Home Project

Aliaksei Rak
Nerses Bagiyan

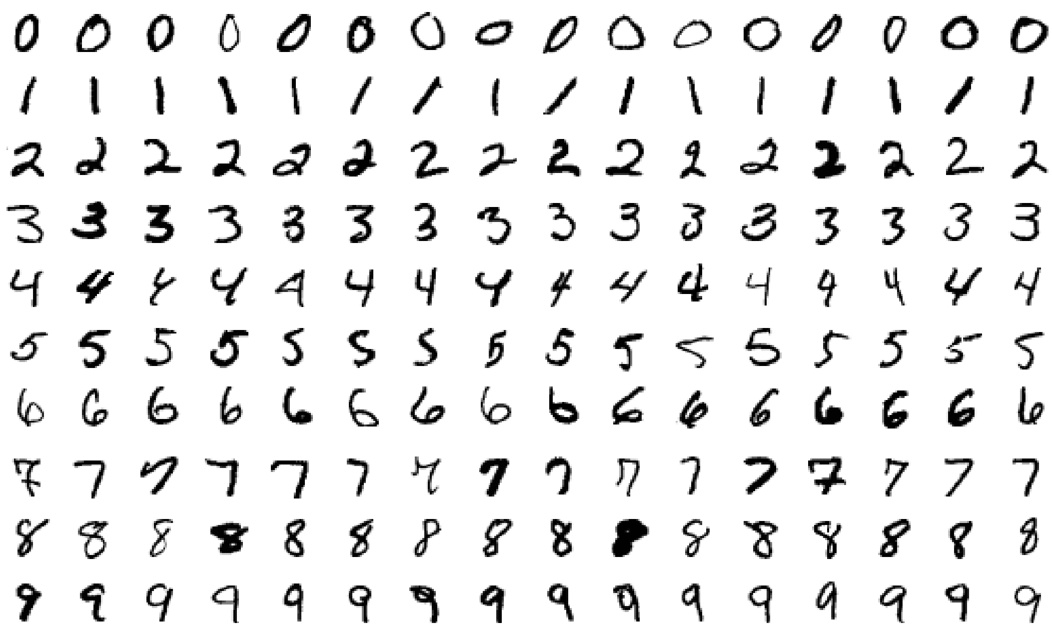
December 12, 2019

Code

All code and data you can found in our repo `https://github.com/kventinel/hse-ml-project-mnist`

Data

For our project we get MNIST dataset, that consists from 60000 images of digits from 0 to 9 in train and 10000 in test. From this images we randomly choose 1000 images in train and 1000 images in test. For each image in original dataset presents 28×28 features in range from 0 to 255 – value of each pixel of image.



We preprocess this image and get next features:

- count – count of nonzero pixels in image
- mean – average value of pixels in image
- vert_symmetry – difference between mean of pixels in top half of image and bottom half of image
- hor_symmetry – difference between mean of pixels in right half of image and bottom half of image
- vert_mass_center – weighted mean of pixels by this equation, where wight of pixel – it is index of row with this pixel
- hor_mass_center – weighted mean of pixels by this equation, where wight of pixel – it is index of column with this pixel

And 3 features received using filters Viola-Jones:

- vert_viola – difference between (mean of pixels in top 7 and bottom 7 rows) and (mean of pixels in center 14 rows)
- hor_viola – difference between (mean of pixels in left 7 and right 7 columns) and (mean of pixels in center 14 columns)
- all_viola – difference between (mean of pixels in top left quarter and right bottom quarter) and (mean of pixels in top right quarter and bottom left quarter)

We have got this dataset, because it's very common dataset for all machine learning courses and articles. And we were interesting to know more different facts about patterns in this dataset.

[2]:
```
train = pd.read_csv(constants.TRAIN)
test = pd.read_csv(constants.TEST)
```

[3]:
```
train
```

[3]:
```
        mean  count  vert_symmetry  hor_symmetry       vert_mass_center  \
0  26.538265    122      10.081633    -18.117347              14.299144
1  40.753827    155      -2.125000    -11.864796              13.984069
2  41.187500    176       0.323980     -4.267857              14.086804
3  38.184949    155      -2.043367     -2.477041              14.146140
4  48.011480    204      -6.772959    -21.385204              13.838341

         hor_mass_center  vert_viola  hor_viola  all_viola  label
0              14.299385  -32.581633 -42.862245  19.836735      7
1              14.207067  -23.267857 -77.915816   0.829082      3
2              13.838004  -35.079082 -76.956633 -24.625000      8
3              13.564352  -48.140306 -76.369898 -13.706633      9
4              14.332457  -21.022959 -74.385204 -24.446429      3

[1000 rows x 12 columns]
```

K-means

Let's firstly define functions for scaling the data and fitting KMeans clustering. First function returns us KMeans criterion with clustering labels based on 10 random intializations. We use sklearn implementation which optimexes inertia as the fuctional.

[6]:
```
def fit_kmeans(X, features, n_clusters, n_init):
    kmeans_criterion = []
    clustering_results = []
    for _ in range(n_init):
        clusterer = KMeans(n_clusters=n_clusters, n_init=1)
        predicted_labels = clusterer.fit_predict(X[features])
        clustering_results.append(predicted_labels)
        kmeans_criterion.append(clusterer.inertia_)
    return kmeans_criterion, clustering_results[np.argmin(kmeans_criterion)]
```

[7]:
```
def scale_data(X):
    scaler = MinMaxScaler()
    X_scaled = pd.DataFrame(
        scaler.fit_transform(X),
        columns = X.columns
    )

    return X_scaled
```

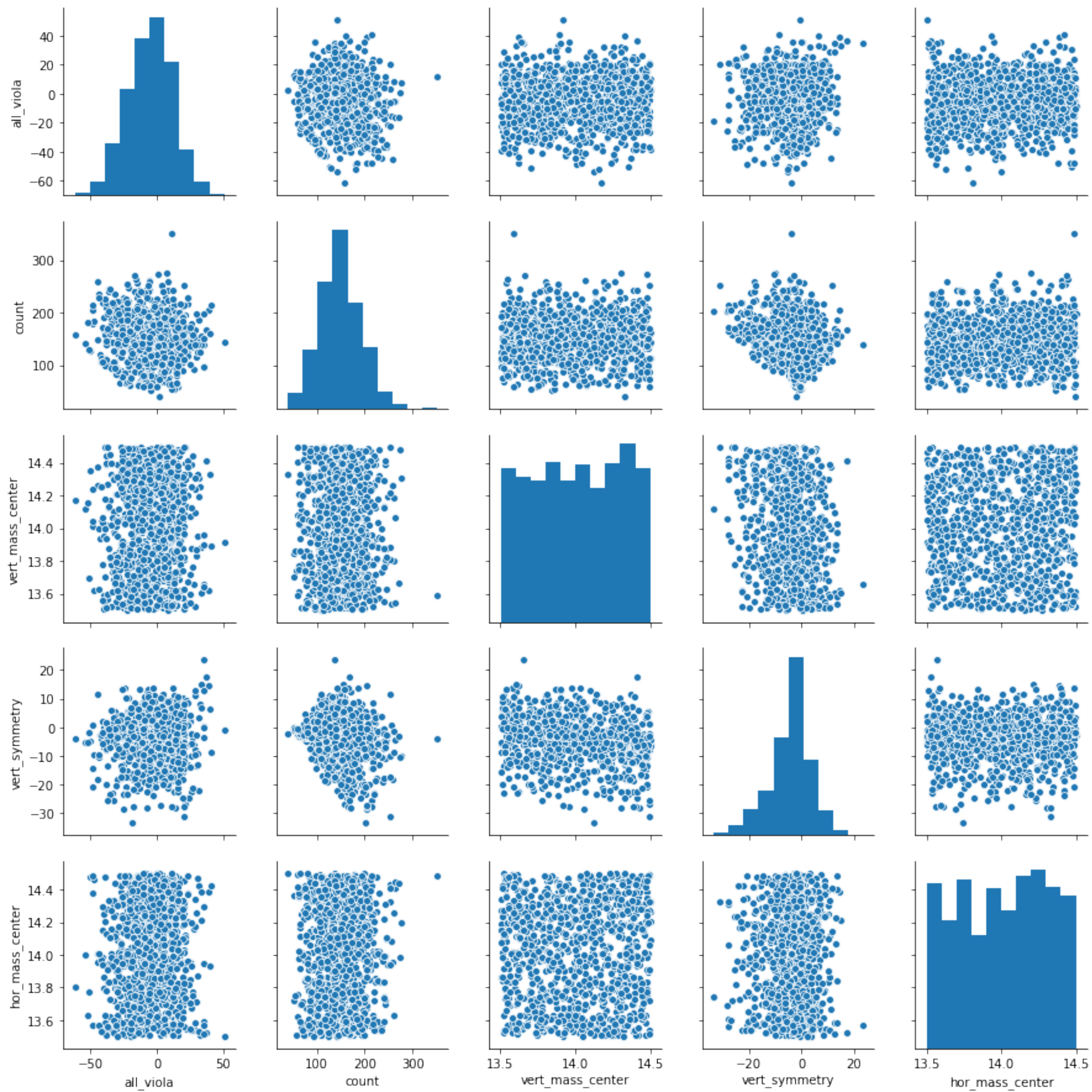
Now we could read the data and proceed to feature normalizing. We will normalize only continous features using MinMaxScaler. This scaler transform feature range to [0, 1] using its minimum and maximum value.

