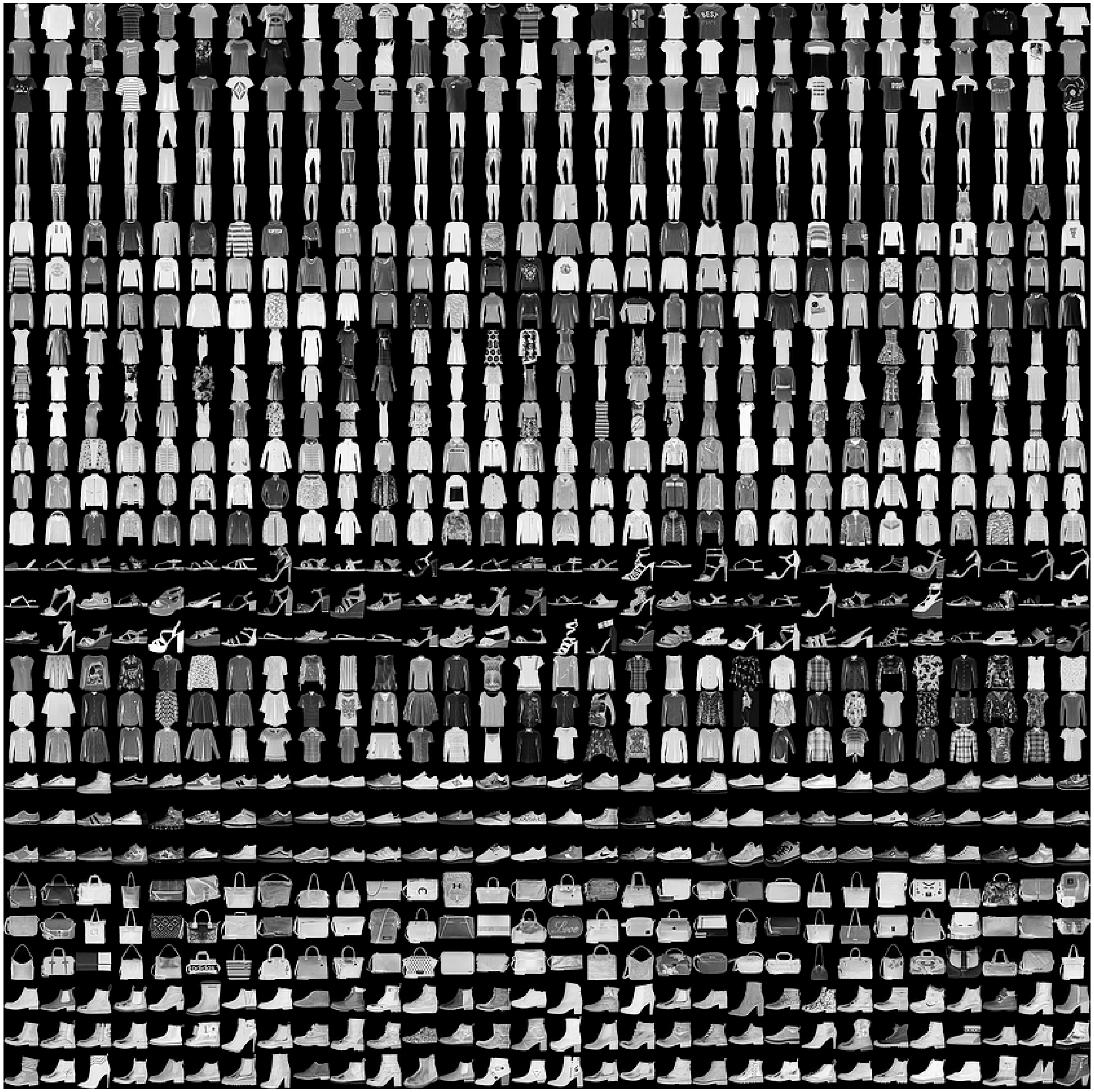
Fashion MNIST

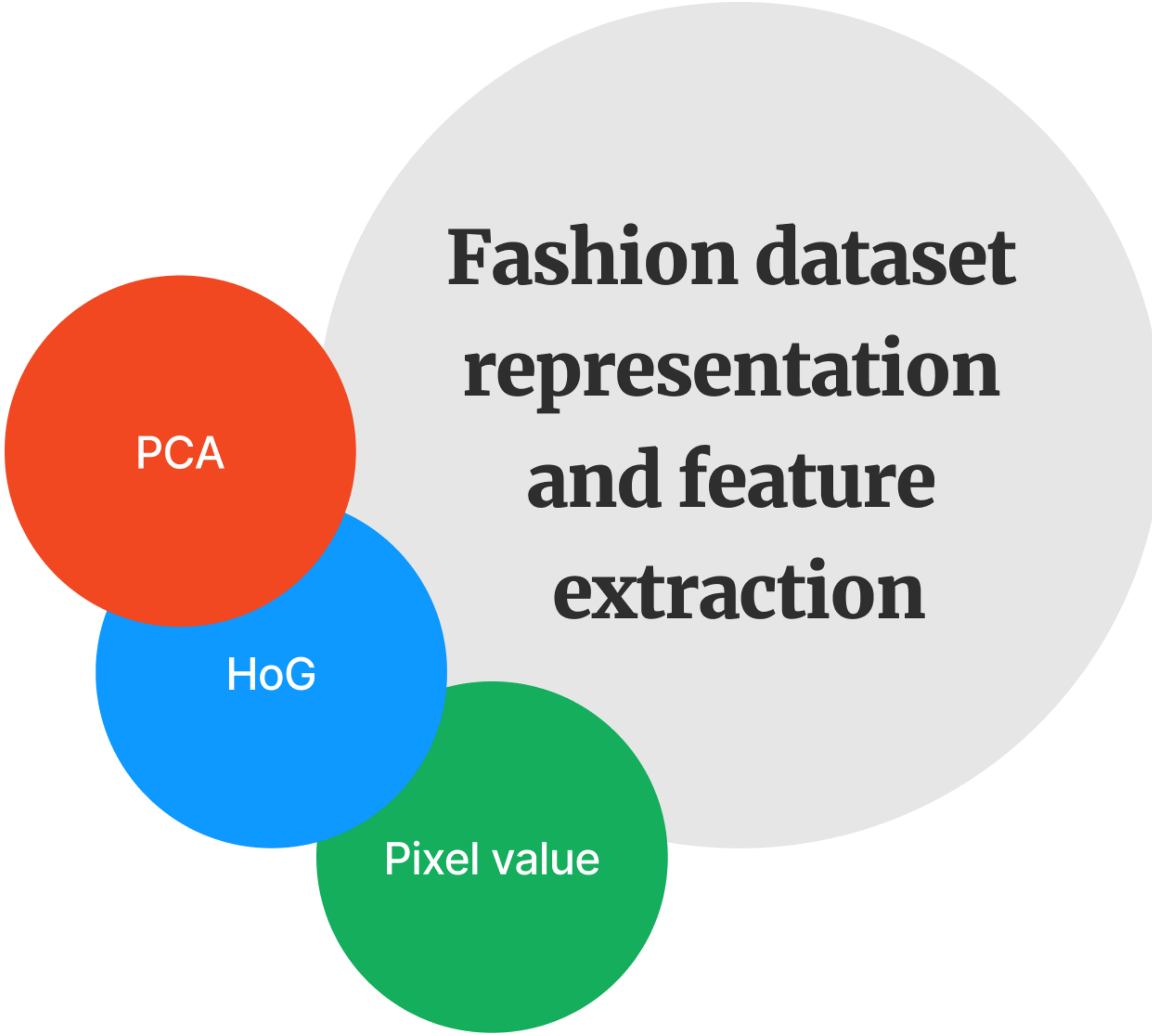
About the dataset

👕 **Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. it was intend for Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.**



