Building a machine learning pipeline - a case study

This notebook demonstrates how to build a machine learning pipeline using the open source machine learning library scikit-learn (https://scikit-learn.org) based on the famous kaggle Titanic dataset (https://www.kaggle.com/c/titanic).

Set up

Load modules

In [1]:
```
# data handling
import numpy as np
import pandas as pd

# visualisation
from matplotlib import pyplot as plt

# preprocessing
from sklearn.preprocessing import OrdinalEncoder

# classification algorithms
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

# model tuning tools
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import StackingClassifier

#local modules
from barplot import *
```

Set display options

In [2]:
```
# allow multiple outputs per cell
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# Plot the Figures Inline
%matplotlib inline

# Prevent label cut off from figures
from matplotlib import rcParams
rcParams.update({'figure.autolayout': True})
```

Data loader

In [3]:
```
# get metadata
meta_data = pd.read_csv("data/metadata.csv")
meta_data
```

Out[3]:

	Variable	Definition	Key
0	survival	Survival	0 = No 1 = Yes
1	pclass	Ticket class	1 = 1st 2 = 2nd 3 = 3rd
2	sex	Sex	NaN
3	Age	Age in years	NaN
4	sibsp	# of siblings / spouses aboard the Titanic	NaN
5	parch	# of parents / children aboard the Titanic	NaN
6	ticket	Ticket number	NaN
7	fare	Passenger fare	NaN
8	cabin	Cabin number	NaN
9	embarked	Port of Embarkation	C = Cherbourg Q = Queenstown S = Southampton

In [4]:
```
# load train data
train_data = pd.read_csv("data/titanic-train.csv")
print("Shape: ", train_data.shape)
train_data.head()
```

```
Shape:  (891, 12)
```

Out[4]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	0
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	1
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	1
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S	1
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S	0

In [5]:
```
# load test data
test_data = pd.read_csv("data/titanic-test.csv")
print("Shape: ", test_data.shape)
test_data.head()
```

```
Shape:  (418, 11)
```

Out[5]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

Data exploration

Check the frequency of classes in the predicted variable (label) in the training set

```
In [6]: num_deceased = (train_data["Survived"] == 0).sum()
        num_survived = (train_data["Survived"] == 1).sum()
        assert num_deceased + num_survived == 891
        print("deceased total: ", num_deceased, " - deceased %: ", round(100/891*num_deceased, 1))
        print("survived total: ", num_survived, " - deceased %: ", round(100/891*num_survived, 1))
```

```
deceased total:  549  - deceased %:  61.6
survived total:  342  - deceased %:  38.4
```

Conclusion: the number of fatalities is much higher than the number of survivors. This means that the dataset is imbalanced.

Check if the datasets contain missing values

In [7]:
```
missing_values = pd.DataFrame({'Training set': train_data.isna().sum(),
                               'Test set': test_data.isna().sum()})
missing_values
```

Out[7]:

	Training set	Test set
Age	177	86.0
Cabin	687	327.0
Embarked	2	0.0
Fare	0	1.0
Name	0	0.0
Parch	0	0.0
PassengerId	0	0.0
Pclass	0	0.0
Sex	0	0.0
SibSp	0	0.0
Survived	0	NaN
Ticket	0	0.0

Conclusion: There are many missing values for the age of passengers and the cabin type. Therefore, these features will be excluded from the following analyses.

Count the number of unique values of features of interest

```
In [8]: train_data["Sex"].nunique()
        train_data["SibSp"].nunique()
        train_data["Parch"].nunique()
        train_data["Fare"].nunique()
```

```
Out[8]: 2

Out[8]: 7

Out[8]: 7

Out[8]: 248
```

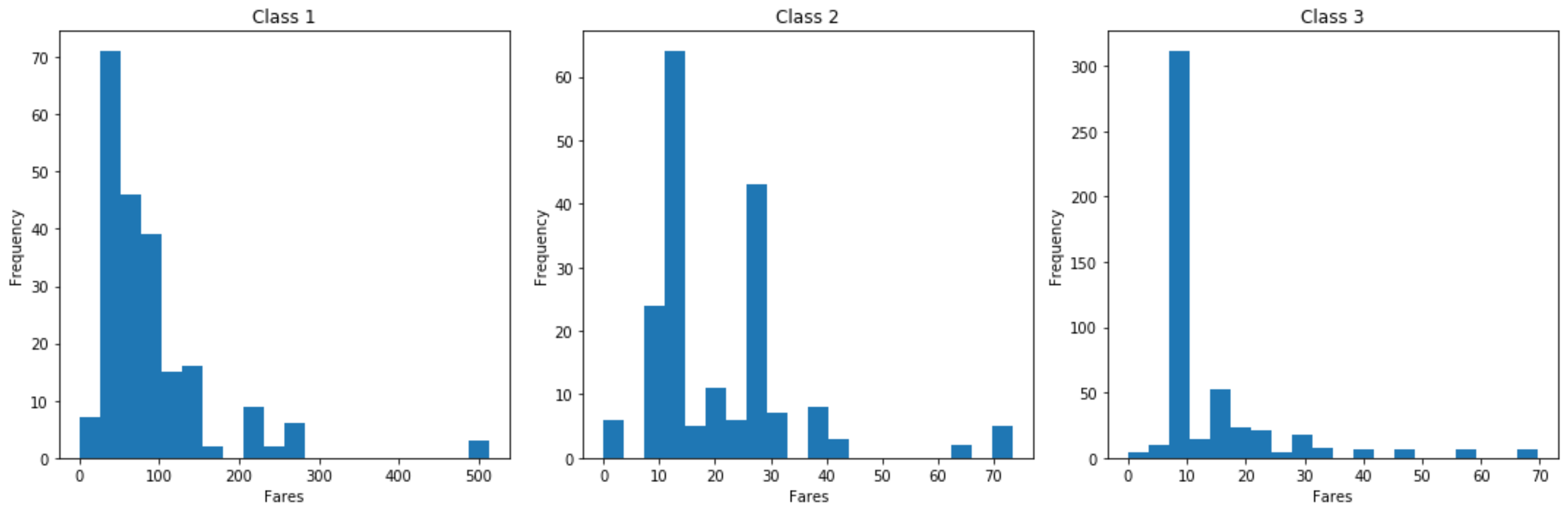
Conclusion: there are many different fares that are assumably associated with the ticket class. Let's check this:

Investigate fares

In [9]:
```
# check min and max prices of fares per class

# divide training dataset per class
class1 = train_data.loc[train_data['Pclass'] == 1]
class2 = train_data.loc[train_data['Pclass'] == 2]
class3 = train_data.loc[train_data['Pclass'] == 3]
# save classes in list
classes = [class1, class2, class3]

# print fare ranges
for i, pclass in enumerate(classes):
    print(f"Max fare class {i+1}: ", pclass["Fare"].max())
    print(f"Min fare class {i+1}: ",pclass["Fare"].min())
    print()
```

```
Max fare class 1:  512.3292
Min fare class 1:  0.0

Max fare class 2:  73.5
Min fare class 2:  0.0

Max fare class 3:  69.55
Min fare class 3:  0.0
```

In [10]:
```
# plot fares per class as histograms

# save fares in numpy array
fares_per_class = [class1["Fare"].to_numpy(),
                   class2["Fare"].to_numpy(),
                   class3["Fare"].to_numpy()]

# plot fares
fig, ax = plt.subplots(1,len(fares_per_class), figsize=(15, 5))
for i, data in enumerate(fares_per_class):
    _ = ax[i].hist(data, bins=20)
    _ = ax[i].set_title(f"Class {i+1}")
    _ = ax[i].set_xlabel("Fares")
    _ = ax[i].set_ylabel("Frequency")
```



Conclusion: The fares of the 3 different classes overlap, especially the fares of class 2 and 3. It might therefore be more useful to predict survival rates depending on passenger class rather than fare. Let's check among the categorical features if there are categories that are (strongly) associated with survival rate.

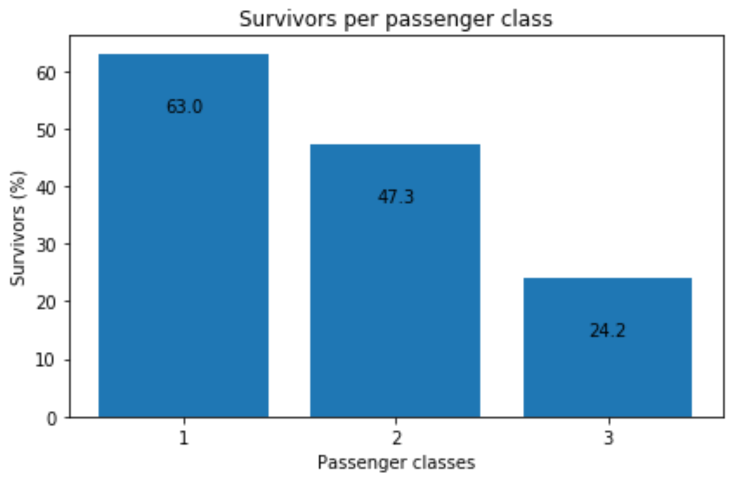
Investigate survival rates per categories

Passenger class:

In [11]:
```
# save categories in list and convert them to string variables for plotting
categories_class = list(map(str, train_data["Pclass"].unique()))
categories_class.sort()
categories_class # check result

# calculate percentage of survivors per passenger class
survivors_per_class = []
for pclass in classes:
    surv = round(pclass["Survived"].sum()/len(pclass["Survived"])*100, 1)
    survivors_per_class.append(surv)
```

Out[11]:
```
['1', '2', '3']
```

In [12]:
```
# plot survivors per class
plot_survivors_per_category(categories_class,
             survivors_per_class,
             title="Survivors per passenger class",
             xlabel="Passenger classes")
```



In [13]:
```
# compare result to total survivors per passenger class
train_data["Pclass"].value_counts()
```

Out[13]:
```
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

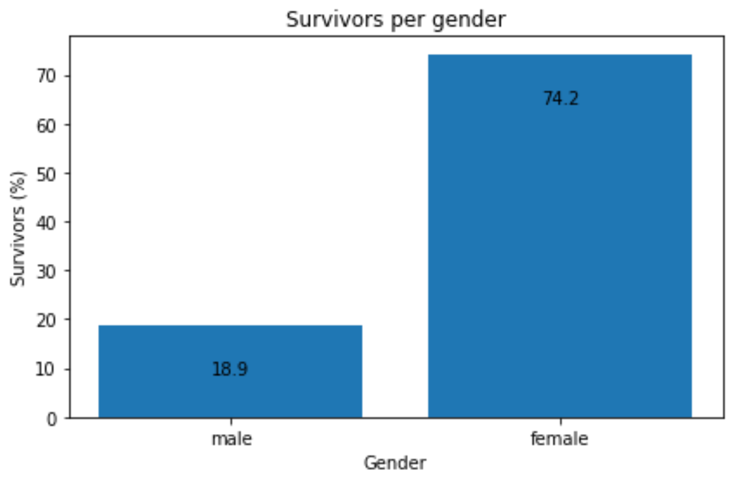
Conclusion: the survival rate seems to be correlated to the passenger class and therefore likely influences the prediction of survival. Although the survival rate is highest for passengers in class 1, most people travelled in class 3.

Gender:

In [14]:
```
# save categories in list
categories_gender = list(map(str, train_data["Sex"].unique()))
categories_gender # check result

