ML Milestone 2 Report

Team id: CS_3

Team Members:

Student Name	**Seat Number**	**Department**
مينا خليفة جندي خله	20201700892	Computer Science
مينا لطفي فايز عبد الله	20201700895	Computer Science
كيرلس عزيز جلال عزيز	20201701116	Computer Science
ماري سعد يوسف سعد سليمان	20201700631	Computer Science
مينا انيس شكري معوض	20201700889	Computer Science
ميشيل مجدي حلمي زرق	20201700886	Computer Science

1. Preprocessing

Pre-Processing of Production Companies and Production Countries Columns

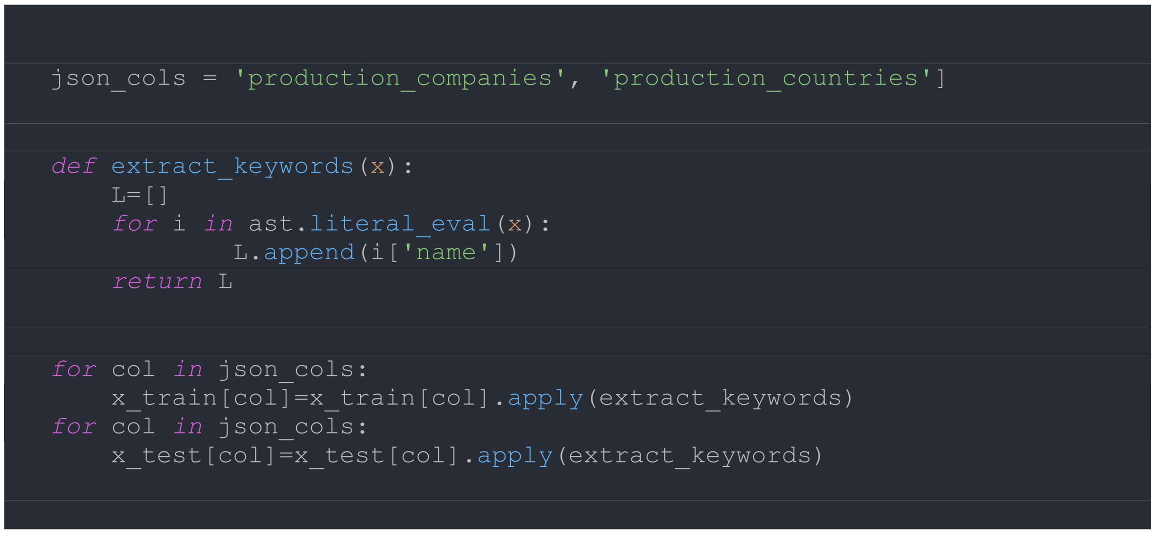
The pre-processing of the “**`production_companies`**” and "**`production_countries`**" columns involved several steps to extract meaningful information from these columns and transform them into one-hot encoded features. Here is a detailed explanation of each step:

1. **Extracting Names from Columns**

   The "production_companies" and "production_countries" columns were initially in JSON format, which contained multiple pieces of information such as the name, ID for "production_companies" and "iso_3166_1", country name for "production_countries".

   To extract only the names of the production companies and countries from these columns, we defined a function `extract_keywords(x)` that uses the `ast` library to convert the JSON string to a list of dictionaries.

   We then looped over the list and appended the name of each production company or country to a new list. Finally, this new list of names was returned by the function.



```
json_cols = 'production_companies', 'production_countries']

def extract_keywords(x):
    L=[]
    for i in ast.literal_eval(x):
            L.append(i['name'])
    return L

for col in json_cols:
    x_train[col]=x_train[col].apply(extract_keywords)
for col in json_cols:
    x_test[col]=x_test[col].apply(extract_keywords)
```

2. Frequency Analysis and Selection of Keywords

After extracting the names from the columns, we performed a frequency analysis to identify the most common keywords in both columns. This was done using the `Counter()` function from the `collections` library.