Feature Extraction Phase

Fashion dataset representation and feature extraction



1. Pixel value where each image of size(28*28) flatten to compose a a single observation of 784 feature from (pixel1→pixel784) ana a label for each observation(image🖼️)

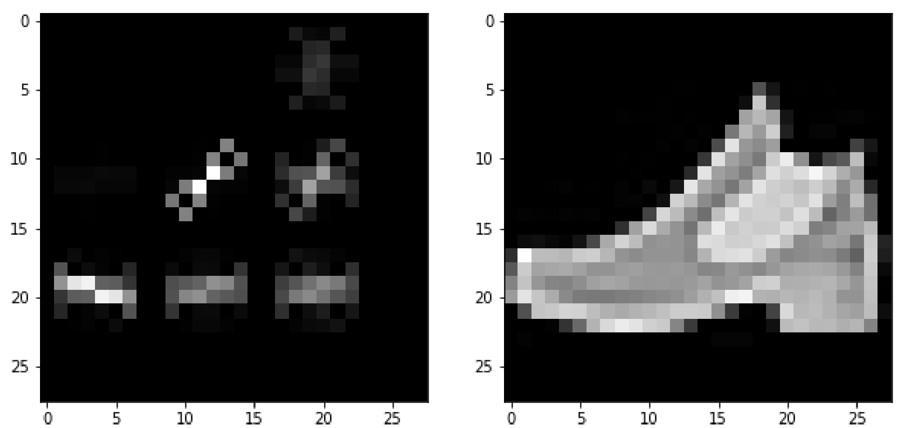
	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783	pixel784
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0	0	0
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Head of the train set

2. Histogram Of Oriented Gradients (HOG)

skimage.feature — skimage 0.22.0 documentation
cikit-image
https://scikit-image.org/docs/stable/api/skimage.feature.html#skimage.feature.hog

```
hog(image, orientations=8, visualize=True)
```



number of features we got for each image (72,) using this parameters

	0	1	2	3	4	5	6	7	8	9	...	62	63	64	65	66	67	68	69	70	71
0	0.057845	0.194040	0.207260	0.068280	0.058807	0.000489	0.000000	0.003777	0.011969	0.084600	...	0.025372	0.014512	0.237629	0.067363	0.006902	0.041049	0.087453	0.201157	0.081416	0.065727
1	0.000394	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001181	0.000000	...	0.013579	0.030951	0.041209	0.020112	0.198062	0.270952	0.259899	0.092271	0.074821	0.047910
2	0.257989	0.225401	0.105295	0.028898	0.009615	0.000000	0.001612	0.013724	0.083109	0.166780	...	0.046172	0.059946	0.290640	0.022658	0.068237	0.000000	0.022346	0.039562	0.020390	0.290640
3	0.128701	0.175918	0.190101	0.058635	0.044771	0.016389	0.018411	0.001277	0.153229	0.044253	...	0.148202	0.166557	0.127487	0.056144	0.091727	0.088432	0.046796	0.080052	0.097072	0.205857
4	0.257131	0.141440	0.000000	0.000000	0.000000	0.000000	0.000000	0.040207	0.135338	0.046181	...	0.047053	0.253988	0.265136	0.000772	0.000244	0.000546	0.002763	0.000000	0.000867	0.166677

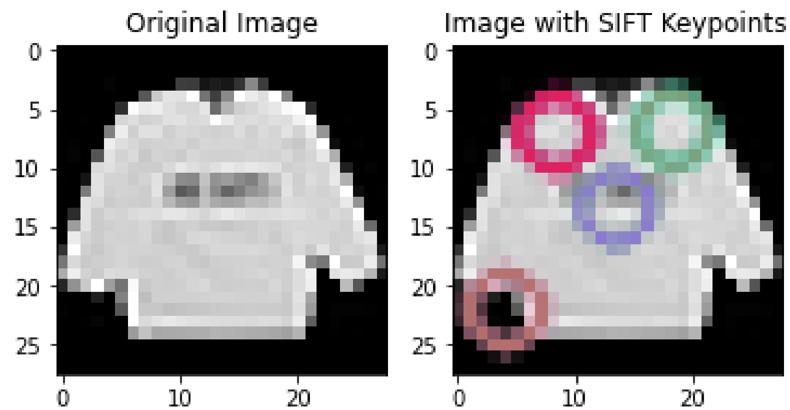
HOG features

3. Principle Component Analysis (PCA) where unsupervised method to dimensions reduction by trying to capture most of the variance in n-component, in our case we choose to take 40 feature out of all the 784 pixel features

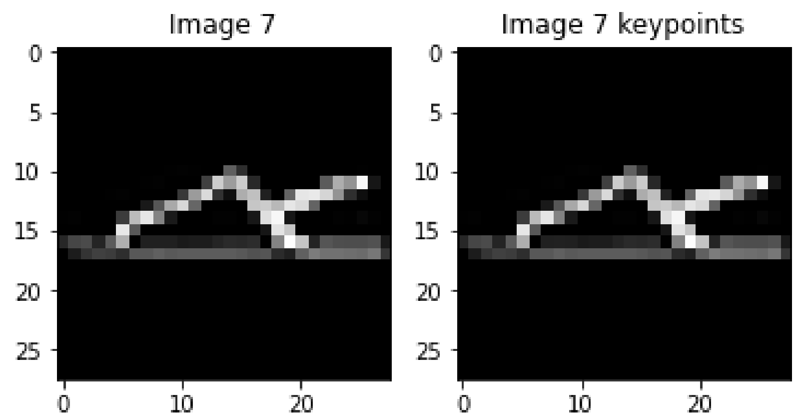
	0	1	2	3	4	5	6	7	8	9	...	30	31	32	33	34	35	36	37	38	39
0	10.664973	14.993363	-0.689468	-10.980911	4.788254	0.559623	2.141115	-2.582798	-4.525370	-0.758689	...	-1.530579	1.394830	-2.278333	-2.294341	-2.454768	-2.905128	-1.991745	0.108772	-1.531449	4.137939
1	-11.989748	11.812770	-5.801049	-3.418629	-4.630650	2.061772	-3.109565	-3.308504	-4.461980	5.669063	...	-0.467369	1.778804	-0.102749	0.320264	1.653639	-2.177671	1.208585	-2.240644	0.649311	-1.785557
2	20.517671	1.579784	6.770122	-2.884371	-5.379185	2.618096	-0.246181	-2.936941	3.415352	-1.236736	...	0.260708	-0.345817	-0.378379	0.305361	0.547448	0.535585	-0.082853	-0.491304	0.785215	-1.202624
3	9.634535	-6.790917	-0.907860	4.577228	8.377438	-0.094376	-8.672806	-0.604500	-2.566944	-3.070932	...	0.635726	1.108873	0.650098	-0.662520	-1.287922	2.529832	1.885708	0.995510	0.101587	2.176586
4	11.493442	-11.655488	-7.208334	-4.772964	-0.594114	0.553685	0.388620	-0.384219	2.063611	-4.824827	...	-1.518739	-1.130757	0.420996	-2.294038	0.773056	-2.698066	2.177149	1.082523	0.384146	0.839651

PCA features

4. Scale-Invariant Feature Transform (SIFT), we tried to see if we could use SIFT for feature extraction it didn’t work the listed reasons
   a. Not all the images we can extract from it SIFT key-points due to the small size of our images(28×28)



SIFT here get some key-points



No key-points at all

b. SIFT don’t give us a constant number of key points for each image it vary from image to image

c. in our use-case SIFT was not suitable it work better in

- Object detection
- Image matching
- Large images

👕 Note we used the **same PCA** object we **fit on the train(fit-transform)** data **to transform the test data (transform-only)**, that is important step as we would use only what was fit on the train to transform any new test observation or any unseen data, this PCA object would be saved for later use in deployment using `joblib`

Model Building and Evaluation

```
# Spliting for Pixel Value dataset 🟢
X_train_px, X_test_px, y_train_px, y_test_px = split_df(train_scaled, test_scaled)
# Spliting for Hog 🔵
X_train_hog, X_test_hog, y_train_hog, y_test_hog = split_df(train_hog_df, test_hog_df)
# Spliting for PCA 🔴
X_train_40pca, X_test_40pc, y_train_40pca, y_test_40pca = split_df(train_pca40_df, test_pca40_df)
```

Notes about Logistic Regression parameters

You can use Logistic regression for multi-classification problem by one of the two ways:

`multi-class ='ovr'`	`multi-class ='multinomial'`
one-vs-rest (OvR) scheme	**use cross-entropy loss**
	supported only by the `'lbfgs', 'sag', 'saga'` **and** `'newton-cg'` **solvers.**

the default is auto `multi_class='auto'`

multiclass	binary
`'newton-cg''sag''saga''lbfgs'` ***handle mutinomial loss, auto, multinomial***	***OvR, auto***

```
if auto is set :
    if liblinear or binary:
```

```
    select OvR
else:
    select multinomial
```

👕 >OvR: separate binary classifier is trained to distinguish that class from all the other classes combined. So, if you have K classes, you train K binary classifiers.

