Софийски университет „Св. Кл. Охридски”

Факултет по математика и информатика

Курсов Проект

на тема: „***DigitsRecognizer***”

Студент: Атанас Бисеров Василев, 62577

Курс: 4, Учебна година: 2023/24

Преподаватели: **проф. Иван Койчев**:

==========

Декларация за липса плагиатство:

- Плагиатство е да използваш, идеи, мнение или работа на друг, като претендираш, че са твои. Това е форма на преписване.
- Тази курсова работа е моя, като всички изречения, илюстрации и програми от други хора са изрично цитирани.
- Тази курсова работа или нейна версия не са представени в друг университет или друга учебна институция.
- Разбирам, че ако се установи плагиатство в работата ми ще получа оценка “Слаб”.

21.1.24 г. Подпис на студента:

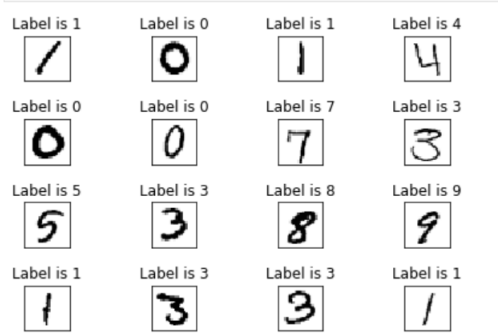
Съдържание

1 Увод

Проектът е фокусиран върху разпознаването на ръчно написани цифри. Идея е да създадем модел, който е способен с висока точност да идентифицира цифри от изображения. Този модел може да се приложи в следните сфери:

- ✓ **Банкови услуги** – Автоматично разпознаване на ръчно попълнени чекове и формуляри за по-ефективни финансови транзакции
- ✓ **Медицина** - Разпознаване на ръчно написани бележки и рецепти, улеснявайки процесите на документация и управление на пациентски данни
- ✓ **Образование** - Разпознаване на ръчно написани отговори на тестове и домашни работи за по-бърза и ефективна оценка.

Моделът приема снимка, на която има ръчно написана цифра и разпознава коя е цифрата. Може да видите примера по-долу:



В този проект ще проучим колко добре различни популярни алгоритми за машинно обучение обработват многомерни данни (снимката я представяме като двумерен масив). Извършваме ***Principal Component Analysis*** върху характеристиките, за да намалим размерността и ускорим обучението на моделите. В този случай PCA се използва като техника за избор на характеристики, за да се намали размерността и така да улесни ***Gaussian SVM*** алгоритъма да извърши задачата по класификацията на ръчно писани цифри.

Ще видим, че най-добрият модел по отношение на точността върху тестовия набор от данни е ***Gaussian*** **SVM** с **98%**, въпреки че обучението му е доста бавно. Не много далеч с точност от **97%** върху тестовия набор е методът на ***K-Nearest Neighbors,*** който е много по-бърз за обучение. Логистичната регресия е най-бързата за обучение, но достига само **90%** точност. Останалите алгоритми (**Decision Tree, Random Forrest, AdaBoost and LinearSVC**) се справят по-слабо по отношение на точността в метриките за ***cross-validation***.

2 Преглед на областа *на разпознаване на ръчно написани символи*

В областта на разпознаването на ръчно написани символи, наблюдаваме активен напредък като се използват различни подходи и техники. Изследванията включват не само традиционни методи, но и иновации, фокусирани върху машинното обучение и невронните мрежи.

Невронните мрежи изпъкват като ключов инструмент за постигане на висока точност и ефективност. Различни архитектури като конволюционни невронни мрежи (CNN) се използват за подобряване на способностите за разпознаване на различни стилове на ръкопис. Трансферното обучение с невронни мрежи се проучва за увеличаване на темповете на обучение и адаптиране към различни стилове на писане. Рекурентните невронни мрежи (RNN) от своя страна подобряват разпознаването на последователни и свързани символи.

Тези иновации представляват ключов фактор за постигане на висока прецизност в разпознаването на ръчно написани цифри. Нашият проект се насочва към интегриране на тези напредъци в създаването на модел, който не само отговаря на изискванията на различни бизнес сфери, където разпознаването на ръчно написани символи е от съществено значение, но и предоставя иновативни решения за бъдещето.

Текущото състояние на тази област показва нарастващ интерес и приложения във всички сфери на бизнеса, като търсенето на ефективни решения за разпознаване на ръчно написани символи продължава да расте.

В областта се използват и различни видове алгоритми за машинно обучение за справяне с предизвикателствата. Цялостно се прилагат различни технологии и иновации, за да се подобряват вече съществуващи модели и разпознаването на ръчно написан текст да става все по-точно.

3 Проектиране

Използваме Jupyter Notebook за разработка на проекта – поставяме различни хипотези, тестваме различни алгоритми, мерим резултата и документираме експериментите. Най-добрият модел се експортира в .sav файл, който може да се използва от софтуерните инженери, за да създадат потребителски интерфейс и цялостно приложение, което да се използва от крайните потребители.

Нека разгледаме как са представени данните:

- Използваме Dataset от Kaggle
- Оригиналният dataset на ръчно написани цифри включва обучителен набор от 60,000 примера и тестов набор от 10,000 примера

- Kaggle dataset съдържа два файла - **train.csv** и **test.csv** - и двата съдържат ръчно написани цифри от нула до девет
- Всяка картинка е **28** пиксела висока и **28** пиксела широка, което прави общо **784** пиксела
- ***Всеки пиксел има единична стойност, която показва колко е светъл или тъмен този пиксел, като по-големите числа означават по-тъмни. Тази стойност е цяло число между 0 и 255, включително***
- Обучителният dataset (train.csv) има **785** колони. Първата колона, наречена "**label**", е цифрата, нарисувана от потребителя. Останалите колони съдържат стойностите на пикселите на съответната картинка. Файлът съдържа **42,000** наблюдения
- Всяка колона с пиксели в обучителния набор има име като **pixel_x**, където x е цяло число между 0 и 783, включително. За да намерите този пиксел на картинката, предположете, че сте разложили x като **x = i * 28 + j**, където i и j са цели числа между 0 и 27, включително. Тогава **pixel_x** се намира в ред **i** и колона **j** на матрица **28 x 28** (индексиране от нула)
- Например, pixel_32 показва пиксела, който е в петата колона от ляво и втория ред отгоре
- Тестовият dataset (test.csv) е същият като обучителния, с изключение на това, че не съдържа колоната "label"
- Ще използваме само train.csv, за да можем да разделим данните сами

