Libraries

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import pandas as pd
import warnings
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDisc
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, accuracy_score, confusion_matrix, precis
import pickle as pkl
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

my_seed = 56
```

Tools

In [ ]:
```
def get_save_estimator(estimator, train_score, test_score, name, save=True, folder=
    if save:
        with open(f"{folder}/"+name+"_best_estimator.pkl", 'wb') as f:
            pkl.dump(estimator, f)
        with open(f"{folder}/"+name+"_best_train_score.pkl", 'wb') as f:
            pkl.dump(train_score, f)
        with open(f"{folder}/"+name+"_best_test_score.pkl", 'wb') as f:
            pkl.dump(test_score, f)
    else:
        with open(f"{folder}/"+name+"_best_estimator.pkl", 'rb') as f:
            estimator = pkl.load(f)
        with open(f"{folder}/"+name+"_best_train_score.pkl", 'rb') as f:
            train_score = pkl.load(f)
        with open(f"{folder}/"+name+"_best_test_score.pkl", 'rb') as f:
            test_score = pkl.load(f)

    return estimator, train_score, test_score
```

In [ ]:
```
def get_acc_f1_prec_rec(y_true, y_pred):
    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, average='micro')
    prec = precision_score(y_true, y_pred, average='micro')
```

```
    rec = recall_score(y_true, y_pred, average='micro')
    return acc, f1, prec, rec
```

1. Dataset Description

1.3. Loading the data

1.3.1. Load the data with np.load("filename.npz") and store the different matrices in memory (for instance in variables x1,y1 for Pima data and x2,y2,xt2,yt2 for digits data). For the digits dataset,it is better to perform one simple pre-processing that scales the values between [0, 1] by dividing the data matrix by 255

For Pima Dataset :

In [ ]:
```
pima_data = np.load("pima.npz")
pima_data.files
```

Out[ ]:
```
['xall', 'yall', 'varnames']
```

In [ ]:
```
x1 = pima_data['xall']
y1 = pima_data['yall']
pima_vars = pima_data['varnames']

print("X shape:", x1.shape)
print("Y shape:", y1.shape)
print("Variables:", *pima_vars)
```

```
X shape: (709, 8)
Y shape: (709,)
Variables: Pregnancies Glucose BloodPressure SkinThickness Insulin BMI DiabetesPedig
reeFunction Age
```

For Digits Dataset :

In [ ]:
```
digits_data = np.load("digits.npz")
digits_data.files
```

Out[ ]:
```
['xt', 'yt', 'y', 'x']
```

In [ ]:
```
xt2 = digits_data['xt']
yt2 = digits_data['yt']
x2 = digits_data['x']
y2 = digits_data['y']

print("X train shape:", x2.shape, "| Y train shape:", y2.shape)
print("X test shape:", xt2.shape, "| Y test shape:", yt2.shape)
```

```
X train shape: (3000, 784) | Y train shape: (3000, 1)
X test shape: (1500, 784) | Y test shape: (1500, 1)
```

We rescale pixel values between 0 and 1 by dividing them by 255.0. We do this for both training and testing data.

In [ ]:
```
x2 = x2/255
xt2 = xt2/255
```

1.3.2. Do a quick look at the data, compute the mean values for each variable and interpret it.

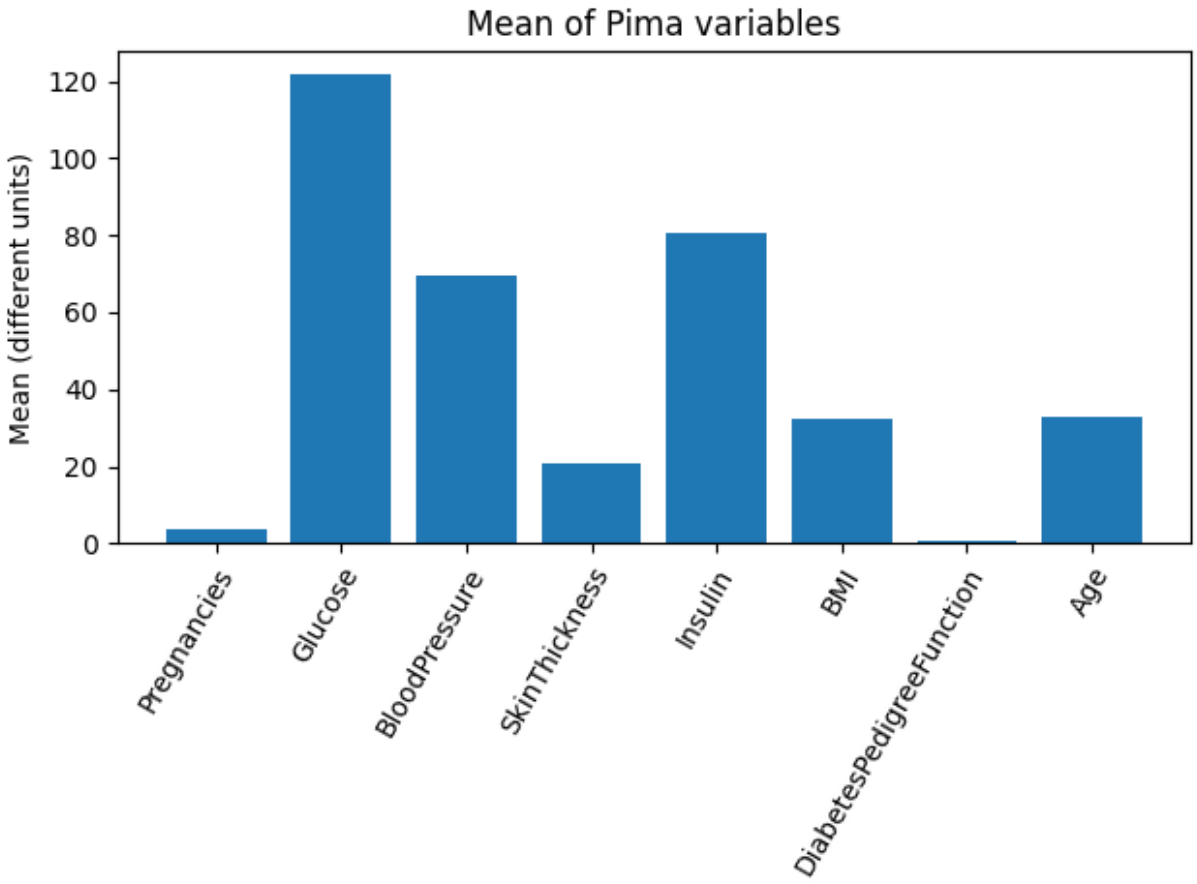
For Pima Dataset :

In [ ]:
```
pd.DataFrame(x1, columns=pima_vars).describe().round(2)
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigr
count	709.00	709.00	709.00	709.00	709.00	709.00	
mean	3.75	121.61	69.64	20.77	80.53	32.45	
std	3.34	30.49	18.14	15.90	112.68	6.95	
min	0.00	44.00	0.00	0.00	0.00	18.20	
25%	1.00	99.00	64.00	0.00	0.00	27.50	
50%	3.00	117.00	72.00	23.00	40.00	32.30	
75%	6.00	141.00	80.00	32.00	130.00	36.60	
max	17.00	199.00	122.00	99.00	744.00	67.10	

In [ ]:
```
plt.bar(np.arange(8), np.mean(x1, axis=0))
plt.xticks(np.arange(8), pima_vars, rotation=60, ha='right', rotation_mode='anchor'
plt.ylabel("Mean (different units)")
plt.title("Mean of Pima variables")
plt.tight_layout()
```



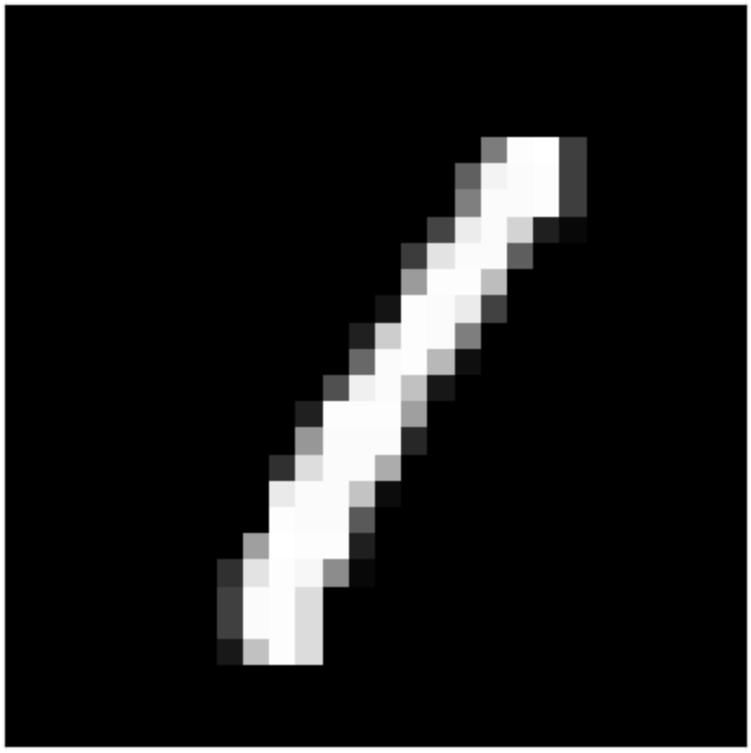
In [ ]:
```
print("Pima Labels:", np.unique(y1))
```

```
Pima Labels: [-1  1]
```

For Digits Dataset :

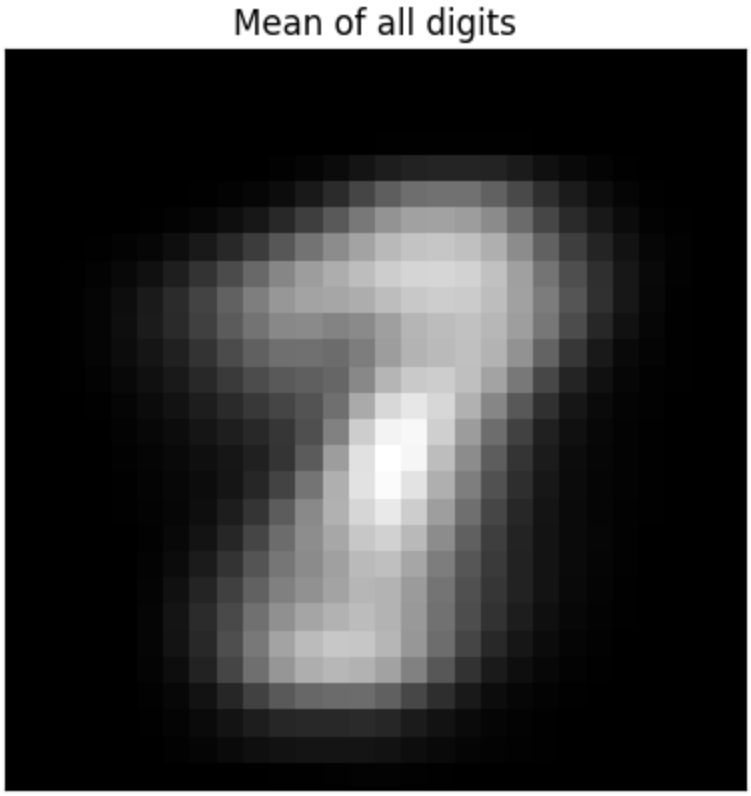
Example of a digit :

In [ ]:
```
fig = plt.imshow(x2[0].reshape(28, 28), cmap='gray')
fig.axes.get_xaxis().set_visible(False)
fig.axes.get_yaxis().set_visible(False)
plt.show()
```



Now, let's visualize the mean values for each variable (i.e. each pixel) to obtain a mean digit :

In [ ]:
```
fig = plt.imshow(x2.mean(axis=0).reshape(28, 28), cmap='gray')
fig.axes.get_xaxis().set_visible(False)
fig.axes.get_yaxis().set_visible(False)
plt.title("Mean of all digits")
plt.show()
```



In [ ]:
```
print("All different digits Labels:", np.unique(y2))
```

```
All different digits Labels: [1 7 8]
```