[8]:
```
clustering_features = ['all_viola', 'count', 'vert_mass_center',␣
 ↪'vert_symmetry', 'hor_mass_center']
clustering_features_scale = ['all_viola', 'vert_mass_center', 'vert_symmetry',␣
 ↪'hor_mass_center']

train = pd.read_csv(constants.TRAIN,
                    usecols = ['all_viola', 'count', 'hor_mass_center',␣
 ↪'hor_symmetry',
                               'hor_viola', 'idx', 'label', 'mean',␣
 ↪'vert_mass_center',
                                'vert_symmetry', 'vert_viola'])

train_scaled = scale_data(train[clustering_features_scale])
train_scaled['count'] = train['count']
train_scaled['label'] = train['label']
```

After we normalized the feature, we proceed to choosing the most basic features for clusterization. They are based only on general information about pixels and also their vertical axis. These are simple features that could tell us about different pixels on the images and the orientation of the object. Moreover, we tried to exclude the linear dependent feature in order to bring better generalization to the clustering.

[9]:
```
sns.pairplot(train[clustering_features])
```

[9]:
```
<seaborn.axisgrid.PairGrid at 0x134ac4710>
```



Now, after normalizing the data we could fit KMeans and anylize result. We will look at features means and comprare them to the grand mean (the mean of the all dataset):

[10]:
```
kmeans_five_criterion, labels_five = fit_kmeans(train_scaled,␣
 ↪clustering_features, 5, 10)
kmeans_nine_criterion, labels_nine = fit_kmeans(train_scaled,␣
 ↪clustering_features, 9, 10)

data_mean = train[clustering_features + ['label']].mean()
data_mean.name = 'data_mean'
```

[11]:
```
train['clusters_five'] = labels_five
```

```
results_five_clusters = train.groupby('clusters_five')[clustering_features +␣
 ↪['label']].mean()
results_five_clusters = results_five_clusters.append(data_mean)
results_five_clusters
```

[11]:
```
                all_viola       count  vert_mass_center  vert_symmetry  \
clusters_five
0               -8.027731  191.709845         14.008225      -5.470339
1               -2.530407  123.515050         14.001630      -2.464183
2               -6.526573   82.371212         13.997653      -1.700854
3              -11.112132  235.852941         13.961299      -7.241409
4               -3.219222  156.629870         14.028761      -5.422144
data_mean       -4.914597  149.084000         14.007992      -4.179515

                hor_mass_center     label
clusters_five
0                     14.036303  3.968912
1                     14.020967  5.394649
2                     14.008330  2.628788
3                     13.997987  3.117647
4                     13.988845  5.032468
data_mean             14.010802  4.488000
```

But before looking at the statistics let's look at the difference between intializations:

[12]:
```
pd.DataFrame(kmeans_five_criterion,
             index = ["run #" + str(x) for x in range(1, 11)],
             columns = ['inertia']
)
```

[12]:
```
                inertia
run #1    202252.260511
run #2    143905.693049
run #3    145129.696990
run #4    143696.083568
run #5    142912.145003
run #6    145129.696990
run #7    145129.696990
run #8    145129.696990
run #9    145129.696990
run #10   143725.213854
```

[13]:
```
pd.DataFrame(kmeans_nine_criterion,
             index = ["run #" + str(x) for x in range(1, 11)],
             columns = ['inertia']
)
```

[13]:

	inertia
run #1	50980.675144
run #2	51068.938199
run #3	50280.957256
run #4	53876.890407
run #5	52484.961622
run #6	52516.126284
run #7	50213.154914
run #8	50749.273014
run #9	51096.792553
run #10	51423.927640

According to the inertia the best runs are #10 and #3 for five and nine clusters case respectively

[14]:
```
rel_difference = 100 * (results_five_clusters.iloc[:-1] - results_five_clusters.
 ↪loc['data_mean']) \
                / results_five_clusters.loc['data_mean']
rel_difference.columns = [x + '_diff, %' for x in rel_difference.columns]


rel_difference
```

[14]:

clusters_five	all_viola_diff, %	count_diff, %	vert_mass_center_diff, %
0	63.344642	28.591830	0.001665
1	-48.512411	-17.150700	-0.045415
2	32.799764	-44.748456	-0.073803
3	126.104653	58.201377	-0.333330
4	-34.496722	5.061489	0.148267

clusters_five	vert_symmetry_diff, %	hor_mass_center_diff, %	label_diff, %
0	30.884541	0.182008	-11.566134
1	-41.041409	0.072545	20.201623
2	-59.304989	-0.017646	-41.426295
3	73.259542	-0.091471	-30.533711
4	29.731411	-0.156718	12.131630

[15]:
```
pd.crosstab(train['clusters_five'], train['label'])
```

[15]:

label / clusters_five	0	1	2	3	4	5	6	7	8	9
0	53	0	25	24	9	10	13	5	41	13
1	7	20	23	16	45	38	31	55	18	46
2	0	88	2	5	3	5	6	20	0	3
3	28	0	9	8	0	3	3	0	16	1
4	23	2	40	30	38	37	37	26	38	37

[16]:
```
train['clusters_nine'] = labels_nine
results_nine_clusters = train.groupby('clusters_nine')[clustering_features +␣
 ↪['label']].mean()
results_nine_clusters = results_nine_clusters.append(data_mean)
results_nine_clusters
```

[16]:
```
                all_viola       count  vert_mass_center  vert_symmetry  \
clusters_nine
0               -2.874304  150.898990         14.036684      -4.666048
1               -7.343899  194.550000         14.014254      -5.087096
2               -9.262894   75.923913         13.993732      -1.615600
3               -2.845576  106.726027         13.993887      -1.386847
4              -11.906353  249.419355         13.981966      -6.900428
5               -5.692571  170.487654         14.003582      -6.940271
6               -1.815851  130.761658         14.008533      -3.161600
7               11.522959  351.000000         13.589157      -4.002551
8               -9.935911  216.736842         13.986490      -5.993734
data_mean       -4.914597  149.084000         14.007992      -4.179515

                hor_mass_center     label
clusters_nine
0                     13.990923  5.186869
1                     14.035304  3.966667
2                     14.029493  1.891304
3                     14.033635  5.157534
4                     14.016093  3.451613
5                     13.992744  4.598765
6                     13.998676  5.352332
7                     14.487500  0.000000
8                     14.020768  3.035088
data_mean             14.010802  4.488000
```

[17]:
```
abs_difference = results_nine_clusters.iloc[:-1] - results_nine_clusters.
 ↪loc['data_mean']
abs_difference.columns = [x + '_diff' for x in abs_difference.columns]

abs_difference
```

[17]:
```
                all_viola_diff  count_diff  vert_mass_center_diff  \
clusters_nine
0                     2.040293    1.814990               0.028692
1                    -2.429302   45.466000               0.006263
2                    -4.348297  -73.160087              -0.014259
3                     2.069021  -42.357973              -0.014105
4                    -6.991756  100.335355              -0.026025
5                    -0.777974   21.403654              -0.004410
6                     3.098746  -18.322342               0.000541
```

```
7                 16.437556  201.916000              -0.418834
8                 -5.021314   67.652842              -0.021502

               vert_symmetry_diff  hor_mass_center_diff  label_diff
clusters_nine
0                       -0.486533             -0.019879    0.698869
1                       -0.907581              0.024502   -0.521333
2                        2.563915              0.018691   -2.596696
3                        2.792669              0.022832    0.669534
4                       -2.720913              0.005290   -1.036387
5                       -2.760756             -0.018059    0.110765
6                        1.017915             -0.012126    0.864332
7                        0.176964              0.476698   -4.488000
8                       -1.814219              0.009965   -1.452912
```

