DAKO Spring 2022 - Team Work

Group 2: Markus Ijäs, Susanna Mikola, Markus Murto and Joonas Mäki

April 24, 2022

1 Description

The aim of this group work is to use machine learning to create a model predicting which passengers survived the Titanic shipwreck. The data used is from Kaggle Titanic ML Competition, and Kaggle is used to evaluate our models as well. There is also additional tasks which are completed during this group work.

Team Work Group 2: Markus Ijäs, Susanna Mikola, Markus Murto and Joonas Mäki.

Our aim for the model is to get over 80 % score on our eventual Kaggle submit.

2 Imports and definitions

First, we need to import and define dependencies for our notebook.

[2]:
```
import os

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.metrics import (accuracy_score, classification_report,
                             confusion_matrix, precision_score, recall_score)
from sklearn.model_selection import (GridSearchCV, cross_val_score,
                                     train_test_split)
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

[3]:
```
DATA_PATH = os.path.join("data")
```

[4]:
```
def load_csv_data(data_directory, csv_filename):
    '''Read csv data from data_directory/csv_filename'''
    csv_path = os.path.join(data_directory, csv_filename)
    return pd.read_csv(csv_path)
```

3 Loading data and taking a look

We are now ready to load the data and take a look at it.

[5]:
```
train = load_csv_data(DATA_PATH, "train.csv")
test = load_csv_data(DATA_PATH, "test.csv")
```

[6]:
```
train
```

[6]:
```
     PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
3              4         1       1
4              5         0       3
..           …         …     …
886          887         0       2
887          888         1       1
888          889         0       3
889          890         1       1
890          891         0       3

                                                  Name     Sex   Age  SibSp  \
0                              Braund, Mr. Owen Harris    male  22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0      1
2                               Heikkinen, Miss. Laina  female  26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                             Allen, Mr. William Henry    male  35.0      0
..                                                 …     …   …    …
886                              Montvila, Rev. Juozas    male  27.0      0
887                       Graham, Miss. Margaret Edith  female  19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female   NaN      1
889                              Behr, Mr. Karl Howell    male  26.0      0
890                                Dooley, Mr. Patrick    male  32.0      0

     Parch            Ticket     Fare Cabin Embarked
0        0         A/5 21171   7.2500   NaN        S
1        0          PC 17599  71.2833   C85        C
2        0  STON/O2. 3101282   7.9250   NaN        S
3        0            113803  53.1000  C123        S
4        0            373450   8.0500   NaN        S
..     …               …      …   …      …
886      0            211536  13.0000   NaN        S
887      0            112053  30.0000   B42        S
888      2        W./C. 6607  23.4500   NaN        S
889      0            111369  30.0000  C148        C
890      0            370376   7.7500   NaN        Q
```

```
[891 rows x 12 columns]
```

[7]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

[8]:
```
train['Embarked'].value_counts()
```

[8]:
```
S    644
C    168
Q     77
Name: Embarked, dtype: int64
```

[9]:
```
test
```

[9]:
```
     PassengerId  Pclass                                          Name  \
0            892       3                              Kelly, Mr. James
1            893       3              Wilkes, Mrs. James (Ellen Needs)
2            894       2                     Myles, Mr. Thomas Francis
3            895       3                              Wirz, Mr. Albert
4            896       3  Hirvonen, Mrs. Alexander (Helga E Lindqvist)
..           …     …                                            …
413         1305       3                            Spector, Mr. Woolf
414         1306       1                  Oliva y Ocana, Dona. Fermina
415         1307       3                  Saether, Mr. Simon Sivertsen
416         1308       3                           Ware, Mr. Frederick
417         1309       3                      Peter, Master. Michael J

        Sex   Age  SibSp  Parch              Ticket      Fare Cabin Embarked
```

```
0      male  34.5      0      0               330911     7.8292   NaN         Q
1    female  47.0      1      0               363272     7.0000   NaN         S
2      male  62.0      0      0               240276     9.6875   NaN         Q
3      male  27.0      0      0               315154     8.6625   NaN         S
4    female  22.0      1      1              3101298    12.2875   NaN         S
..      …    …     …     …                    …        …     …      …
413    male   NaN      0      0            A.5. 3236     8.0500   NaN         S
414  female  39.0      0      0             PC 17758   108.9000  C105         C
415    male  38.5      0      0   SOTON/O.Q. 3101262     7.2500   NaN         S
416    male   NaN      0      0               359309     8.0500   NaN         S
417    male   NaN      1      1                 2668    22.3583   NaN         C

