Fake News Project

The goal of this project is to create a fake news prediction system. Fake news is a major problem that can have serious negative effects on how people understand the world around them. You will work with a dataset containing real and fake news in order to train a simple and a more advanced classifier to solve this problem. This project covers the full Data Science pipeline, from data processing, to modelling, to visualization and interpretation.

We ran the notebook with the following specs:

- CPU: Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz
- Cores: 10
- Threads: 20
- Memory: 64GB Ram

Part 1 Data Processing

Task 1

Pandas is used to process The fake news corpus. Since content will be used for our models we drop any rows that don't have any content.

In [ ]:
```
import pandas as pd
df = pd.read_csv("news_sample.csv")
dfcpy = df.copy()
dfcpy = dfcpy.dropna(subset=['content'])
```

In [ ]:
```
import nltk
nltk.download('punkt')
nltk.download('stopwords')
```

We've implemented data processing functions to do the following:

- Clean the text
- Tokenize the text
- Remove stopwords
- Remove word variations with stemming

We use nltk because it has built-in support for many of these operations.

In [ ]:
```
import re
import nltk
from nltk.tokenize.regexp import RegexpTokenizer
from nltk.stem import PorterStemmer
from collections import Counter
from cleantext import clean
```

```
def clean_text(text):
  clean_text = re.sub(r'([A-Z][A-z]+.?) ([0-9]{1,2}?), ([0-9]{4})', '<DAT
  clean_text = clean(clean_text,
    lower=True,
    no_urls=True, replace_with_url="<URL>",
    no_emails=True, replace_with_email="<EMAIL>",
    no_numbers=True, replace_with_number= r"<NUM>",
    no_currency_symbols=True, replace_with_currency_symbol="<CUR>",
    no_punct=True, replace_with_punct="",
    no_line_breaks=True
  )
  return clean_text

def rmv_stopwords(tokens):
  stop_words = set(nltk.corpus.stopwords.words('english'))
  tokens = [word for word in tokens if word not in stop_words]
  return tokens

def stem_tokens(tokens):
  stemmer=PorterStemmer()
  Output=[stemmer.stem(word) for word in tokens]
  return Output

# build a vocabulary from a dataframe with list of tokens
def build_vocabulary(df_tokens):
    # Flatten the list of tokens
  tokens = []
  for lst in df_tokens:
    tokens += lst
  token_counter = Counter(tokens)
  return token_counter
```

In [ ]:

```
dfcpy = df.copy()

dfcpy.content = dfcpy.content.apply(clean_text)

tokenizer = RegexpTokenizer(r'<[\w]+>|[\w]+')
dfcpy["tokenized"] = dfcpy.content.apply(tokenizer.tokenize)

vocab = build_vocabulary(dfcpy.tokenized)
vocab_size = len(vocab)
print("After cleaning:")
print(f"vocabulary size: {vocab_size}\n")

dfcpy.tokenized = dfcpy.tokenized.apply(rmv_stopwords)
vocab = build_vocabulary(dfcpy.tokenized)
# reduction rate of the vocabulary size
reduction = ((vocab_size - len(vocab))/vocab_size)*100
vocab_size = len(vocab)
print("After removing stopwords:")
print(f"vocabulary size: {vocab_size}")
print(f"reduction rate of the vocabulary size: {reduction:.2f}%\n")

dfcpy.tokenized = dfcpy.tokenized.apply(stem_tokens)
vocab = build_vocabulary(dfcpy.tokenized)
reduction = ((vocab_size - len(vocab))/vocab_size)*100
vocab_size = len(vocab)
print("After stemming:")
print(f"vocabulary size: {vocab_size}")
```

```
print(f"reduction rate of the vocabulary size: {reduction:.2f}%\n")
```

```
After cleaning:
vocabulary size: 16577

After removing stopwords:
vocabulary size: 16445
reduction rate of the vocabulary size: 0.80%

After stemming:
vocabulary size: 11031
reduction rate of the vocabulary size: 32.92%
```

Task 2

We apply our data processing pipeline from task 1 on the *995k FakeNewsCorpus*. Due to the of the size of the dataset and to avoid crashes, each part in the data proccessing pipeline is executed in it's own cell on the *995k FakeNewsCorpus*.

We've explored the dataset and made some observations which are used to determine importance of certain metadata in the fake news corpus, such observations are:

- The amount of numerics in the dataset
- The 100 most frequent words
- The 20 most frequent domains and how their articles are classified in terms of type
- The distrubtion of types in the dataset
- The amount of rows missing content, title or type (amount of rows that will be dropped from the dataset).

Pandas is slow when used on bigger amounts of data, this is because it dosen't allow for multithreading. Modin and ray are libaries that optimize pandas by allowing pandas to run on all cores, thereby giving a speed up for the data processing. By using modin with ray as an engine you can use pandas as usual, but have it use all threads in the CPU. We used a Intel Xeon cpu with 20 threads and therefore saw huge performance gain by using modin.

Modin and ray can be installed by running the following command: pip install "modin[ray]"

In [ ]:
```
import modin.config as modin_cfg
modin_cfg.Engine.put("ray")
import modin.pandas as pd
```

In [ ]:
```
# only read the columns we need
df = pd.read_csv("995,000_rows.csv", usecols=['content', 'type', 'title',
dfcpy = df.copy()
dfcpy = dfcpy.dropna(subset=['content'])
dfcpy = dfcpy.dropna(subset=['type'])
dfcpy = dfcpy.dropna(subset=['title'])
```

In [ ]:
```
from time import time
start = time()
dfcpy.title = dfcpy.title.apply(clean_text)
dfcpy.content = dfcpy.content.apply(clean_text)
print(f"time to clean the data: {time() - start} sec")

t = time()
tokenizer = RegexpTokenizer(r'<[\w]+>|[\w]+')
dfcpy.title = dfcpy.title.apply(tokenizer.tokenize)
dfcpy.content = dfcpy.content.apply(tokenizer.tokenize)
print(f"time to tokenize the data: {(time() - t)/60} min" )

t = time()
dfcpy.title = dfcpy.title.apply(rmv_stopwords)
dfcpy.content = dfcpy.content.apply(rmv_stopwords)
print(f"time to remove stopwords: {(time() - t)/60} min")

t = time()
dfcpy.title = dfcpy.title.apply(stem_tokens)
dfcpy.content = dfcpy.content.apply(stem_tokens)
print(f"time to stem the data: {time() - t} sec")

print(f"total time: {(time() - start)/60} min")
```

```
time to clean the data: 16.306692361831665 sec
time to tokenize the data: 6.8294127702713014 min
time to remove stopwords: 1.1825481534004212 min
time to stem the data: 73.61578845977783 sec
total time: 9.510708979765575 min
```

Data exploration

In [ ]:
```
start = time()
vocab_content = build_vocabulary(dfcpy.content)
print(f"time to build vocabulary for content: {(time() - start)/60} min")

start = time()
vocab_title = build_vocabulary(dfcpy.title)
print(f"time to build vocabulary for title: {(time() - start)/60} min")
```

```
time to build vocabulary for content: 14.65651472012202 min
time to build vocabulary for title: 0.06745206912358602 min
```

