Zalando Practice Challenge, Modelling Notebook

1. Introduction

In this notebook a research on which machine learning model will be able to have the highest prediction rate for the Fashion MNIST dataset.

---

2. SVM

2.1 Introduction

Firstly, we will be experimenting with an SVM (support vector machine). As this problem is of a classification type, a SVM would probably yield great results.

2.2 Importing libraries

Firstly, we will be importing the necessary tools for the problem. We will be using the *statistics* module for any type of statistical computation, *matplotlib* will be used for visualization (for more complex visualizations we will be using *seaborn*) and *numpy* will be used for vector computation(for the great performance benefits mainly). For data manipulation and data loading, and the advantage of using a DataFrame we will be using *pandas*. For the machine learning portion of the project, we will be using *scikit-learn* with all the available mechanics inside.

In [1]:
```
from statistics import mean
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.model_selection import cross_validate
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
```

2.3 Importing the data

In [2]:
```
_train_data = pd.read_csv('../data/fashion-mnist_train.csv')
_test_data = pd.read_csv('../data/fashion-mnist_test.csv')
```

In [3]:
```
# select all the rows and columns omitting the label column on index 0
train_data = _train_data.iloc[:,1:]
# select all the rows omitting all the pixel data and taking only the label column at i
# transposing the label data so we can get label as an index
train_label = pd.DataFrame([_train_data.iloc[:, 0]]).T
```

```
# select all the rows and columns from the test dataset
test_data = _test_data.iloc[:,0:]
```

2.4 Image data visualization

In [4]:
```
train_label.value_counts()
```

Out[4]:
```
label
0        6000
1        6000
2        6000
3        6000
4        6000
5        6000
6        6000
7        6000
8        6000
9        6000
dtype: int64
```

The data consists of 10 categories each having 6000 entries

Now it would be easier visualize the data by mapping a human-readable category.

In [5]:
```
# taken from the official documentation of the dataset
categories_mapping={
    0 :'T-shirt/Top',
    1 :'Trouser',
    2 :'Pullover',
    3 :'Dress',
    4 :'Coat',
    5 :'Sandal',
    6 :'Shirt',
    7 :'Sneaker',
    8 :'Bag',
    9 :'Ankle boot'
}
train_label['category'] = train_label['label'].map(categories_mapping)
train_label.head(10)
```

Out[5]:

	label	category
0	2	Pullover
1	9	Ankle boot
2	6	Shirt
3	0	T-shirt/Top
4	3	Dress
5	4	Coat
6	4	Coat
7	5	Sandal
8	4	Coat

	label	category
9	8	Bag

In [6]:
```
# images are 28x28 pixels, totaling to 28^2 (784) in each row
print(len(train_data.to_numpy()[0]))
```

784

In [7]:
```
# length of visualization
length = 7
# width of visualization
width = 7
fig, axes = plt.subplots(length, width, figsize = (10, 10))
# flatten the axes
axes = axes.ravel()
for i in range(length * width):
    # the columns in the csv are the pixels so we reshape them to a 28x28, and we plot
    axes[i].imshow(train_data.to_numpy().reshape(train_data.shape[0], 28, 28)[i])
    axes[i].axis('off')
```

This way of representing the data seems off, so it would be helpful to display the images as

grayscale and have the category visible

In [8]:
```
# length of visualization
length = 7
# width of visualization
width = 7
fig, axes = plt.subplots(length, width, figsize = (15, 15))
# flatten the axes without making a copy
axes = axes.ravel()
for i in range(length * width):
    # the columns in the csv are the pixels so we reshape them to a 28x28, and we plot
    axes[i].imshow(train_data.to_numpy().reshape(train_data.shape[0], 28, 28)[i], cmap=
    axes[i].set_title('cat {label}: {category}'.format(label = train_label['label'][i],
    axes[i].axis('off')
```

2.5 Splitting the data

1. We are splitting the data using _sklearn.model_selection.test_train_split_.

2. We are splitting the data into training and testing. **X_train** denotes the training data of all the independent variables, **y_train** denotes the dependent variable that we will try to predict using this model, **X_test** denotes the independent variables that will not be used in the training process as they will be used to make predictions about the accuracy of the model, **y_test** denotes the labels for the test data that will be used to test the accuracy between the actual and predicted category.