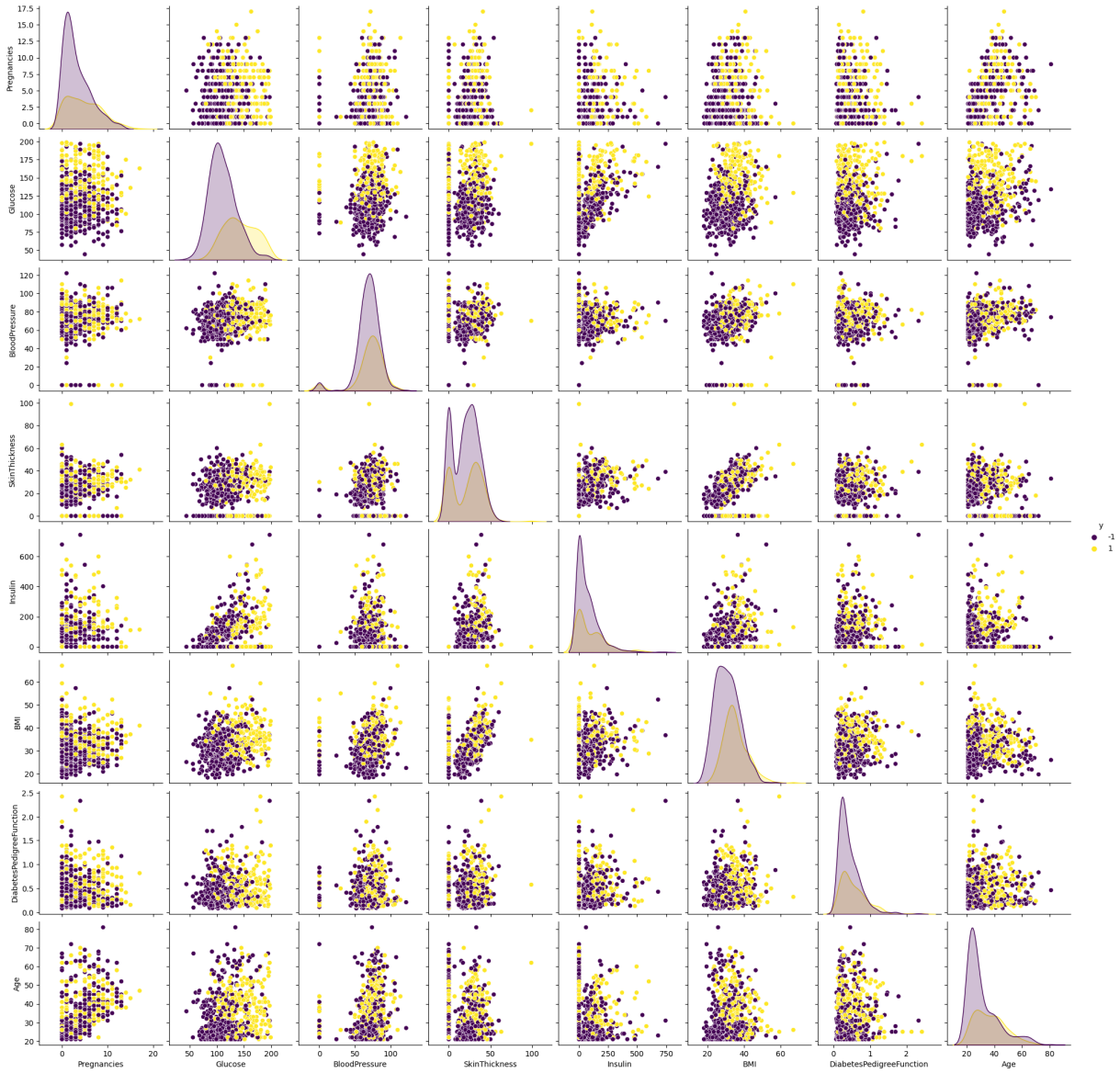
2. Predicting Diabetes on Pima Dataset

2.1. Know the data

2.1.1. Visualize the data as scatterplots between pairs of variables (where the color is the class)

In [ ]:
```
df1 = pd.concat((pd.DataFrame(x1, columns=pima_vars), pd.DataFrame(y1, columns=["y"
```

In [ ]:
```
with warnings.catch_warnings():
    warnings.simplefilter('ignore')
    sns.pairplot(df1, hue="y", palette="viridis")
    plt.show()
```



2.1.2. What are the variables that seem to help predict the class? Do those variable make sense from a medical perspective ?

From visualizing variable combinations, we notice that some variables are more important than others to predict the class. For instance, if we consider all combinations of glucose feature with other variables, we notice that the two classes (non-diabetic and diabetic) are very well separated meaning that glucose is a very important feature to predict the class. On the other hand, considering the Skin Thickness feature with respect to other variables, the two classes are not separated in a satisfactory manner. This means that this feature is not as important to predict the class. Now that we have the intuition, let's classify variables in a naive way.

Most Important variables (naive approach):

- Glucose
- Insulin
- BMI

- DiabetesPedigreeFunction

Less Important variables (naive approach):

- Thickness of Skin

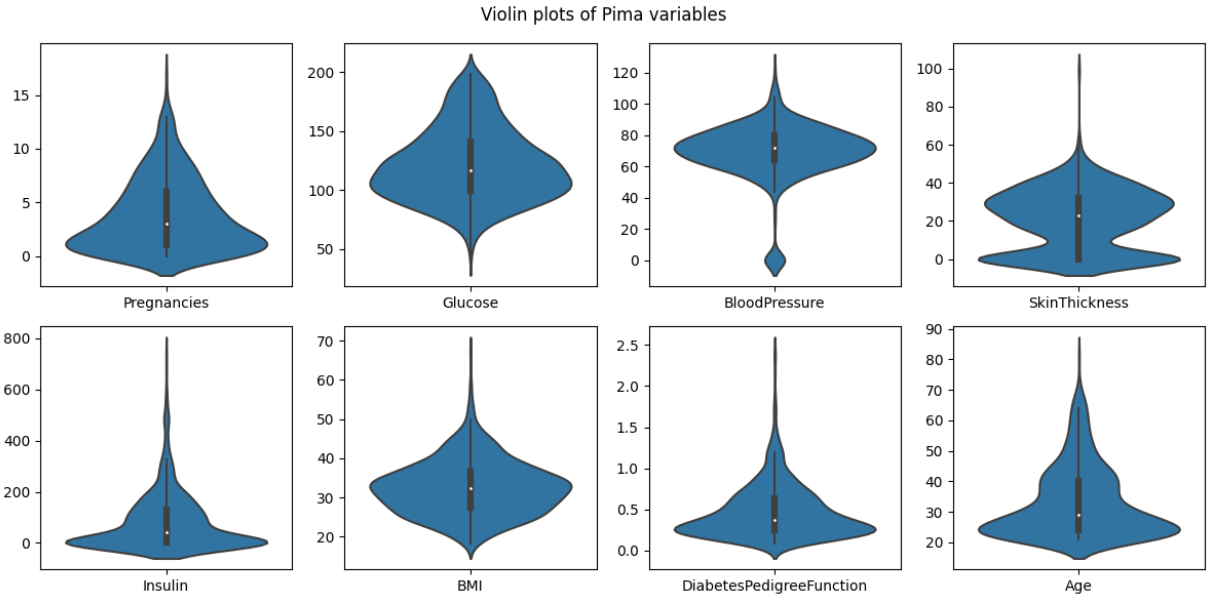
From a medical perspective, this in sensible because glucose and insulin are the main indicators of diabetes. As BMI measures body fat, it also makes sense that it is a good predictor of the class. Finally, DiabetesPedigreeFunction is a function indicating diabetes likelihood depending on the subject's age and his/her diabetic family history making it a relevant feature.

2.1.3. Split the data in training/test by keeping n = 300 samples for training/validation and the remaining for test

In [ ]:
```
x1_train, x1_test, y1_train, y1_test = train_test_split(x1, y1, train_size=300, ran
```

2.1.4. Do the feature have similar variances/scaling? Is that a problem ?

In [ ]:
```
f, axes = plt.subplots(2, 4, figsize=(12, 6))
for i, var in enumerate(pima_vars):
    warnings.simplefilter('ignore')
    sns.violinplot(df1[var], ax=axes[i//4, i%4])
    axes[i//4, i%4].set_xticklabels(labels=[var])
f.suptitle("Violin plots of Pima variables")
plt.tight_layout()
```



From the plots above, we notice very different variances between features. We can confirm this by looking at numerical values of variances for each feature :

In [ ]:
```
pd.DataFrame(x1, columns=pima_vars).describe().round(2).loc["std"]
```

Out[ ]:
```
Pregnancies                 3.34
Glucose                    30.49
BloodPressure              18.14
SkinThickness              15.90
Insulin                   112.68
BMI                         6.95
DiabetesPedigreeFunction    0.33
Age                        11.67
Name: std, dtype: float64
```

2.1.5. Apply a standardization to the training and test data (StandardScaler).

In [ ]:
```
std_scaler = StandardScaler()
x1_train = std_scaler.fit_transform(x1_train)
x1_test = std_scaler.transform(x1_test)
```

In [ ]:
```
pd.DataFrame(x1_train, columns=pima_vars).describe().round(2).loc["std"]
```

Out[ ]:
```
Pregnancies                 1.0
Glucose                     1.0
BloodPressure               1.0
SkinThickness               1.0
Insulin                     1.0
BMI                         1.0
DiabetesPedigreeFunction    1.0
Age                         1.0
Name: std, dtype: float64
```

Our variables are now scaled and have similar variances.

2.2. Bayesian Descision and linear classification

2.2.1. Train a Linear Discriminant Analysis (LDA) classifier with the default parameters, compute its accuracy and AUC on the test data (LinearDiscriminantAnalysis,roc_auc_score). Note that in order to compute the AUC you will need to compute the score with est.predict_proba and keep the second column (probability of the class 1).

In [ ]:
```
lda = LinearDiscriminantAnalysis(solver="lsqr")
lda.fit(x1_train, y1_train)
pred = lda.predict(x1_test)
pred_prob = lda.predict_proba(x1_test)

print("Test ROC score:", round(roc_auc_score(y1_test, pred_prob[:, 1]), 2), "| Test
```

```
Test ROC score: 0.84 | Test Accuracy score: 0.78
```

2.2.2. Perform a cross validation GridSearchCV) for the parameters of the method (shrinkage). Does the validation leads to better performance? What is the optimal value for the parameter?

In [ ]:
```
params = {
    "shrinkage": np.linspace(0, 1, 50)
}

gs = GridSearchCV(lda, params, scoring="roc_auc")
res = gs.fit(x1_train, y1_train)
```

In [ ]:
```
lda = gs.best_estimator_
```

In [ ]:
```
pred = lda.predict(x1_test)
pred_prob = lda.predict_proba(x1_test)

lda_acc_train = accuracy_score(y1_train, lda.predict(x1_train))
lda_roc_train = roc_auc_score(y1_train, lda.predict_proba(x1_train)[:, 1])
lda_acc_test= accuracy_score(y1_test, pred)
lda_roc_test = roc_auc_score(y1_test, pred_prob[:, 1])

print("Train ROC score:", round(lda_roc_train, 2), "| Train Accuracy score:",  roun
print("Test ROC score:", round(lda_roc_test, 2), "| Test Accuracy score:",  round(l
```

```
Train ROC score: 0.86 | Train Accuracy score: 0.8
Test ROC score: 0.85 | Test Accuracy score: 0.77
```

Performances haven't changed significantly. Roc have increased of 1% while accurancy decreased of 1%. This makes sense because we are using roc as performance metric for the grid search cross validation.

2.2.3. Train a Quadratic Discriminant Analysis (QDA) classifier with the default parameters, compute its accuracy and AUC on the test data (QuadraticDiscriminantAnalysis). Is the performance better than LDA?

In [ ]:
```
qda = QuadraticDiscriminantAnalysis()
qda.fit(x1_train, y1_train)
pred = qda.predict(x1_test)
pred_prob = qda.predict_proba(x1_test)

print("Test ROC score:", round(roc_auc_score(y1_test, pred_prob[:, 1]), 2), "| Test
```

```
Test ROC score: 0.84 | Test Accuracy score: 0.76
```

Performances are slightly worse than LDA for accuracy (-2%) and equivalent for Roc.

2.2.4. Perform a cross validation for the parameters of the method (reg_param). Does the validation leads to better performance? What is the optimal value for the parameter.

In [ ]:
```
params = {
    "reg_param": np.linspace(0, 1, 50)
}

gs = GridSearchCV(qda, params)
res = gs.fit(x1_train, y1_train)
```

In [ ]:
```
qda = gs.best_estimator_
```

In [ ]:
```
pred = qda.predict(x1_test)
pred_prob = qda.predict_proba(x1_test)

qda_acc_train = accuracy_score(y1_train, qda.predict(x1_train))
qda_roc_train = roc_auc_score(y1_train, qda.predict_proba(x1_train)[:, 1])
qda_acc_test= accuracy_score(y1_test, pred)
qda_roc_test = roc_auc_score(y1_test, pred_prob[:, 1])

print("Train ROC score:", round(qda_roc_train, 2), "| Train Accuracy score:",  roun
print("Test ROC score:", round(qda_roc_test, 2), "| Test Accuracy score:",  round(q
```

```
Train ROC score: 0.85 | Train Accuracy score: 0.79
Test ROC score: 0.85 | Test Accuracy score: 0.78
```

Here, Roc and accuracy have increased of 1% compared to non-cross-validated QDA and accuracy has increase of 1% compared to fine-tuned LDA. This is a non-negligible improvement.

2.2.5. Train a Gaussian Naive Bayes (NB) classifier (GaussianNB). What is its performance with respect to QDA and LDA?

In [ ]:
```
gnb = GaussianNB()
gnb.fit(x1_train, y1_train)

pred = gnb.predict(x1_test)
pred_prob = gnb.predict_proba(x1_test)

gnb_acc_train = accuracy_score(y1_train, gnb.predict(x1_train))
gnb_roc_train = roc_auc_score(y1_train, gnb.predict_proba(x1_train)[:, 1])
gnb_acc_test = accuracy_score(y1_test, pred)
gnb_roc_test = roc_auc_score(y1_test, pred_prob[:, 1])

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, qda_acc_train, gnb_acc_train],
    "Test Accuracy": [lda_acc_test, qda_acc_test, gnb_acc_test],
    "Train ROC": [lda_roc_train, qda_roc_train, gnb_roc_train],
    "Test ROC": [lda_roc_test, qda_roc_test, gnb_roc_test]
}, index=["LDA", "QDA", "GNB"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LDA	0.80	0.77	0.86	0.85
QDA	0.79	0.78	0.85	0.85
GNB	0.78	0.77	0.83	0.84

Its performances are slightly worse than LDA and QDA for both accuracy and Roc. However, we should take into account that LDA and QDA have been fine-tuned while NB has not.

2.2.6. Train a Logistic regression classifier (LogisticRegression) with the default parameters. Compute its performance and compare it to the previous classifiers.

In [ ]:
```
logreg = LogisticRegression()
logreg.fit(x1_train, y1_train)

pred = logreg.predict(x1_test)
pred_prob = logreg.predict_proba(x1_test)

logreg_acc_train = accuracy_score(y1_train, logreg.predict(x1_train))
logreg_roc_train = roc_auc_score(y1_train, logreg.predict_proba(x1_train)[:, 1])
logreg_acc_test = accuracy_score(y1_test, pred)
logreg_roc_test = roc_auc_score(y1_test, pred_prob[:, 1])

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, qda_acc_train, gnb_acc_train, logreg_acc_trai
    "Test Accuracy": [lda_acc_test, qda_acc_test, gnb_acc_test, logreg_acc_test],
    "Train ROC": [lda_roc_train, qda_roc_train, gnb_roc_train, logreg_roc_train],
    "Test ROC": [lda_roc_test, qda_roc_test, gnb_roc_test, logreg_roc_test]
}, index=["LDA", "QDA", "GNB", "LogReg"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LDA	0.80	0.77	0.86	0.85
QDA	0.79	0.78	0.85	0.85
GNB	0.78	0.77	0.83	0.84
LogReg	0.80	0.79	0.86	0.84

Logistic Regression (not fine-tuned) performs better than LDA, QDA and NB for both accuracy and Roc. In addition, it doesn't overfit the training data as much as the other classifiers such as LDA for instance (at least it looks like it starts overfitting).