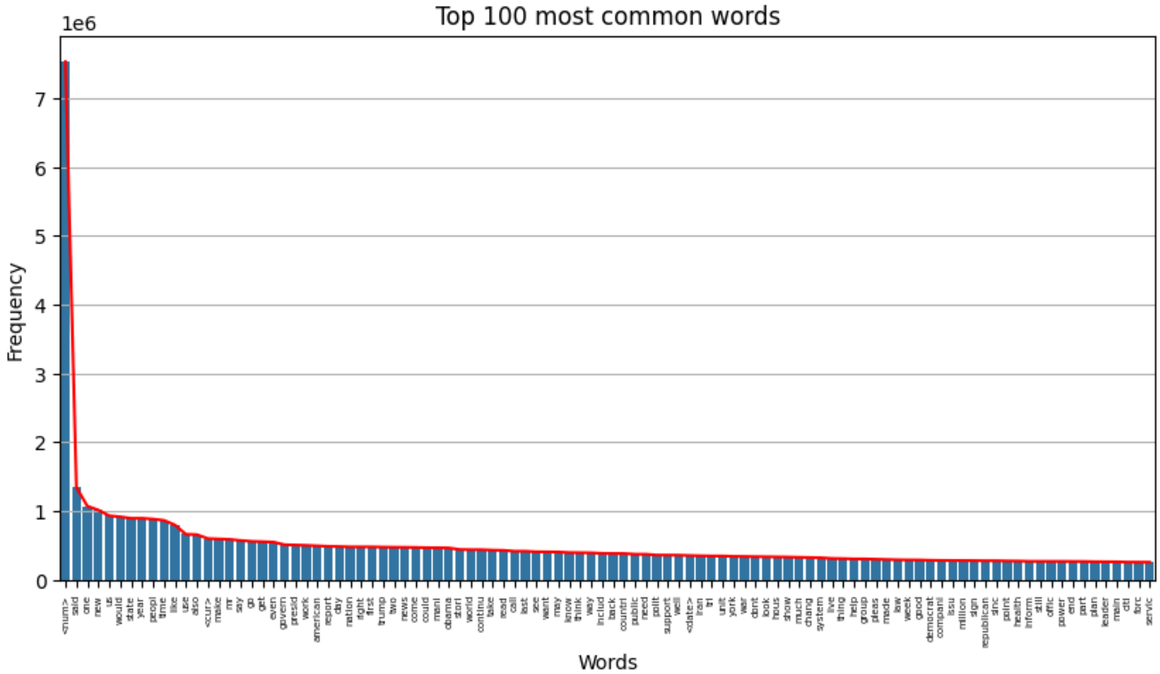
In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
# plot the frequency of the top n words
def plot_freq(counter, top_n):
  common_words = counter.most_common(top_n)

  all_freq = {}
  for word, freq in common_words:
    all_freq[word] = freq

  plt.figure(figsize = (top_n*0.1, 5))
  plt.xticks(rotation = 90,fontsize = 5)
  sns.lineplot(x = list(all_freq.keys()), y = list(all_freq.values()), co
  sns.barplot(x = list(all_freq.keys()), y = list(all_freq.values()))
  plt.title(f'Top {top_n} most common words')
  plt.xlabel('Words')
```

```
    plt.ylabel('Frequency')
    plt.grid(axis = 'y')
    plt.show()
    return

def plot_domain_with_type(df):
    top_domains = df.domain.value_counts().head(20).index
    df = df[df.domain.isin(top_domains)]
    df = df.groupby(['domain', 'type']).size().unstack().fillna(0)

    df.plot(kind='bar', stacked=True, figsize=(10,5), title='Domain distrib
    plt.show()
    return
```

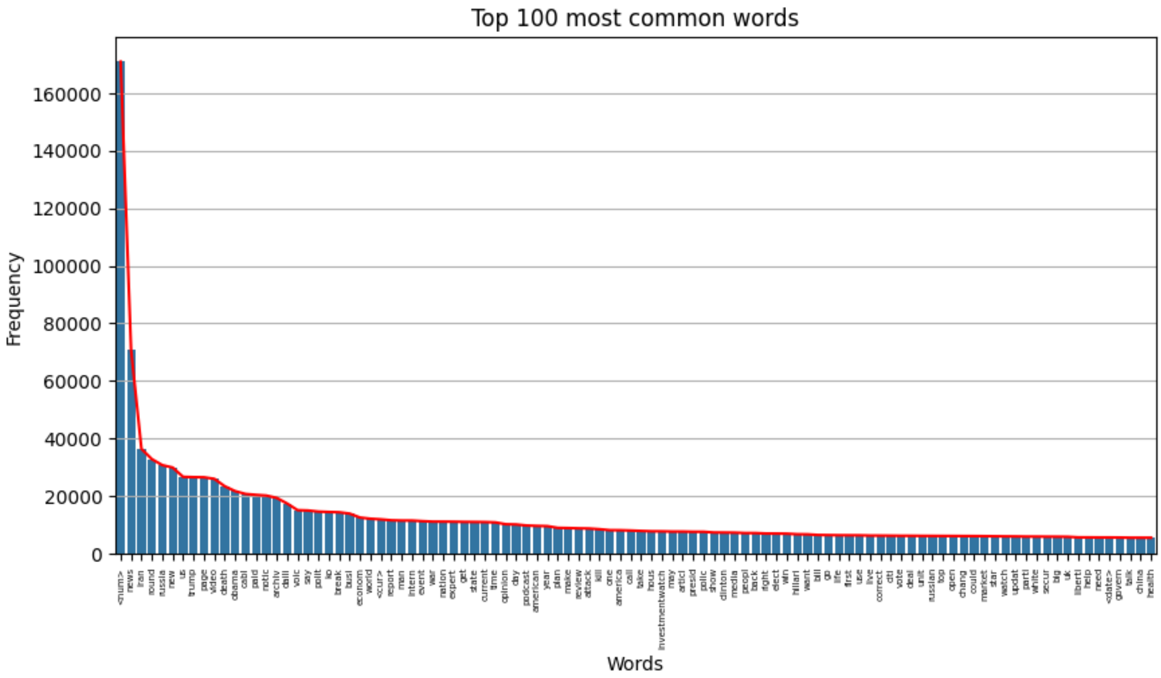
In [ ]:

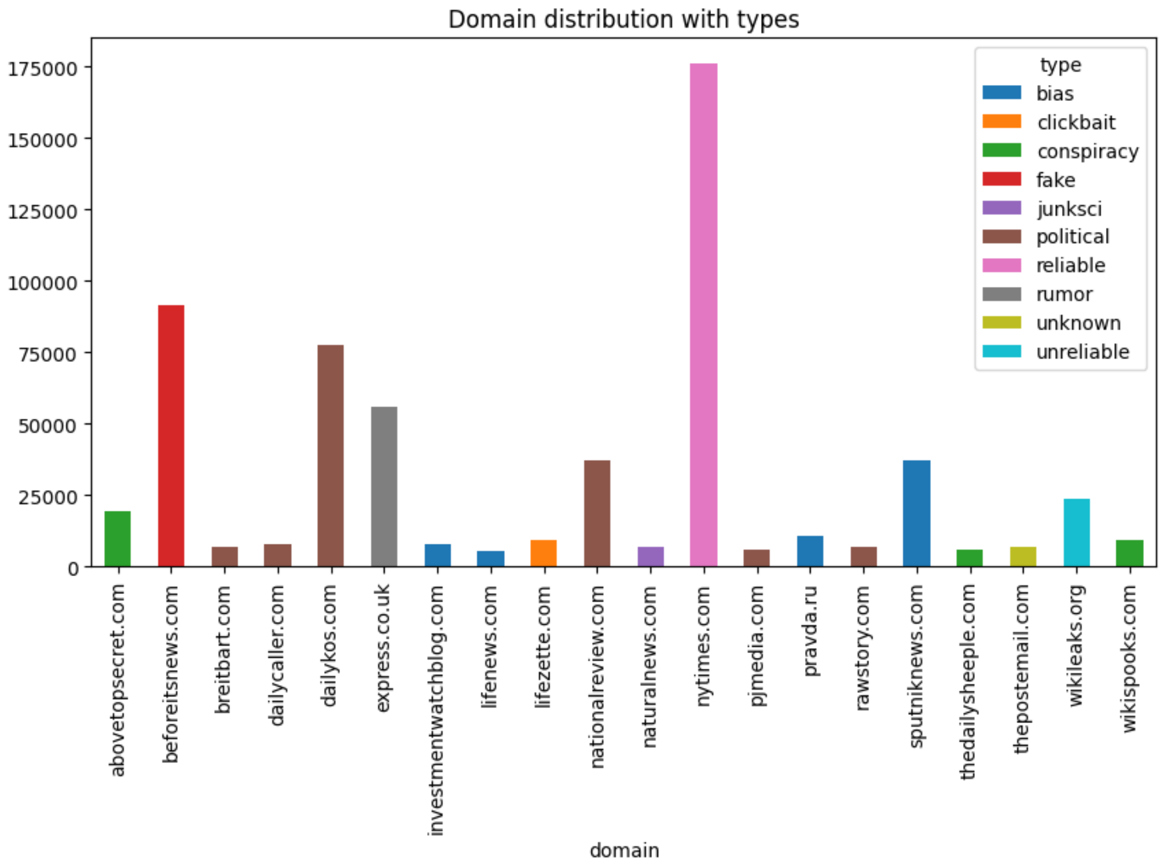
```
# top 100 most frequent words
print("numerics in content: ", vocab_content["<num>"])
plot_freq(vocab_content, 100)
print("numerics in titles: ", vocab_title["<num>"])
plot_freq(vocab_title, 100)