In [9]:
```
train_label = pd.DataFrame([_train_data.iloc[:, 0]]).T
X_train, X_test, y_train, y_test = train_test_split(train_data, train_label, random_sta
```

2.5 Hyperparameter tuning

Following, we have to find the best possible set of parameters that will give us the most accurate model.

In [10]:
```
data_points = [500, 1000, 2000, 5000, 10_000, 15_000, 20_000, 30_000, 40_000]
```

Here we create an empty accuracy list that will later be populated with the accuracy each of the previously created data points yield.

In [11]:
```
accuracy_points = []
```

Next we loop through all the arbitrary data points we have previously set up and check the cross validation scores. The cross validation scores help us not loop over the same labels meaning that we will be overfitting the model, which is bad since we want to make it usable for more than this dataset alone.

In [12]:
```
#for point_amount in data_points:
#    # These are the default parameters according to scikit-learn
#    clf = SVC(gamma='scale', kernel='rbf', C=1)
#
#    # This yields a dictonary having 'test_score' as one of the keys which gives the s
#    scores = cross_validate(clf, X_train[:point_amount], y_train[:point_amount], cv=3)
#
#    print('score for each fold: {fold_score}'.format(fold_score=scores['test_score']))
#    print('mean accuracy: {mean_accuracy}'.format(mean_accuracy=mean(scores['test_scor
#
#    accuracy_points.append(mean(scores['test_score']))
#print(accuracy_points)
```

The code above does a great job putting the accuracy scores for each fold for each of the data points, but maybe they are a bit too many, so we should lower the data points a little bit. Also for higher accuracy we have increased the number of folds to 5 and then getting the mean value.

In [13]:
```
data_points = [500, 1000, 2000, 5000, 10_000, 15_000, 20_000]
```

In [14]:
```
# hardcoded the accuracy points findings to save time on re-running
accuracy_points = [0.732, 0.8019999999999999, 0.8195, 0.8382, 0.8579, 0.865533333333333
for point_amount in data_points:
```

