Machine Learning

Homework 1

Yusupha Juwara
juwara.1936515@studenti.uniroma1.it

December 3, 2023

Contents

Chapter 1

Report about Dataset One (1)

1.1 Plots and visualizations

The first rule that I always follow is to always inspect my data before anything else!

So, let us first visualize some samples of the training and testing data.

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
0	0.994210	0.339437	0.0	0.481943	4.104690	0.0	0.0	1.003648	0.0	1.353046	...	2.507152	0.000000	3.588297	0.000000	0.000000	0.156997	3.890810	0.000000	0.000000	0.0
1	0.000000	1.415991	0.0	2.885742	0.000000	0.0	0.0	0.000000	0.0	0.000000	...	0.211144	0.000000	3.315986	0.965814	3.044368	3.456200	0.000000	9.485678	0.028253	0.0
2	0.000000	1.320484	0.0	2.814479	0.832429	0.0	0.0	0.000000	0.0	0.000000	...	0.000000	0.000000	3.118805	0.399551	2.272334	2.339240	0.000000	7.692220	0.000000	0.0
3	0.459657	1.202142	0.0	0.000000	2.281308	0.0	0.0	1.488012	0.0	0.808162	...	3.554601	0.996622	2.015630	0.000000	2.926057	0.000000	1.943083	3.075836	0.000000	0.0
4	0.023708	0.000000	0.0	1.006909	0.000000	0.0	0.0	1.312168	0.0	1.016113	...	1.661898	0.000000	10.896092	1.081312	6.260812	4.976063	6.542039	1.703158	0.000000	0.0

Figure 1.1: 5 samples from the entire training dataset

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
0	0.433888	0.584314	0.0	2.060552	2.945309	0.0	0.369958	2.949629	0.0	1.648048	...	3.010407	0.406688	1.360133	0.000000	2.449337	0.000000	0.764747	0.000000	0.0	0.0
1	0.000000	3.600694	0.0	0.858300	0.967494	0.0	0.000000	3.073008	0.0	0.000000	...	0.311528	0.000000	8.238040	1.033474	2.611662	1.917893	5.071940	4.870718	0.0	0.0
2	0.000000	3.472290	0.0	1.457396	1.032517	0.0	0.000000	3.134619	0.0	0.000000	...	0.054966	0.000000	6.244536	0.803233	2.717448	1.502867	4.175643	4.662583	0.0	0.0
3	0.000000	1.591720	0.0	3.381381	3.537153	0.0	0.000000	2.946058	0.0	0.000000	...	0.000000	0.000000	1.126994	0.000000	3.449582	0.301905	2.213734	3.914564	0.0	0.0
4	1.217804	0.037316	0.0	0.770278	4.479984	0.0	0.000000	0.734885	0.0	1.228074	...	2.362442	0.000000	4.044020	0.113741	0.122593	0.169172	5.096256	0.000000	0.0	0.0

Figure 1.2: 5 samples from the entire testing dataset

1.1.1 Check non numeric values

```
# Count the number of NaN values in each column
X_nan_counts = X_df.isna().sum(axis=0) # shape (ncols,)
X_blind_nan_counts = X_df.isna().sum(axis=0)

# Count the number of null values in each column
X_null_counts = X_df.isnull().sum(axis=0)
X_blind_null_counts = X_df.isnull().sum(axis=0)

print(f"X_nan_counts: {X_nan_counts.sum()} \tX_blind_nan_counts: {X_blind_nan_counts.sum()}")
print(f"X_null_counts: {X_null_counts.sum()} \tX_blind_null_counts: {X_blind_null_counts.sum()}")
```

Results:

```
X_nan_counts: 0 X_blind_nan_counts: 0
X_null_counts: 0 X_blind_null_counts: 0
```

After the **non numeric values check**, let us see the description of our datasets.

```
X_df.describe()
```

```
X_blind_df.describe()
```

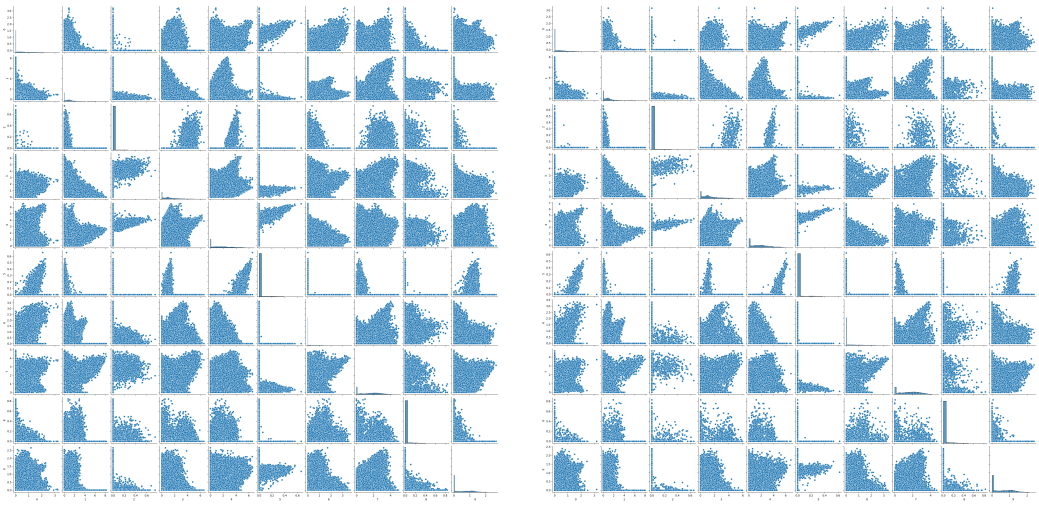
	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	...	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000
mean	0.378181	1.317949	0.009003	1.391512	1.814728	0.011052	0.369846	1.626467	0.014639	0.545710	...	2.272932	0.296990	3.042890	0.339678	2.625464	0.842655	1.926343	2.506435	0.035964	0.009226
std	0.516047	1.496512	0.050664	1.230377	1.508633	0.050906	0.630221	1.112125	0.062849	0.557098	...	2.104927	0.495357	2.883971	0.446056	1.673630	1.331757	2.016545	2.694744	0.134504	0.057310
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.248946	0.000000	0.409221	0.463299	0.000000	0.000000	0.603037	0.000000	0.000000	...	0.483817	0.000000	0.683260	0.000000	1.323167	0.000000	0.020918	0.108381	0.000000	0.000000
50%	0.075048	0.910430	0.000000	1.076016	1.624676	0.000000	0.000000	1.722971	0.000000	0.410361	...	1.973584	0.000000	2.138780	0.023752	2.541312	0.036439	1.263868	1.690138	0.000000	0.000000
75%	0.608842	1.645869	0.000000	2.084849	2.857240	0.000000	0.562436	2.507033	0.000000	1.017619	...	3.381041	0.460844	4.736313	0.722400	3.915264	0.973730	3.399809	3.824580	0.000000	0.000000
max	3.206600	8.212224	0.748968	6.523547	6.554948	0.658043	3.620714	4.950512	0.839790	2.649603	...	12.077601	2.320764	13.996863	2.380158	8.275156	6.449932	8.829996	13.680472	1.375294	1.057508

Figure 1.3: Training dataset (X_df). A lot of zero values!

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
count	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	...	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000
mean	0.371903	1.290253	0.009115	1.396846	1.830216	0.010462	0.354332	1.615608	0.014489	0.546114	...	2.229352	0.292754	3.048722	0.330959	2.592000	0.844586	1.941248	2.493135	0.033515	0.008127
std	0.509466	1.480494	0.051749	1.206766	1.491688	0.050339	0.611197	1.095775	0.063313	0.549835	...	2.062532	0.486647	2.846174	0.443211	1.651005	1.323865	1.985435	2.673338	0.132592	0.055139
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.229160	0.000000	0.443358	0.488734	0.000000	0.000000	0.638269	0.000000	0.000000	...	0.473890	0.000000	0.745337	0.000000	1.309943	0.000000	0.070376	0.148131	0.000000	0.000000
50%	0.057141	0.882379	0.000000	1.105087	1.675764	0.000000	0.000000	1.675463	0.000000	0.421687	...	1.940213	0.000000	2.192284	0.004386	2.498253	0.054994	1.339522	1.689639	0.000000	0.000000
75%	0.619030	1.616386	0.000000	2.058674	2.835237	0.000000	0.533029	2.474927	0.000000	1.012854	...	3.310865	0.453145	4.681890	0.702819	3.832808	1.002640	3.310388	3.793412	0.000000	0.000000
max	3.160051	7.984558	0.658491	5.927709	6.844651	0.622780	3.346777	4.707161	0.829520	2.390183	...	11.403351	2.244202	13.847556	1.923470	7.687513	6.308266	8.464344	12.968296	1.229173	0.812376

Figure 1.4: Testing dataset (X_blind_df). A lot of zero values as well

Now, let us do some ploting with **Seaborn Pairplot**, which allows us to plot pairwise relationships between variables within a dataset. This creates a nice visualisation and helps us understand the data by summarising a large amount of data in a single figure, as the saying goes, a picture is worth a thousand words. This is essential when we are exploring our dataset and trying to become familiar with it, hence **Exploratory Data Analysis (EDA))**.



(a) Training dataset with 10 features for visualization purposes

(b) Testing dataset with 10 features for visualization purposes

Figure 1.5: Visualizing how the various features of samples correlate.

It can be easily seen that the two datasets were drawn from the same distribution since they look almost exactly the same. Furthermore, it can be seen that the correlation coefficients between most of the features of both datasets are very small (mostly scattered). But I will support the latter claim with a **Correlation Matrix** that assign numerical values to how much the features correlate with one another.

But first, let's normalize the datasets.

1.1.2 Data Normalization

It's recommended to normalize the data since almost always the features in any dataset would have different units of measurements such as age vs income.

Several types of normalizations calculated based on the dataset D are:

- **min-max**:

$$\bar{X} = \frac{X - \textbf{min}}{\textbf{max - min}}$$

- **normalization (standardization)**:

$$\bar{X} = \frac{x - \mu}{\sigma}$$

  This is the most used one.

- **unit vector**:

$$\bar{X} = \frac{x}{||x||}$$

In this homework, however, I use the **standardization** as thus.

```
def standardise(X):
    # mean and standard deviation of all the features irrespective of class differences
    mean_all    = X.mean()
    std_all     = X.std()

    # standardised values
    X_standardised = (X-mean_all)/std_all
    return X_standardised

# X_standardised = standardise(X_original)
# X_standardised.head(10)
```

same as this below using sk-learn that also handles the case when the standard deviation is zero and also numerical stability:

```
scaler = StandardScaler()
scaler.fit(X_df)
X_standardised = scaler.transform(X_df)
X_blind_standardised = scaler.transform(X_blind_df)
```

After standaridizing the datasets, let us split the training one into train-test-split in the ratio 75:25 and view some statistics and plots.