[18]:

```python
rel_difference = 100 * (results_nine_clusters.iloc[:-1] - results_nine_clusters.
 ↪loc['data_mean']) \
                / results_nine_clusters.loc['data_mean']
rel_difference.columns = [x + '_diff, %' for x in rel_difference.columns]


rel_difference
```

[18]:

```
               all_viola_diff, %  count_diff, %  vert_mass_center_diff, %  \
clusters_nine
0                     -41.514954       1.217428                  0.204824
1                      49.430338      30.496901                  0.044707
2                      88.477181     -49.073064                 -0.101795
3                     -42.099506     -28.412152                 -0.100694
4                     142.265093      67.301223                 -0.185789
5                      15.829856      14.356775                 -0.031484
6                     -63.051890     -12.289945                  0.003862
7                    -334.463972     135.437740                 -2.989967
8                     102.171436      45.379009                 -0.153495

               vert_symmetry_diff, %  hor_mass_center_diff, %  label_diff, %
clusters_nine
0                          11.640894                -0.141887      15.571940
1                          21.714977                 0.174879     -11.616162
2                         -61.344799                 0.133404     -57.858638
3                         -66.818006                 0.162963      14.918321
4                          65.101152                 0.037758     -23.092404
5                          66.054458                -0.128893       2.468035
6                         -24.354868                -0.086549      19.258726
7                          -4.234086                 3.402361    -100.000000
8                          43.407402                 0.071125     -32.373268
```

[19]:
```
pd.crosstab(train['clusters_nine'], train['label'])
```

[19]:

label clusters_nine	0	1	2	3	4	5	6	7	8	9
0	11	2	21	20	30	24	24	18	19	29
1	35	0	14	16	1	8	9	2	27	8
2	0	76	0	2	0	2	2	10	0	0
3	0	26	6	8	18	13	13	39	4	19
4	14	0	2	2	0	1	2	0	10	0
5	20	0	29	16	16	14	17	10	27	13
6	7	6	19	11	30	28	22	26	14	30
7	1	0	0	0	0	0	0	0	0	0
8	23	0	8	8	0	3	1	1	12	1

Interpretation:

9 clusters case:

- Cluster five has the biggest difference with grand mean in feature "count". It means that in average case there are more black pixels in cluster five than in the whole sample.
- Cluster zero has a lot of similar statistics compared to the data mean. For example, mean_diff and vert_mass_center_diff are 1.4 and 0.12 percents respectively. It shows us that the distribution of different digits in cluster zero is almost the same as in the whole sample.
- Cluster three has a big difference in all_viola feature because there most popular digit in this clusters are zero and which have one of the biggest viola mean in the whole dataset.

5 clusters case:

This case is more interesting for analysis because we see a big relative difference in almost every claster when comparing to the grand mean of every feature. Let's see:

- Cluster two has the smallest mean of the feature named count. It could be easily explained by analyzing the most popular digit in the cluster. It's a digit 1, which almost doesn't have white pixels. That is why this cluster has the smallest mean of feature count.
- Like in the 9 clusters case there is a cluster which has some features that does not differ from grand mean. For example, mean and vert_mass_center_diff

Overall, the clusterization is very noisy for both cases. We could see that by looking ath distribution of digits between the clusters. There are always some clusters that contain each type of digits and they also have very small relative difference in their mean. This clusterization technique will be much better if we add raw features from the data or more features created based on raw data.

Bootstrap

Firstly, let us define fucntion which computes both pivotal and non-pivotal version of the bootstrap. It takes the following arguments:

- X - our data
- feature - feature for which we computing bootstrap mean

- cluster_label - the name of the cluster feature where we store clusterinf results
- first_cluster - the first cluster number for bootstrap computation
- first_cluster - the second cluster number for bootstrap computation
- pivotal - booalean variable which tells us if use pivotal version or not
- :return: dictionary with three keys, each one corresponds to first cluster mean, second cluster mean or grand mean. The value is a list which contains left and right bounf for CI

[20]:
```
def bootstrap(X, feature, cluster_label, first_cluster=None,
              second_cluster=None, pivotal=True):
    first_sample_means = []
    second_sample_means = []
    grand_sample_means = []
    for _ in range(1000): # bootstraping
        if first_cluster is not None:
            first_sample_means.append(
                X[X[cluster_label] == first_cluster][feature] \
                                                    .sample(100, replace=True) \
                                                    .mean()
            )
        if second_cluster is not None:
            second_sample_means.append(
                X[X[cluster_label] == second_cluster][feature]
                                                    .sample(100, replace=True) \
                                                    .mean()
            )

        grand_sample_means.append(
            X[feature].sample(500).mean()
        )

    if pivotal:
        if first_cluster is not None:
            first_mean = np.mean(first_sample_means)
            first_std = np.std(first_sample_means)

        if second_cluster is not None:
            second_mean = np.mean(second_sample_means)
            second_std = np.std(second_sample_means)

        grand_std = np.std(grand_sample_means)
        grand_mean = np.mean(grand_sample_means)

        print(first_mean)
        return {
```

```
            'first_CI' : [first_mean - 1.96 * first_std, first_mean + 1.96 *␣
↪first_std],
            # 1.96 is quantile of normal distribution
            'second_CI' : [second_mean - 1.96 * second_std, second_mean + 1.96 *␣
↪second_std],
            'grand_CI' : [grand_mean - 1.96 * grand_std, grand_mean + 1.96 *␣
↪grand_std]
        }
    else:
        if first_cluster is not None:
            first_lb = np.percentile(first_sample_means, 2.5)
            first_rb = np.percentile(first_sample_means, 97.5)

        if second_cluster is not None:
            second_lb = np.percentile(second_sample_means, 2.5)
            second_rb = np.percentile(second_sample_means, 97.5)

        grand_lb = np.percentile(grand_sample_means, 2.5)
        grand_rb = np.percentile(grand_sample_means, 97.5)

        return {
            'first_CI' : [first_lb, first_rb],
            'second_CI' : [second_lb, second_rb],
            'grand_CI' : [grand_lb, grand_rb]
        }
```

Pivotal version:

[21]:
```
bootstrap(train, 'count', 'clusters_five', 0, 3)
```

```
191.64603999999997
```

[21]:
```
{'first_CI': [189.56506920613344, 193.7270107938665],
 'second_CI': [231.78550413444978, 240.08575586555023],
 'grand_CI': [146.39047234839848, 151.7739476516016]}
```

Non-pivotal version:

[22]:
```
bootstrap(train, 'count', 'clusters_five', 0, 3, False)
```

[22]:
```
{'first_CI': [189.59925, 193.88025],
 'second_CI': [231.93975, 240.45125],
 'grand_CI': [146.32350000000002, 151.62405]}
```

Conclusion:

- We could see that pivotal and non-pivotal bootstrap has almost no difference in results and speed. It means we could use both of the methods for our calculation

- If we compare two clusters we will see that first one has a lot of ones and that's why CI for it is much different. It is caused by the number of ones that appeared in the cluster zero.
- Bootstrap estimate for the second CI and grand CI are almost the same. If we look at the distribution of classes in cluster in| the previous task we will notice that the distributions are very similar.

