Configuration Manual

MSc Research Project
Data Analytics

Marcelo Fischer
Student ID: 20118872

School of Computing
National College of Ireland

Supervisor: Rejwanul Haque

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Marcelo Fischer
Student ID:	20118872
Programme:	Data Analytics
Year:	2021
Module:	MSc Research Project
Supervisor:	Rejwanul Haque
Submission Due Date:	16/08/2021
Project Title:	Configuration Manual
Word Count:	688
Page Count:	36

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	15th September 2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Marcelo Fischer
20118872

1 Introduction

This configuration manual lists all hardware and software requirements to reproduce this research. The steps taken from data acquisition to model implementation are shown in this document.

2 Hardware and Software Requirements

Table 1 shows the hardware specifications used in the research. Table 2 shows the programming language used, the libraries used and their respective versions.

Table 1: Hardware Specifications.

RAM	32GB
Processor	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
OS	Windows 10 and Ubuntu 20.04

Table 2: Python Libraries and Versions.

Library	***Version***
Python	3.8.5
Jupyter Lab	3.0.14
pandas	1.2.4
numpy	1.19.2
re	2.2.1
tensorflow	2.3.0
keras	2.4.3
scikit learn	0.24.2
nltk	3.6.2

3 Dataset

3.1 Folder Structure

The files need to be inside the *project* folder as shown in Figure 1 for the paths in the code to work correctly. Also, the paths must be changed if executing inside a Linux or

Windows machine. Each file shown in Figure 1 will be explained in this manual.

Figure 1: Structure of the *project* folder. All shown files must be inside this folder for the files and paths to work.

3.2 Dataset Creation

The dataset was download from a Github repository which can be found at ***https://github.com/roneysco/Fake.br-Corpus***. The downloaded zip file contains three folders and a README file. The *full_texts* folder contains the full texts of the news, and also the metadata information about each label. The *size_normalized_texts* folder contains the truncated texts so that each fake-real pair has the same text length. The *preprocessed* folder contains a *.csv* file with three columns: index, label, and the pre-processed text. Pre-processed text means the removal of diacritic, accent, and Portuguese stopwords. Only the original full texts (the first folder) was used in this research.

Figure 2 shows the complete script to generate the master dataframe. For the code to work it is needed to have the *Fake.br-Corpus-master* folder inside the *project* folder. The necessary imports are shown at the top of the script.

4 Preprocessing

Before dealing with the actual data, some functions were defined to make the code cleaner and more organized. Figure 3 shows the necessary imports for the script. Figure 4 shows the function used to clean the texts. Figure 5 shows the function used to remove Portuguese stopwords from the texts. Figure 6 shows the function used to evaluate the models. Figure 7 shows the function used to save the models if wanted.

5 Experiments

5.1 Term Frequency - Full Texts Experiments

Figure 8 shows the necessary imports for these experiments. All TF experiments did not remove stopwords from the texts. Figure 9 shows how to load the data. Figure 10 shows the pre-processing steps and the creation of the train and test sets.

```python
from collections import defaultdict
from pathlib import Path
import pandas as pd

FAKE_FOLDER_PATH = "..\\project\\Fake.br-Corpus-master\\full_texts\\fake"
TRUE_FOLDER_PATH = "..\\project\\Fake.br-Corpus-master\\full_texts\\true"
SAVE_DF_FOLDER = "..\\project\\fakeBR_df.csv"

# Create the dateframe with the fake news
fakes = defaultdict(list)
for file in Path(FAKE_FOLDER_PATH).iterdir():
    with open(file, "r", encoding='utf-8') as f:
        fakes["file_name"].append(file.name)
        fakes["text"].append(f.read())
fake_df = pd.DataFrame(fakes)

# Create the dateframe with the real news
trues = defaultdict(list)
for file in Path(TRUE_FOLDER_PATH).iterdir():
    with open(file, "r", encoding='utf-8') as f:
        trues["file_name"].append(file.name)
        trues["text"].append(f.read())
true_df = pd.DataFrame(trues)

# Create the labels for both dataframes
fake_df['label'] = 1
true_df['label'] = 0

# Concatenate both dataframes to create the master dataframe
fakeBR_df = pd.concat([fake_df, true_df])

print(f"Shape of the fake_df dataframe: {fake_df.shape}")
print(f"Shape of the true_df dataframe: {true_df.shape}")

print(f"Saving the dataframe to {SAVE_DF_FOLDER}...")
fakeBR_df.to_csv(SAVE_DF_FOLDER, index=False)
```

Figure 2: Full Python script to generate the master dataframe.

```python
from collections import Counter

# Tools to wrangle natural language
import re
from nltk.corpus import stopwords

# Evaluation
from sklearn.metrics import accuracy_score, precision_score, f1_score, recall_score

from joblib import dump

```

Figure 3: Necessary imports for the *useful_funcs* script.

```python
def cleanText(text):
    """Remove urls, emails, digits, unicode chars, inverted commas and "-" from the given text.

    Args:
        text (str): text to be cleaned.

    Returns:
        clean_text (str): cleaned text.
    """
    ######## NORMALIZATION ########
    # https://www.tutorialexample.com/best-practice-to-extract-and-remove-urls-from-python-string-python-tutorial/
    clean_text = re.sub(
        r'(?i)\b((?:[a-z][\w-]+:(?:/{1,3}|[a-z0-9%])|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))*\))+(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]{};:\'".,<>?«»“”‘’]))', 'url', text)  # change any url to the text 'url'
    clean_text = re.sub(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', 'email', clean_text) # change any email to the text 'email'
    # consider only letters and make the whole clean_text lower case
    clean_text = re.sub(r"[^a-zA-Z0-9]", " ", clean_text.lower())
    # remove words that contain numbers
    clean_text = re.sub('\w*\d\w*', '', clean_text)
    clean_text = re.sub('\[.*?\]', '', clean_text)  # remove square brackets
    clean_text = re.sub('\ufeff', '', clean_text)  # remove unicode chars
    clean_text = re.sub('\u200b', '', clean_text)  # remove unicode chars
    clean_text = re.sub('[‘’“”…]', '', clean_text)  # remove inverted commas
    clean_text = clean_text.replace("-", "")
    # Replace all single characters with a space
    # clean_text = re.sub(r'\b[a-zA-Z]\b', ' ', clean_text)
    # Replace all double spaces with one space
    clean_text = re.sub(' +', ' ', clean_text)
    clean_text = clean_text.strip()  # Remove leading and trailing spaces

    return clean_text
```

Figure 4: Function used to clean the texts.

```python
def removeStopwords(text):
    """Remove Portuguese stopwords from the given text.

    Args:
        text (str): text with stopwords.

    Returns:
        text (str): text without the stopwords in it.
    """
    ######## STOPWORDS REMOVAL ########
    stop_words = stopwords.words('portuguese')
    stopwords_dict = Counter(stop_words)
    text = ' '.join(
        [word for word in text.split() if word not in stopwords_dict])

    return text
```

Figure 5: Function used to remove the Portuguese stopwords from the text.

```python
def modelEval(model, y_test, predictions):
    """Prints out the accuracy, f1_score, recall and precision for the given {model}."

    Args:
        model (str): name of the model being evaluated.
        y_test (array): array containing the test labels.
        predictions (array): array containing the predictions for the given model.
    """
    print(
        f"The accuracy for the {model} model is: {round(accuracy_score(y_test, predictions), 4)}")
    print(
        f"The f1_score for the {model} model is: {round(f1_score(y_test, predictions), 4)}")
    print(
        f"The recall for the {model} model is: {round(recall_score(y_test, predictions), 4)}")
    print(
        f"The precision for the {model} model is: {round(precision_score(y_test, predictions), 4)}")
```

Figure 6: Function used to evaluate the models.

```python
def saveModels(model, model_file_path):
    """Save a given machine learning model to a file in the model_file_path folder.

    Args:
        model (sklearn model): a trained machine learning model to be saved for later use.
        file_path (str): path of the folder + name of the file to save the model.
    """
    print(
        f"Saving the {model} model in {model_file_path} for later use...")
    dump(model, model_file_path)
    print(f"The {model} model has been saved.")