# calculate percentage of survivors per gender
men = train_data.loc[train_data.Sex == 'male']["Survived"].to_numpy()
women = train_data.loc[train_data.Sex == 'female']["Survived"].to_numpy()
men_surv = round(sum(men)/len(men)*100, 1)
women_surv = round(sum(women)/len(women)*100, 1)

# store results in list
survivors_per_gender = [men_surv, women_surv]
```

Out[14]:
```
['male', 'female']
```

In [15]:
```
# plot survivors per gender
plot_survivors_per_category(categories_gender,
             survivors_per_gender,
             title="Survivors per gender",
             xlabel="Gender")
```



```
In [16]: # compare result to total survivors per gender
         train_data["Sex"].value_counts()
```

```
Out[16]: male      577
         female    314
         Name: Sex, dtype: int64
```

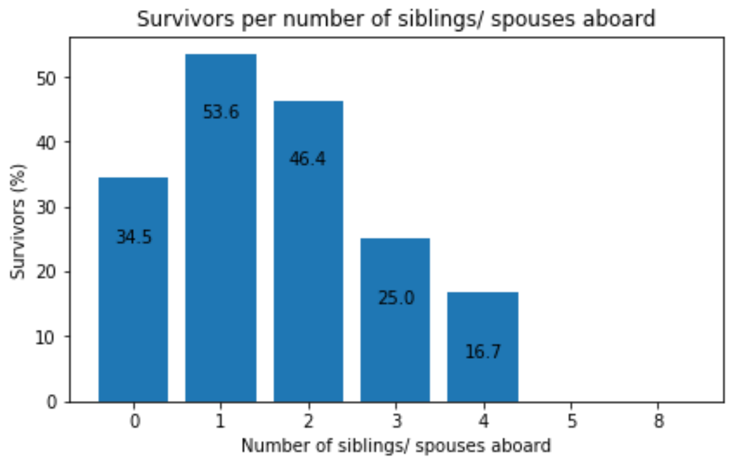
Conclusion: the survival rate of women is much higher than the survival rate of men although the number of men aboard the Titanic was much higher compared to the number of women. Therefore, the gender likely has a strong influence on the prediction of survival.

Number of siblings/ spouses aboard

```
In [17]: # save categories in list
         categories_sibsp = list(train_data["SibSp"].unique())
         categories_sibsp.sort()

         # calculate percentage of survivors per number of siblings/ spouses aboard
         # and save results in list
         survivors_per_sibsp = []
         for i in categories_sibsp:
             sibsp = train_data.loc[train_data.SibSp == i]["Survived"].to_numpy()
             survivors_per_sibsp.append(round(sum(sibsp)/len(sibsp)*100, 1))

         # convert categories to string variables for plotting
         categories_sibsp = list(map(str, categories_sibsp))
         categories_sibsp # check result
```

```
Out[17]: ['0', '1', '2', '3', '4', '5', '8']
```

In [18]:
```
# plot survivors per number of siblings/ spouses aboard
plot_survivors_per_category(categories_sibsp,
             survivors_per_sibsp,
             title="Survivors per number of siblings/ spouses aboard",
             xlabel="Number of siblings/ spouses aboard")
```



In [19]:
```
# compare result to total survivors per siblings/ spouses aboard
train_data["SibSp"].value_counts()
```

Out[19]:
```
0    608
1    209
2     28
4     18
3     16
8      7
5      5
Name: SibSp, dtype: int64
```

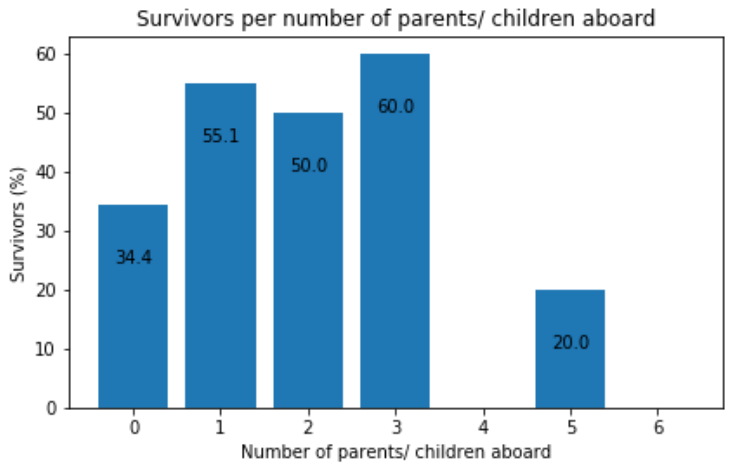
Conclusion: The people with 1 or 2 siblings/ spouses aboard had the highest rate of survival. This could mean that these people had support from family members with getting a spot in one of the lifeboats. However, the number of people who travelled with no siblings or spouses is more than twice as high as the number of people who travelled in company. Therefore, the number of siblings/ spouses might be weakly associated with the chance of survival.

Number of parents/ children aboard

In [20]:
```
# save categories in list
categories_parch = list(train_data["Parch"].unique())
categories_parch.sort()

# calculate percentage of survivors per number of parents/ children aboard
# and save results in list
survivors_per_parch = []
for i in categories_parch:
    parch = train_data.loc[train_data.Parch == i]["Survived"].to_numpy()
    survivors_per_parch.append(round(sum(parch)/len(parch)*100, 1))