Contingency Table

Let us take the following features for the analysis:

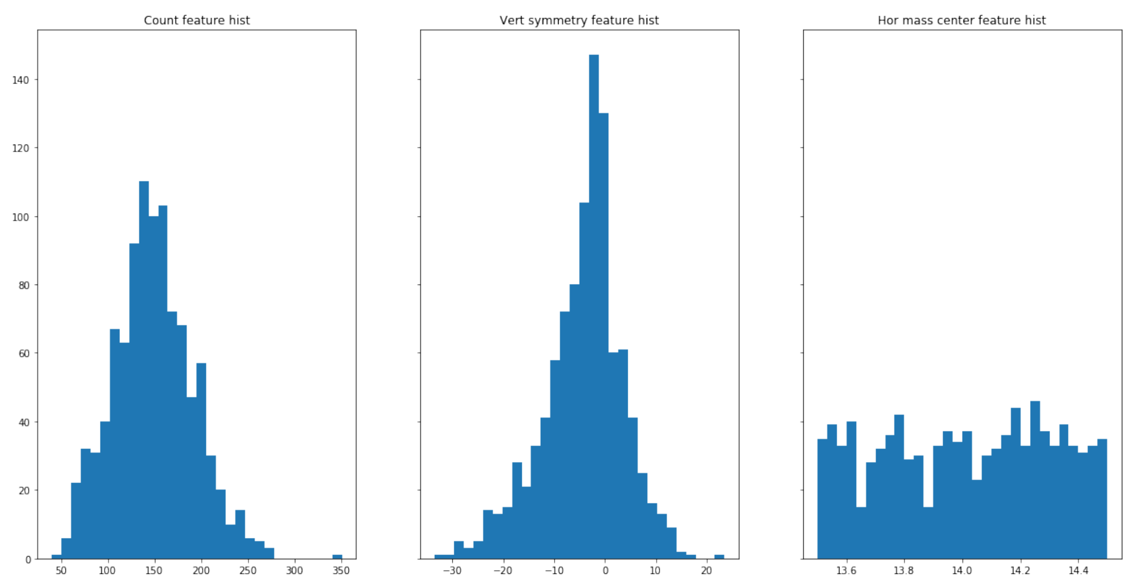
- count
- vert_symmetry
- hor_mass_center

Now we will look at their histogramms and choose the cutoffs:

[23]:
```
plt.figure()
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize=(20, 10))
ax1.hist(train['count'], bins = 30)
ax1.set_title('Count feature hist')
ax2.hist(train['vert_symmetry'], bins = 30)
ax2.set_title('Vert symmetry feature hist')
ax3.hist(train['hor_mass_center'], bins = 30)
ax3.set_title('Hor mass center feature hist')
```

[23]:
```
Text(0.5, 1.0, 'Hor mass center feature hist')
```

```
<Figure size 432x288 with 0 Axes>
```



Boundaries for the first feature:

- 0-100
- 100-175
- 175-300
- 300-400

Boundaries for the second feature:

- -35-(-10)
- -10-0
- 0-20
- 20-40

Boundaries for the third feature:

- 0-13.7
- 13.7-14.1
- 14.2-...

Now we will use method from pandas library for categorizing the feature.

[24]:
```
train['count_cat'] = pd.cut(train['count'],
                            bins=[0, 100, 175, 300, 400],
                            labels=["0-100", "100-175", "175-300", "300-400"])
```

[25]:
```
train['vert_symmetry_cat'] = pd.cut(train['vert_symmetry'],
                            bins=[-35, -10, 0, 20, 40],
                            labels=["-35-(-10)", "-10-0", "0-20", "20-40"])
```

[26]:
```
train['hor_mass_center_cat'] = pd.cut(train['hor_mass_center'],
                            bins=[0, 13.7, 14.1, 14.5],
                            labels=["0-13.7", "13.7-14.1", "14.2-..."])
```

Now we will create conditional frequency table, based on our categorized features:

[27]:
```
pd.crosstab(train['label'], train['count_cat'],
            margins=True, margins_name='Total')
```

[27]:

count_cat	0-100	100-175	175-300	300-400	Total
label					
0	0	33	77	1	111
1	88	22	0	0	110
2	1	64	34	0	99
3	4	48	31	0	83
4	2	84	9	0	95
5	4	77	12	0	93

6	5	69	16	0	90
7	19	82	5	0	106
8	0	56	57	0	113
9	3	84	13	0	100
Total	126	619	254	1	1000

[28]:
```
pd.crosstab(train['label'], train['vert_symmetry_cat'],
            margins=True, margins_name='Total')
```

[28]:

vert_symmetry_cat	-35-(-10)	-10-0	0-20	20-40	Total
label					
0	8	88	15	0	111
1	1	104	5	0	110
2	68	27	4	0	99
3	4	53	26	0	83
4	32	52	11	0	95
5	8	44	41	0	93
6	66	23	1	0	90
7	2	16	87	1	106
8	7	70	36	0	113
9	2	57	41	0	100
Total	198	534	267	1	1000

[29]:
```
pd.crosstab(train['label'], train['hor_mass_center_cat'],
            margins=True, margins_name='Total')
```

[29]:

hor_mass_center_cat	0-13.7	13.7-14.1	14.2-...	Total
label				
0	20	46	45	111
1	25	36	49	110
2	21	44	34	99
3	13	24	46	83
4	15	34	46	95
5	20	43	30	93
6	14	37	39	90
7	22	37	47	106
8	20	46	47	113
9	20	31	49	100
Total	190	378	432	1000

From this table we could see the following:

- The most frequent result if there less than 100 pixels is number 1, because we need only few pixels to draw one.
- If there are more than 300 pixels it means that this an outlier for the dataset because there is only one such sample
- When talking about vertical simmetry we could notice that the in the range 0-20 the most

frequent digit is seven. This is normal because this digit has more pixel in the upper part.

- This is also true for the ones. But for most of them simmetry is near zero beacause of its form :)
- Hor mass center feature has distribution form closed to the Uniform that is what we nearly see in the frequency table

Now, for calculating quetlet table we define a function that takes the following arguments:

- X - data
- first_feature - first feature for calculating Quételet index
- second_feature - second feature for calculating Quételet index
- :return: dict with table and summary Quetelet index

This function iterates through all possible values of the both features and then calculates the index by deviding the probabilities. Also, it calculates chi square using the same probs.

[30]:
```
def build_quetlet_table(X, first_feature, second_feature):
    quetlet_table = {}
    chi_square = 0
    for k in X[first_feature].unique():
        quetlet_index = dict()
        for l in X[second_feature].unique():
            first_count = (X[first_feature] == k)
            second_count = (X[second_feature] == l)
            p_hg = (first_count & second_count).mean()
            p_h = first_count.mean()
            p_g = second_count.mean()
            quetlet_index[l] = p_hg / (p_h * p_g) - 1
            chi_square += (p_hg - p_h * p_g)**2 / (p_h * p_g + 1e-10)
        quetlet_table[k] = quetlet_index
    return quetlet_table, chi_square
```

[31]:
```
table_count_hor, chi_count_hor = build_quetlet_table(train, 'count_cat',␣
 ↪'hor_mass_center_cat')
```

[32]:
```
chi_count_hor
```

[32]: 0.006041453360232884

[33]:
```
table_count_vert, chi_count_vert =  build_quetlet_table(train, 'count_cat',␣
 ↪'vert_symmetry_cat')
```

[34]:
```
chi_count_vert
```

[34]: 0.06065758855161415

[35]:
```
pd.DataFrame(table_count_vert)
```

[35]:
```
              100-175   175-300     0-100   300-400
0-20         0.173815 -0.233242 -0.375780 -1.000000
-10-0       -0.116613  0.024802  0.515962  0.872659
-35-(-10)    0.077006  0.252684 -0.879750 -1.000000
20-40        0.615509 -1.000000 -1.000000 -1.000000
```

[36]:
```
pd.DataFrame(table_count_hor)
```

[36]:
```
              100-175   175-300     0-100   300-400
14.2-...    -0.031443  0.057160  0.028807  1.314815
13.7-14.1    0.064184 -0.083448 -0.139162 -1.000000
0-13.7      -0.056203  0.036055  0.211362 -1.000000
```

Conclusions:

- From both summary Quetelet index we could see that this features are most likely to be independent.
- We see that value 0-100 is 51% more likey to appear when vert_symmetry has value -10-0
- Also we see that value 300-400 is 87% more likey to appear when vert_symmetry has value -10-0

Let's calculate the number of observations that would suffice to see the features as associated. Remember that if the hypothesis of independence is true, then:

$NX^2 \sim \chi^2((K-1)(L-1))$, where K, L are numbers of possible different values for the features.