```

Figure 7: Function used to save the models.

```python
import pandas as pd
import numpy as np
from useful_funcs import cleanText, modelEval, saveModels

# Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.feature_extraction.text import CountVectorizer
import xgboost as xgb

# Graphics
import matplotlib.pyplot as plt
```

Figure 8: Necessary imports for the TF experiments.

```python
path_linux = '../project/fakeBR_df.csv'
path_windows = "..\\project\\fakeBR_df.csv"

fakeBR_df = pd.read_csv(path_windows)
fakeBR_df.head()
```

Figure 9: Loading the dataframe inside the environment.

```python
##### DATA PREPROCESSING FOR THE MODELS #####
fakeBR_df['text'] = fakeBR_df['text'].apply(cleanText)

##### CREATE THE X AND Y SETS #####
x = fakeBR_df['text']
y = fakeBR_df['label']

##### TF #####
# Create the TF Vectorizer object
tf_vectorizer = CountVectorizer()

# Make the sparse matrix
cv = tf_vectorizer.fit_transform(x)

##### TRAIN TEST SPLIT #####
x_train, x_test, y_train, y_test = train_test_split(cv, y, test_size=0.2, random_state=42)

print(f"Shape of the train data: {x_train.shape}")
print(f"Shape of the test data: {x_test.shape}")
print(f"Shape of the train labels: {y_train.shape}")
print(f"Shape of the test labels: {y_test.shape}")
```

Figure 10: Pre-processing steps and creation of train and test sets for the full texts.

Next, all machine learning models were trained. Their code snippets are shown below. For all models, the parameters were optimized with the use of grid search and the code snippets for these are also shown. Figure 11 depicts an example of the output of the *modelEval* function.

The KNN model was not optimized by the use of grid search but by the use of the elbow method. Figure 24 shows the method.

5.2 Term Frequency - Truncated Texts Experiments

All models and the code flow were exactly the same as shown for the Section 5.1. The only difference was in the `CountVectorizer` parameter *max_features* that was set to 200, which can be seen in Figure 37.

5.3 Term Frequency-Inverse Document Frequency Experiments

All models were trained and optimized exactly like shown in Section 5.1. The only differences are in the imports (Figure 38) and in the pre-processing steps (Figure 39). The only difference when considering truncated texts is that the *max_features* parameter of the `TfidfVectorizer` is set to 200.

5.4 Neural Networks

Figure 40 shows the necessary imports for these experiments. Figure 41 shows how to load the dataframe and define the path to the folder where the models will be saved. Figure 42 shows the pre-processing for these experiments. Two different scenarios were tried, with and without stopwords. The only difference in the code from one to the other is the third line in Figure 42 which is removed when the stopwords are not removed from

Logistic Regression

```python
lr = LogisticRegression(random_state=42, max_iter=500)

lr.fit(x_train, y_train)

lr_predictions = lr.predict(x_test)

modelEval("Logistic Regression", y_test, lr_predictions)
```

```
The accuracy for the Logistic Regression model is: 0.9694
The f1_score for the Logistic Regression model is: 0.9694
The recall for the Logistic Regression model is: 0.9721
The precision for the Logistic Regression model is: 0.9668
```

Figure 11: Standard logistic regression model.

```python
%%time
lr_params = {'C': [0.01, 0.1, 1, 10, 100]}
lr_search = GridSearchCV(LogisticRegression(random_state=42, max_iter=500),
                         param_grid = lr_params,
                         n_jobs = -1,
                         cv = 5,
                         verbose=1)

lr_search.fit(x_train, y_train)
```

Figure 12: Grid search for the logistic regression.

```python
lr_search.best_params_
```

```
{'C': 0.1}
```

```python
lr_best = LogisticRegression(C=0.1, max_iter=500, random_state=42)

lr_best.fit(x_train, y_train)

lr_best_predictions = lr_best.predict(x_test)

modelEval("best Logistic Regression", y_test, lr_best_predictions)
```

Figure 13: Optimized parameters for the logistic regression.

Decision Tree ¶

```python
dt = DecisionTreeClassifier(random_state=42)

dt.fit(x_train, y_train)

dt_predictions = dt.predict(x_test)

modelEval("Decision Tree", y_test, dt_predictions)
```

Figure 14: Standard decision tree model.

the texts. The rest of the images show the neural networks architectures, parameters and callbacks.

5.5 mBERT

The mBERT model was run using an online interface at https://platform.peltarion.com/. From Figure 56 to Figure 65 it is possible to see the necessary steps to reproduce the model.

```
%%time
# Measure the qualit of each split
criterion = ['gini', 'entropy']
# The strategy used to choose the split at each node
splitter = ['best', 'random']
# Number of features to consider at every split
max_features = [None, 'auto', 'log2']
# Maximum number of levels in the tree
max_depth = [None, 2, 3, 4, 5, 6]
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 7, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]

# Create the random grid
dt_params = {'criterion': criterion,
                  'splitter': splitter,
                  'max_features': max_features,
                  'max_depth': max_depth,
                  'min_samples_split': min_samples_split,
                  'min_samples_leaf': min_samples_leaf}

dt_search = GridSearchCV(DecisionTreeClassifier(random_state=42),
                         param_grid = dt_params,
                         n_jobs = -1,
                         cv = 5,
                         verbose=1)

dt_search.fit(x_train, y_train)
```

Figure 15: Grid search for the decision tree.

```
dt_search.best_params_
```

```
{'criterion': 'entropy',
 'max_depth': 5,
 'max_features': None,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'splitter': 'best'}
```

```
dt_best = DecisionTreeClassifier(splitter = 'best',
                                 min_samples_split = 2,
                                 min_samples_leaf = 2,
                                 max_features = None,
                                 max_depth = 5,
                                 criterion = 'entropy',
                                 random_state=42)

dt_best.fit(x_train, y_train)

dt_best_predictions = dt_best.predict(x_test)

modelEval("best Decision Tree", y_test, dt_best_predictions)
```

Figure 16: Optimized parameters for the decision tree.

Support Vector Machine

```
linear_svc = LinearSVC(random_state=42)

linear_svc.fit(x_train, y_train)

linear_svc_predictions = linear_svc.predict(x_test)

modelEval('LinearSVC', y_test, linear_svc_predictions)
```

Figure 17: Standard LinearSVC model.

```python
%%time
# Create the regularization parameters for the SVC model.
C_values = [0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1, 0.3, 0.6]

# Create the random grid
linear_svc_params = {'C': C_values}

linear_svc_search = GridSearchCV(LinearSVC(random_state=42, max_iter=50000),
                                 param_grid = linear_svc_params,
                                 n_jobs=-1,
                                 cv = 5,
                                 verbose=1)

linear_svc_search.fit(x_train, y_train)
```

Figure 18: Grid search for the LinearSVC.

```python
linear_svc_search.best_params_
```

```
{'C': 0.006}
```

```python
linear_svc_best = LinearSVC(C = 0.006, random_state=42)

linear_svc_best.fit(x_train, y_train)

linear_svc_best_predictions = linear_svc_best.predict(x_test)

modelEval("best LinearSVC", y_test, linear_svc_best_predictions)
```

Figure 19: Optimized parameters for the LinearSVC.

```python
svc = SVC(random_state=42)

svc.fit(x_train, y_train)

svc_predictions = svc.predict(x_test)

modelEval('SVC', y_test, svc_predictions)
```

Figure 20: Standard SVC model.

```
%%time
# Create the random grid
svc_params = {'C': C_values,
              'kernel': ['linear', 'rbf']}

svc_search = GridSearchCV(SVC(random_state=42),
                          param_grid = svc_params,
                          n_jobs = -1,
                          cv = 5,
                          verbose=1)

svc_search.fit(x_train, y_train)
```

Figure 21: Grid search for the SVC.

```
svc_search.best_params_
```

```
{'C': 0.01, 'kernel': 'linear'}
```

```
svc_best = SVC(C = 0.01, kernel = 'linear', random_state=42)

svc_best.fit(x_train, y_train)

svc_best_predictions = svc_best.predict(x_test)

modelEval("best SVC", y_test, svc_best_predictions)
```

Figure 22: Optimized parameters for the SVC.