[418 rows x 11 columns]
```

[10]:
```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  418 non-null    int64
 1   Pclass       418 non-null    int64
 2   Name         418 non-null    object
 3   Sex          418 non-null    object
 4   Age          332 non-null    float64
 5   SibSp        418 non-null    int64
 6   Parch        418 non-null    int64
 7   Ticket       418 non-null    object
 8   Fare         417 non-null    float64
 9   Cabin        91 non-null     object
 10  Embarked     418 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 36.0+ KB
```

All seems to be fine.

4 Describing data

Basic describe with mean, min etc.

[11]:
```
train.describe()
```

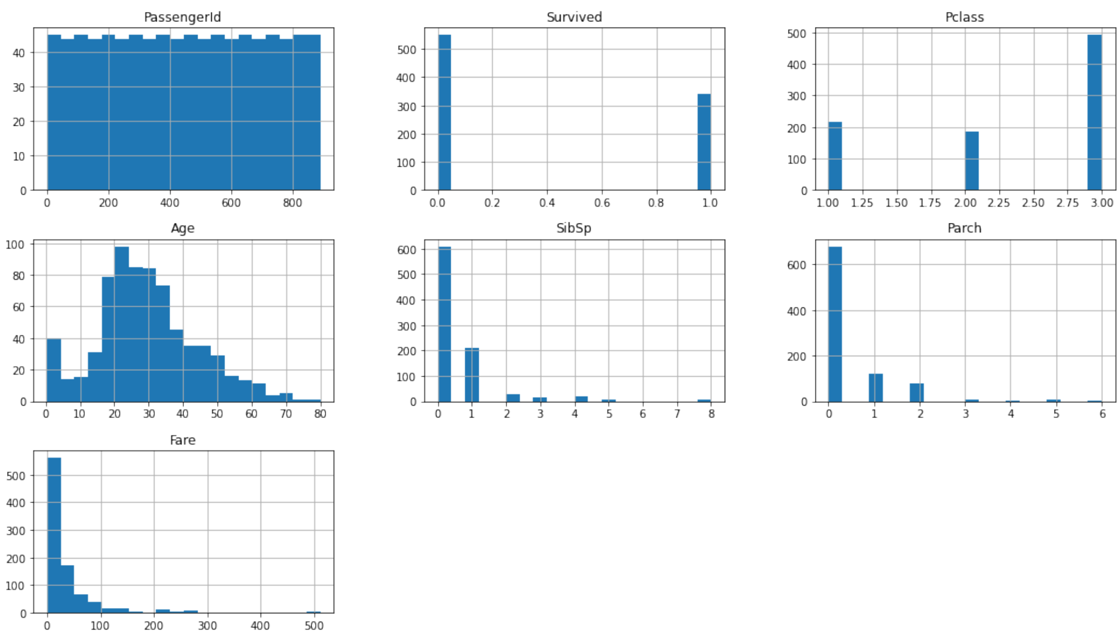
[11]:
```
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
```

```
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

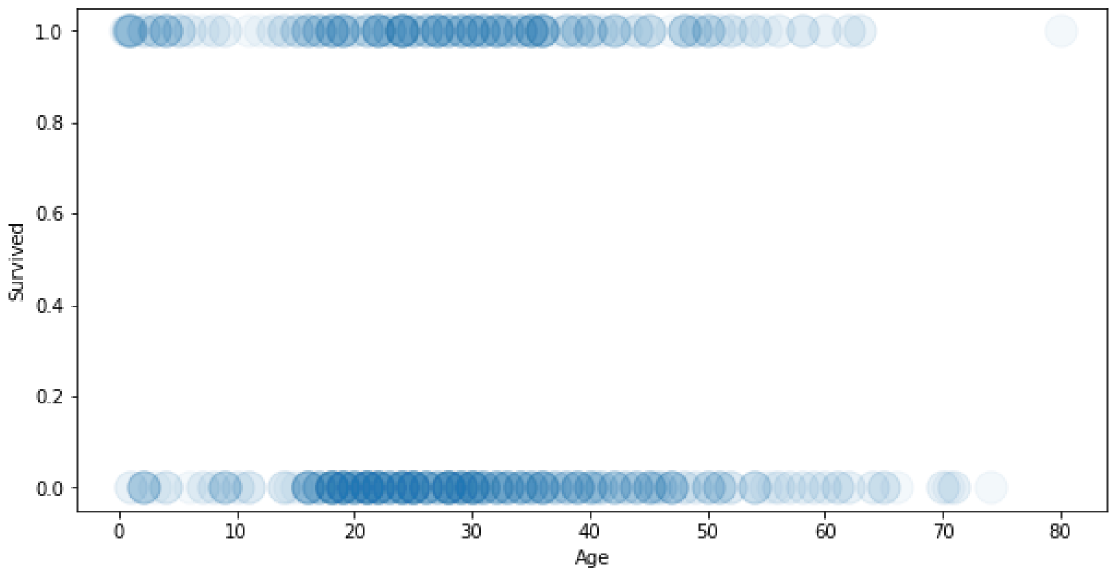
Interestingly largely passangers were quite young on the Titanic. Many were also travelling without siblings or parents/children. Unfortunately we can also see that most of the passangers did not survive. Let's look at the distribution of data in different columns next using histograms.