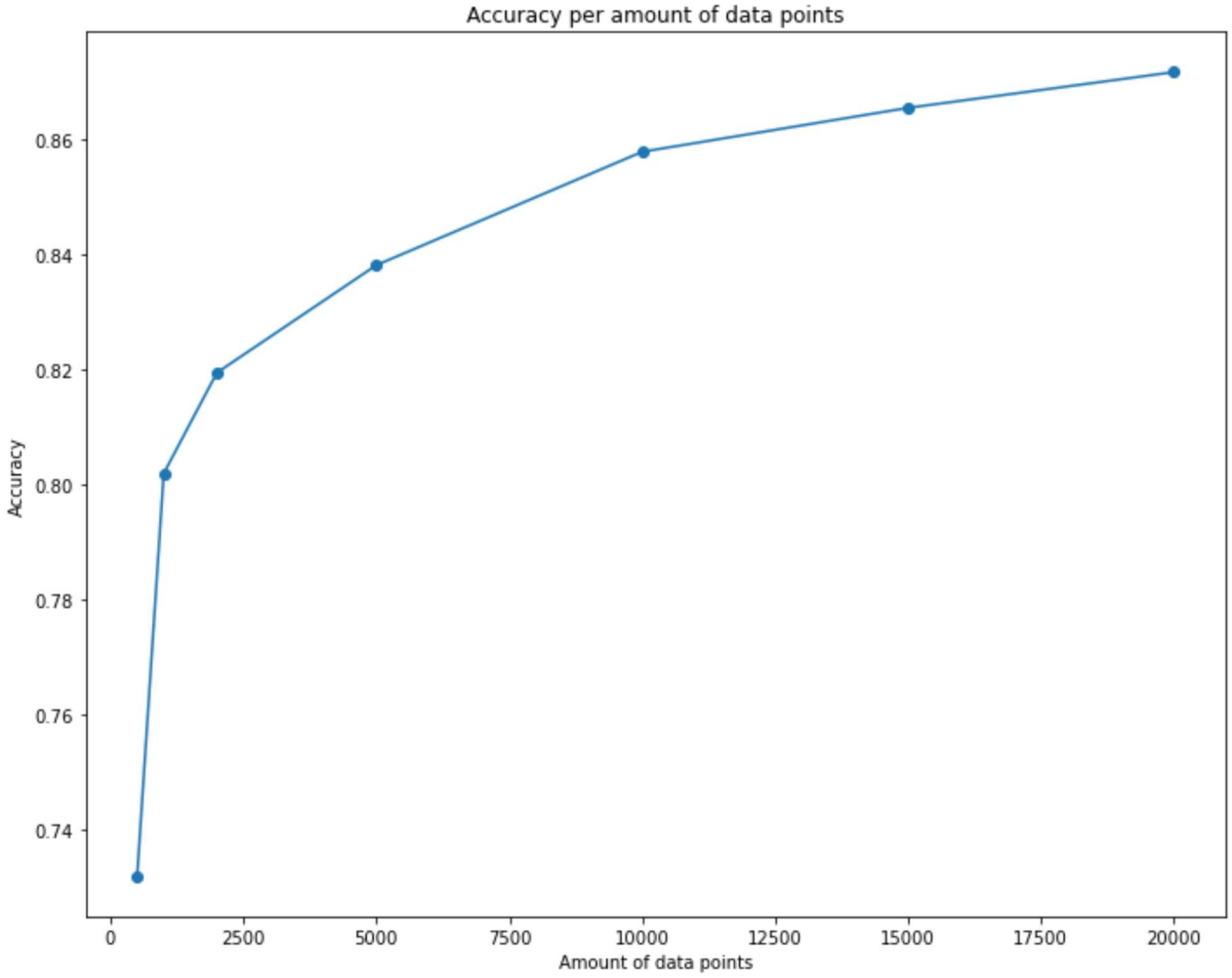
```
    # This stops the loop from executing if there are already calculated accuracy point
    if len(accuracy_points) > len(data_points)-1:
        break

    # These are the default parameters according to scikit-learn
    clf = SVC(gamma='scale', kernel='rbf', C=1)

    # This yields a dictonary having 'test_score' as one of the keys which gives the sc
    scores = cross_validate(clf, X_train[:point_amount], y_train.to_numpy().ravel()[:po

    print('score for each fold: {fold_score}'.format(fold_score=scores['test_score']))
    print('mean accuracy: {mean_accuracy}'.format(mean_accuracy=mean(scores['test_score

    accuracy_points.append(mean(scores['test_score']))
print(accuracy_points)
```

```
[0.732, 0.8019999999999999, 0.8195, 0.8382, 0.8579, 0.8655333333333334, 0.87175]
```

We can use this information to visualize the classifier's accuracy related to how many data points we give it to train with.

In [15]:

```
plt.figure(figsize=(10,8))
plt.title('Accuracy per amount of data points')
plt.xlabel('Amount of data points')
plt.ylabel('Accuracy')
plt.tight_layout(rect=(1, 1, 2, 2))
plt.plot(data_points, accuracy_points, '-o')
plt.show()
```



The data provides us a clear overview of how the accuracy improves with the amount of data points and it looks satisfying at the 12_500-17_500 mark. For our purposes using 15_000 data points looks reasonable. We can now move on to the actual tuning of the hyperparameters.