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pi
41990	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41991	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41992	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41993	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41994	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41996	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41997	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41998	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41999	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

10 rows × 785 columns

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pix
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	...	42000.000000	42000.000000	42000.000000	42000.00000	42000.0
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.219286	0.117095	0.059024	0.02019	0.0
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	6.312890	4.633819	3.274488	1.75987	1.8
min	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
25%	2.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
50%	4.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
75%	7.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
max	9.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	254.000000	254.000000	253.000000	253.00000	254.0

8 rows × 785 columns

4 Реализация, тестване/експерименти

4.1 Използвани технологии, платформи и библиотеки

За реализацията на проекта използваме Python и следните библиотеки:

1. Scikit-learn – библиотека, която съдържа имплементация на модели за машинно обучение и помощни методи за обработка на данни, валидация, разделяне на данните и т.н.
    a. Използваме следните модели:
        i. *KNeighborsClassifier*
        ii. *LogisticRegression*
        iii. *DecisionTreeClassifier*
        iv. *RandomForestClassifier*
        v. *AdaBoostClassifier*
        vi. *LinearSVC*
        vii. *SVC*
2. *Numpy* – за работа с данните
3. *Pandas* – за работа с данните
4. *Matplotlib* – за чертане на диаграми

4.2 Реализация/Провеждане на експерименти

Работен процес:

1. Ще отделим характеристиките (features) от етикетите (labels)
2. На второ място, ще скалираме характеристиките (features), така че да можем да приложим PCA – превръщаме стойността на всяка колона в диапазона от 0 до 1 използвайки *MixMaxScaler*
3. Извършваме PCA
4. Разделяме набора данни на 2 части - обучаващ и тестов поднабор
5. Използваме обучаващия набор, за да оценим средната точност при *cross validation* на няколко алгоритъма и да изберем три най-добре представящи се. Тестваме *Decision Trees, Random Forests, AdaBoost, Logistic Regression, Linear SVM, Gaussian SVM, K-Nearest Neighbors***.** Избираме три с най-висока средна точност при крос валидация
6. Извършваме grid search върху трите алгоритъма, които са в краткия списък, с цел да ги подобрим
7. Тестваме моделите върху тестовия набор

8. Запазваме обучените модели като файлове за бъдещи корекции и също така запазваме Excel файл с някои важни параметри на трите модела, което позволява последващи анализи и, надяваме се, оптимизация.
9. Оценяваме ефективността на най-добрия алгоритъм, използвайки различен брой характеристики и повторно извършваме grid search. Извършваме PCA, за да достигнем ***explained variance*** от **60%, 70%, 80%, 90% и 95%**, като по този начин получаваме различни характеристики. Повторно обучаваме най-добрия алгоритъм с различните характеристики чрез grid search, за да намерим най-добрия набор от характеристики, водещ до най-висока точност без да попадаме в сценарии на ***overfitting***.
10. Намираме най-добрия модел, който постига **98%** точност

Дефиниране на помощни функции:

Започваме с дефиниране на някои функции, които ще бъдат често използвани. Ще дефинираме помощна функция **fit_model**, която ще извърши **grid search** върху нашите модели и ще върне речник със следната информация:

- **model** - съдържа името на алгоритъма
- **number_features_training** - съдържа броя на характеристиките в текущия модел
- **explained_variance_after_pca** - показва *explained variance* след намаляването на размерността, извършено от *PCA*
- **best_estimator** - съдържа най-добрия модел от проведения *grid search*
- **best_params** - съдържа най-добрите хиперпараметри, които дават най-високите метрики (в нашия случай "точност")
- **best_score** - съдържа най-високият резултат за точност
- **time_sec** - съдържа времето за изпълнение на *grid_search*

Този речник се добавя към списъка модели, където ще събираме гореспоменатата информация за всеки алгоритъм. Накрая ще създадем *DataFrame* от списъка с модели и ще го запазим като Excel файл. Това се извършва от помощната функция ***persist_data***, която също така запазва модела.

Също така дефинираме някои функции за изчертаване, които ще ни помогнат да визуализираме цифрите и матрицата по по-приятен начин.

[13]:
```
# performs grid search on the model
def fit_model(model, parameters_grid, k_fold):
    t0 = time()
    grid_search = GridSearchCV(model, parameters_grid, n_jobs = 1,scoring = 'accuracy', cv = k_fold)
    grid_search.fit(X_tr, y_tr)
    t1 = time()-t0

    return{"model":model.__class__.__name__,
           "number_of_observations_training":X_tr.shape[0],
           "number_features_training":X_tr.shape[1],
           "explained_variance_after_pca":explained_variance,
           "best_estimator":grid_search.best_estimator_,
           "best_params": grid_search.best_params_,
           "best_score": grid_search.best_score_,
           "time_sec": t1}
```

[15]:
```
# saves model on the disc and best model parameters in Excel
def persist_data(models, model_fit):
    #save to disc model
    filename = model_fit['model']+".sav"
    dump(model_fit,filename)
    models.append(model_fit)
    model_df = pd.DataFrame.from_dict(models)
    writer = pd.ExcelWriter("output.xlsx")
    model_df.to_excel(writer,'Sheet1')
    writer.save()
```

[16]:
```
# plots the confusion matrix
def draw_confusion_marix(y_ts, predictions, score):
    cm = confusion_matrix(y_ts, predictions)
    plt.figure(figsize=(9,9))
    sns.heatmap(cm, annot=True, fmt="", linewidths=.5, square = True, cmap = 'Blues_r');
    plt.ylabel('Actual label');
    plt.xlabel('Predicted label');
    all_sample_title = 'Accuracy Score: {0:.3%}'.format(score)
    plt.title(all_sample_title, size = 15);
```