2.2.7. Perform a a cross validation for the parameters of the model (C) by setting the penalization to L1. Is the model sparse? What variables were removed from the model? Is the classifier performing well?

In [ ]:
```
params = {
    "C": np.logspace(-3, 3, 50)
}

gs = GridSearchCV(logreg, params, scoring="roc_auc")
res = gs.fit(x1_train, y1_train)

logreg = gs.best_estimator_
```

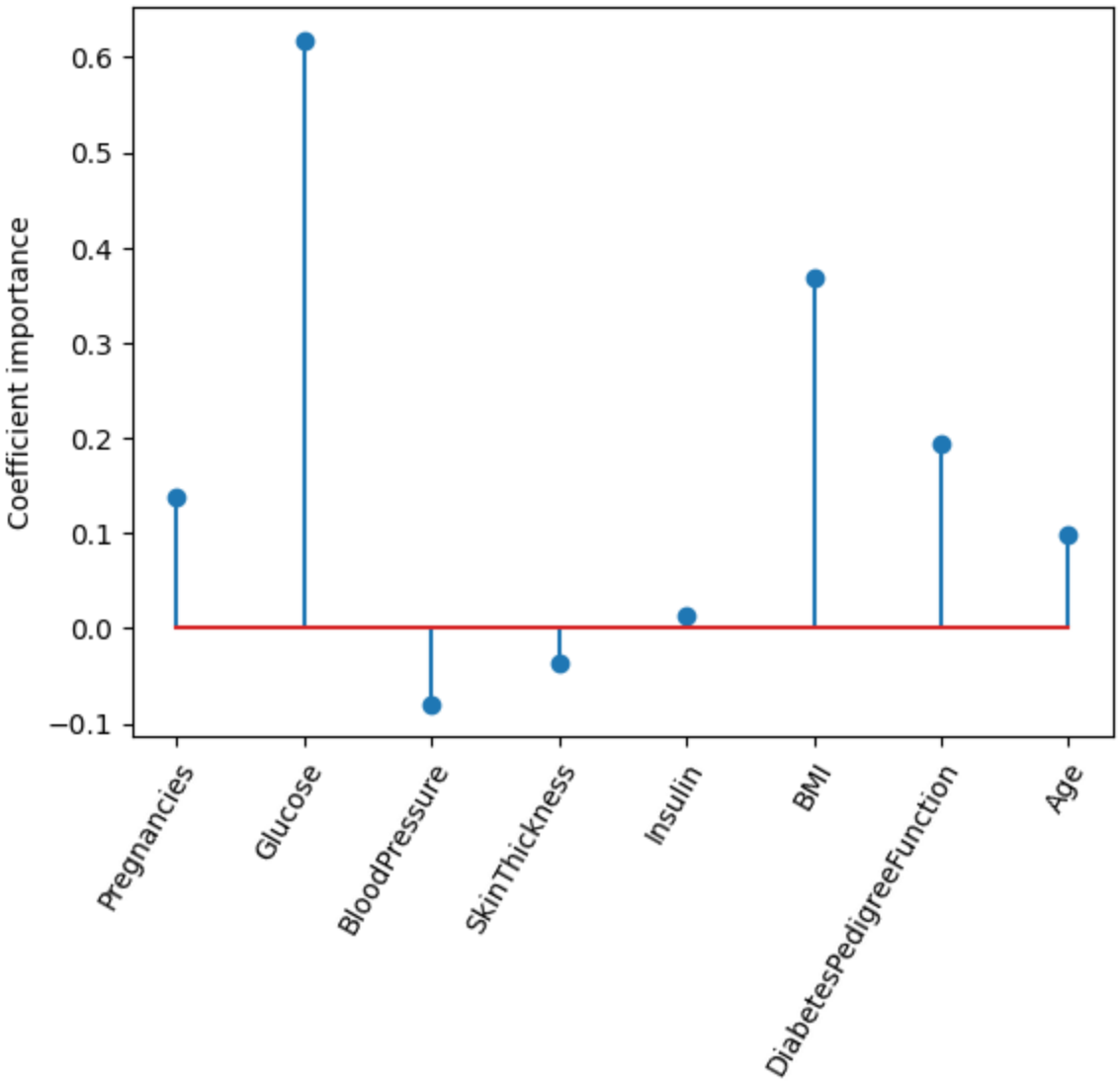
From the figure bellow, we notice that L1 regularization created some parsity in the model. 3 variables seem to have the most significant importance :

- Glucose
- BMI
- Diabetes Pedigree Function

On the other hand, 2 variables have been almost fully removed from the model :

- Skin Thickness
- Insulin

In [ ]:
```
plt.stem(logreg.coef_.flatten())
plt.xticks(np.arange(8), pima_vars, rotation=60, ha="right", rotation_mode="anchor"
plt.ylabel("Coefficient importance")
plt.show()
```



In [ ]:
```
pred = logreg.predict(x1_test)
pred_prob = logreg.predict_proba(x1_test)


logreg_acc_train = accuracy_score(y1_train, logreg.predict(x1_train))
logreg_roc_train = roc_auc_score(y1_train, logreg.predict_proba(x1_train)[:, 1])
```

```
Logreg_acc_test = accuracy_score(y1_test, pred)
Logreg_roc_test = roc_auc_score(y1_test, pred_prob[:, 1])

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, qda_acc_train, gnb_acc_train, logreg_acc_trai
    "Test Accuracy": [lda_acc_test, qda_acc_test, gnb_acc_test, logreg_acc_test],
    "Train ROC": [lda_roc_train, qda_roc_train, gnb_roc_train, logreg_roc_train],
    "Test ROC": [lda_roc_test, qda_roc_test, gnb_roc_test, logreg_roc_test]
}, index=["LDA", "QDA", "GNB", "LogReg"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LDA	0.80	0.77	0.86	0.85
QDA	0.79	0.78	0.85	0.85
GNB	0.78	0.77	0.83	0.84
LogReg	0.80	0.79	0.86	0.84

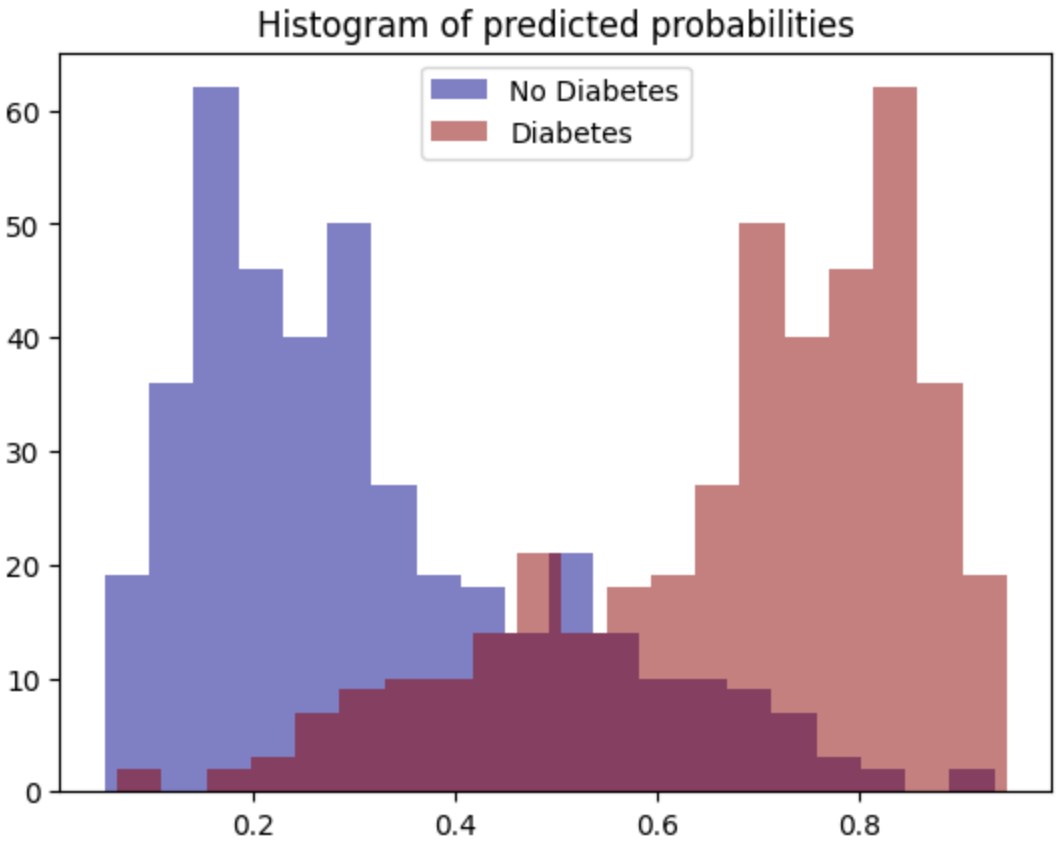
Given that we cross validate with ROC metric, its value have been inscreased (+0.4%) while accuracy decreased (-1.5%). Depending on the final use of the model, we should choose between accuracy and ROC to evaluate the model.

2.2.8. What is the best decision method so far? Is the best model linear (LAD,LR) on quadratic (QDA,NB)?

Based on the results above, the best model is Logistic Regression with L1 regularization. This model is linear so it's an explainable, simple and efficient model ; a very good compromise.

2.2.9. Interpret the separability of the sample in the predicted score space by plotting histograms for the samples for each class in 1D.

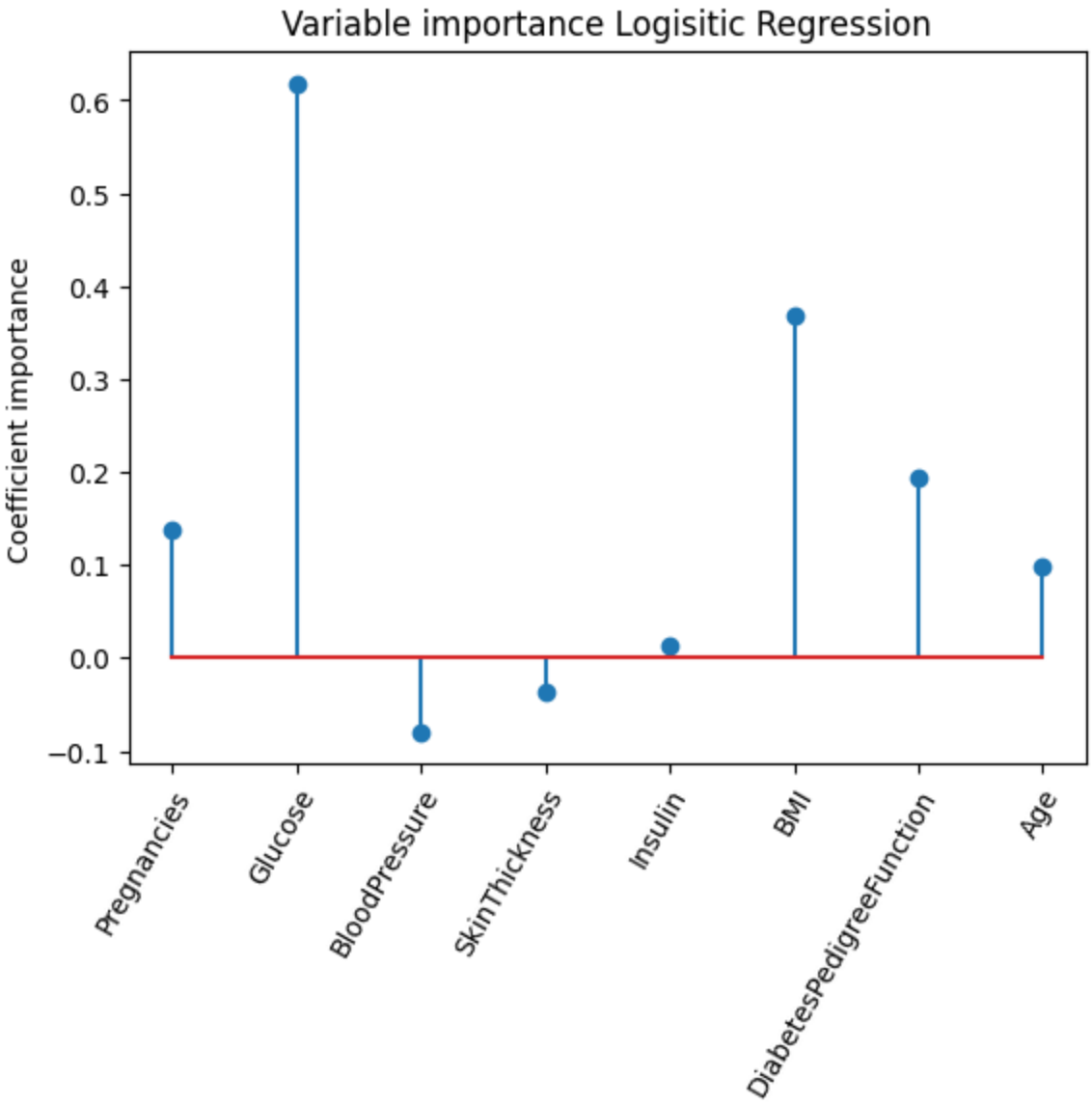
In [ ]:
```
plt.hist(pred_prob[:, 1], bins=20, color="darkblue", alpha=0.5)
plt.hist(pred_prob[:, 0], bins=20, color="darkred", alpha=0.5)
plt.title("Histogram of predicted probabilities")
plt.legend(["No Diabetes", "Diabetes"])
plt.show()
```



The classes are mainly separated in the predicted score space. However, there is still a consequent overlap (deep red area) between the two classes indicating that our predictor is far from perfect.

2.2.10. Interpret the weight for a good linear model. What is the effect of each variable on the risk of diabetes? Does it make medical sense?

In [ ]:
```
plt.stem(logreg.coef_.flatten())
plt.xticks(np.arange(8), pima_vars, rotation=60, ha="right", rotation_mode="anchor"
plt.ylabel("Coefficient importance")
plt.title("Variable importance Logisitic Regression")
plt.show()
```



As we can see from the figure above, the most important variables are (ranked by importance) :

- Glucose
- BMI
- Diabetes Pedigree Function

From a medical perspective, this is sensible :

- Glucose is tighly correlated to diabetes.
- BMI is a good indicator of body fat and consequently of diabetes (carefull to differentiate diabetes types).
- Diabetes Pedigree Function is a function indicating diabetes likelihood depending on the subject's age and his/her diabetic family history making it a relevant feature.

2.3. Non linear methods

In [ ]:
```
save = False
```

2.3.1. Random Forest Classifier

2.3.1.1. Fit the model with the default parameters and compute its prediction performance. Is it better than a linear estimator?

In [ ]:
```
rf = RandomForestClassifier(random_state=my_seed)
rf.fit(x1_train, y1_train.reshape(-1))

