# Titanic - Machine Learning from Disaster

CSCE 4143 Final Project Report - Group 5

By: Ahmed Moustafa, Justin Kilgo, and Rohit Kala

# Introduction and Motivation

The RMS Titanic was a luxury British steamship that sank on April 15, 1912, after striking an iceberg. This disaster resulted in the deaths of more than 1,500 of the 2,240 passengers and crew [1]. As the ship was sinking, the only method of survival would be the lifeboats on the ship. Over an hour after the Titanic struck the iceberg, evacuation began with the 1st lifeboat. These rafts were designed to hold up to 65 people but the confusion and chaos resulted in it leaving with only 28. The law of the sea stated that women and children would be the first to board the lifeboats which may have affected their survival rate, yet this rule was not always maintained.

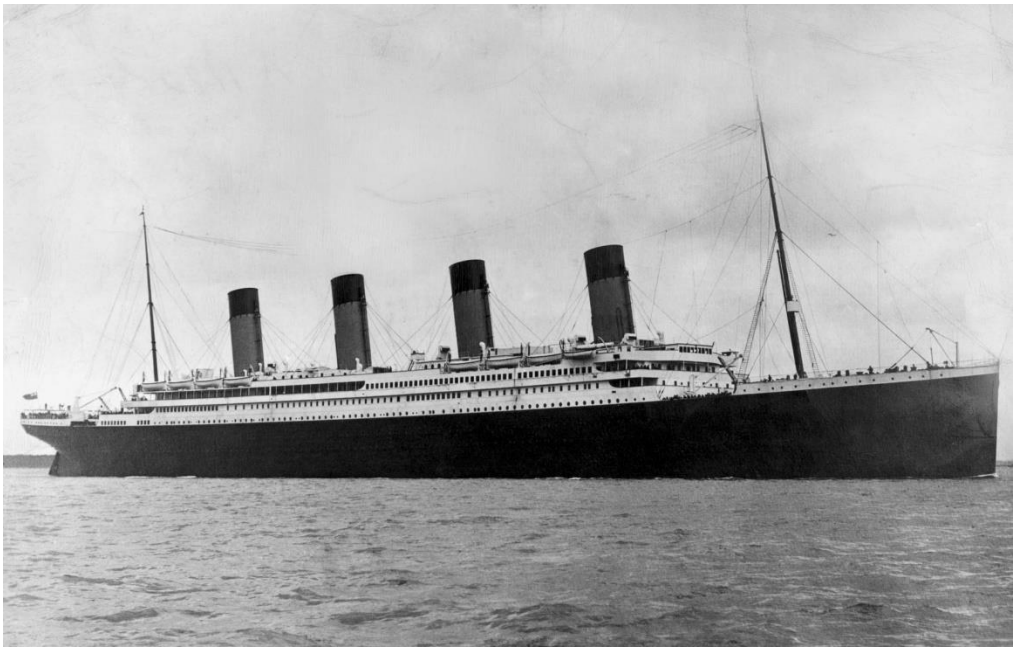

*Figure 1 – RMS Titanic*

From a data analysis perspective, we must consider the event itself as well as the information that was kept through the logbooks of passengers who purchased tickets. As news spread out of the disaster, Britain and the United States launched an investigation that produced extensive tables of passengers and crew from the available paper trail by July 1912. These tables contained passenger data that was broken down by age group, gender, class, and survival. Over 80 years later, the results of the investigation of the disaster would be organized and packaged for Data Analysis [2]. This conversion and packaging of paper data in December 1999 would later enable modern classification studies such as the 2012 Kaggle Machine Learning Competition. The context of the disaster and compilation of data is extremely important to understand for the sake of producing results in the Kaggle competition. There are many limitations resulting from the century-old paper-soured data including missing, unreadable, and possibly incorrect data but that adds to the motivation of this project. Can we take a look back into history and accurately determine if a passenger would survive the disaster solely based on the data from their boarding? If so, the results of this project might help recognize some patterns during the chaos of the

disaster and could even help us extract and apply some of those patterns to other historical disasters.

## Proposed Goals

As a participant in the "Titanic – Machine Learning from Disaster" Kaggle competition, our goal is to dive into some of the basic operations of Machine Learning to produce efficient and accurate methods of classifying passengers as survivors or not. To produce good results, we need to apply our knowledge of modern data analysis and classification through data exploration, preprocessing, feature engineering, model training, and evaluation. The biggest goal for our group is to dig deep into the detectable patterns of the Titanic data and experiment with some of the most practical and recent technologies/methods of Machine Learning. Our entire project with the notebook (code), presentation, and report is posted in this public GitHub repository: https://github.com/a-mufasa/CSCE-4143-Final-Project.