We then removed any stopwords and selected the 60 most common keywords for the "production_companies" column and the 21 most common keywords for the "production_countries" column. These keywords were stored in two separate sets, “`keywords_freq`” and “`freq_pro_countries`”.

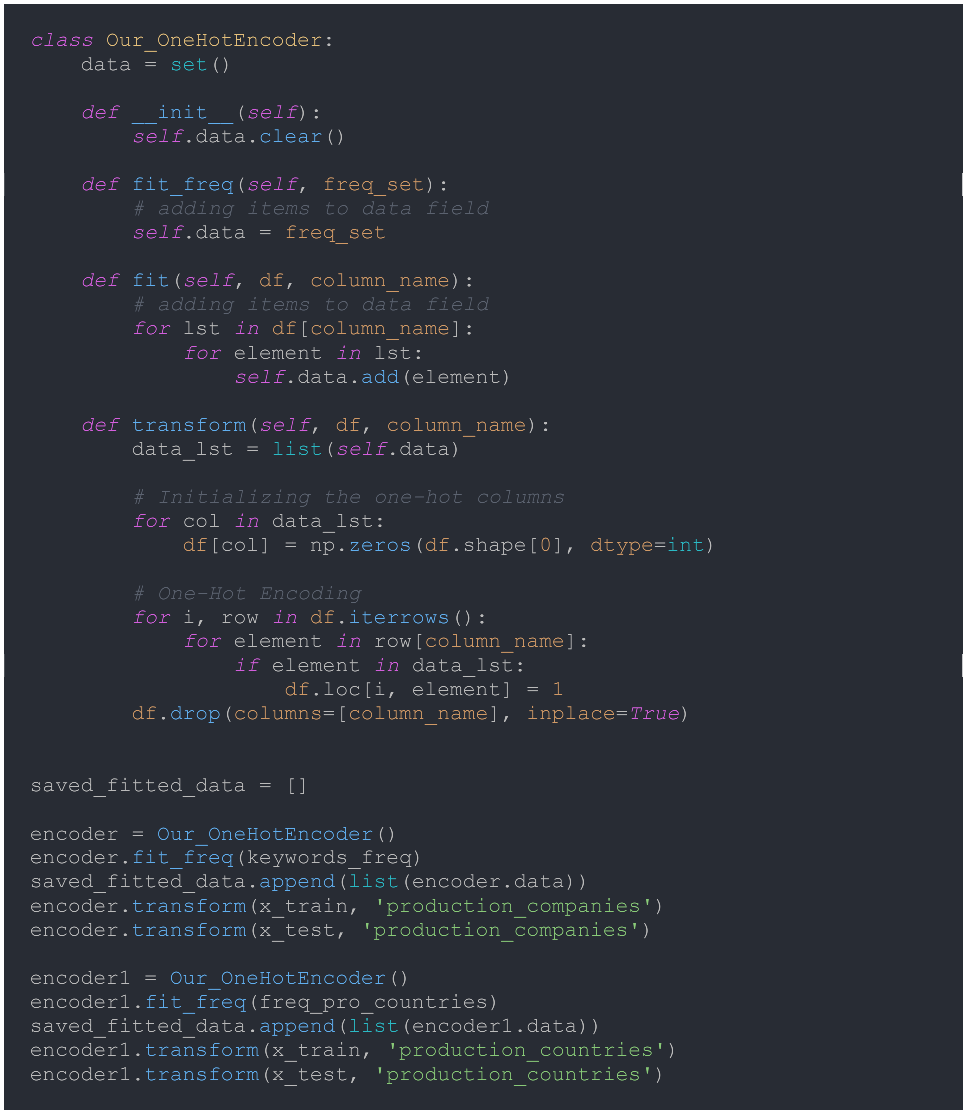
```python
from collections import Counter

# Set to store most frequent production companies keywords
keywords_freq = set()

# Flatten the list of keywords into a single list of words
words = [keyword for keyword_list in x_train['production_companies']
                 for keyword in keyword_list]

# Get the frequency count of each word
freq_count = Counter(words)

# Save the 60 most common words that not stopwords
for word, count in freq_count.most_common(60):
    if word not in stop_words:
        keywords_freq.add(word)


# Set to store most frequent production countries keywords
freq_pro_countries = set()

# Flatten the list of keywords into a single list of words
words = [keyword for keyword_list in x_train['production_countries']
                 for keyword in keyword_list]

# Get the frequency count of each word
freq_count = Counter(words)

# Save the 21 most common words that not stopwords
for word, count in freq_count.most_common(21):
    if word not in stop_words:
        freq_pro_countries.add(word)
```

