Udacity – Machine Learning Engineer Nanodegree – Capstone Report – Predict Survival on the Titanic

DEFINITION

Project Overview

As described in detail in my Capstone Proposal, my Capstone Project is about solving a challenge retrieved from Kaggle: Predicting survival on the RMS Titanic.

On April 15, 1912, during her maiden voyage, the widely considered "unsinkable" RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren't enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew. While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others.

Kaggle provides a dataset containing information about the passengers such as name, age, gender, socio-economic class etc., which can be used to apply machine learning techniques to predict for each passenger whether the passenger survived the disaster or not.

The challenge can be found here: https://www.kaggle.com/c/titanic.

Figure 1: Sinking of the RMS Titanic

With the goal to predict survival from a labeled relational dataset, the problem can be categorized into the supervised learning domain. More specifically, it is a classification problem with a binary target variable (Survived: "Yes" or "No"). In supervised learning, as opposed to unsupervised learning, we have the (groundtruth) labels available in our dataset and use this label to train our algorithms with a cost function that punishes the algorithm for predictions that are not in line with these target labels.

Problem Statement

By now it is obvious that predictive modeling techniques have to be used to solve the problem of predicting the RMS Titanic passengers' survival chances based on the available passenger data. In this section, I want to discuss a more specific problem statement that I came up with in addition to this general goal posed in the Kaggle challenge.

In advance of the project, I have set myself the goal to build an algorithm that comes as close as possible to 90% accuracy on the test set (i.e. 90% of the data points are correctly predicted) as this seemed reasonable looking at the results of other participants in the Kaggle challenge. However, in addition to building a well-performing model in terms of prediction quality, I also set myself the goal to create some nice custom plots and a well-rounded and reproducible pipeline that would allow me to train different models and tune hyperparameters easily without much extra effort.

Based on feedback from a Udacity reviewer for my Capstone Proposal, I also want to add to this previously defined problem statement the part of model explainability. While it is one thing to be able to predict whether a passenger survived or not, it is yet another problem to determine what were the decisive factors or features leading to this prediction. Recently, model explainability has experienced a significant rise in relevance when building machine learning solutions as business stakeholders and other model users often want to understand how a model arrived at a certain prediction. This becomes even more important when thinking about topics such as model bias and responsible and ethical AI.

I already outlined my solution approach to the above problem statement in the Capstone Proposal but want to again give a holistic overview over the individual steps, including the newly added model explainability goal.

It can therefore be expected that my solution to this challenge covers the following topics:

- Exploratory Data Analysis (EDA) incl. Visualizations of the Data
- Data Cleansing and Preprocessing
- Feature Engineering
- Creation of a reproducible Pipeline
- Training of a Benchmark Model
- Hyperparameter Tuning with Cross Validation to train several different Models
- Evaluation of the Models
- Explanation of the Model Results with SHAP

For more details on each of the points, please also refer to my Capstone Proposal as I already gave a quite detailed outline on how each of these tasks look in detail.

In terms of technology, I use Jupyter Notebooks with several data science Python libraries such as scikit-learn, numpy, pandas, matplotlib, seaborn, plotly and shap. I have also written some custom utility functions and helper classes over the course of the project.

Metrics

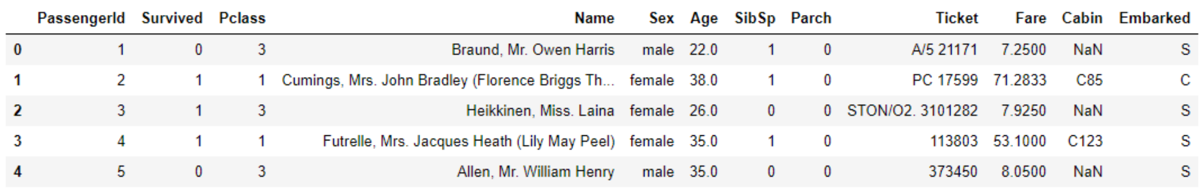
In my Capstone Proposal I discussed several different possible metrics to evaluate a binary classification model such as Precision and Recall. One of the most common metrics that also works well for imbalanced datasets is the AUC (Area under the Curve), which refers to the area under the ROC (Receiver Operating Characteristic) curve. This curve plots the True Positive Rate against the False Positive Rate.

However, the RMS Titanic dataset is quite balanced, i.e. that we have quite similar numbers of data points for each label, and the challenge explicitly states that accuracy should be the target metric, which is why I look into optimizing my models for accuracy. Accuracy just refers to the percentage of data points that have been predicted correctly.

ANALYSIS

Data Exploration

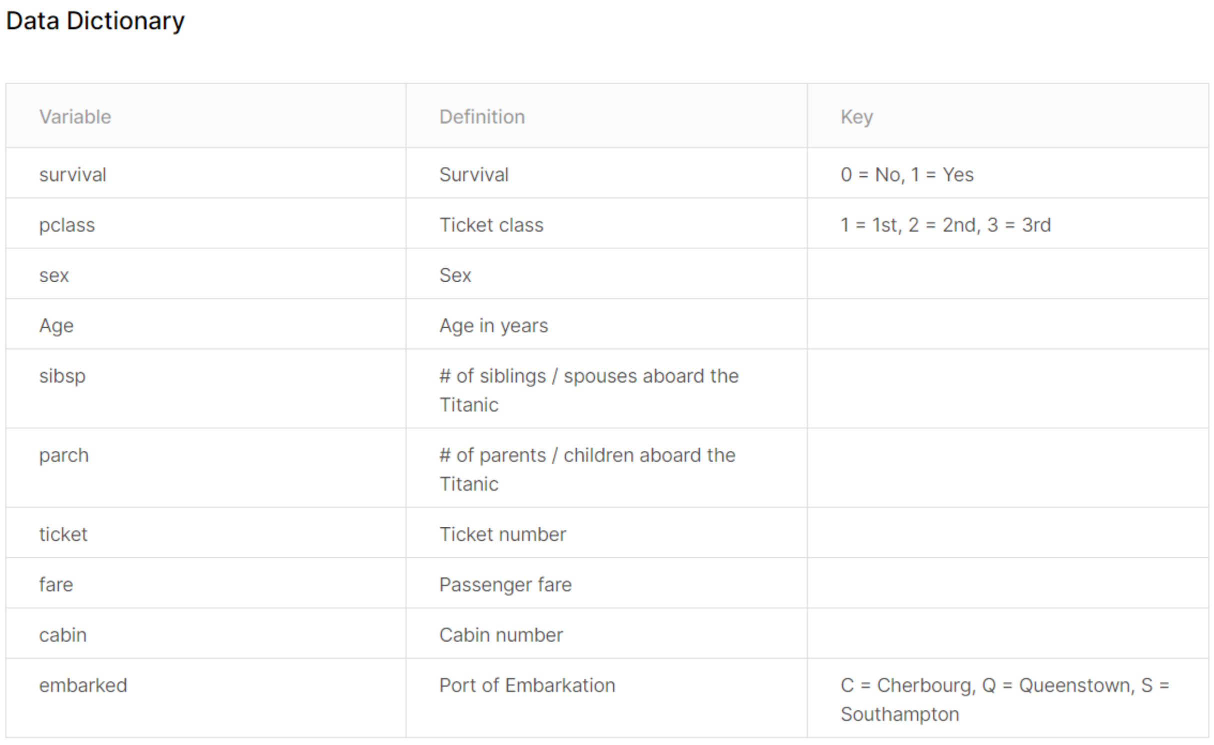
The original dataset contains 12 columns: the target variable “Survived” as well as “Passenger Id”, “Pclass”, “Name”, “Sex”, “Age”, “SibSp”, “Parch”, “Ticket”, “Fare”, “Cabin” and “Embarked.



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

Figure 2: First Glance at the Dataset

Kaggle provides a nice data dictionary to describe the content of all columns (it is always useful for the data understanding and feature engineering to know exactly what individual columns contain):



Data Dictionary

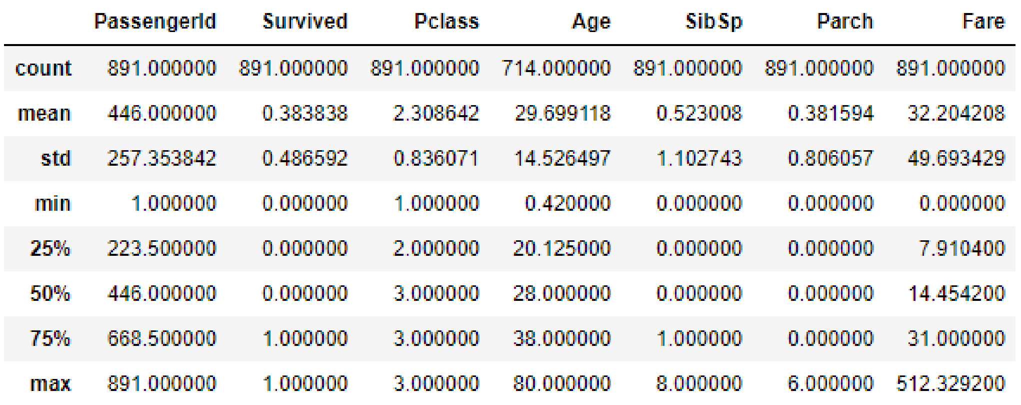
Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked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

Figure 3: Kaggle Data Dictionary

Some first analysis in Jupyter Notebooks using the Python pandas library leads to the following results:

- There are both numeric and categorical columns. I have to handle categorical columns later and one-hot-encode them as many machine learning algorithms do not accept non-numeric data.

