train)

[ ] knn_predictions = knn.predict(encoded_test_imgs)

[ ] # plt.countourf(xx, yy, y_pred, cmap = plt.cm.RdYlBl, alpha = 0.7)
plt.figure(figsize = (10, 10))
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], c = y_train, cmap = 'plasma') # c = df_train['label'].values,
plt.xlabel('pc1')
plt.ylabel('pc2')

```



9. Определите на первоначальной тестовой выборке изображение, имеющее

- ✓ наибольшую ошибку реконструкции. Выведите для этого изображения первоначальное и реконструированное изображения.

```
[ ] RE = ((X_test_MLP - decoded_imgs)**2).mean(axis = 1)
RE_orig = RE.copy()

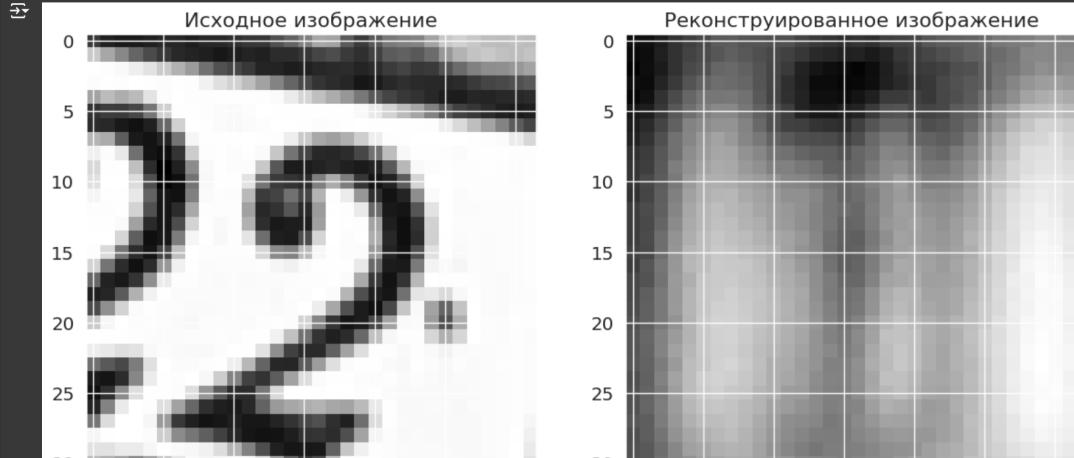
[ ] RE.shape
RE (16397,)

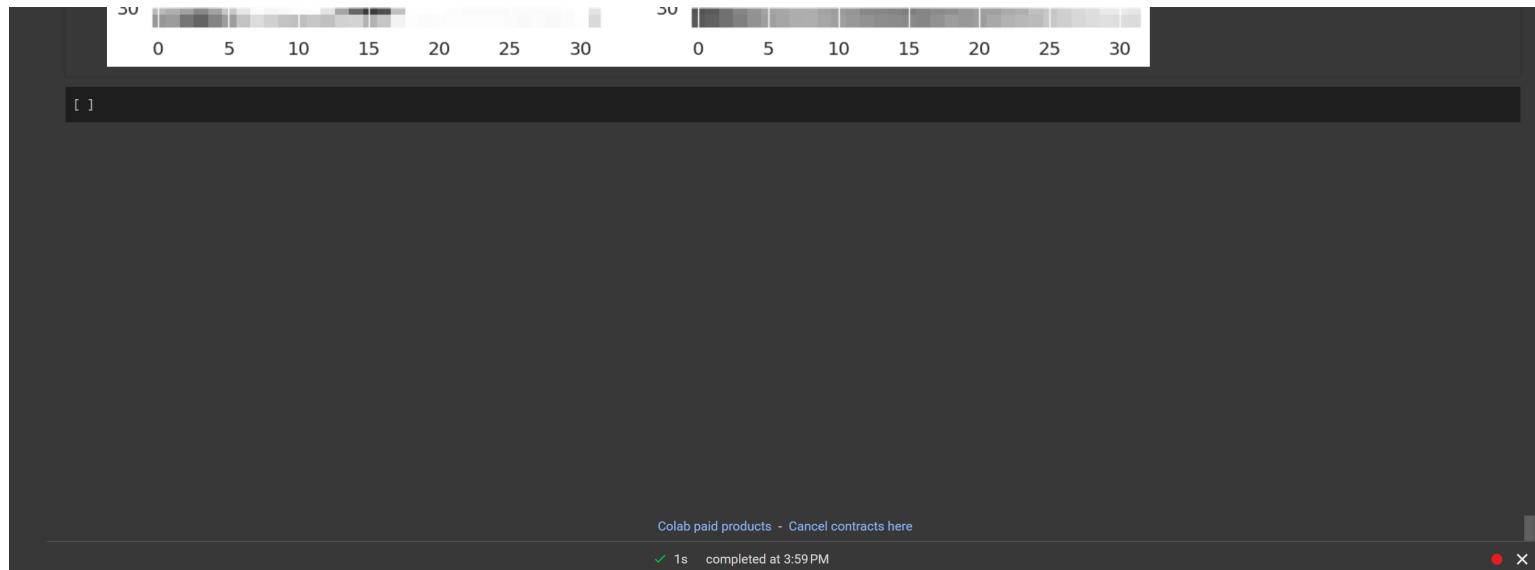
[ ] RE.sort()
print(RE[16387:])

RE [0.06340223 0.0664329 0.06709305 0.06770665 0.07038349 0.07225961
 0.0727417 0.0731889 0.07516913 0.08634971]

[ ] biggest_re_ind = np.argmax(RE_orig)

[ ] X_test_MLP.shape
X (16397, 1024)
```





Colab paid products - Cancel contracts here

✓ 1s completed at 3:59 PM