K-Nearest Neighbours

```python
error_rate = []

for i in range(1,40):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(x_train, y_train)
    pred_i = knn.predict(x_test)
    error_rate.append(np.mean(pred_i != y_test))
```

Figure 23: Standard KNN model.

```python
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='blue', linestyle='dashed', marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K')
plt.xlabel('K')
plt.ylabel('Error Rate')
plt.plot()
```

```
[]
```

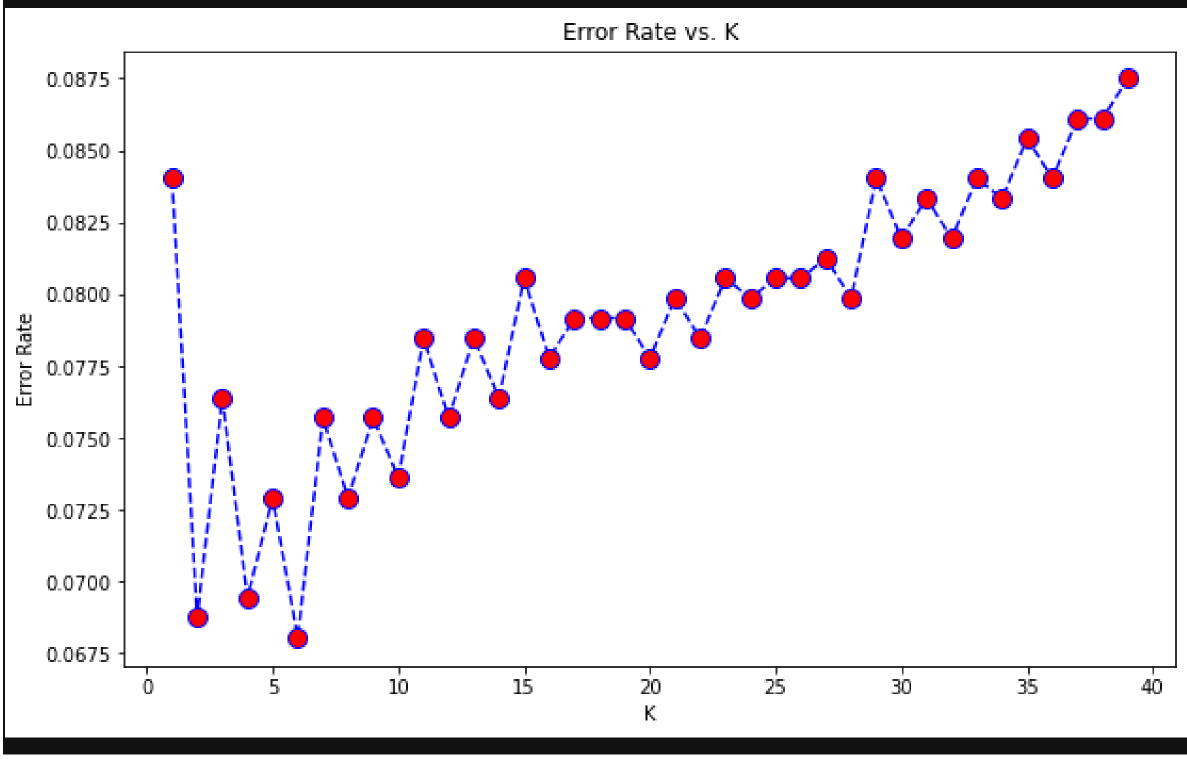

Figure 24: Elbow method for the KNN.

```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(x_train, y_train)

knn_predictions = knn.predict(x_test)

modelEval("KNN", y_test, knn_predictions)
```

Figure 25: Optimized parameters for the KNN.

Random Forest

```
# Basic random forest model
rf1 = RandomForestClassifier(n_jobs=4, random_state=42)
rf1.fit(x_train, y_train)

rf1_predictions = rf1.predict(x_test)

modelEval('Random Forest', y_test, rf1_predictions)
```

Figure 26: Standard random forest model.

```
%%time
# Number of trees in random forest
n_estimators = list(range(10, 160, 20))
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [2**x for x in range(6)]
max_depth.append(None)
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
rf_params = {'n_estimators': n_estimators,
             'max_features': max_features,
             'max_depth': max_depth,
             'bootstrap': bootstrap}

rf_search = GridSearchCV(RandomForestClassifier(random_state=42),
                         param_grid = rf_params,
                         n_jobs = -1,
                         cv = 5,
                         verbose=1)

rf_search.fit(x_train, y_train)
```

Figure 27: Grid search for the random forest.

```
rf_search.best_params_
```

```
{'bootstrap': False,
 'max_depth': None,
 'max_features': 'auto',
 'n_estimators': 150}
```

```
rf_best = RandomForestClassifier(n_estimators = 150,
                                 max_features = "auto",
                                 max_depth = None,
                                 bootstrap = False,
                                 random_state = 42)
rf_best.fit(x_train, y_train)

rf_best_predictions = rf_best.predict(x_test)

modelEval('best Random Forest', y_test, rf_best_predictions)
```

Figure 28: Optimized parameters for the random forest.

Multinomial Naive Bayes

```
nb = MultinomialNB()

nb.fit(x_train, y_train)

nb_predictions = nb.predict(x_test)

modelEval('Multinomial Naive Bayes', y_test, nb_predictions)
```

Figure 29: Standard Naive Bayes model.

```
%%time
nb_params = {'alpha': [0.0001, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1]}
nb_search = GridSearchCV(MultinomialNB(),
                         param_grid = nb_params,
                         n_jobs = -1,
                         cv = 5,
                         verbose=1)

nb_search.fit(x_train, y_train)
```

Figure 30: Grid search for the Naive Bayes.

```
nb_search.best_params_
```

```
{'alpha': 0.1}
```

```
nb_best = MultinomialNB(alpha=0.1)

nb_best.fit(x_train, y_train)

nb_best_predictions = nb_best.predict(x_test)

modelEval('best Multinomial Naive Bayes', y_test, nb_best_predictions)
```

Figure 31: Optimized parameters for the Naive Bayes.

Stacking

```
%%time
estimator_lst = [
    ("lr", lr_best),
    ("dt", dt_best),
    ("rf", rf_best),
    ("linear_svc", linear_svc_best),
    ('SVC', svc_best),
    ("knn", knn),
    ('nb', nb_best)
]

stack_model = StackingClassifier(estimators=estimator_lst,
                                 final_estimator=LinearSVC(random_state=42, max_iter=500000),
                                 n_jobs = -1)

stack_model.fit(x_train, y_train)
```

Figure 32: Stacking model.

```
stack_model_predictions = stack_model.predict(x_test)

modelEval("Stacking Model", y_test, stack_model_predictions)
```

Figure 33: Predictions for the stacking model.