In this case, we have 2 degrees of freedom:

[40]:
```
print(chi2(df=2).ppf(0.95))
print(chi2(df=2).ppf(0.99))
```

```
5.991464547107979
9.21034037197618
```

If the features are independent, there is only 5% chance that NX^2 will be greater than 5.99, and 1% chance that NX^2 will be greater than 9.21. We want to find such N that NX^2 will exceed specified values, so we will calculate 5.99 and $\frac{9.21}{X^2}$

[41]:
```
def chi_square(dataframe, ver_feature, hor_feature):
    ver_column = dataframe[ver_feature]
    hor_column = dataframe[hor_feature]
    result = 0
    for ver_value in sorted(ver_column.unique()):
        for hor_value in sorted(hor_column.unique()):
            p_vk = ((ver_column == ver_value) &
                    (hor_column == hor_value)).mean()
            p_k = (ver_column == ver_value).mean()
```

```
            p_v = (hor_column == hor_value).mean()
            result += (p_vk - p_k * p_v)**2 / (p_k * p_v)
    max_result = min(len(ver_column.unique()), len(hor_column.unique())) - 1
    return result, max_result
```

[43]:
```
feature1 = 'count_cat'
for feature2 in ('hor_mass_center_cat', 'vert_symmetry_cat'):
    for confidence in [0.95, 0.99]:
        min_N = (chi2(df=2).ppf(confidence) / chi_square(train, feature1,␣
 ↪feature2)[0])
        print('{} and {} are associated with confidence {:.2f}\n\twhen N >= {:.
 ↪1f}'.format(
            feature1, feature2, confidence, min_N
        ))
```

```
count_cat and hor_mass_center_cat are associated with confidence 0.95
        when N >= 991.7
count_cat and hor_mass_center_cat are associated with confidence 0.99
        when N >= 1524.5
count_cat and vert_symmetry_cat are associated with confidence 0.95
        when N >= 98.8
count_cat and vert_symmetry_cat are associated with confidence 0.99
        when N >= 151.8
```

In our case we have 1000 of samples and we can conlude, that we have enough data to say that the vert_symmetry depends on count of pixels with 99% confidence, and enough data to say that ho_symmetry depends on count of pixels with 95% confidence (but not 99% confidence).

PCA/SVD

Features

In this part we have get such features, as **mean**, **count**, **vert_symmetry**. We choose this features, because **mean**, **count** high correlated and interesting, how good pca and svd decorelate this features.

[69]:
```
features = ['mean', 'count', 'vert_symmetry']
```

[70]:
```
X_train = np.array([train[feature].values for feature in features]).T
X_test = np.array([test[feature].values for feature in features]).T
```

Standardize and SVD

[71]:
```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
X_train_norm = (X_train - means) / stds
```

```
print('data scatter:', np.sum(X_train ** 2))
print('data scatter after centering: ', np.sum((X_train - means) ** 2))
print('data scatter after standardize: ', np.sum(X_train_norm ** 2))
```

```
data scatter: 25337027.391464494
data scatter after centering:  1998165.1463920956
data scatter after standardize:  2999.9999999999995
```

[74]:
```
pca = PCA()
transformed_array = pca.fit_transform(X_train_norm)
transformed = pd.DataFrame(transformed_array, columns=['PC' + str(i) for i in␣
 ↪range(1, len(features) + 1)])
```

[75]:
```
transformed.head()
```

[75]:

	PC1	PC2	PC3
0	-1.317348	1.511233	0.042066
1	0.479915	0.407563	0.375165
2	0.757433	0.808916	0.052019
3	0.324062	0.374046	0.216130
4	1.857682	0.182502	0.009792

[76]:
```
print('pca components: ', pca.components_)
```

```
pca components:  [[ 0.68032123  0.68058897 -0.27195897]
 [ 0.19338644  0.19122157  0.96230764]
 [ 0.70694038 -0.7072715  -0.00152458]]
```

[77]:
```
scatter = np.sum(X_train_norm ** 2)
for col_name in transformed:
    col_scatter = np.sum(transformed[col_name] ** 2)
    scatter_percent = 100 * col_scatter / scatter
    print(col_name, 'contributes {0:.3f}, or {1:.2f}%, to the data scatter'.
 ↪format(col_scatter, scatter_percent))
```

```
PC1 contributes 2046.691, or 68.22%, to the data scatter
PC2 contributes 916.404, or 30.55%, to the data scatter
PC3 contributes 36.905, or 1.23%, to the data scatter
```

Hidden ranking factor

To obtain a hidden factor expressed in the 0-100 rank scale, we first rescale all the features to this range and then decompose the result using SVD and output the first component.

[78]:
```
def rescale(df):
    return (df - df.min()) / (df.max() - df.min()) * 100

rescaled_X = rescale(X_train)
```

```
U, s, V = np.linalg.svd(rescaled_X)
contribution = 100 * s[0] ** 2 / np.sum(rescaled_X ** 2)
```

```
[79]: print('First component:', V[0])
      print('Its contribution to the data scatter: {0:.3f}%'.format(contribution))
```

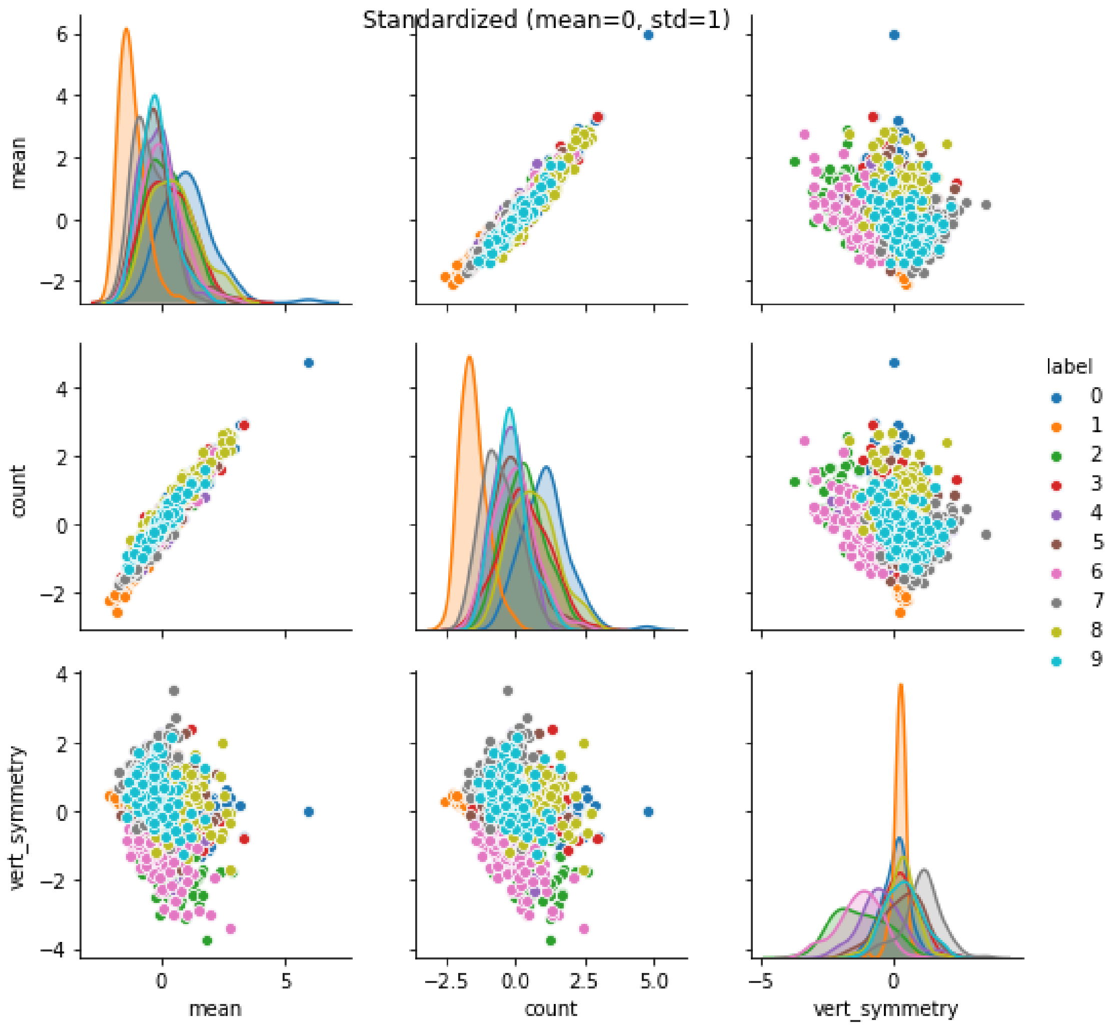
```
First component: [-0.33506822 -0.93161987 -0.14076113]
Its contribution to the data scatter: 99.638%
```

That the contribution of the first component is much higher than in the previous case should not be surprising: the data is not centered, so its mean is the source of most of the data scatter.