[12]:
```python
train.hist(bins=20, figsize=(18,10));
```



Again we can see that there were many fairly young people. It could be interesting to see, how the distribution of age between passangers who survived and who did not.

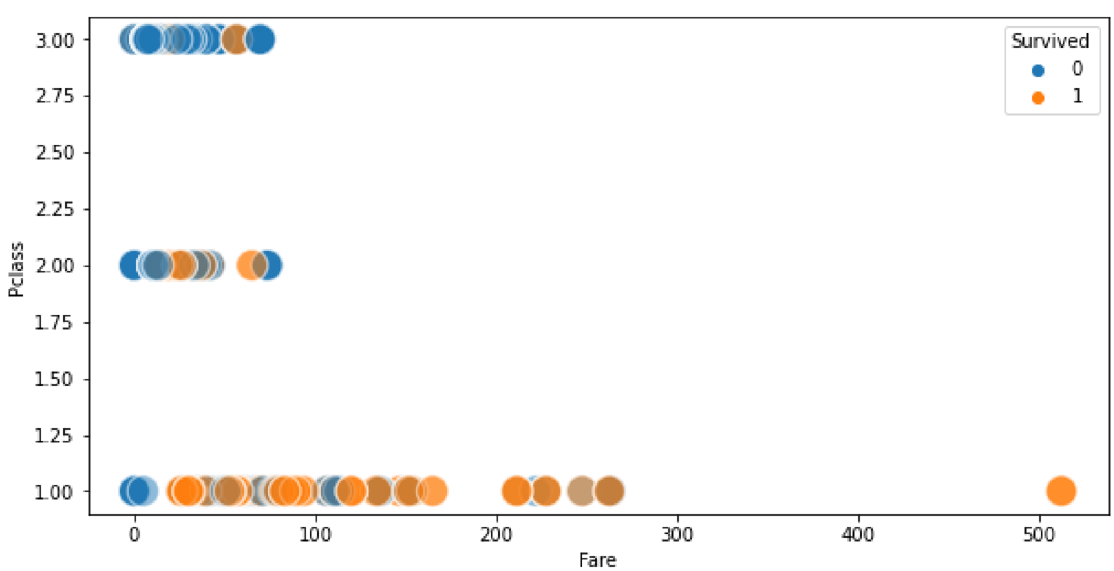
[13]:
```python
train.plot(kind="scatter", x="Age", y="Survived", alpha=0.05, s=300,␣
↪figsize=(10,5));
```



The youngsters might have been slightly more nimble to run towards the lifeboats, but you cannot see any strong signs of correlation.

Could you though buy yourself a survival?

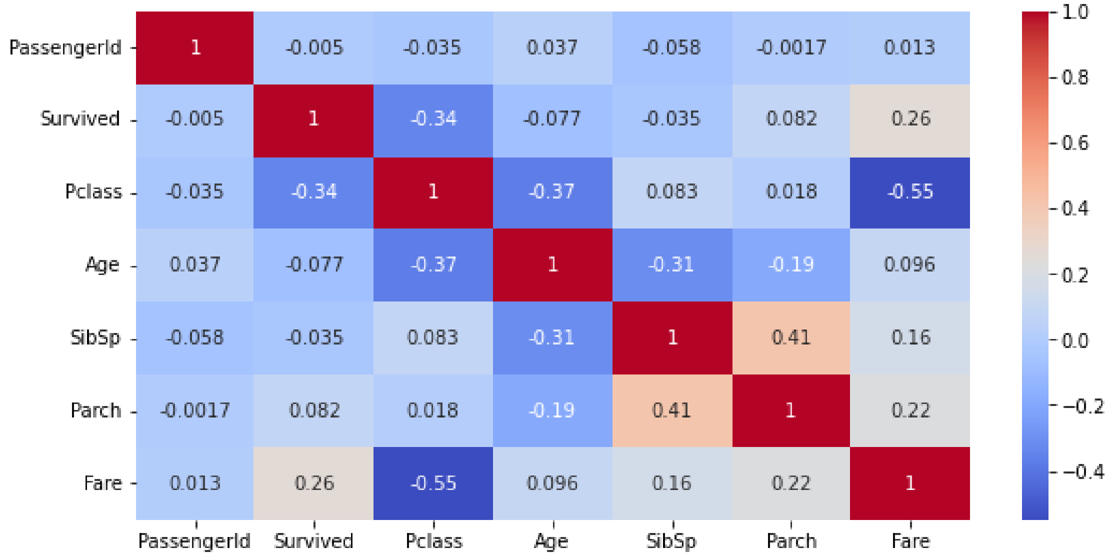
[14]:
```
plt.figure(figsize=(10,5))
sns.scatterplot(data=train, x="Fare", y="Pclass", hue="Survived", s=300,␣
↪alpha=0.5);
```