# top 20 domains with their types
plot_domain_with_type(dfcpy)

# pie chart for the distribution of the types
dfcpy.type.value_counts().plot.pie(autopct='%1.1f%%', figsize=(10,5), tit
plt.show()

# ammount of dropped rows
print(f"Number of dropped rows: {df.shape[0] - dfcpy.shape[0]}")
```

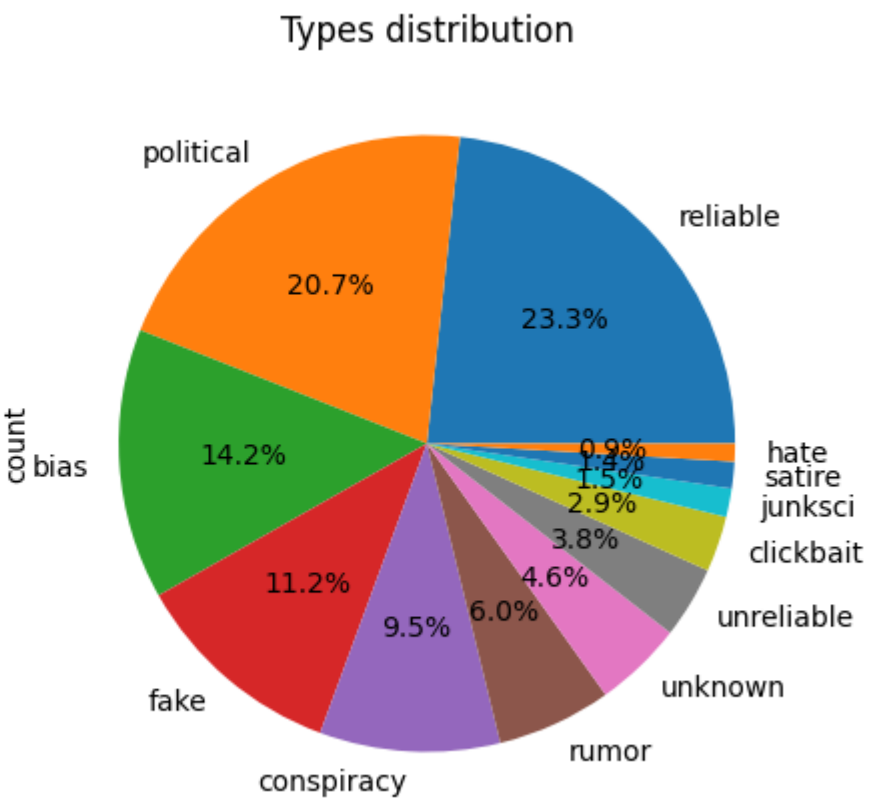
numerics in content: 7530933



numerics in titles: 170894


Top 100 most common words
Frequency
160000
140000
120000
100000
80000
60000
40000
20000
0
Words


Domain distribution with types
175000
150000
125000
100000
75000
50000
25000
0
type
bias
clickbait
conspiracy
fake
junksci
political
reliable
rumor
unknown
unreliable
abovetopsecret.com
beforeitsnews.com
breitbart.com
dailycaller.com
dailykos.com
express.co.uk
investmentwatchblog.com
lifenews.com
lifezette.com
nationalreview.com
naturalnews.com
nytimes.com
pjmedia.com
pravda.ru
rawstory.com
sputniknews.com
thedailysheeple.com
thepostemail.com
wikileaks.org
wikispooks.com
domain




```
Number of dropped rows: 56368
```

In [ ]:
```
dfcpy.content = dfcpy.content.apply(lambda x: ' '.join(x))
dfcpy.title = dfcpy.title.apply(lambda x: ' '.join(x))
dfcpy.to_csv('cleaned_news.csv', index=False)
print("done cleaning the data")

import ray
ray.shutdown()
```

```
done cleaning the data
```

When exporting the cleaned dataset we have to make sure the tokens are stored correctly in the csv. A csv can correctly store a python list, therefore we store the tokens as a string using space as a seperator for each token.

Task 4

Using the types we label articles as either fake or reliable. Some article types are omitted since it's ambigious wheter they are fake news or not.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.read_csv('cleaned_news.csv', usecols=['content', 'type', 'title']
dfcpy = df.copy()
# label is 1 if the article is fake, 0 if the article is reliable
dfcpy['label'] = dfcpy['type'].map({'fake': 1,
                                    'conspiracy': 1,
                                    'junksci': 1,
                                    'bias': 1,
```

```
                                        'clickbait': 0,
                                        'political': 0,
                                        'reliable': 0})
dfcpy = dfcpy.dropna(subset=['label'])
dfcpy['label'] = dfcpy['label'].astype(int)

dfcpy = dfcpy.dropna(subset=['content'])
dfcpy = dfcpy.dropna(subset=['title'])
```