>The prediction is then made by running all K classifiers on a test instance and selecting the class for which the corresponding classifier gives the highest confidence or probability.

Solvers and their suitable penalty

`elasticnet`: both L1 and L2 penalty terms are added

`l1`	`l2`	`elasticnet`	None
`'liblinear''saga'`	`'lbfgs' 'liblinear''newton-cg''sag''saga'`	only with `saga`	`'lbfgs''newton-cg''sag''saga'`

Small datasets	Large datesets
`'liblinear'`	`'sag''saga'`

👕 We used solver `'saga'` as it faster with large dataset we used `max-iter` as lower as possible to make the model training in a faster time almost from (10min→2.5s) that is helpful for testing and in grid search when we try alot of models or using cross-validation

`max-iter` default is 100, we used in grid-search and cross-validation `max-iter = 1` and on the final model with best parameters I we used `max-iter = 3` **, another reason why higher number of iteration is not helpful that it won't converge so wasting time in iteration won't give us a worthy increasing in performance.**

Logistic Regression on pixel features

- Initial model

```
logReg = LogisticRegression(solver='saga', max_iter=3)
```

Train score 0.8609
Test score 0.8571
Cross Validation 0.8535666666666668 (5 Folds)

- Grid Search to find the best solver and best C

c →Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

```
logRegCV = LogisticRegression(max_iter=1)

# params to try
params = {
    'C' : [10, 100, 0.1],
    'solver': ['saga', 'newton-cg']
}
```

Best Hyperparameters: {'C': 100, 'solver': 'saga'}
Best Cross-Validated Accuracy: 0.8376833333333334 (3 Folds)

who run first	mean_fit_time	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	rank_test_score
0	51.65	{'C': 10, 'solver': 'saga'}	0.83475	0.8355	0.84055	**0.836933**	2
1	64.95482	{'C': 10, 'solver': 'newton-cg'}	0.6282	0.6186	0.62985	**0.62555**	4
2	66.09061	{'C': 100, 'solver': 'saga'}	0.83835	0.84145	0.83325	**0.837683**	1 🌟
3	83.47487	{'C': 100, 'solver': 'newton-cg'}	0.6282	0.6186	0.62985	**0.62555**	4

who run first	mean_fit_time	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	rank_test_score
4	2.262331	{'C': 0.1, 'solver': 'saga'}	0.83315	0.83565	0.8381	**0.835633**	3
5	1.6846	{'C': 0.1, 'solver': 'newton-cg'}	0.6282	0.6186	0.62985	**0.62555**	4

- Grid Search to find whether OVR or Multinomial

```
logRegCV = LogisticRegression(max_iter=1, solver='saga')

# params to try
params = {
    'C' : [10, 100, 0.1],
    'multi_class': ['ovr', 'multinomial']
}
```

Best Hyperparameters: {'C': 10, 'multi_class': 'multinomial'}
Best Cross-Validated Accuracy: 0.8422000000000001

who run first	mean_fit_time	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	rank_test_score
0	76.76555	{'C': 10, 'multi_class': 'ovr'}	0.8364	0.8294	0.8375	**0.834433**	4
1	46.08132	{'C': 10, 'multi_class': 'multinomial'}	0.8426	0.84195	0.84205	**0.8422**	1🌟
2	90.49641	{'C': 100, 'multi_class': 'ovr'}	0.8297	0.82315	0.83265	**0.8285**	6
3	84.42747	{'C': 100, 'multi_class':	0.8395	0.82415	0.84125	**0.834967**	3

who run first	mean_fit_time	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	rank_test_score
		'multinomial'}					
4	6.666688	{'C': 0.1, 'multi_class': 'ovr'}	0.83225	0.83575	0.82885	**0.832283**	5
5	4.678921	{'C': 0.1, 'multi_class': 'multinomial'}	0.84075	0.83675	0.84145	**0.83965**	2

- Final Model

```
logRegFinal = LogisticRegression(C=10, max_iter=3, solver='saga')
```

Train score 0.8631666666666666
Test score 0.8574

Cross Validation 6 folds [0.8565 0.8534 0.857 0.8511 0.8488 0.8507]

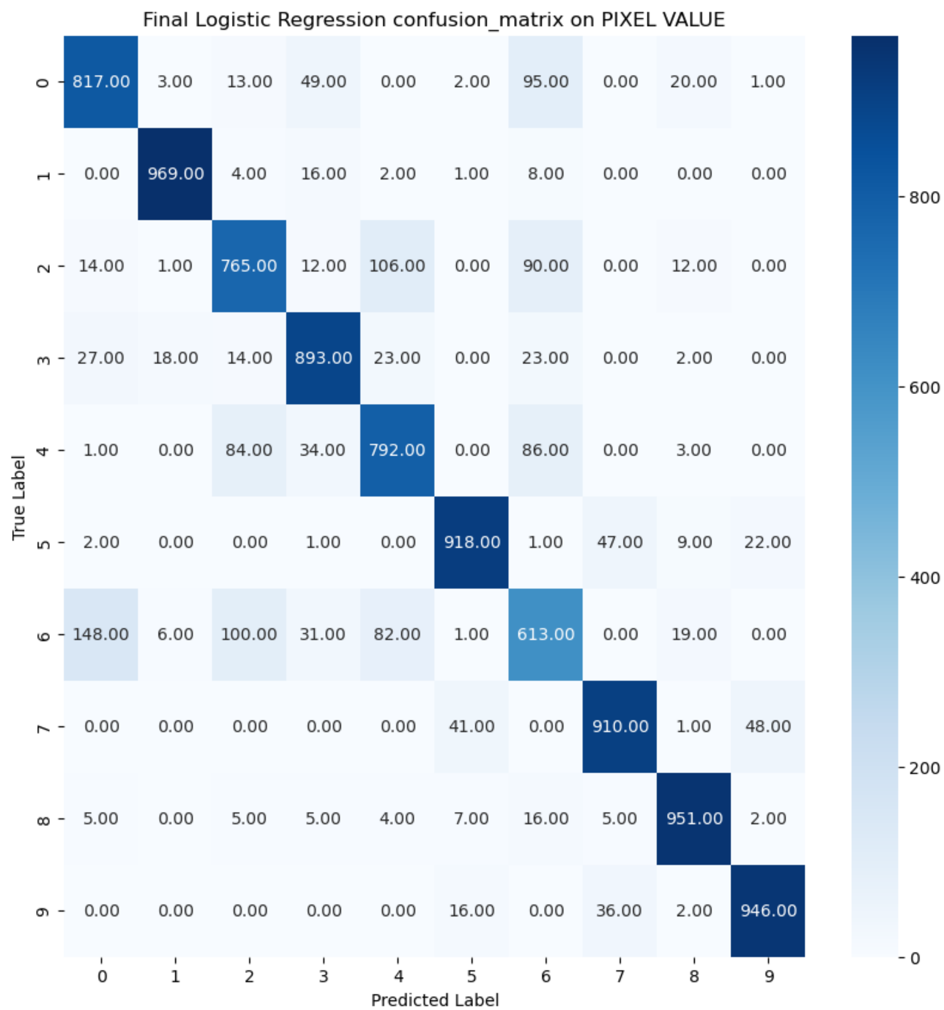
mean cross validation for 6 folds 0.8529166666666667