[21]:
```
# Plots a single image
def plot_digit(row, w = 28, h = 28, labels = True):
    if labels:
        # the first column contains the label
        label = row.iloc[0]
        # The rest of columns are pixels
        pixels = row[1:]
    else:
        label = ''
        # The rest of columns are pixels
        pixels = row[0:]

#    print(row.shape, pixels.shape)


    # Make those columns into a array of 8-bits pixels
    # This array will be of 1D with length 784
    # The pixel intensity values are integers from 0 to 255
    pixels = 255 - np.array(pixels, dtype = 'uint8')

    # Reshape the array into 28 x 28 array (2-dimensional array)
    pixels = pixels.reshape((w, h))

    # Plot
    if labels:
        plt.title('Label is {label}'.format(label = label))
    plt.imshow(pixels, cmap = 'gray')
```

```
# Plot pictures
def plot_digits(images, size_w = 28, size_h = 28, labels = True, images_per_row = 5):
    images_count = images.shape[0]
    h = np.ceil(images_count / images_per_row).astype(int)
    fig, plots = plt.subplots(h, images_per_row)
    fig.tight_layout()

    for n in range(0, images_count):
        s = plt.subplot(h, images_per_row, n + 1)
        s.set_xticks(())
        s.set_yticks(())
        plot_digit(images.iloc[n], size_w, size_h, labels)
    plt.show()


plot_digits(data[0:20])
```

Зареждане и анализиране на данните

```
[9]: data = pd.read_csv('data/train.csv')
```

```
[11]: data.tail(10)
```

[11]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pi
41990	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41991	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41992	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41993	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41994	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41996	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41997	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41998	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
41999	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

10 rows × 785 columns

```
[6]: data.shape
```

```
[6]: (42000, 785)
```

```
[8]: data.describe()
```

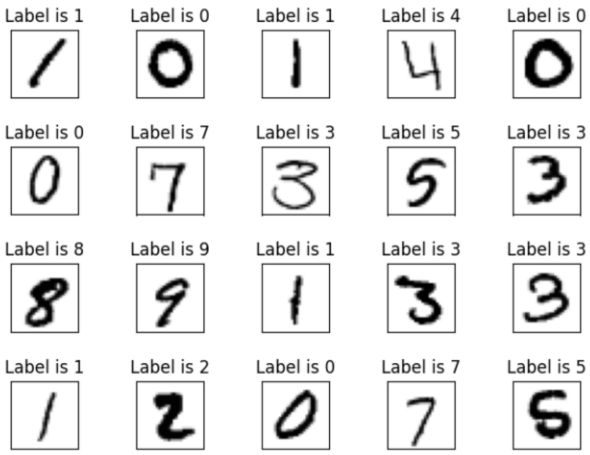
[8]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pix
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	...	42000.000000	42000.000000	42000.000000	42000.00000	42000.0
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.219286	0.117095	0.059024	0.02019	0.0
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	6.312890	4.633819	3.274488	1.75987	1.8
min	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
25%	2.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
50%	4.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
75%	7.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.00000	0.0
max	9.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	254.000000	254.000000	253.000000	253.00000	254.0

8 rows × 785 columns

От последната таблица може да видим за всяка колона колко е средното на данните, стандартното отклонение, максимална стойност, минимална стойност и т.н.

Използвайки помощните функции можем да визуализираме данните:



Разделяме целия dataset на характерситки(features) и етикети(labels)

```
[13]: features = data.drop(['label'], axis='columns', inplace=False)
      features.shape

[13]: (42000, 784)

[14]: labels = data['label']
      labels.shape

[14]: (42000,)

[15]: labels.unique()

[15]: array([1, 0, 4, 7, 3, 5, 8, 9, 2, 6], dtype=int64)

[17]: label_counts = labels.value_counts()
      label_counts

[17]: label
      1    4684
      7    4401
      3    4351
      9    4188
      2    4177
      6    4137
      0    4132
      4    4072
      8    4063
      5    3795
      Name: count, dtype: int64
```

Labels са почти равномерно разпределени и са представени от сходен брой наблюдения. Няма големи неравенства. Можем да използваме метриката за точност (*accuracy*).

Метрики за измерване на производителността на модела

Въпреки че има други начини за измерване на производителността на модела (precision, recall, F1 Score, ROC Curve и др.), ще запазим нещата прости и ще използваме accuracy като наша метрика. Accuracy е подходяща в този случай, защото различните класове са със сходен размер на примерите и изглежда, че са добре балансирани. Също така ще покажем confusion matrix за реални и предсказани етикети и classification report.

Намаляване на размерността

Обикновено намаляването на размерността се използва по три основни причини:

- помага за визуализация на данни с висока размерност и откриване на структури или модели
- избира добри характеристики и избягва висока корелация между някои от тях, което води до следващата причина
- - ефективно обучение на моделите - по-малки размерности и по-малко характеристики означават по-кратки времена за обучение

В този случай ще го използваме, за да намалим броя на характеристиките с цел намаляване на времето за обучение на алгоритмите.

PCA (Principal Component Analysis

PCA (Principal Component Analysis) е линейна техника за намаляване на размерността, която се стреми да максимизира дисперсията. *PCA* е въздействан от представянето на данните, затова трябва да скалираме характеристиките в нашите данни преди да приложим *PCA*. Ще използваме ***MinMaxScaler***. Нашата цел е да запазим голяма част

от дисперсията и в същото време да намалим броя на характеристиките. Когато по-късно оценяваме моделите, можем да се върнем на този етап и да променим баланса между запазената дисперсия и характеристиките. За начало решаваме да запазим дисперсията до **90%** и да видим резултиращото намаление на характеристиките. По-късно ще се върнем на този етап и ще тестваме избраният алгоритъм при различни комбинации.

```
[18]: scaler = MinMaxScaler().fit(features)
      features_scaled = scaler.transform(features)
      pca_features = PCA(0.90).fit(features_scaled)

[19]: features_scaled_pca = pca_features.transform(features_scaled)

[20]: pca_features.n_components_

[20]: 87

[21]: explained_variance = pca_features.explained_variance_ratio_.sum()

[22]: explained_variance

[22]: 0.9005709788011417

      Виждаме, че сме запазили 90% от дисперсията и размерностите са били намалени на 87. Намалихме броя на характеристиките с фактор от 9.
```

Виждаме, че сме запазили 90% от дисперсията и размерностите са били намалени на 87. Намалихме броя на характеристиките с фактор от 9.