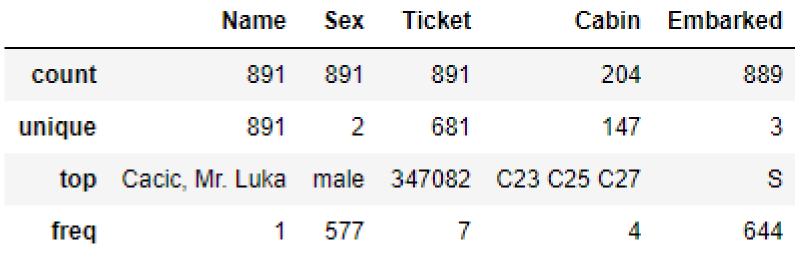
- For the numeric columns we can take a look at their summary statistics to get a first feeling on how the variables are distributed. The min and max columns help us to understand the range of a variable, while the count can be used to detect missing values (if the count is lower than the number of rows in the dataset). The mean is also valuable to get a feeling for the "average" passenger.



	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Figure 4: Summary Statistics Numeric Columns

- For the categorical columns we can take a look whether there are missing values and how many unique values appear in the dataset.



	Name	Sex	Ticket	Cabin	Embarked
count	891	891	891	204	889
unique	891	2	681	147	3
top	Cacic, Mr. Luka	male	347082	C23 C25 C27	S
freq	1	577	7	4	644

Figure 5: Summary Statistics Categorical Columns

We can see that we have 3 columns with missing values: "Age", "Cabin" and "Embarked". Many machine learning algorithms cannot handle missing values, which is why I have to take care of them in a preprocessing step. For the "Cabin" column one possible option is to drop the column completely since the share of missing data is too high. However, I use this column to create a new feature called "CabinCategory" which just uses the first character of the "Cabin" column to reflect the location of the cabin on the ship since maybe certain locations, and thus cabins in the ship, had a better chance of survival. After creating this column, I dropped the original "Cabin" column as it had too many different and missing values.

Another important thing to observe is the distribution of the target variable "Survived". We find that the dataset is quite balanced and, therefore, accuracy will be a suitable metric to evaluate our models. More specifically, out of the 891 passengers that our dataset contains, 342 people survived the disaster and 549 people died. This means that ~38% of the people survived. This number also more or less corresponds with the total survival rate of passengers on the titanic which was ~40%. We therefore have a reasonable subsample of the total data in our dataset.

Based on the given features in the dataset, I also decided to engineer some additional features. Whereas the full name of a passenger per se is useless to make a prediction, maybe the salutation or the last name could help in the prediction as they give information about the marital and family status as well as potentially the social class of a passenger. I therefore created these two additional columns in favor of the full name column. Moreover, I also created a squared, cubed and logarithmic version of the two numeric columns “Age” and “Fare” to give also linear models a chance to model these features in a non-linear way. In addition, I used the “SibSp” and “Parch” columns to create the “FamilyOnBoard” column, which nicely and easily shows whether the passenger had family members with them on board. As a final pre-visualization step, I dropped the “Name” and “PassengerId” columns as I consider these columns not useful to solve the problem.

Exploratory Visualization

I have written 4 utility functions in the graph_utils.py file which make use of the Python plotly graph library to plot a pie plot, histogram, correlation matrix and radar plot respectively.

The pie plot function contains logic to plot the distribution (in terms of percentage shares) of an arbitrary column over the two target labels (Survived or Dead) separately. This can be very useful to detect differences in the distributions of the column under consideration between the two target labels and therefore get a first indication of which columns could be relevant for a prediction. I use this function to create one plot for each categorical column that does not have more than 15 unique values (otherwise the plot would get quite messy and would not be readable anymore).

It seems that the gender column is the one that shows the greatest difference between the two target labels, which is also to expect as usually women get evacuated before men and therefore have a much higher chance of surviving.

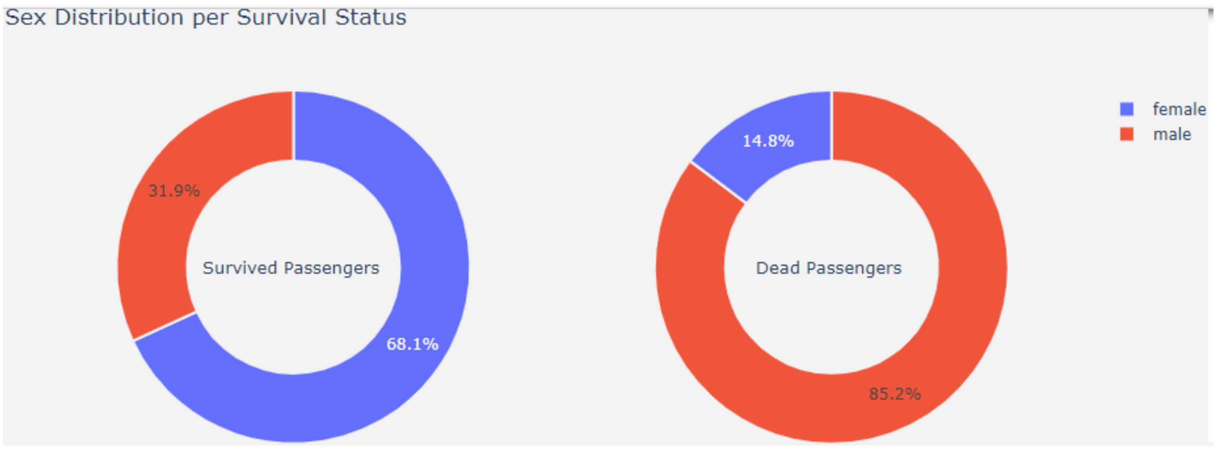


Figure 6: Pie Plot Sex Distribution per Survival Status

Whereas the pie plot is only suited for categorical columns with low cardinality, the histogram function can be used to get a feeling for the distribution of numerical columns with high cardinality. The histogram function works conceptually similar to the pie plot function in that it plots the distribution of a column over the two target labels separately.

We can see that the age column also seems to contain very relevant information to differentiate between the two target labels. Younger passengers have a relatively high chance of surviving compared to older passengers. Again, this makes sense as children are evacuated first. However, it seems that the age groups with the worst chances to survive are the ones between 18 and 30 years and the group > 65 years. The theory behind the age group of 18-30 years could be that these are the people that are in the best physical conditions and therefore tried to help other people giving up their own lifes in the process.

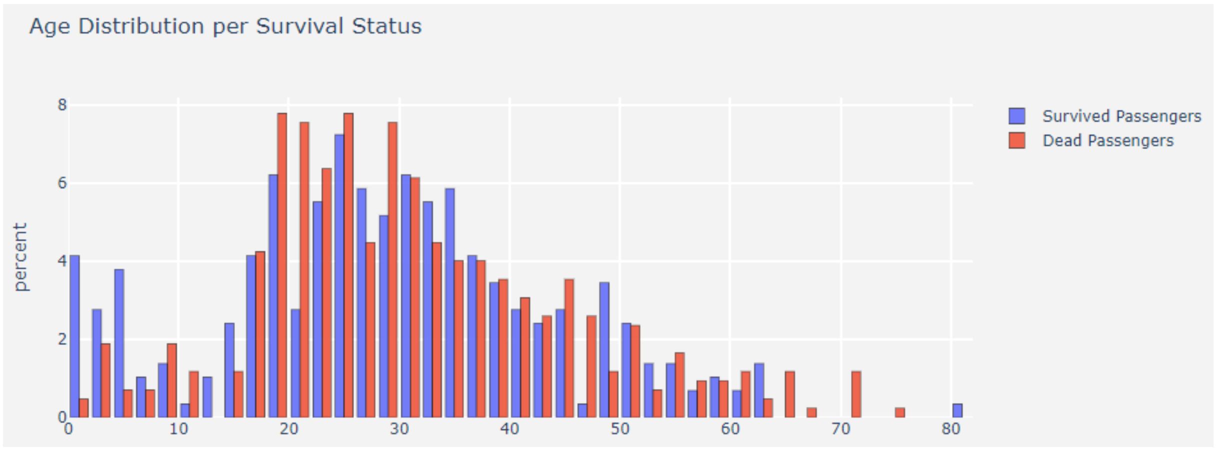


Figure 7: Histogram Age Distribution per Survival Status

The correlation matrix function can be used to plot the correlation between different variables in the dataset. This really only makes sense for numeric columns. However, dummy variables can be built from categorical columns to also include them in the correlation matrix. I have done this for the columns with low cardinality. For columns with a high cardinality this does not make too much sense as the correlation plot would explode and also only include too little data points per dummy variable.