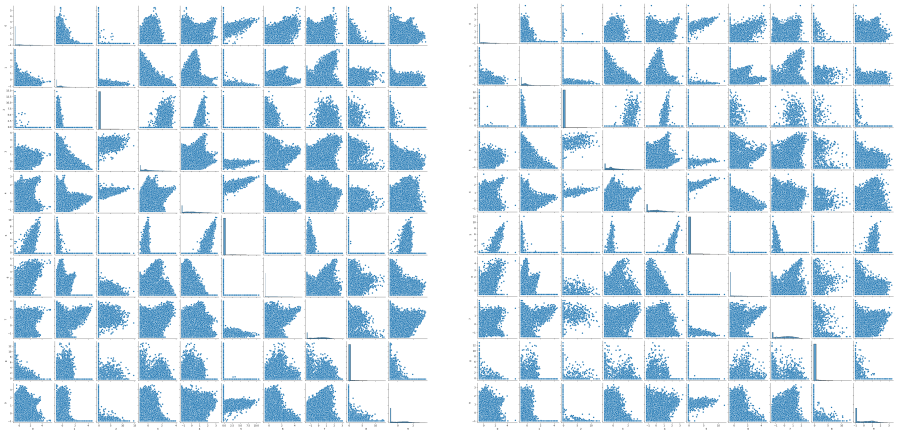
	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
0	2.069044	0.222402	-0.177707	-0.606206	-1.202908	-0.217107	2.311027	1.014067	-0.232932	2.317361	...	1.214978	-0.289548	-1.055115	-0.761522	0.674856	-0.629350	-0.955278	-0.930129	-0.267381	-0.160982
1	0.002352	0.260570	-0.177707	-0.254026	-1.202908	-0.217107	0.845193	0.896561	-0.232932	-0.317107	...	0.535356	0.706132	-1.055115	-0.761522	0.120861	-0.632746	-0.955278	-0.576329	-0.267381	-0.160982
2	-0.732850	-0.143250	-0.177707	0.105016	0.897620	-0.217107	-0.586856	0.896735	-0.232932	1.650732	...	0.501491	-0.599552	-0.220853	-0.761522	-0.221355	-0.632746	-0.381699	-0.408828	-0.267381	-0.160982
3	-0.732850	0.174422	-0.177707	1.252781	-1.202908	-0.217107	-0.586856	-1.462501	-0.232932	-0.979568	...	-0.946782	-0.599552	0.040373	0.996978	0.112295	1.705102	-0.955278	2.776677	1.260830	-0.160982
4	-0.732850	-0.366337	-0.177707	1.175636	-1.202908	-0.217107	-0.586856	-1.462501	-0.232932	-0.621715	...	-1.079826	-0.599552	0.615196	1.715981	0.337519	1.971741	-0.955278	2.579280	-0.267381	-0.160982

Figure 1.6: 5 samples from the entire training dataset

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
0	0.107951	-0.490235	-0.177707	0.543774	0.749415	-0.217107	0.000178	1.189772	-0.232932	1.978734	...	0.350360	0.221455	-0.583492	-0.761522	-0.105237	-0.632746	-0.576038	-0.930129	-0.267381	-0.160982
1	-0.732850	1.525392	-0.177707	-0.433378	-0.561596	-0.217107	-0.586856	1.300713	-0.232932	-0.979568	...	-0.931825	-0.599552	1.801406	1.555415	-0.008247	0.807391	1.559910	0.877377	-0.267381	-0.160982
2	-0.732850	1.439589	-0.177707	0.053548	-0.518495	-0.217107	-0.586856	1.356113	-0.232932	-0.979568	...	-1.053713	-0.599552	1.110163	1.039239	0.054962	0.495750	1.115434	0.800139	-0.267381	-0.160982
3	-0.732850	0.182941	-0.177707	1.617301	1.141724	-0.217107	-0.586856	1.186561	-0.232932	-0.979568	...	-1.079826	-0.599552	-0.664333	-0.761522	0.492419	-0.406047	0.142518	0.522552	-0.267381	-0.160982
4	1.627045	-0.855754	-0.177707	-0.504919	1.766687	-0.217107	-0.586856	-0.801701	-0.232932	1.224866	...	0.042524	-0.599552	0.347139	-0.506527	-1.495489	-0.505715	1.571968	-0.930129	-0.267381	-0.160982

Figure 1.7: 5 samples from the entire testing dataset

Let's plot again the **Pair Plots** as shown in Figure 1.8.



(a) Training dataset with 10 features for visualization purposes

(b) Testing dataset with 10 features for visualization purposes

Figure 1.8: Visualizing how the various features of samples correlate.

Have you noticed any difference in the figure on the left? It is slightly different from the previous one due to splitting into train-test-split. The one on the right is still the same, i.e., standardization scales all the data and, so, preserves the previous correlation properties.

Let us see a real-valued correlations between each feature, this way we know how much they correlate with one another.

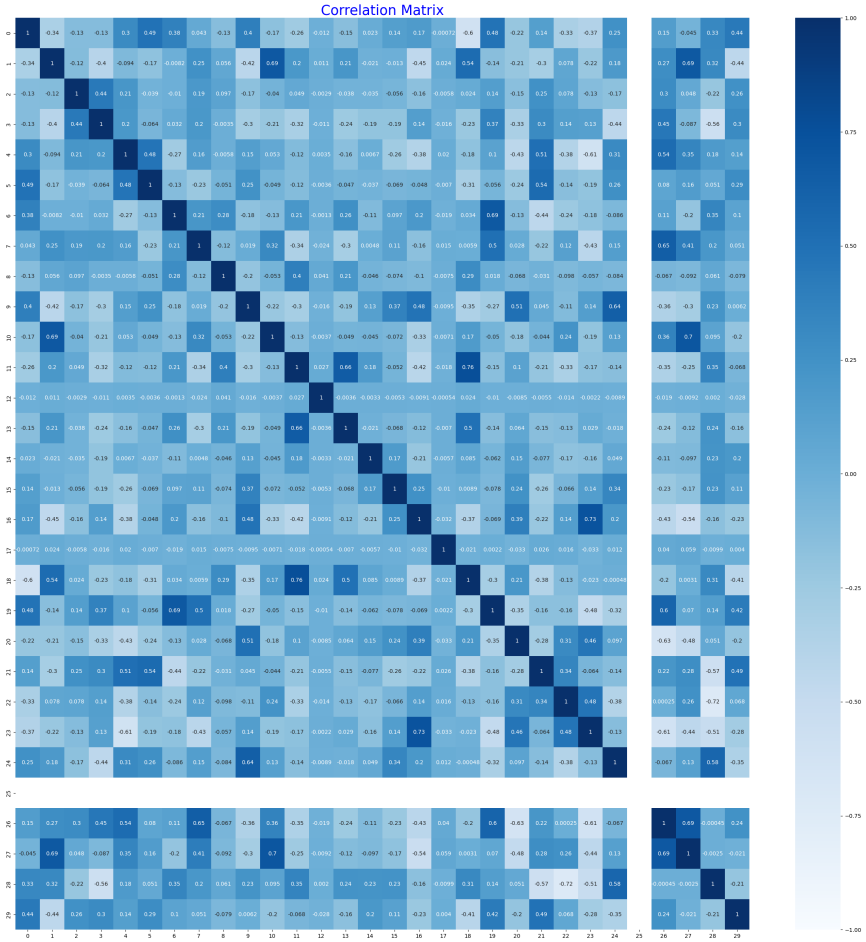


Figure 1.9: Correlation Matrix of the training dataset that was split. This plot shows only 30 rows and columns.

Although not all the matrix is plotted to not clutter the visualization, the bottom left triangle portion of the matrix is actually the same as the top right portion of the matrix, with the axes flipped. The diagonal row is simply a (univariate) histogram of each variable and number of occurrences.

But, have you noticed 25 in the matrix (the white cross)? Very interesting! Let's first plot a pair-plot so that we can visualize it better.

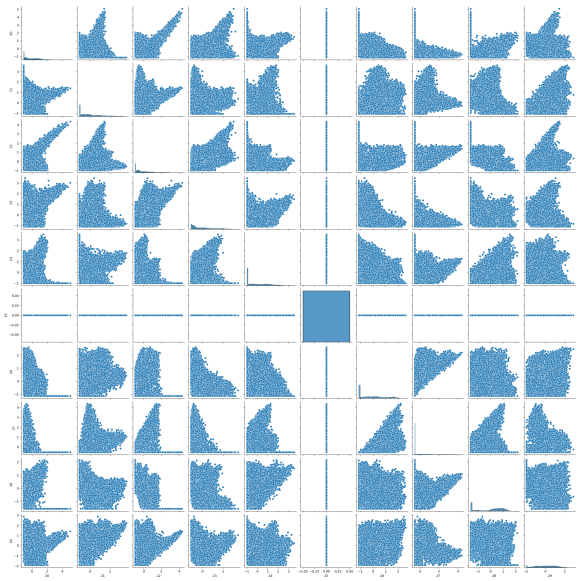


Figure 1.10: Feature 25 with all zero values

What have we observed there? We can clearly see that all its values are zeros, which means that it has a mean and variance of zero. This means that during the correlation calculations, we cannot divide by zero, and so it has NaN values for the

Correlation matrix. That is why it has no values in the correlation matrix plot.

Does this pose a problem for the training? Not really! The two algorithms that I will be using in this homework will handle it. That is, the **Gaussian Naive Bayes** has a hyperparameter called **var_smoothing** that handles these cases when the variance is zero or very small. Also, SVM has a way to stabilize it. More on those in the later sections.

To read more about a column or a row having all zero values, see this post on stackoverflow.

1.2 Grid Search and Cross-Validation Process

Before delving into the models and their training, it is especially useful to first elaborate on **Cross-validation** and **Grid Search** of sk-learn for hyperparameter tuning.

The grid search approach systematically explores different combinations of hyperparameter values to identify the set that maximizes a model's performance. Cross-validation is crucial to obtain robust performance estimates and avoid overfitting to a specific subset of the data.

The process involves training the models (like SVM, K-means) with different combinations of hyperparameters on subsets of the training data (folds) and evaluating their performance. The combination that yields the best performance, often measured by metrics like accuracy or F1-score, is then selected as the optimal set of hyperparameters. The best performing model is then retrained on the entire training dataset.

This iterative process ensures that the model's performance is generalized and not overly tailored to the peculiarities of a particular training subset. The final model, trained with the identified optimal hyperparameters, is expected to perform well on unseen data.

1.3 Overview of Support Vector Machines (SVM)

Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms used for classification and regression tasks. SVMs are particularly well-suited for scenarios where clear decision boundaries are desired. The fundamental idea behind SVM is to find a hyperplane that best separates different classes in the feature space.

1.3.1 Mathematical Formulation

Consider a binary classification problem with a dataset $D = \{(x_n, t_n)_{n=1}^N\}$, where x_n represents the input data, and t_n is the corresponding class label ($t_n \in \{+1, -1\}$). SVM aims to find a hyperplane that maximizes the margin between different classes in the feature space.

- Assume D is linearly separable

$$\exists w, w_0 s.t \begin{cases} y(x_n) > 0, & \textbf{if } t_n = +1 \\ y(x_n) < 0, & \textbf{if } t_n = -1 \end{cases}$$
$$t_n y(x_n) > 0 \quad \forall n = 1, \cdots N$$

- Let x_k be the closest point of the dataset D to the hyperplane $\bar{h} : \bar{w}^T x + \bar{w}_0 = 0$
- the margin (smallest distance between x_k and $\bar{h}$) is $\frac{|y(x_k)|}{||w||}$
- Given dataset D and hyperplane $\bar{h}$ and using the property $|y(x_n)| = t_n y(x_n)$, the **margin** is computed as

$$\min_{n=1,\cdots,N} \frac{|y(x_k)|}{||w||} = \cdots = \frac{1}{||w||} \min_{n=1,\cdots,N} \left[t_n(\bar{w}^T x_n + \bar{w}_0)\right]$$

- Likewise, given the dataset D, the hyperplane $h^* : w^{*T} x + w_0^*$ with the **maximum margin** is computed as

$$w^*, \quad w_0^* = \textbf{argmax}_{w,w_0} \frac{1}{||w||} \min_{n=1,\cdots,N} \left[t_n(\bar{w}^T x_n + \bar{w}_0)\right]$$

- Rescale the data points in such a way that for the closest point x_k we have

$$t_k(w^T x_k + w_0) = 1.$$

  Note that rescaling all the points does not affect the solution.