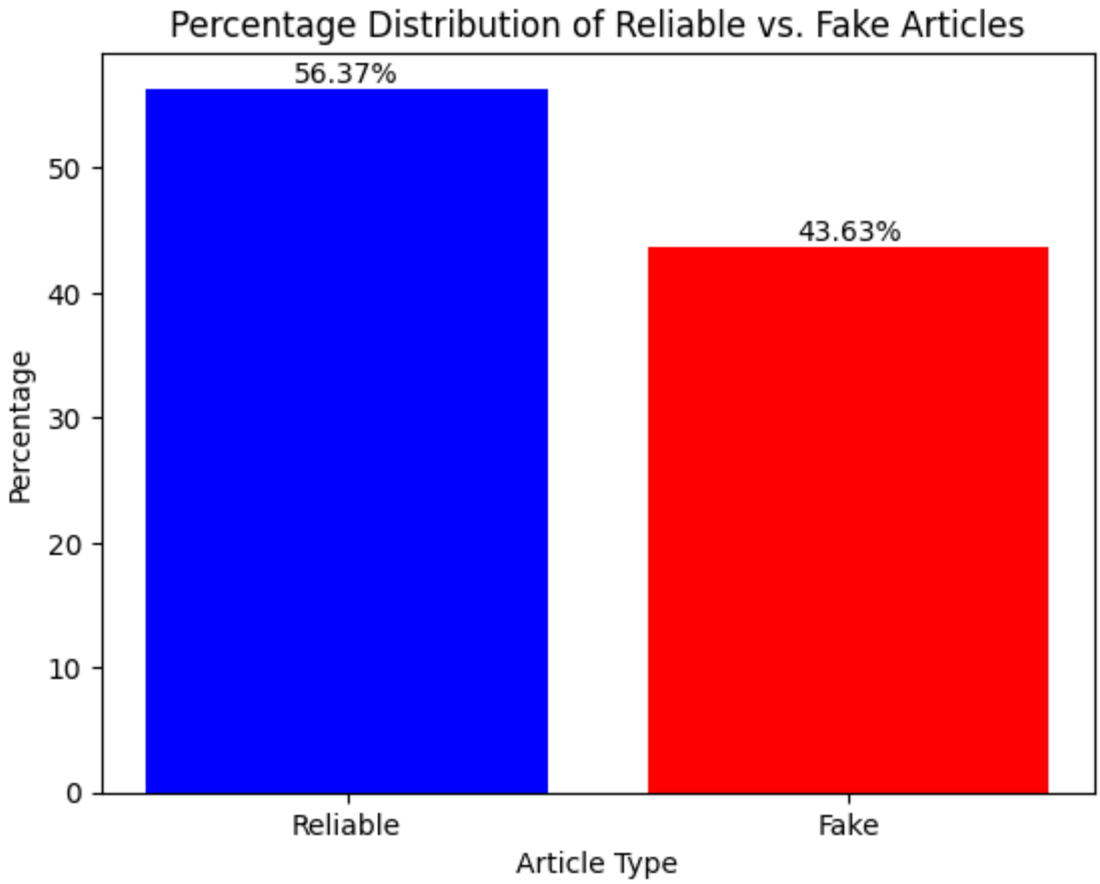
We split the dataset into a random 80/10/10 split where 80% is used for training. 10% is used for validation and another 10% is used for testing.

In [ ]:
```
from sklearn.model_selection import train_test_split
# Splitting the data into training (80%) and the rest (20%)
train_df, rest_df = train_test_split(dfcpy, test_size=0.2, random_state=4
# Splitting the rest into validation (50%) and test (50%)
validation_df, test_df = train_test_split(rest_df, test_size=0.5, random_

content_train, title_train ,y_train = train_df['content'], train_df['titl
content_val, title_val, y_val = validation_df['content'], validation_df['
content_test, title_test, y_test = test_df['content'], test_df['title'],

print("Training Set:")
print(train_df.content.head())
print(train_df.title.head())
```

```
Training Set:
779426    <num> year old iranian man share life iran wor...
51493     love love harlem said could walk favorit resta...
325203    plu one articl googl plu thank ali alfoneh ass...
272377    larri silverstein caught admit camera plan bui...
38281     artifici intellig complex creator cant trust m...
Name: content, dtype: object
779426                             tale iranian blogger
51493                   home daniel brook orang new black
325203                                  iran news round
272377                          news wire mintpress news
38281     artifici intellig complex creator cant trust m...
Name: title, dtype: object
```

We plot the Distrubution of Fake and reliable articles to get and idea on wheter our data is balanced or not

In [ ]:
```
# Examine the percentage distribution of 'reliable' vs. 'fake' articles
grouped_type = dfcpy['label'].value_counts()
grouped_type = grouped_type / grouped_type.sum() * 100

# make a bar plot with percentages on bars
plt.bar([0, 1], grouped_type, tick_label=['Reliable', 'Fake'], color=['bl
plt.text(0, grouped_type[0], f'{grouped_type[0]:.2f}%', ha='center', va='
plt.text(1, grouped_type[1], f'{grouped_type[1]:.2f}%', ha='center', va='
plt.xlabel('Article Type')
plt.ylabel('Percentage')
plt.title('Percentage Distribution of Reliable vs. Fake Articles')
plt.show()
```



In [ ]:
```
df_extra = pd.read_csv("scraped_articles.csv", usecols=['content'])
df_extra_cpy = df_extra.copy()
df_extra_cpy = df_extra_cpy.dropna(subset=['content'])
df_extra_cpy.content = df_extra_cpy.content.apply(clean_text)
tokenizer = RegexpTokenizer(r'<[\w]+>|[\w]+')
df_extra_cpy.content = df_extra_cpy.content.apply(tokenizer.tokenize)
df_extra_cpy.content = df_extra_cpy.content.apply(rmv_stopwords)
df_extra_cpy.content = df_extra_cpy.content.apply(stem_tokens)
df_extra_cpy['label'] = 0


df_extra_cpy.content = df_extra_cpy.content.apply(lambda x: ' '.join(x))
x_train_extra = pd.concat([content_train, df_extra_cpy.content], ignore_i
y_train_extra = pd.concat([y_train, df_extra_cpy.label], ignore_index=Tru
```

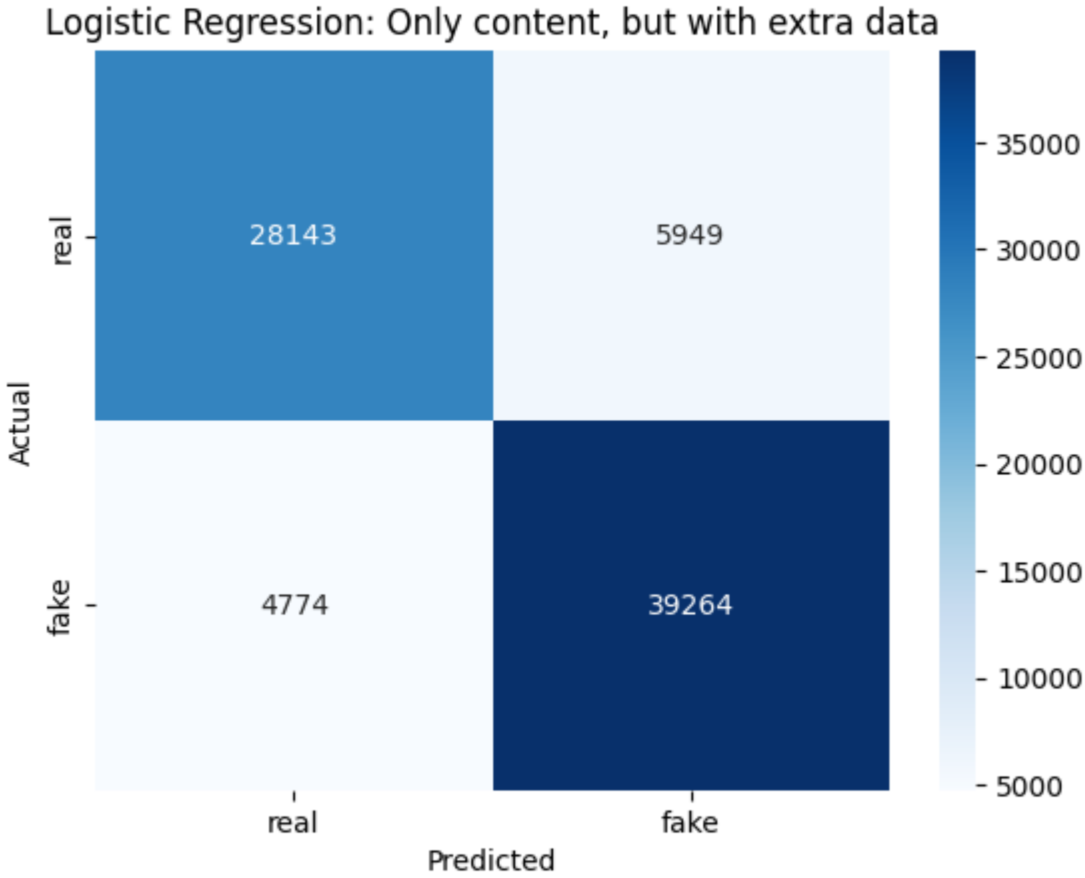
In [ ]:
```
import seaborn as sns
from sklearn import metrics
def make_confusion_matrix(y_val, y_pred,model_name):
    # Confusion matrix
    confusion_matrix = metrics.confusion_matrix(y_val, y_pred, labels=[1,
    sns.heatmap(confusion_matrix,
                annot=True,
                fmt='g',
                cmap='Blues',
                xticklabels=['real', 'fake'],
                yticklabels=['real', 'fake'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'{model_name}')
    plt.show()
```