Разделяме данните на обучаващи и тестови

Разделяме набора от характеристики, върху който току-що извършихме скалиране и PCA, на 70% обучиаващи и 30% тестови. Разделяме характеристиките едновременно с етикетите.

```
X_tr, X_ts, y_tr, y_ts = train_test_split(features_scaled_pca, labels, test_size = 0.30)
```

```
print("Shape of training features: "+str(X_tr.shape))
print("Shape of testing features: "+str(X_ts.shape))
print("-----------------------------------------")
print("Shape of training labels: "+str(y_tr.shape))
print("Shape of testing labels: "+str(y_ts.shape))
```

```
Shape of training features: (29400, 87)
Shape of testing features: (12600, 87)
-----------------------------------------
Shape of training labels: (29400,)
Shape of testing labels: (12600,)
```

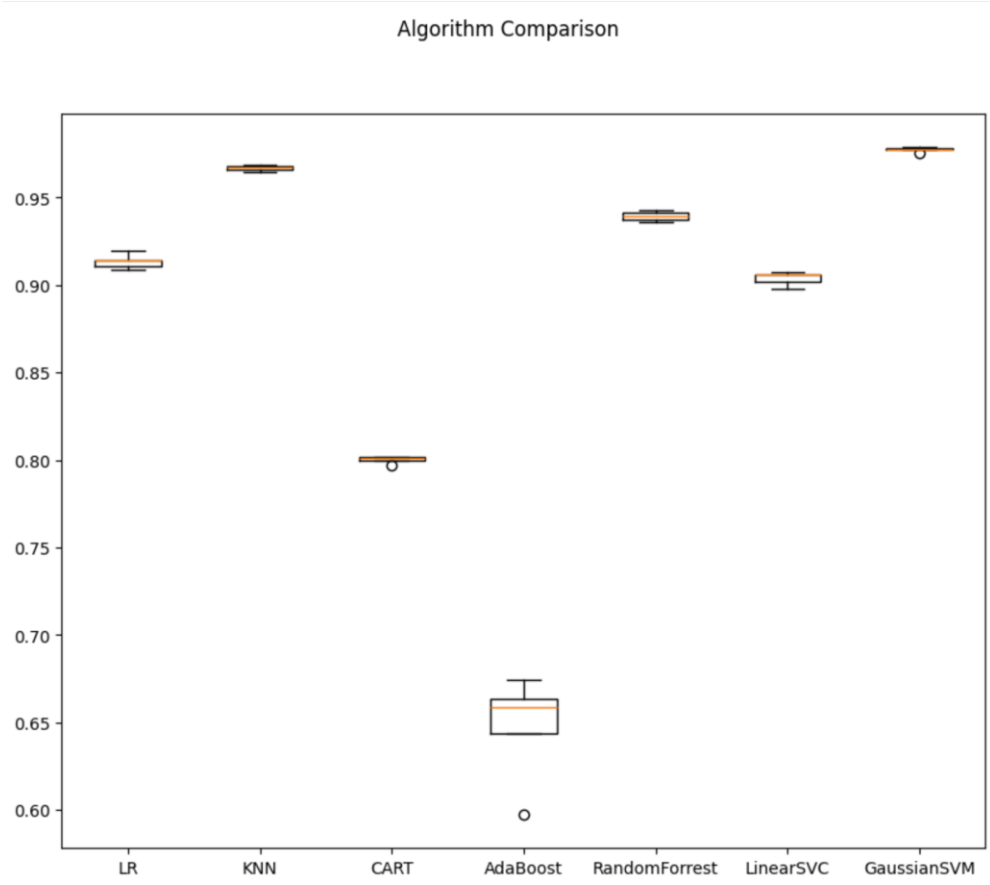
Избиране на алгоритми

Използвайки *cross validation* с 5 фолда върху обучаващия набор с *'accuracy'* като метрика, ще изберем три от най-добрите алгоритъма. Ще ги стартираме с техните настройки по подразбиране. Седемте кандидата са:

1. *Logistic Regression*
2. *K-Nearest Neighbors*
3. *Decision Tree*
4. *AdaBoost*
5. *Random Forest*
6. *Linear SVC*
7. *Guassin SVC*

```
# това отнема около 10 мин
results = []
names = []
scoring = 'accuracy'
algorithms = []
algorithms.append(("LR", LogisticRegression()))
algorithms.append(("KNN", KNeighborsClassifier()))
algorithms.append(("CART", DecisionTreeClassifier()))
algorithms.append(("AdaBoost", AdaBoostClassifier()))
algorithms.append(("RandomForrest", RandomForestClassifier()))
algorithms.append(("LinearSVC", LinearSVC()))
algorithms.append(("GaussianSVM", SVC()))
for name, algorithm in algorithms:
    t0=time()
    cv_results = cross_val_score(algorithm, X_tr, y_tr, cv= k_fold, scoring = scoring)
    t1 = time() - t0
    results.append(cv_results)
    names.append(name)
    print("{0}: mean accuracy {1:.3%} - standard deviation {2:.4} - time {3:n}".format(name, cv_results.mean(), cv_results.std(), t1))
```

Алгоритъм	Средно accuracy
Logistic Regression	91.344%
K-Nearest Neighbors	96.687%
Decision Tree	80.024%
Ada Boost	64.748%
Random Forest	93.939 %
Linear SVC	90.384%
Gauassian SVM	97.759%



Виждаме, че 3-те най-добри алгоритъма са *KNN, Random Forest u GauassianSVM*. Понеже *Random Forest* отенаме повече време ще изберем *Ligostic Regression* вместо Random Forest.

Оконачтелно избираме KNN, GauassianSVM и Logistic Regression моделите за допълнителни подобрения и тестване на различни хипермараметри.