rf_pred_test = rf.predict(x1_test)
rf_pred_prob_test = rf.predict_proba(x1_test)

rf_acc_train = accuracy_score(y1_train, rf.predict(x1_train))
rf_roc_train = roc_auc_score(y1_train, rf.predict_proba(x1_train)[:, 1])
rf_acc_test = accuracy_score(y1_test, rf_pred_test)
rf_roc_test = roc_auc_score(y1_test, rf_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train],
    "Test Accuracy": [logreg_acc_test, rf_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test]
}, index=["LogReg", "RF"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.8	0.79	0.86	0.84
RF	1.0	0.76	1.00	0.84

Random forest (non-tuned) is slightly worse than Logistic Regression (fine-tuned) for both accuracy and Roc. However, we notice that it overfits the training data much more than Logistic Regression.

2.3.1.2. Do a quick validation of some of the important parameters (manually or with sklearn classes). Can you find a better performance?

In [ ]:
```
if save:
    params = {
        "n_estimators": [20, 25, 30, 35],
        "criterion": ["gini", "entropy", "logloss"],
        "max_depth": [None, 5, 6, 7],
        "min_samples_leaf": [10, 15, 20],
        "max_features": [None, 2, 3, 4]
    }

    gs = GridSearchCV(rf, params, scoring="roc_auc")
    res = gs.fit(x1_train, y1_train.reshape(-1))
    rf = gs.best_estimator_
```

In [ ]:
```
if save:
    with open('params/rf_best_estimator.pkl', 'wb') as f:
        pkl.dump(rf, f)
else:
    with open('params/rf_best_estimator.pkl', 'rb') as f:
        rf = pkl.load(f)
```

In [ ]:
```
rf_pred_test = rf.predict(x1_test)
rf_pred_prob_test = rf.predict_proba(x1_test)

rf_acc_train = accuracy_score(y1_train, rf.predict(x1_train))
rf_roc_train = roc_auc_score(y1_train, rf.predict_proba(x1_train)[:, 1])
rf_acc_test = accuracy_score(y1_test, rf_pred_test)
rf_roc_test = roc_auc_score(y1_test, rf_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train],
    "Test Accuracy": [logreg_acc_test, rf_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test]
}, index=["LogReg", "RF"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83

The model still overfits but much less than previously. In addition, its performances are unchanged but still inferior to Logistic Regression.

In [ ]:
```
rf.get_params()
```

Out[ ]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'entropy',
 'max_depth': 5,
 'max_features': 2,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 15,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 30,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 56,
 'verbose': 0,
 'warm_start': False}
```

2.3.2. Support Vector Machine Classifier

2.3.2.1. Fit the model with the default parameters and compute its prediction performance. Is it better than a linear estimator?

In [ ]:
```
svc = SVC(probability=True, random_state=my_seed)
svc.fit(x1_train, y1_train.reshape(-1))

svc_pred_test = svc.predict(x1_test)
svc_pred_prob_test = svc.predict_proba(x1_test)

svc_acc_train = accuracy_score(y1_train, svc.predict(x1_train))
svc_roc_train = roc_auc_score(y1_train, svc.predict_proba(x1_train)[:, 1])
svc_acc_test = accuracy_score(y1_test, svc_pred_test)
svc_roc_test = roc_auc_score(y1_test, svc_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train],
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test]
}, index=["LogReg", "RF", "SVC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.86	0.76	0.91	0.85

SVC untuned performs better around as good as Logistic Regression (fine-tuned) for both accuracy and Roc. However, we notice that it overfits the training data slightly more than Logistic Regression.

2.3.2.2. Do a quick validation of some of the important parameters (manually or with sklearn classes). Can you find a better performance?

In [ ]:
```
if save:
    params = {
        "C": np.logspace(-3, 3, 5),
        "gamma": np.logspace(-3, 3, 5),
        "kernel": ["rbf", "sigmoid"],
        "degree": [2, 3, 4],
        "coef0": [0, 1, 2],
        "shrinking": [True, False]
    }

    gs = GridSearchCV(svc, params, scoring="roc_auc")
    res = gs.fit(x1_train, y1_train.reshape(-1))
    svc = gs.best_estimator_
```

In [ ]:
```
if save:
    with open('params/svc_best_estimator.pkl', 'wb') as f:
        pkl.dump(svc, f)
else:
    with open('params/svc_best_estimator.pkl', 'rb') as f:
        svc = pkl.load(f)
```

In [ ]:
```
svc_pred_test = svc.predict(x1_test)
svc_pred_prob_test = svc.predict_proba(x1_test)

svc_acc_train = accuracy_score(y1_train, svc.predict(x1_train))
svc_roc_train = roc_auc_score(y1_train, svc.predict_proba(x1_train)[:, 1])
svc_acc_test = accuracy_score(y1_test, svc_pred_test)
svc_roc_test = roc_auc_score(y1_test, svc_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train],
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test]
}, index=["LogReg", "RF", "SVC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84

With this setting, SVC performs almost as good as Logistic Regression (-3% on test accuracy) but it doesn't overfitting as before.

In [ ]:
```
svc.get_params()
```

Out[ ]:
```
{'C': 1.0,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 1,
 'decision_function_shape': 'ovr',
 'degree': 2,
 'gamma': 0.03162277660168379,
 'kernel': 'sigmoid',
 'max_iter': -1,
 'probability': True,
 'random_state': 56,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

2.3.3. Multi-layer Perceptron Classifier

2.3.3.1. Fit the model with the default parameters and compute its prediction performance. Is it better than a linear estimator?

In [ ]:
```
mlp = MLPClassifier(random_state=my_seed)
mlp.fit(x1_train, y1_train.reshape(-1))

mlp_pred_test = mlp.predict(x1_test)
mlp_pred_prob_test = mlp.predict_proba(x1_test)

mlp_acc_train = accuracy_score(y1_train, mlp.predict(x1_train))
mlp_roc_train = roc_auc_score(y1_train, mlp.predict_proba(x1_train)[:, 1])
mlp_acc_test = accuracy_score(y1_test, mlp_pred_test)
mlp_roc_test = roc_auc_score(y1_test, mlp_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train, mlp_acc_train
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test, mlp_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train, mlp_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test, mlp_roc_test]
}, index=["LogReg", "RF", "SVC", "MLP"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84
MLP	0.84	0.76	0.92	0.84

Default MLP performs as good as fine-tuned SVC on test but if overfits the training data much more.

2.3.3.2. Do a quick validation of some of the important parameters (manually or with sklearn classes). Can you find a better performance?

In [ ]:
```
if save:
    params = {
        "hidden_layer_sizes": [(32, 32, 16), (32, 32, 16, 16)],
        "activation": ["identity", "relu"],
        "solver": ["lbfgs", "sgd", "adam"],
        "alpha": np.logspace(-4, -2, 3),
        "learning_rate": ["constant", "invscµaling", "adaptive"]
    }

    gs = GridSearchCV(mlp, params, scoring="roc_auc")
    res = gs.fit(x1_train, y1_train.reshape(-1))
    mlp = gs.best_estimator_
```

In [ ]:
```
if save:
    with open('params/mlp_best_estimator.pkl', 'wb') as f:
        pkl.dump(mlp, f)
```

```
else:
    with open('params/mlp_best_estimator.pkl', 'rb') as f:
        mlp = pkl.load(f)
```

In [ ]:
```
mlp_pred_test = mlp.predict(x1_test)
mlp_pred_prob_test = mlp.predict_proba(x1_test)

mlp_acc_train = accuracy_score(y1_train, mlp.predict(x1_train))
mlp_roc_train = roc_auc_score(y1_train, mlp.predict_proba(x1_train)[:, 1])
mlp_acc_test = accuracy_score(y1_test, mlp_pred_test)
mlp_roc_test = roc_auc_score(y1_test, mlp_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train, mlp_acc_train
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test, mlp_acc_test],
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train, mlp_roc_train],
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test, mlp_roc_test]
}, index=["LogReg", "RF", "SVC", "MLP"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84
MLP	0.81	0.78	0.86	0.85

Thanks to the fine-tuning, MLP performs better than all other models (+1% on roc score on test set). Overfitting is still present but much less than before.

In [ ]:
```
mlp.get_params()
```

Out[ ]:
```
{'activation': 'identity',
 'alpha': 0.0001,
 'batch_size': 'auto',
 'beta_1': 0.9,
 'beta_2': 0.999,
 'early_stopping': False,
 'epsilon': 1e-08,
 'hidden_layer_sizes': (32, 32, 16),
 'learning_rate': 'constant',
 'learning_rate_init': 0.001,
 'max_fun': 15000,
 'max_iter': 200,
 'momentum': 0.9,
 'n_iter_no_change': 10,
 'nesterovs_momentum': True,
 'power_t': 0.5,
 'random_state': 56,
 'shuffle': True,
 'solver': 'sgd',
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': False,
 'warm_start': False}
```

2.3.4. Gradient Boosting Classifier

2.3.4.1. Fit the model with the default parameters and compute its prediction performance. Is it better than a linear estimator?

In [ ]:
```
gbc = GradientBoostingClassifier(random_state=my_seed)
gbc.fit(x1_train, y1_train.reshape(-1))

gbc_pred_test = gbc.predict(x1_test)
gbc_pred_prob_test = gbc.predict_proba(x1_test)

gbc_acc_train = accuracy_score(y1_train, gbc.predict(x1_train))
gbc_roc_train = roc_auc_score(y1_train, gbc.predict_proba(x1_train)[:, 1])
gbc_acc_test = accuracy_score(y1_test, gbc_pred_test)
gbc_roc_test = roc_auc_score(y1_test, gbc_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train, mlp_acc_train
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test, mlp_acc_test, gbc
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train, mlp_roc_train, gbc
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test, mlp_roc_test, gbc_roc_
}, index=["LogReg", "RF", "SVC", "MLP", "GBC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84
MLP	0.81	0.78	0.86	0.85
GBC	0.99	0.77	1.00	0.81

We notice that Gradient Boosting Classifier dangerously overfits the training data. It means that we should make the model less complex with the use of hyperparameters.

2.3.4.2. Do a quick validation of some of the important parameters (manually or with sklearn classes). Can you find a better performance?