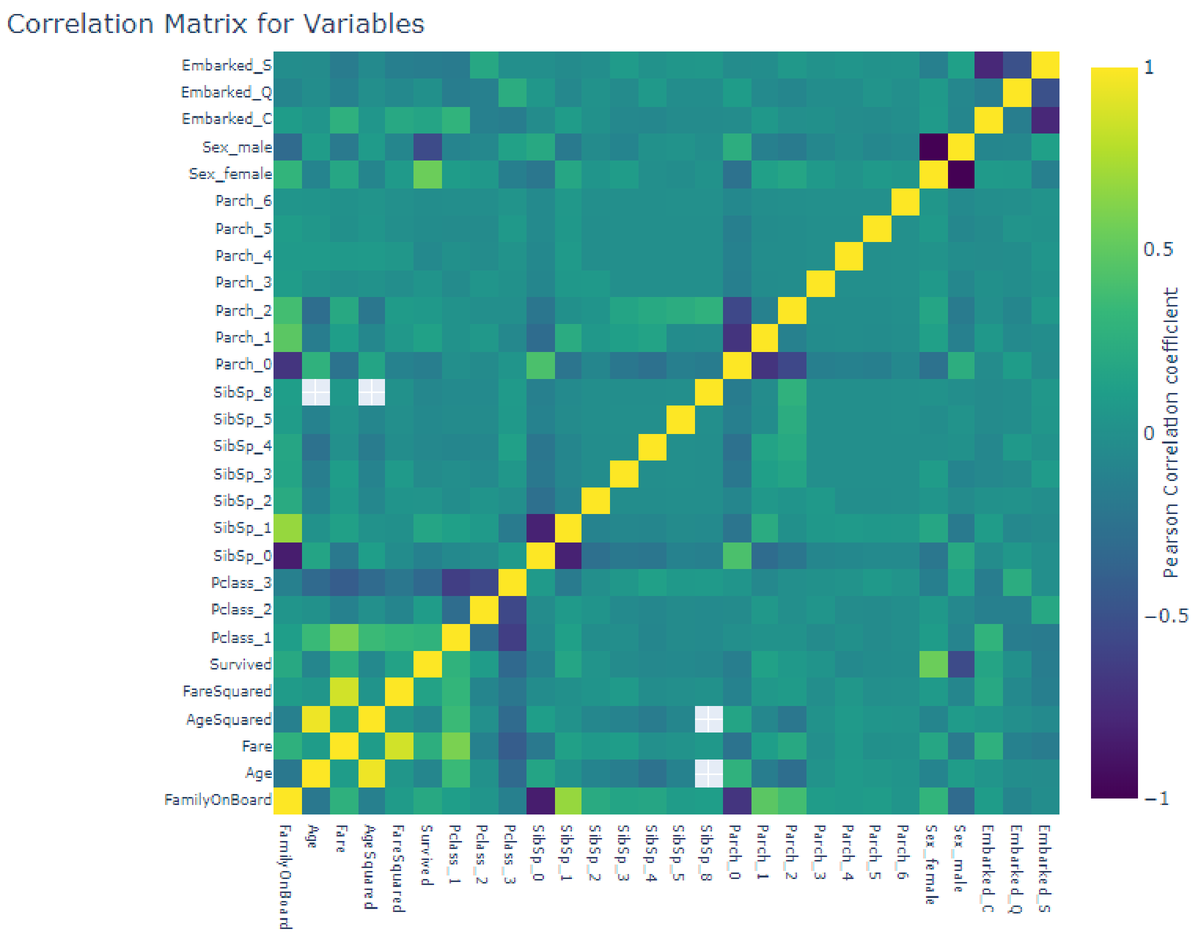


Figure 8: Correlation Matrix for Variables

Moreover, the radar plot function can be really nice to show binary variables and their differences in distribution over the two target labels.