Logistic Regression

[29]:
```
# това е списъка, където ще пазим подобрените модели
models = []
```

[30]:
```
model = LogisticRegression()
parameters_grid = [{
    "C": [ 1, 10, 100, 1000, 10000],
    "solver": ['lbfgs','liblinear']
}]
model_fit = fit_model(model, parameters_grid, k_fold)
```

[31]:
```
# разкоментирай долния ред код, ако не искаш да чакаш тренирането на LogisticRegression модела
model_fit = load("LogisticRegression.sav")
```

[32]:
```
model_fit
```

[32]:
```
{'model': 'LogisticRegression',
 'number_of_observations_training': 29400,
 'number_features_training': 87,
 'explained_variance_after_pca': 0.9005709788011417,
 'best_estimator': LogisticRegression(C=100),
 'best_params': {'C': 100, 'solver': 'lbfgs'},
 'best_score': 0.9143197278911565,
 'time_sec': 263.3169934749603,
 'test_score': 0.9242857142857143}
```

[33]:
```
print("Mean cross-validated score of the best_estimator {0:.3%}".format(model_fit["best_score"]))
```

```
Mean cross-validated score of the best_estimator 91.432%
```

Добре, оценката от крос валидацията не се повиши толкова много. Сега ще видим резултата на модела с тестовия набор от данни.

[34]:
```
best_model = model_fit['best_estimator']
score = best_model.score(X_ts, y_ts)
print("Test score {0:.3%}".format(score))
```

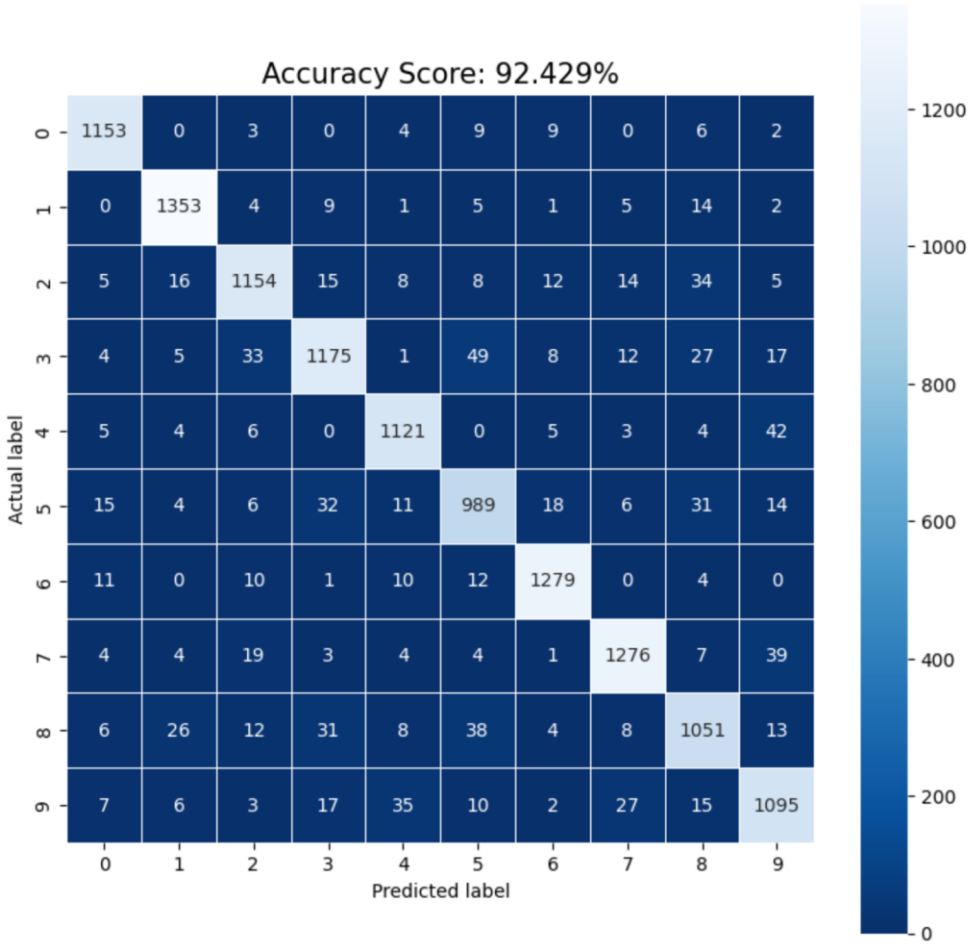
```
Test score 92.429%
```

Тестовата оценка е добра върху данни, които никога преди не са били видяни, което означава, че моделът обобщава добре. Ще добавим тестовата оценка към нашия речник с информация за моделите, където съхраняваме информацията за моделите.

[35]:
```
model_fit["test_score"] = score
```

Да разгледаме confusion matrix, за да видим каква е класификацията.

[36]:
```
predictions = best_model.predict(X_ts)
draw_confusion_marix(y_ts, predictions, score)
```



[37]:
```
print(classification_report(y_ts, predictions))
```

```
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      1186
           1       0.95      0.97      0.96      1394
           2       0.92      0.91      0.92      1271
           3       0.92      0.88      0.90      1331
           4       0.93      0.94      0.94      1190
           5       0.88      0.88      0.88      1126
           6       0.96      0.96      0.96      1327
           7       0.94      0.94      0.94      1361
           8       0.88      0.88      0.88      1197
           9       0.89      0.90      0.90      1217

    accuracy                           0.92     12600
   macro avg       0.92      0.92      0.92     12600
weighted avg       0.92      0.92      0.92     12600
```

[38]:
```
# запазване на модела в текущата директория и запазване на ексел файл с параметрите на модела
persist_data(models, model_fit)
```

Gaussian SVM

```
[39]: model = SVC(cache_size=1000)
parameters_grid = [{
    "C": [5, 10, 20],
    "gamma": [0.01, 0.05, 0.1]
}]
model_fit = fit_model(model, parameters_grid, k_fold)
```

```
[40]: # зареди този файла ако нямаш време да тренираш модел
# model_fit = load("SVCP.sav")
```

```
[41]: model_fit
```

```
[41]: {'model': 'SVC',
 'number_of_observations_training': 29400,
 'number_features_training': 87,
 'explained_variance_after_pca': 0.9005709788011417,
 'best_estimator': SVC(C=5, cache_size=1000, gamma=0.05),
 'best_params': {'C': 5, 'gamma': 0.05},
 'best_score': 0.981326530612245,
 'time_sec': 2749.29559135437}
```

```
[42]: print("Mean cross-validated score of the best_estimator {0:.3%}".format(model_fit["best_score"]))
```

```
Mean cross-validated score of the best_estimator 98.133%
```

Отнема доста повече време да се направи grid search и да се обучи модела. Най-добрият валидационен резултат се е подобрил спрямо резултата от cross validation в частта за избор на модел.