3. **One-Hot Encoding**

Finally, we used our class `Our_OneHotEncoder` to transform the extracted keywords into one-hot encoded features. This class has three methods:

1. `fit_freq()` method fits the set of keywords obtained in step 2.
2. `fit()` method fits the keywords extracted from the columns for the training dataset.
3. `transform()` method transforms the extracted keywords into binary features, where each keyword corresponds to a column and contains a 1 if the keyword is present in the row, and 0 otherwise.

These methods were applied separately to both the training and testing datasets for the "production_companies" and "production_countries" columns.



```python
class Our_OneHotEncoder:
    data = set()

    def __init__(self):
        self.data.clear()

    def fit_freq(self, freq_set):
        # adding items to data field
        self.data = freq_set

    def fit(self, df, column_name):
        # adding items to data field
        for lst in df[column_name]:
            for element in lst:
                self.data.add(element)

    def transform(self, df, column_name):
        data_lst = list(self.data)

        # Initializing the one-hot columns
        for col in data_lst:
            df[col] = np.zeros(df.shape[0], dtype=int)

        # One-Hot Encoding
        for i, row in df.iterrows():
            for element in row[column_name]:
                if element in data_lst:
                    df.loc[i, element] = 1
        df.drop(columns=[column_name], inplace=True)


saved_fitted_data = []

encoder = Our_OneHotEncoder()
encoder.fit_freq(keywords_freq)
saved_fitted_data.append(list(encoder.data))
encoder.transform(x_train, 'production_companies')
encoder.transform(x_test, 'production_companies')

encoder1 = Our_OneHotEncoder()
encoder1.fit_freq(freq_pro_countries)
saved_fitted_data.append(list(encoder1.data))
encoder1.transform(x_train, 'production_countries')
encoder1.transform(x_test, 'production_countries')
```

2. The effect of HyperParameter tuning

For Hyper Parameter tuning for our models we used a class called `HyperTuning` that obtained the best parameters for the model based on the accuracy. The class searched for the combinations of parameters given in the `param_grid` and trained the model using them. Then, it evaluated the accuracy score of each combination and stored it in a dataframe. We selected the set of hyperparameters that resulted in the highest accuracy score as the best parameter set.

SVM:

For SVM we chose this `param_grid` = { 'kernel': ['linear', 'rbf', 'poly', 'sigmoid'], 'C': [1, 1.5, 2.0, 2.2, 2.4, 2.6,2.8], 'gamma': ['scale','auto'] } to use it in `HyperTuning` class, and conclude that:

For '**linear**' kernel, we can see that changing the values of '**C**' and '**gamma**' does not effect on the accuracy. The accuracy remains at range of 0.72 for all the values of 'C' and 'gamma'. This means that for linear kernel, 'C' and 'gamma' not determine the accuracy.

For '**rbf**' kernel, we can see changing the value of '**gamma**' have much effect on the accuracy as '**scale**' always give better accuracy than '**auto**'. And also the increase in the value of C make the accuracy increase until the value of 2.8 the accuracy start to decrease Therefore, we can say that 'C' and 'gamma' play a more significant role in determining accuracy for 'rbf' kernel than determine it for '**linear**' kernel.