- Canonical representation:

$$t_n(w^T x_n + w_0) \geq 1 \quad n = 1, \cdots, N$$

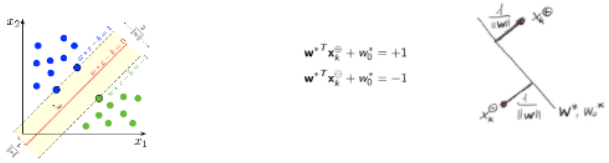


Figure 1.11: SVM example

- When the maximum margin hyperplane w^*, w_0^* is found, there will be at least 2 closest points x_k^+ and x_k^- (one for each class)
- In the canonical representation of the problem the maximum margin hyperplane can be found by solving the optimization problem

$$w^*, w_0^* = \textbf{argmax}\frac{1}{||w||} = \textbf{argmin}\frac{1}{2}||w||^2$$

subject to

$$t_n(w^T x_n + w_0) \geq 1 \quad n = 1, \cdots, N$$

- This is a **Quadratic programming** problem solved with **Lagrangian** method
- The solution to this problem using **Lagrangian multipliers** method is:

$$w^* = \sum_{n=1}^{N} a_n^* t_n x_n$$

a_i^* (Lagrange multipliers) are the results of the Lagrangian optimization problem

$$\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m t_n t_m x_n^T x_m$$

subject to

$$a_n \geq 0 \quad \forall n = 1, \cdots, N$$
$$\sum_{n=1}^{N} a_n t_n = 0$$

- This formulation exploits a really important property called the **Karush-Kuhn-Tucker (KKT)** condtion, which states that: for each $x_n \in D$ either $a_n^* = 0$ or $t_n y(x_n) = 1$. This means that only the support vectors where $t_n y(x_n) = 1$ have their Lagrangian multipliers $a_n^* = 1$, while all the other non support vectors have $a_k^* = 0$ and do not contribute to the solution (thus $t_n y(x_n) > 1 \implies a_n^* = 0$)
- Therefore, the **Support vectors** x_k such that $t_k y(x_n) = 1$ and $a_k^* > 0$ are the set:

$$\textbf{SV} = \{x_k \in D | t_k y(x_k) = 1\}$$

- Consequently, the hyperplane is expressed only in terms of the support vectors

$$y(x) = \sum_{x_j \in \textbf{SV}} a_j^* t_j x^T x_j + w_0^* = 0.$$

Note: all the other vectors $x_n \notin \textbf{SV}$ do not contribute ($a_n^* = 0$)

- In order to compute the w_0^*, we pick any support support in SV (those that satisfy $t_k y(x_k) = 1$) and solve the equation

$$t_k \left(\sum_{x_j \in \textbf{SV}} a_j^* t_j x_k^T x_j + w_0^* \right) = 1$$

- Multiplying by t_k and using $t_k^2 = 1$

$$w_0^* = t_k - \sum_{x_j \in \textbf{SV}} a_j^* t_k x_k^T x_j$$

- Actually, a better solution that is more robust to noisy data or a more stable solution is obtained by averaging over all the support vectors

$$w_0^* = \frac{1}{|\mathbf{SV}|} \sum_{x_k \in \mathbf{SV}} \left(t_k - \sum_{x_j \in \mathbf{SV}} a_j^* t_k x_k^T x_j \right)$$

- Once we find the maximum margin hyperplane determined by a_k^*, w_0^*, classification of a new instance x' is

$$y(x') = \mathbf{sign} \left(\sum_{x_k \in \mathbf{SV}} a_k^* t_k x'^T x_k + w_0^* \right)$$

1.3.1.1 Soft Margin SVM

In scenarios where the data is not perfectly separable, **soft margin SVM** introduces slack variables ξ_n to allow for some misclassification. The optimization problem becomes:

$$\text{minimize } \frac{1}{2}||w||^2 + C \sum_{n=1}^{N} \xi_n$$

subject to $t_n(w^T x_n + w_0) \geq 1 - \xi_n$ and $\xi_n \geq 0$, where C is a regularization parameter.

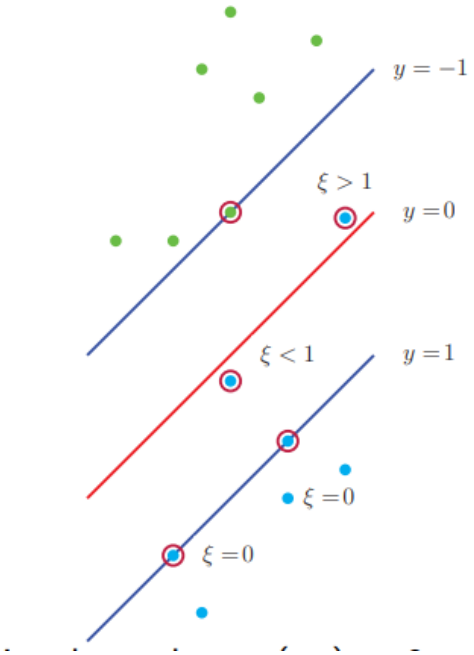


Figure 1.12: SVM slack variable example

- What if data are "almost" linearly separable (e.g., a few points are on the "wrong side")
- In this case, we introduce slack variables $\xi \geq 0, \quad n = 1, \cdots, N$
- $\xi_n = 0$ if point on or inside the correct margin boundary
- $0 < \xi_n \leq 1$ if point inside the margin but correct side
- $\xi_n > 1$ if point on the wrong side of boundary
- When $\xi_n = 1$, the sample lies on the the decision boundary $y(x_n) = 0$
- When $\xi_n > 1$, the sample will be misclassified

1.3.2 Kernel Methods

Kernel methods extend SVMs to handle non-linear decision boundaries by mapping the input data to a higher-dimensional feature space by using **Basis (Kernel) Functions**.

There are input spaces of unknown and fixed dimensions, such as $X \in \mathbb{R}^M$, but in real situations this is not always the case. We often have to deal with input spaces of variable length and possibly infinite dimensions, such as strings, images, trees, time-series, etc.

This is when **kernel functions** come into play. Fundamentally, they are functions that, given two inputs, return a real value that represents their **similarity**. That is, **Kernel methods** extend SVMs to handle non-linear decision boundaries by mapping the input data to a higher-dimensional feature space by using **Basis (Kernel) Functions**.

A kernel is a real-valued function $k(x, x') \in \mathbb{R}$ where $x, x' \in \mathbb{X}$ for some abstract space $\mathbb{X}$.
Kernels typically satisfy these conditions
- They are **symmetrical**: $k(x, x') = k(x', x)$
- They are **non-negative**: $k(x, x') \geq 0$

The more the two inputs are similar the closer the output value is to zero.

Input data in the dataset D must be normalized in order for the kernel to be a good **similarity measure** in practice. Several types of normalizations calculated based on the dataset D are:

- **min-max**:
$$\bar{X} = \frac{X - \textbf{min}}{\textbf{max - min}}$$
- **normalization (standardization)**:
$$\bar{X} = \frac{x - \mu}{\sigma}$$
This is the most used one.
- **unit vector**:
$$\bar{X} = \frac{x}{||x||}$$

1.3.2.1 Kernel Trick

The kernel trick allows replacing the inner product $x_n^T x_m$ with a kernel function $k(x_n, x_m)$. This transformation is essential for dealing with non-linear relationships in the data.

1.3.2.2 Common Kernel Functions

Various kernel functions can be employed, including:

- Linear Kernel: $k(x_n, x_m) = x_n^T x_m$
- Polynomial Kernel: $k(x_n, x_m) = (\beta x_n^T x_m + \gamma)^d$
- Radial Basis Function (RBF) Kernel: $k(x_n, x_m) = \exp(-\beta ||x_n - x_m||^2)$
- Sigmoid Kernel: $k(x_n, x_m) = \tanh(\beta x_n^T x_m + \gamma)$

1.3.3 Kernelized SVM - Classification

The kernelized SVM classification model is given by:

$$y(x; \alpha) = \text{sign}\left(w_0 + \sum_{n=1}^{N} \alpha_n k(x_n, x)\right)$$

where α_n are Lagrange multipliers. The solution involves expressing the decision boundary solely in terms of support vectors.

1.3.4 SVM Hyperparameters

Support Vector Machines (SVM) come with several parameters and hyperparameters that play a crucial role in determining their performance on a given dataset. Here, I will discuss the hyperparameters mentioned in code snippets.

```
param_grid = {'C':[10, 100], 'gamma':['scale', 'auto'], 'kernel':['rbf', 'sigmoid']}
grid = GridSearchCV(SVC(), param_grid, refit = True, verbose=2, cv=3)
grid.fit(X_train, Y_train)
```

Figure 1.13: Code snippet for the **RBF** and **Sigmoid Kernels** using **Grid Search** and **Cross Validation** of **3-Folds**

In this code snippet, a grid search is performed with the following hyperparameters:

- **C (Regularization Parameter):** The regularization parameter controls the trade-off between achieving a low training error and a low testing error. The grid search tests values of 10 and 100 for C.
- **Gamma:** Gamma is a parameter for the RBF, polynomial, and sigmoid kernels, influencing the reach of a single training example. The grid search tests 'scale' and 'auto' for gamma. If 'scale' is used, it calculates gamma as $\frac{1}{\text{n\_features} \times X.\text{var}()}$; if 'auto' is used, it calculates gamma as $\frac{1}{\text{n\_features}}$.
- **Kernel:** The choice of the kernel function significantly affects the shape of the decision boundary. In this code snippet, the grid search tests the RBF and sigmoid kernels.

The grid search is conducted with 3-fold cross-validation (`cv=3`), and the results are displayed with verbosity (`verbose=2`).

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time=   4.2s
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time=   4.1s
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time=   6.2s
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=   3.8s
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=   2.6s
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=   2.7s
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time=   3.7s
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time=   6.1s
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time=   4.3s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=   2.6s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=   2.7s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=   3.2s
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time=   6.4s
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time=   4.1s
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time=   4.3s
[CV] END ................C=100, gamma=scale, kernel=sigmoid; total time=   2.4s
[CV] END ................C=100, gamma=scale, kernel=sigmoid; total time=   2.9s
[CV] END ................C=100, gamma=scale, kernel=sigmoid; total time=   3.0s
[CV] END ....................C=100, gamma=auto, kernel=rbf; total time=   4.1s
[CV] END ....................C=100, gamma=auto, kernel=rbf; total time=   4.0s
[CV] END ....................C=100, gamma=auto, kernel=rbf; total time=   6.3s
[CV] END ................C=100, gamma=auto, kernel=sigmoid; total time=   2.8s
[CV] END ................C=100, gamma=auto, kernel=sigmoid; total time=   2.1s
[CV] END ................C=100, gamma=auto, kernel=sigmoid; total time=   2.2s
▸ GridSearchCV
```

Figure 1.14: The time spent on each fit of the combination of the models with their respective hyperparameters

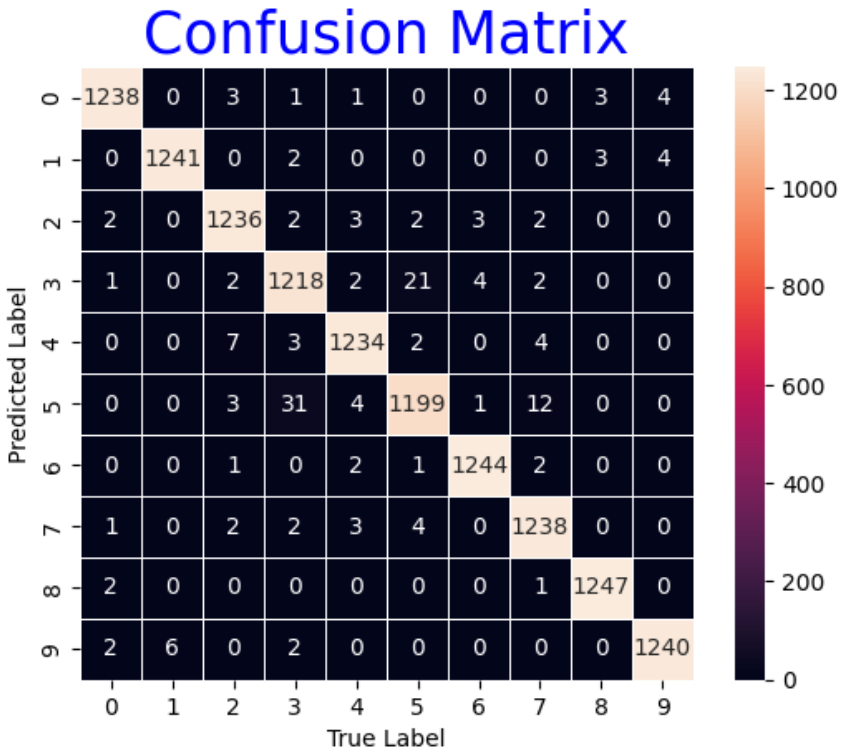


Figure 1.15: Confusion matrix of the best estimator of the above code snippet in Fig 1.14 retrained on the entire training dataset (all the 3-folds)