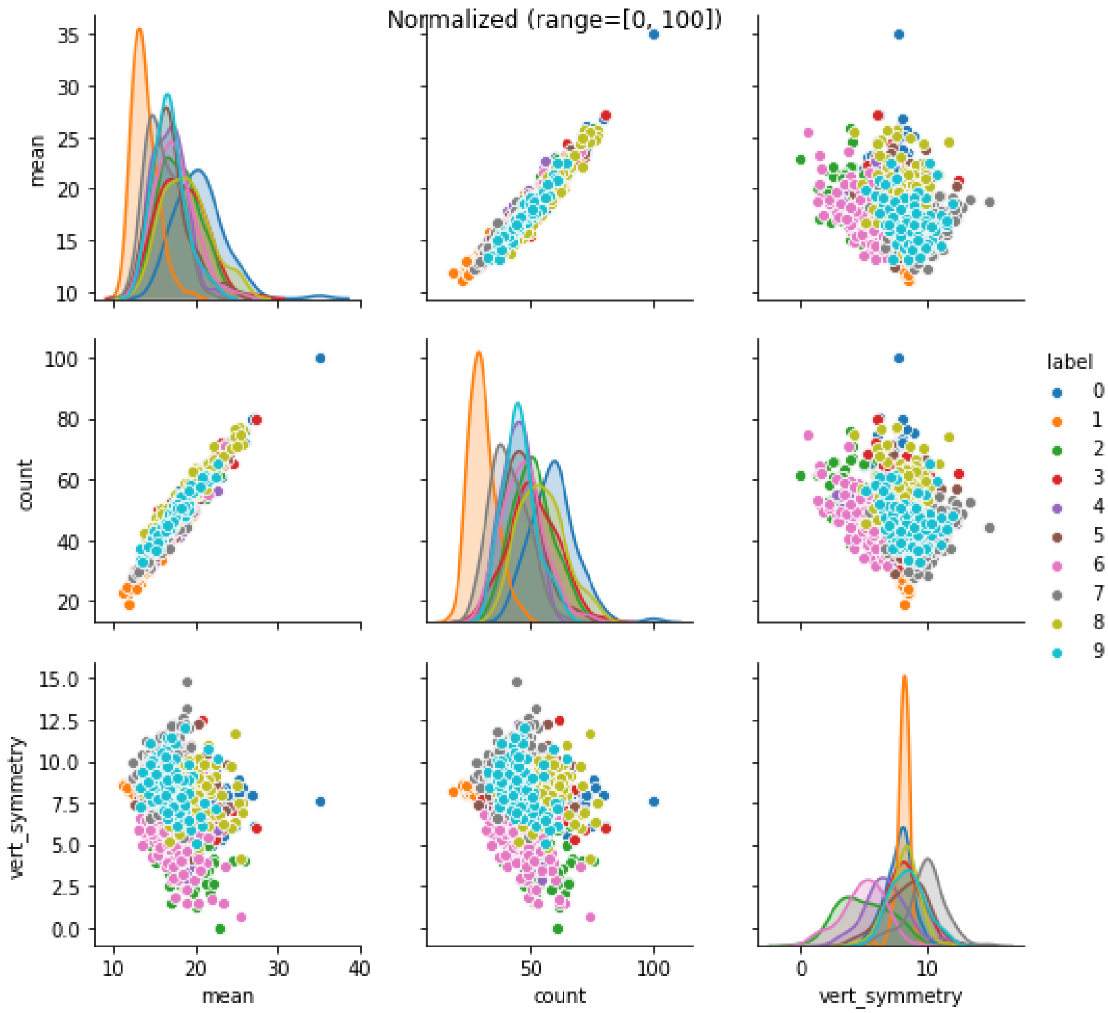
Visualization

At first visualize all labels.

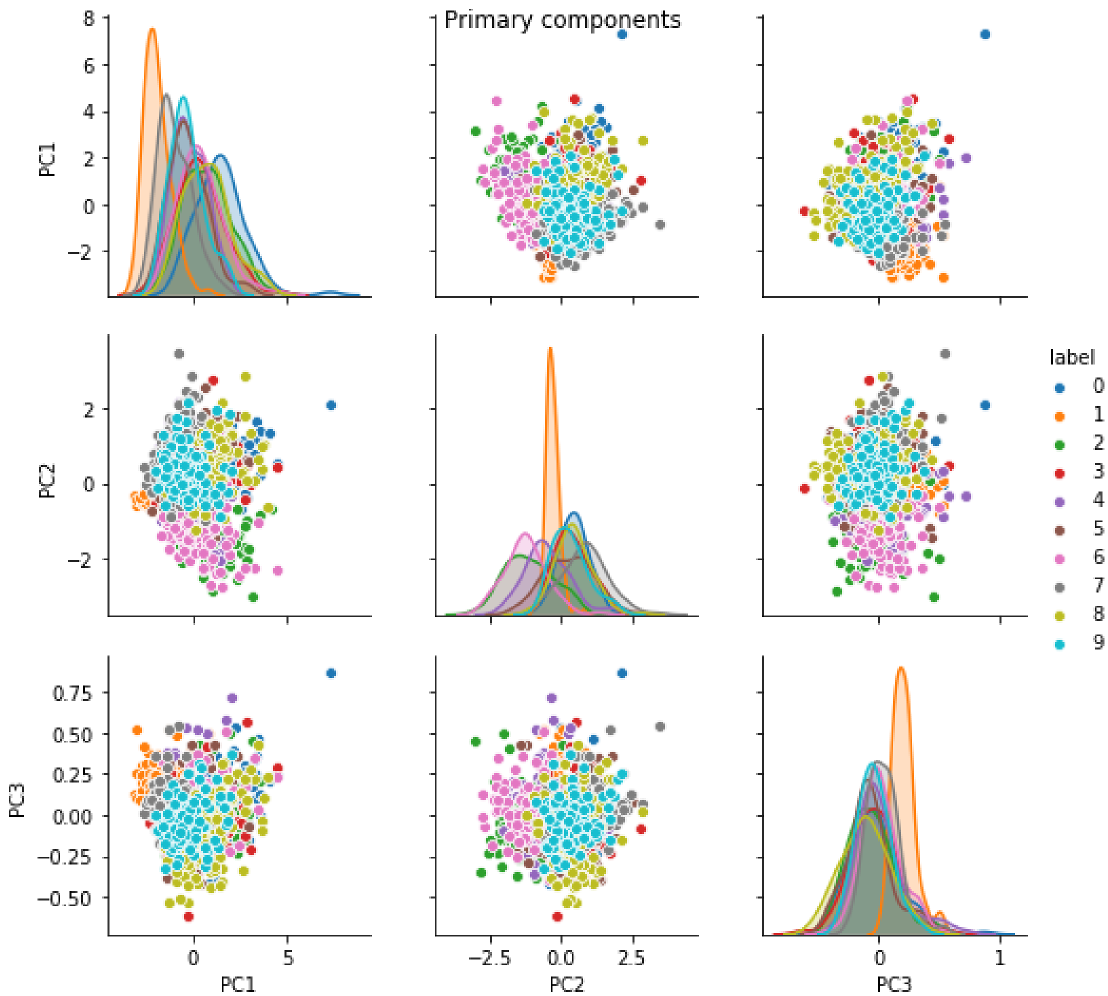
```
[99]: def pairplot(df, title):
          plot = sns.pairplot(df, vars=df.columns[:-1], hue='label')
          plot.fig.suptitle(title)

      df_standart = pd.DataFrame(X_train_norm, columns=features)
      df_rescale = pd.DataFrame(rescaled_X, columns=features)
      df_transform = pd.DataFrame(transformed_array, columns=['PC' + str(i) for i in␣
       ↪range(1, len(features) + 1)])

      for df in (df_standart, df_rescale, df_transform):
          df['label'] = train['label']

      pairplot(df_standart, 'Standardized (mean=0, std=1)')
      pairplot(df_rescale, 'Normalized (range=[0, 100])')
      pairplot(df_transform, 'Primary components')
```