Maybe? It certainly looks like passangers in the first class had more survivors (orange colour) among them. Data points representing passangers in the third class do look very blue.

Lets properly calculate and visualise the Pearsons correlation coefficiency.

[15]:
```
plt.figure(figsize=(10,5))
sns.heatmap(train.corr(method="pearson"), annot=True, cmap="coolwarm");
```



Here cooler colours represent negative correlation and more warm colours represent positive correlation. Now we can see that being in higher class (lower numerical value), and also higher fare correlates pretty substantially with survival.

5 Preparing data

5.1 Building pipelines

For numeric columns we built a pipeline which replaces missing values with mean value for that column, and scales all the values with `StandardScaler`.

[16]:
```
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="mean")),
    ('std_scaler', StandardScaler()),
])
```

Non-numeric columns can't be prepared with the pipeline above, therefore we built a special pipeline for handling them. The pipeline first replaces missing data with most frequent data, then encoding non-binary categories with `OneHotEncoder`.

[17]:
```
cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="most_frequent")),
    ('onehotenc', OneHotEncoder()),
```

```
])
```

5.2 Selecting data and running pipelines

There were a few columns we decided to drop entirely, based on their perceived unusability. Dropped columns are:

- Cabin (a lot of data missing)
- Ticket (doesn't seem meaningful for us)
- Name (we at least hope that name is not a significant factor on survival)

This leaves us with numeric columns of:

- Age
- Number of siblings (*SibSp*)
- Number of parents (*Parch*)
- Fare

And category columns of:

- Ticket class (*Pclass*) (this is a category with possible classes: 1, 2 and 3)
- Sex
- Embarked (the port of embarkation)

[18]:
```
num_attribs = ["Age", "SibSp", "Parch", "Fare"]
cat_attribs = ["Pclass", "Sex", "Embarked"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])

train_prepared = full_pipeline.fit_transform(train)
test_prepared = full_pipeline.fit_transform(test)
```

Let's take a look…

[19]:
```
train_prepared[0]
```

[19]:
```
array([-0.5924806 ,  0.43279337, -0.47367361, -0.50244517,  0.        ,
        0.        ,  1.        ,  0.        ,  1.        ,  0.        ,
        0.        ,  1.        ])
```

[20]:
```
test_prepared[0]
```

[20]:
```
array([ 0.3349926 , -0.49947002, -0.4002477 , -0.49840706,  0.        ,
        0.        ,  1.        ,  0.        ,  1.        ,  0.        ,
        1.        ,  0.        ])
```

Seems OK.

6 Random Forest

Using Random Forest as our first machine learning model.

Creating labels for the data. Naming them as rf_ indicating that the usage is for Random Forest.

[21]:
```
rf_labels = train['Survived']
rf_labels
```

[21]:
```
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

Splitting the data into training and test sets to evaluate our model.

[22]:
```
rf_features_train, rf_features_test, rf_labels_train, rf_labels_test =␣
 ↪train_test_split(train_prepared, rf_labels, test_size=0.25, random_state = 0)

# Checking the shape
print('Training Features Shape:', rf_features_train.shape)
print('Training Labels Shape:', rf_labels_train.shape)
print('Testing Features Shape:', rf_features_test.shape)
print('Testing Labels Shape:', rf_labels_test.shape)
```

```
Training Features Shape: (668, 12)
Training Labels Shape: (668,)
Testing Features Shape: (223, 12)
Testing Labels Shape: (223,)
```

Next instantiating the model with 100 decision trees.

[23]:
```
rf = RandomForestClassifier(random_state=42)
```

Then training the model with training data.

[24]:
```
rf.fit(rf_features_train, rf_labels_train)
```

[24]:
```
RandomForestClassifier(random_state=42)
```

Next we are using predict method on the test data.

[25]:
```
rf_predictions_test = rf.predict(rf_features_test)
rf_predictions_test
```

[25]:
```
array([0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0,
       1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
       0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,
       0, 1, 1])
```

Lastly checking the accuracy score.

[26]:
```
accuracy_score(rf_labels_test, rf_predictions_test)
```

[26]:
```
0.8385650224215246
```

The classification score for Random Forest is ok, but could be better.

7 Neural Network as an alternative

We selected MultiLayer Perceptron as our second machine learning model.

First we need to get labels for the data. We will also name the variable `alt_*` to keep them specific to this “alternative model”.

[27]:
```
alt_labels = train['Survived']
# Let's also take a look just to be sure
alt_labels
```

[27]:
```
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