```
The accuracy score is 98.68%.

classification_report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1250
           1       1.00      0.99      0.99      1250
           2       0.99      0.99      0.99      1250
           3       0.97      0.97      0.97      1250
           4       0.99      0.99      0.99      1250
           5       0.98      0.96      0.97      1250
           6       0.99      1.00      0.99      1250
           7       0.98      0.99      0.99      1250
           8       1.00      1.00      1.00      1250
           9       0.99      0.99      0.99      1250

    accuracy                           0.99     12500
   macro avg       0.99      0.99      0.99     12500
weighted avg       0.99      0.99      0.99     12500


Elapsed Time: 6.517286777496338
```

Figure 1.16: Some performance statistics about the above snippet in Fig 1.14 achieving an accuracy of 98.68%

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_gamma	param_kernel	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score
0	2.389188	0.127660	2.465632	0.818953	10	scale	rbf	{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}	0.98696	0.98808	0.98864	0.987893	0.000698	1
1	1.643037	0.478650	1.390701	0.039499	10	scale	sigmoid	{'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}	0.98088	0.97944	0.98096	0.980427	0.000698	6
2	2.296732	0.352073	2.416161	0.719391	10	auto	rbf	{'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}	0.98704	0.98792	0.98856	0.987840	0.000623	2
3	1.307135	0.042573	1.499591	0.235195	10	auto	sigmoid	{'C': 10, 'gamma': 'auto', 'kernel': 'sigmoid'}	0.98112	0.98032	0.98096	0.980800	0.000346	5
4	2.567083	0.491502	2.343567	0.535950	100	scale	rbf	{'C': 100, 'gamma': 'scale', 'kernel': 'rbf'}	0.98432	0.98616	0.98656	0.985680	0.000975	4
5	1.472019	0.266916	1.319727	0.016653	100	scale	sigmoid	{'C': 100, 'gamma': 'scale', 'kernel': 'sigmoid'}	0.97768	0.97736	0.97752	0.977520	0.000131	8
6	2.269415	0.234467	2.509901	0.818738	100	auto	rbf	{'C': 100, 'gamma': 'auto', 'kernel': 'rbf'}	0.98440	0.98624	0.98648	0.985707	0.000929	3
7	1.264010	0.219705	1.094763	0.106571	100	auto	sigmoid	{'C': 100, 'gamma': 'auto', 'kernel': 'sigmoid'}	0.97768	0.97832	0.97752	0.977840	0.000346	7

Figure 1.17: More statistics about the training of the above code snippet in Fig 1.14. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

As can be seen in the figures, the best (hyper) parameter setting for that particular code snippet in Fig 1.14 are:

```
grid.best_estimator_: SVC(C=10),
grid.best_params_: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'},
grid.best_score_: 0.9868
```

Now, let us consider the default values $C = 1$ and **'gamma' = 'scale'** and inspect how the model performs with respect to the previous result.

```
svc = SVC(kernel="rbf", C=1, gamma="scale", )
```

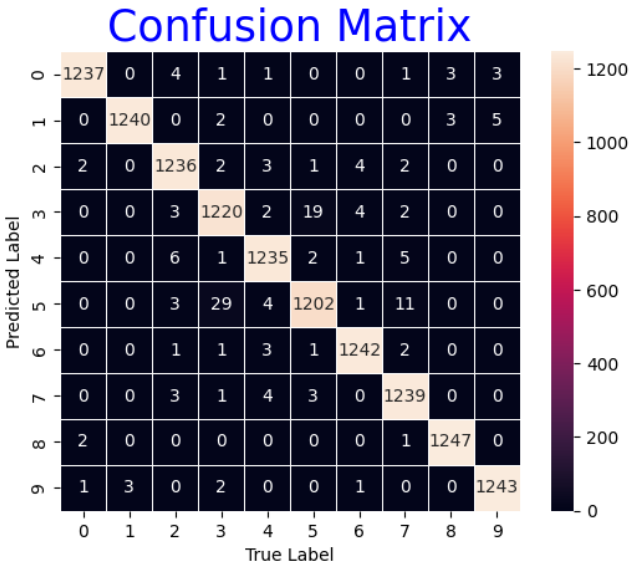


Figure 1.18: Confusion matrix for the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'='scale'**. This achieves an accuracy of 97.2%; slightly better than the previous result

```
The accuracy score is 98.73%.

classification_report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      1250
           1       1.00      0.99      0.99      1250
           2       0.98      0.99      0.99      1250
           3       0.97      0.98      0.97      1250
           4       0.99      0.99      0.99      1250
           5       0.98      0.96      0.97      1250
           6       0.99      0.99      0.99      1250
           7       0.98      0.99      0.99      1250
           8       1.00      1.00      1.00      1250
           9       0.99      0.99      0.99      1250

    accuracy                           0.99     12500
   macro avg       0.99      0.99      0.99     12500
weighted avg       0.99      0.99      0.99     12500


Elapsed time (sec):  12.36385703086853
```

Figure 1.19: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0** achieving an accuracy of 75.12%

> How much time did it take to train and fit this model with the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**?
> Elapsed time: 2778.8797969818115 seconds = 46 minutes 18 seconds.
> This requires so much compute time and, unfortunately, performs very bad!

You can notice that I am avoiding to use floats for gamma! Why is this? Simply, it is very computational heavy on my machine. However let us inspect some variations of gamma with some float values.

```
start_time = time.time()

# Train a single SVC model
svc_rdf_c1 = SVC(kernel="rbf", C=1, gamma=1.0 )
svc_rdf_c1.fit(X_train, Y_train)
Y_pred = svc_rdf_c1.predict(X_test)
confusion_matrix_score(Y_test, Y_pred)

end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)
```

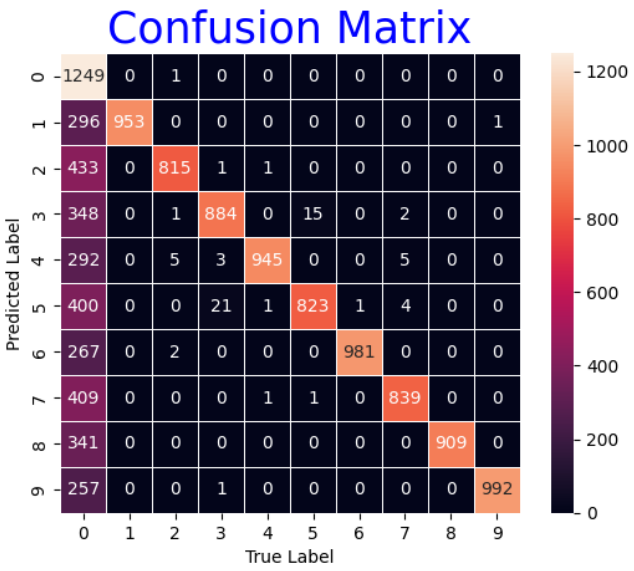


Figure 1.20: Confusion matrix for the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**. This achieves an accuracy of 75.12%; much worse than any of the previous results (oops!). It can be seen that the matrix is very sparse.

```
The accuracy score is 75.12%.

classification_report:
              precision    recall  f1-score   support

           0       0.29      1.00      0.45      1250
           1       1.00      0.76      0.87      1250
           2       0.99      0.65      0.79      1250
           3       0.97      0.71      0.82      1250
           4       1.00      0.76      0.86      1250
           5       0.98      0.66      0.79      1250
           6       1.00      0.78      0.88      1250
           7       0.99      0.67      0.80      1250
           8       1.00      0.73      0.84      1250
           9       1.00      0.79      0.88      1250

    accuracy                           0.75     12500
   macro avg       0.92      0.75      0.80     12500
weighted avg       0.92      0.75      0.80     12500


Elapsed time (sec):  518.0994868278503
```

Figure 1.21: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0** achieving an accuracy of 75.12%

How much time did it take to train and fit this model with the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**?
With this dataset, it took just a negligible time, However, with the dataset2, it took a whopping 46 minutes 18 seconds!

Let us inspect other parameter settings to have even a better understanding!

I will try the **Sigmoid** version of the above model.

```
start_time = time.time()

# Train a single SVC model
svc_sig_c1 = SVC(kernel="sigmoid", C=1, gamma=1.0 )
svc_sig_c1.fit(X_train, Y_train)
Y_pred = svc_sig_c1.predict(X_test)
confusion_matrix_score(Y_test, Y_pred)

end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)
```

Now, enough of the **RBF** and **Sigmoid**. Let us also delve into the **Polynomial kernel** to see how it also compares with the RBF and Sigmoid on this dataset and a few selected hyperparameters!

```
param_grid = {
    'C':[1, 10],
    'degree':[3, 4, 5],
    'kernel':['poly']
    }

# Create the GridSearchCV object with 3-fold cross-validation
grid_search = GridSearchCV(SVC(), param_grid, cv=3, scoring='accuracy', refit=True, n_jobs=-1, verbose=2)
```

In this code snippet, the focus is on the polynomial kernel:

- **C (Regularization Parameter):** Similar to the one explained earlier, the regularization parameter is tested with values of 1 and 10.
- **Degree:** For the polynomial kernel, the degree parameter determines the degree of the polynomial. The grid search tests degrees 3, 4, and 5.
- **Kernel:** The kernel is explicitly set to 'poly' for the polynomial kernel.

Similar to the previous Grid Search, this grid search is also conducted with 3-fold cross-validation and verbosity.

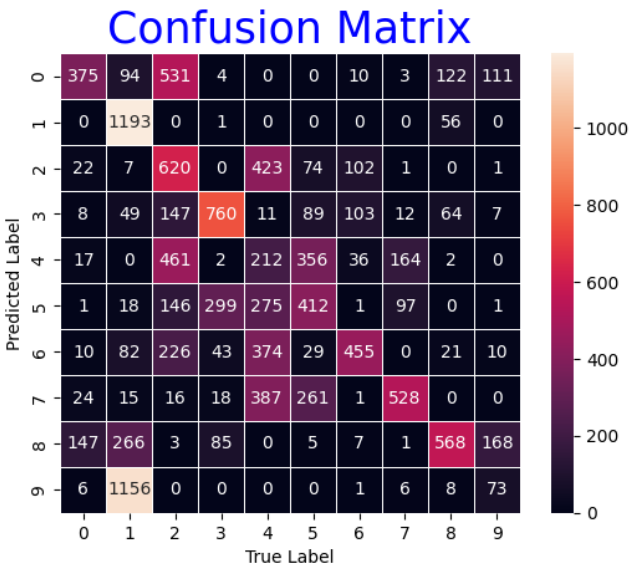


Figure 1.22: Confusion matrix for the parameter setting **'C'=1, 'kernel'='sigmoid', 'gamma'=1.0**. This achieves an accuracy of 41.57%; still bad, but better than its rbf counterpart with this parameter setting

```
The accuracy score is 41.57%.

classification_report:
              precision    recall  f1-score   support

           0       0.61      0.30      0.40      1250
           1       0.41      0.95      0.58      1250
           2       0.29      0.50      0.36      1250
           3       0.63      0.61      0.62      1250
           4       0.13      0.17      0.14      1250
           5       0.34      0.33      0.33      1250
           6       0.64      0.36      0.46      1250
           7       0.65      0.42      0.51      1250
           8       0.68      0.45      0.54      1250
           9       0.20      0.06      0.09      1250

    accuracy                           0.42     12500
   macro avg       0.46      0.42      0.40     12500
weighted avg       0.46      0.42      0.40     12500


Elapsed time (sec):  153.39914679527283
```

Figure 1.23: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='sigmoid', 'gamma'=1.0** achieving an accuracy of 41.57%

As can be seen in the figures, the best (hyper) parameter setting for that particular code snippet in Fig **??** are:

```
best_svm_polymodel: SVC(C=10, kernel='poly'),
grid_search.best_params_: {'C': 10, 'degree': 3, 'kernel': 'poly'},
grid_search.best_score_: 0.9878933333333334
```

1.3.5 Best Performing SVM Model

Among the diverse SVM models with varying hyperparameters, the one that emerges as the best performer is as follows:

The SVM model with default values ($C = 1$, **gamma = 'scale'**, and **kernel = 'rbf'**) achieves exceptional results on the test dataset, demonstrating an outstanding **Accuracy of** 99%.

While the **Best Performing Model** showcases only a slight difference in performance (less than 0.2%), it significantly outshines others (the best performers in both the Grid Searches) in terms of computational efficiency. Notably, this model did not undergo cross-validation, raising the question of potential luck in finding an easier fold to fit. To address this uncertainty, cross-validation is conducted to average the results and provide a more reliable estimate of performance.

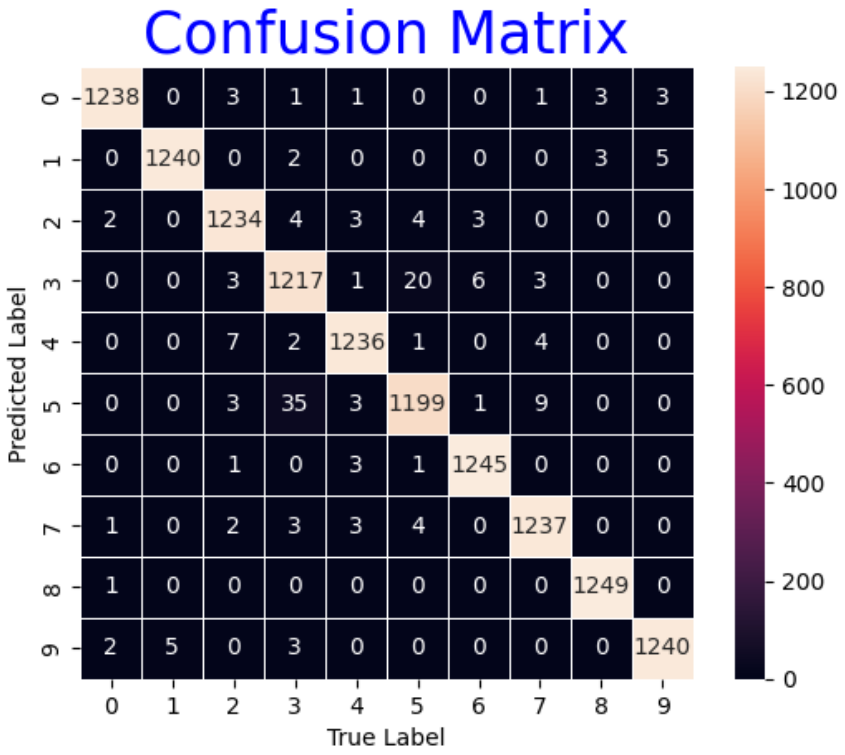


Figure 1.24: Confusion matrix of the best estimator of the above code snippet, retrained on the entire training dataset (all the 3-folds)

```
The accuracy score is 98.68%.

classification_report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      1250
           1       1.00      0.99      0.99      1250
           2       0.98      0.99      0.99      1250
           3       0.96      0.97      0.97      1250
           4       0.99      0.99      0.99      1250
           5       0.98      0.96      0.97      1250
           6       0.99      1.00      0.99      1250
           7       0.99      0.99      0.99      1250
           8       1.00      1.00      1.00      1250
           9       0.99      0.99      0.99      1250

    accuracy                           0.99     12500
   macro avg       0.99      0.99      0.99     12500
weighted avg       0.99      0.99      0.99     12500


Elapsed time:  166.33037114143372
```

Figure 1.25: Some performance statistics about the above snippet, achieving an accuracy of 98.68%

```
start_time = time.time()

clf = SVC(kernel="rbf", C=1, gamma="scale" )
scores = cross_val_score(clf, X_train, Y_train, cv=5)

end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)

```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_degree	param_kernel	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score
0	10.777543	1.576238	3.165872	0.111864	1	3	poly	{'C': 1, 'degree': 3, 'kernel': 'poly'}	0.98320	0.98632	0.98544	0.984987	0.001313	2
1	18.154834	2.213291	5.243068	0.985892	1	4	poly	{'C': 1, 'degree': 4, 'kernel': 'poly'}	0.97096	0.97560	0.97248	0.973013	0.001931	5
2	25.012154	1.003986	8.831890	1.550733	1	5	poly	{'C': 1, 'degree': 5, 'kernel': 'poly'}	0.94736	0.95424	0.95064	0.950747	0.002810	6
3	4.681069	0.836095	2.248015	0.657769	10	3	poly	{'C': 10, 'degree': 3, 'kernel': 'poly'}	0.98648	0.98856	0.98864	0.987893	0.001000	1
4	7.010328	1.595078	2.436760	0.224644	10	4	poly	{'C': 10, 'degree': 4, 'kernel': 'poly'}	0.98328	0.98600	0.98536	0.984880	0.001161	3
5	10.850535	1.223429	3.286395	1.354776	10	5	poly	{'C': 10, 'degree': 5, 'kernel': 'poly'}	0.97304	0.97920	0.97712	0.976453	0.002559	4

Figure 1.26: More statistics about the training of the above code snippet. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

```
10 scores
```

Elapsed time (sec): 35.94395565986633 over 5-folds
array([0.98746667, 0.98826667, 0.98853333, 0.9888 , 0.98893333])

The mean score and the standard deviation are hence given by:

```
1 print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

0.9884 accuracy with a standard deviation of 0.0005 The cross-validated accuracy is **98.84%** with a remarkably low standard deviation of 0.0005. (Impressive!)

This cross-validated result not only reaffirms the model's top position but also provides a deeper understanding of its stability and consistency.

Therefore, we can conclude that this is the best performant model among all the SVM variants tried out in this report.

1.3.6 Conclusion About SVM

Support Vector Machines (SVM) present a robust framework for classification tasks, showcasing the ability to identify optimal decision boundaries with maximum margins. The integration of kernel methods enhances SVMs' capability to adeptly handle non-linear relationships within the data, establishing them as valuable tools in diverse machine learning applications.

In the realm of SVM hyperparameter tuning, the combination of grid search with cross-validation proves to be a formidable technique. This approach systematically explores various hyperparameter combinations, striking a delicate balance between model complexity and generalization. The iterative process of training SVM models on distinct subsets of the training data, coupled with cross-validation evaluation, ensures a reliable estimation of performance. This methodology not only prevents overfitting to specific training subsets but also pinpoints the optimal hyperparameters. The final SVM model, trained with these optimal hyperparameters on the entire training set, stands ready to deliver exceptional performance across diverse datasets.

The cross-validated results solidify the identified SVM model as not only the top performer but also one that demonstrates remarkable stability and consistency. In summary, the synergy between SVMs and grid search with cross-validation offers a comprehensive and effective strategy for constructing high-performance models with the versatility to generalize well across a spectrum of real-world scenarios.

Let us also explore another algorithm different from SVM!

1.4 Overview of Naive Bayes

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. There is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

Abstractly, naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector $\mathbf{x} = (x_1, \ldots, x_n)$ representing some n features (independent variables), it assigns to this instance probabilities

$$p(C_k \mid x_1, \ldots, x_n)$$

for each of K possible outcomes or classes C_k.

Using Bayes' theorem, the conditional probability can be decomposed as

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\ p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

In plain English, using Bayesian probability terminology, the above equation can be written as

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$$

In practice, there is interest only in the numerator of that fraction, because the denominator does not depend on C and the values of the features $\mathbf{x_i}$ are given, so that the denominator is effectively constant. So the evidence, $\sum_k p(C_k)\ p(\mathbf{x} \mid C_k)$ is a scaling factor dependent only on $x_1, \ldots, x_n$, that is, a constant if the values of the feature variables are known. So we shall drop it since it does not influence the classification.

Again, in our case, since we have K classes each of which occurring with equal probability of $\frac{1}{K}$, we can see that the **prior**, $\mathbb{P}(\mathbf{C}_k)$, is effectively constant just like the evidence above. So, we shall drop it as well.

Now we have only the **likelihood**, $p(\mathbf{x} \mid C_k)$, to work with. The likelihood can be rewritten as follows, using the chain rule for repeated applications of the definition of conditional probability:

$$p(x_1, x_2, \ldots, x_n \mid C_k) = p(x_1 \mid x_2, \ldots, x_n, C_k)\ p(x_2 \mid x_3, \ldots, x_n, C_k) \cdots p(x_{n-1} \mid x_n, C_k)\ p(x_n \mid C_k)$$

Now the naive conditional independence assumptions come into play: assume that all features in $\mathbf{x}$ are mutually independent, conditional on the category $\mathbf{C_k}$. Under this assumption,

$$p(x_i \mid x_{i+1}, \ldots, x_n, C_k) = p(x_i \mid C_k)\,.$$

Thus, the joint model, after dropping the prior and the evidence, can be expressed as **proportional** to $\prod_{i=1}^{n} p(x_i \mid C_k)$

> Which Distribution should we use for the joint model **proportional** to $\prod_{i=1}^{n} p(x_i \mid C_k)$? Even though there are many distributions to use here, I opt for Gaussian Naive Bayes.

1.4.1 Gaussian Naive Bayes

When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a normal (or Gaussian) distribution. For example, suppose the training data contains a continuous attribute, $\mathbf{x}$. The data is first **segmented** by the class, and then the **mean** and **variance** of $\mathbf{x}$ are computed in each class. Let $\mu_\mathbf{k}$ be the mean of the values in $\mathbf{x}$ associated with class $\mathbf{C}_k$, and let σ_k^2 be the unbiased sample variance of the values in $\mathbf{x}$ associated with class $\mathbf{C}_k$ (that is, the degree of freedom is 1 => n-1). Suppose one has collected some observation value v. Then, the probability density of v given a class $\mathbf{C}_k$, $p(x = v \mid C_k)$, can be computed by plugging v into the equation for a normal distribution parameterized by μ_k and σ_k^2 as thus:

$$\mathbb{P}(\mathbf{X} = x|\mathbb{C}\neg) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

In order to classify samples, one has to determine which posterior is greater between the K classes: $C_1, \ldots, C_k$.

For example, for the classification as C_1 the posterior is given by

$$\text{posterior}(C_1) = \frac{\mathbb{P}(C_1)\mathbb{P}(\text{feature } 1|C_1) \ldots \mathbb{P}(\text{feature } N-1|C_1)\mathbb{P}(\text{feature } N|C_1)}{\text{evidence}}$$

where the

$$\begin{aligned} \textbf{evidence} = {} & \mathbb{P}(C_1)\mathbb{P}(\text{feature } 1|C_1) \ldots \mathbb{P}(\text{feature N}|C_1) \\ & + \mathbb{P}(C_2)\mathbb{P}(\text{feature } 1|C_2) \ldots \mathbb{P}(\text{feature N}|C_2) \\ & \vdots \\ & + \mathbb{P}(C_k)\mathbb{P}(\text{feature } 1|C_k) \ldots \mathbb{P}(\text{feature N}|C_k) \end{aligned}$$

and prior = $\frac{1}{K}$

But since we know that the evidence and the prior are effectively constants, we can drop them as already explained above. Thus:

- posterior(C_1) = $\mathbb{P}(C_1)\mathbb{P}(\text{feature 1}|C_1)\dots\mathbb{P}(\text{feature N}|C_1)$
- posterior(C_2) = $\mathbb{P}(C_2)\mathbb{P}(\text{feature 1}|C_2)\dots\mathbb{P}(\text{feature N}|C_2)$
- $\vdots$
- posterior(C_k) = $\mathbb{P}(C_k)\mathbb{P}(\text{feature 1}|C_k)\dots\mathbb{P}(\text{feature N}|C_k)$