# convert categories to string variables for plotting
categories_parch = list(map(str, categories_parch))
categories_parch # check result
```

Out[20]:
```
['0', '1', '2', '3', '4', '5', '6']
```

In [21]:
```
# plot survivors per number of parents/ children aboard
plot_survivors_per_category(categories_parch,
             survivors_per_parch,
             title="Survivors per number of parents/ children aboard",
             xlabel="Number of parents/ children aboard")
```



In [22]:
```
# compare result to total survivors per parents/ children aboard
train_data["Parch"].value_counts()
```

Out[22]:
```
0    678
1    118
2     80
5      5
3      5
4      4
6      1
Name: Parch, dtype: int64
```

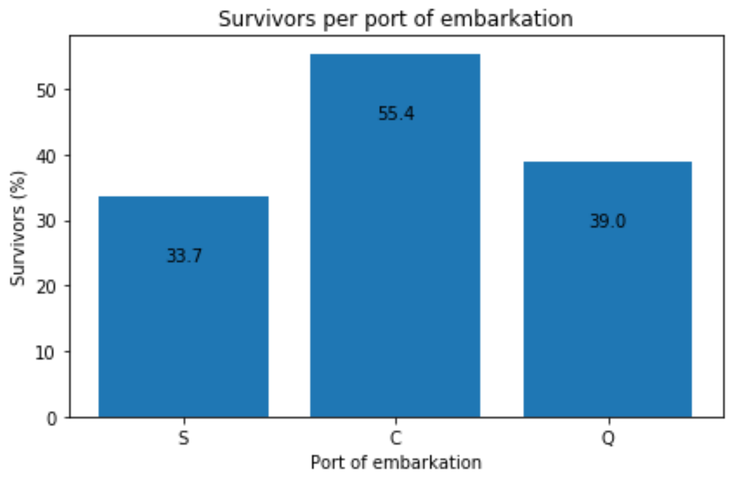
Conclusion: The people who had between 1 and 3 parents/ children aboard had the highest rate of survival. As above, this could mean that these people had support from family members with getting a spot in one of the lifeboats. However, most people travelled without company. Therefore, the number of parents/ children aboard might be weakly associated with the chance of survival.

Port of embarkation

In [23]:
```
# save categories in list
categories_embarked = list(map(str, train_data["Embarked"].unique()))
categories_embarked

# calculate percentage of survivors per port of embarkation
# note: leave out the two passengers of unknown port of embarkation
survivors_per_port = []
for i in categories_embarked[:3]:
    port = train_data.loc[train_data.Embarked == i]["Survived"].to_numpy()
    survivors_per_port.append(round(sum(port)/len(port)*100, 1))
```

Out[23]:
```
['S', 'C', 'Q', 'nan']
```

In [24]:
```
# plot survivors per port of embarkation
plot_survivors_per_category(categories_embarked[:3],
             survivors_per_port,
             title="Survivors per port of embarkation",
             xlabel="Port of embarkation")
```



In [25]:
```
# compare result to total survivors per port of embarkation
train_data["Embarked"].value_counts()
```

Out[25]:
```
S    644
C    168
Q     77
Name: Embarked, dtype: int64
```

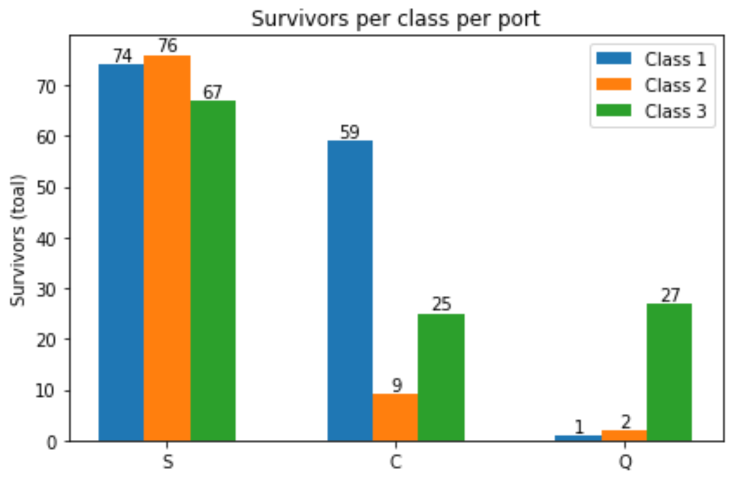
Conclusion: Although by far the most people embarked in Southampton the percentage of survivors who embarked in Cherbourg is higher compared to Southampton and Queenstown. This could be due to many first class passengers having embarked here. Let's check this:

Passengers per class per port

In [26]:
```
# calculate number of survivors per class and port of embarkation
# note: leave out the two passengers of unknown port of embarkation

survivors_class_port = []
# loop over classes
for pclass in classes:
    survivors_per_port_pclass = []
    # loop over ports
    for cat in categories_embarked[:3]:
        port = pclass.loc[pclass.Embarked == cat]["Survived"].to_numpy().sum()
        survivors_per_port_pclass.append(port)
    survivors_class_port.append(survivors_per_port_pclass)

survivors_class_port
```

Out[26]:
```
[[74, 59, 1], [76, 9, 2], [67, 25, 27]]
```

In [27]:
```
# plot survivors per class per port of embarkation

# set variables
x = np.arange(len(categories_embarked[:3]))  # the label locations
width = 0.2  # the width of the bars

# set up plot
fig, ax = plt.subplots()
_ = ax.set_title("Survivors per class per port")
_ = ax.set_ylabel("Survivors (toal)")
_ = ax.set_xticks(x)
_ = ax.set_xticklabels(categories_embarked[:3])

# plot barplot
for i,j in zip(survivors_class_port,range(-1,2)):
    _ = ax.bar(x=x+width*j, height=i, width=width, label=f'Class {j+2}')
    # annotate barplot
    for k, data in enumerate(i):
        _ = ax.annotate(s=data, xy=(k+width*j, data+0.7), ha='center')
_ = ax.legend()
```



Conclusion: Most survivors, irrespective of class, embarked in Southampton. However, in Cherbourg a higher number of survivors belonging to the first class embarked compared to second and thrid class survivors. Additionally, in Queenstown a higher number of survivors belonging to the third class embarked compared to first and second class survivors. Therefore, the port of embarkation might have a weak influence on the prediction of survival.

Summary

Based on this data exploration, the features that likely influence the prediction of survival are in presumed descending order of strength:

- gender
- passenger class
- siblings/ spouses aboard; children/ parents aboard; port of embarkation/ fare

---

Data preparation

Select features to be included in the models

In [28]:
```
# select features and label column
features = ["Sex", "Pclass", "SibSp", "Parch", "Embarked", "Survived"]
```

In [29]:
```
# create pruned train and test datasets
train_data_pruned = train_data[features]
test_data_pruned = test_data[features[:-1]]
```