Firstly, we will adjust the necessary data based on our findings

In [16]:
```
X_train = X_train.iloc[:15000]
y_train = y_train.iloc[:15000]
```

Next, we will be using an exhaustive grid search to fine-tune the hyperparameters and see which outcomes will yield the highest accuracy

In [17]:
```
#param_grid = {
#    'C': [0.1, 1, 10, 100, 1000],
#    'kernel': ['linear', 'poly', 'rbf', 'sigmoid', 'precomputed'],
#    'gamma': ['auto', 'scale']
#}
#
## We want to be able to predict to test out the different parameters
#svm, cv = SVC(**param_grid, probability=True), KFold(n_splits=5)
#
#clf = GridSearchCV(svm, param_grid=param_grid, cv=cv, n_jobs=-1)
#clf.fit(X_train, y_train)
#print(clf)
```

This iteration of the testing takes a really long time, so we will be using only one gamma value and that is passed as 'scale' in the parameter list. We will also reduce the number of folds to 2. Moreover, we will also be looking at the log loss scoring with smaller numbers meaning higher predictrion rate (closer to 0 means less uncertainty).

In [18]:
```
param_grid = {
    'C': [0.1, 1, 7, 10, 50, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'gamma': ['scale']
}

# We want to be able to predict to test out the different parameters
svm, cv = SVC(**param_grid, probability=True), KFold(n_splits=2)

clf = GridSearchCV(svm, param_grid=param_grid, cv=cv, n_jobs=-1, scoring='neg_log_loss'
clf.fit(X_train, y_train.to_numpy().ravel())
```

```
Fitting 2 folds for each of 24 candidates, totalling 48 fits
```

Out[18]:
```
GridSearchCV(cv=KFold(n_splits=2, random_state=None, shuffle=False),
             estimator=SVC(C=[0.1, 1, 7, 10, 50, 100], gamma=['scale'],
                           kernel=['linear', 'poly', 'rbf', 'sigmoid'],
                           probability=True),
             n_jobs=-1,
             param_grid={'C': [0.1, 1, 7, 10, 50, 100], 'gamma': ['scale'],
                         'kernel': ['linear', 'poly', 'rbf', 'sigmoid']},
             scoring='neg_log_loss', verbose=7)
```

After that fitting we can take a look at all the parameters used in this training session.

In [19]:
```
print(clf.cv_results_['params'])
```

```
[{'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}, {'C': 0.1, 'gamma': 'scale', 'kerne
l': 'poly'}, {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'}, {'C': 0.1, 'gamma': 'scale',
'kernel': 'sigmoid'}, {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}, {'C': 1, 'gamma':
'scale', 'kernel': 'poly'}, {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}, {'C': 1, 'gamm
a': 'scale', 'kernel': 'sigmoid'}, {'C': 7, 'gamma': 'scale', 'kernel': 'linear'}, {'C':
7, 'gamma': 'scale', 'kernel': 'poly'}, {'C': 7, 'gamma': 'scale', 'kernel': 'rbf'},
{'C': 7, 'gamma': 'scale', 'kernel': 'sigmoid'}, {'C': 10, 'gamma': 'scale', 'kernel':
'linear'}, {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}, {'C': 10, 'gamma': 'scale', 'k
ernel': 'rbf'}, {'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}, {'C': 50, 'gamma': 'sc
ale', 'kernel': 'linear'}, {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}, {'C': 50, 'gam
ma': 'scale', 'kernel': 'rbf'}, {'C': 50, 'gamma': 'scale', 'kernel': 'sigmoid'}, {'C':
100, 'gamma': 'scale', 'kernel': 'linear'}, {'C': 100, 'gamma': 'scale', 'kernel': 'pol
y'}, {'C': 100, 'gamma': 'scale', 'kernel': 'rbf'}, {'C': 100, 'gamma': 'scale', 'kerne
l': 'sigmoid'}]
```

With that information we can firstly, create 2 DataFrames that would contain all the 'C' and 'kernel' values, and secondly visualize the most fitting set of parameters via a heatmap with all the correlations.