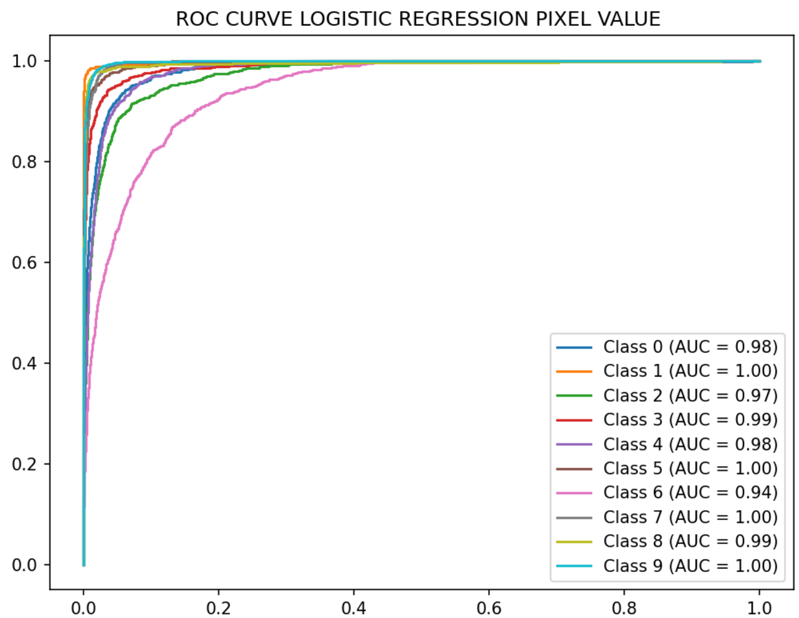
precision   recall f1-score   support

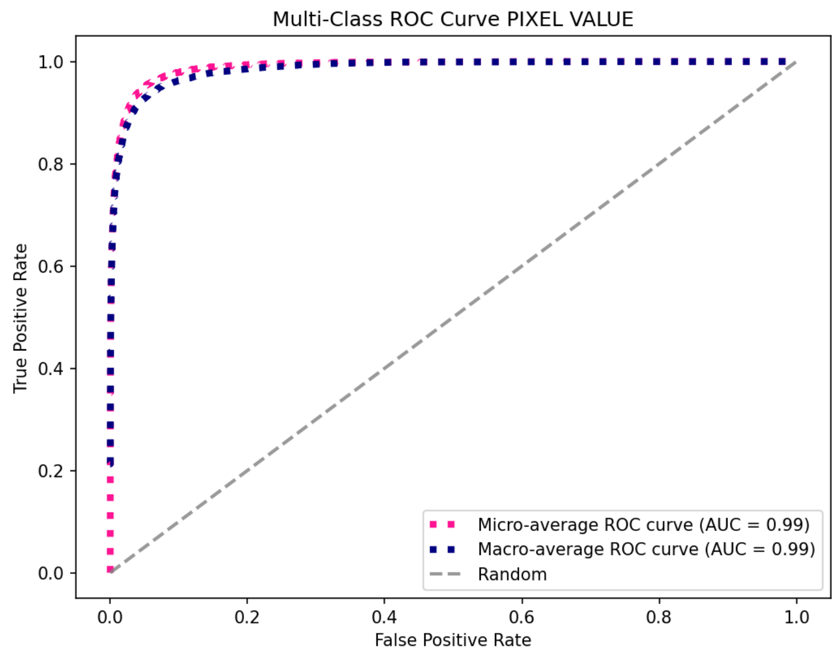
```
        0       0.81      0.82      0.81      1000
        1       0.97      0.97      0.97      1000
        2       0.78      0.77      0.77      1000
        3       0.86      0.89      0.88      1000
        4       0.78      0.79      0.79      1000
        5       0.93      0.92      0.92      1000
        6       0.66      0.61      0.63      1000
        7       0.91      0.91      0.91      1000
        8       0.93      0.95      0.94      1000
        9       0.93      0.95      0.94      1000

 accuracy                           0.86     10000
```

```
macro avg        0.86      0.86      0.86      10000
weighted avg     0.86      0.86      0.86      10000
```







Logistic Regression on HOG features

```
logRegHog = LogisticRegression(solver='saga')
```

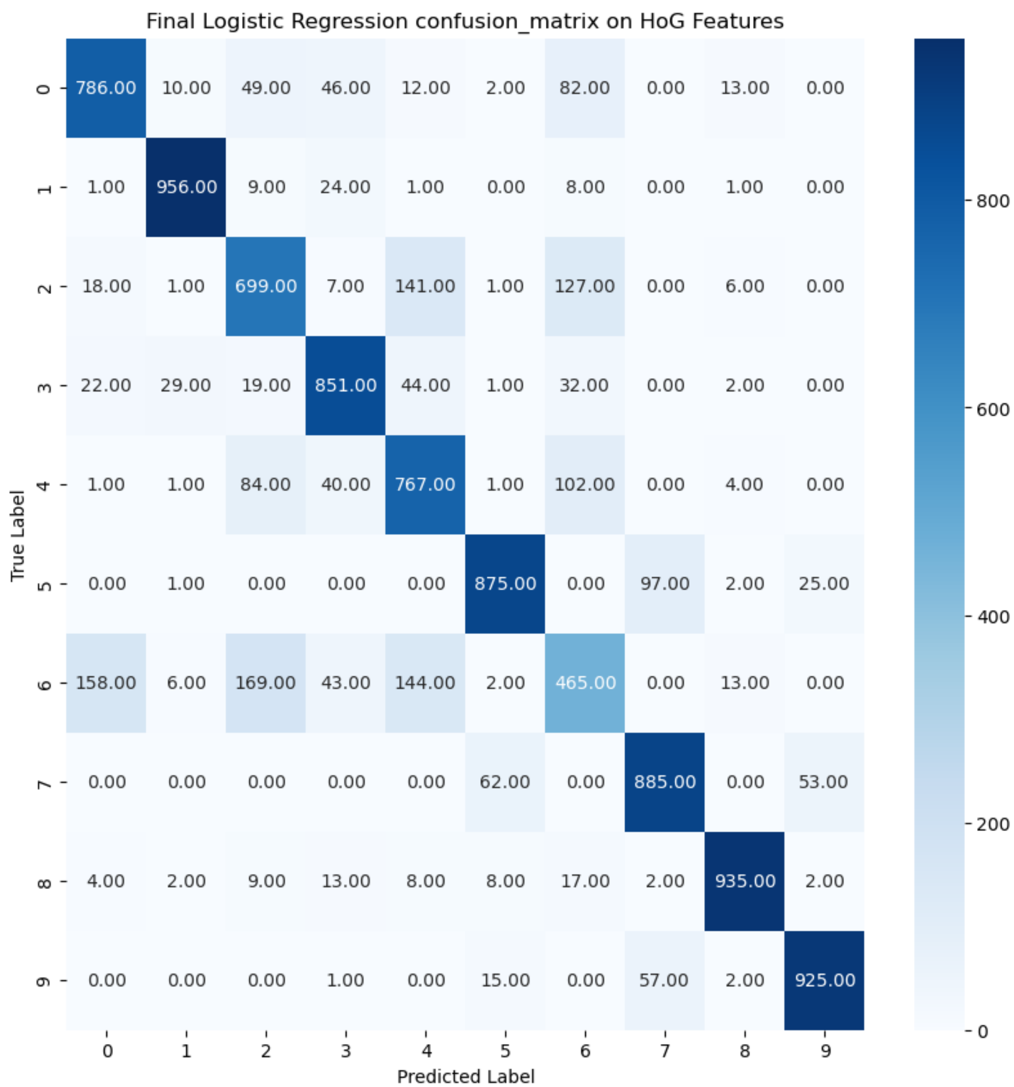
Train score 0.81395
Test score 0.8144
Cross Validation 0.8107000000000001 (6 Folds)