```
In [30]: # drop rows with missing values in training data and check result
         train_data_pruned = train_data_pruned.dropna()
         train_data_pruned.shape
         train_data_pruned.head()
```

```
Out[30]: (889, 6)
```

Out[30]:

	Sex	Pclass	SibSp	Parch	Embarked	Survived
0	male	3	1	0	S	0
1	female	1	1	0	C	1
2	female	3	0	0	S	1
3	female	1	1	0	S	1
4	male	3	0	0	S	0

```
In [31]: # store labels of training data in separate variable
         y_train = train_data_pruned['Survived']
         X_no_nan = train_data_pruned.drop(columns=['Survived'])
```

Encode categorical features as ordinal integers

In [32]:
```
# define encoder
enc = OrdinalEncoder()

# encode training data and check result
enc.fit(X_no_nan)
X_train = enc.transform(X_no_nan)
enc.categories_

# encode test data and check result
enc.fit(test_data_pruned)
X_test = enc.transform(test_data_pruned)
enc.categories_
```

Out[32]:
```
OrdinalEncoder(categories='auto', dtype=<class 'numpy.float64'>)
```

Out[32]:
```
[array(['female', 'male'], dtype=object),
 array([1, 2, 3]),
 array([0, 1, 2, 3, 4, 5, 8]),
 array([0, 1, 2, 3, 4, 5, 6]),
 array(['C', 'Q', 'S'], dtype=object)]
```

Out[32]:
```
OrdinalEncoder(categories='auto', dtype=<class 'numpy.float64'>)
```

Out[32]:
```
[array(['female', 'male'], dtype=object),
 array([1, 2, 3]),
 array([0, 1, 2, 3, 4, 5, 8]),
 array([0, 1, 2, 3, 4, 5, 6, 9]),
 array(['C', 'Q', 'S'], dtype=object)]
```

Note: the ordinal encoding for the training and test data is identical. However, in the test data there is at least one person who travelled with 9 children. Let's check how many people in total travelled with 9 children:

In [33]:
```
test_data_pruned[test_data_pruned["Parch"] == 9]
```

Out[33]:

	Sex	Pclass	SibSp	Parch	Embarked
342	male	3	1	9	S
365	female	3	1	9	S

Conclusion: there are two people who travelled with 9 children (probaby both parents of the nine children). Because there are no people with 9 children in the training set, this category will introduce bias when making predictions on the test set. Because there are only two people with 9 children, this bias will be very small and we'll therefore just ignore this.

Models

We now train different classic machine learning models on the training set and evaluate their performance with k-fold cross validation.

Run different classic machine learning models:

- Random Forest
- Gradient Boosting
- Logistic Regression
- Suport Vector Machine

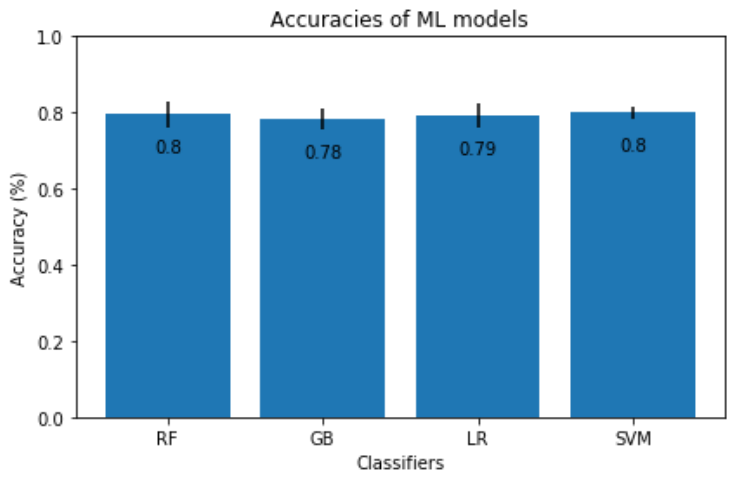
In [34]:
```
# define models
rf = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=0, n_jobs=-1, class_weight={0:2, 1:1})
gb = GradientBoostingClassifier(n_estimators=100, max_depth=5, random_state=0)
lr = LogisticRegression(random_state=0, solver='liblinear', multi_class='ovr', max_iter=100, class_weight={0:2, 1:
1})
svm = LinearSVC(random_state=0, C=1.0, max_iter=1000, class_weight={0:2, 1:1})

# save models in list
models = [rf, gb, lr, svm]
# save model names in list
model_names = ['RF', 'GB', 'LR', 'SVM']
```

In [35]:
```
%%time
# perform cross validation
accuracies = []
std = []
for model, names in zip(models, model_names):
    scores = cross_val_score(model, X_train, y_train, cv=5, n_jobs=-1)
    accuracies.append(scores.mean())
    std.append(scores.std())
    print(f"Cross val scores {names}: ", np.around(scores, decimals=2))
    print(f"Mean and stdev: {scores.mean():.2f} +/- {scores.std():.2f}")
    print()
```

```
Cross val scores RF:  [0.73 0.81 0.83 0.79 0.81]
Mean and stdev: 0.80 +/- 0.03