XGBoost

```
xgboost = xgb.XGBClassifier(random_state = 42,
                            use_label_encoder = False,
                            n_jobs = -1)

xgboost.fit(x_train, y_train)

xgboost_predictions = xgboost.predict(x_test)

modelEval('XGBoost', y_test, xgboost_predictions)
```

Figure 34: XGBoost model.

```
%%time
# Define the grid of hyperparameters to search
xgb_params = {
    'n_estimators': [100, 300, 500, 700],
    'max_depth': [2, 3, 4, 5, 6],
    'learning_rate': [0.01, 0.05, 0.1, 0.3],
    'min_child_weight': [1, 2, 3, 4, 5],
    'booster': ['gbtree', 'gblinear', 'dart']
    }

xgb_search = RandomizedSearchCV(estimator = xgb.XGBClassifier(random_state = 42, use_label_encoder = False),
                                param_distributions = xgb_params,
                                cv = 5,
                                n_iter = 50,
                                n_jobs = -1,
                                verbose = 3,
                                random_state = 42)

xgb_search.fit(x_train, y_train)
```

Figure 35: Grid search for the XGBoost.

```
xgb_search.best_params_
```

```
{'n_estimators': 700,
 'min_child_weight': 2,
 'max_depth': 2,
 'learning_rate': 0.05,
 'booster': 'gbtree'}
```

```
xgboost_best = xgb.XGBClassifier(n_estimators = 700,
                                 min_child_weight = 2,
                                 max_depth = 2,
                                 learning_rate = 0.05,
                                 booster = 'gbtree',
                                 random_state = 42,
                                 use_label_encoder = False,
                                 n_jobs = 6)

xgboost_best.fit(x_train, y_train)

xgboost_best_predictions = xgboost_best.predict(x_test)

modelEval("xgboost_best", y_test, xgboost_best_predictions)
```

Figure 36: Optimized parameters for the XGBoost.

```
##### DATA PREPROCESSING FOR THE MODELS #####
fakeBR_df['text'] = fakeBR_df['text'].apply(cleanText)

##### CREATE THE X AND Y SETS #####
x = fakeBR_df['text']
y = fakeBR_df['label']

##### TF #####
# Create the TF Vectorizer object
tf_vectorizer = CountVectorizer(max_features=200)

# Make the sparse matrix
cv = tf_vectorizer.fit_transform(x)

##### TRAIN TEST SPLIT #####
x_train, x_test, y_train, y_test = train_test_split(cv, y, test_size=0.2, random_state=42)

print(f"Shape of the train data: {x_train.shape}")
print(f"Shape of the test data: {x_test.shape}")
print(f"Shape of the train labels: {y_train.shape}")
print(f"Shape of the test labels: {y_test.shape}")
```

Figure 37: Pre-processing steps and creation of train and test sets for the truncated texts.

```python
import pandas as pd
import numpy as np
from useful_funcs import cleanText, modelEval, saveModels

# Machine learning
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
import xgboost as xgb

# Graphics
import matplotlib.pyplot as plt
```

Figure 38: Pre-processing steps and creation of train and test sets for the truncated texts.

```python
##### DATA PREPROCESSING FOR THE MODELS #####
fakeBR_df['text'] = fakeBR_df['text'].apply(cleanText)

##### CREATE THE X AND Y SETS #####
x = fakeBR_df['text']
y = fakeBR_df['label']

##### TF-IDF #####
# Create the TF-IDF Vectorizer object
tfidf_vectorizer = TfidfVectorizer()

# Make the sparse matrix
tfidf = tfidf_vectorizer.fit_transform(x)

##### TRAIN TEST SPLIT #####
x_train, x_test, y_train, y_test = train_test_split(tfidf, y, test_size=0.2, random_state=42)

print(f"Shape of the train data: {x_train.shape}")
print(f"Shape of the test data: {x_test.shape}")
print(f"Shape of the train labels: {y_train.shape}")
print(f"Shape of the test labels: {y_test.shape}")
```

Figure 39: Pre-processing steps and creation of train and test sets for the full texts and the TF-IDF technique.

```python
import pandas as pd
import numpy as np
from useful_funcs import cleanText, modelEval, saveModels, removeStopwords
import os

from sklearn.model_selection import train_test_split

import tensorflow as tf
from keras.models import Sequential, load_model
from keras.layers import Embedding, Dense, Dropout, GlobalMaxPooling1D, Conv1D, GRU, LSTM
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras import callbacks
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Graphics
import matplotlib.pyplot as plt

# Needed for Linux
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
```

Figure 40: Necessary imports for the NN experiments.

```python
MODELS_FOLDER_LINUX = "../project/models/"
MODELS_FOLDER_WINDOWS = "..\\project\\models\\"
```

```python
path_linux = '../project/fakeBR_df.csv'
path_windows = "..\\project\\fakeBR_df.csv"

fakeBR_df = pd.read_csv(path_windows)
fakeBR_df.head()
```

Figure 41: The folder to save the models and how to load the dataframe.

```
##### DATA PREPROCESSING FOR THE MODELS #####
fakeBR_df['text'] = fakeBR_df['text'].apply(cleanText)
fakeBR_df['text'] = fakeBR_df['text'].apply(removeStopwords)

##### CREATE THE X AND Y SETS #####
x_text = fakeBR_df['text']
y = fakeBR_df['label']

##### TRAIN TEST SPLIT #####
dnn_text_train, dnn_text_test, y_train, y_test = train_test_split(x_text, y, test_size=0.2, random_state=42)

maxlen = 300
embedding_dim = 100

# Tokenize words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(dnn_text_train)
X_train = tokenizer.texts_to_sequences(dnn_text_train)
X_test = tokenizer.texts_to_sequences(dnn_text_test)

# Adding 1 because of reserved 0 index
vocab_size = len(tokenizer.word_index) + 1

# Pad sequences with zeros
X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)
```

Figure 42: Pre-processing steps for the NN experiments.

Convolutional Neural Network

```python
cnn = Sequential()
cnn.add(Embedding(vocab_size, embedding_dim,
                               input_length=maxlen,
                               trainable=True))
cnn.add(Dropout(0.3))
cnn.add(Conv1D(128, 4, activation='relu'))
cnn.add(GlobalMaxPooling1D())
cnn.add(Dropout(0.3))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dense(1, activation='sigmoid'))

cnn.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_8 (Embedding)      (None, 300, 100)          6810100
_________________________________________________________________
dropout_10 (Dropout)         (None, 300, 100)          0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 297, 128)          51328
_________________________________________________________________
global_max_pooling1d_2 (Glob (None, 128)               0
_________________________________________________________________
dropout_11 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_13 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 129
=================================================================
Total params: 6,878,069
Trainable params: 6,878,069
Non-trainable params: 0
_________________________________________________________________
```

Figure 43: CNN architecture.

```python
cnn.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

es_cnn = callbacks.EarlyStopping(monitor='val_accuracy', patience=4, min_delta=0.0001, verbose=1)
mc_cnn = callbacks.ModelCheckpoint(f'{MODELS_FOLDER_WINDOWS}cnn_best.h5', monitor='val_accuracy', save_best_only=True, verbose=1)
cb_list_cnn = [es_cnn, mc_cnn]
```

```python
history_cnn = cnn.fit(X_train, y_train, epochs=30, batch_size = 8, validation_split=0.3, callbacks=cb_list_cnn)
```

Figure 44: CNN callbacks and fit.

```python
cnn_predictions = (cnn.predict(X_test) > 0.5).astype("int32")
cnn_predictions
```

```
array([[1],
       [1],
       [0],
       ...,
       [0],
       [1],
       [1]])
```

```python
modelEval("cnn", y_test, cnn_predictions)
```

Figure 45: CNN predictions.

```python
best_cnn = load_model(f"{MODELS_FOLDER_WINDOWS}cnn_best.h5")
best_cnn_predictions = (best_cnn.predict(X_test) > 0.5).astype("int32")
modelEval("best CNN", y_test, best_cnn_predictions)
```

Figure 46: Best CNN model being loaded.

```
acc = history_cnn.history['accuracy']
val_acc = history_cnn.history['val_accuracy']
loss = history_cnn.history['loss']
val_loss = history_cnn.history['val_loss']
epochs = range(len(acc))

fig, ax = plt.subplots(1, 2, figsize=(16,8))
ax[0].plot(epochs, acc, 'r', label='Training accuracy')
ax[0].plot(epochs, val_acc, 'b', label='Validation accuracy')
ax[0].set_title('Training and validation accuracy')
ax[0].legend(loc=0)

ax[1].plot(epochs, loss, 'r', label='Training loss')
ax[1].plot(epochs, val_loss, 'b', label='Validation loss')
ax[1].set_title('Training and validation loss')
ax[1].legend(loc=0)

# plt.savefig("cnn_acc.png", dpi=300, bbox_inches='tight')

plt.show()
```

Figure 47: Code snippet to plot the loss and accuracy of the CNN.