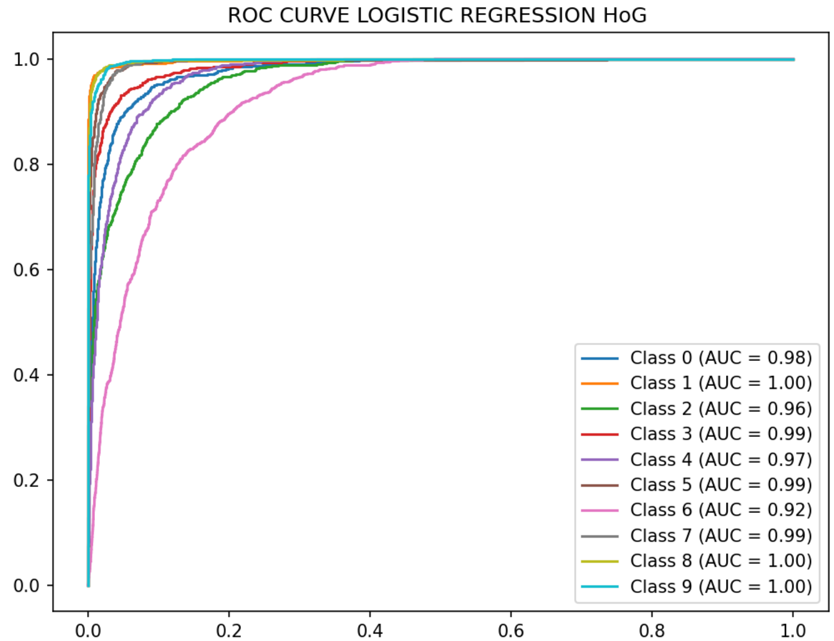
precision recall f1-score support

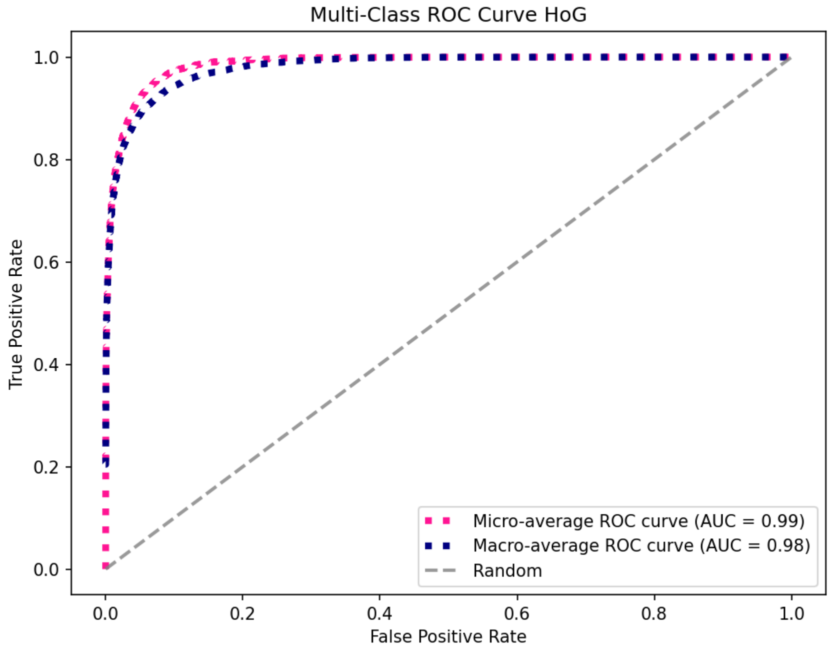
```
       0       0.79      0.79      0.79      1000
       1       0.95      0.96      0.95      1000
       2       0.67      0.70      0.69      1000
```

```
           3       0.83      0.85      0.84      1000
           4       0.69      0.77      0.72      1000
           5       0.90      0.88      0.89      1000
           6       0.56      0.47      0.51      1000
           7       0.85      0.89      0.87      1000
           8       0.96      0.94      0.95      1000
           9       0.92      0.93      0.92      1000

    accuracy                           0.81     10000
   macro avg       0.81      0.81      0.81     10000
weighted avg       0.81      0.81      0.81     10000
```


Final Logistic Regression confusion_matrix on HoG Features
True Label
Predicted Label
786.00 10.00 49.00 46.00 12.00 2.00 82.00 0.00 13.00 0.00
1.00 956.00 9.00 24.00 1.00 0.00 8.00 0.00 1.00 0.00
18.00 1.00 699.00 7.00 141.00 1.00 127.00 0.00 6.00 0.00
22.00 29.00 19.00 851.00 44.00 1.00 32.00 0.00 2.00 0.00
1.00 1.00 84.00 40.00 767.00 1.00 102.00 0.00 4.00 0.00
0.00 1.00 0.00 0.00 0.00 875.00 0.00 97.00 2.00 25.00
158.00 6.00 169.00 43.00 144.00 2.00 465.00 0.00 13.00 0.00
0.00 0.00 0.00 0.00 0.00 62.00 0.00 885.00 0.00 53.00
4.00 2.00 9.00 13.00 8.00 8.00 17.00 2.00 935.00 2.00
0.00 0.00 0.00 1.00 0.00 15.00 0.00 57.00 2.00 925.00
0 1 2 3 4 5 6 7 8 9
0
200
400
600
800