Cross val scores GB:  [0.74 0.79 0.79 0.78 0.82]
Mean and stdev: 0.78 +/- 0.03

Cross val scores LR:  [0.75 0.81 0.82 0.76 0.82]
Mean and stdev: 0.79 +/- 0.03

Cross val scores SVM:  [0.8  0.8  0.81 0.77 0.81]
Mean and stdev: 0.80 +/- 0.02

CPU times: user 123 ms, sys: 44.5 ms, total: 167 ms
Wall time: 3.1 s
```

Visualise results

In [36]:
```
# plot accuracies of different ML models
plot_model_accuracies(model_names, accuracies, std)
```



Conclusion: the performances of the different classifiers are in a narrow range with mean accuracies between 78% and 80%, and standard deviations between 1% and 3%. In order to better understand the models, let's have a look into model explainability by investigating the importances of individual features.

Check feature importances

Calculate the impurity-based feature importances of the tree classifiers

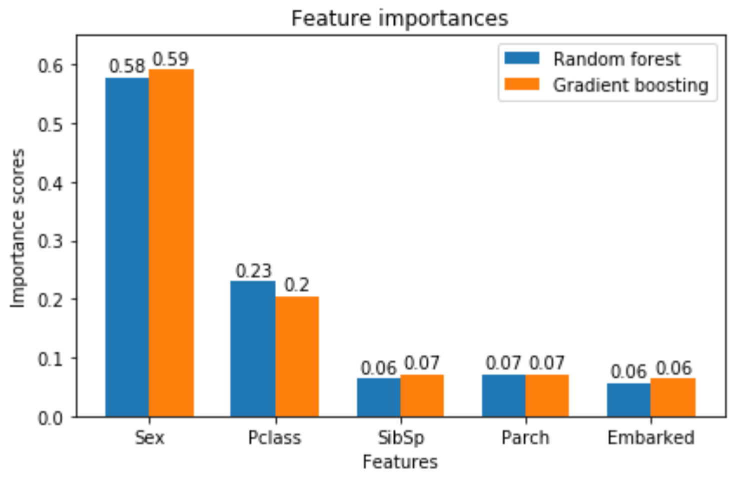
In [37]:
```
# random forest classifier
_ = rf.fit(X_train, y_train)
importances_rf = rf.feature_importances_
```

In [38]:
```
# gradient boosting classifier
_ = gb.fit(X_train, y_train)
importances_gb = gb.feature_importances_
```

In [39]:
```
# store values in list
importances_trees = [importances_rf.tolist(), importances_gb.tolist()]
```

Plot the impurity-based feature importances of the tree classifiers

In [40]:
```
clf_names = ["Random forest", "Gradient boosting"] # classifier names

plot_feature_importance(clf=clf_names,
                        feat=features,
                        ylabel="Importance scores",
                        importances=importances_trees)
```



Conclusion: the feature importance scores are almost identical between the random forest classifier and the gradient boosting classifier. By far the most important feature is gender followed by passenger class. As postulated above, the features siblings/ spouses aboard, parents/ children aboard and port of embaraktion play a minor role in survival prediction.

Calculate weights assigned to the features in the linear classifiers

In [41]:
```
# logistic regression
_ = lr.fit(X_train, y_train)
importances_lr = lr.coef_.ravel()*-1 # flatten array and make weights positive
```

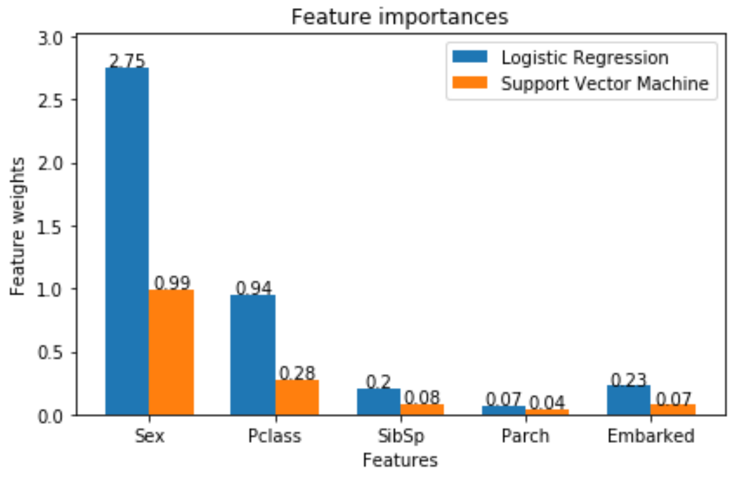
In [42]:
```
# support vector machine
_ = svm.fit(X_train, y_train)
importances_svm = svm.coef_.ravel()*-1 # flatten array and make weights positive
```

```
/home/maren/anaconda3/lib/python3.6/site-packages/sklearn/svm/_base.py:947: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

In [43]:
```
# store values in list
importances_linear = [importances_lr.tolist(), importances_svm.tolist()]
```

Plot weights assigned to the features in the linear classifiers

```
In [44]: clf_names = ["Logistic Regression", "Support Vector Machine"] # classifier names

         plot_feature_importance(clf=clf_names,
                                 feat=features,
                                 ylabel="Feature weights",
                                 importances=importances_linear)
```



Conclusion: overall, feature weights are higher in the logistic regression model compared to the support vector machine model. However, the relative importances between features within the models are roughly equal. As observed in the random forest model and the gradient boosting model, the most important feature is gender followed by passenger class while the features siblings/ spouses aboard, parents/ children aboard and port of embaraktion play a minor role in survival prediction. It is notable that the feature parents/ children aboard seems to be of less importance in the linear models compared to the tree classifiers.
Note: the feature weights are negative but I turned them into positive numbers in order to make them more comparable to the importance scores of the tree classifiers. In binary classification negative feature weights indicate that a feature tends more towards predicting 0. Because the dataset is imbalanced towards fatalities (see data exploration part), feature weights are negative.

Perform hyperparameter tuning

The hyperparameter settings for the models might not be optimal. Let's do a hyperparameter grid search to find out if the hyperparameter settings can be optimised.