In [ ]:
```
if save:
    params = {
        "loss": ["log_loss", "exponential"],
        "learning_rate": np.logspace(-4, -2, 3),
        "criterion": ["friedman_mse", "squarred_error"],
        "n_estimators": [20, 25, 30],
        "max_depth": [3, 4, 5],
        "min_samples_leaf": [15, 20, 25],
        "max_features": [None, 2, 3]
    }

    gs = GridSearchCV(gbc, params, scoring="roc_auc")
    res = gs.fit(x1_train, y1_train.reshape(-1))
    gbc = gs.best_estimator_
```

In [ ]:
```
if save:
    with open('params/gbc_best_estimator.pkl', 'wb') as f:
        pkl.dump(gbc, f)
else:
    with open('params/gbc_best_estimator.pkl', 'rb') as f:
        gbc = pkl.load(f)
```

In [ ]:
```
gbc_pred_test = gbc.predict(x1_test)
gbc_pred_prob_test = gbc.predict_proba(x1_test)

gbc_acc_train = accuracy_score(y1_train, gbc.predict(x1_train))
gbc_roc_train = roc_auc_score(y1_train, gbc.predict_proba(x1_train)[:, 1])
gbc_acc_test = accuracy_score(y1_test, gbc_pred_test)
gbc_roc_test = roc_auc_score(y1_test, gbc_pred_prob_test[:, 1])

pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train, mlp_acc_train
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test, mlp_acc_test, gbc
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train, mlp_roc_train, gbc
```

```
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test, mlp_roc_test, gbc_roc_
}, index=["LogReg", "RF", "SVC", "MLP", "GBC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84
MLP	0.81	0.78	0.86	0.85
GBC	0.67	0.65	0.92	0.82

Despite many attempts on the fine-tuning, Gradient Boosting Classifier performs worse than all other models (-2% for roc score on test set) and overfits the training data much more. This might be due to complexity of the model in comparison to the others.

In [ ]:
```
gbc.get_params()
```

Out[ ]:
```
{'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 0.001,
 'loss': 'exponential',
 'max_depth': 4,
 'max_features': 2,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 20,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 25,
 'n_iter_no_change': None,
 'random_state': 56,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

2.4. Comparision and interpretation

2.4.1. Collect the test performances for all methods investigated above in a table (in a dataframe and printing it for instance). Which methods work the best in practice?

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [logreg_acc_train, rf_acc_train, svc_acc_train, mlp_acc_train
    "Test Accuracy": [logreg_acc_test, rf_acc_test, svc_acc_test, mlp_acc_test, gbc
    "Train ROC": [logreg_roc_train, rf_roc_train, svc_roc_train, mlp_roc_train, gbc
```

```
    "Test ROC": [logreg_roc_test, rf_roc_test, svc_roc_test, mlp_roc_test, gbc_roc_
}, index=["LogReg", "RF", "SVC", "MLP", "GBC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train ROC	Test ROC
LogReg	0.80	0.79	0.86	0.84
RF	0.83	0.75	0.92	0.83
SVC	0.80	0.76	0.86	0.84
MLP	0.81	0.78	0.86	0.85
GBC	0.67	0.65	0.92	0.82

For this study, I chose to use the roc score as performance metric for the cross validation. From looking at the above results, MLP is the best method overall in terms of roc score and even accuracy. However, the delta between MLP and Logistic Regression is very small. Considering that Logistic Regression is a much simpler, more explainable, less costly model, I would choose Logistic Regression as the best method in practice.

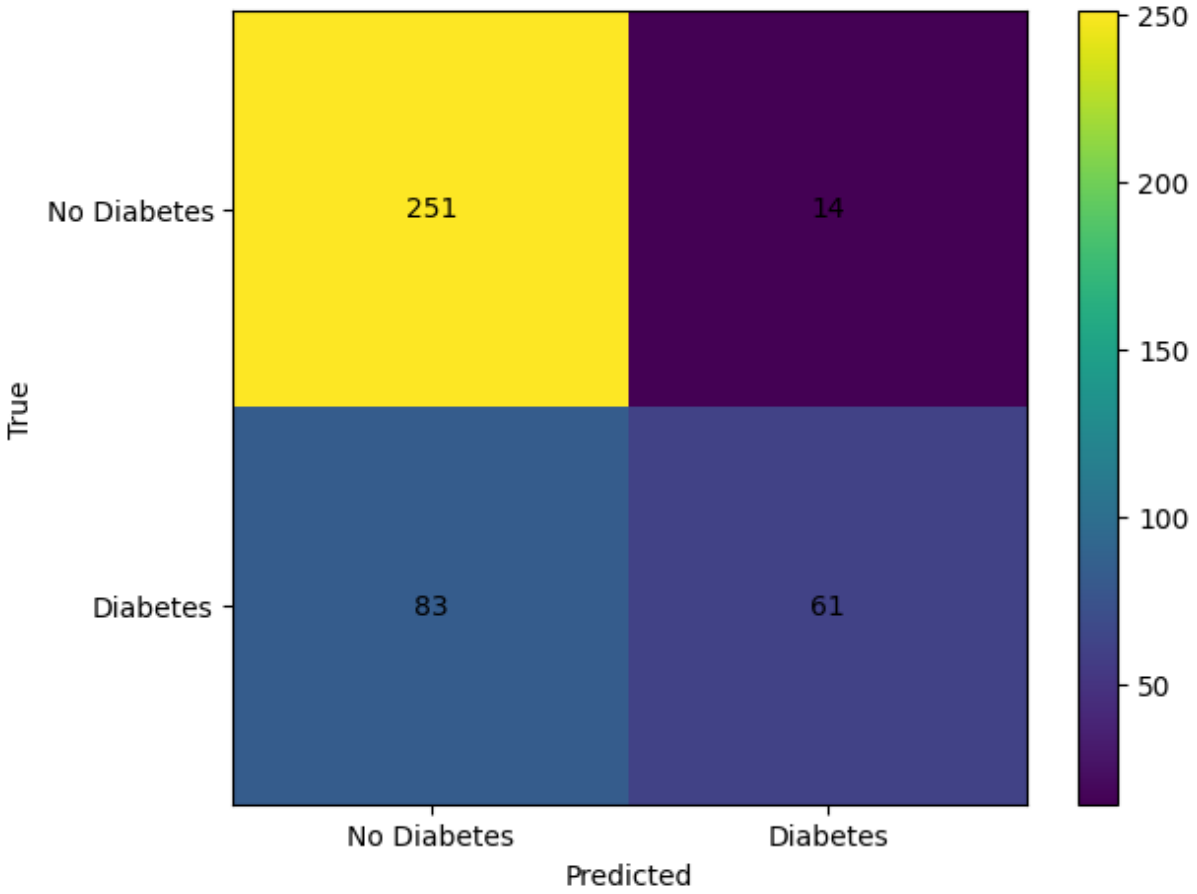
2.4.2. Which model is best from a medical/practical perspective? Do we need non-linearity in this applica-tion?

For medical application, we would prefer an explainable model. In this case Logistic Regression with L1 regularization is working well and doesn't overfit the data so it's a good candidate. However, MLP is performing better and is not overfitting the data so it's also a good candidate even though it isn't explainable.

2.4.3. For the best model, compute the confusion matrix for the test data. What is the false negative rate (FNR) (FN/(FN+TP)) for this classifier? Is it good for this kind of applications?

In [ ]:

```
plt.imshow(confusion_matrix(y1_test, logreg.predict(x1_test)))
for i in range(2):
    for j in range(2):
        plt.text(j, i, confusion_matrix(y1_test, logreg.predict(x1_test))[i, j], ha
plt.xticks([0, 1], ["No Diabetes", "Diabetes"])
plt.yticks([0, 1], ["No Diabetes", "Diabetes"])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.colorbar()
plt.tight_layout()
```



Here the false negative rate is : $FNR = \frac{FN}{(FN+TP)} = \frac{83}{83+61} = 0.57 = 57\%$

The false negative rate is very high. Such a False Negative Rate is relatively bad as it indicates the probability of telling someone that he is healthy while he actually suffers from diabetes. This kind of error could lead to people dying so it has to be avoided by all means. This error is most likely due to class imbalance as you can see below :

In [ ]:
```
class_perc = {i:str(round(np.sum(y1_train == i)/y1_train.shape[0]*100, 2))+"%" for
print("Non-Diabetes:", class_perc[-1], "| Diabetes:", class_perc[1])
```

```
Non-Diabetes: 67.33% | Diabetes: 32.67%
```

2.4.4. Since a false negative can have an important medical impact, propose a new threshold for the predicted score that leads to a FNR of less that 10% (this can be done by changing manually the value of the intercept_ in the trained classifier).

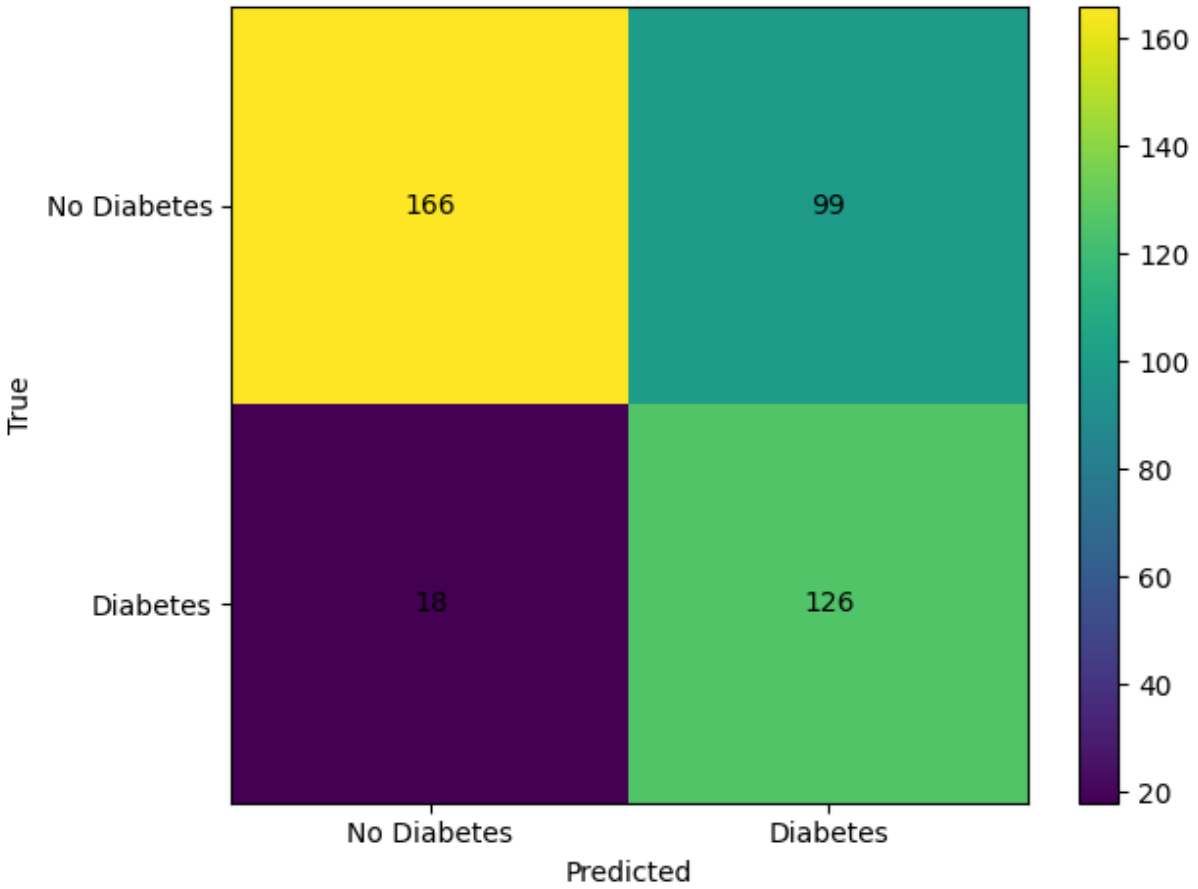
In [ ]:
```
intercepts = np.linspace(-5, 5, 20)
FNR = lambda conf_mat: conf_mat[1, 0]/(conf_mat[1, 0]+conf_mat[1, 1])

for intercept in intercepts:
    logreg.intercept_ = intercept
    conf_mat = confusion_matrix(y1_train, logreg.predict(x1_train))
    if FNR(conf_mat) < 0.1:
        found_intercept = intercept
        found_conf_mat = conf_mat
```

```
        break
print("Intercept:", round(found_intercept, 2), "| FNR:", round(FNR(found_conf_mat),
```

```
Intercept: 0.26 | FNR: 0.09
```

In [ ]:
```
plt.imshow(confusion_matrix(y1_test, logreg.predict(x1_test)))
for i in range(2):
    for j in range(2):
        plt.text(j, i, confusion_matrix(y1_test, logreg.predict(x1_test))[i, j], ha
plt.xticks([0, 1], ["No Diabetes", "Diabetes"])
plt.yticks([0, 1], ["No Diabetes", "Diabetes"])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.colorbar()
plt.tight_layout()
```



Now the false negative rate is : $FNR = \frac{FN}{(FN+TP)} = \frac{18}{18+126} = 12.5\%$ which is much better than before and close to the 10% threshold.

3. Predicting Classes for Digits Dataset

3.1. Evaluate the different supervised methods

In [ ]:
```
save = False
```

3.1.1. Linear Discriminant Analysis

3.1.1.1. Fit the model with the default parameters and compute its prediction performance (accuracy on test data).

In [ ]:
```
lda = LinearDiscriminantAnalysis(solver="lsqr")
lda.fit(x2, y2)

lda_pred_train = lda.predict(x2)
lda_pred_test = lda.predict(xt2)

pd.DataFrame({
    "Train Accuracy": [accuracy_score(y2, lda_pred_train)],
    "Test Accuracy": [accuracy_score(yt2, lda_pred_test)]
}, index=["LDA"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.98	0.93

It looks like default LDA slightly overfits the training data. It still performs well on test.

3.1.1.2. Do a quick validation of some of the important parameters (manually or with sklearn classes) to get a better performance if possible.

In [ ]:
```
if save:
    params = {
        "shrinkage": np.linspace(0, 1, 50)
    }

    gs = GridSearchCV(lda, params, scoring="accuracy")
    res = gs.fit(x2, y2)

    lda = gs.best_estimator_

    lda_pred_train = lda.predict(x2)
    lda_pred_test = lda.predict(xt2)

    lda_acc_train = accuracy_score(y2, lda_pred_train)
    lda_acc_test = accuracy_score(yt2, lda_pred_test)
```

3.1.1.3. Store the model and the accuracy for the best parameter configuration.

In [ ]:
```
lda, lda_acc_train, lda_acc_test = get_save_estimator(lda, lda_acc_train, lda_acc_t
```

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [lda_acc_train],
```

```
    "Test Accuracy": [lda_acc_test]
}, index=["LDA"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95

Overfitting has been almost completely removed. Test accuracy has increased by 2% while training accuracy has decreased by 2%. This is excellent as it indicates that the model generalizes better.

3.1.2. Logistic Regression

3.1.2.1. Fit the model with the default parameters and compute its prediction performance (accuracy on test data).

In [ ]:

```
logreg = LogisticRegression()
logreg.fit(x2, y2)

logreg_pred_train = logreg.predict(x2)
logreg_pred_test = logreg.predict(xt2)

logreg_acc_train = accuracy_score(y2, logreg_pred_train)
logreg_acc_test = accuracy_score(yt2, logreg_pred_test)

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train],
    "Test Accuracy": [lda_acc_test, logreg_acc_test]
}, index=["LDA", "LogReg"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97

3.1.2.2. Do a quick validation of some of the important parameters (manually or with sklearn classes) to get a better performance if possible.