We use the **negative log-likelihood** since the product of a large number of small probabilities can easily underflow the numerical precision of the computer. And this is resolved by computing instead the sum of the negative log probabilities as thus:

$$\begin{aligned}
-log\prod_{i=1}^{N} p(x_i \mid C_k) &= \sum_{i=1}^{N} -log\left(\mathbb{P}(\mathbf{X}=x_i|C_k)\right) \\
&= \sum_{i=1}^{N} -log\left(\frac{1}{\sqrt{2\pi\sigma_{j,k}^2}} e^{-\frac{(x_i-\mu_{j,k})^2}{2\sigma_{j,k}^2}}\right) \\
&= \sum_{i=1}^{N} \frac{1}{2} log\left(2\pi\sigma_{j,k}^2\right) + \frac{(x_i-\mu_{j,k})^2}{2\sigma_{j,k}^2}
\end{aligned}$$

Where k is the class label and j is the index of the feature at column j

1.4.1.1 Getiing the MLE for the Mean and the Variance

MEAN:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mu_{j,k}} &= -\frac{1}{2}\sum_{i=1}^{N} \frac{2}{\sigma_{j,k}^2}(x_i-\mu_{j,k})(-1) = 0 \\
&\implies \sum_{i=1}^{N} \frac{x_i-\mu_{j,k}}{\sigma_{j,k}^2} = 0 \\
&\implies \frac{1}{\sigma_{j,k}^2}\sum_{i=1}^{N} x_i - \frac{1}{\sigma_{j,k}^2}\sum_{i=1}^{N}\mu_{j,k} = 0 \\
&\implies \sum_{i=1}^{N}\mu_{j,k} = \sum_{i=1}^{N} x_i \\
&\implies \mu_{j,k} = \frac{1}{N}\sum_{i=1}^{N} x_i
\end{aligned}$$

VARIANCE:

$$\begin{aligned}
\frac{\partial \ell}{\partial \sigma_{j,k}^2} &= -\sum_{i=1}^{N}\frac{1}{\sigma_{j,k}} + \sum_{i=1}^{N}\frac{(x_i-\mu_{j,k})^2}{\sigma_{j,k}^3} = 0 \\
&\implies \frac{-N}{\sigma_{j,k}} + \frac{1}{\sigma_{j,k}^3}\sum_{i=1}^{N}(x_i-\mu_{j,k})^2 = 0 \\
&\implies \frac{1}{\sigma_{j,k}^3}\sum_{i=1}^{N}(x_i-\mu_{j,k})^2 = \frac{N}{\sigma_{j,k}} \\
&\implies \sigma_{j,k}^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu_{j,k})^2
\end{aligned}$$

Therefore, the best estimate for the mean and variance parameters of a Gaussian are simply the empirical estimates of the mean and variance respectively. These means and variances are for each feature w.r.t each class. For example, feature 1 w.r.t class 1 has a mean and variance different than feature 1 w.r.t class K. So for each feature x, it has a mean and variance for each class K.Therefore, the means and variances must be calculated for each feature.

1.4.1.2 Gaussian NB Training and Testing Details

```
# Define the parameter grid to search
params_gnb = {'var_smoothing': [1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6]}
# params_gnb = {'var_smoothing': np.logspace(0,-9, num=10)}


# Create the GridSearchCV object with 3-fold cross-validation
grid_search_gnb = GridSearchCV(gnb, params_gnb, cv=10, scoring='accuracy', refit=True, n_jobs=-1, verbose=2)
```

Gaussian Naive Bayes, in my case in this homework, has almost no hyperparameters to tune, so it usually generalizes well. However, we can vary the variance smoothing as a hyperparameter:

- **Variance smoothing:** In the Scikit-learn repository, they wrote:

```

# If the ratio of data variance between dimensions is too small, it
# will cause numerical errors. To address this, we artificially
# boost the variance by epsilon, a small fraction of the standard
# deviation of the largest dimension.

self.epsilon_ = self.var_smoothing * np.var(X, axis=0).max()
```

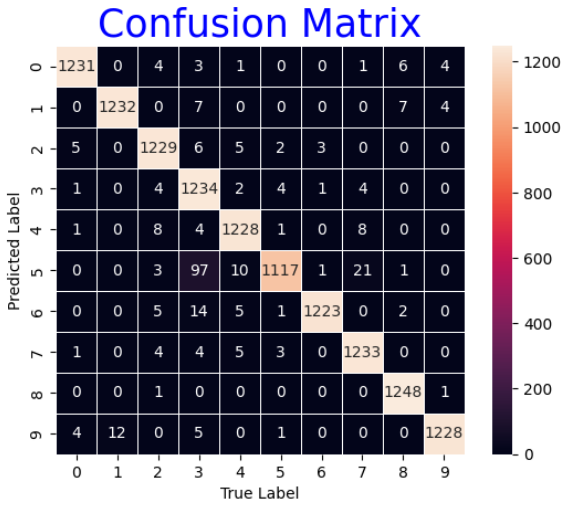
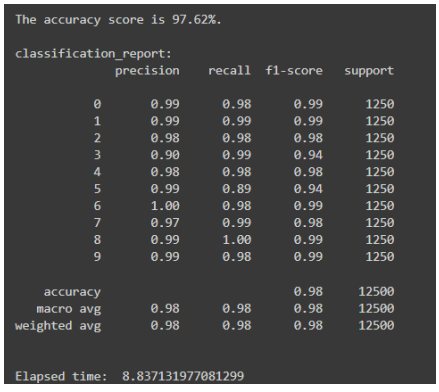


Figure 1.27: Confusion matrix of the best estimator of Gaussian Naive Bayes, retrained on the entire training dataset (all the 10-folds)



The accuracy score is 97.62%.

classification_report:

	precision	recall	f1-score	support
0	0.99	0.98	0.99	1250
1	0.99	0.99	0.99	1250
2	0.98	0.98	0.98	1250
3	0.90	0.99	0.94	1250
4	0.98	0.98	0.98	1250
5	0.99	0.89	0.94	1250
6	1.00	0.98	0.99	1250
7	0.97	0.99	0.98	1250
8	0.99	1.00	0.99	1250
9	0.99	0.98	0.99	1250
accuracy			0.98	12500
macro avg	0.98	0.98	0.98	12500
weighted avg	0.98	0.98	0.98	12500

Elapsed time: 8.837131977081299

Figure 1.28: Some performance statistics about the Gaussian Naive Bayes, achieving an accuracy of 97.62%

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_degree	param_kernel	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score
0	10.777543	1.576238	3.165872	0.111864	1	3	poly	{'C': 1, 'degree': 3, 'kernel': 'poly'}	0.98320	0.98632	0.98544	0.984987	0.001313	2
1	18.154834	2.213291	5.243068	0.985892	1	4	poly	{'C': 1, 'degree': 4, 'kernel': 'poly'}	0.97096	0.97560	0.97248	0.973013	0.001931	5
2	25.012154	1.003986	8.831890	1.550733	1	5	poly	{'C': 1, 'degree': 5, 'kernel': 'poly'}	0.94736	0.95424	0.95064	0.950747	0.002810	6
3	4.681069	0.836095	2.248015	0.657769	10	3	poly	{'C': 10, 'degree': 3, 'kernel': 'poly'}	0.98648	0.98856	0.98864	0.987893	0.001000	1
4	7.010328	1.595078	2.436760	0.224644	10	4	poly	{'C': 10, 'degree': 4, 'kernel': 'poly'}	0.98328	0.98600	0.98536	0.984880	0.001161	3
5	10.850535	1.223429	3.286395	1.354776	10	5	poly	{'C': 10, 'degree': 5, 'kernel': 'poly'}	0.97304	0.97920	0.97712	0.976453	0.002559	4

Figure 1.29: More statistics about the training of Gaussian Naive Bayes. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

As can be seen in the figures, the best (hyper) parameter settings for the Gaussian Naive Bayes are:

```
best_gnb_model: GaussianNB(var_smoothing=1e-06),
grid_search_gnb.best_params_: {'var_smoothing': 1e-06},
grid_search_gnb.best_score_: 0.9776000000000001
```

1.5 Conclusion

In this part about **dataset 1**, we have gone through a lot of different hyperparameter settings and different algorithms. We have explored SVM and three of its basis functions, namely: RBF, Sigmoid, and Polynomial. Among them all, the RBFs were computationally heavier than the others but also gives better results. In fact, the best model found is the RBF with the default parameter settings of sk-learn. It gives an **Accuracy** score of 99%, partly because the dataset was too simple (a lot of zeros).

To have a different taste, we have also explored another algorithm called **Gaussian Naive Bayes** based on **Bayes Theorem** and the assumption that each data point is **i.i.d**. It also gives a very good result of 98%. Furthermore, it normally requires much less compute than the SVM counterpart, very noticeable on the **dataset 2**.

Chapter 2

Report about Dataset Two (2)

All the methods, algorithms and hyperparameter settings, etc that I have used for dataset 1, have also been used for dataset 2. That is, the only difference between the report and notebook of dataset 1 and dataset 2 is only the results and the computational time; everything else is exactly the same. From the data preprocessing, to visualization, to algorithms, to different parameter settings, etc, are exactly the same for both! So, to not repeat the same sections over again, I will be talking about only the results and the computational times in this part of the report (otherwise, I will be writing 50 pages!).

2.1 SVM Hyperparameters for dataset 2

Just like in the dataset 1 part, I have used the same hyperparameters here too for dataset 2. So, I will be illustrating the results and comments that are specific to dataset 2 and haven't been discussed in the dataset 1 part.

```
param_grid = {'C':[10, 100], 'gamma':['scale', 'auto'], 'kernel':['rbf', 'sigmoid']}
grid = GridSearchCV(SVC(), param_grid, refit = True, verbose=2, cv=3)
grid.fit(X_train, Y_train)
```

Figure 2.1: Code snippet for the **RBF** and **Sigmoid Kernels** using **Grid Search** and **Cross Validation** of **3-Folds**

In this code snippet, a grid search and a 3-fold cross-validation is performed with the following hyperparameters as shown above.

Without further ado, let us visualize some astounding results of the dataset 2!