## Background and Related Work

Before going into the process of designing and implementing our solution we need to give context on some of the technologies we used and the related works/submissions that we researched for inspiration and feature definitions.

### Key Technologies

Before starting our project we had to look at all the options that we could use for data analysis and subsequent classification. Some of the main contenders were Weka, Python with Pandas library, and R. When weighing these options, we wanted to choose one that was easy to connect to Kaggle for result submission and allowed all group members to learn the Machine Learning concepts in a practical format. Weka, an application that contains tools for many data mining tasks like data preparation, classification, regression, and clustering, was a good option but would've been difficult to format the results in and connect to Kaggle when compared to a programming approach. Our group members had more experience with Python than R and found that the frameworks and libraries in Python were well-documented and accessible. As such, **we** chose to use **Python** for its durability, real-world applications, diverse libraries, and simple API integration with Kaggle.

For the sake of working together on the project and breaking down the code into easily readable blocks, we decided to use Jupyter Notebook which allowed us to document code as we built it. Furthermore, we created our notebook on Google Colab, an online method to run Jupyter Notebooks with connections to Google Drive and GitHub which allowed us to work together on different parts of the code. The biggest benefit of Google Colab is the free GPU, RAM, and processing capabilities and pre-downloaded libraries for quick use. The list of the Python libraries we used include Pandas for data manipulation, NumPy for linear algebra, Kaggle for data download and submission, Seaborn & Matplotlib for data visualization, and scikit-learn & xgboost for model training and evaluation.

```
# Imports
import pandas as pd
import numpy as np
import zipfile
import seaborn as sns
import matplotlib.pyplot as plt
import kaggle
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.model_selection import ShuffleSplit
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
```

*Figure 2 – All imports used in Notebook*

### Related Works

Since Kaggle's Titanic competition has been open since 2012, there have been over 14,000 people working to achieve the best solutions. As such, there are many submissions available online and articles that contain helpful research on the features to enable correct preprocessing and feature engineering. The main article we used specifically taught us the meaning of columns like Parch and SibSp which would enable feature engineering. They also explained Data Imputation well through the context of this competition which is the process of substituting values for missing data that does not hinder our later model training [3]. Furthermore, this article helped us break down the dataset and apply Data Mining processes which allowed us to surpass our initial plateau caused by limited preprocessing/data manipulation. One of these processes which we learned in the practice project was encoding categorical variables to use those variables in our classification models. This came up a lot in this project and we used dummy encoding which is very similar to one-hot encoding except with resultant columns compared to one-hot encoding's columns where  is the number of categories [4].

## Design and Implementation

The first thing we needed to do is download and extract the data from Kaggle which contained three files: **train.csv** which contained training data (features and target), **test.csv** which contained the feature data used to make predictions and send to Kaggle, and **gender_submission.csv** which was an example submission file for formatting. To do this we

used the Kaggle API to import the CSVs and converted them into training and testing Pandas DataFrames which enabled advanced data analysis. To start, we checked the types and basic statistics of our columns in the training DataFrame.

```
train_df = pd.read_csv('train.csv')
test_df = pd.read_csv('test.csv')

# Printing the data types of our columns and getting statistics for the DataFrame
print(train_df.dtypes, end='\n----------------------')
train_df.describe(include = 'all')
```