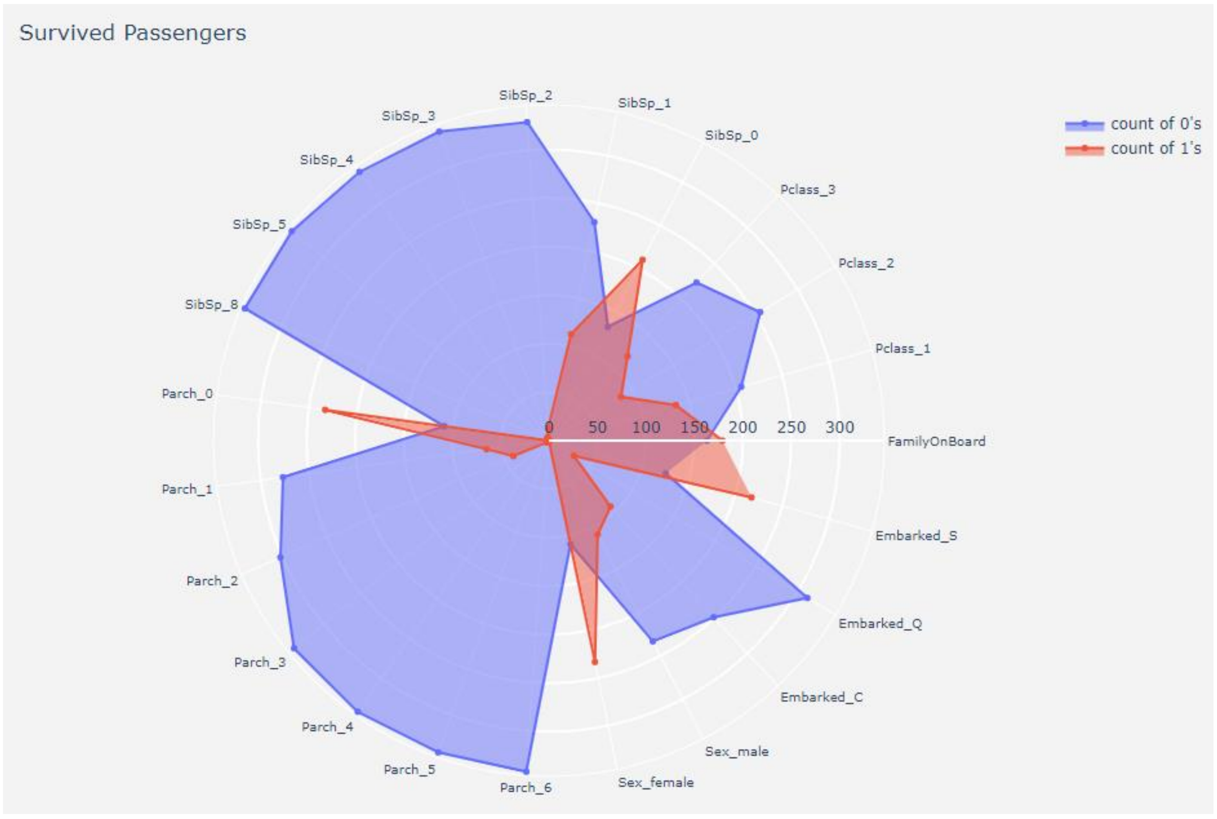


Figure 9: Radar Plot Survived Passengers

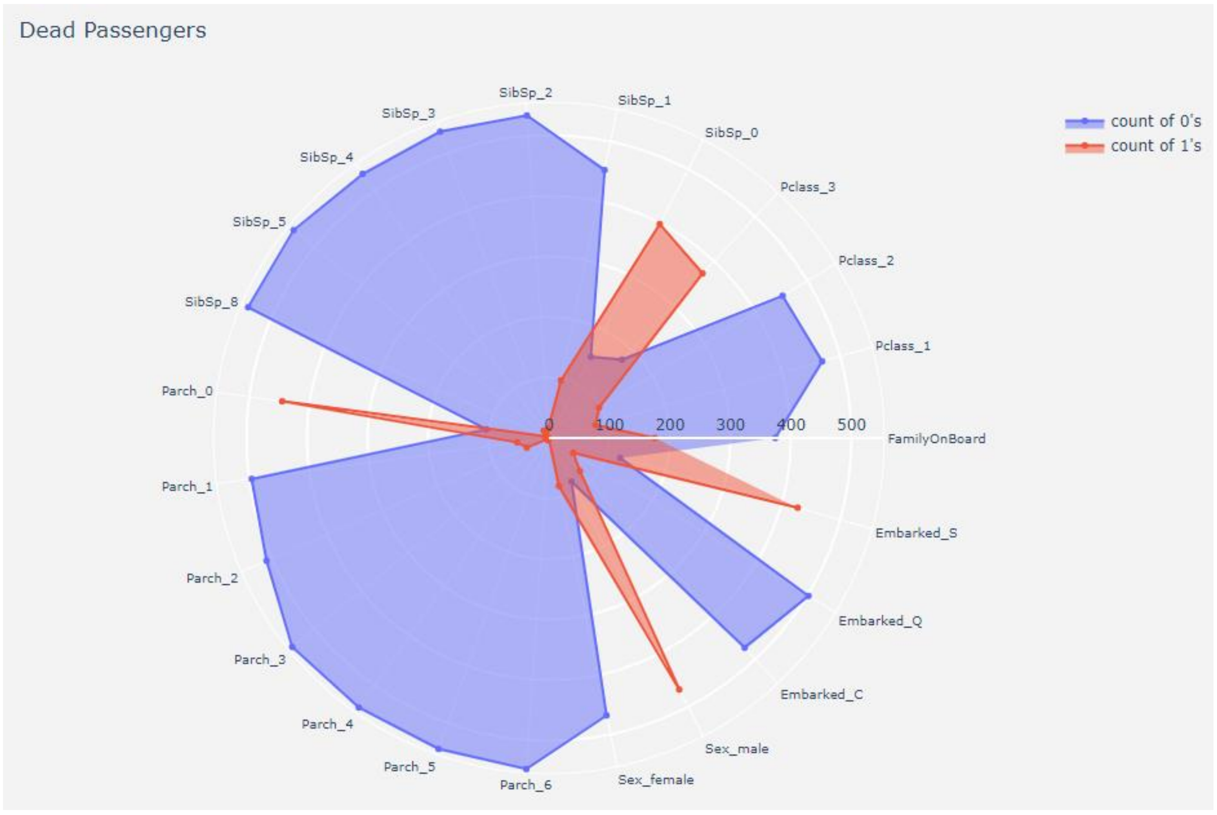


Figure 10: Radar Plot Dead Passengers

By taking a look at these two graphs, we can for example identify that, proportionally, there were many more males under the dead passengers than under the survived passengers. In contrast, when people had family members on board, they were more likely to survive. One interesting finding might be that embarkment in Southampton seems to be more prevalent for dead passengers. It could be that people who embarked in Southampton were primarily in a certain part of the ship which made it less likely to survive. However, this number is not so big, so it could just be produced by random noise.

Last but not least, for the numeric columns I also checked whether they contain outliers that should be removed:

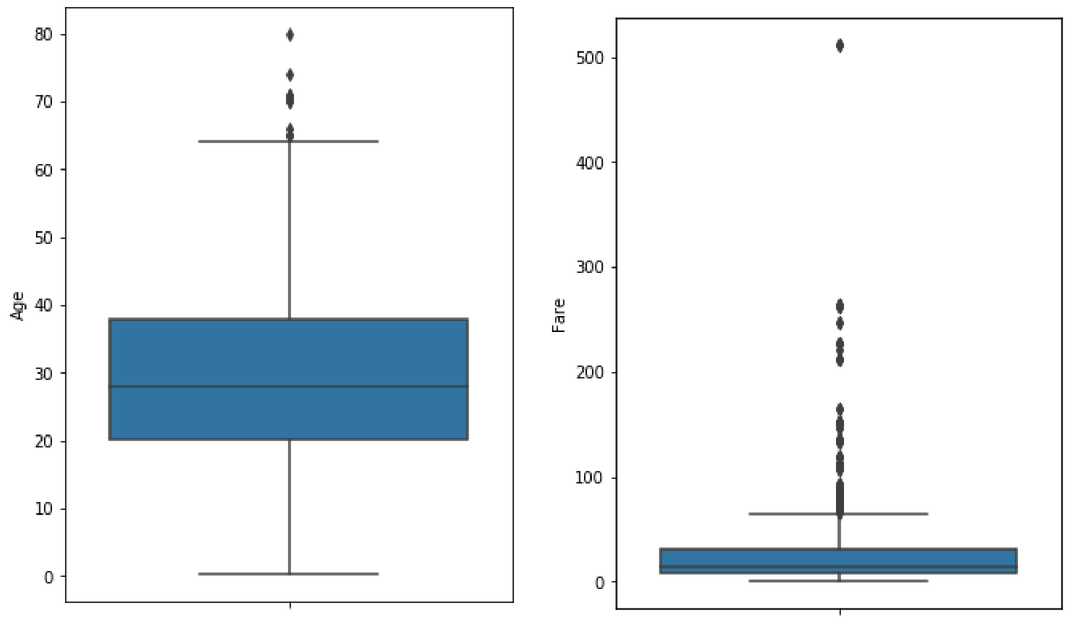


Figure 11: Boxplots of Numeric Columns

We can oberserve that both numeric columns seem reasonable. While there are some outliers for the fare, this could just be “VIP” tickets and is nothing we should worry about for training our models.

Algorithms and Techniques

In general, as mentioned in the project overview, we are facing a binary classification problem here. It is also important to note that the dataset is very small, so techniques that work well on or need a big amount of data might not be the ones to be favored here. This is why, for example, I have not looked into training deep learning models for this problem. “Simpler” classifiers should do the job or do an even better job on this problem than sophisticated deep neural networks.

My specific procedure was to try the following model algorithms, which are known to work well on small datasets and binary classification problems:

- Logistic Regression
- Decision Tree
- Random Forest
- Support Vector Machine
- XGBoost

Whereas the first 4 that are listed here are implemented in the well-known Python scikit-learn package, the XGBoost algorithm has its own Python library called xgboost which, to my delight, also offers a sci-kit learn API and can be seamlessly used with other classes and functions from the scikit-learn package. Using these libraries, the actual model training is quite easy and can be done by creating a model instance using the respective class and then using the fit and predict methods to fit the model's parameters to the dataset and finally make a prediction using the learned parameters.

In order to achieve a really robust well-performing model, however, it is not enough to train these algorithms with their default hyperparameter configuration. Usually, the training of models is a very iterative process and one tries different hyperparameter configurations for different algorithms to see which configuration works well with the dataset at hand. There is no general solution that works for any problem, so the technique that is nowadays often used is that multiple different hyperparameter configurations are tried out and the models are evaluated to then decide which hyperparameter configuration is the one to be favored. I followed this procedure and made use of two techniques called cross-validation and hyperparameter tuning for each of the listed algorithms respectively. More specifically, I used sci-kit-learn's GridSearchCV class, which goes through a grid of parameters and trains several models in multiple rounds for each possible combination of hyperparameters in the grid. How many models are trained per hyperparameter combination depends on the cv parameter of the instance, which indicates how many folds should be used for the cross validation. I used 10 folds, which means that the training data is split into 90% actual training data and 10% validation data. The validation data is then used to evaluate the model based on accuracy in my case. Cross-validation will repeat this process over 10 rounds, each time using a different chunk of the training set as validation set, so that all the data is used as validation set in one round. It will then calculate the final accuracy by averaging the calculated accuracy over all rounds. This is especially important if the dataset is very small and one cannot afford to lose any data for training, which is the case for my problem. The following image, which I retrieved from a blog post of Joseph Nelson explains this procedure very nicely:

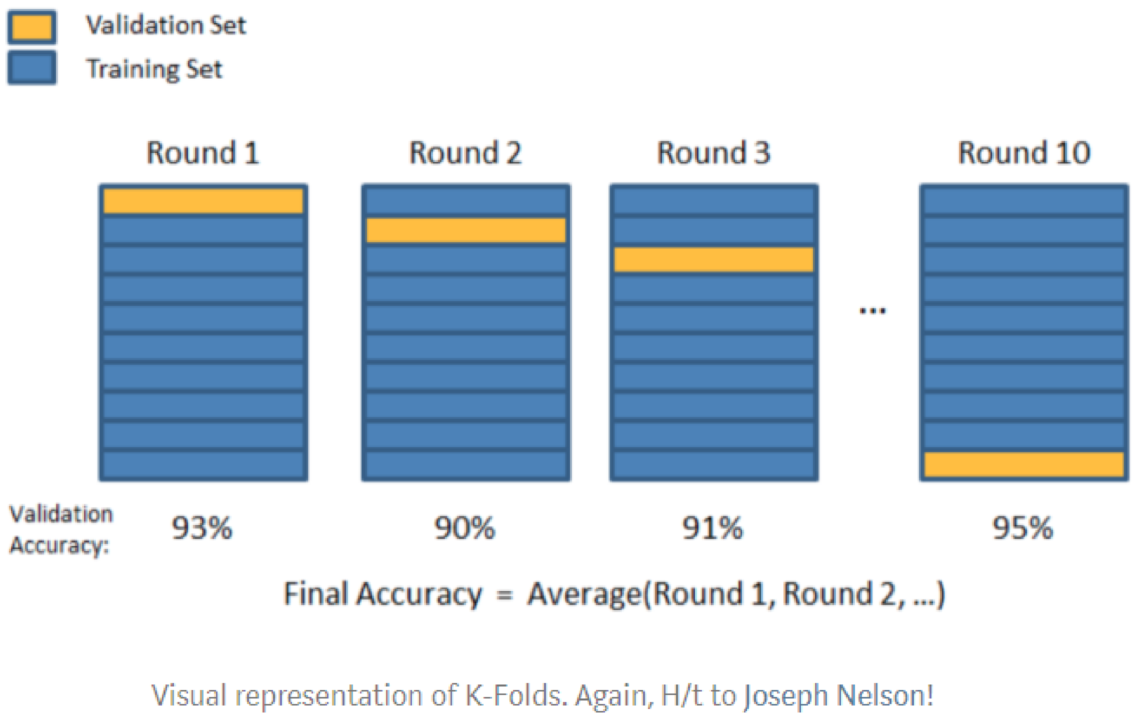

Validation Set
Training Set
Round 1
Round 2
Round 3
Round 10
...
Validation Accuracy:
93%
90%
91%
95%
Final Accuracy = Average(Round 1, Round 2, ...)
Visual representation of K-Folds. Again, H/t to Joseph Nelson!

Figure 12: Cross Validation Process

After this cross-validation process has been repeated for every possible combination of hyperparameters in the provided grid, the GridSearchCV class can be used to return the model that has performed best. By doing this for all the listed algorithms, I will probably get a quite well-performing and robust model in the end, at least one that was favored over all the other models that have been trained.

It also has to be mentioned that GridSearchCV, as the name suggests, goes through all possible combinations of the hyperparameter grid. There are also some other possibilities such as randomized search or even search that relies on Bayesian learning to learn which parts of the hyperparameter grid are not so promising. However, in this project I will just make use of the grid search technique as the dataset is quite small and thus many different models can be trained in quite a short time.

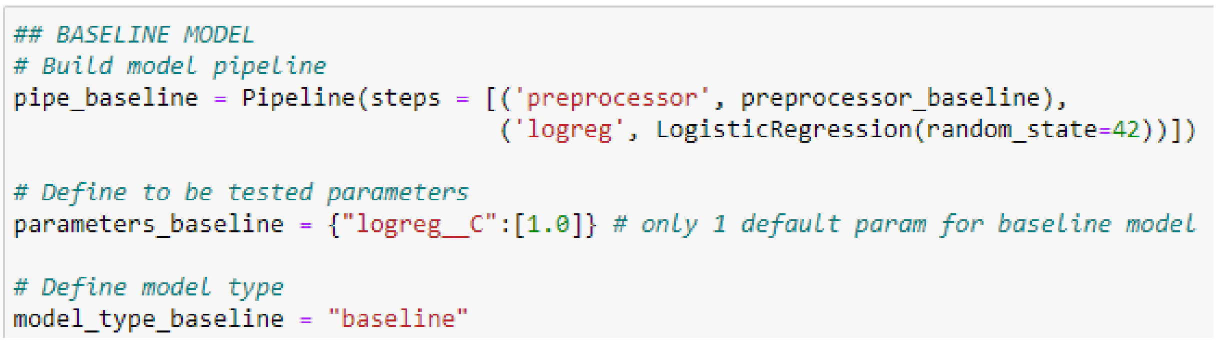
Finally, after having trained multiple different models and optimized them using hyperparameter tuning, I also make use of the shap library to interpret the best-performing model. I already pointed out that model interpretability has gained significance in recent times. Shap is a game theoretic approach to explain the output of any machine learning model and is one of the most promising packages in this area (next to lime and InterpretML from Microsoft).

While a detailed explanation of what shap does is out of the scope of this project and can be found on the github repo instead (https://github.com/slundberg/shap), I want to mention that shap can basically be used to determine the contribution of each feature to the final prediction of a model. This can be done both on a global level, meaning which features are important for the model prediction over the entire dataset as well as on the local level, which means that individual data points can be investigated and it can be determined which features were relevant for the prediction outcome of this particular data point. This is very useful to explain why the model acted the way it did and can make a huge difference in the real-world in terms of trust in and adoption of machine learning models as well as to make sure that machine learning models behave ethically and responsibly as biases can be brought to the surface by using these model interpretability libraries.

Benchmark

As a benchmark model I trained a simple logistic regression model, which is considered to be one of the simplest algorithms for binary classification problems. In addition, I also ensured that this benchmark model only considers the initial features of the dataset to assess whether the features that I engineered make sense or do not help in predicting the outcome at all. In the Jupyer Notebook, this is reflected by the dataframe “titanic_original” as opposed to the feature-engineered dataframe “titanic” and leads to the fact that I have two training-tests splits, one with this original dataframe for training the baseline model and one with my feature-engineered dataframe to train all other models.

I also included the benchmark model in my normal cross-validation, hyperparameter tuning training pipeline, but the parameter space only contains one default parameter for the inverse of the regularization strength (C), and, thus, effectively no hyperparameter tuning is conducted for this model (see figure below).



```python
## BASELINE MODEL
# Build model pipeline
pipe_baseline = Pipeline(steps = [('preprocessor', preprocessor_baseline),
                                  ('logreg', LogisticRegression(random_state=42))])

# Define to be tested parameters
parameters_baseline = {"logreg__C":[1.0]} # only 1 default param for baseline model

# Define model type
model_type_baseline = "baseline"
```

Figure 13: Baseline Model Configuration

With this simple model, I receive an **accuracy of 78,36%** on the test set.

Data Preprocessing

As mentioned in the "Data Exploration" section, the dataset at hand contains missing values and categorical columns that have to be preprocessed before model training can be started.

In a first (by now legacy) step, in order to later be able to one-hot-encode string columns, I wrote a CustomStringImputer class, which can be found in the custom_transformers.py file. This class just replaces missing values (NaNs) with the string "N/A" for the two categorical columns that contain missing values ("Cabin" and "Embarked"), which will then receive its own category after one-hot-encoding. The CustomStringImputer class inherits from scikit-learn's BaseEstimator and TransformerMixIn classes, so that it can be used seamlessly in a Pipeline object, which I am using to make the model training process easily reproducible.

Moreover, there were also missing values in the "Age" column. I could have decided to simply impute this column using the mean, for example. However, looking at the correlation matrix from the "Exploratory Visualization" section, it can be observed that it might be more accurate to use a more sophisticated imputation method to make use of these correlations in guessing a value for the missing values. I therefore decided to use KNN imputation instead, which uses the k-nearest-neighbor data points (in terms of a distance measure, here Euclidean distance) to impute the numeric values. In order to be able to use scikit learn's KNNImputer on my complete dataframe, which also contains string columns, I wrote a custom class that inherits from the KNNImputer and contains the respective logic to ignore string columns for the KNN imputation.

Later in the project, I discovered other solutions which worked to solve the two above mentioned problems out of the box: I now use scikit-learns SimpleImputer to replace missing values in categorical columns and use scikit-learns ColumnTransformer to create two separate pipelines for numeric and string columns and combine them before model training, thus enabling me to use scikit-learn's KNNImputer on only the numeric columns. Nonetheless, I consider it useful practice to have written my own estimators.