Part 2: A simple model

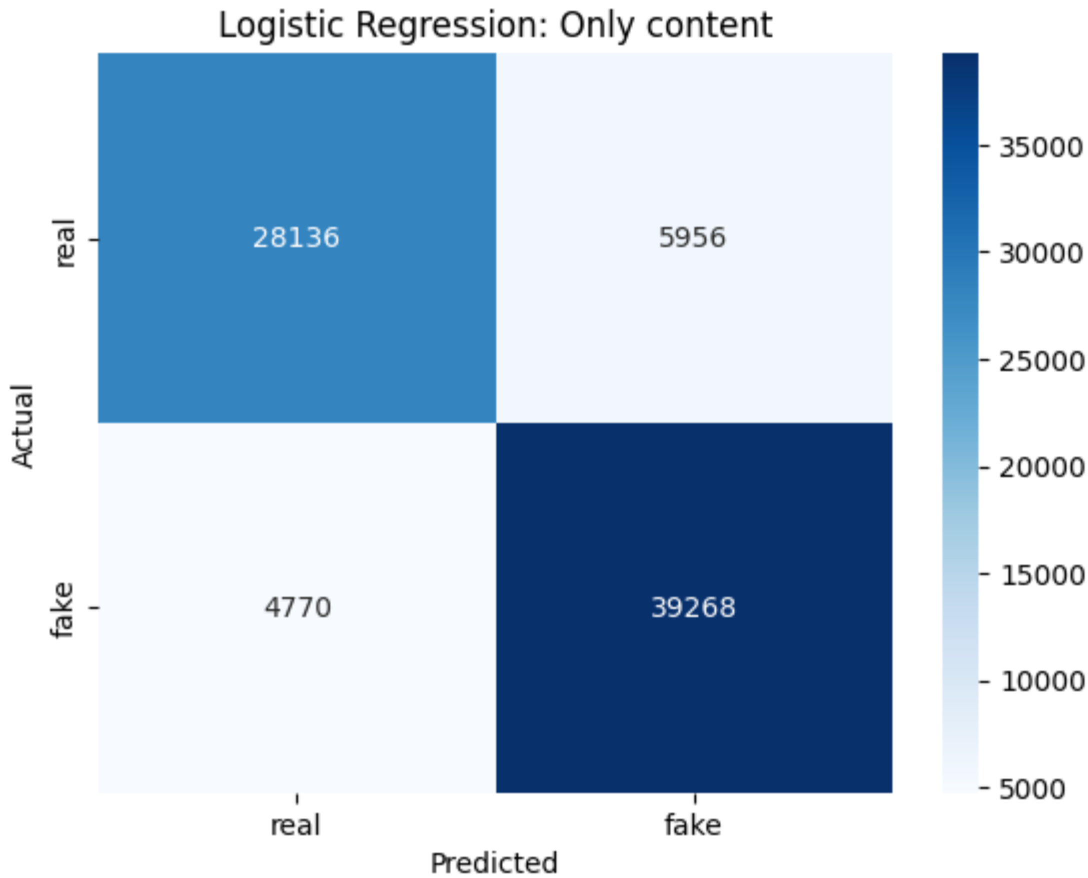
In [ ]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
import sklearn.metrics as metrics
from scipy.sparse import hstack
from joblib import dump


vectorrizer = CountVectorizer(lowercase = False, max_features = 10000, to
pipeline = Pipeline([
    ('vectorizer', vectorrizer),
    ('scaler', StandardScaler(with_mean=False))
    ])

model = LogisticRegression(max_iter=10000)

BoW_extra = pipeline.fit_transform(x_train_extra)
BoW_content_val = pipeline.transform(content_val)

model.fit(BoW_extra, y_train_extra)
y_pred = model.predict(BoW_content_val)
accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("\nOnly content, but with extra data:")
print("f1 score:", f1)
print("accuracy:", accuracy)
make_confusion_matrix(y_val, y_pred, "Logistic Regression: Only content,

BoW_content_train = pipeline.fit_transform(content_train)
BoW_content_val = pipeline.transform(content_val)

model.fit(BoW_content_train, y_train)
y_pred = model.predict(BoW_content_val)
accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("Only content:")
print("f1 score:", f1)
print("accuracy:", accuracy)
make_confusion_matrix(y_val, y_pred, "Logistic Regression: Only content")
dump(model, 'models/simple_model_content.joblib')

BoW_title_train = pipeline.fit_transform(title_train)
BoW_title_val = pipeline.transform(title_val)
BoW_combined_train = hstack((BoW_content_train, BoW_title_train))
BoW_combined_val = hstack((BoW_content_val, BoW_title_val))

model.fit(BoW_combined_train, y_train)
y_pred = model.predict(BoW_combined_val)
accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("\nContent and title:")
print("f1 score:", f1)
print("accuracy:", accuracy)
make_confusion_matrix(y_val, y_pred, "Logistic Regression: Content and ti
```

```
dump(model, 'models/simple_model_combined.joblib')
```

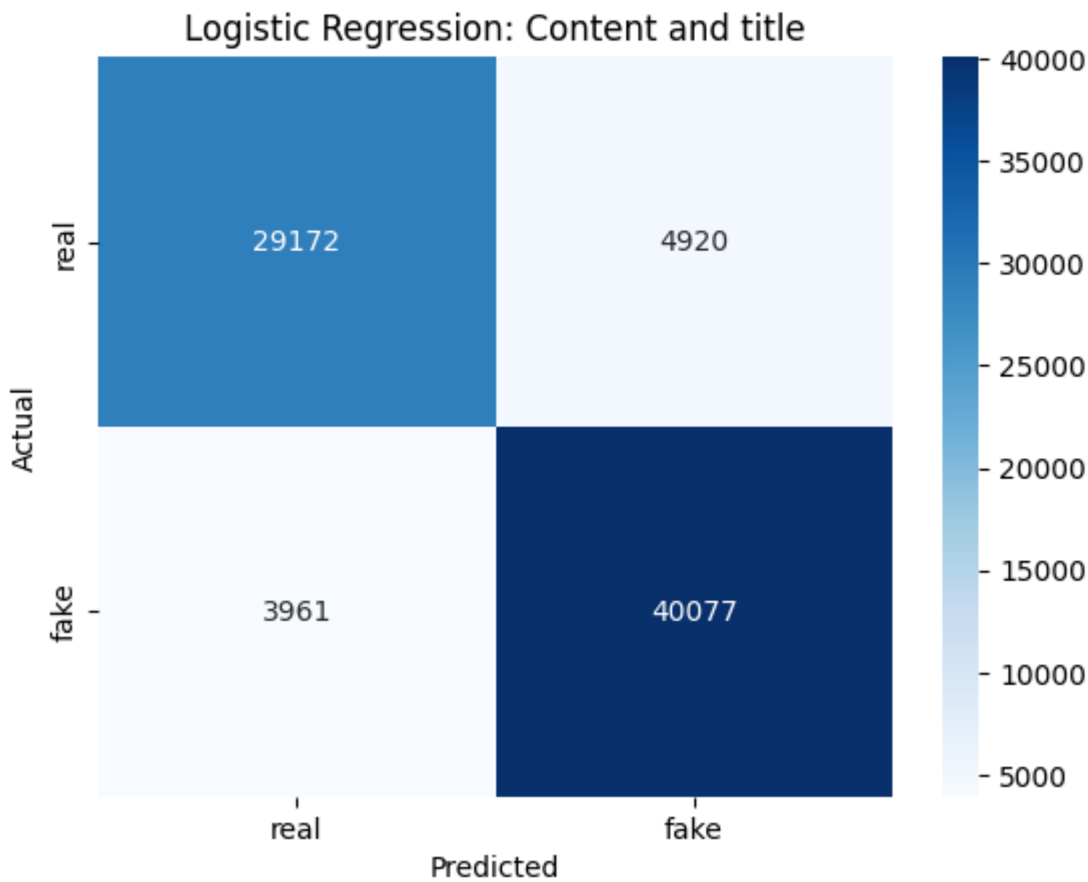
```
Only content, but with extra data:
f1 score: 0.8399767195451358
accuracy: 0.8627543837194419
```