Сега нека да оценим модел с тестовите данни:

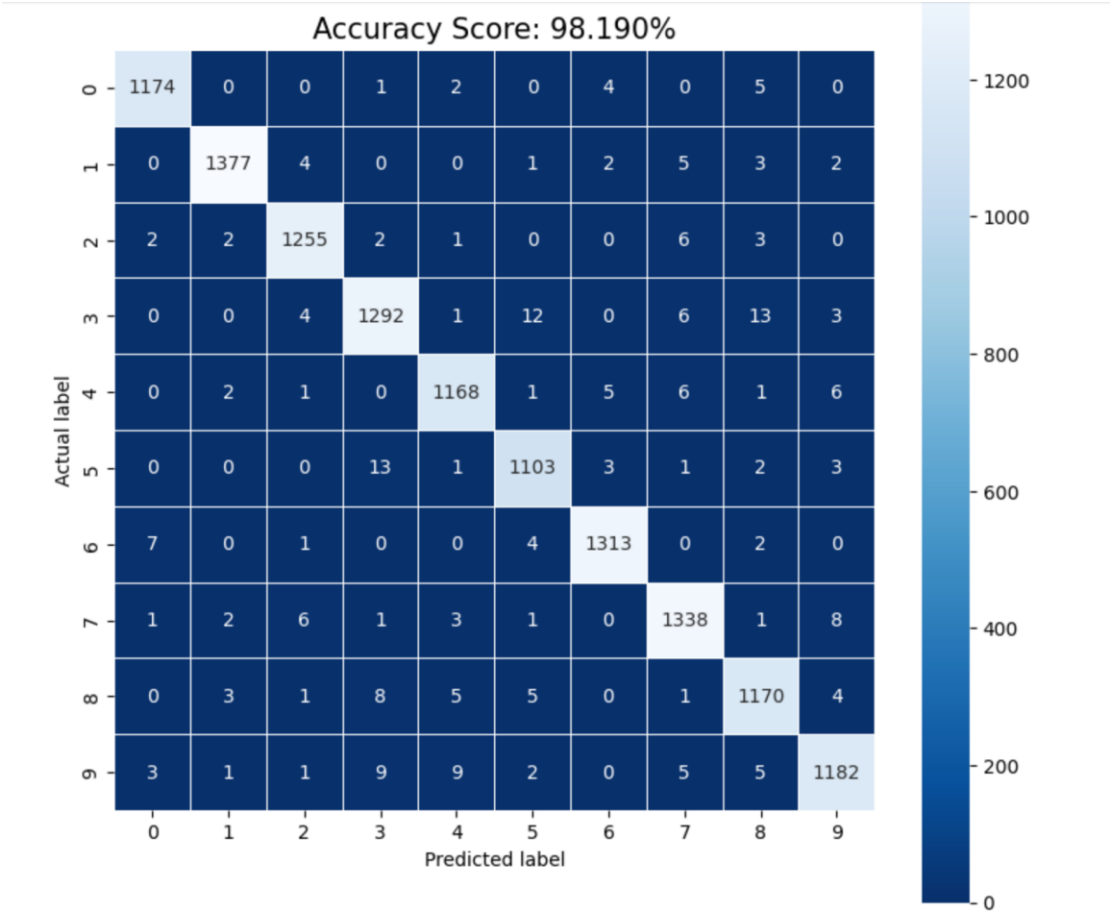
```
[43]: best_model = model_fit['best_estimator']
score = best_model.score(X_ts, y_ts)
print("Test score {0:.3%}".format(score))
```

```
Test score 98.190%
```

```
[44]: model_fit["test_score"] = score
```

Нека анализираме confusion matrix

```
[45]: predictions = best_model.predict(X_ts)
draw_confusion_marix(y_ts, predictions, score)
```



[46]:
```
print(classification_report(y_ts, predictions))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1186
           1       0.99      0.99      0.99      1394
           2       0.99      0.99      0.99      1271
           3       0.97      0.97      0.97      1331
           4       0.98      0.98      0.98      1190
           5       0.98      0.98      0.98      1126
           6       0.99      0.99      0.99      1327
           7       0.98      0.98      0.98      1361
           8       0.97      0.98      0.97      1197
           9       0.98      0.97      0.97      1217

    accuracy                           0.98     12600
   macro avg       0.98      0.98      0.98     12600
weighted avg       0.98      0.98      0.98     12600
```

[47]:
```
# запазваме модела в текущата директория и запазване на excel file с най-добрите параметри
persist_data(models, model_fit)
```

K-Nearest Neighbors

Сега ще опитаме да подобрим K-Nearest Neighbors алгоритъма:

[48]:
```
model = KNeighborsClassifier()
k = np.arange(3,10,2)
print(k)

parameters_grid = [{
    'n_neighbors': k
}]

model_fit = fit_model(model, parameters_grid, k_fold)
```

```
[3 5 7 9]
```

[49]:
```
# разкоментирай реда по-долу ако нямаш време да чакаш да се тренира модела
# model_fit = load("KNeighborsClassifier.sav")
```

[50]:
```
model_fit
```

[50]:
```
{'model': 'KNeighborsClassifier',
 'number_of_observations_training': 29400,
 'number_features_training': 87,
 'explained_variance_after_pca': 0.9005709788011417,
 'best_estimator': KNeighborsClassifier(n_neighbors=3),
 'best_params': {'n_neighbors': 3},
 'best_score': 0.9672789115646256,
 'time_sec': 17.840768098831177}
```

[51]:
```
print("Mean cross-validated score of the best_estimator {0:.3%}".format(model_fit["best_score"]))
```

```
Mean cross-validated score of the best_estimator 96.728%
```