For '**poly**' kernel, we can see that changing the values of both 'C' does not has much effect on the accuracy from the value 2.0 to more. For example, for 'C' value of 2.0, the changing of the value of '**gamma**' have big effect on the accuracy as '**scale**' always give better accuracy than '**auto**'. Therefore, we can say '**gamma**' play a more significant role than 'C' in determining accuracy for '**poly**' kernel.

For '**sigmoid**' kernel, in sigmoid there diffrence from rbf and poly as 'auto' for gamma give better accurracy that 'scale' and the value of C does not have anyy effect on the accurray

As result we found that the best accurracy come form 'Kernel': 'rbf' and as we conclude above that always 'scale' for 'gamma' give better accurracy so the best parameters are Params: {'kernel': 'rbf', 'C': 2.2, 'gamma': 'scale'} that give accuracy: 0.762

Logistic Regression:

For Logistic Regression We chose this param_grid: {'penalty': ['l2'], 'C': [0.1, 0.001, 1.5, 1.3, 1.4, 1.0, 0.9, 0.2, 0.3], 'solver':['lbfgs', 'liblinear', 'newton-cg','sag', 'saga']} to use it in HyperTuning class, Based on the hyperparameters and their corresponding accuracies, we can see that there is a range of possible values for 'C' that can give high accuracy. For example, when 'C' is equal to 0.1 or 1.3, the accuracy ranges from 0.733 to 0.738 depending on the solver used. Similarly, setting 'C' to 1.4, 1.5, or 0.9 can also result in high accuracy.

Additionally, we can see that the choice of solver has less impact on the accuracy than the value of 'C'. Among the solvers used, 'lbfgs', 'newton-cg', and 'sag' gave the highest accuracy scores for various values of 'C'. However, 'liblinear' and 'saga' resulted in lower accuracy scores across all tested values of 'C'.

Overall, the best combination of hyperparameters was Params: {'penalty': 'l2', 'C': 0.2, 'solver': 'newton-cg'} which resulted in an accuracy score of 0.74299. This indicates that using L2 regularization with 'newton-cg' as the solver and setting 'C' to 0.2 can result in the highest accuracy for the logistic regression model.

Gradient Boosting:

We choose this param_grid: param_grid = {'n_estimators': [40,41,42,43,44], 'max_depth': [1,2,3,4,5], 'learning_rate':[0.1]} to use it in HyperTuning class, Based on the hyperparameters and their corresponding accuracies, we can see that:

The first hyperparameter that was tuned was "**max_depth**", which controls the maximum depth of each decision tree in the model. The accuracy increased from 0.6820428336079077 to 0.728171334431631 when the value of "**max_depth**" was set to 3. More increasing the "**max_depth**" resulted in a slight decrease in accuracy for some values.

Next, the hyperparameter "**n_estimators**" was tuned. It controls the number of decision trees in the model. The accuracy increased steadily as the value of "**n_estimators**" increased up to 43, after which it started to be the same or decrese alittle bit.

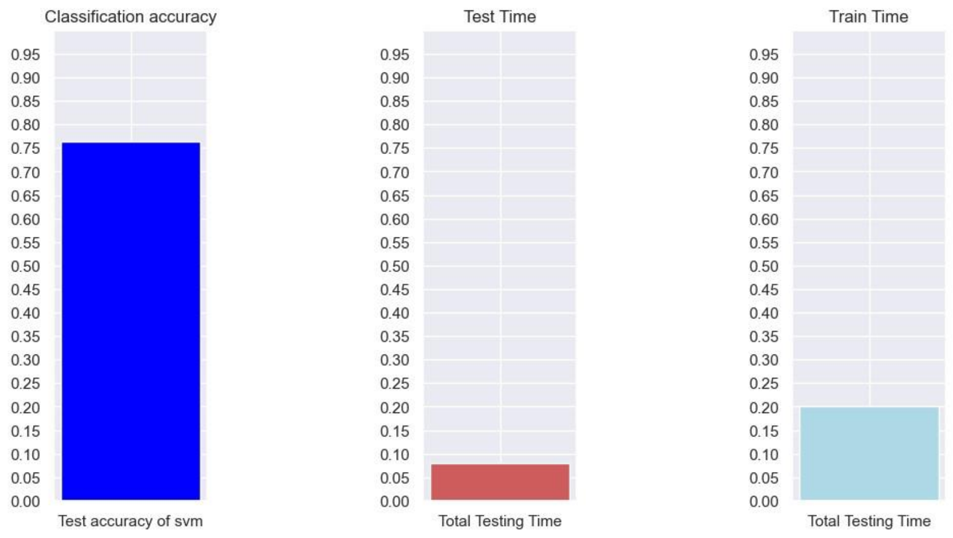
Finally, the "**learning_rate**" hyperparameter was tuned. The accuracy always increased largely when the "**learning_rate**" value increased.