Gated Recurrent Unit

```
gru = Sequential()
gru.add(Embedding(vocab_size,
                  embedding_dim,
                  input_length=maxlen,
                  trainable=False))
gru.add(GRU(128))
gru.add(Dropout(0.25))
gru.add(Dense(1, activation='sigmoid'))

gru.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 300, 100)          6810100
_________________________________________________________________
gru_1 (GRU)                  (None, 128)               88320
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 129
=================================================================
Total params: 6,898,549
Trainable params: 88,449
Non-trainable params: 6,810,100
_________________________________________________________________
```

Figure 48: GRU architecture.

```
gru.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy'])

es_gru = callbacks.EarlyStopping(monitor='val_accuracy', patience=4, min_delta=0.0001, verbose=1)
mc_gru = callbacks.ModelCheckpoint(f'{MODELS_FOLDER_WINDOWS}gru_best.h5', monitor='val_accuracy', save_best_only=True, verbose=1)
cb_list_gru = [es_gru, mc_gru]
```

```
history_gru = gru.fit(X_train, y_train,
                      epochs = 20,
                      batch_size = 8,
                      validation_split=0.3,
                      callbacks=cb_list_gru)
```

Figure 49: GRU callbacks and fit.

```
gru_predictions = (gru.predict(X_test) > 0.5).astype("int32")
gru_predictions
```

```
array([[1],
       [1],
       [0],
       ...,
       [0],
       [1],
       [1]])
```

```
modelEval("GRU", y_test, gru_predictions)
```

Figure 50: GRU predictions.

```
best_gru = load_model(f"{MODELS_FOLDER_WINDOWS}gru_best.h5")
best_gru_predictions = (best_gru.predict(X_test) > 0.5).astype("int32")
modelEval("best GRU", y_test, best_gru_predictions)
```

Figure 51: Best GRU model being loaded.

Long-short Term Memory

```
lstm = Sequential()
lstm.add(Embedding(vocab_size, embedding_dim,
                            input_length=maxlen,
                            trainable=True))
lstm.add(LSTM(256, return_sequences=True))
lstm.add(Dropout(0.3))
lstm.add(LSTM(64))
lstm.add(Dense(32, activation='relu'))
lstm.add(Dense(1, activation='sigmoid'))

lstm.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (Embedding)      (None, 300, 100)          6810100
_________________________________________________________________
lstm_5 (LSTM)                (None, 300, 256)          365568
_________________________________________________________________
dropout_7 (Dropout)          (None, 300, 256)          0
_________________________________________________________________
lstm_6 (LSTM)                (None, 64)                82176
_________________________________________________________________
dense_9 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 33
=================================================================
Total params: 7,259,957
Trainable params: 7,259,957
Non-trainable params: 0
_________________________________________________________________
```

Figure 52: LSTM architecture.

```python
lstm.compile(optimizer='adam',
             loss='binary_crossentropy',
             metrics=['accuracy'])

es_lstm = callbacks.EarlyStopping(monitor='val_accuracy', patience=4, min_delta=0.0001, verbose=1)
mc_lstm = callbacks.ModelCheckpoint(f'{MODELS_FOLDER_WINDOWS}lstm_best.h5', monitor='val_accuracy', save_best_only=True, verbose=1)
cb_list_lstm = [es_lstm, mc_lstm]
```

```python
history_lstm = lstm.fit(X_train, y_train,
                        epochs=20,
                        batch_size = 8,
                        validation_split=0.3,
                        callbacks=cb_list_lstm)
```

Figure 53: LSTM callbacks and fit.

```python
lstm_predictions = (lstm.predict(X_test) > 0.5).astype("int32")
lstm_predictions
```

```
array([[1],
       [1],
       [0],
       ...,
       [0],
       [1],
       [1]])
```

```python
modelEval("LSTM", y_test, lstm_predictions)
```

Figure 54: LSTM predictions.

```python
best_lstm = load_model(f"{MODELS_FOLDER_WINDOWS}lstm_best.h5")
best_lstm_predictions = (best_lstm.predict(X_test) > 0.5).astype("int32")
modelEval("LSTM", y_test, best_lstm_predictions)
```

Figure 55: Best LSTM model being loaded.

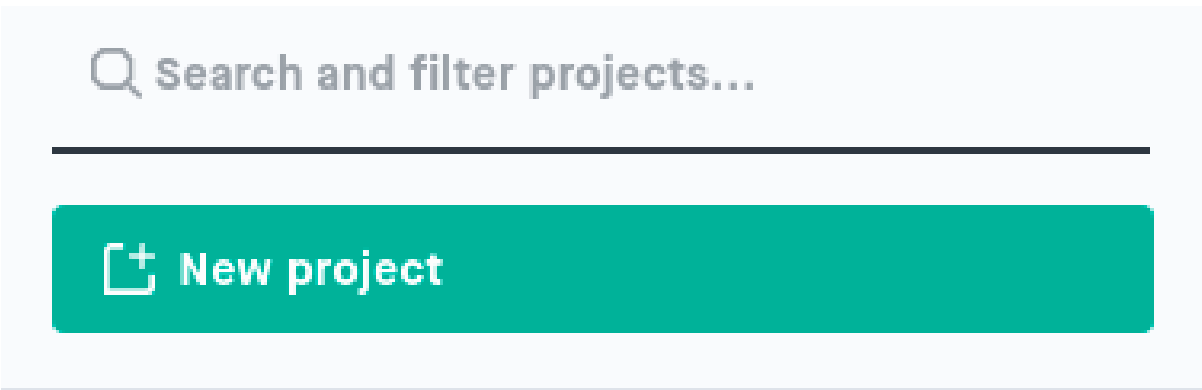

Figure 56: Create a project. It is locate din the top left of the page.

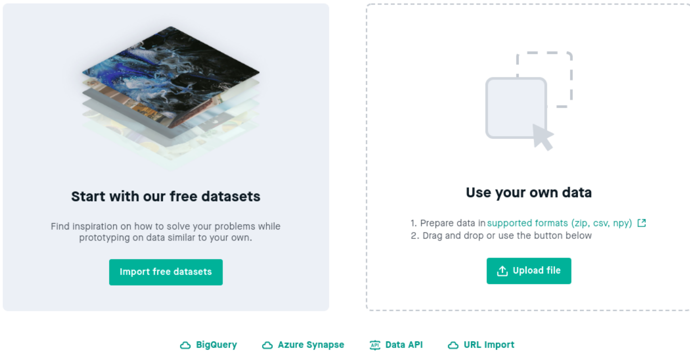

Figure 57: Upload the dataset as a csv file.