After having imputed all missing values, I use scikit-learn's OneHotEncoder class to convert my categorical columns to dummy columns that contain one column for each possible value of a column and use 0s and 1s to indicate which value is "active" for each data point. For an example of how this works, see the following table:

Color	**Color_Red**	**Color_Blue**	**Color_Green**	**Color_Yellow**
Red	1	0	0	0
Blue	0	1	0	0
Yellow	0	0	0	1
Green	0	0	1	0

Figure 14: Example One Hot Encoding

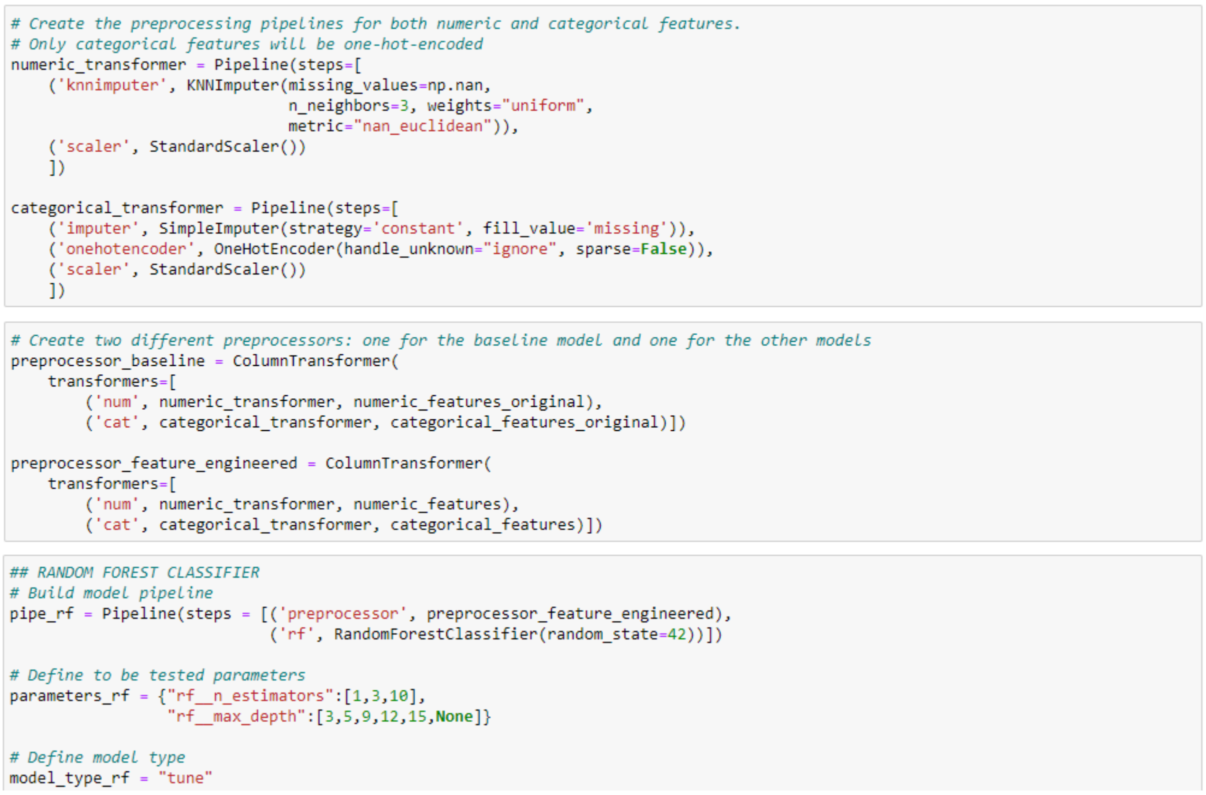
I set the sparse attribute to False so that it would produce dense vectors that can then be used in a final preprocessing step with scikit-learn's StandardScaler class, which normalizes the column values. This is needed for many machine learning models for various reasons, such as fair evaluation of feature importance, faster training or a correct convergence of the algorithm.

All these preprocessing steps are packaged into a pipeline and used for all models to train. The advantage of such a pipeline is also that it is later quite easy to convert new data, for example the test set or also new data that an inference should be made on. This also facilitates model deployment quite a bit.

Implementation

The core of my implementation as I already defined in the project design phase is to have a robust pipeline that I can reuse to train multiple models and tune their hyperparameters.

In the "Data Preprocessing" section I described already parts of this pipeline. The final step of the pipeline that is still missing is the actual model. With the scikit-learn library it is quite easy to create and train such models. An example of a complete pipeline is given in the figure below.



```python
# Create the preprocessing pipelines for both numeric and categorical features.
# Only categorical features will be one-hot-encoded
numeric_transformer = Pipeline(steps=[
    ('knnimputer', KNNImputer(missing_values=np.nan,
                              n_neighbors=3, weights="uniform",
                              metric="nan_euclidean")),
    ('scaler', StandardScaler())
    ])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehotencoder', OneHotEncoder(handle_unknown="ignore", sparse=False)),
    ('scaler', StandardScaler())
    ])
```

```python
# Create two different preprocessors: one for the baseline model and one for the other models
preprocessor_baseline = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features_original),
        ('cat', categorical_transformer, categorical_features_original)])

preprocessor_feature_engineered = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

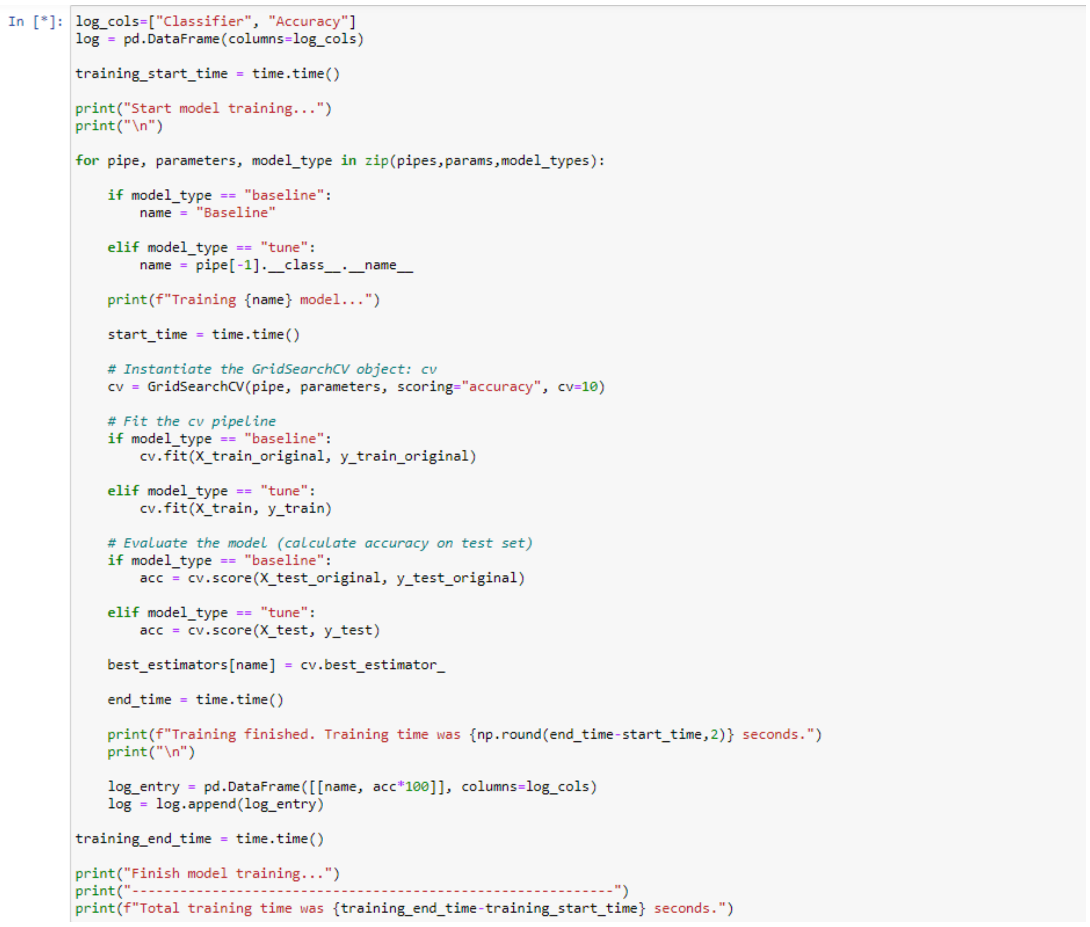
```python
## RANDOM FOREST CLASSIFIER
# Build model pipeline
pipe_rf = Pipeline(steps = [('preprocessor', preprocessor_feature_engineered),
                            ('rf', RandomForestClassifier(random_state=42))])

# Define to be tested parameters
parameters_rf = {"rf__n_estimators":[1,3,10],
                 "rf__max_depth":[3,5,9,12,15,None]}