Overall, the hyperparameter tuning significantly improved the model's performance, as the final accuracy of 0.7495881383855024 was higher than the initial accuracy of 0.6820428336079077. also, max_depth and learning_rate had the most significant impact on the model's performance, in the end the best parameters is {'n_estimators': 44, 'max_depth': 2, 'learning_rate': 0.3}

3. The classification accuracy, total test time and total training time Summarization

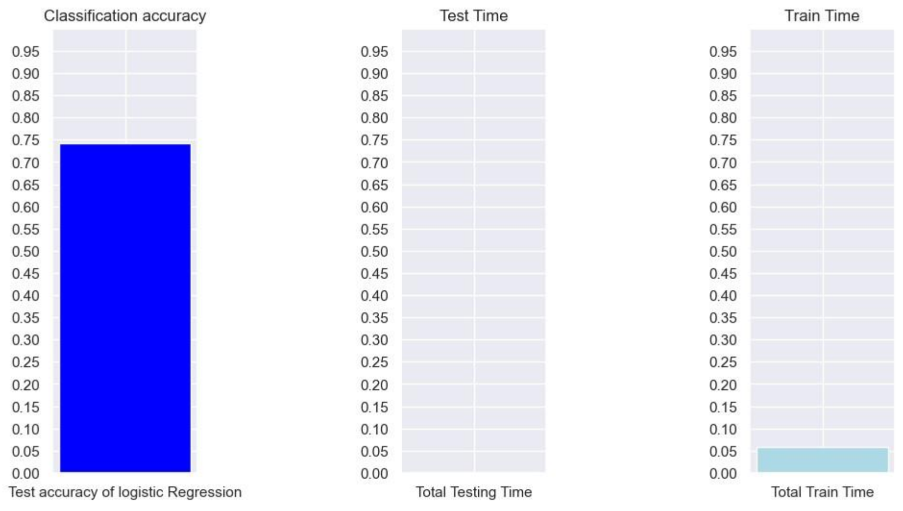
a) SVM:

a. **Model accuracy: 0.76**27677100494233
b. **Total test time: 0.08**037114143371582 seconds
c. **Total training time: 0.2**0023822784423828 seconds