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
----------------------
```

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 891.000000 | 891.000000 | 891.000000 | 891 | 891 | 714.000000 | 891.000000 | 891.000000 | 891 | 891.000000 | 204 | 889 |
| unique | NaN | NaN | NaN | 891 | 2 | NaN | NaN | NaN | 681 | NaN | 147 | 3 |
| top | NaN | NaN | NaN | Braund, Mr. Owen Harris | male | NaN | NaN | NaN | 347082 | NaN | B96 B98 | S |
| freq | NaN | NaN | NaN | 1 | 577 | NaN | NaN | NaN | 7 | NaN | 4 | 644 |
| mean | 446.000000 | 0.383838 | 2.308642 | NaN | NaN | 29.699118 | 0.523008 | 0.381594 | NaN | 32.204208 | NaN | NaN |
| std | 257.353842 | 0.486592 | 0.836071 | NaN | NaN | 14.526497 | 1.102743 | 0.806057 | NaN | 49.693429 | NaN | NaN |
| min | 1.000000 | 0.000000 | 1.000000 | NaN | NaN | 0.420000 | 0.000000 | 0.000000 | NaN | 0.000000 | NaN | NaN |
| 25% | 223.500000 | 0.000000 | 2.000000 | NaN | NaN | 20.125000 | 0.000000 | 0.000000 | NaN | 7.910400 | NaN | NaN |
| 50% | 446.000000 | 0.000000 | 3.000000 | NaN | NaN | 28.000000 | 0.000000 | 0.000000 | NaN | 14.454200 | NaN | NaN |
| 75% | 668.500000 | 1.000000 | 3.000000 | NaN | NaN | 38.000000 | 1.000000 | 0.000000 | NaN | 31.000000 | NaN | NaN |
| max | 891.000000 | 1.000000 | 3.000000 | NaN | NaN | 80.000000 | 8.000000 | 6.000000 | NaN | 512.329200 | NaN | NaN |

*Figure 3 – Initial Column Analysis*

With all of our column statistics displayed, we now need to research what some of these columns actually represent to understand what features are actually relevant to determining survival. The first is the **Survived** column is the binary target column which tells us whether the passenger lived. Next is **Pclass** which is a categorical feature with 3 distinct classes (1-3) with 1 being those of higher status and 3 being those of lower status/commuters. **Sex and Age** are self-explanatory but **SipSp** and **Parch** are more difficult to define. These columns denote the number of siblings + spouse and number of parents + children respectively for each passenger. Next is **Ticket** which is not very relevant to predicting survival (thus we drop it later). **Fare** gives us the price of the ticket that that the passenger had to pay. The **Cabin** column is the room that the passenger was assigned to which consisted of a ship sector letter followed by the room number (ex: B96). Finally, the **Embarked** column tells us where the passenger boarded from which consisted of 3 cities: Southampton, Cherbourg, and Queenstown. Now that we know what all the columns stand for, we can do some data inspection and preprocessing.

```python
# Inspecting the Data

# Check for missing values in training DataFrame
print("Training Missing Values:")
print(train_df.isna().sum())

# Check for missing values  in testing DataFrame
print("----------------------\nTesting Missing Values:")
print(test_df.isna().sum())
```

```
Training Missing Values:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
----------------------
Testing Missing Values:
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

*Figure 4 – Null Checks*

We start by checking our Training DataFrame for missing values in each column. We find that there are 177 missing Age values, 687 Cabin values, and 2 missing Embarked values. Afterwards, we check our Testing DataFrame for missing values in each column. We find that there are 86 mising Age values, 1 missing Fare value, and 327 missing Cabin values.

## Data Preprocessing

The main goals of our Data Preprocessing are to clean our datasets of null values and encode features to allow our model to fit and predict using them. You'll see in the code blocks below, we expand our 12 columns into 69 columns using dummy encoding.

```
# Combine both our datasets to simplify preprocessing

df = train_df.append(test_df)
df.reset_index(inplace=True)
df.drop(['index'], inplace=True, axis=1)

# Extract the titles from Name and create a new Title Column with mapped titles
df['Title'] = df['Name'].map(lambda name:name.split(',')[1].split('.')[0].strip())
df['Title'] = df['Title'].map({
  "Capt": "Officer",
  "Col": "Officer",
  "Major": "Officer",
  "Jonkheer": "Royalty",
  "Don": "Royalty",
  "Sir" : "Royalty",
  "Dr": "Officer",
  "Rev": "Officer",
  "the Countess":"Royalty",
  "Mme": "Mrs",
  "Mlle": "Miss",
  "Ms": "Mrs",
  "Mr" : "Mr",
  "Mrs" : "Mrs",
  "Miss" : "Miss",
  "Master" : "Master",
  "Lady" : "Royalty"
})
```

*Figure 5 – Unify DataFrames and Title Mapping*

We start our preprocessing by combining the train and test DataFrames so that all our data manipulation occurs in both sets. There are 891 elements in the training dataset and 418 in the testing dataset. Since names are unique to each passenger, they aren't very valuable to our model since it is not easy to encode. Originally, we chose to just drop this entire column, but our submission results were not as high as we expected so we tried a different method. After analyzing this column, we found that all the names contain a title prefix (ex: "Mr. Owen Harris Braund"). In Figure 5 you can see that there were a total of 17 distinct titles which we then mapped to 5 categories (Officer, Royalty, Mrs. Miss, Mr, and Master) and added to a new Title column as part of our feature engineering.

```
# Group factors and their respective Age Medians

def get_age_median(row):
  train_median_df = df[:891].groupby(['Sex', 'Pclass', 'Title']).median().\
    reset_index()[['Sex', 'Pclass', 'Title', 'Age']]
  conditional = (
        (train_median_df['Sex'] == row['Sex']) &
        (train_median_df['Title'] == row['Title']) &
        (train_median_df['Pclass'] == row['Pclass'])
  )
  return train_median_df[conditional]['Age'].values[0]

df['Age'] = df.apply(lambda row: get_age_median(row) if np.isnan(row['Age']) else row['Age'], axis=1)
```