And split the data in to training and test sets so we can actually evaluate our model. Remember the original “test” data doesn’t contain the Survived column, which means we can’t use it in actual evaluation. It's for when we want to train and predict the best of our models for Kaggle competition.

[28]:
```
alt_features_train, alt_features_test, alt_labels_train, alt_labels_test = train_test_split(train_prepared, alt_labels, test_size=0.25, random_state = 0)
alt_features_train
```

[28]:
```
array([[-0.1307545 , -0.4745452 , -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       [-0.97725235,  3.15480905,  2.00893337, …,  0.        ,
         0.        ,  1.        ],
       [ 0.02315421,  0.43279337, -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       …,
       [ 0.        , -0.4745452 , -0.47367361, …,  0.        ,
         1.        ,  0.        ],
       [ 0.48488031,  0.43279337, -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       [ 2.33178473,  0.43279337,  0.76762988, …,  0.        ,
         0.        ,  1.        ]])
```

Seems good. Let's next instantiate the perceptron with just the default values, just to see what happens.

[29]:
```
mlp = MLPClassifier(random_state=42)
```

And let's try to fit the model…

[30]:
```
mlp.fit(alt_features_train, alt_labels_train)
```

```
/opt/conda/lib/python3.9/site-packages/sklearn/neural_network/_multilayer_perceptron.py:692:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

[30]:
```
MLPClassifier(random_state=42)
```

That didn't go as expected. We probably need to increase maximum iteration count significantly. Let's try 2000 (the default was 200).

[31]:
```
mlp = MLPClassifier(max_iter=2000, random_state=42)
mlp.fit(alt_features_train, alt_labels_train)
```

[31]:
```
MLPClassifier(max_iter=2000, random_state=42)
```

Yay, it converged! Let's try to predict without tuning anything more. Again just to see what happens.

[32]:
```
alt_predictions_test = mlp.predict(alt_features_test)
alt_predictions_test
```

[32]:
```
array([0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1,
       0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,
       0, 1, 0])
```

And let's evaluate the results with a confusion matrix and classification score.

[33]:
```
confusion_matrix(alt_labels_test, alt_predictions_test)
```

[33]:
```
array([[125,  14],
       [ 30,  54]])
```

[34]:
```
accuracy_score(alt_labels_test, alt_predictions_test)
```

[34]:
```
0.8026905829596412
```

Not that good of an accuracy. That's expected though knowing our neural network consists of just one level of 200 hidden neurons.

8 Selecting one of the models for finer tuning

Let's first calculate accuracy scores for our models above

[35]:
```
rf_predictions_train = rf.predict(rf_features_train)
alt_predictions_train = mlp.predict(alt_features_train)

rf_train_accuracy = accuracy_score(rf_labels_train, rf_predictions_train)
rf_test_accuracy = accuracy_score(rf_labels_test, rf_predictions_test)
alt_train_accuracy = accuracy_score(alt_labels_train, alt_predictions_train)
alt_test_accuracy = accuracy_score(alt_labels_test, alt_predictions_test)

print(f'Random Forest accuracies: training {rf_train_accuracy}, test␣
 ↪{rf_test_accuracy}')
print(f'Multi-Layer Perceptron (our alternative) accuracies: training␣
 ↪{alt_train_accuracy}, test {alt_test_accuracy}')
```

```
Random Forest accuracies: training 0.9805389221556886, test 0.8385650224215246
Multi-Layer Perceptron (our alternative) accuracies: training
```

0.8847305389221557, test 0.8026905829596412

A bigger difference in training and testing accuracy for Random Forest than for MLP. This would suggest MLP would be better.

Then let's calculate the precision scores

[36]:
```
rf_train_precision = precision_score(rf_labels_train, rf_predictions_train)
rf_test_precision = precision_score(rf_labels_test, rf_predictions_test)
alt_train_precision = precision_score(alt_labels_train, alt_predictions_train)
alt_test_precision = precision_score(alt_labels_test, alt_predictions_test)

print(f'Random Forest precisions: training {rf_train_precision}, test␣
↪{rf_test_precision}')
print(f'Multi-Layer Perceptron (our alternative) precisions: training␣
↪{alt_train_precision}, test {alt_test_precision}')
```