In [20]:
```
C_values = [clf.cv_results_['params'][i]['C'] for i in range(len(clf.cv_results_['param
```

In [21]:
```
C_values
```

Out[21]:
```
[0.1,
 0.1,
```

```
0.1,
0.1,
1,
1,
1,
1,
7,
7,
7,
7,
10,
10,
10,
10,
50,
50,
50,
50,
100,
100,
100,
100]
```

In [22]:
```
kernel_values = [clf.cv_results_['params'][i]['kernel'] for i in range(len(clf.cv_resul
kernel_values
```

Out[22]:
```
['linear',
 'poly',
 'rbf',
 'sigmoid',
 'linear',
 'poly',
 'rbf',
 'sigmoid',
 'linear',
 'poly',
 'rbf',
 'sigmoid',
 'linear',
 'poly',
 'rbf',
 'sigmoid',
 'linear',
 'poly',
 'rbf',
 'sigmoid',
 'linear',
 'poly',
 'rbf',
 'sigmoid']
```

We will also need the sought after log loss.

In [23]:
```
log_loss_scores = clf.cv_results_['mean_test_score']
log_loss_scores
```

Out[23]:
```
array([-0.53736294, -0.63001458, -0.53264917, -1.43187504, -0.53690614,
       -0.54458555, -0.39408855, -1.58368036, -0.53762569, -0.5311421 ,
       -0.35913935, -1.7197236 , -0.53729415, -0.53199235, -0.36089725,
       -1.73659341, -0.53690072, -0.5371747 , -0.37216665, -1.7680862 ,
       -0.53693408, -0.54015065, -0.37323339, -1.75613003])
```

In [24]:
```
results = pd.DataFrame({'C': C_values, 'kernel': kernel_values, 'log_loss_score': log_l
results.head()
```

Out[24]:

	C	kernel	log_loss_score
0	0.1	linear	-0.537363
1	0.1	poly	-0.630015
2	0.1	rbf	-0.532649
3	0.1	sigmoid	-1.431875
4	1.0	linear	-0.536906

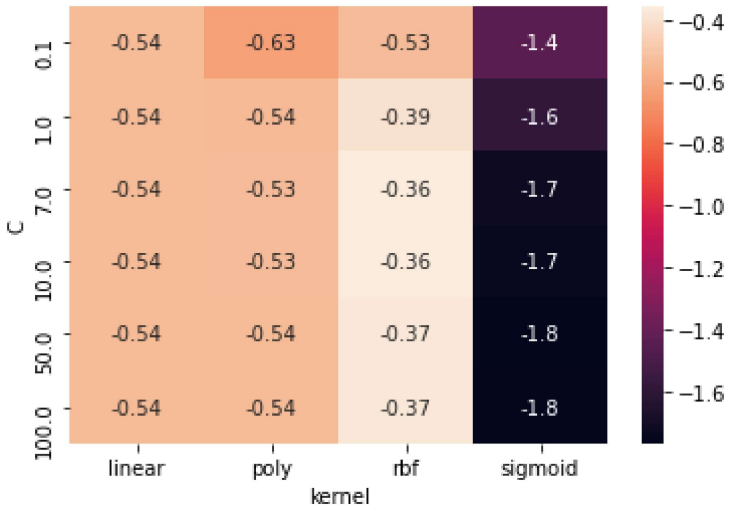
We are looking for the log loss score for each of the kernels with one of the predefined C values.