*Figure 6 – Age Data Imputation*

The next column we manipulate is Age which we saw had a very large number of missing values. Originally, we tried dropping all the null rows but lost a very big portion of our already small dataset which negatively impacted our models' accuracy. The solution to this was Data Imputation which we originally attempted by just replacing empty Age values with the mean of the non-missing values. We improved on this to avoid overfitting on incorrect/skewed ages by choosing a conditional median as a substitute instead. The **get_age_median** function in Figure 6 would use the Sex, Pclass, and Title values (never empty) for each row and if that row had a missing age, we fill it in with the median age of all the rows that have the same Sex, Pclass, and Title as these have decent correlation with Age.

```
#Dummy encode our Titles & drop Name Column

df.drop('Name', axis=1, inplace=True)

titles_dummies = pd.get_dummies(df['Title'], prefix='Title')
df = pd.concat([df, titles_dummies], axis=1)
df.drop('Title', axis=1, inplace=True)

# Replace empty Embarked values w/ the Mode (S) and Dummy encode the column

df.Embarked.fillna('S', inplace=True)

embarked_dummies = pd.get_dummies(df['Embarked'], prefix='Embarked')
df = pd.concat([df, embarked_dummies], axis=1)
df.drop('Embarked', axis=1, inplace=True)

# Set null Cabin to "X", set existing values to their 1st letter, and Dummy encode

df.Cabin.fillna('X', inplace=True)
df['Cabin'] = df['Cabin'].map(lambda c: c[0])

cabin_dummies = pd.get_dummies(df['Cabin'], prefix='Cabin')
df = pd.concat([df, cabin_dummies], axis=1)
df.drop('Cabin', axis=1, inplace=True)

# Dummy encode Pclass

pclass_dummies = pd.get_dummies(df['Pclass'], prefix="Pclass")
df = pd.concat([df, pclass_dummies],axis=1)
df.drop('Pclass',axis=1,inplace=True)
```

*Figure 7 – Dummy Encoding*

The next step of preprocessing is to dummy encode the categorical variables so that we can input our data into the models for training. First, we dummy encode the Title column we created and drop Title and Name. Next, we fill the null Embarked values we found with 'S' which is the mode of the column (the most common city of departure was Southampton). We replace nulls in Cabin with a placeholder 'X' to denote missing value and map each passenger's specific cabin/room number to a more generalized class of boat sector using the first character of the value. After cleaning up those columns we dummy encode Embarked, Cabin, and Pclass.

```
# Extract the ticket prefix and Dummy encode the column

def clean_ticket(ticket):
  ticket = ticket.replace('.','')
  ticket = ticket.replace('/','')
  ticket = ticket.split()
  ticket = map(lambda t : t.strip(), ticket)
  ticket = list(filter(lambda t : not t.isdigit(), ticket))
  if len(ticket) > 0:
      return ticket[0]
  else:
      return 'XXX'

df['Ticket'] = df['Ticket'].map(clean_ticket)

tickets_dummies = pd.get_dummies(df['Ticket'], prefix='Ticket')
df = pd.concat([df, tickets_dummies], axis=1)
df.drop('Ticket', inplace=True, axis=1)
```

*Figure 8 – Cleaning Tickets*

Our Ticket column has some specific cleaning that needs to happen before we can dummy encode/create categories from it. First, we remove any dots and slashes then split the ticket to extract the prefix if there is one. If the ticket only contains numbers then there is no prefix which we replace with a placeholder 'XXX' to denote. Afterwards we dummy encode our new Ticket column the same way we did our other columns.

```
# Create a Family Size column and columns that correspond to groups of sizes

df['FamilySize'] = df['Parch'] + df['SibSp'] + 1

df['Singleton'] = df['FamilySize'].map(lambda s: 1 if s == 1 else 0)
df['SmallFamily'] = df['FamilySize'].map(lambda s: 1 if 2 <= s <= 4 else 0)
df['LargeFamily'] = df['FamilySize'].map(lambda s: 1 if 5 <= s else 0)
```