```
Only content:
f1 score: 0.8399056688259352
accuracy: 0.8627159861768847
```




```
Content and title:
f1 score: 0.8678914094458907
accuracy: 0.8863304748496096
```



Out[ ]: ['models/simple_model_combined.joblib']

Part 3: Advanced model

3 models:

- LinearSVM
- Naive bayes
- Random forrest

2 vector representations:

- TF-IDF, 2 grams
- Word embedding (word2vec)

We perfrom cross validation on hyper paramaters to find the best hyperparameters for each model

Model 1: Linear SVC

In [ ]:
```
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
import sklearn.metrics as metrics
from joblib import dump
def svm(x_train, y_train, x_val, model_name):
    svc = LinearSVC(max_iter=10000, dual=False, random_state=42)
    parameters = dict(C=[0.001, 0.1, 1, 10])
    # Cross-validation
    grid_search = GridSearchCV(svc, parameters, cv=3, n_jobs=-1, scoring
    grid_search.fit(x_train, y_train)

    best_params = grid_search.best_params_
    print("Best Parameters for svm:", best_params)

    dump(grid_search, f'models/{model_name}.joblib')

    return grid_search.predict(x_val)
```

Model 2: Naive Bayes

In [ ]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
import sklearn.metrics as metrics
from joblib import dump

def naive_bayes(x_train, y_train, x_val, model_name):
    nb = MultinomialNB()
    parameters = dict(alpha=[0.01,0.1, 1, 10])
    # Cross-validation
    grid_search = GridSearchCV(nb, parameters, cv=3, n_jobs=-1, scoring =
    grid_search.fit(x_train, y_train)
```

```
    best_params = grid_search.best_params_
    print("Best parameters for Naive Bayes model:", best_params)

    dump(grid_search, f'models/{model_name}.joblib')

    return grid_search.predict(x_val)
```

Model 3: Logistic regression

We noticed our simple model performed quite well, therefore we tried to optimize hyperparameters and use n-grams to see if this would improve the simple model further

In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import sklearn.metrics as metrics
def logistic_advanced(x_train, y_train, x_val, model_name):
    logistic = LogisticRegression(max_iter = 10000)
    parameters = dict(C=[0.1, 1, 10], solver=['sag','saga'])

    grid_search = GridSearchCV(logistic, parameters, cv=3, n_jobs=-1, sco
    grid_search.fit(x_train, y_train)

    best_params = grid_search.best_params_
    print("Best parameters for logistic regression model:", best_params)

    dump(grid_search, f'models/{model_name}.joblib')

    return grid_search.predict(x_val)
```

TF-IDF

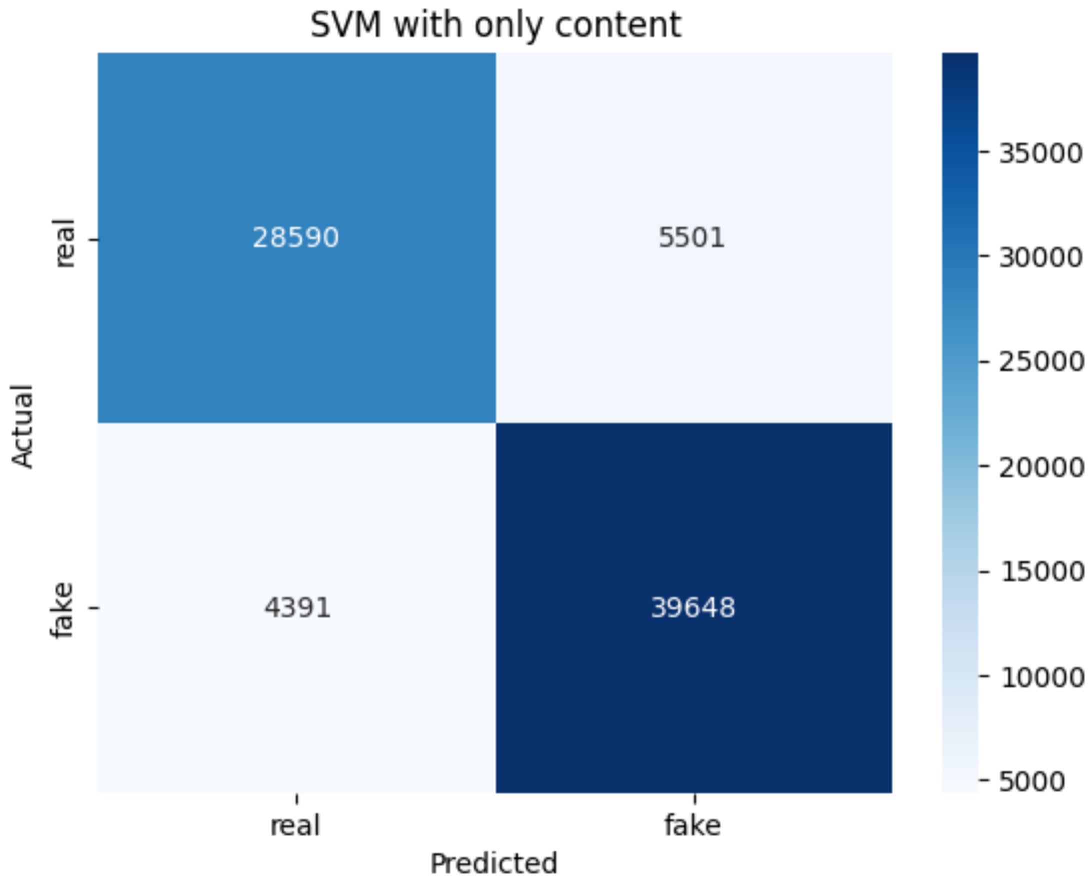
In [ ]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from scipy.sparse import hstack

def make_TFIDF(features, ngrams, metadata):
    global content_test, content_train, content_val, title_test, title_tr
    global y_test, y_train, y_val
    pipeline = Pipeline([
    ('vectorizer', TfidfVectorizer(lowercase = False,
                                   max_features=features,
                                   min_df = 1,
                                   max_df= 0.9,
                                   token_pattern=r'<[\w]+>|[\w]+',
                                   ngram_range =  ngrams)),
    ('scaler', StandardScaler(with_mean=False)),
    ])
    content_train_TFIDF = pipeline.fit_transform(content_train, y_train)
    content_val_TFIDF = pipeline.transform(content_val)
    content_test_TFIDF = pipeline.transform(content_test)

    title_train_TFIDF = pipeline.fit_transform(title_train, y_train)
    title_val_TFIDF = pipeline.transform(title_val)
    title_test_TFIDF = pipeline.transform(title_test)
```

```
    X_train_TFIDF = hstack((content_train_TFIDF, title_train_TFIDF))
    X_val_TFIDF = hstack((content_val_TFIDF, title_val_TFIDF))
    X_test_TFIDF = hstack((content_test_TFIDF, title_test_TFIDF))
    if metadata == "content":
        return content_train_TFIDF, content_val_TFIDF, content_test_TFIDF
    if metadata == "title":
        return title_train_TFIDF, title_val_TFIDF, title_test_TFIDF
    if metadata == "combined":
        return X_train_TFIDF, X_val_TFIDF, X_test_TFIDF
```

