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 _mnist_reader_ provided in the dataset for reading the data, 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 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 [25]:
```
import sys

sys.path.insert(1, '../utils')

import mnist_reader
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.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 and splitting the data

1. We are reading the data using the provided tooling.
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 [26]:
```
X_train, y_train = mnist_reader.load_mnist('../data/fashion', kind='train')
X_test, y_test = mnist_reader.load_mnist('../data/fashion', kind='t10k')
```

2.4 Image data visualization

2.5 Hyperparameter tuning

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

Firstly, out of curiosity, we are having a look at how many data points exist in the provided dataset.

In [27]:
```
np.shape(X_train)
```

Out[27]: (60000, 784)

In [28]:
```
np.shape(y_train)
```

Out[28]: (60000,)

In [29]:
```
np.shape(X_test)
```

Out[29]: (10000, 784)

In [30]:
```
np.shape(y_test)
```

Out[30]: (10000,)

As we observe we can see that we have 60_000 data points for the training data and 10_000 for the testing data. We might want to find out how much data we might need to take full advantage of the model. We can create a list of arbitrary values that we will later use to plot out and find the most suitable sample size.

In [31]:
```
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 [32]:
```
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 [33]:
```
#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 [34]:
```
data_points = [500, 1000, 2000, 5000, 10_000, 15_000, 20_000]
```

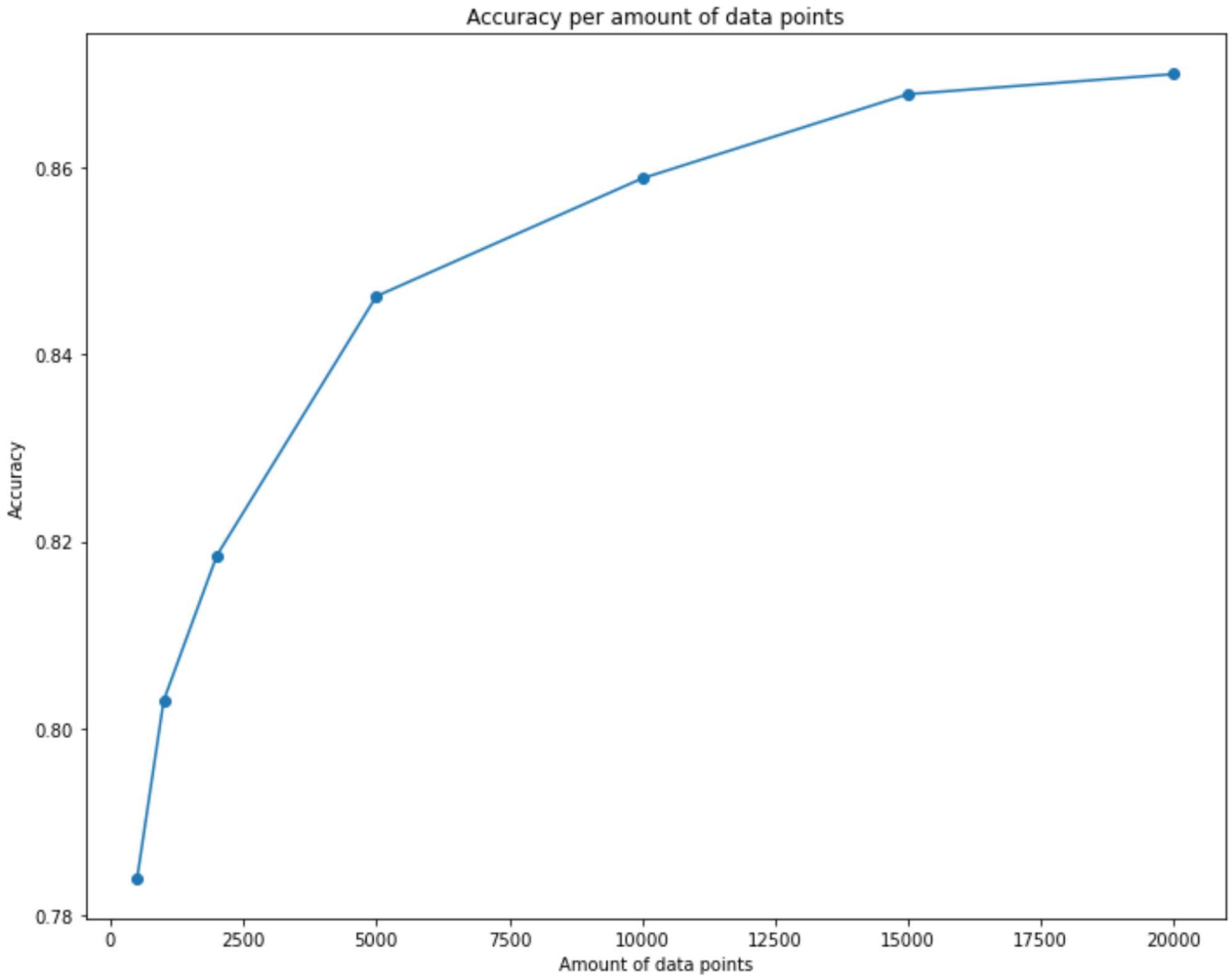
In [35]:
```
# hardcoded the accuracy points findings to save time on re-running
accuracy_points = [0.784, 0.803, 0.8185, 0.8462, 0.8588, 0.8678, 0.86995]
for point_amount in data_points:
    # This stops the loop from executing if there are already calculated accuracy point
    if len(accuracy_points) > 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[:point_amount], 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

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