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time= 1.0min
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time= 1.0min
[CV] END .....................C=10, gamma=scale, kernel=rbf; total time= 1.0min
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=  23.9s
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=  23.0s
[CV] END ................C=10, gamma=scale, kernel=sigmoid; total time=  22.1s
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time=  58.7s
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time= 1.0min
[CV] END ......................C=10, gamma=auto, kernel=rbf; total time=  57.3s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=  22.5s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=  23.2s
[CV] END ................C=10, gamma=auto, kernel=sigmoid; total time=  22.3s
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time= 1.2min
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time= 1.2min
[CV] END ....................C=100, gamma=scale, kernel=rbf; total time= 1.1min
[CV] END ...............C=100, gamma=scale, kernel=sigmoid; total time=  18.5s
[CV] END ...............C=100, gamma=scale, kernel=sigmoid; total time=  17.8s
[CV] END ...............C=100, gamma=scale, kernel=sigmoid; total time=  20.1s
[CV] END .....................C=100, gamma=auto, kernel=rbf; total time= 1.2min
[CV] END .....................C=100, gamma=auto, kernel=rbf; total time= 1.1min
[CV] END .....................C=100, gamma=auto, kernel=rbf; total time= 1.2min
[CV] END ...............C=100, gamma=auto, kernel=sigmoid; total time=  18.4s
[CV] END ...............C=100, gamma=auto, kernel=sigmoid; total time=  18.0s
[CV] END ...............C=100, gamma=auto, kernel=sigmoid; total time=  18.2s
▸ GridSearchCV
▸ estimator: SVC
```

Figure 2.2: The time spent on each fit of the combination of the models with their respective hyperparameters

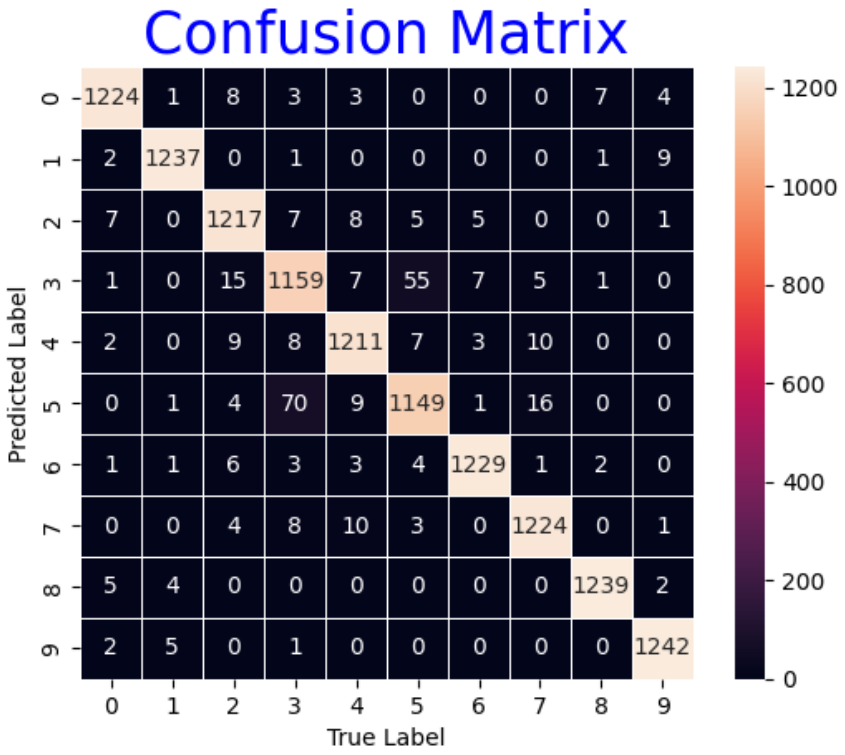


Figure 2.3: Confusion matrix of the best estimator of the above code snippet in Fig 2.2 retrained on the entire training dataset (all the 3-folds)

The accuracy score is 97.05%.

classification_report:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	1250
1	0.99	0.99	0.99	1250
2	0.96	0.97	0.97	1250
3	0.92	0.93	0.92	1250
4	0.97	0.97	0.97	1250
5	0.94	0.92	0.93	1250
6	0.99	0.98	0.99	1250
7	0.97	0.98	0.98	1250
8	0.99	0.99	0.99	1250
9	0.99	0.99	0.99	1250
accuracy			0.97	12500
macro avg	0.97	0.97	0.97	12500
weighted avg	0.97	0.97	0.97	12500

Figure 2.4: Some performance statistics about the above snippet in Fig 2.2 achieving an accuracy of 97.05%

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_gamma	param_kernel	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score
0	25.004735	1.050580	37.009827	1.406520	10	scale	rbf	{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}	0.97136	0.97056	0.97216	0.971360	0.000653	2
1	13.377144	0.454227	9.632312	0.485551	10	scale	sigmoid	{'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}	0.96352	0.96192	0.96096	0.962133	0.001056	6
2	24.154848	0.228670	34.689411	1.460022	10	auto	rbf	{'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}	0.97144	0.97064	0.97224	0.971440	0.000653	1
3	13.188532	0.296111	9.475902	0.453240	10	auto	sigmoid	{'C': 10, 'gamma': 'auto', 'kernel': 'sigmoid'}	0.96328	0.96216	0.96112	0.962187	0.000882	5
4	27.759470	0.536171	42.891144	1.270643	100	scale	rbf	{'C': 100, 'gamma': 'scale', 'kernel': 'rbf'}	0.96464	0.96344	0.96376	0.963947	0.000507	4
5	11.646164	1.005962	7.146739	0.321285	100	scale	sigmoid	{'C': 100, 'gamma': 'scale', 'kernel': 'sigmoid'}	0.95816	0.96080	0.95880	0.959253	0.001124	8
6	27.805652	0.317624	42.117170	1.932757	100	auto	rbf	{'C': 100, 'gamma': 'auto', 'kernel': 'rbf'}	0.96480	0.96344	0.96392	0.964053	0.000563	3
7	10.781667	0.236947	7.434692	0.313958	100	auto	sigmoid	{'C': 100, 'gamma': 'auto', 'kernel': 'sigmoid'}	0.95824	0.96104	0.95904	0.959440	0.001178	7

Figure 2.5: More statistics about the training of the above code snippet in Fig 2.2. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

As can be seen in the figures, the best (hyper) parameter setting for that particular code snippet in Fig 2.2 are:

```
grid.best_estimator_: SVC(C=10, gamma='auto'),
grid.best_params_: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'},
grid.best_score_: 0.97144
```

Now, let us consider the default values $C = 1$ and **'gamma' = 'scale'** and inspect how the model performs with respect to the previous result.

```
svc = SVC(kernel="rbf", C=1, gamma="scale", )
```

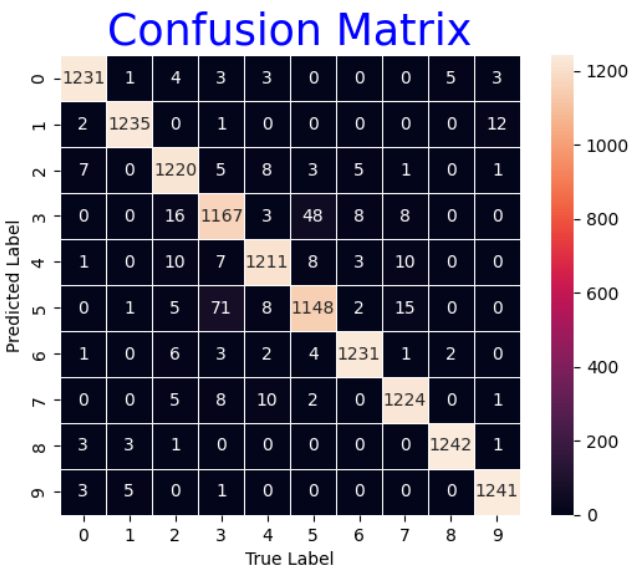


Figure 2.6: Confusion matrix for the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'='scale'**. This achieves an accuracy of 97.20%; slightly better than the previous result and the best seen so far for the dataset 2

```
The accuracy score is 97.20%.

classification_report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.99      1250
           1       0.99      0.99      0.99      1250
           2       0.96      0.98      0.97      1250
           3       0.92      0.93      0.93      1250
           4       0.97      0.97      0.97      1250
           5       0.95      0.92      0.93      1250
           6       0.99      0.98      0.99      1250
           7       0.97      0.98      0.98      1250
           8       0.99      0.99      0.99      1250
           9       0.99      0.99      0.99      1250

    accuracy                           0.97     12500
   macro avg       0.97      0.97      0.97     12500
weighted avg       0.97      0.97      0.97     12500


Elapsed time (sec):  120.62428879737854
```

Figure 2.7: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0** achieving an accuracy of 97.20%

How much time did it take to train and fit this model with the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**?
Elapsed time: 120.62428879737854 seconds = 2 minutes.
This requires so much "less" compute time and and performs the best so far!

You may have noticed that I am avoiding to use floats for gamma! Why is this? Simply, it is very computationally heavy on my machine. However let us inspect some variations of gamma with some float values.

```
start_time = time.time()

# Train a single SVC model
svc_rdf_c1 = SVC(kernel="rbf", C=1, gamma=1.0 )
svc_rdf_c1.fit(X_train, Y_train)
Y_pred = svc_rdf_c1.predict(X_test)
confusion_matrix_score(Y_test, Y_pred)
```

```
end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)
```

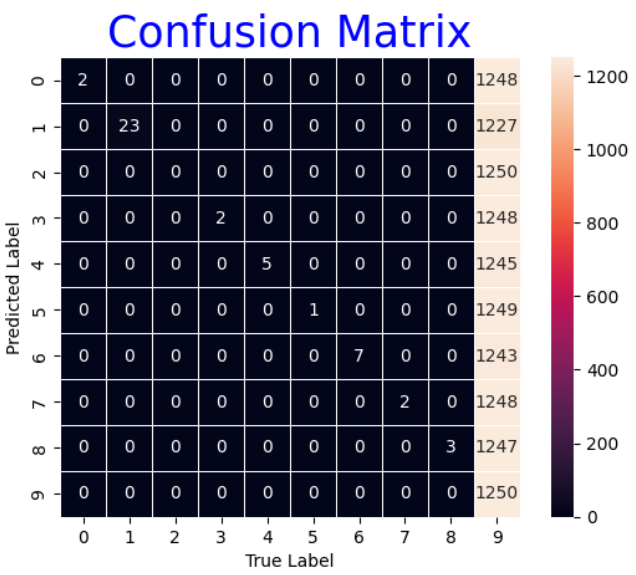


Figure 2.8: Confusion matrix for the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**. This achieves an accuracy of 10.36%; much worse than any of the previous results (oops!). It can be seen that the matrix is very sparse.

```
The accuracy score is 10.36%.

classification_report:
              precision    recall  f1-score   support

           0       1.00      0.00      0.00      1250
           1       1.00      0.02      0.04      1250
           2       0.00      0.00      0.00      1250
           3       1.00      0.00      0.00      1250
           4       1.00      0.00      0.01      1250
           5       1.00      0.00      0.00      1250
           6       1.00      0.01      0.01      1250
           7       1.00      0.00      0.00      1250
           8       1.00      0.00      0.00      1250
           9       0.10      1.00      0.18      1250

    accuracy                           0.10     12500
   macro avg       0.81      0.10      0.03     12500
weighted avg       0.81      0.10      0.03     12500


Elapsed time (sec):  2953.586817264557
```

Figure 2.9: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0** achieving an accuracy of 10.36%

How much time did it take to train and fit this model with the parameter setting **'C'=1, 'kernel'='rbf', 'gamma'=1.0**?
With this dataset, it took a whopping 49 minutes 13 seconds!

Let us inspect other parameter settings to have even a better understanding!

I will try the **Sigmoid** version of the above model.

```
start_time = time.time()

# Train a single SVC model
svc_sig_c1 = SVC(kernel="sigmoid", C=1, gamma=1.0 )
svc_sig_c1.fit(X_train, Y_train)
Y_pred = svc_sig_c1.predict(X_test)
```

```
confusion_matrix_score(Y_test, Y_pred)

end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)
```

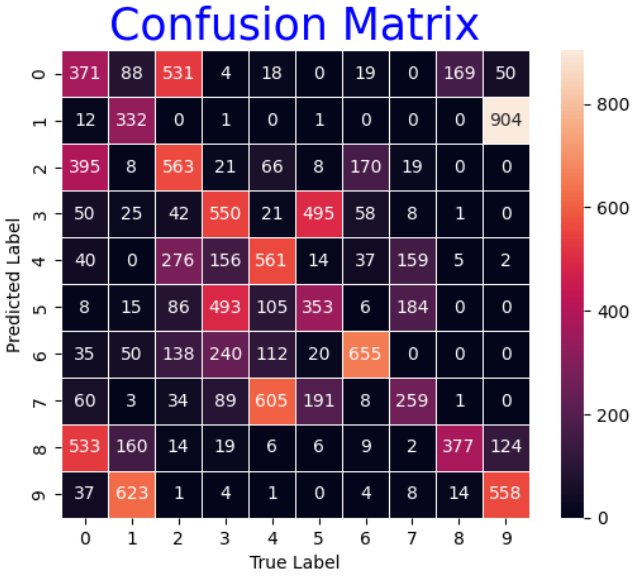


Figure 2.10: Confusion matrix for the parameter setting **'C'=1, 'kernel'='sigmoid', 'gamma'=1.0**. This achieves an accuracy of 36.63%; still bad, but better than its rbf counterpart with this parameter setting

```
classification_report:
              precision    recall  f1-score   support

           0       0.24      0.30      0.27      1250
           1       0.25      0.27      0.26      1250
           2       0.33      0.45      0.38      1250
           3       0.35      0.44      0.39      1250
           4       0.38      0.45      0.41      1250
           5       0.32      0.28      0.30      1250
           6       0.68      0.52      0.59      1250
           7       0.41      0.21      0.27      1250
           8       0.66      0.30      0.41      1250
           9       0.34      0.45      0.39      1250

    accuracy                           0.37     12500
   macro avg       0.40      0.37      0.37     12500
weighted avg       0.40      0.37      0.37     12500


Elapsed time (sec):  631.9693984985352
```

Figure 2.11: Some performance statistics about the above snippet with parameter setting **'C'=1, 'kernel'='sigmoid', 'gamma'=1.0** achieving an accuracy of 36.63%

How much time did it take to train and fit this model with the parameter setting **'C'=1, 'kernel'='sigmoid', 'gamma'=1.0**?
It took just 631.9693984985352 seconds = 10 minutes 32 seconds!
I have noticed that the **Sigmoid** kernels take much less compute time than their **RBF** counterparts with the same parameter settings!

Now, enough of the **RBF** and **Sigmoid**. Let us also delve into the **Polynomial kernel** to see how it also compares with the RBF and Sigmoid on this dataset and a few selected hyperparameters!