# Define model type
model_type_rf = "tune"
```

Figure 15: Example Pipeline Random Forest Classifier

The biggest challenge for me personally was to find out how to only one-hot-encode categorical columns. I even had written a custom estimator to do this until I found the ColumnTransformer class which helps a lot in this task.

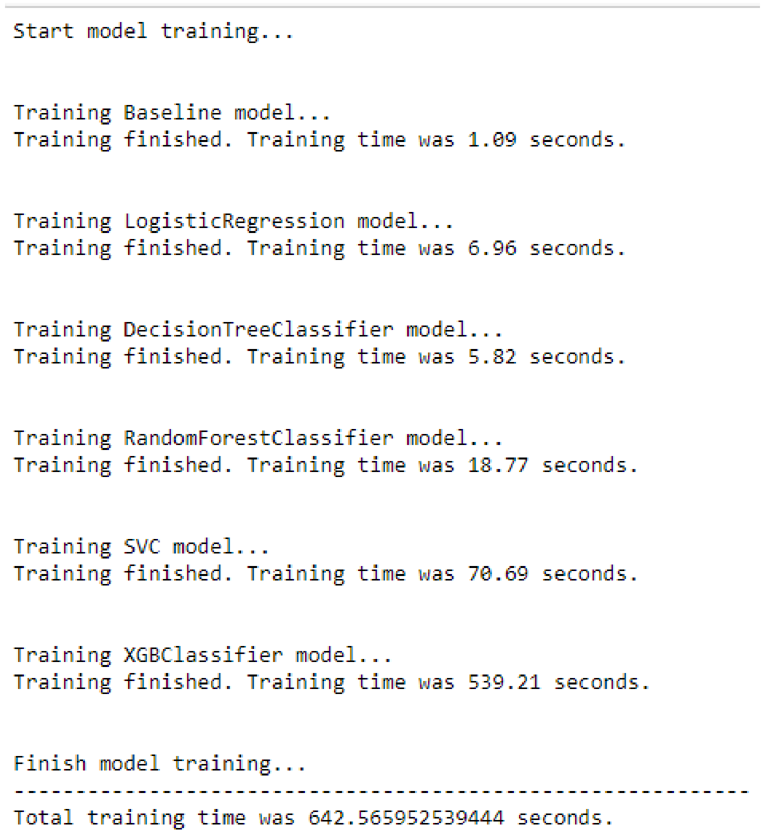
Of course, before preprocessing, the data is split into a training and test set, with 80% of the data going into the train set and 20% into the test set. This is a very common splitting method for traditional machine learning problems.

I then created different lists to hold all the pipelines and hyperparameter configurations that I wanted to train and created a loop structure that would tune me all my defined models. This basically provided me with an automated procedure to train all my models using cross validation and hyperparameter tuning. The implementation is shown in the figure below.



```
In [*]: log_cols=["Classifier", "Accuracy"]
        log = pd.DataFrame(columns=log_cols)

        training_start_time = time.time()

        print("Start model training...")
        print("\n")

        for pipe, parameters, model_type in zip(pipes,params,model_types):

            if model_type == "baseline":
                name = "Baseline"

            elif model_type == "tune":
                name = pipe[-1].__class__.__name__

            print(f"Training {name} model...")

            start_time = time.time()

            # Instantiate the GridSearchCV object: cv
            cv = GridSearchCV(pipe, parameters, scoring="accuracy", cv=10)

            # Fit the cv pipeline
            if model_type == "baseline":
                cv.fit(X_train_original, y_train_original)

            elif model_type == "tune":
                cv.fit(X_train, y_train)

            # Evaluate the model (calculate accuracy on test set)
            if model_type == "baseline":
                acc = cv.score(X_test_original, y_test_original)

            elif model_type == "tune":
                acc = cv.score(X_test, y_test)

            best_estimators[name] = cv.best_estimator_

            end_time = time.time()

            print(f"Training finished. Training time was {np.round(end_time-start_time,2)} seconds.")
            print("\n")

            log_entry = pd.DataFrame([[name, acc*100]], columns=log_cols)
            log = log.append(log_entry)

        training_end_time = time.time()

        print("Finish model training...")
        print("-------------------------------------------------------------")
        print(f"Total training time was {training_end_time-training_start_time} seconds.")
```

Figure 16: Model Pipeline Loop



```
Start model training...


Training Baseline model...
Training finished. Training time was 1.09 seconds.


Training LogisticRegression model...
Training finished. Training time was 6.96 seconds.


Training DecisionTreeClassifier model...
Training finished. Training time was 5.82 seconds.


Training RandomForestClassifier model...
Training finished. Training time was 18.77 seconds.


Training SVC model...
Training finished. Training time was 70.69 seconds.


Training XGBClassifier model...
Training finished. Training time was 539.21 seconds.


Finish model training...
-------------------------------------------------------------
Total training time was 642.565952539444 seconds.
```

Figure 17: Model Pipeline Loop Output

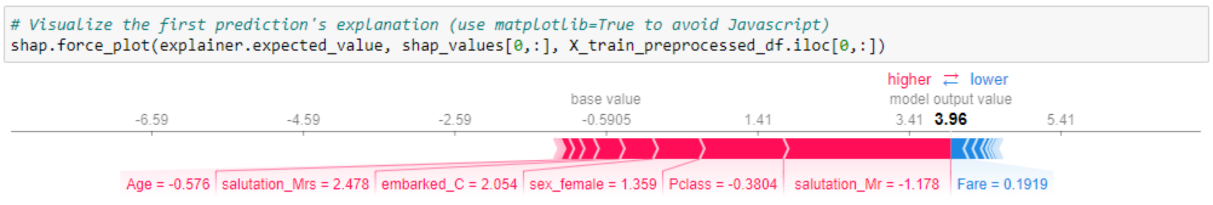
Figure 18: Training Pipeline Loop

Finally, I created a dataframe and seaborn plot to be able to nicely compare the different algorithms: The results of this can be found in the “Model Evaluation and Validation” section.

With regard to the model interpretation, I had multiple challenges to solve, which took me quite a while. The first challenge was that shap TreeExplainer only worked when I installed it via conda, not via pip as it has some not obvious dependencies. Also, in order to make nice plots, I needed a preprocessed dataframe **including column names**. My pipeline did not provide this out of the box as it transforms the data to a numpy array.

I therefore splitted the pipeline in separate tasks and added some logic to extract the column names.

The result, however, is really cool in my opinion. I can see the impact of different features for the prediction of an individual data point (here one example data point):



```
# Visualize the first prediction's explanation (use matplotlib=True to avoid Javascript)
shap.force_plot(explainer.expected_value, shap_values[0,:], X_train_preprocessed_df.iloc[0,:])
```

Figure 19: Shap Individual Prediction Explanation

All features that contributed to a higher prediction (meaning a higher chance of survival) are shown in red, all features that had effects in the contrary direction are shown in blue. I can for example see that the fact that the person did not have a “Mr” salutation, increased the chances of survival quite significantly. The same applies for the fact that the person was very young and was female (see values for Age and Sex). The fact that the person seemed to have paid quite a low fare had a little effect in the contrary direction.

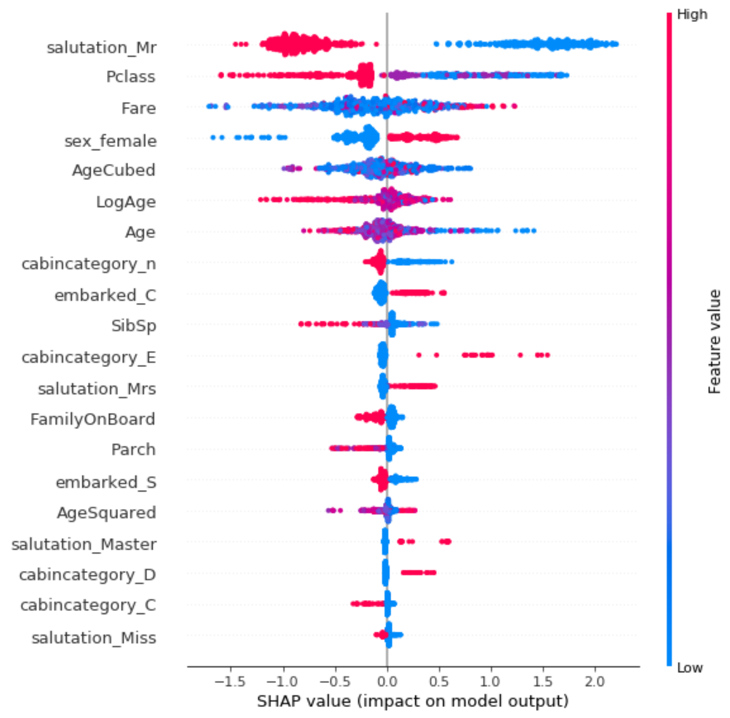


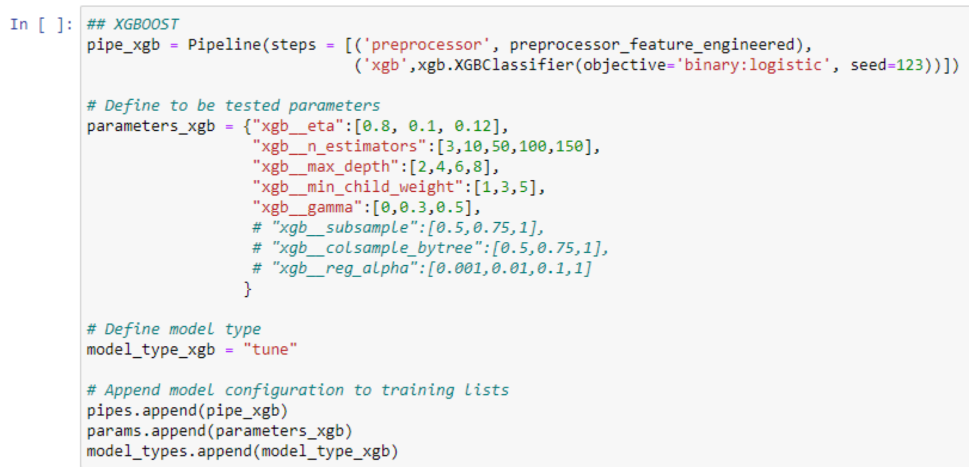
Figure 20: Shap Summary Plot

We can also see the effects of all features on the entire dataset. The color represents the feature value here (red high, blue low). This reveals for example that a high scaled value for "salutation_Mr" lowers the chances of survival quite significantly. A high scaled value for "Fare" increases the chances of survival.

Refinement

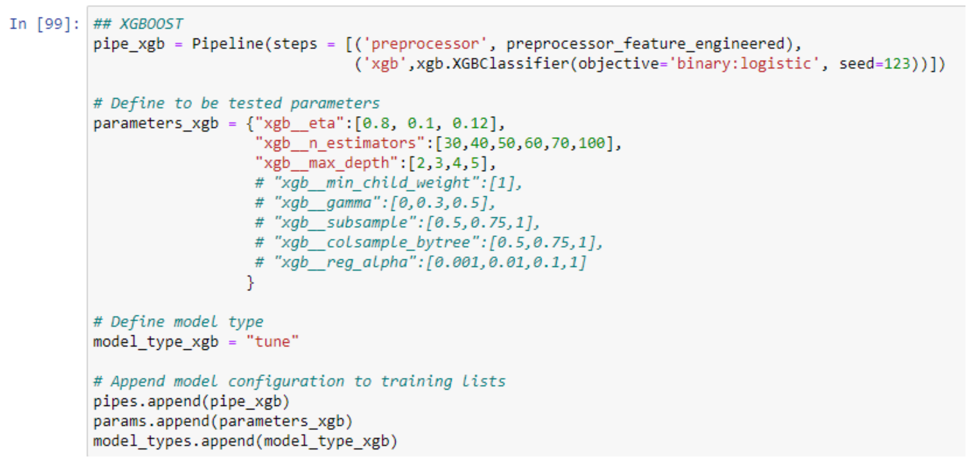
Earlier I described that I made use of the grid search technique to tune the hyperparameters of different model algorithms. However, even with such a small dataset, computation power is limited and it is prohibitively expensive to define too big of a hyperparameter grid (especially since I used my local machine to train models).

My approach was therefore to run multiple grid searches with small hyperparameter grids and only for a limited amount of hyperparameters to get a feeling for which hyperparameters work better and which are worth tuning. Then, in an iterative manner, I focused on improving these hyperparameters. This helped me to get very good results with less computation time/power. Without this method, since grid search trains several models for each possible hyperparameter combination, it is easy to get quite fast into thousands or even millions of trained models, particularly for algorithms that already make use of ensembling such as the xgboost algorithm. An example of this iterative hyperparameter tuning approach is shown in the figures below.