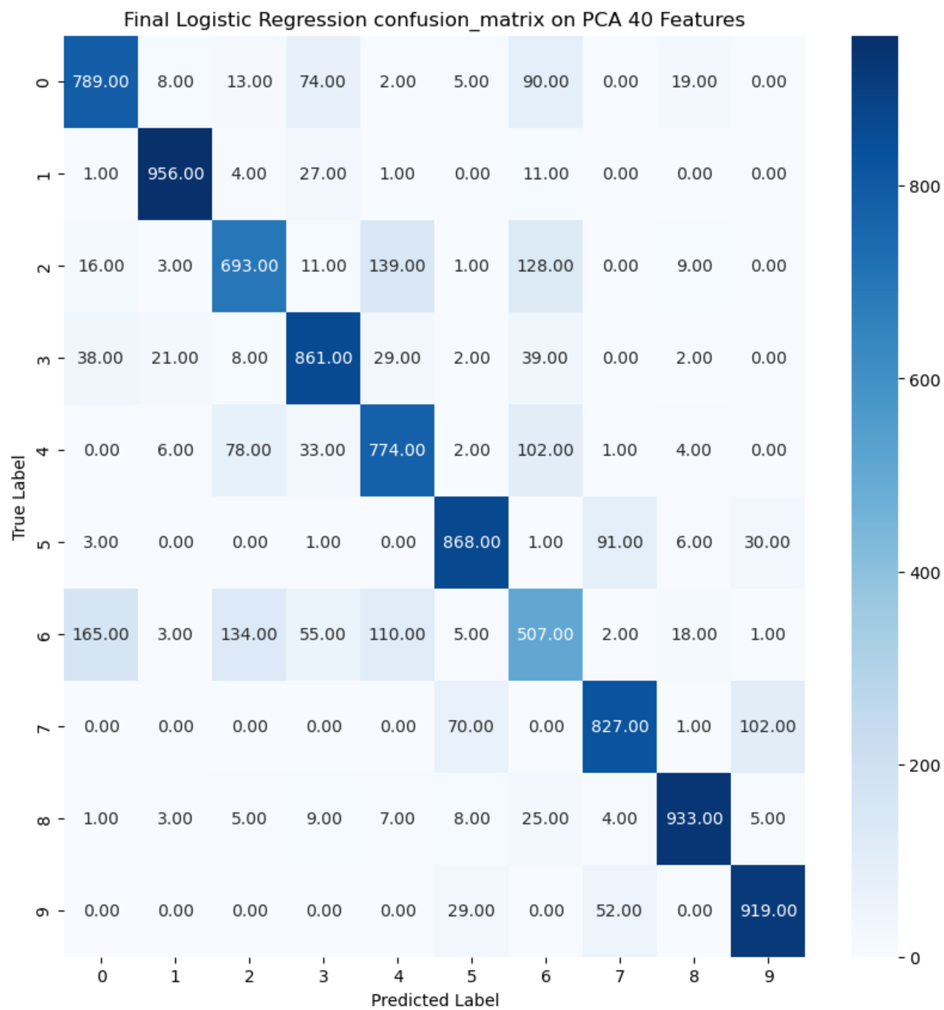
Logistic Regression on PCA features

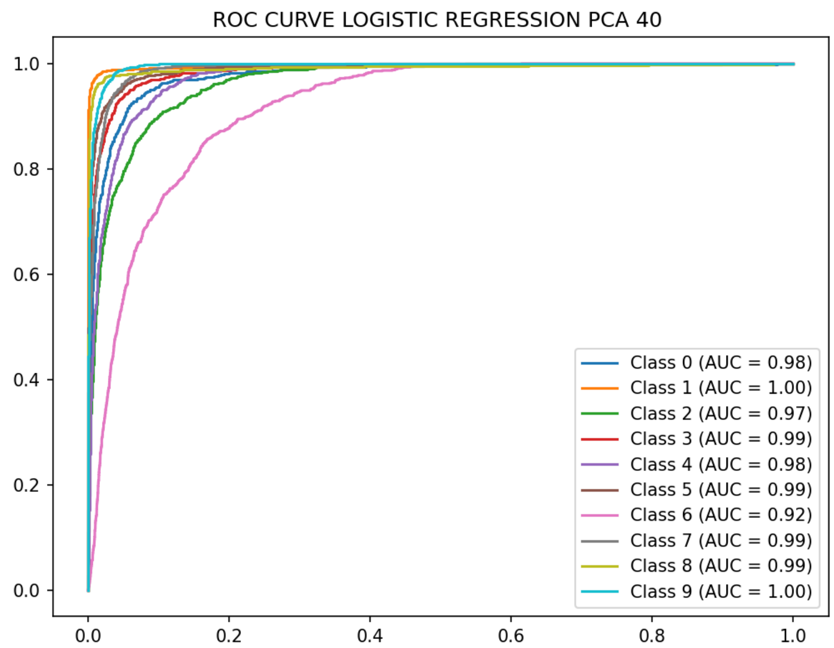
Train score 0.8199666666666666
Test score 0.8208
Cross Validation 0.8182666666666666 (6 Folds)

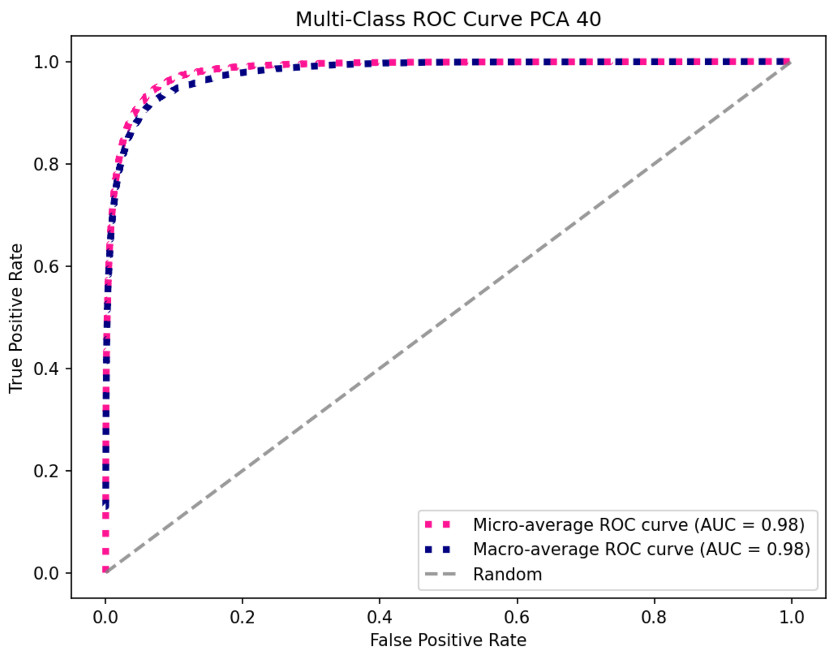
	precision	recall	f1-score	support
0	0.78	0.79	0.78	1000
1	0.96	0.96	0.96	1000
2	0.74	0.69	0.72	1000
3	0.80	0.86	0.83	1000
4	0.73	0.77	0.75	1000
5	0.88	0.87	0.87	1000

```
           6       0.56      0.51      0.53      1000
           7       0.85      0.83      0.84      1000
           8       0.94      0.93      0.94      1000
           9       0.87      0.92      0.89      1000

    accuracy                           0.81     10000
   macro avg       0.81      0.81      0.81     10000
weighted avg       0.81      0.81      0.81     10000
```


Final Logistic Regression confusion_matrix on PCA 40 Features
True Label
Predicted Label
789.00 8.00 13.00 74.00 2.00 5.00 90.00 0.00 19.00 0.00
1.00 956.00 4.00 27.00 1.00 0.00 11.00 0.00 0.00 0.00
16.00 3.00 693.00 11.00 139.00 1.00 128.00 0.00 9.00 0.00
38.00 21.00 8.00 861.00 29.00 2.00 39.00 0.00 2.00 0.00
0.00 6.00 78.00 33.00 774.00 2.00 102.00 1.00 4.00 0.00
3.00 0.00 0.00 1.00 0.00 868.00 1.00 91.00 6.00 30.00
165.00 3.00 134.00 55.00 110.00 5.00 507.00 2.00 18.00 1.00
0.00 0.00 0.00 0.00 0.00 70.00 0.00 827.00 1.00 102.00
1.00 3.00 5.00 9.00 7.00 8.00 25.00 4.00 933.00 5.00
0.00 0.00 0.00 0.00 0.00 29.00 0.00 52.00 0.00 919.00
0 1 2 3 4 5 6 7 8 9
0
200
400
600
800