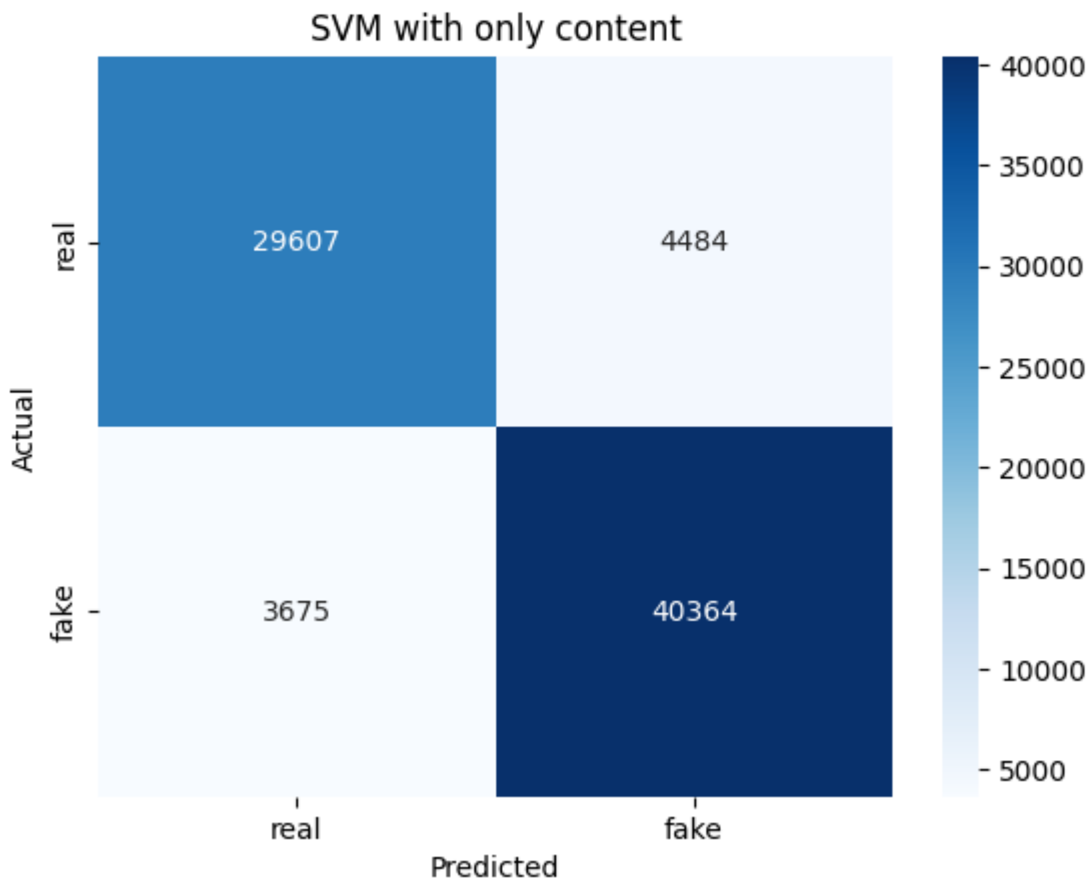
1 gram:

In [ ]:
```
from joblib import load
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(10000, (1, 1), "con
y_pred = svm(X_train_TFIDF, y_train, X_val_TFIDF, 'svm_1gram_content')
model = load('models/svm_1gram_content.joblib')
y_pred = model.predict(X_test_TFIDF)
accuracy = metrics.accuracy_score(y_test, y_pred)
f1 = metrics.f1_score(y_test, y_pred)
print("SVM with only content:")
print("f1 score:", f1)
print("accuracy:", accuracy)
make_confusion_matrix(y_test, y_pred, "SVM with only content")

X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(10000, (1, 1), "com
y_pred = svm(X_train_TFIDF, y_train, X_val_TFIDF, 'svm_1gram_combined')
model = load('models/svm_1gram_combined.joblib')
y_pred = model.predict(X_test_TFIDF)
accuracy = metrics.accuracy_score(y_test, y_pred)
f1 = metrics.f1_score(y_test, y_pred)
print("SVM with only content:")
print("f1 score:", f1)
print("accuracy:", accuracy)
make_confusion_matrix(y_test, y_pred, "SVM with only content")
```

```
Best Parameters for svm: {'C': 0.001}
SVM with only content:
f1 score: 0.8525166984732824
accuracy: 0.8733905030078075
```




```
Best Parameters for svm: {'C': 0.001}
SVM with only content:
f1 score: 0.8788980748964719
accuracy: 0.8955714834250608
```



In [ ]:
```
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(20000, (1, 1), "con
```

```
y_pred = svm(X_train_TFIDF, y_train, X_val_TFIDF, 'svm_1gram')

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("Support vector machine:")
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

```
Best Parameters for svm: {'C': 0.001}
Support vector machine:
f1 score: 0.861127234302213
accuracy score: 0.8805708434660182
```

In [ ]:
```
print("\nLogistic regression:")
y_pred = logistic_advanced(X_train_TFIDF, y_train, X_val_TFIDF, 'logistic

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

```
Logistic regression:
Best parameters for logistic regression model: {'C': 0.1, 'solver': 'sag
a'}
f1 score: 0.8796636423987606
accuracy score: 0.8955970817867657
```

In [ ]:
```
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(None, (1, 1))
print("\nNaive Bayes:")
y_pred = naive_bayes(X_train_TFIDF, y_train, X_val_TFIDF, 'naive_bayes_1g

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

2 grams:

In [ ]:
```
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(10000, (2, 2))
```

In [ ]:
```
y_pred = svm(X_train_TFIDF, y_train, X_val_TFIDF, 'svm_2gram')

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("Support vector machine:")
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

```
Best Parameters for svm: {'C': 10}
Support vector machine:
f1 score: 0.8613882402886861
accuracy score: 0.8829898886471266
```

In [ ]:
```
print("\nLogistic regression:")
y_pred = logistic_advanced(X_train_TFIDF, y_train, X_val_TFIDF, 'logistic

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

```
Logistic regression:
Best parameters for logistic regression model: {'C': 0.1, 'solver': 'sag
a'}
f1 score: 0.8614772144645172
accuracy score: 0.8824267246896199
```

In [ ]:
```
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(None, (2, 2))
print("\nNaive Bayes:")
y_pred = naive_bayes(X_train_TFIDF, y_train, X_val_TFIDF, 'naive_bayes_2g

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

3 grams:

In [ ]:
```
X_train_TFIDF, X_val_TFIDF, X_test_TFIDF = make_TFIDF(None, (3, 3))
print("\nNaive Bayes:")
y_pred = naive_bayes(X_train_TFIDF, y_train, X_val_TFIDF, 'naive_bayes_2g

accuracy = metrics.accuracy_score(y_val, y_pred)
f1 = metrics.f1_score(y_val, y_pred)
print("f1 score:", f1)
print("accuracy score:", accuracy)
```