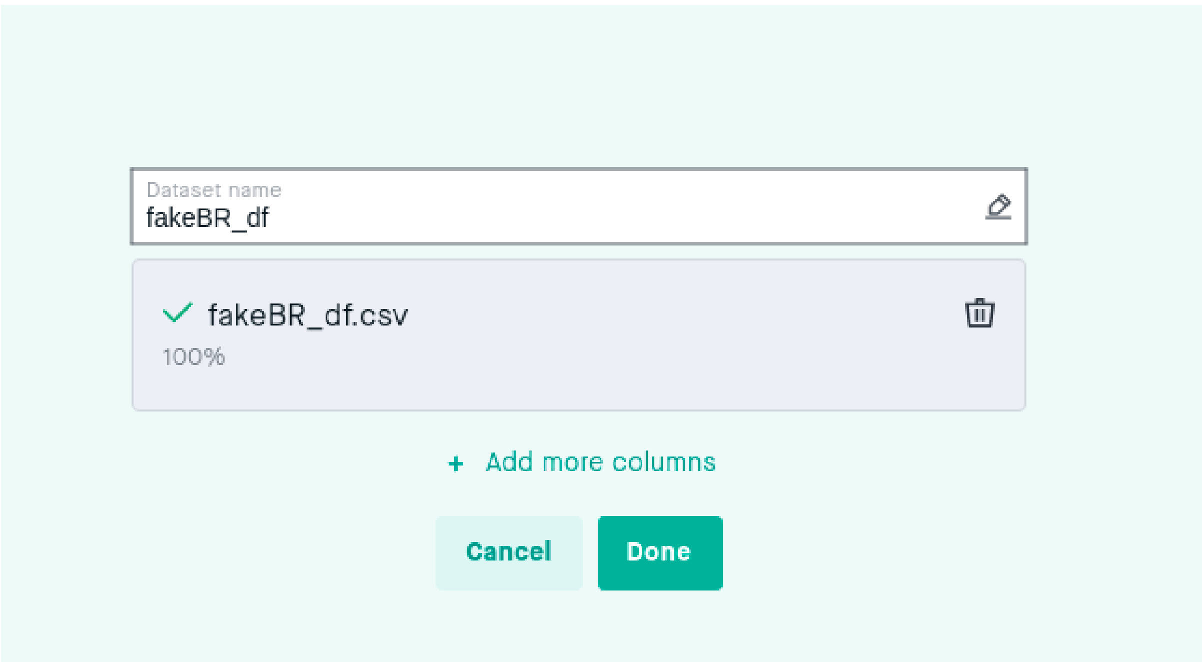

Figure 58: Click in "done".

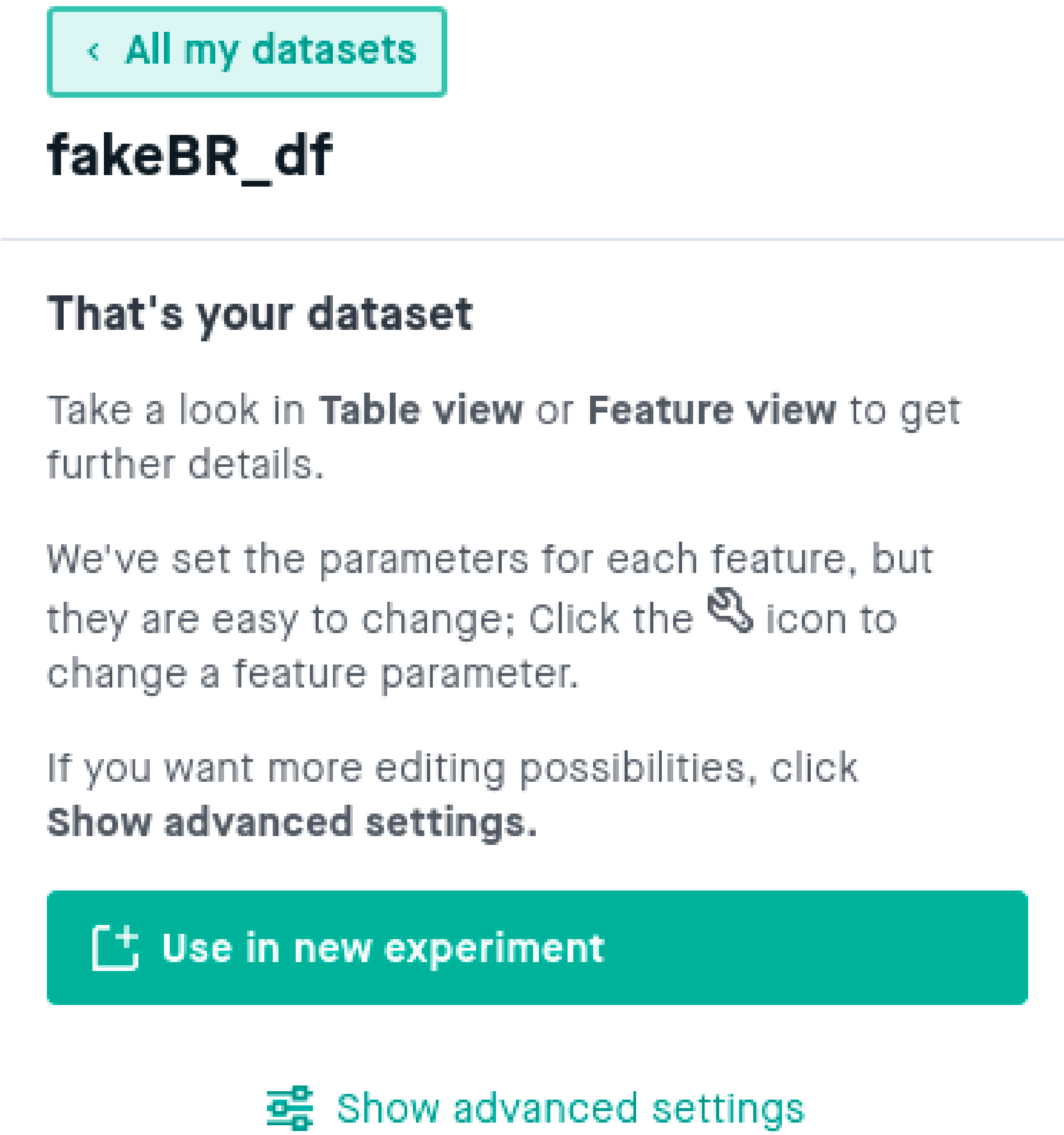

Figure 59: Click in "Use in new experiment".