```
[0.784, 0.803, 0.8185, 0.8462, 0.8588, 0.8678, 0.86995]
```

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

In [36]:
```
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')
```

Out[36]:
```
[<matplotlib.lines.Line2D at 0x1eaea9fdf40>]
```



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 [37]:
```
X_train = X_train[:15000]
y_train = y_train[: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 [38]:
```
#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 [15]:
```
param_grid = {
    'C': [0.1, 1, 10, 100, 1000],
    '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)
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits
[-0.53889925 -0.62292649 -0.52800663 -1.4424809  -0.53865711 -0.54069928
 -0.39115319 -1.58459449 -0.53956922 -0.53172984 -0.35640576 -1.72514058
 -0.5392021  -0.54759828 -0.36885034 -1.70939767 -0.53907017 -0.56022741
 -0.36898101 -1.72521135]
```

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

In [39]:
```
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': 10, 'gamma': 'scale', 'kernel': 'linear'},
{'C': 10, 'gamma': 'scale', 'kernel': 'poly'}, {'C': 10, 'gamma': 'scale', 'kernel': 'rb
f'}, {'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}, {'C': 100, 'gamma': 'scale', 'ker
nel': 'linear'}, {'C': 100, 'gamma': 'scale', 'kernel': 'poly'}, {'C': 100, 'gamma': 'sc
ale', 'kernel': 'rbf'}, {'C': 100, 'gamma': 'scale', 'kernel': 'sigmoid'}, {'C': 1000,
'gamma': 'scale', 'kernel': 'linear'}, {'C': 1000, 'gamma': 'scale', 'kernel': 'poly'},
{'C': 1000, 'gamma': 'scale', 'kernel': 'rbf'}, {'C': 1000, 'gamma': 'scale', 'kernel':
'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 [40]:
```
C_values = [clf.cv_results_['params'][i]['C'] for i in range(len(clf.cv_results_['param
```

In [41]:
```
C_values
```

Out[41]:
```
[0.1,
 0.1,
 0.1,
 0.1,
 1,
 1,
 1,
 1,
```

```
 10,
 10,
 10,
 10,
 100,
 100,
 100,
 100,
 1000,
 1000,
 1000,
 1000]
```

In [42]:
```
kernel_values = [clf.cv_results_['params'][i]['kernel'] for i in range(len(clf.cv_resul
kernel_values
```

Out[42]:
```
['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 [48]:
```
log_loss_scores = clf.cv_results_['mean_test_score']
log_loss_scores
```

Out[48]:
```
array([-0.53889925, -0.62292649, -0.52800663, -1.4424809 , -0.53865711,
       -0.54069928, -0.39115319, -1.58459449, -0.53956922, -0.53172984,
       -0.35640576, -1.72514058, -0.5392021 , -0.54759828, -0.36885034,
       -1.70939767, -0.53907017, -0.56022741, -0.36898101, -1.72521135])
```

In [49]:
```
results = pd.DataFrame({'C': C_values, 'kernel': kernel_values, 'log_loss_score': log_l
results.head()
```

Out[49]:

	C	kernel	log_loss_score
0	0.1	linear	-0.538899
1	0.1	poly	-0.622926
2	0.1	rbf	-0.528007
3	0.1	sigmoid	-1.442481
4	1.0	linear	-0.538657

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

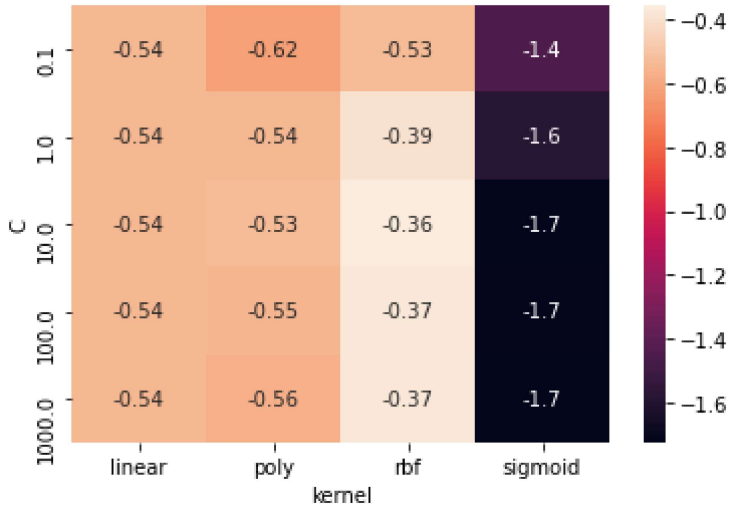
In [59]:
```
findings = results.pivot(values='log_loss_score', columns='kernel', index=['C'])
findings
```

Out[59]:

kernel	linear	poly	rbf	sigmoid
C				
0.1	-0.538899	-0.622926	-0.528007	-1.442481
1.0	-0.538657	-0.540699	-0.391153	-1.584594
10.0	-0.539569	-0.531730	-0.356406	-1.725141
100.0	-0.539202	-0.547598	-0.368850	-1.709398
1000.0	-0.539070	-0.560227	-0.368981	-1.725211

In [75]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(findings, annot=True)
```

Out[75]:
```
<AxesSubplot:xlabel='kernel', ylabel='C'>
```



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

3. Training the model

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

In [77]:
```
clf = SVC(gamma='scale', C=10, kernel='rbf')

clf.fit(X_train, y_train)

scores = cross_validate(clf, X_train, y_train, 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: [0.88033333 0.88533333 0.88233333 0.88533333 0.881     ]
mean accuracy: 0.8828666666666667
```

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