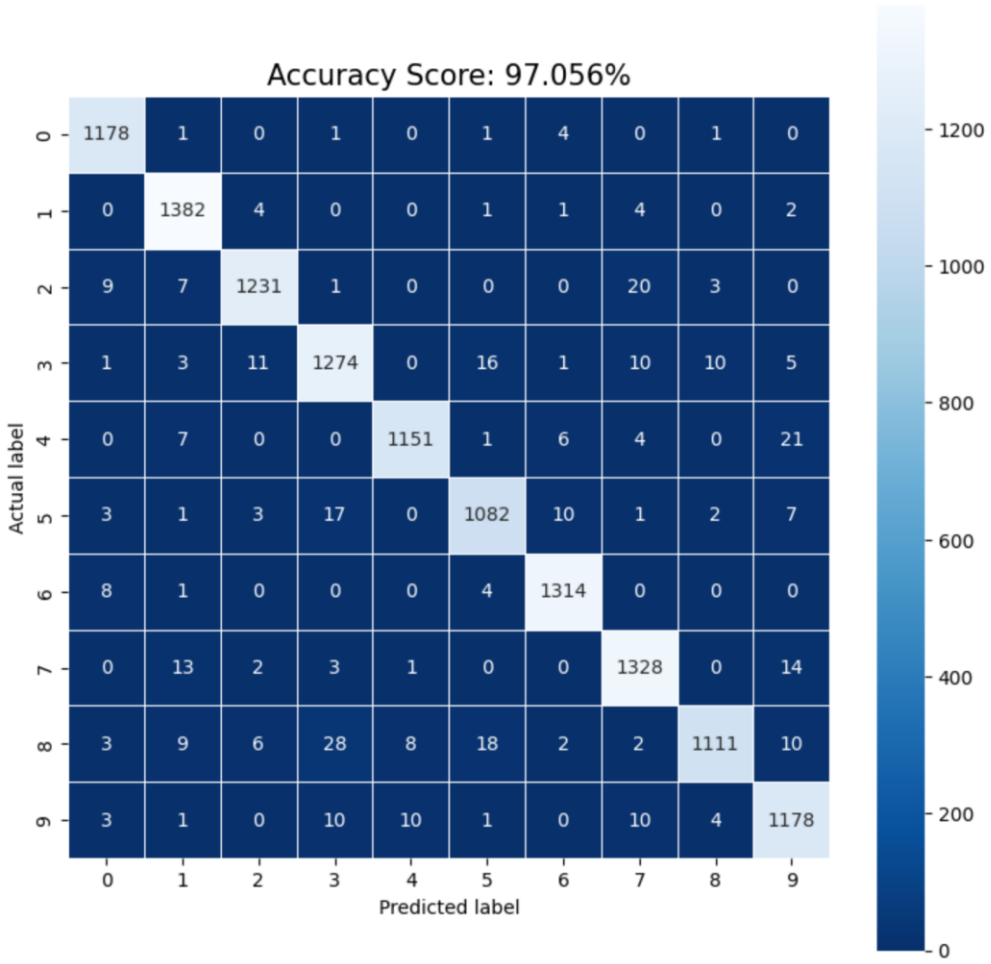
Сега нека да оценим как се справя модела със тестовите данни:

[52]:
```
best_model = model_fit['best_estimator']
score = best_model.score(X_ts, y_ts)
print("Test score {0:.3%}".format(score))
```

```
Test score 97.056%
```

[53]:
```
model_fit["test_score"] = score
```

[54]:
```
predictions = best_model.predict(X_ts)
draw_confusion_marix(y_ts, predictions, score)
```



[55]:
```
print(classification_report(y_ts, predictions))
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      1186
           1       0.97      0.99      0.98      1394
           2       0.98      0.97      0.97      1271
           3       0.96      0.96      0.96      1331
           4       0.98      0.97      0.98      1190
           5       0.96      0.96      0.96      1126
           6       0.98      0.99      0.99      1327
           7       0.96      0.98      0.97      1361
           8       0.98      0.93      0.95      1197
           9       0.95      0.97      0.96      1217

    accuracy                           0.97     12600
   macro avg       0.97      0.97      0.97     12600
weighted avg       0.97      0.97      0.97     12600
```

[56]:
```
# запазваме модела в текущата директория и запаваме най-добрите параметри в Excel file
persist_data(models, model_fit)
```

Сравнение на най-добрите модели

[57]:
```
print("SUMMARY:")
print("-------------")
for model in models:
    print("{0} - test score {1:.2%} - features {2} - observations {3} - grid search time {4:n}s"
          .format(model["model"], model["test_score"], model["number_features_training"], model["number_of_observations_training"], model["time_sec"]))
```

```
SUMMARY:
-------------
LogisticRegression - test score 92.43% - features 87 - observations 29400 - grid search time 263.317s
SVC - test score 98.19% - features 87 - observations 29400 - grid search time 2749.3s
KNeighborsClassifier - test score 97.06% - features 87 - observations 29400 - grid search time 17.8408s
```

Най-добрият модел за нашия набор от данни е *Gaussian SVM* с резултат от тестовия набор от 98%, но се наложи да се обучава доста повече време. От друга страна, методът на k-най-близките съседи (*K-Nearest Neighbors*) постигна много

близък резултат от 97%, но се научи доста по-бързо. Логистичната регресия е значително отстъпила. Ще потърсим допълнителни подобрения на моделите за k-най-близките съседи и *Gaussian SVM*

Допълнителни подобрения със селекция на характереситики

Сега, след като сме избрали най-добрите модели и сме извършили *grid search*, можем да потърсим други начини за подобрение на моделите ни. Ще се върнем към стъпката с *PCA* и ще опитаме да подобрим моделите чрез *grid search* с различен брой признаци.

Ние произволно избрахме да запазим 90% от дисперсията, като така получаваме **87** признака. Сега можем да тестваме моделите *SVM* и *KNN* с по-малко или малко повече признаци.

Например, можем да изпълним *PCA* с цел *explained variance* от 60%, 70%, 80% и 95%. Така ще получим 5 набора от признаци. Повторно можем да изпълним *grid search* за *SVM* и *KNN* и да оценим техните резултати.