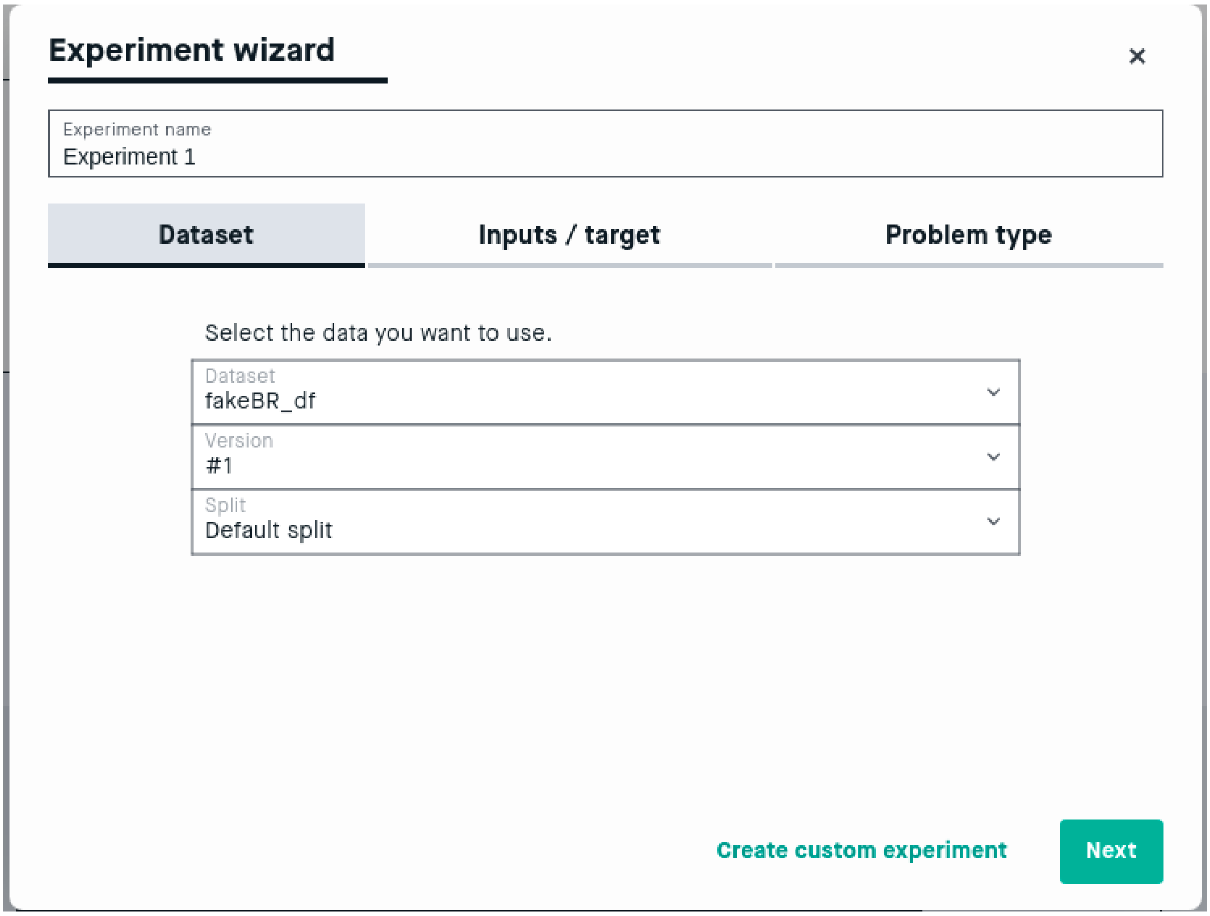

Figure 60: Choose these settings for the first tab.

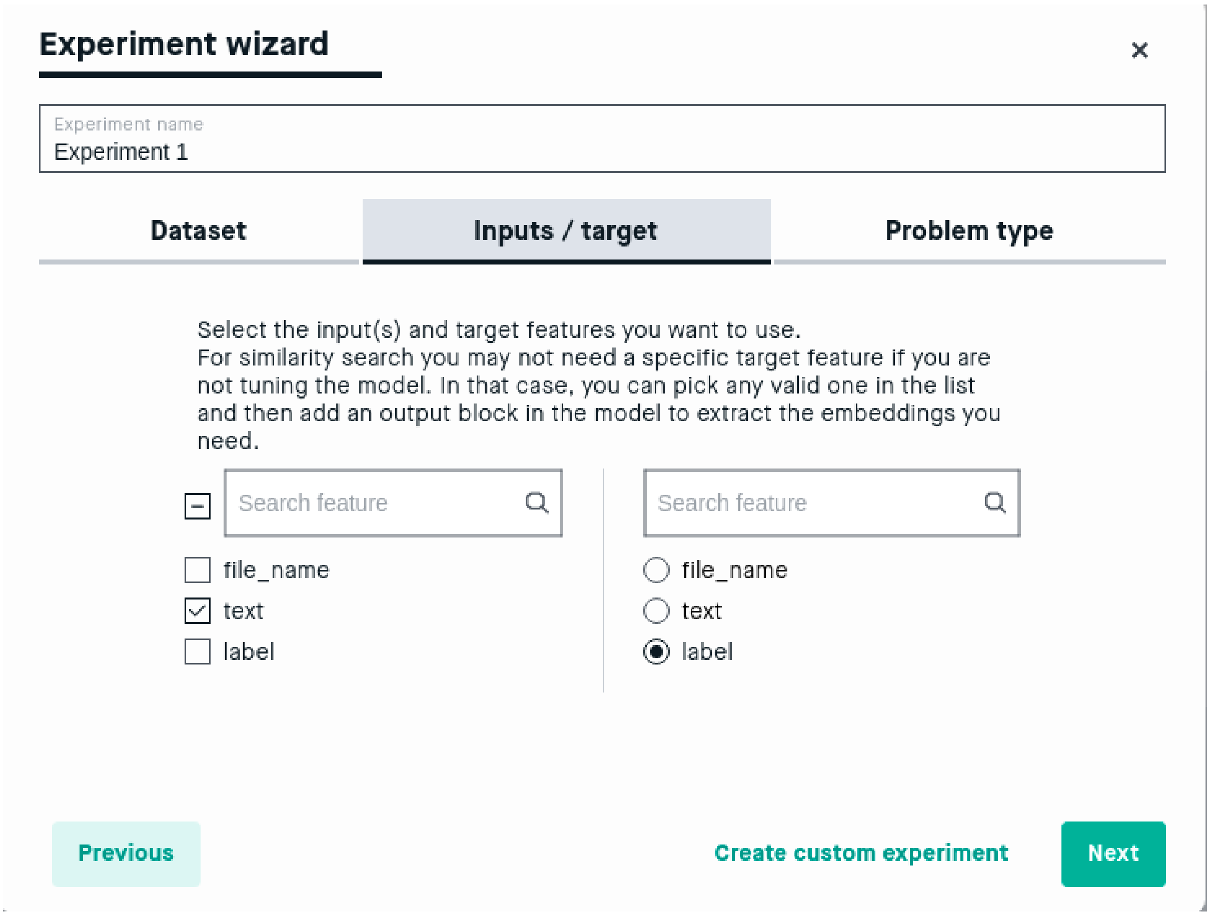

Figure 61: Choose these settings for the second tab.