Standardized (mean=0, std=1)
mean
count
vert_symmetry
label
0
1
2
3
4
5
6
7
8
9

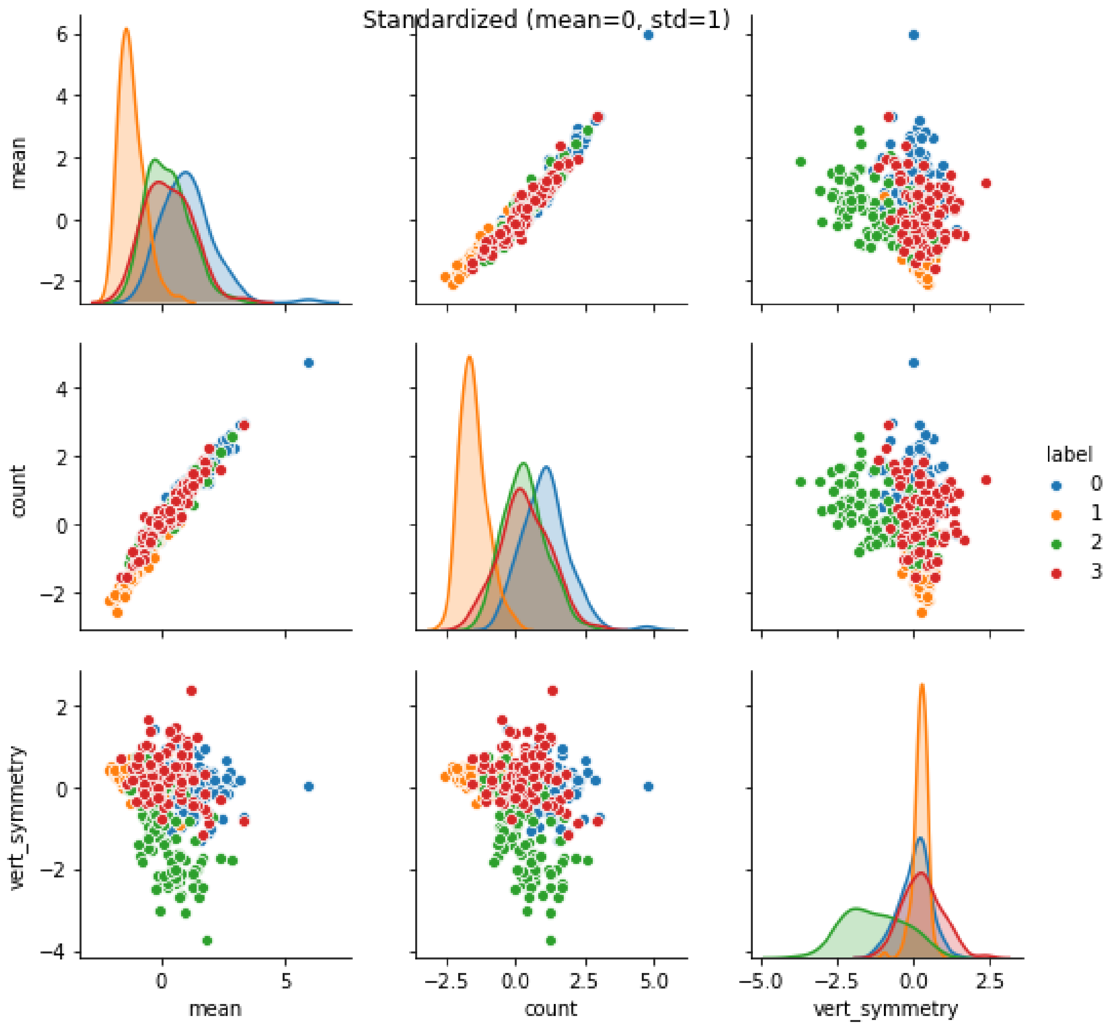


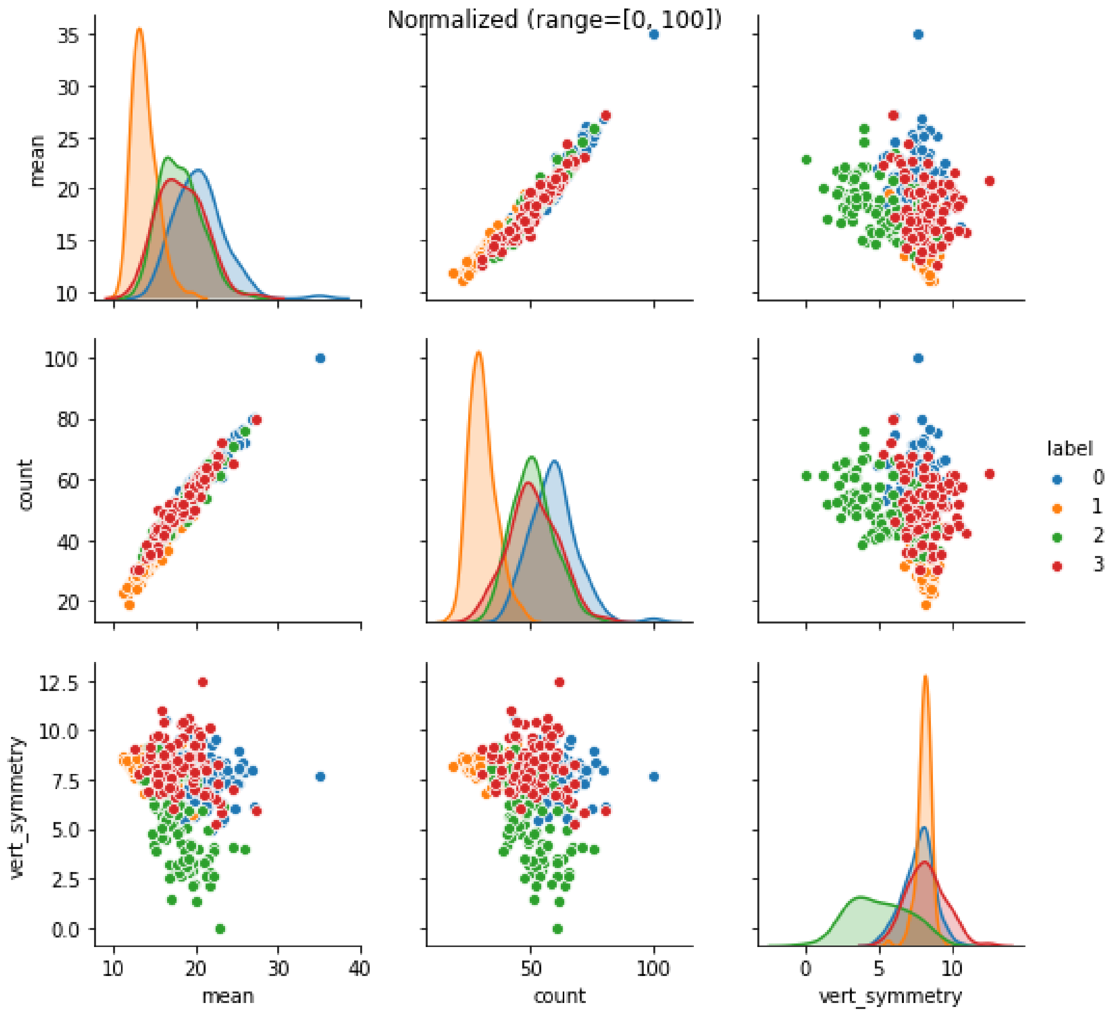
Normalized (range=[0, 100])
mean
count
vert_symmetry
label
0
1
2
3
4
5
6
7
8
9