In [25]:
```
findings = results.pivot(values='log_loss_score', columns='kernel', index=['C'])
findings
```

Out[25]:

kernel	linear	poly	rbf	sigmoid
C				
0.1	-0.537363	-0.630015	-0.532649	-1.431875
1.0	-0.536906	-0.544586	-0.394089	-1.583680
7.0	-0.537626	-0.531142	-0.359139	-1.719724
10.0	-0.537294	-0.531992	-0.360897	-1.736593
50.0	-0.536901	-0.537175	-0.372167	-1.768086
100.0	-0.536934	-0.540151	-0.373233	-1.756130

In [26]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(findings, annot=True)
plt.show()
```



We can observe that the smallest log loss is observed with a C value of either 7 or 10 and the rbf kernel.

3. Training the model

Now that we are settled on our parameters we can train the actual model.

In [27]:
```
clf = SVC(gamma='scale', C=7, kernel='rbf')

clf.fit(X_train, y_train.to_numpy().ravel())

scores = cross_validate(clf, X_train, y_train.to_numpy().ravel(), cv=5)

print('score for each fold: {fold_score}'.format(fold_score=scores['test_score']))
print('mean accuracy: {mean_accuracy}'.format(mean_accuracy=mean(scores['test_score'])))
```

```
score for each fold: [0.882      0.873      0.88033333 0.88633333 0.885     ]
mean accuracy: 0.8813333333333333
```

This concludes that our model has an approximate accuracy of 0.88.

4 Predicting the outcomes

Finally, now that the model has been fitted we can predict based on the test dataset.

In [31]:
```
y_test_pred = clf.predict(X_test)
y_test_pred
```

Out[31]:
```
array([7, 8, 8, ..., 3, 5, 0], dtype=int64)
```

Firstly, we can compute the frequency of categories from the test dataset compared to the predicted results. This will give us a confusion matrix that we can later visualize.

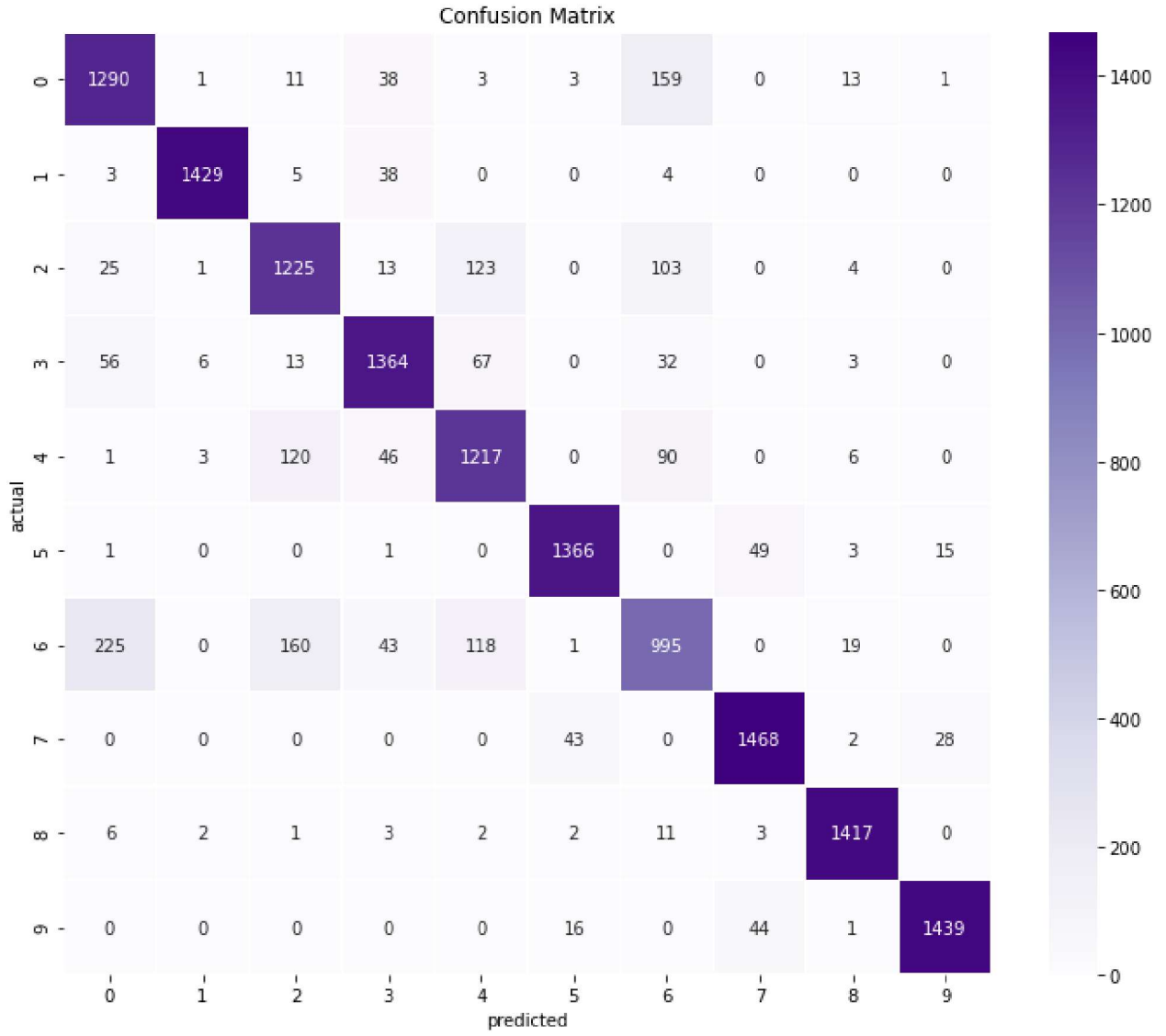
In [34]:
```
cross_tabulation = pd.crosstab(pd.Series(y_test.to_numpy().flatten(), name='actual'),
                               pd.Series(y_test_pred, name='predicted'))
cross_tabulation
```

Out[34]:

predicted	0	1	2	3	4	5	6	7	8	9
actual										
0	1290	1	11	38	3	3	159	0	13	1
1	3	1429	5	38	0	0	4	0	0	0
2	25	1	1225	13	123	0	103	0	4	0
3	56	6	13	1364	67	0	32	0	3	0
4	1	3	120	46	1217	0	90	0	6	0
5	1	0	0	1	0	1366	0	49	3	15
6	225	0	160	43	118	1	995	0	19	0
7	0	0	0	0	0	43	0	1468	2	28
8	6	2	1	3	2	2	11	3	1417	0
9	0	0	0	0	0	16	0	44	1	1439

Now we can create the confusion matrix based on the results we got from the cross tabulation computation.

In [60]:
```
plt.figure(figsize=(12, 10))
plt.title('Confusion Matrix')
sns.heatmap(cross_tabulation, annot=True, fmt='d', cmap='Purples', linewidths=.25)
plt.show()
```



5. Conclusion

Based on our tuning and manipulations, we can conclude that the model has an accuracy of 88%. Further optimizations to make the model run faster and improve accuracy could include scaling dimensiality via PCA i.e. since SVMs are affected by distance.

Extra:

1. Fitting the model with scaled data

In [46]:
```
train_data = _train_data.iloc[:,1:]
```

```
train_label = pd.DataFrame([_train_data.iloc[:, 0]]).T
test_data = _test_data.iloc[:,0:]
X_train, X_test, y_train, y_test = train_test_split(train_data, train_label, random_sta
X_train = X_train.iloc[:15000]
y_train = y_train.iloc[:15000]
```

Since RGB are 8 bit each, we can reduce the combination range to its 0-1 range representation by dividing by 255.

In [48]:
```
X_train = X_train/255
```

In [49]:
```
clf = SVC(gamma='scale', C=7, kernel='rbf')

clf.fit(X_train, y_train.to_numpy().ravel())

scores = cross_validate(clf, X_train, y_train.to_numpy().ravel(), cv=5)

print('score for each fold: {fold_score}'.format(fold_score=scores['test_score']))
print('mean accuracy: {mean_accuracy}'.format(mean_accuracy=mean(scores['test_score'])))
```

```
score for each fold: [0.882      0.873      0.88033333 0.88633333 0.885     ]
mean accuracy: 0.8811333333333333
```

We can observe that there is no real difference except performance-wise.