mix1 (PCA🔴+HOG🔵) 🟣

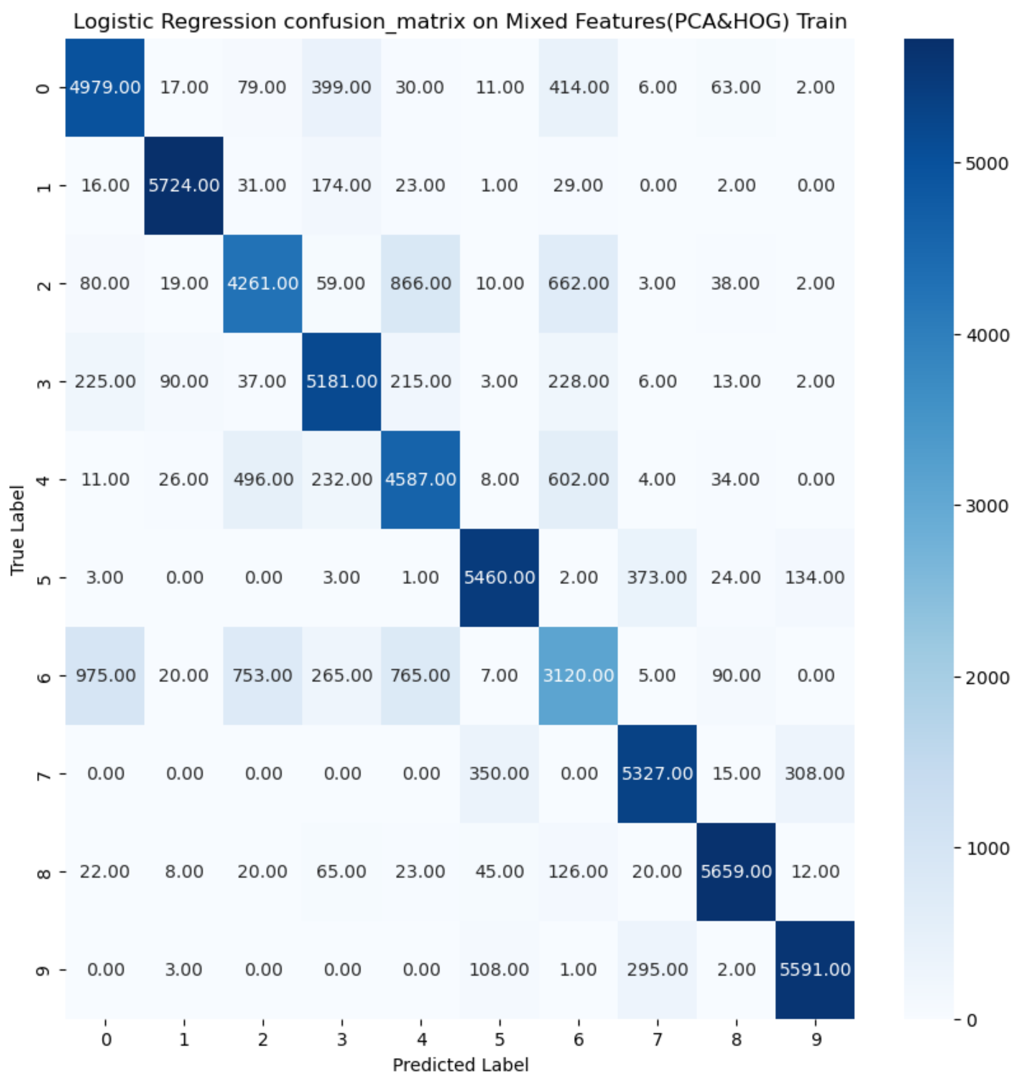
```
logRegMix1 = LogisticRegression(solver='saga', max_iter=300)
# ⚠️ 300 iterations (3.14 mins) to reach this performance
```

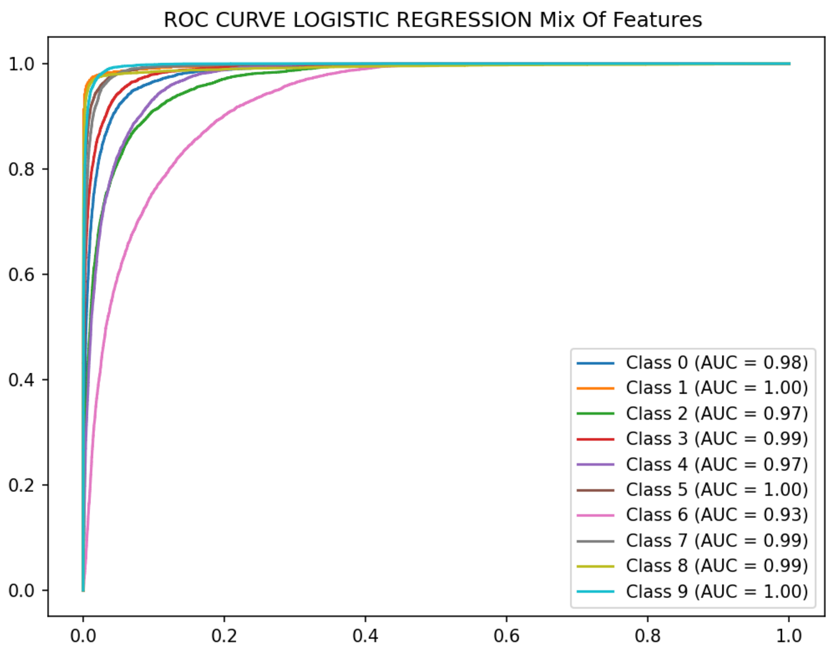
Train score 0.8314833333333334
Cross Validation 0.8300666666666667 (6 Folds)

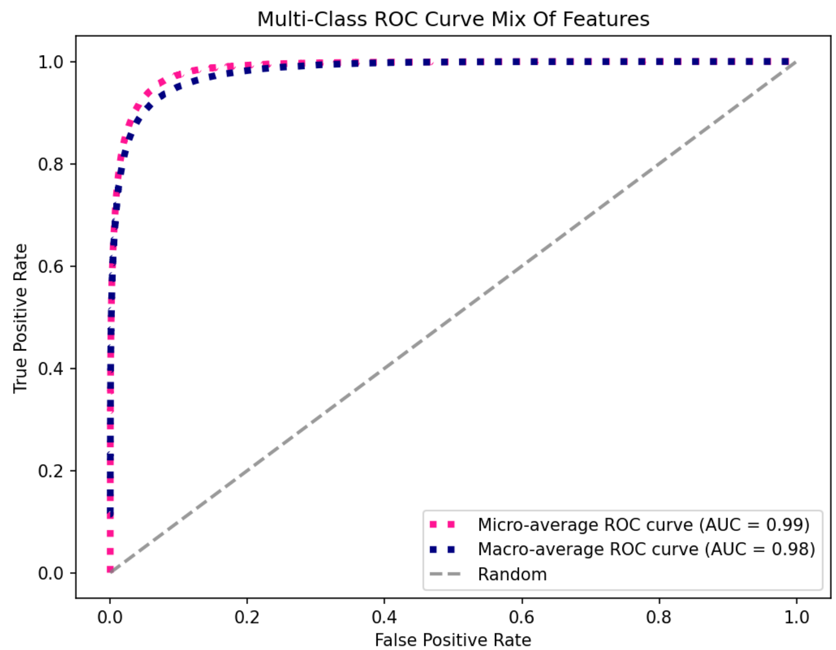
	precision	recall	f1-score	support
0	0.79	0.83	0.81	6000
1	0.97	0.95	0.96	6000
2	0.75	0.71	0.73	6000

```
           3       0.81      0.86      0.84      6000
           4       0.70      0.76      0.73      6000
           5       0.91      0.91      0.91      6000
           6       0.60      0.52      0.56      6000
           7       0.88      0.89      0.88      6000
           8       0.95      0.94      0.95      6000
           9       0.92      0.93      0.93      6000

    accuracy                           0.83     60000
   macro avg       0.83      0.83      0.83     60000
weighted avg       0.83      0.83      0.83     60000
```


Logistic Regression confusion_matrix on Mixed Features(PCA&HOG) Train
True Label
Predicted Label
0 1 2 3 4 5 6 7 8 9
4979.00 17.00 79.00 399.00 30.00 11.00 414.00 6.00 63.00 2.00
16.00 5724.00 31.00 174.00 23.00 1.00 29.00 0.00 2.00 0.00
80.00 19.00 4261.00 59.00 866.00 10.00 662.00 3.00 38.00 2.00
225.00 90.00 37.00 5181.00 215.00 3.00 228.00 6.00 13.00 2.00
11.00 26.00 496.00 232.00 4587.00 8.00 602.00 4.00 34.00 0.00
3.00 0.00 0.00 3.00 1.00 5460.00 2.00 373.00 24.00 134.00
975.00 20.00 753.00 265.00 765.00 7.00 3120.00 5.00 90.00 0.00
0.00 0.00 0.00 0.00 0.00 350.00 0.00 5327.00 15.00 308.00
22.00 8.00 20.00 65.00 23.00 45.00 126.00 20.00 5659.00 12.00
0.00 3.00 0.00 0.00 0.00 108.00 1.00 295.00 2.00 5591.00
5000
4000
3000
2000
1000
0





K-means with (5-classes subset of the data)