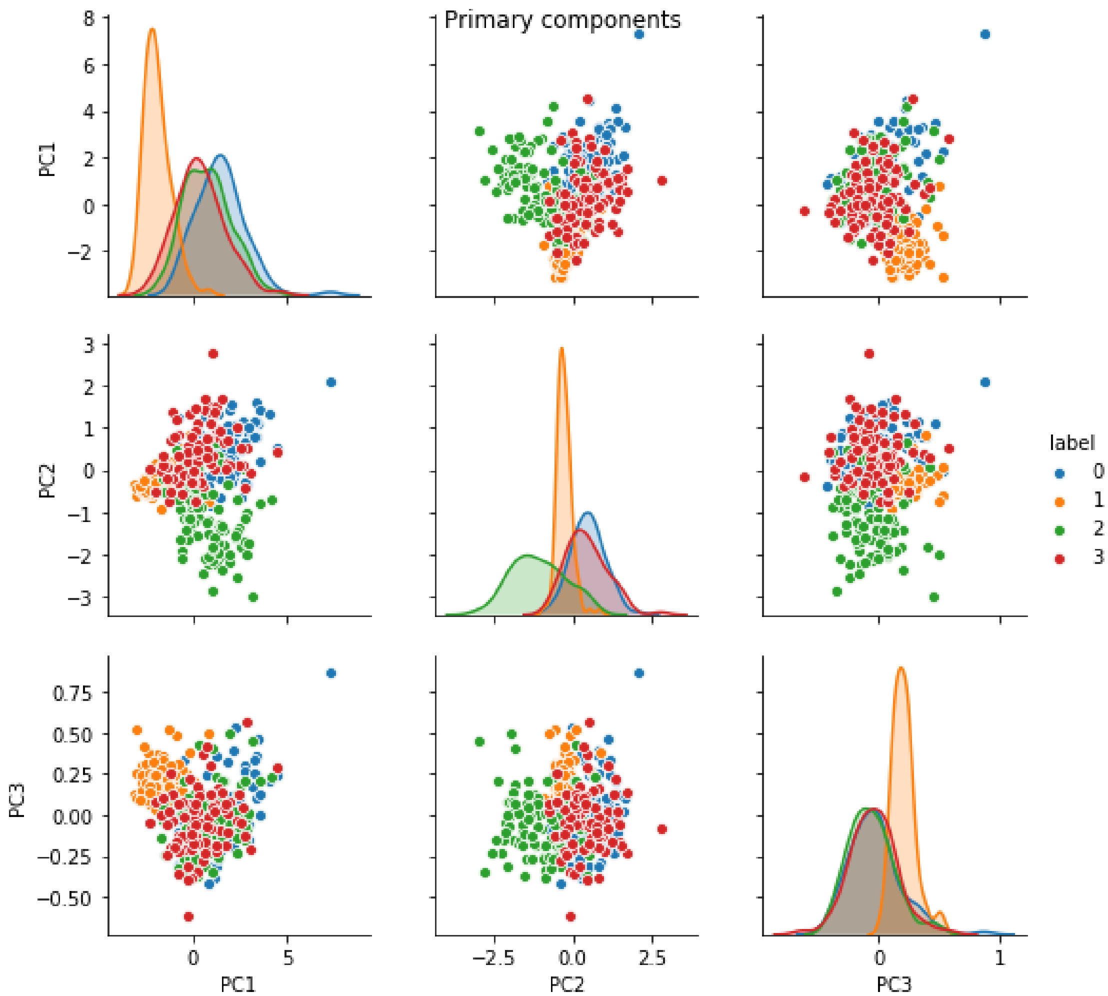


Too difficult understand something in case with many labels. Visualize only first 4 digits.

[100]:
```
indexes = train['label'].values <= 3
df_standart, df_rescale, df_transform = [df[indexes] for df in (df_standart,␣
↪df_rescale, df_transform)]

pairplot(df_standart, 'Standardized (mean=0, std=1)')
pairplot(df_rescale, 'Normalized (range=[0, 100])')
pairplot(df_transform, 'Primary components')
```


Standardized (mean=0, std=1)
mean
count
vert_symmetry
label
0
1
2
3


Normalized (range=[0, 100])
mean
count
vert_symmetry
label
0
1
2
3
10
20
30
40
50
100
0
5
10
35
30
25
20
15
10
100
80
60
40
20
12.5
10.0
7.5
5.0
2.5
0.0



Obviously, since we only use line plots and pairwise scatter plots, the difference between standardization and normalization amounts to relabeling of the axes (which might be helpful by itself because it makes interpreting the coordinates in the graph conceptually easier).

However, if we were, for example, using a 3D scatter plot with fixed scales of the axes, normalization into the [0, 1] range would help a lot as it guarantees that the features have the same scale (if there are no outliers).

PCA doesn't seem to help a lot with distinguishing the points with digits labeling. All features can divede ones from others class, but with other digits all features have problems. But PC2 can divide all digits on three gropus: {2, 4, 6}, {1}, {0, 3, 5, 7, 8, 9}

Correlation coefficient

Features

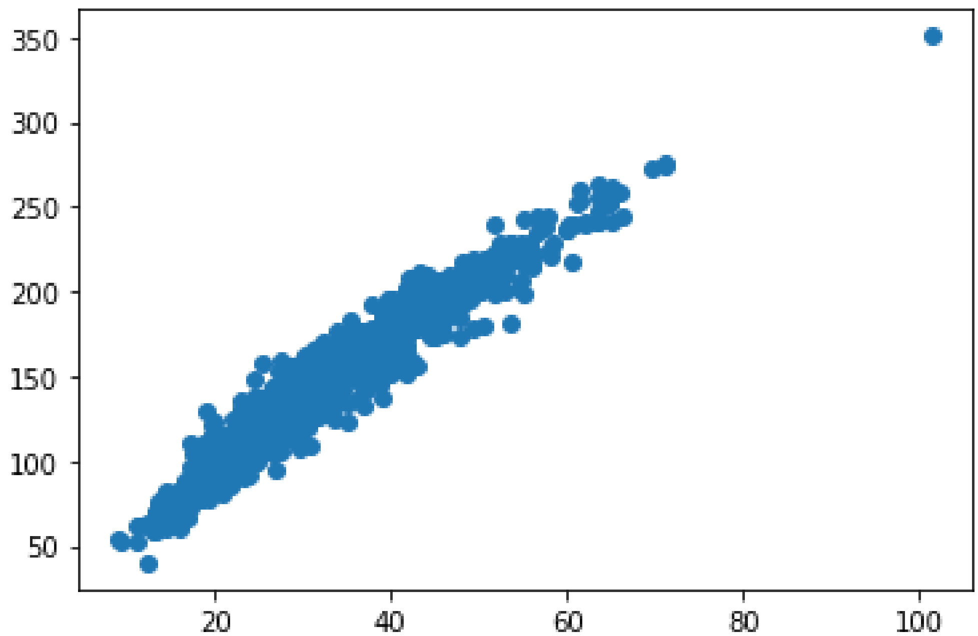
We have get two features mean and count, because this features high correlated. Mean it's sum of values of all pixels on image. Count it's count of nonzero pixels on image.

[5]:
```
x_feature = 'mean'
y_feature = 'count'
```

[6]:
```
X_train = train[x_feature].values
X_test = test[x_feature].values
Y_train = train[y_feature].values
Y_test = test[y_feature].values
```

Visualize

[7]:
```
plt.plot(X_train, Y_train, 'o')
plt.show()
```



Linear Regression

[8]:
```
model = LinearRegression()
model.fit(X_train.reshape(-1, 1), Y_train.reshape(-1, 1))
```

[8]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

[9]:
```
model.coef_
```

[9]: array([[3.5836792]])

We have coef of linear regression equal to 3.58, that mean, that count of pixels in 3.58 times greater than mean. We know, that mean – it's sum of all pixels divided by $28 \times 28 = 784$. And we know, that large part of pixels equal to 255. It means, that count approximately equal to sum of all pixels divided by 255. In such way we know, that koef .3.58 hight correlated with out knowledge.

Correlation and determinacy

[10]:
```
x_mean = np.mean(X_train)
y_mean = np.mean(Y_train)
cov = np.sum((X_train - x_mean) * (Y_train - y_mean))
x_var = np.sum((X_train - x_mean) ** 2)
y_var = np.sum((Y_train - y_mean) ** 2)
cor = cov / np.sqrt(x_var * y_var)
print('correlation coefficient:', cor)
```

```
correlation coefficient: 0.9630931048114645
```

It means, that features high correlated, such as shown on picture above.

[11]:
```
y_mean = np.mean(Y_test)
y_var = np.sum((Y_test - y_mean) ** 2)
pred_y = model.predict(X_test.reshape(-1, 1)).reshape(-1)
pred_var = np.sum((pred_y - Y_test) ** 2)
det = 1 - pred_var / y_var
print('determinacy coefficient:', det)
```

```
determinacy coefficient: 0.9234620267287004
```

It means, that large part of variance of count variable explains by mean variable.

Prediction sample

Predict three samples from test.

[36]:
```
print('Predictions for 3 points:')
print('\t\ty_true\t\ty_pred')
for i in range(3):
    print('\t\t', Y_test[i], '\t\t {0:.4}'.format(pred_y[i]))
```

```
Predictions for 3 points:
                y_true          y_pred
                 144             144.3
```

127	118.8
188	176.4

Result too close to true value.

MRAE

[38]:
```
mrae = np.mean(np.abs(pred_y - Y_test) / Y_test)
print('Mean relative absolute error:', mrae)
```

```
Mean relative absolute error: 0.06477857524441351
```

We have small MRAE, that can be explain by determinacy coefficient close to one. That means, that model make relatively good predict on whole span of values for given features.