*Figure 9 – Create and categorize FamilySize feature*

The only feature we engineer other than the Title from earlier is FamilySize which is the combination of the passenger, their parents/children, and their siblings/spouse. This value ended up ranging from 1 to 11 which would encode to too many columns. As such, we create 3 groups

to split the family sizes into relevant categories: Singleton (no family), SmallFamily (1-3 family members), and LargeFamily (4+ family members).

```
# Replace missing values with the average fare
df['Fare'].fillna(df.iloc[:891]['Fare'].mean(), inplace=True)

# Replace "male" with 1 and "female" with 0
df['Sex'] = df['Sex'].map({"male":1, "female":0})

# Split back into our two DataFrames
train_df = df.iloc[:891]
test_df = df.iloc[891:]
```

*Figure 10 – Final Preprocessing*

The final step of preprocessing is replacing all our missing Fare values with the mean of the non-null values (32.204208) and mapping sex to binary values as there were only 2 values in this dataset (male & female). After completing our preprocessing, we split our dataset back into the training and testing DataFrames.

## Data Visualization

With the preprocessing done, we can go back to analyzing the features and use Seaborn and Matplotlib to visualize some of the analysis.

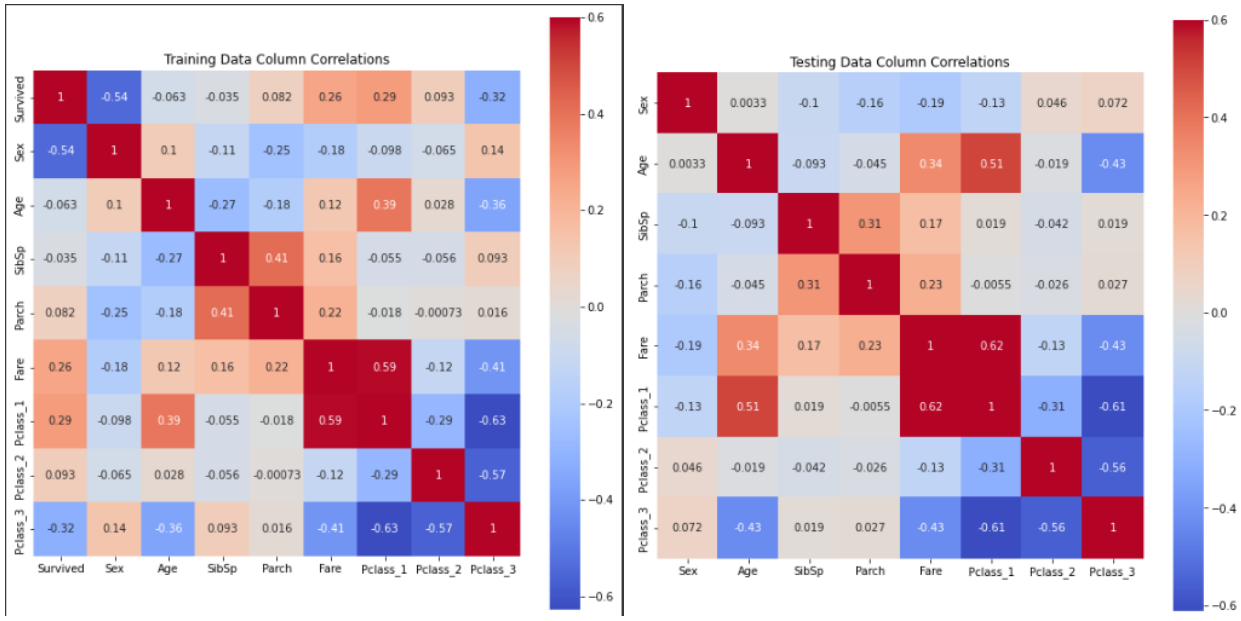


*Figure 11 – Seaborn Heatmaps on train/test original feature correlation*

The heatmaps in Figure 11 show the correlation between columns for our Training and Testing DataFrames. We can see some **similar patterns** for both sets with relatively strong negative correlations between Lower Classes & Fare and Lower Classes & Age. These are reasonable correlation as people in lower classes (3rd class) often cannot afford expensive tickets and

younger people are more likely to be lower class. We can see the opposite is true for Higher Classes & the mentioned columns as there are strong positive correlations.

When comparing Training & Testing correlations, it's important to note **differences** since they could play a large part in low accuracy of our models (also a sign of overfitting). A significant difference is that there seems to be a much higher correlation between Age and Fare for Testing than for Training.