```
param_grid = {
    'C':[1, 10],
    'degree':[3, 4, 5],
    'kernel':['poly']
    }
```

```
# Create the GridSearchCV object with 3-fold cross-validation
grid_search = GridSearchCV(SVC(), param_grid, cv=3, scoring='accuracy', refit=True, n_jobs=-1, verbose=2)
```

Similar to the previous Grid Search, this grid search is also conducted with 3-fold cross-validation, and as such, obtained the following results below.

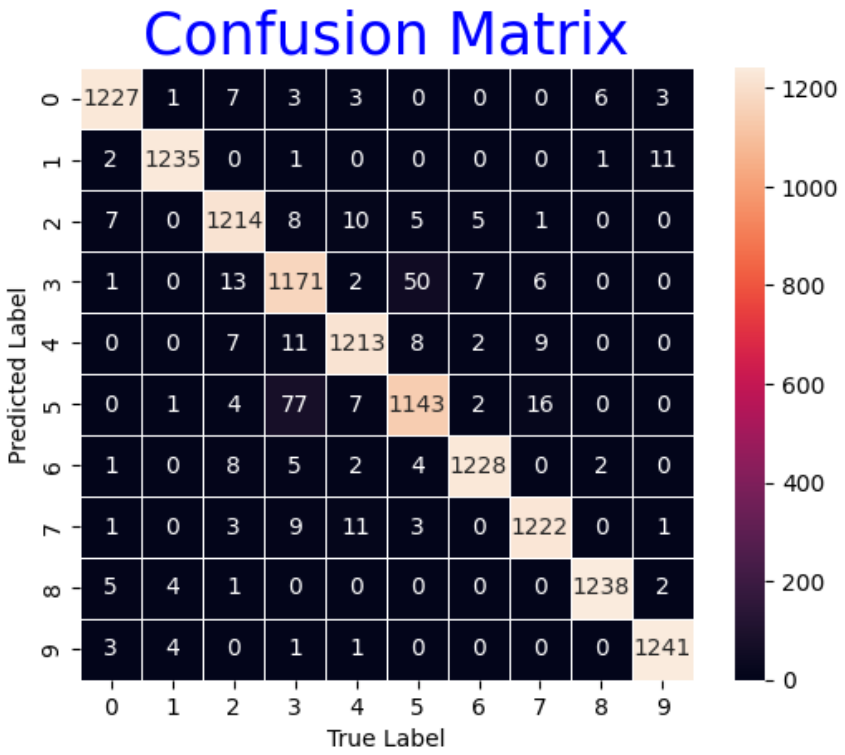


Figure 2.12: Confusion matrix of the best estimator of the above code snippet, retrained on the entire training dataset (all the 3-folds)

```
The accuracy score is 97.06%.

classification_report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      1250
           1       0.99      0.99      0.99      1250
           2       0.97      0.97      0.97      1250
           3       0.91      0.94      0.92      1250
           4       0.97      0.97      0.97      1250
           5       0.94      0.91      0.93      1250
           6       0.99      0.98      0.98      1250
           7       0.97      0.98      0.98      1250
           8       0.99      0.99      0.99      1250
           9       0.99      0.99      0.99      1250

    accuracy                           0.97     12500
   macro avg       0.97      0.97      0.97     12500
weighted avg       0.97      0.97      0.97     12500


Elapsed time:  1874.115937948227
```

Figure 2.13: Some performance statistics about the above snippet, achieving an accuracy of 97.06%

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_degree	param_kernel	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score
0	104.746063	0.838085	55.401628	2.090254	1	3	poly	{'C': 1, 'degree': 3, 'kernel': 'poly'}	0.96976	0.97000	0.97120	0.970320	0.000630	2
1	175.225010	0.770712	74.187569	0.931219	1	4	poly	{'C': 1, 'degree': 4, 'kernel': 'poly'}	0.95432	0.95536	0.95960	0.956427	0.002284	5
2	258.339731	2.289120	98.369546	1.322390	1	5	poly	{'C': 1, 'degree': 5, 'kernel': 'poly'}	0.92272	0.92384	0.92880	0.925120	0.002642	6
3	53.475822	0.394455	38.880293	1.412372	10	3	poly	{'C': 10, 'degree': 3, 'kernel': 'poly'}	0.97288	0.97136	0.97360	0.972613	0.000934	1
4	82.889883	1.961443	49.871847	1.426062	10	4	poly	{'C': 10, 'degree': 4, 'kernel': 'poly'}	0.97016	0.96896	0.97048	0.969867	0.000654	3
5	119.407791	0.610622	51.076756	12.596276	10	5	poly	{'C': 10, 'degree': 5, 'kernel': 'poly'}	0.96032	0.96136	0.96360	0.961760	0.001369	4

Figure 2.14: More statistics about the training of the above code snippet. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

As can be seen in the figures, the best (hyper) parameter setting for the Polynomial is:

```
best_svm_polymodel: SVC(C=10, kernel='poly'),
grid_search.best_params_: {'C': 10, 'degree': 3, 'kernel': 'poly'},
grid_search.best_score_: 0.9726133333333333
```

How much time did it take to train and fit the polynomials with Grid search and 3-fold cross-validation?
It took around 1874.11593794822749 seconds = 31 minutes 14 seconds!
This compute time is still better than the RBF counterparts (without the sk-learn's default parameters)

2.1.1 Best Performing SVM Model

Among the diverse SVM models with varying hyperparameters, the one that emerges as the best performer is as follows:

The SVM model with default values ($C = 1$, **gamma = 'scale'**, and **kernel = 'rbf'**) achieves the best results on the test dataset, as also in the case of the dataset 1 part. In this dataset 2, it achieves an **Accuracy of** $\approx 97\%$.

Just like previously stated in the dataset 1 part, the RBF with the default hyperparameters achieves the best result. However, since I did not perform any cross-validation on it to ensure that it was not lucky on that particular dataset, the accuracy may not be a stable estimate. To Achieve a better estimate of the accuracy, as done previously, we can perform a cross-validation on it and average the results. This way, we can report the average result over the folds.

```
start_time = time.time()

clf = SVC(kernel="rbf", C=1, gamma="scale" )
scores = cross_val_score(clf, X_train, Y_train, cv=5)

end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time (sec): ", elapsed_time)

scores
```

Elapsed time (sec): 342.9431915283203
array([0.97266667, 0.97386667, 0.97146667, 0.9736 , 0.97533333])

The mean score and the standard deviation are hence given by:

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

The cross-validated accuracy is **97.34%** with a remarkably low standard deviation of 0.0013. (Impressive as well!)

This cross-validated result not only reaffirms the model's top position but also provides a deeper understanding of its stability and consistency.

Therefore, we can conclude that this is the best performant model among all the SVM variants tried out in this report with the dataset 2.

2.2 Gaussian NB Training and Testing Details

```python
# Define the parameter grid to search
params_gnb = {'var_smoothing': [1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6]}
# params_gnb = {'var_smoothing': np.logspace(0,-9, num=10)}

# Create the GridSearchCV object with 3-fold cross-validation
grid_search_gnb = GridSearchCV(gnb, params_gnb, cv=10, scoring='accuracy', refit=True, n_jobs=-1, verbose=2)
```

In this homework I tried varying only the variance smoothing as a hyperparameter as already explained in the dataset 1 part.

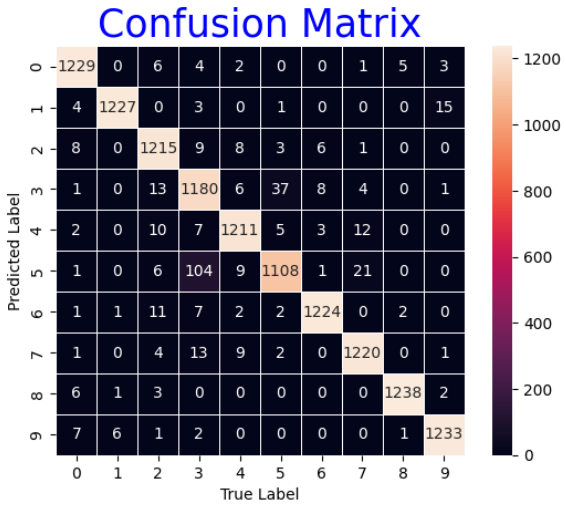


Figure 2.15: Confusion matrix of the best estimator of Gaussian Naive Bayes, retrained on the entire training dataset (all the 10-folds)

```
The accuracy score is 96.68%.

classification_report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      1250
           1       0.99      0.98      0.99      1250
           2       0.96      0.97      0.96      1250
           3       0.89      0.94      0.92      1250
           4       0.97      0.97      0.97      1250
           5       0.96      0.89      0.92      1250
           6       0.99      0.98      0.98      1250
           7       0.97      0.98      0.97      1250
           8       0.99      0.99      0.99      1250
           9       0.98      0.99      0.98      1250

    accuracy                           0.97     12500
   macro avg       0.97      0.97      0.97     12500
weighted avg       0.97      0.97      0.97     12500


Elapsed time:  53.72991371154785
```

Figure 2.16: Some performance statistics about the Gaussian Naive Bayes, achieving an accuracy of 96.68%

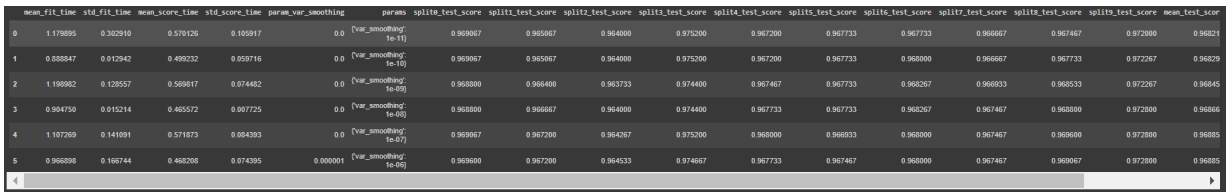


Figure 2.17: More statistics about the training of Gaussian Naive Bayes. It also shows the performances of each model on each split during the Grid Search and Cross-validation hyperparameter search processes.

As can be seen in the figures, the best (hyper) parameter settings for the Gaussian Naive Bayes are:

```
best_gnb_model: GaussianNB(var_smoothing=1e-07),
grid_search_gnb.best_params_: {'var_smoothing': 1e-07},
grid_search_gnb.best_score_: 0.9688533333333332
```

Chapter 3

Conclusion

Firstly, we have seen that both datasets 1 and 2 were simple enough for the models and various hyperparameters to fit them well, achieving astounding results of 99% and 97% respectively!

Secondly, about performance, we have seen that the **RBF** outperforms its **Sigmoid** and **Poly** conterparts using the **SVM** with the default sk-learn values. The default values also outperform the other ones I have tried out in this homework. The SVM with the RBF kernel also outperforms the Gaussian Naive Bayes on both datasets, but does so with just a slight difference in performance results of less that 0.5%.

Thirdly, about compute time, we have seen how computational hungry is the RBF kernels. They use far more compute time than any of their counterparts! With the **gamma='scale' and 'auto'**, the RBF kernel was comparable with the others, and the time it took was almost negligible. However, if different gamma is used, especially when gamma is a float value, the RBF takes a lot of compute time for each of the parameter settings I have tried out in this homework. Some of the compute time were close to one hour (1hr) in the extreme! For the Gaussian Naive Bayes, it took far less time for the computations. The exact compute times and the performances are illustrated throughout this report for reference.

Was this result surprising that the RBF outperforms the others? We know that the RBF is the equivalent of the polynomial kernel with infinite dimension. So, it is not surprising that it outperforms!