In [ ]:

```
if save:
    params = {
        "C": np.logspace(-3, 3, 50)
    }

    gs = GridSearchCV(logreg, params, scoring="accuracy")
    res = gs.fit(x2, y2)

    logreg = gs.best_estimator_

    logreg_pred_train = logreg.predict(x2)
    logreg_pred_test = logreg.predict(xt2)
```

```
    logreg_acc_train = accuracy_score(y2, logreg_pred_train)
    logreg_acc_test = accuracy_score(yt2, logreg_pred_test)
```

3.1.2.3. Store the model and the accuracy for the best parameter configuration.

In [ ]:
```
logreg, logreg_acc_train, logreg_acc_test = get_save_estimator(logreg, logreg_acc_t
```

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train],
    "Test Accuracy": [lda_acc_test, logreg_acc_test]
}, index=["LDA", "LogReg"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97

Performances are unchanged, the model overfits the training data a bit more than LDA but performs better.

3.1.3. Support Vector Classifier

3.1.3.1. Fit the model with the default parameters and compute its prediction performance (accuracy on test data).

In [ ]:
```
svc = SVC(random_state=my_seed)
svc.fit(x2, y2)

svc_pred_train = svc.predict(x2)
svc_pred_test = svc.predict(xt2)

svc_acc_train = accuracy_score(y2, svc_pred_train)
svc_acc_test = accuracy_score(yt2, svc_pred_test)

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train, svc_acc_train],
    "Test Accuracy": [lda_acc_test, logreg_acc_test, svc_acc_test]
}, index=["LDA", "LogReg", "SVC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97
SVC	1.00	0.98

3.1.3.2. Do a quick validation of some of the important parameters (manually or with sklearn classes) to get a better performance if possible.

In [ ]:
```
if save:
    params = {
        "C": [1, 10, 100, 1000],
        "gamma": [1, 0.1, 0.001, 0.0001],
        "kernel": ["linear", "rbf"]
    }

    gs = GridSearchCV(svc, params, scoring="accuracy")
    res = gs.fit(x2, y2)

    svc = gs.best_estimator_

    svc_pred_train = svc.predict(x2)
    svc_pred_test = svc.predict(xt2)

    svc_acc_train = accuracy_score(y2, svc_pred_train)
    svc_acc_test = accuracy_score(yt2, svc_pred_test)
```

3.1.3.3. Store the model and the accuracy for the best parameter configuration.

In [ ]:
```
svc, svc_acc_train, svc_acc_test = get_save_estimator(svc, svc_acc_train, svc_acc_t
```

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train, svc_acc_train],
    "Test Accuracy": [lda_acc_test, logreg_acc_test, svc_acc_test]
}, index=["LDA", "LogReg", "SVC"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97
SVC	1.00	0.98

Same observation as for Logistic Regression but it performs slightly better (+1% on test accuracy).

3.1.4. Multi-layer Perceptron Classifier

3.1.4.1. Fit the model with the default parameters and compute its prediction performance (accuracy on test data).

In [ ]:
```
mlp = MLPClassifier(random_state=my_seed)
mlp.fit(x2, y2)

mlp_pred_train = mlp.predict(x2)
```

```
mlp_pred_test = mlp.predict(xt2)

mlp_acc_train = accuracy_score(y2, mlp_pred_train)
mlp_acc_test = accuracy_score(yt2, mlp_pred_test)

pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train, svc_acc_train, mlp_acc_trai
    "Test Accuracy": [lda_acc_test, logreg_acc_test, svc_acc_test, mlp_acc_test]
}, index=["LDA", "LogReg", "SVC", "MLP"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97
SVC	1.00	0.98
MLP	1.00	0.98

3.1.4.2. Do a quick validation of some of the important parameters (manually or with sklearn classes) to get a better performance if possible.

In [ ]:
```
if save:
    params = {
        "hidden_layer_sizes": [(64), (64, 64)],
        "activation": ["identity", "relu"],
        "solver": ["sgd", "adam"],
    }

    gs = GridSearchCV(mlp, params, scoring="accuracy")
    res = gs.fit(x2, y2)

    mlp = gs.best_estimator_

    mlp_pred_train = mlp.predict(x2)
    mlp_pred_test = mlp.predict(xt2)

    mlp_acc_train = accuracy_score(y2, mlp_pred_train)
    mlp_acc_test = accuracy_score(yt2, mlp_pred_test)
```

3.1.4.3. Store the model and the accuracy for the best parameter configuration.

In [ ]:
```
mlp, mlp_acc_train, mlp_acc_test = get_save_estimator(mlp, mlp_acc_train, mlp_acc_t
```

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [lda_acc_train, logreg_acc_train, svc_acc_train, mlp_acc_trai
    "Test Accuracy": [lda_acc_test, logreg_acc_test, svc_acc_test, mlp_acc_test]
}, index=["LDA", "LogReg", "SVC", "MLP"]).round(2)
```

Out[ ]:

	Train Accuracy	Test Accuracy
LDA	0.96	0.95
LogReg	1.00	0.97
SVC	1.00	0.98
MLP	1.00	0.98

The MLP model performs exactly as good as SVC. Condisering that SVC is more explainable and less costly, we should prefer SVC.

3.2. Interpreting the classifier

3.2.1. Compare the performances of the different methods (with different metrics). Which model is the best on test data?

As asked in the practical session, we chose accuracy as performance metric for the fine-tuning. Now let's try other metrics in addition to accuracy :

- F1 score
- Precision
- Recall

First, let's get predictions for each model :

In [ ]:
```
lda_train_pred = lda.predict(x2)
lda_test_pred = lda.predict(xt2)

logreg_train_pred = logreg.predict(x2)
logreg_test_pred = logreg.predict(xt2)

svc_train_pred = svc.predict(x2)
svc_test_pred = svc.predict(xt2)

mlp_train_pred = mlp.predict(x2)
mlp_test_pred = mlp.predict(xt2)
```

Now, we can compute the different metrics :

In [ ]:
```
lda_train_acc, lda_train_f1, lda_train_prec, lda_train_rec = get_acc_f1_prec_rec(y2
lda_test_acc, lda_test_f1, lda_test_prec, lda_test_rec = get_acc_f1_prec_rec(yt2, l

logreg_train_acc, logreg_train_f1, logreg_train_prec, logreg_train_rec = get_acc_f1
logreg_test_acc, logreg_test_f1, logreg_test_prec, logreg_test_rec = get_acc_f1_pre

svc_train_acc, svc_train_f1, svc_train_prec, svc_train_rec = get_acc_f1_prec_rec(y2
svc_test_acc, svc_test_f1, svc_test_prec, svc_test_rec = get_acc_f1_prec_rec(yt2, s
```

```
mlp_train_acc, mlp_train_f1, mlp_train_prec, mlp_train_rec = get_acc_f1_prec_rec(y2
mlp_test_acc, mlp_test_f1, mlp_test_prec, mlp_test_rec = get_acc_f1_prec_rec(yt2, m
```

In [ ]:
```
pd.DataFrame({
    "Train Accuracy": [lda_train_acc, logreg_train_acc, svc_train_acc, mlp_train_ac
    "Test Accuracy": [lda_test_acc, logreg_test_acc, svc_test_acc, mlp_test_acc],
    "Train F1": [lda_train_f1, logreg_train_f1, svc_train_f1, mlp_train_f1],
    "Test F1": [lda_test_f1, logreg_test_f1, svc_test_f1, mlp_test_f1],
    "Train Precision": [lda_train_prec, logreg_train_prec, svc_train_prec, mlp_trai
    "Test Precision": [lda_test_prec, logreg_test_prec, svc_test_prec, mlp_test_pre
    "Train Recall": [lda_train_rec, logreg_train_rec, svc_train_rec, mlp_train_rec]
    "Test Recall": [lda_test_rec, logreg_test_rec, svc_test_rec, mlp_test_rec]
}, index=["LDA", "LogReg", "SVC", "MLP"]).round(2).style.highlight_max(color="purpl
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train F1	Test F1	Train Precision	Test Precision	Train Recall	Test Recall
LDA	0.960000	0.950000	0.960000	0.950000	0.960000	0.950000	0.960000	0.950000
LogReg	1.000000	0.970000	1.000000	0.970000	1.000000	0.970000	1.000000	0.970000
SVC	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000
MLP	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000

From the table above, the two best models are SVC and MLP. Given that MLP is less explainable and more complex than SVC, we will consider SVC as showing the best performances.

3.2.2. Select the best classifier from the previous section and use it to predict labels on the test data.

As explained above, SVC is the best overall classifier considering our fine-tuning. We will consider this model for the next part of this pratical session :

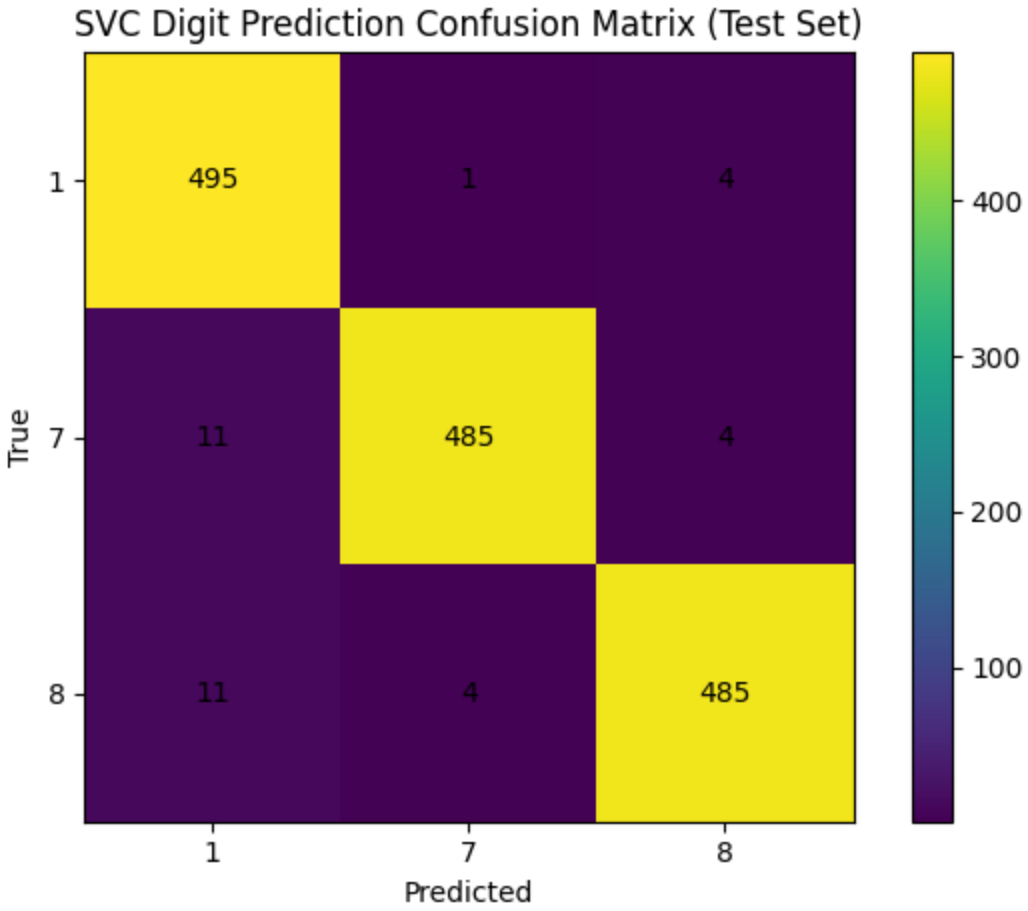
In [ ]:
```
svc_test_pred = svc.predict(xt2)
```

3.2.3. Compute the confusion matrix and interpret the errors made by the classifier. What is the class that is the most difficult to recognize? Are there somme classes that are harder to discriminate?

In [ ]:
```
conf_mat = confusion_matrix(yt2, svc_test_pred)
labels = ["1", "7", "8"]

plt.imshow(conf_mat)
for i in range(len(conf_mat)):
    for j in range(len(conf_mat)):
        plt.text(j, i, conf_mat[i, j], ha="center", va="center")
plt.xticks([0, 1, 2], labels)
plt.yticks([0, 1, 2], labels)
plt.xlabel("Predicted")
plt.ylabel("True")
```

```
plt.colorbar()
plt.title("SVC Digit Prediction Confusion Matrix (Test Set)")
plt.tight_layout()
```



From reading the confusion matrix :

- The most difficult classes to recognize are 7 and 8 as they both have 15 missclassified samples.
- The easiest class to recognize is 1 as it has only 5 missclassified samples.