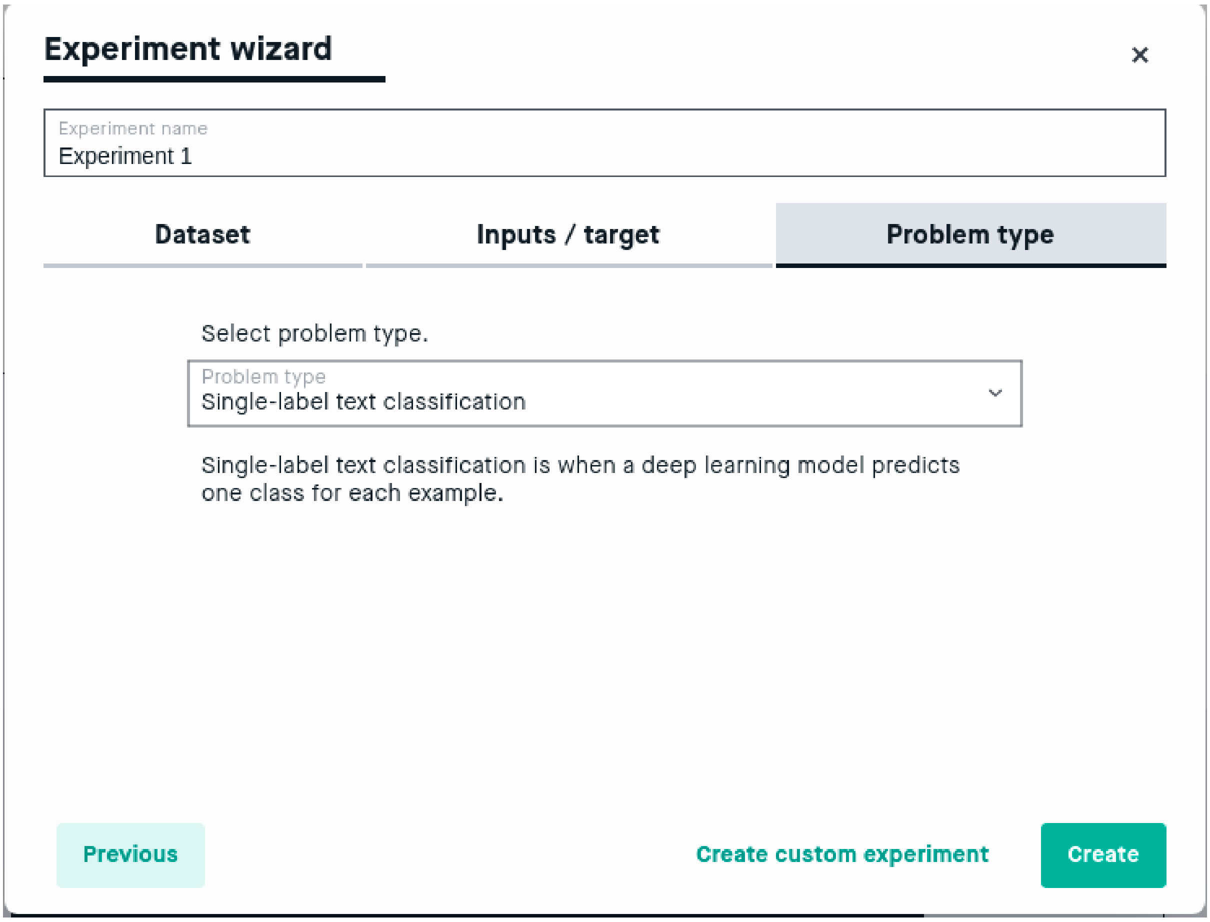

Figure 62: Choose these settings for the third tab and click in "create".

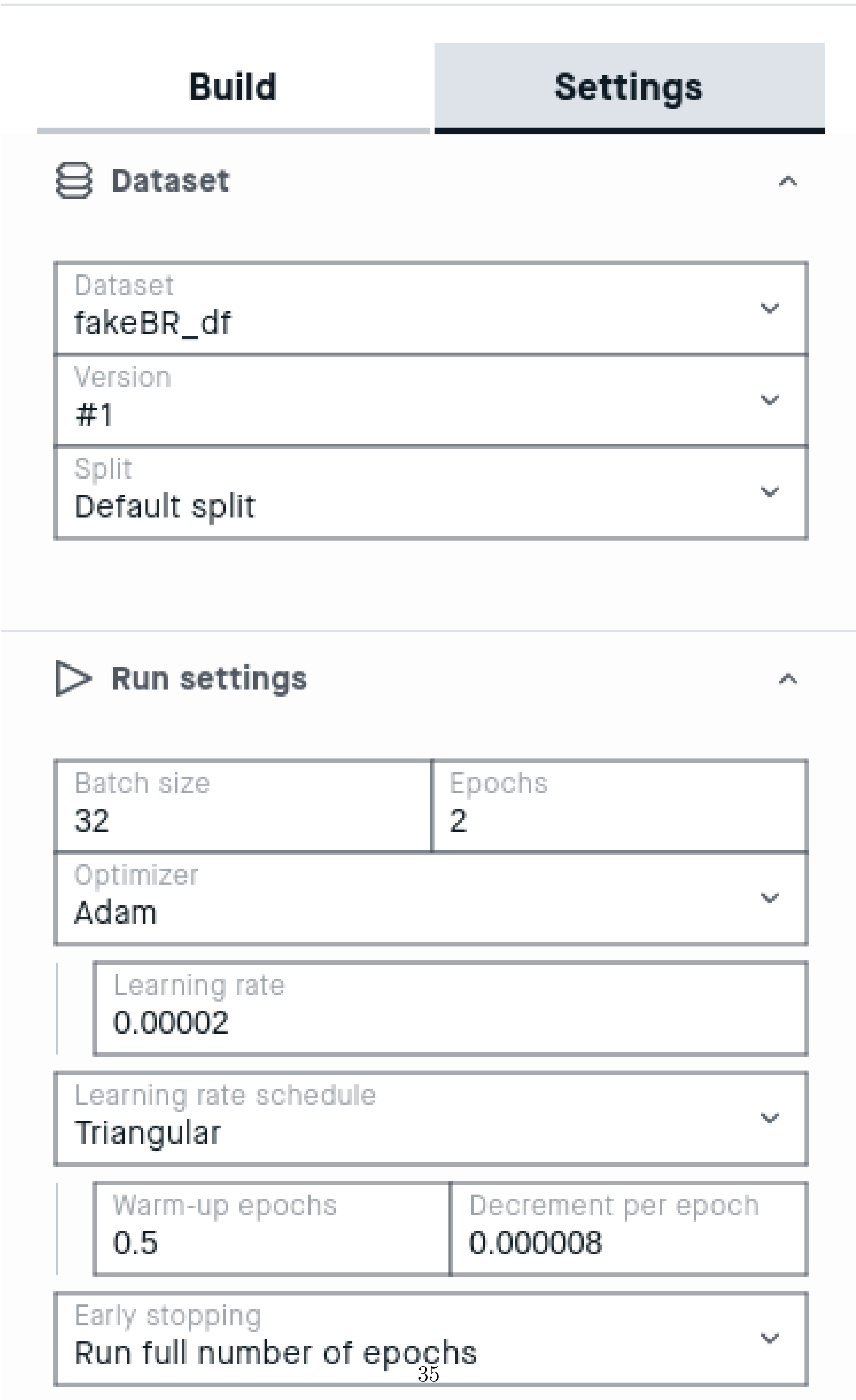

Figure 63: The parameters used are shown in this figure.

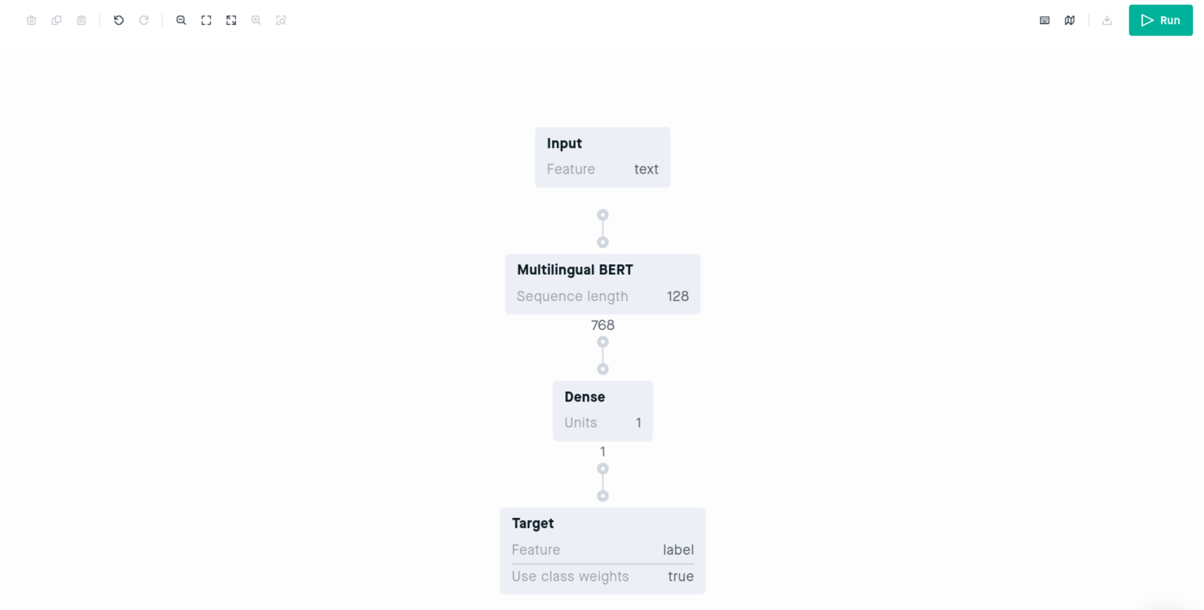

Figure 64: Click in "run" on the top right corner to run the model.

Figure 65: Go to the evaluation tab and check the results after the model ran.