```
Random Forest precisions: training 0.9880478087649402, test 0.8157894736842105
Multi-Layer Perceptron (our alternative) precisions: training 0.91324200913242,
test 0.7941176470588235
```

As above, MLP seems to win again.

And finally calculate the recall scores

[37]:
```
rf_train_recall = recall_score(rf_labels_train, rf_predictions_train)
rf_test_recall = recall_score(rf_labels_test, rf_predictions_test)
alt_train_recall = recall_score(alt_labels_train, alt_predictions_train)
alt_test_recall = recall_score(alt_labels_test, alt_predictions_test)

print(f'Random Forest recalls: training {rf_train_recall}, test␣
↪{rf_test_recall}')
print(f'Multi-Layer Perceptron (our alternative) recalls: training␣
↪{alt_train_recall}, test {alt_test_recall}')
```

```
Random Forest recalls: training 0.9612403100775194, test 0.7380952380952381
Multi-Layer Perceptron (our alternative) recalls: training 0.7751937984496124,
test 0.6428571428571429
```

Overfitting in Random Forest once again. MLP, while having quite poor results as well, doesn't overfit so much.

Next, let's take a look at cross valuation scoring of both models.

[38]:
```
rf_scores = cross_val_score(rf, rf_features_train, rf_labels_train, cv=10)
rf_scores.mean()
```

[38]:
```
0.799502487562189
```

[39]:
```
alt_scores = cross_val_score(mlp, alt_features_train, alt_labels_train, cv=10)
alt_scores.mean()
```

[39]: 0.811420171867933

The mean for our alternative model (MLPClassifier) seems to be a bit higher. Let's select that for further fine-tuning.

9 Fine-tuning the selected model

Let's first define a tuning function for easier scoring different hyperparameter combinations with different scores

[40]:
```
def tuning_function():
    for score in scores:
        print("# Tuning hyper-parameters for %s" % score)
        print()

        clf = GridSearchCV(MLPClassifier(), tuned_parameters,␣
 ↪scoring="%s_macro" % score)
        clf.fit(features_train, labels_train)

        print("Best parameters set found on development set:")
        print()
        print(clf.best_params_)
        print()
        print("Grid scores on development set:")
        print()
        means = clf.cv_results_["mean_test_score"]
        stds = clf.cv_results_["std_test_score"]
        for mean, std, params in zip(means, stds, clf.cv_results_["params"]):
            print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))
        print()

        print("Detailed classification report:")
        print()
        print("The model is trained on the full development set.")
        print("The scores are computed on the full evaluation set.")
        print()
        y_true, y_pred = labels_test, clf.predict(features_test)
        print(classification_report(y_true, y_pred))
        print()
```

Split the data (this is mainly to get rid of any optimizations or things done earlier in this notebook)

[41]:
```
labels = train['Survived']
features_train, features_test, labels_train, labels_test =␣
 ↪train_test_split(train_prepared, labels, test_size=0.25, random_state = 0)
features_train
```

[41]:
```
array([[-0.1307545 , -0.4745452 , -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       [-0.97725235,  3.15480905,  2.00893337, …,  0.        ,
         0.        ,  1.        ],
       [ 0.02315421,  0.43279337, -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       …,
       [ 0.        , -0.4745452 , -0.47367361, …,  0.        ,
         1.        ,  0.        ],
       [ 0.48488031,  0.43279337, -0.47367361, …,  0.        ,
         0.        ,  1.        ],
       [ 2.33178473,  0.43279337,  0.76762988, …,  0.        ,
         0.        ,  1.        ]])
```

And try some values for fine-tuning

[42]:
```
tuned_parameters = [
    {
        "random_state": [42],
        "early_stopping": [True],
        "activation": ["relu"],
        "max_iter": [2000],
        "solver": ["adam"],
        "hidden_layer_sizes": [
            (100,),
            (100,10),
            (100, 50, 25),
            (200, 100, 50),
            (200, 100, 50, 25)
        ]
    },
    {
        "random_state": [42],
        "early_stopping": [True],
        "activation": ["relu"],
        "max_iter": [20000],
        "solver": ["lbfgs"],
        "hidden_layer_sizes": [
            (100,),
            (100,10),
            (100, 50, 25),
            (200, 100, 50),
            (200, 100, 50, 25)
        ]
    },
]
```

```python
scores = ["precision", "recall"]

tuning_function()
```

```
# Tuning hyper-parameters for precision

Best parameters set found on development set:

{'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100,),
'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}

Grid scores on development set:

0.832 (+/-0.020) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100,), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.800 (+/-0.020) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 10), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.805 (+/-0.027) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.817 (+/-0.019) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.804 (+/-0.044) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.765 (+/-0.058) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100,), 'max_iter': 20000, 'random_state': 42, 'solver':
'lbfgs'}
0.775 (+/-0.047) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 10), 'max_iter': 20000, 'random_state': 42,
'solver': 'lbfgs'}
0.752 (+/-0.018) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 20000, 'random_state': 42,
'solver': 'lbfgs'}
0.758 (+/-0.059) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 20000, 'random_state': 42,
'solver': 'lbfgs'}
0.751 (+/-0.049) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 20000, 'random_state': 42,
'solver': 'lbfgs'}

Detailed classification report:

The model is trained on the full development set.
```

The scores are computed on the full evaluation set.

```
              precision    recall  f1-score   support

           0       0.78      0.91      0.84       139
           1       0.80      0.57      0.67        84

    accuracy                           0.78       223
   macro avg       0.79      0.74      0.75       223
weighted avg       0.79      0.78      0.78       223
```

Tuning hyper-parameters for recall

Best parameters set found on development set:

{'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}

Grid scores on development set:

0.799 (+/-0.044) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100,), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}
0.777 (+/-0.025) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100, 10), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}
0.785 (+/-0.043) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}
0.801 (+/-0.041) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}
0.785 (+/-0.042) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}
0.765 (+/-0.060) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100,), 'max_iter': 20000, 'random_state': 42, 'solver': 'lbfgs'}
0.773 (+/-0.054) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100, 10), 'max_iter': 20000, 'random_state': 42, 'solver': 'lbfgs'}
0.748 (+/-0.011) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (100, 50, 25), 'max_iter': 20000, 'random_state': 42, 'solver': 'lbfgs'}
0.755 (+/-0.053) for {'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100, 50), 'max_iter': 20000, 'random_state': 42, 'solver': 'lbfgs'}

```
0.749 (+/-0.043) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 20000, 'random_state': 42,
'solver': 'lbfgs'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.83      0.85      0.84       139
           1       0.74      0.71      0.73        84

    accuracy                           0.80       223
   macro avg       0.79      0.78      0.78       223
weighted avg       0.80      0.80      0.80       223
```

That took 218 seconds on i5-9600K stock, not too long it seems. Adam seems to perform better, so let's take a closer look with more hyperparameter variations

[43]:
```python
tuned_parameters = [
    {
        "random_state": [42],
        "early_stopping": [True],
        "activation": ["relu"],
        "max_iter": [2000],
        "solver": ["adam"],
        "hidden_layer_sizes": [
            (50,),
            (50, 25),
            (100, 50, 25),
            (200, 100, 50),
            (200, 150, 100),
            (200, 100, 50, 25),
            (200, 150, 100, 50),
            (50, 40, 30, 20, 10),
            (400, 200, 100, 50, 25),
        ]
    },
    {
        "random_state": [42],
        "early_stopping": [True],
        "activation": ["tanh"],
        "max_iter": [2000],
        "solver": ["adam"],
```

```
        "hidden_layer_sizes": [
            (50,),
            (50, 25),
            (100, 50, 25),
            (200, 100, 50),
            (200, 150, 100),
            (200, 100, 50, 25),
            (200, 150, 100, 50),
            (50, 40, 30, 20, 10),
            (400, 200, 100, 50, 25),
        ]
    },
]

scores = ["precision", "recall"]

tuning_function()
```

```
# Tuning hyper-parameters for precision

Best parameters set found on development set:

{'activation': 'tanh', 'early_stopping': True, 'hidden_layer_sizes': (50, 40,
30, 20, 10), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}

Grid scores on development set:

0.699 (+/-0.195) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50,), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.807 (+/-0.043) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50, 25), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.805 (+/-0.027) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.817 (+/-0.019) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.813 (+/-0.029) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.804 (+/-0.044) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.812 (+/-0.022) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100, 50), 'max_iter': 2000, 'random_state': 42,
```

```
'solver': 'adam'}
0.811 (+/-0.035) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50, 40, 30, 20, 10), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.808 (+/-0.062) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (400, 200, 100, 50, 25), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.758 (+/-0.059) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50,), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.798 (+/-0.046) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50, 25), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.801 (+/-0.038) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.798 (+/-0.049) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.790 (+/-0.047) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.782 (+/-0.023) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.814 (+/-0.035) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.819 (+/-0.033) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50, 40, 30, 20, 10), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.798 (+/-0.042) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (400, 200, 100, 50, 25), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.77      0.91      0.83       139
           1       0.78      0.56      0.65        84

    accuracy                           0.78       223
   macro avg       0.78      0.73      0.74       223
weighted avg       0.78      0.78      0.77       223
```