In [45]:
```
# define classifiers and store them in list
rf = RandomForestClassifier(random_state=0, class_weight={0:2, 1:1})
gb = GradientBoostingClassifier(random_state=0)
lr = LogisticRegression(random_state=0, multi_class='ovr', class_weight={0:2, 1:1})
svm = LinearSVC(random_state=0, class_weight={0:2, 1:1})

classifiers = [rf, gb, lr, svm]
```

In [46]:
```
# define parameters of classifiers and store them in list
parameters_rf = {'n_estimators':[50, 100, 200], 'max_depth':[None, 5, 10]}
parameters_gb = {'learning_rate':[0.05, 0.1, 0.5], 'n_estimators':[50, 100, 200], 'max_depth':[None, 5, 10]}
parameters_lr = {'solver':['newton-cg', 'lbfgs', 'liblinear'], 'max_iter':[100, 500, 1000]}
parameters_svm = {'max_iter':[1000, 3000, 5000]}

parameters = [parameters_rf, parameters_gb, parameters_lr, parameters_svm]
```

In [47]:
```
# perform best parameter grid search per classifier and print results
for est, params, name in zip(classifiers, parameters, model_names):
    clf = GridSearchCV(est, params, cv=5, n_jobs=-1)
    _ = clf.fit(X_train, y_train)

    # show results
    print("Model: ", name)
    print("Scores per params combination: ", np.around(clf.cv_results_['mean_test_score'], decimals=2))
    print("Best score: ", round(clf.best_score_, 2))
    print("Best parameter combination: ", clf.best_params_)
    print()
```

```
Model:  RF
Scores per params combination:  [0.79 0.79 0.79 0.79 0.8  0.8  0.79 0.79 0.79]
Best score:  0.8
Best parameter combination:  {'max_depth': 5, 'n_estimators': 200}

Model:  GB
Scores per params combination:  [0.79 0.79 0.79 0.79 0.78 0.78 0.79 0.79 0.79 0.79 0.79 0.79 0.78 0.78
 0.78 0.79 0.79 0.79 0.79 0.79 0.79 0.78 0.79 0.78 0.79 0.79 0.79]
Best score:  0.79
Best parameter combination:  {'learning_rate': 0.5, 'max_depth': None, 'n_estimators': 100}

Model:  LR
Scores per params combination:  [0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79]
Best score:  0.79
Best parameter combination:  {'max_iter': 100, 'solver': 'liblinear'}

Model:  SVM
Scores per params combination:  [0.8 0.8 0.8]
Best score:  0.8
Best parameter combination:  {'max_iter': 1000}


/home/maren/anaconda3/lib/python3.6/site-packages/sklearn/svm/_base.py:947: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

Conclusion: after performing hyperparameter tuning, the model accuracies are almost identical (± 1%) compared to the model accuracies previous to hyperparameter tuning. This means 1) that the hyperparameter settings that were initially chosen were likely (close to) optimal and 2) that all tested models show a certain robustness against changing hyperparameters. Let's now check how the models perform on the test set.

Evaluation on the test set

In [48]:
```
# choose best models from hyperarameter grid search
rf = RandomForestClassifier(n_estimators=200, max_depth=5, random_state=0, n_jobs=-1, class_weight={0:2, 1:1})
gb = GradientBoostingClassifier(n_estimators=100, max_depth=None, random_state=0, learning_rate=0.5)
lr = LogisticRegression(random_state=0, solver='liblinear', multi_class='ovr', max_iter=100, class_weight={0:2, 1:
1})
svm = LinearSVC(random_state=0, C=1.0, max_iter=1000, class_weight={0:2, 1:1})

# store models in list
classifiers = [rf, gb, lr, svm]
```

In [49]:
```
# train models
for clf in classifiers:
    _ = clf.fit(X_train, y_train)
```

```
/home/maren/anaconda3/lib/python3.6/site-packages/sklearn/svm/_base.py:947: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

In [50]:
```
# make predictions
predictions = []
for clf in classifiers:
    y_pred = clf.predict(X_test)
    predictions.append(y_pred)
```

In [51]:
```
# save predictions in csv file
for pred, clf in zip(predictions, model_names):
    output = pd.DataFrame({'PassengerId': test_data.PassengerId, 'Survived': pred})
    output.to_csv(f'predictions_{clf}.csv', index=False)
```

The prediction accuracies on the test set as calculated by Kaggle are:

- random forest: 77.0%
- gradient boosting: 75.1%
- logistic regression: 76.6%
- support vector machine: 77.5%

Conclusion: the prediction accuracies on the test are about 3% lower compared to the predictions accuracies on the training set calculated with 5-fold cross validation. Let's check if we can improve the prediction accuracies on the test set by combining the models.

Combine models

In this step we'll combine model predictions using ensemble techniques

In [52]:
```
# choose best models from hyperarameter grid search
# note: models need to be unfitted for this process which is why they are defined again here
rf = RandomForestClassifier(n_estimators=200, max_depth=5, random_state=0, n_jobs=-1, class_weight={0:2, 1:1})
gb = GradientBoostingClassifier(n_estimators=100, max_depth=None, random_state=0, learning_rate=0.5)
lr = LogisticRegression(random_state=0, solver='liblinear', multi_class='ovr', max_iter=100, class_weight={0:2, 1:
1})
svm = LinearSVC(random_state=0, C=1.0, max_iter=1000, class_weight={0:2, 1:1})
```

Majority voting (hard voting)

In [53]:
```
# define voting classifier
vclf_h = VotingClassifier(
    estimators=[('rf', rf), ('gb', gb), ('lr', lr), ('svm', svm)],
    voting='hard', n_jobs=-1)
```

In [54]:
```
# check performance on the training set
scores_h = cross_val_score(vclf_h, X_train, y_train, cv=5, n_jobs=-1)
print("Cross val scores: ", np.around(scores_h, decimals=2))
print(f"Mean and stdev: {scores_h.mean():.2f} +/- {scores_h.std():.2f}")
```

```
Cross val scores:  [0.74 0.82 0.83 0.78 0.81]
Mean and stdev: 0.80 +/- 0.03
```