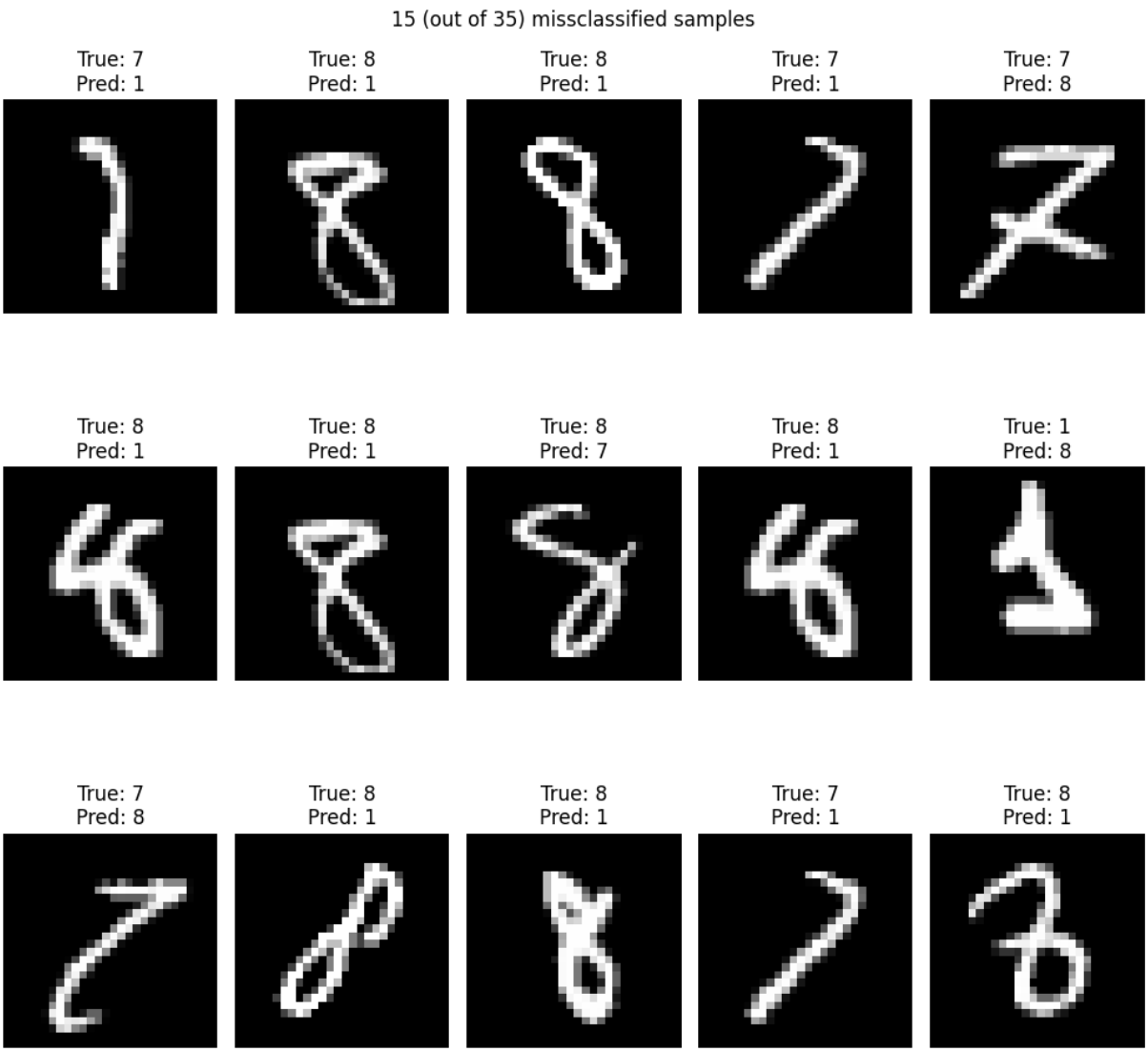
In fact, 1 has the simplest shape of all 3 classes this could be an explanation for this result.

3.2.4. Plot some of the sample that are miss-classified. Are they difficult to recognize ? Why did the classifier fail to recognize them?

In [ ]:
```
missed = [i for i in range(len(yt2)) if yt2[i] != svc_test_pred[i]]
missed_rand = np.random.choice(missed, 15)
```

In [ ]:
```
plt.figure(figsize=(10, 10))
for i in range(15):
    plt.subplot(3, 5, i+1)
    plt.imshow(xt2[missed_rand[i]].reshape(28, 28), cmap="gray")
    plt.title("True: "+str(yt2[missed_rand[i]][0])+"\nPred: "+str(svc_test_pred[mis
    plt.axis("off")
plt.tight_layout()
```

```
plt.suptitle("15 (out of 35) missclassified samples")
plt.show()
```



Many of the above samples are hard to recognize even for a human being. Consequently, it isn't surprising that the classifier failed to recognize them. Others are easier but positionned and structured in an unusual way (curvy 7 predicted as a 8 or stretched 8 predicted as a 1).

Here are my hypothesis as of why the classifier fails on these samples :

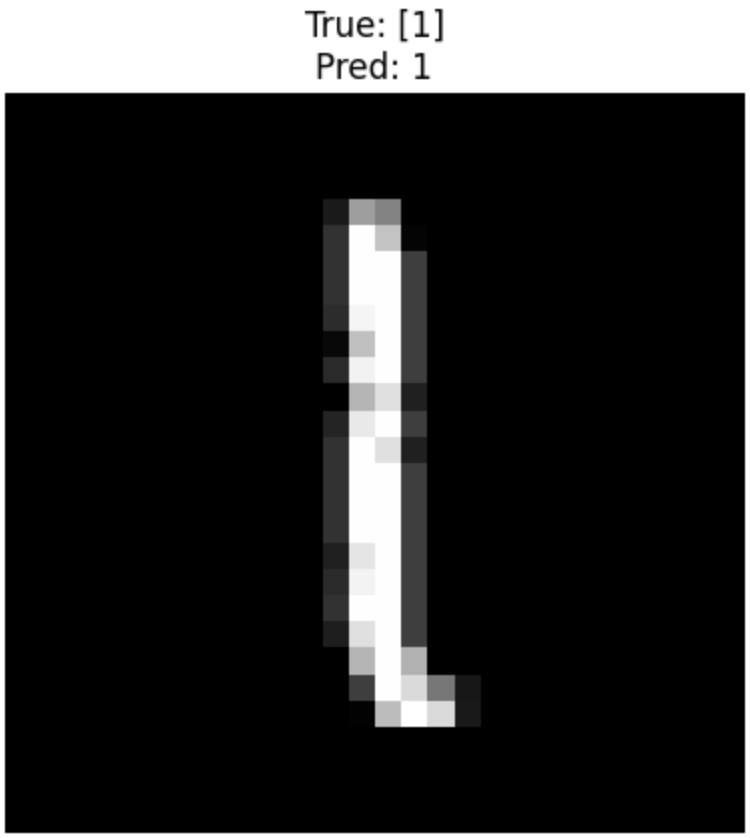
- The sample is hard to recognize even for a human being.
- The sample is unusual and the classifier hasn't been trained on such samples (data augmentation or CNN and its translation invariance property could help).

3.2.5. Pick a well classified sample and create 1000 noisy samples of it by adding gaussian noise (np.random.randn). Pick a level of noise that allows you to still distinguish clearly the class. Compute the accuracy of the classifier on those 1000 noisy samples. If the accuracy is 1, increase the noise level or choose another well classified sample.

First, we must pick a well classified sample :

In [ ]:
```
well = [i for i in range(len(yt2)) if yt2[i] == svc_test_pred[i]]
np.random.seed(my_seed)
well_rand_index = np.random.choice(well, 1, )
```

In [ ]:
```
plt.imshow(xt2[well_rand_index[0]].reshape(28, 28), cmap="gray")
plt.title("True: "+str(yt2[well_rand_index[0]])+"\nPred: "+str(svc_test_pred[well_r
plt.axis("off")
plt.show()
```

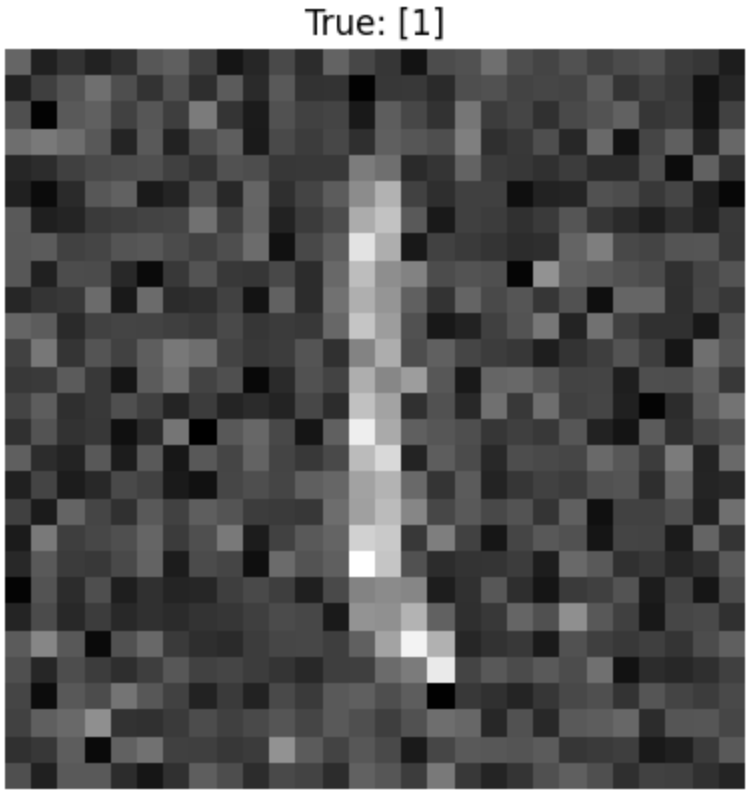


Secondly, we will add gaussian noise to this sample :

In [ ]:
```
n_noisy = 1000
noise_factor = 0.25
noisy_samples = []

for i in range(n_noisy):
    noisy_samples.append(xt2[well_rand_index].reshape(28, 28) + noise_factor * np.r
```

In [ ]:
```
plt.imshow(noisy_samples[0].reshape(28, 28), cmap="gray")
plt.title("True: "+str(yt2[well_rand_index[0]]))
plt.axis("off")
plt.show()
```



In [ ]:
```
noisy_samples = np.array(noisy_samples).reshape(-1, 28*28)
```

In [ ]:
```
svc_pred_noisy = svc.predict(noisy_samples)
svc_acc_noisy = accuracy_score([yt2[well_rand_index][0][0] for i in range(len(svc_p
print(f"Accuracy on noisy samples : {str(svc_acc_noisy*100)}%")
```

```
Accuracy on noisy samples : 99.2%
```

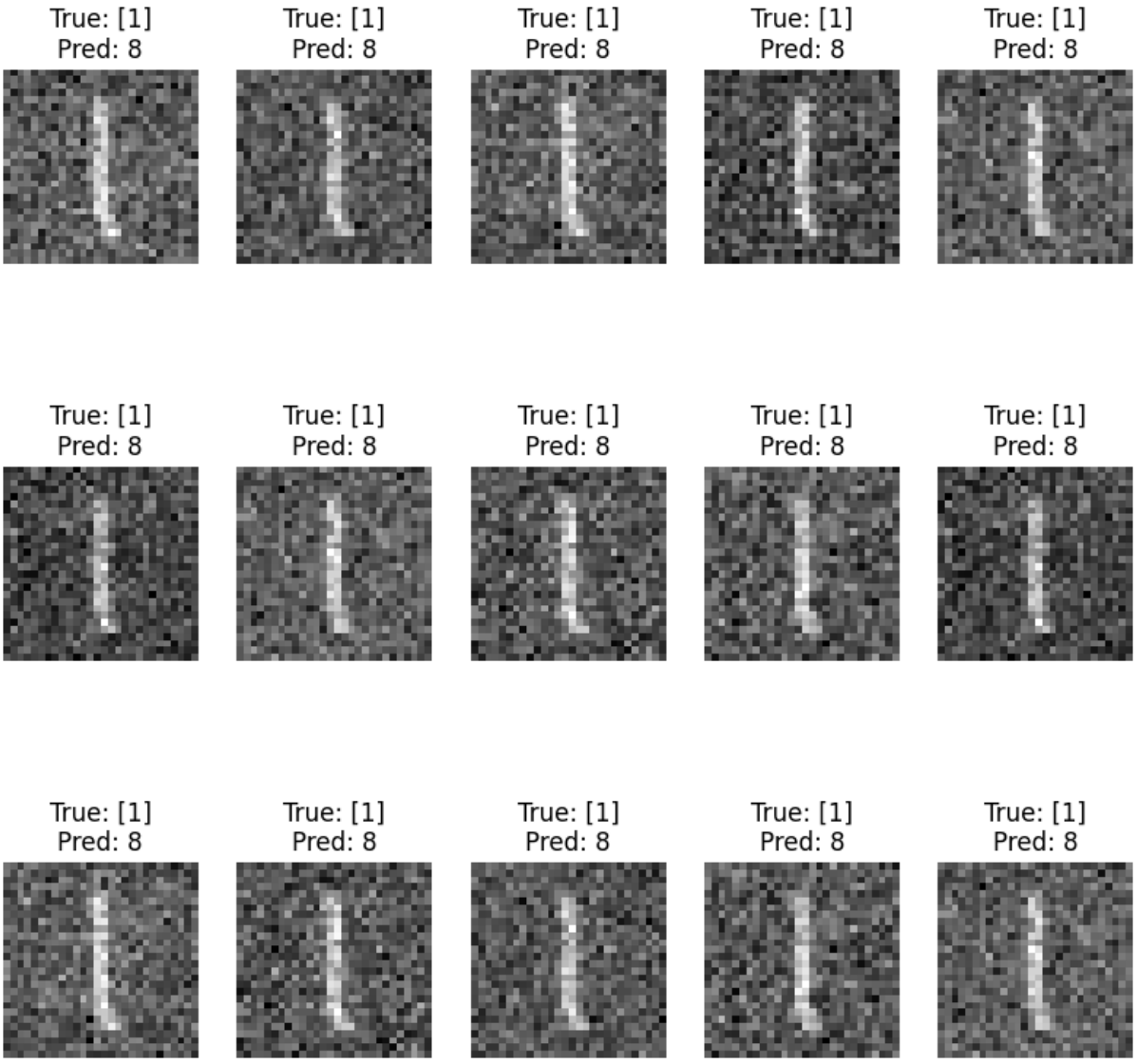
While the original sample was well classified, around 0.7% of adversarial examples have been misclassified. It means that gaussian noise contributed to misslead the model.