```
# Tuning hyper-parameters for recall

Best parameters set found on development set:

{'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100,
50), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}

Grid scores on development set:

0.649 (+/-0.166) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50,), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.763 (+/-0.093) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50, 25), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.785 (+/-0.043) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.801 (+/-0.041) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.795 (+/-0.036) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.785 (+/-0.042) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.792 (+/-0.041) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.777 (+/-0.037) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (50, 40, 30, 20, 10), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.788 (+/-0.067) for {'activation': 'relu', 'early_stopping': True,
'hidden_layer_sizes': (400, 200, 100, 50, 25), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.721 (+/-0.084) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50,), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.780 (+/-0.057) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50, 25), 'max_iter': 2000, 'random_state': 42, 'solver':
'adam'}
0.781 (+/-0.043) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.780 (+/-0.049) for {'activation': 'tanh', 'early_stopping': True,
```

```
'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.777 (+/-0.042) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.769 (+/-0.035) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 100, 50, 25), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.789 (+/-0.040) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (200, 150, 100, 50), 'max_iter': 2000, 'random_state': 42,
'solver': 'adam'}
0.748 (+/-0.072) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (50, 40, 30, 20, 10), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}
0.781 (+/-0.054) for {'activation': 'tanh', 'early_stopping': True,
'hidden_layer_sizes': (400, 200, 100, 50, 25), 'max_iter': 2000, 'random_state':
42, 'solver': 'adam'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.83      0.85      0.84       139
           1       0.74      0.71      0.73        84

    accuracy                           0.80       223
   macro avg       0.79      0.78      0.78       223
weighted avg       0.80      0.80      0.80       223
```

Seems like we found a suitable candidate for cross-validation: `{'activation': 'relu', 'early_stopping': True, 'hidden_layer_sizes': (200, 100, 50), 'max_iter': 2000, 'random_state': 42, 'solver': 'adam'}`

10 Cross-validating the fine tuned model

Let's first instantiate a new MLPClassifier with the found hyperparameter set

[44]:
```python
tuned_mlp = MLPClassifier(
    activation = "relu",
    early_stopping = True,
    hidden_layer_sizes = (200, 100, 50),
    max_iter = 2000,
    random_state = 42,
```

```
    solver = 'adam'
)
```

And then calculate the scores

[45]:
```
scores = cross_val_score(tuned_mlp, features_train, labels_train, cv=10)
scores
```

[45]:
```
array([0.76119403, 0.8358209 , 0.82089552, 0.80597015, 0.79104478,
       0.80597015, 0.7761194 , 0.85074627, 0.87878788, 0.81818182])
```

[46]:
```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(),␣
 ↪scores.std()))
```

```
0.81 accuracy with a standard deviation of 0.03
```

Not good, not terrible. At least it's over 50 % so it's likely not completely random.

11 Making predictions

Since we don't know how to perform any better, let's just make some predictions.

[47]:
```
tuned_mlp.fit(train_prepared, train['Survived'])
```

[47]:
```
MLPClassifier(early_stopping=True, hidden_layer_sizes=(200, 100, 50),
              max_iter=2000, random_state=42)
```

[48]:
```
predicted = tuned_mlp.predict(test_prepared)
predicted
```

[48]:
```
array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0,
       1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1,
       1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0,
       0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
       1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,
       0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,
       1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
       1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0,
       0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0,
       1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
```

```
        0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0])
```

And combine the predicted data with ids from the test dataset

```
[49]: final_data = np.stack((test["PassengerId"], predicted), axis=1)
      df = pd.DataFrame(data=final_data, columns=["PassengerId", "Survived"])
      df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  418 non-null    int64
 1   Survived     418 non-null    int64
dtypes: int64(2)
memory usage: 6.7 KB
```

```
[50]: df.to_csv(DATA_PATH + "/submission.csv", index=False)
```

And that's it. Exported data is found under the data folder as `submission.csv`.

12 How well did it all work out?

We got score of `0.76555` when we uploaded our submission to Kaggle, which means our model predicted Titanic survival correctly for 77 % of people. It doesn't look bad, but it certainly could be better. This shows that we weren't quite that successful in determining the hyperparameters for the MLPClassifier, and that we might have gotten better results with some other kind of machine learning technique.

Our original target was somewhere above 80 %, and even though we didn't quite reach it, we feel to have built a solid foundation for learning more on fine-tuning hyperparameters and selecting a proper model for the task at hand. On the positive side, we got everything done and were able to actually submit our results to Kaggle, and we now have much better understanding on how to complete a machine learning project in real life.