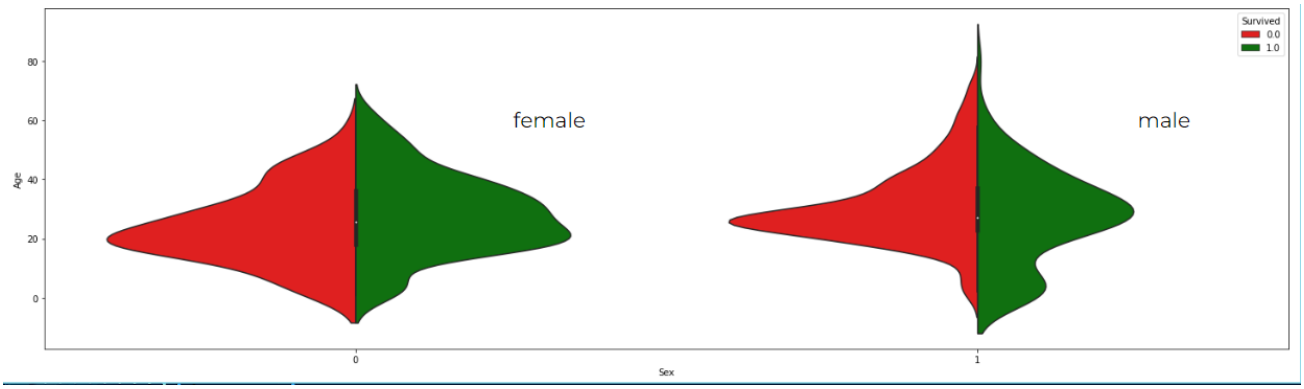


*Figure 12 – Seaborn Violin Plot for Age & Sex vs Survival*

Based on the Violin Plot above, we can see that women survive more than men (bigger green section). We can also see the distribution of age in the survival for each sex. The age distribution for men seems approximately normal, centered around ~30 but we see a jump in survival for young boys. There is also a large dip in survival for men around 18-20 which makes sense as they might have been working class commuters or “brave” sacrifices.

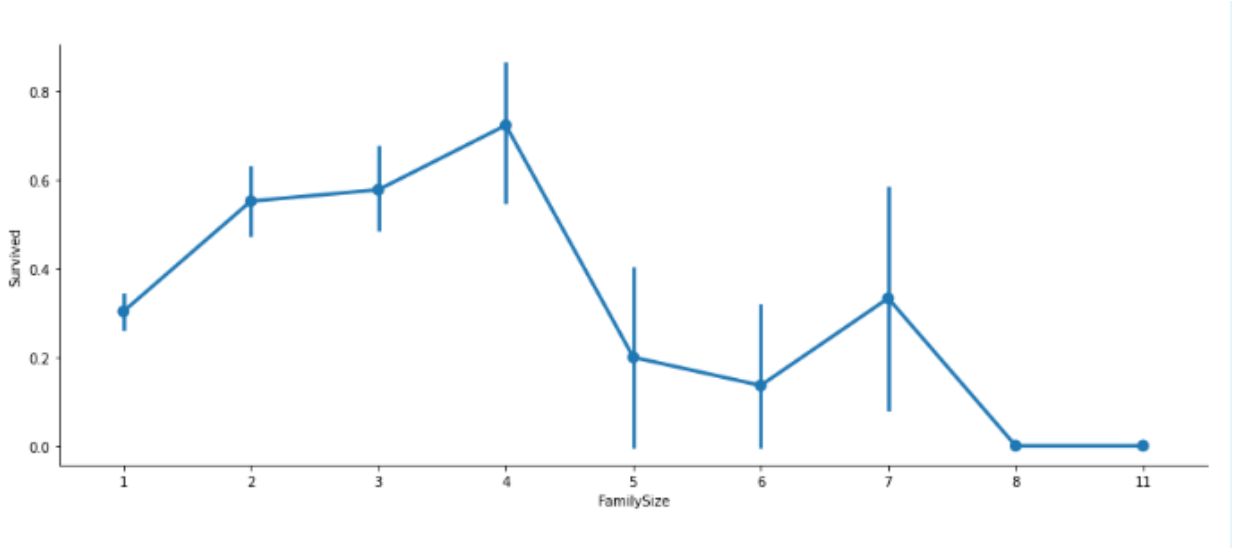


*Figure 13 – Seaborn Factor Plot for Family Size*

The last visualization we create is a Factor Plot for Family Size. This shows us the chances of survival with relation to size of your family (including self). Based on the plot, the peak survival

rates occurred with families of size 3-4. It seems that large families had low likelihood of survival with the biggest dropoff happening after 4 family members.