[58]:
```
k = np.arange(3, 8, 2)
# колекция с подобрените модели
models_features = []
model = None
# explained variance нива
expl_vars = [0.60, 0.70, 0.80, 0.90, 0.95]

# колекция с моделите, които ще тестваме
algorithms = []

algorithms.append((KNeighborsClassifier(), [{
    'n_neighbors': k
}]))

# отнема много време за трениране (разкоментирай, ако имаш повече време за трениране)
# algorithms.append((SVC(),[{
#     "C": [5, 10, 20],
#     "gamma": [0.01, 0.05, 0.1]
# }]))
```

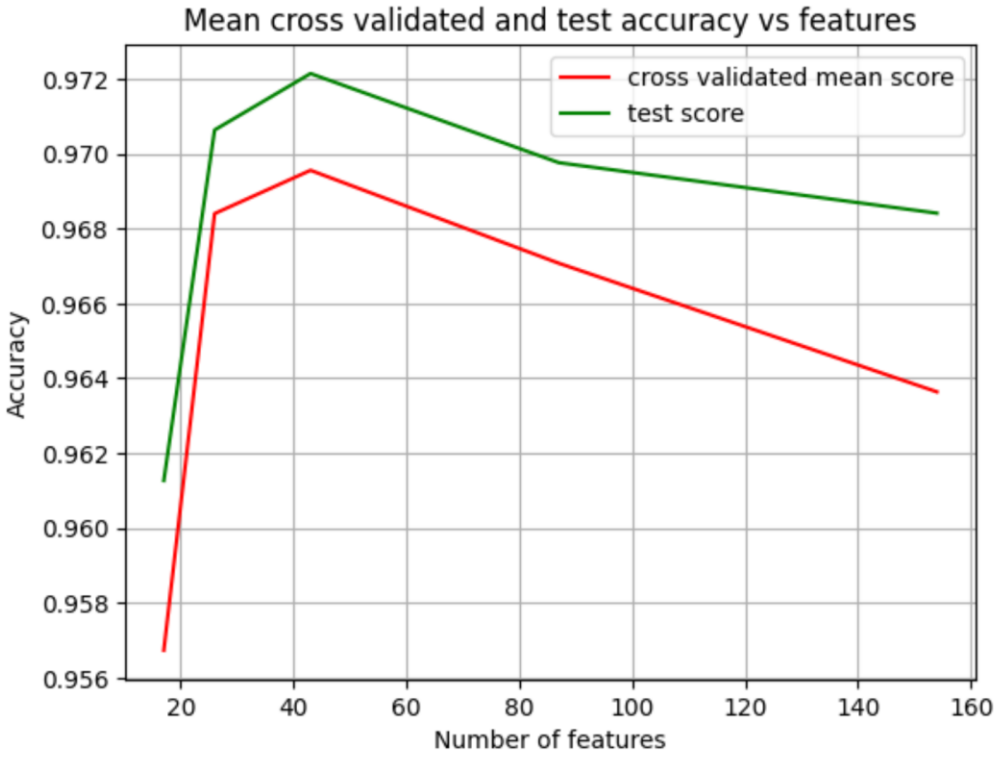
[59]:
```
algorithms
```

[59]:
```
[(KNeighborsClassifier(), [{'n_neighbors': array([3, 5, 7])}])]
```

*Направихме тест само с KNN понеже SVM отнема много време за тестваме

```
[60]: # Изпълняваме PCA за да изберем различен вид features

for variance in expl_vars:
    print("Performing PCA ...")

    pca_features = PCA(variance).fit(features_scaled)

    features_scaled_pca = pca_features.transform(features_scaled)

    explained_variance = pca_features.explained_variance_ratio_.sum()

    components = pca_features.n_components_

    print("Explained variance {0:.2%} with {1} components".format(explained_variance, components))

    print("Splitting training and test set...")

    X_tr, X_ts, y_tr, y_ts = train_test_split(features_scaled_pca, labels, test_size = 0.30)

    print()

    for algorithm, parameters_grid in algorithms:

        print("Performing Grid Search on: {0} with {1} features and explained variance {2:.1%}".format(algorithm.__class__.__name__, components,  explai

        model = algorithm

        model = fit_model(model, parameters_grid, k_fold)

        best_model = model['best_estimator']

        score = best_model.score(X_ts, y_ts)

        model["test_score"] = score

        print("Ready")

        print("{0} - best score {1:.2%} - test score {2:.2%} - grid search time {3:n}s"
          .format(model["model"], model["best_score"], model["test_score"], model["time_sec"]))

        models_features.append(model)

        print()
dump(models_features, "models_features.sav")
```

Брой компоненти	Резултат обучаващи данни	Резултат тестови данни
60% - 17 компонента	95.67%	96.13%
70% - 26 компонента	96.84%	97.06%
80% - 43 компонента	96.06%	97.21%
95% - 154 компонента	96.36%	96.84%
90% - 87 компонента	96.71%	96.98%



Виждаме, че няма смисъл да увеличаваме броя на признаците над 26 за модела на *KNN*. Ако запазим 26 признака, което съответства на *explained variance* от 70%, успяваме да постигнем точност от 97% при тестовите изпитания. Увеличаването на броя на признаците над 26 не донесе допълнителни ползи и може да доведе до прекомерно адаптиране (*overfitting*).

5 Заключение

Използвахме различни алгоритми за машинно обучение и методи за намаляване на размерността на данните. *Gaussian SVM* и *K-Nearest Neighbors* постигнаха добри резултати в класификацията на ръчно написаните цифри, с точност около **98%.**

Използването на метода на *PCA* се яви като полезен подход за намаляване на размерността на данните и ускоряване на обучението на моделите. Също така, моделите се проявиха добре при тестване на тестовия набор от данни, което свидетелства за добро обобщение (***моест постигнахме low bias и low variance***)

За бъдещи подобрения и експерименти, може да се разгледа включването на по-сложни модели, като например невронни мрежи, които могат да се справят по-добре със сложни структури и зависимости в данните. Освен това, допълнителна оптимизацията на хиперпараметрите могат да доведат до допълнително подобрение на производителността на моделите.

6 Използвана литература

- https://www.kaggle.com/c/digit-recognizer/data
- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
- https://en.wikipedia.org/wiki/Support_vector_machine
- https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- https://scikit-learn.org/stable/modules/model_evaluation.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- https://scikit-learn.org/stable/modules/cross_validation.html