3.2.6. How robust is the classifier? Visualise some of those "adversarial" examples when the accuracy on the noisy samples is not 1.

Let's visualize some of the missclassified adversarial examples :

In [ ]:
```
miss_adv = [i for i in range(len(svc_pred_noisy)) if svc_pred_noisy[i] != yt2[well_
np.random.seed(my_seed)
miss_adv_rand = np.random.choice(miss_adv, 15)

plt.figure(figsize=(10, 10))
for i in range(15):
    plt.subplot(3, 5, i+1)
    plt.imshow(noisy_samples[miss_adv_rand[i]].reshape(28, 28), cmap="gray")
    plt.title("True: "+str(yt2[well_rand_index[0]])+"\nPred: "+str(svc_pred_noisy[m
    plt.axis("off")
```



It seems that the classifier always makes the same mistake with this specific noise ; it classifies the sample as a 8 while it is in fact a 1. This could be explained by the shapes of the digits, the model is used to "1" digits that are in a straight lies while the adversarial examples have noise around.

As a conclusion, our model is relatively robust to gaussian noise adversarial examples. I tested multiple well classified samples and the one that caused the most missclassifications still showed 98.2% accuracy on adversarial examples.

4. Convolutionnal Neural Network Bonus

4.1. Implement a CNN and train it on the data (you will need to reshape it to store it as images). Investigate te performance of the CNN when varying its parameters. Does it have better performance than the model above? Is it more robust to adversarial examples?.

4.1.1. Construct the model

In [ ]:
```
x2_im = torch.from_numpy(x2.reshape(-1, 28, 28))
xt2_im = torch.from_numpy(xt2.reshape(-1, 28, 28))

# We only have 3 labels out of the 10 possible ones. We need to map them to 0, 1 an
map = {1:0, 7:1, 8:2}
rev_map = {0:1, 1:7, 2:8}

y2_im = torch.tensor([map[i] for i in y2.squeeze(1)])
yt2_im = torch.tensor([map[i] for i in yt2.squeeze(1)])
```

In [ ]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(1600, 128)
        self.fc2 = nn.Linear(128, 3)  # 3 classes

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = CNN()

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Convert data to torch tensors
train_dataset = TensorDataset(x2_im.unsqueeze(1).float(), y2_im.squeeze().long())
test_dataset = TensorDataset(xt2_im.unsqueeze(1).float(), yt2_im.squeeze().long())

# Create data loaders
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
    # Evaluation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Epoch {epoch+1}/{num_epochs}, Test Accuracy: {accuracy:.2f}%')
```

```
Epoch 1/10, Test Accuracy: 92.73%
Epoch 2/10, Test Accuracy: 97.40%
Epoch 3/10, Test Accuracy: 97.80%
Epoch 4/10, Test Accuracy: 98.60%
Epoch 5/10, Test Accuracy: 98.53%
Epoch 6/10, Test Accuracy: 99.00%
Epoch 7/10, Test Accuracy: 99.13%
Epoch 8/10, Test Accuracy: 98.87%
Epoch 9/10, Test Accuracy: 98.87%
Epoch 10/10, Test Accuracy: 98.87%
```

In [ ]:
```
with torch.no_grad():
    test_outputs = model(xt2_im.unsqueeze(1).float())

predicted_labels = torch.argmax(test_outputs, dim=1)
```

In [ ]:
```
cnn_train_pred = torch.argmax(model(x2_im.unsqueeze(1).float()), dim=1).detach().nu
cnn_test_pred = torch.argmax(model(xt2_im.unsqueeze(1).float()), dim=1).detach().nu

cnn_train_acc, cnn_train_f1, cnn_train_prec, cnn_train_rec = get_acc_f1_prec_rec(y2
cnn_test_acc, cnn_test_f1, cnn_test_prec, cnn_test_rec = get_acc_f1_prec_rec(yt2_im

pd.DataFrame({
    "Train Accuracy": [lda_train_acc, logreg_train_acc, svc_train_acc, mlp_train_ac
    "Test Accuracy": [lda_test_acc, logreg_test_acc, svc_test_acc, mlp_test_acc, cn
    "Train F1": [lda_train_f1, logreg_train_f1, svc_train_f1, mlp_train_f1, cnn_tra
    "Test F1": [lda_test_f1, logreg_test_f1, svc_test_f1, mlp_test_f1, cnn_test_f1]
    "Train Precision": [lda_train_prec, logreg_train_prec, svc_train_prec, mlp_trai
    "Test Precision": [lda_test_prec, logreg_test_prec, svc_test_prec, mlp_test_pre
    "Train Recall": [lda_train_rec, logreg_train_rec, svc_train_rec, mlp_train_rec,
    "Test Recall": [lda_test_rec, logreg_test_rec, svc_test_rec, mlp_test_rec, cnn_
}, index=["LDA", "LogReg", "SVC", "MLP", "CNN"]).round(2).style.highlight_max(color
```

Out[ ]:

	Train Accuracy	Test Accuracy	Train F1	Test F1	Train Precision	Test Precision	Train Recall	Test Recall
LDA	0.960000	0.950000	0.960000	0.950000	0.960000	0.950000	0.960000	0.950000
LogReg	1.000000	0.970000	1.000000	0.970000	1.000000	0.970000	1.000000	0.970000
SVC	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000
MLP	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000	1.000000	0.980000
CNN	1.000000	0.990000	1.000000	0.990000	1.000000	0.990000	1.000000	0.990000

In [ ]:
```
predicted_labels_remaped = [rev_map[int(i.item())] for i in predicted_labels]
```

In [ ]:
```
print(classification_report(yt2, predicted_labels_remaped))
```

```
              precision    recall  f1-score   support

           1       0.99      0.99      0.99       500
           7       1.00      0.97      0.98       500
           8       0.98      1.00      0.99       500

    accuracy                           0.99      1500
   macro avg       0.99      0.99      0.99      1500
weighted avg       0.99      0.99      0.99      1500
```

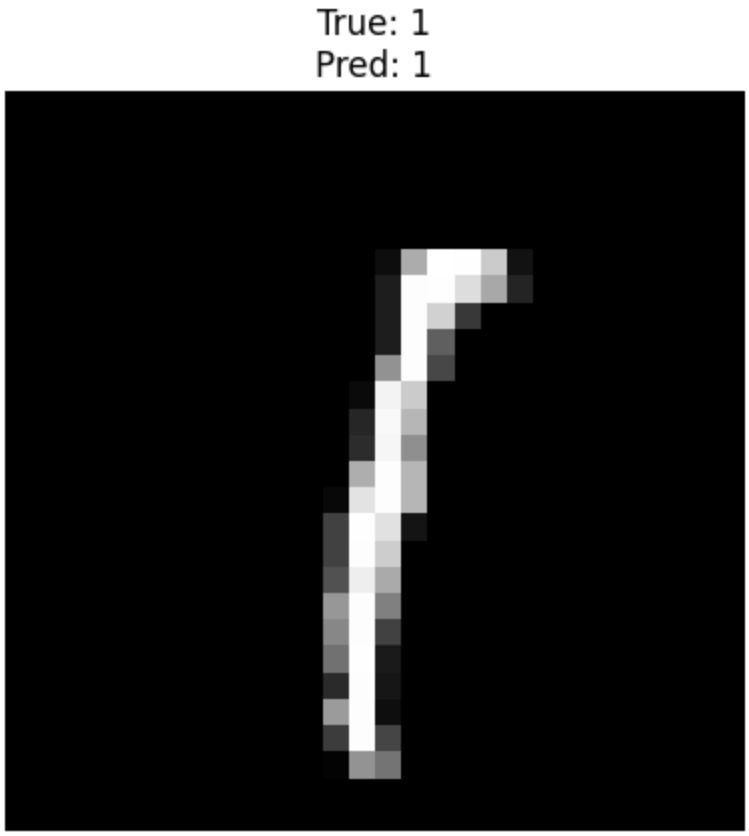
After some testing on the models parameters, the above architecture gave the best results and beats the previous models in terms of performances (accuracy-wise).

4.1.2. Adversarial examples

Now that we have a satisfactory model, let's test its robustness to adversarial examples. We will use the same method as before (adding gaussian noise to the sample) :

In [ ]:
```
np.random.seed(my_seed)
well_class_index = np.random.choice([i for i in range(len(yt2)) if yt2_im[i] == pre
```

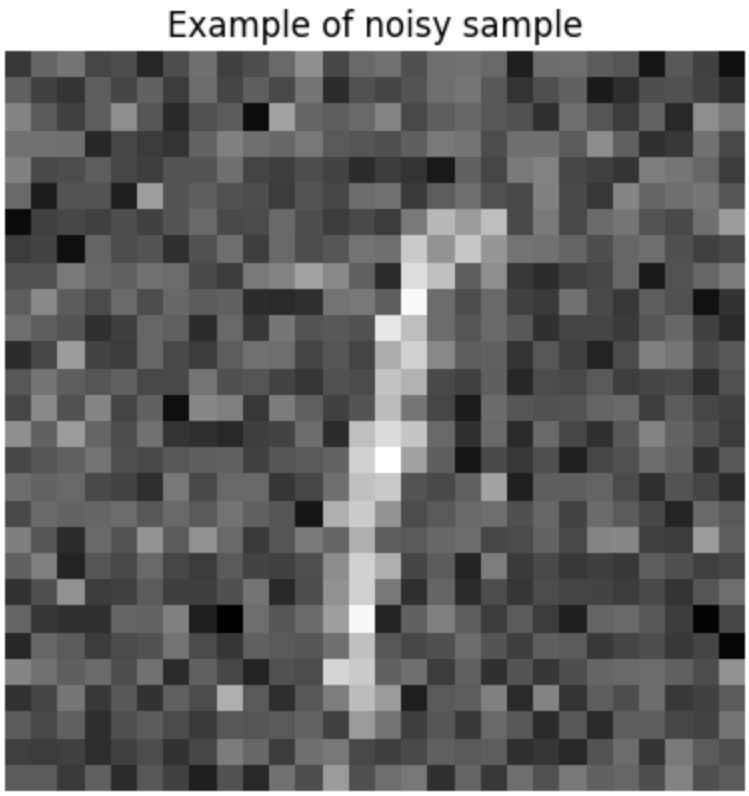
In [ ]:
```
true_class = rev_map[int(yt2_im[well_class_index][0].item())]
plt.imshow(xt2_im[well_class_index][0], cmap="gray")
plt.title("True: "+str(rev_map[int(yt2_im[well_class_index][0].item())])+"\nPred: "
plt.axis("off")
plt.show()
```



In [ ]:
```
n_noisy = 1000
noise_factor = 0.25
noisy_samples = []

for i in range(n_noisy):
    np.random.seed(my_seed)
    noisy_samples.append(xt2[well_class_index].reshape(28, 28) + noise_factor * np.
noisy_samples = torch.from_numpy(np.array(noisy_samples).reshape(-1, 28, 28))
```

In [ ]:
```
plt.imshow(noisy_samples[0].reshape(28, 28), cmap="gray")
plt.title("Example of noisy sample")
plt.axis("off")
plt.show()
```



In [ ]:
```
with torch.no_grad():
    test_outputs = model(noisy_samples.unsqueeze(1).float())

predicted_noisy_labels = torch.argmax(test_outputs, dim=1)

cnn_acc_noisy = accuracy_score([map[true_class] for i in range(len(predicted_noisy_
print(f"Accuracy on noisy samples : {str(cnn_acc_noisy*100)}%")
```

```
Accuracy on noisy samples : 100.0%
```

In [ ]:
```
predicted_noisy_labels
```

```
Out[ ]: tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

We see that the convolutionnal neural network seems to be more robust to adversarial examples than the SVC model. In fact, all samples are well classified and this is the case for many seeds that have been tryed.

To conclude, even if we didn't find adversarial examples for this model, it doesn't mean that there isn't. With more time put in we could probably find some adversarial examples for this model.

5. Discussion

After this 3rd practical session, I feel like I have aquired a broaden knowledge of machine learning. Fine-tuning models, choosing models and metrics depending on the task and comparing them are probably the most crucial aspects of Machine Learning and also the hardest ones in my opinon. I espacially appreciated the bonus part on CNN and adversarial examples as I was able to find an example that fooled the model. Comparing models and metrics felt like a real world application which I very much enjoyed despite how tedious it was.

This last practical session was my favorite, in terms of new knowledges, topics and applicability to the real word.

As a last word, this course genuinely helped me to understand the actual mechanisms behind machine learning. I feel much more confident in my ability to use machine learning in the future.