## Model Training

We used 4 models of different complexities to try and see the tradeoffs and accuracies for each when classifying: Logistic Regression, Random Forest, Extreme Gradient Boosting, and Decision Tree with a Bagging Classifier (meta-estimator).

```
# Create a Logistic Regression Model
lr = LogisticRegression(max_iter = 100000)

# Fit the Model using our training data
lr.fit(train_df[feature_columns], train_df[target_column])
```

*Figure 14 – Logistic Regression*

```
# RANDOM FOREST CLASSIFIER

rf = RandomForestClassifier(n_estimators=200, min_samples_leaf=3, max_features=.5, n_jobs=-1)
rf.fit(train_df[feature_columns], train_df[target_column])
```

*Figure 15 – Random Forest Classifier*

```
# EXTREME GRADIENT BOOSTING

xgb = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=1, gpu_id=0,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.03, max_delta_step=0, max_depth=8, min_child_weight=1,
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=0)
```

*Figure 16 – Extreme Gradient Boosting Classifier*

When picking out models to train on, we went in increasing complexity. To select the hyperparameters for these models, we played around with some of the relevant ones such as the number of estimators in the Random Forest Classifier and the base score in the XGB Classifier. Something to note is that for the XGB classifier, the majority of those hyperparameters are the default ones from the xgboost library but we listed them out since they didn't come from sklearn like our other classifiers. The main goal of this project was to experiment with modern

classification techniques using current libraries which lead us to try out a sklearn meta-estimator.

```
# DECISION TREE CLASSIFIER

from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

params = [{'max_leaf_nodes': list(range(2,100)), 'min_samples_split': [2, 3, 4]}]
b_clf=BaggingClassifier(GridSearchCV(DecisionTreeClassifier(random_state=42),params,cv=3,verbose=1),n_estimators=1000,max_samples=100,bootstrap=True,n_jobs=-1)
b_clf.fit(train_df[feature_columns],train_df[target_column])
y_pred=b_clf.predict(test_df[feature_columns]).astype(int)
```

*Figure 17 – Decision Tree with Bagging Classifier*

Our 4th and final model was an experimental one that we used to get some beginner/entry-level understanding of advanced classification techniques like meta-estimators. Sklearn's Bagging Classifier is defined as "an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions." [5] Some of the parameters we used for our Bagging Classifier were **bootstrap=True** and **n_jobs=-1**. Respectively, these allowed for samples to be drawn with replacement and all processors to run jobs in parallel to improve training speeds. Another method we used in this model was sklearn's GridSearchCV which exhaustively searches over a selection of parameter values for an estimator. The number of parameter value combinations we inputted (3 * 98 = 294) is not ideal but it allowed us to test the full capabilities of the search and estimator.

# Evaluation

After training our models on our processed DataFrames, we took the predictions and put them into a new DataFrame with the PassengerIds they corresponded to which matched the format required for the Kaggle submission. Afterwards, we converted them to CSVs and submitted to Kaggle to get the following scores.


*Figure 18 – Kaggle Scores*

Here we can see that our models' performance ranged from .75837 to .78708. Our models in order of performance from worst to best were Extreme Gradient Boosting, Logistic Regression,

Random Forest, and Decision Tree with Bagging Classifier. The BC + Dtree model took around 15 minutes to run compared to the near-instant Logistic Regression yet only resulted in an ~1% increase so the cost to accuracy benefit was less than expected.