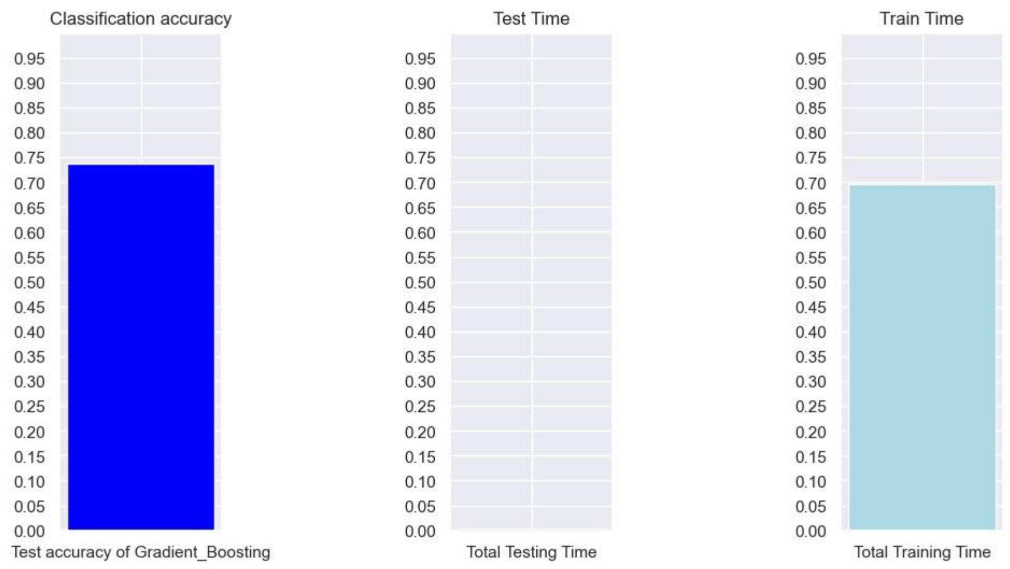
b) Logistic Regression:

a. **Model accuracy: 0.742**998352553542
b. **Total test time: 0.0**020401477813720703 seconds
c. **Total training time: 0.05**792641639709473 seconds



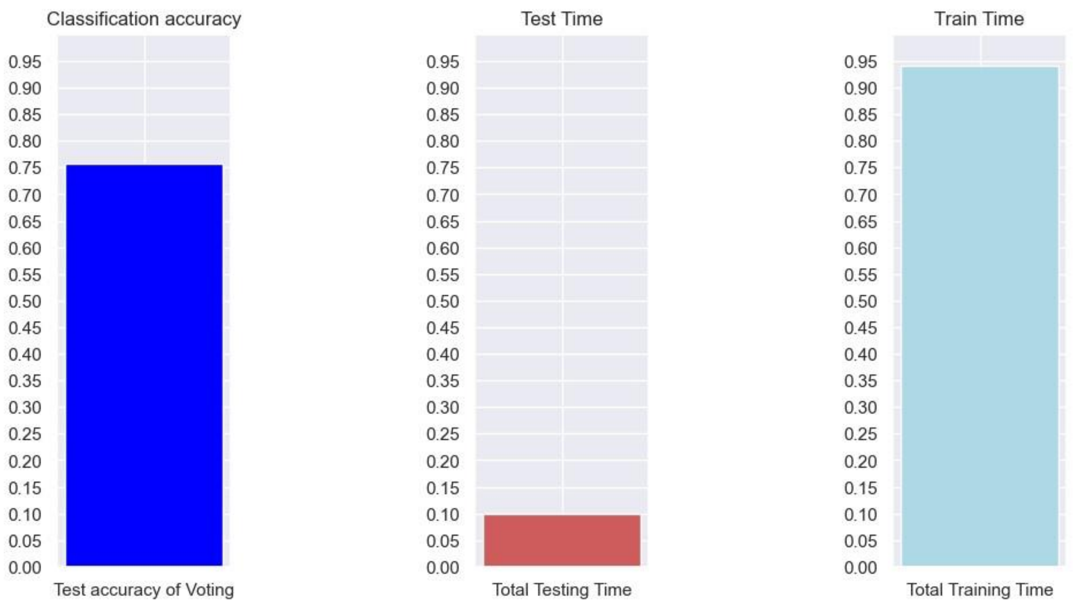
c) Gradient Boosting

1. **Model accuracy: 0.74**95881383855024
2. **Total train time:** 0.6962707042694092 seconds
3. **Total test time:** 0.00450444221496582 seconds



d) Voting

a. **Model accuracy: 0.75**78253706754531
b. **Total train time: 0.9**401323795318604 seconds
c. **Total test time: 0.09**814977645874023 seconds



4. Conclusion