In [55]:
```
# fit classifier and make predictions on test set
vclf_h = vclf_h.fit(X_train, y_train)
y_pred_vclf_h = vclf_h.predict(X_test)
```

Weighted Average Probability (soft voting)

In [56]:
```
# define voting classifier
# note: we'll give the LinearSVC a higher weight because it performed best on the test set
vclf_s = VotingClassifier(
    estimators=[('rf', rf), ('gb', gb), ('lr', lr), ('svm', svm)],
    voting='hard', weights=[1,1,1,2], n_jobs=-1)
```

In [57]:
```
# check performance on the training set
scores_s = cross_val_score(vclf_s, X_train, y_train, cv=5, n_jobs=-1)
print("Cross val scores: ", np.around(scores_s, decimals=2))
print(f"Mean and stdev: {scores_s.mean():.2f} +/- {scores_s.std():.2f}")
```

```
Cross val scores:  [0.8  0.81 0.81 0.76 0.81]
Mean and stdev: 0.80 +/- 0.02
```

In [58]:
```
# fit classifier and make predictions on test set
vclf_s = vclf_s.fit(X_train, y_train)
y_pred_vclf_s = vclf_s.predict(X_test)
```

Save predictions in csv file

In [59]:
```
for pred, clf in zip([y_pred_vclf_h, y_pred_vclf_s], ['vclf_h', 'vclf_s']):
    output = pd.DataFrame({'PassengerId': test_data.PassengerId, 'Survived': pred})
    output.to_csv(f'predictions_{clf}.csv', index=False)
```

The prediction accuracies on the test set as calculated by Kaggle are:

- voting classifier (hard): 77.0%
- voting classifier (soft, equal weights): 77.0%
- voting classifier (soft, double weight for SVC): %76.6%

Conclusion: combining the models by performing majority voting on the predictions (hard voting) and by using the argmax of the weighted probabilities (soft voting) did not increase the prediction accuracies on the test set. Let's try a different ensembling technique: stacking. In this technique the predictions of the individual models are used as input to a final model.

Stacked generalisation

Here we'll use the LinearSVC as the final estimator and the other models as base estimators because the LinearSVC model performed best as a single model. However, it is possible that different combinations might give better results.

In [60]:
```
# choose best models from hyperarameter grid search
# note: models need to be unfitted for this process which is why they are defined again here
rf = RandomForestClassifier(n_estimators=200, max_depth=5, random_state=0, n_jobs=-1, class_weight={0:2, 1:1})
gb = GradientBoostingClassifier(n_estimators=100, max_depth=None, random_state=0, learning_rate=0.5)
lr = LogisticRegression(random_state=0, solver='liblinear', multi_class='ovr', max_iter=100, class_weight={0:2, 1:
1})
svm = LinearSVC(random_state=0, C=1.0, max_iter=1000, class_weight={0:2, 1:1})
```

In [61]:
```
# define stacking classifier
stack_clf = StackingClassifier(
    estimators=[('rf', rf), ('gb', gb), ('lr', lr)],
    final_estimator=svm, n_jobs=-1)
```

In [62]:
```
# check performance on the training set
scores_stack = cross_val_score(stack_clf, X_train, y_train, cv=5, n_jobs=-1)
print("Cross val scores: ", np.around(scores_stack, decimals=2))
print(f"Mean and stdev: {scores_stack.mean():.2f} +/- {scores_stack.std():.2f}")
```

```
Cross val scores:  [0.75 0.81 0.82 0.79 0.81]
Mean and stdev: 0.80 +/- 0.02
```

In [63]:
```
# fit classifier and make predictions on test set
stack_clf = stack_clf.fit(X_train, y_train)
y_pred_stack_clf = stack_clf.predict(X_test)
```

In [64]:
```
# save predictions in csv file
output = pd.DataFrame({'PassengerId': test_data.PassengerId, 'Survived': y_pred_stack_clf})
output.to_csv('predictions_stack_clf.csv', index=False)
```

The prediction accuracy on the test set as calculated by Kaggle using the stacking classifier is 77.0%

Final conclusion

The best prediction accuracy on the test set was achieved by the support vector machine (77.5%), closely followed by the remaining three classifiers (random forest: 77%, gradient boosting: 75.1%, logistic regression 76.6%). Using model ensembling techniques such as hard voting (majority voting), soft voting (weighted average probability) and stacked generalisation resulted in comparable prediction accuracies between 76.6% and 77.0% on the test set.

It has been shown on Kaggle that it is possible to achieve 100% prediction accuracy on the test set. While this information is valuable, it raises the concern of lack of explainability and overfitting. For example, the missing age values could have been imputed or the ticket number information could have been used in this analysis. Including these features into the analysis could have potentially increased the prediction accuracy on the test set. However, including this data would not have provided any insight into improving the safety of passengers travelling on a passenger liner.

In this analysis, the best way to improve prediction accuracies is to look at the models used here and their implementation. For example, more classifiers such as k-nearest neighbors or Naive Bayes could have been included in the machine learning pipeline. Changing the loss functions in classifiers is another possible way of improving prediction accuracies. For example, I changed the loss function in the gradient boosting classifier from ‘deviance’ to ‘exponential’ to make use of the AdaBoost algorithm, however this had no effect on the prediction accuracy. Similarly, in the support vector machine model I changed the loss function from ‘squared_hinge’ to ‘hinge’ but this also did not affect the prediction accuracy. Another avenue I tested is changing the regularisation strength (C) in the logistic regression and support vector machine classifiers but again there was no effect on prediction accuracies when I made these changes.

In summary, the highest survival prediction rate of 77.5% achieved by the support vector machine model is a good result for this comparatively small dataset. In theory, a larger dataset with respect to entries and features would have likely improved the prediction accuracies and reduced bias as well as variance while not compromising explainability.

```
In [ ]:
```