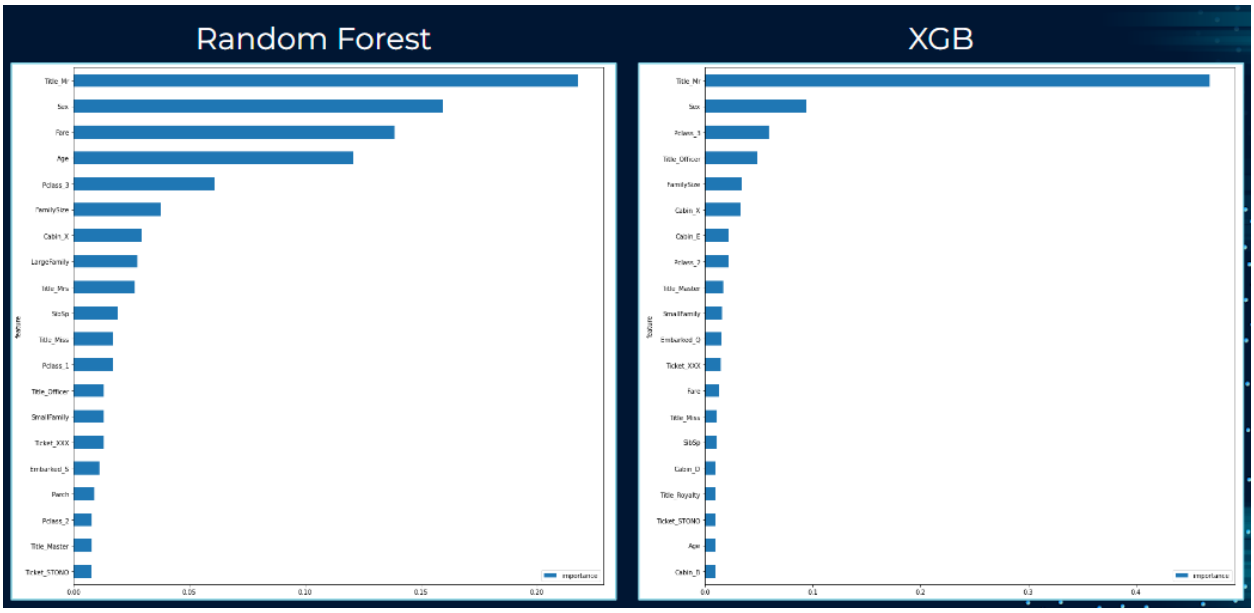


*Figure 19 – Feature Importance Plots*

For our Random Forest and XGB Classifiers, we plotted sklearn and xgboost's **feature_importances_** values to see which of the features had the greatest effect on our model's predictions. Figure 19 shows us that the Mr Title we extracted from the name column was actually the most important feature for predicting survival followed by Sex. The models differ greatly past these 2 features as the XGB classifier was much more dependent on the Mr Title than our RFC.

## Summary and Future Work

| | | | | | |
|---|---|---|---|---|---|
| 1271 | Mohammad Hossein Zardary |  | 0.78947 | 22 | 5h |
| 1272 | Bobo_0701 |  | 0.78708 | 4 | 2mo |
| 1627 | **Ahmed Moustafa** |  | 0.78708 | 5 | 17h |

*Figure 20 – Leaderboard Images*

In the end, our best submission tied with hundreds of other teams for 1271$^{st}$ place out of 14,171 which is in the top 10% of submissions.

| | | | | | |
|---|---|---|---|---|---|
| 140 | timvildanov | | 1.00000 | 1 | 3d |
| 141 | DANUSHKUMAR. V | | 1.00000 | 1 | 2d |
| 142 | sdg888 | | 1.00000 | 1 | 1d |

*Figure 21 – Cheaters*

If you account for the 140+ submissions which are cheated using an answer key posted online, we are in the top 8% or 92nd percentile of submissions. Furthermore, there are hundreds of submissions with a Kaggle score above .97 that use an extended dataset not provided for this Kaggle Competition which allowed for a better trained model.

Overall, we were able to experiment with many advanced Machine Learning and Classification techniques/technologies through this competition. Some goals we have for possible future work include more feature engineering as that seemed to make the largest difference between our older attempts and our best submission. Another goal is to fine tune our models by analyzing the hyperparameters more in parallel with a more powerful system (the limitation of Google Colab is that it is not very powerful). In conclusion, the skills we learned from this project and libraries that we gained experience with will be extremely useful for any future competitions or data analysis/classification that we attempt on other datasets.

## References


1. Editors, History com. "Titanic." *HISTORY*, www.history.com/topics/early-20th-century-us/titanic#&gid=ci0230e632a0212549&pid=rms-sailing-from-southampton.

2. Friendly, Michael, et al. "Visualising the*Titanic*Disaster." *Significance*, vol. 16, no. 1, Feb. 2019, pp. 14–19, 10.1111/j.1740-9713.2019.01229.x.

3. Mukhija, Sumit. "A Beginner's Guide to Kaggle's Titanic Problem." *Medium*, 22 June 2019, towardsdatascience.com/a-beginners-guide-to-kaggle-s-titanic-problem-3193cb56f6ca#:~:text=Embarked%20implies%20where%20the%20traveler. Accessed 9 Dec. 2022.

4. Pramoditha, Rukshan. "Encoding Categorical Variables: One-Hot vs Dummy Encoding." *Medium*, 16 Dec. 2021, towardsdatascience.com/encoding-categorical-variables-one-hot-vs-dummy-encoding-6d5b9c46e2db#:~:text=Both%20expand%20the%20feature%20space. Accessed 9 Dec. 2022.

5. "Sklearn.ensemble.BaggingClassifier — Scikit-Learn 0.23.1 Documentation." *Scikit-Learn.org*, scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html.

6. “Titanic - Machine Learning from Disaster.” *Kaggle.com*, www.kaggle.com/competitions/titanic.

7. “XGBoost Parameters — Xgboost 1.5.2 Documentation.” *Xgboost.readthedocs.io*, xgboost.readthedocs.io/en/stable/parameter.html.