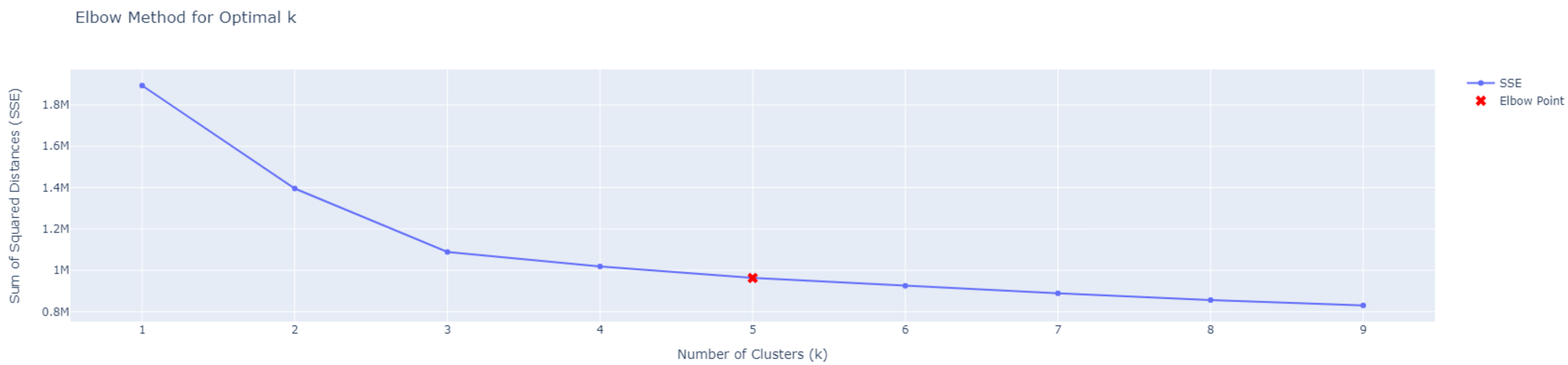
Sum OF Squared Error (SSE)

>The goal of the k-means algorithm is to minimize this SSE. As the number of clusters (k) increases, the SSE tends to decrease because each cluster has fewer points, and centroids are closer to the data points. However, after a certain point, the reduction in SSE becomes marginal, and that point is often referred to as the "elbow" in the SSE plot. The "elbow method" is a heuristic for selecting the optimal number of clusters based on this plot.

>It quantifies the amount of variance or "error" within the clusters.

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} ||x_i^j - c_j \text{🌟}||^2$$

$$n : total\ number\ of\ data\ points$$
$$k : total\ number\ of\ Clusters$$
$$x\ is\ the\ data\ point\ in\ the\ cluster$$
$$c : is\ the\ \text{🌟}\ Centorid\ of\ the\ Cluster$$



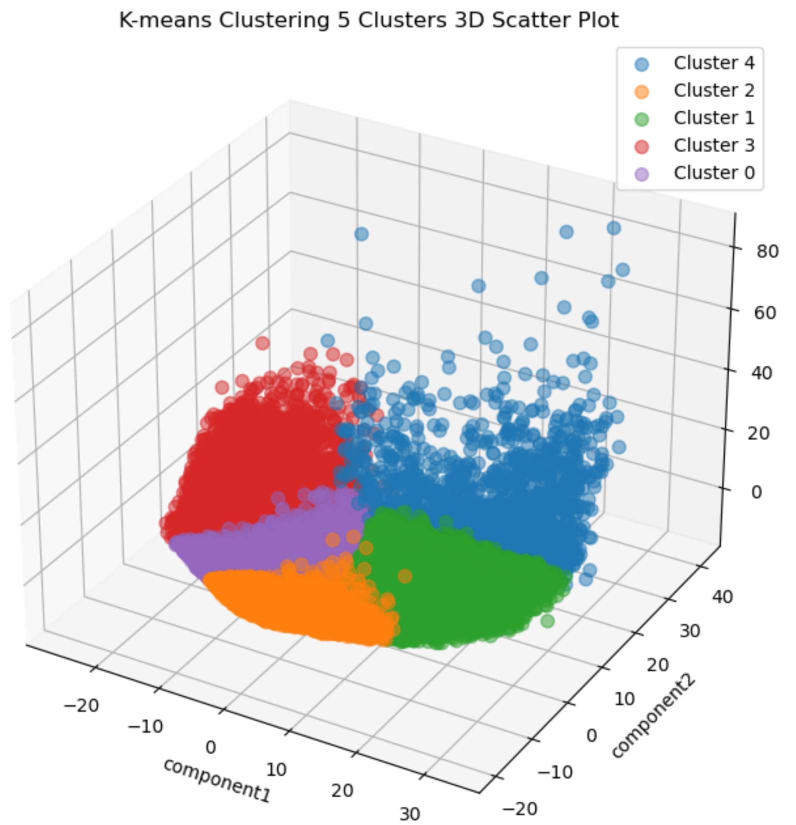
K-means Clustering Visualization with the help of PCA 3 components

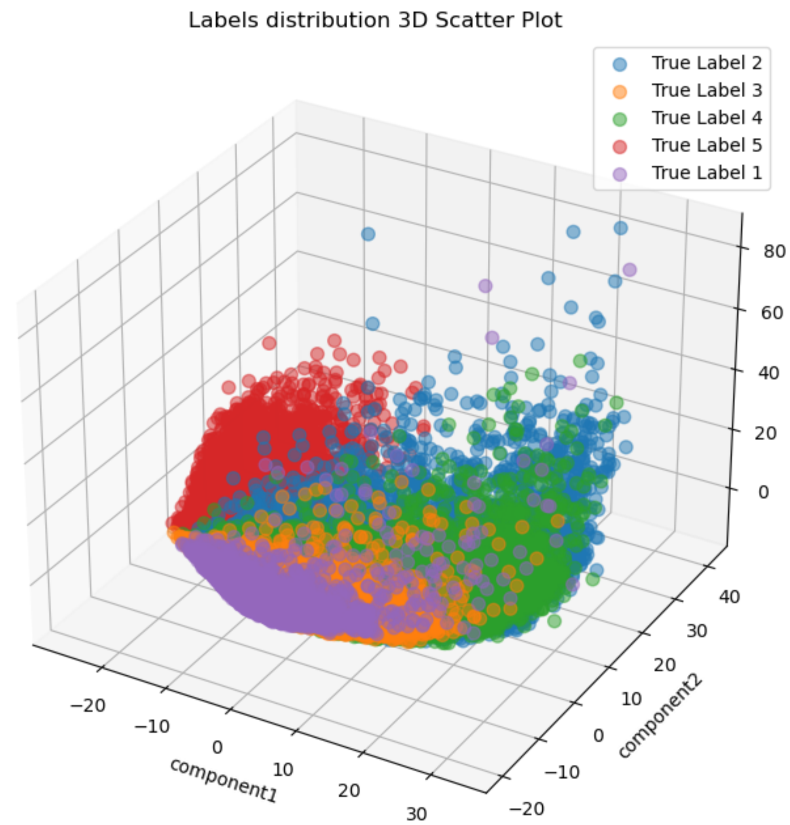
- Transforming all the features (784) into 3 components using PCA

	component1	component2	component3	label
0	11.414313	16.497276	5.625891	2
1	8.545677	-12.316917	1.197881	3
2	9.003173	9.646144	-6.977575	4
3	16.309910	-6.673696	-3.453813	4
4	-23.167697	4.759955	-6.006893	5
...	...	...	...	...
29995	14.417953	-5.001276	-2.819267	4
29996	-8.035726	17.585727	11.474828	5
29997	-24.200379	15.050351	13.468413	5
29998	20.460393	10.269069	-2.129959	2
29999	-12.971900	-8.679104	0.957135	1

30000 rows × 4 columns

- Using the 3 components as axes we can plot the K-means in 3D
- in the notebook you can find an interactive `plotly` graph





Model deployment

a simple GUI that use the saved model in classifications of fashion-test cloth