```
In [ ]: ## XGBOOST
        pipe_xgb = Pipeline(steps = [('preprocessor', preprocessor_feature_engineered),
                                     ('xgb',xgb.XGBClassifier(objective='binary:logistic', seed=123))])

        # Define to be tested parameters
        parameters_xgb = {"xgb__eta":[0.8, 0.1, 0.12],
                          "xgb__n_estimators":[3,10,50,100,150],
                          "xgb__max_depth":[2,4,6,8],
                          "xgb__min_child_weight":[1,3,5],
                          "xgb__gamma":[0,0.3,0.5],
                          # "xgb__subsample":[0.5,0.75,1],
                          # "xgb__colsample_bytree":[0.5,0.75,1],
                          # "xgb__reg_alpha":[0.001,0.01,0.1,1]
                         }

        # Define model type
        model_type_xgb = "tune"

        # Append model configuration to training lists
        pipes.append(pipe_xgb)
        params.append(parameters_xgb)
        model_types.append(model_type_xgb)
```

Figure 21: Iterative Hyperparameter Tuning Step 1



```
In [99]: ## XGBOOST
         pipe_xgb = Pipeline(steps = [('preprocessor', preprocessor_feature_engineered),
                                      ('xgb',xgb.XGBClassifier(objective='binary:logistic', seed=123))])

         # Define to be tested parameters
         parameters_xgb = {"xgb__eta":[0.8, 0.1, 0.12],
                           "xgb__n_estimators":[30,40,50,60,70,100],
                           "xgb__max_depth":[2,3,4,5],
                           # "xgb__min_child_weight":[1],
                           # "xgb__gamma":[0,0.3,0.5],
                           # "xgb__subsample":[0.5,0.75,1],
                           # "xgb__colsample_bytree":[0.5,0.75,1],
                           # "xgb__reg_alpha":[0.001,0.01,0.1,1]
                          }

         # Define model type
         model_type_xgb = "tune"

         # Append model configuration to training lists
         pipes.append(pipe_xgb)
         params.append(parameters_xgb)
         model_types.append(model_type_xgb)
```

Figure 22: Iterative Hyperparameter Tuning Step 2

Since in my first iteration, an n_estimators of 50 worked well, I tried values that are closer to this value to further optimize results.

Model Evaluation and Validation

As I made use of cross-validation in the model training process and therefore averaged the model performance over different splits of the dataset, one can assume that the calculated model performance is already quite robust. In addition, I repeated the model training for different random states of my train-test split. I can therefore quite confidently say that the resulting XGBoost model seems to be a quite robust candidate for a final model. Another idea could be to run a complete statistical analysis to get a better feeling for the noise, for example by assuming a normal distribution and testing on p-values of the results. However, this is out of the scope of this project because of computation power of my machine.

As promised in the section “Implementation”, I created a nice seaborn plot to evaluate my final model results.

```
<matplotlib.axes._subplots.AxesSubplot at 0x2cccacbeac8>
```

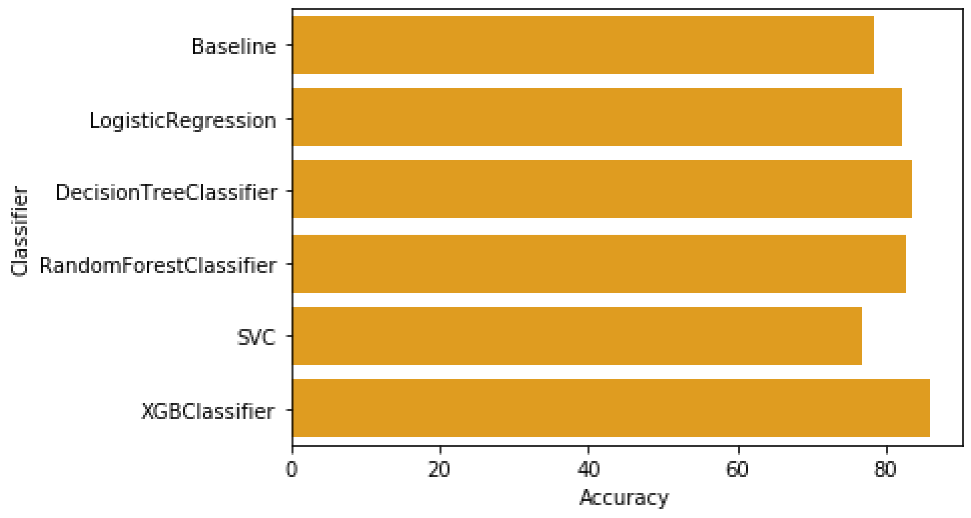


Figure 23: Final Model Evaluation Graph

	Classifier	Accuracy
0	Baseline	78.358209
1	LogisticRegression	82.089552
2	DecisionTreeClassifier	83.582090
3	RandomForestClassifier	82.835821
4	SVC	76.865672
5	XGBClassifier	85.820896

Figure 24: Final Model Evaluation Table

Justification

We can see in the figures above that my XGBClassifier reached an accuracy of 85.82%, more than 7% above the baseline model. In addition, I can also say that these results were quite constant during the hyperparameter tuning process. While sometimes the DecisionTreeClassifier or RandomForestClassifier also reached very good performance, the XGBClassifier seemed to be the most robust performer.

As mentioned, in order to further check for statistical robustness of this model I retrained the model under the optimal hyperparameter configuration multiple times with a different random state of the train-test-split.

We can therefore assume that the model performs (statistically) significantly better than the benchmark model (ballpark estimate even without a formal p-value test). Although I did not achieve the 90% accuracy which I wanted to come close to, I am still satisfied with my modeling approach and the improvement compared to the benchmark model.

One of the reasons why I did not come close to the best Kaggle performers is that they could train their models on the entire dataset and were provided an extra test set from Kaggle. This means that they could train their models on 10% more data, which is quite a significant advantage when the dataset is that small. Often the amount and quality of data going into an algorithm has an even bigger influence than which model or algorithm is used for prediction.