Word2Vec & Doc2vec

In [ ]:
```
## Create a new split using x amount of each type of article
type_amount = 800

dfcpy_subset = dfcpy.groupby('type').head(type_amount)
print("Number of articles of each type in the new dataset:"
        ,dfcpy['type'].value_counts())

dfcpy_subset = dfcpy_subset.dropna(subset=['content'])
dfcpy_subset = dfcpy_subset.dropna(subset=['title'])

X = dfcpy_subset.content
y = dfcpy_subset.label

print("Training Set:")
print(train_df.content.head())
```

```
Number of articles of each type in the new dataset: type
reliable      218527
political     194445
bias          133179
fake          104850
conspiracy     88847
clickbait      27412
junksci        14039
Name: count, dtype: int64
Training Set:
779426    <num> year old iranian man share life iran wor...
51493     love love harlem said could walk favorit resta...
325203    plu one articl googl plu thank ali alfoneh ass...
272377    larri silverstein caught admit camera plan bui...
38281     artifici intellig complex creator cant trust m...
Name: content, dtype: object
```

Doc2Vec

In [ ]:

```python
from sklearn.model_selection import train_test_split
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from nltk.tokenize import word_tokenize
from sklearn.preprocessing import StandardScaler

def doc2vec(X, y, size, win, epo, model_name):
    doc2vec_model = Doc2Vec(vector_size=size, window=win, min_count=1, ep
    tagged_data = [TaggedDocument(words = word_tokenize(doc), tags=[i]) f
    doc2vec_model.build_vocab(tagged_data)
    doc2vec_model.train(tagged_data, total_examples = doc2vec_model.corpu
    doc_vectors = [doc2vec_model.infer_vector(word_tokenize(doc)) for doc
    # scale the data
    scaler = StandardScaler()
    doc_vectors = scaler.fit_transform(doc_vectors)
    X_train_D2V, X_rest_D2V, y_train_D2V, y_res_D2V = train_test_split(do
    X_val_D2V, X_test_D2V, y_val_D2V, y_test_D2V = train_test_split(X_res

    doc2vec_model.save(f'models/{model_name}.model')

    return X_train_D2V, X_val_D2V, X_test_D2V, y_train_D2V, y_val_D2V, y_
```

In [ ]:

```python
X_train_D2V, X_val_D2V, X_test_D2V, y_train_D2V, y_val_D2V, y_test_D2V =
y_pred = svm(X_train_D2V, y_train_D2V, X_val_D2V, 'svm_D2V')
accuracy = metrics.accuracy_score(y_val_D2V, y_pred)
f1 = metrics.f1_score(y_val_D2V, y_pred)
print("Support vector machine:")
print("f1 score:", f1)
print("accuracy score:", accuracy)

# print("\nNaive Bayes:")
# y_pred = naive_bayes(X_train_D2V, y_train_D2V, X_val_D2V, 'naive_bayes_

# accuracy = metrics.accuracy_score(y_val_D2V, y_pred)
# f1 = metrics.f1_score(y_val_D2V, y_pred)
# print("f1 score:", f1)
# print("accuracy score:", accuracy)

# print("\nLogistic regression:")
# y_pred = logistic_advanced(X_train_D2V, y_train_D2V, X_val_D2V, 'logist
```

```
# accuracy = metrics.accuracy_score(y_val_D2V, y_pred)
# f1 = metrics.f1_score(y_val_D2V, y_pred)
# print("f1 score:", f1)
# print("accuracy score:", accuracy)
```

```
Best Parameters for svm: {'C': 0.001}
Support vector machine:
f1 score: 0.802836879432624
accuracy score: 0.7517857142857143
```

Part 4: Evaluation

Logistic regression is slightly (0.xx%) better than linearsvc however it takes double the amount of time to train the logistic regression model, therefore we have chosen the Support vector machine instead as our model to test and evaluate

In [ ]:
```
# load the best model
from joblib import load
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import StandardScaler
import sklearn.metrics as metrics
from sklearn.pipeline import Pipeline

liar_train = pd.read_csv('train.tsv', sep='\t', header=None)
liar_val = pd.read_csv('valid.tsv', sep='\t', header=None)
liar_test = pd.read_csv('test.tsv', sep='\t', header=None)
liar = pd.concat([liar_train, liar_val, liar_test], ignore_index=True)
liar_cpy = liar.copy()

liar_cpy[2] = liar_cpy[2].apply(clean_text)
tokenizer = RegexpTokenizer(r'<[\w]+>|[\w]+')
liar_cpy[2] = liar_cpy[2].apply(tokenizer.tokenize)
liar_cpy[2] = liar_cpy[2].apply(rmv_stopwords)
liar_cpy[2] = liar_cpy[2].apply(stem_tokens)
liar_cpy[2] = liar_cpy[2].apply(lambda x: ' '.join(x))

labels_used = ['pants-fire', 'false', 'mostly-true', 'true']
liar_cpy = liar_cpy.dropna(subset=[1])
liar_cpy = liar_cpy[liar_cpy[1].isin(labels_used)]
liar_cpy[1] = liar_cpy[1].map({'pants-fire': 1,
                                         'false': 1,
                                         'mostly-true': 0,
                                         'true': 0})
liar_cpy = liar_cpy.dropna(subset=[2])

pipeline_bow = Pipeline([
    ('vectorizer', CountVectorizer(max_features=10000, token_pattern=r'<[
    ('scaler', StandardScaler(with_mean=False))
    ])

pipeline_tfidf = Pipeline([
    ('vectorizer', TfidfVectorizer(lowercase = False,
                                   max_features=10000,
                                   min_df = 1,
                                   max_df= 0.9,
```

```
                                   token_pattern=r'<[\w]+>|[\w]+',
                                   ngram_range =  (1, 1))),
    ('scaler', StandardScaler(with_mean=False)),
    ])
```

In [ ]:
```
content_test_bow = pipeline_bow.fit_transform(content_test, y_test)
liar_bow = pipeline_bow.fit_transform(liar_cpy[2], liar_cpy[1])

content_test_tfidf = pipeline_tfidf.fit_transform(content_test, y_test)
liar_tfidf = pipeline_tfidf.fit_transform(liar_cpy[2], liar_cpy[1])

simple_model = load('models/simple_model_content.joblib')
advanced_model = load('models/svm_1gram_content.joblib')

simple_pred_test = simple_model.predict(content_test_bow)
advanced_pred_test = advanced_model.predict(content_test_tfidf)
# simple_pred_liar = simple_model.predict(liar_bow)
# advanced_pred_liar = advanced_model.predict(liar_tfidf)

accuracy_simple = metrics.accuracy_score(y_test, simple_pred_test)
f1_simple = metrics.f1_score(y_test, simple_pred_test)
accuracy_advanced = metrics.accuracy_score(y_test, advanced_pred_test)
f1_advanced = metrics.f1_score(y_test, advanced_pred_test)
# accuracy_simple_liar = metrics.accuracy_score(liar_cpy[1], simple_pred_
# f1_simple_liar = metrics.f1_score(liar_cpy[1], simple_pred_liar)
# accuracy_advanced_liar = metrics.accuracy_score(liar_cpy[1], advanced_p
# f1_advanced_liar = metrics.f1_score(liar_cpy[1], advanced_pred_liar)

print("Simple model:")
print("Test set:")
print("f1 score:", f1_simple)
print("accuracy score:", accuracy_simple)
print("\nLiar dataset:")
# print("f1 score:", f1_simple_liar)
# print("accuracy score:", accuracy_simple_liar)

print("\nAdvanced model:")
print("Test set:")
print("f1 score:", f1_advanced)
print("accuracy score:", accuracy_advanced)
print("\nLiar dataset:")
# print("f1 score:", f1_advanced_liar)
# print("accuracy score:", accuracy_advanced_liar)
```

```
Simple model:
Test set:
f1 score: 0.5001292147821712
accuracy score: 0.5296173044925124

Liar dataset:

Advanced model:
Test set:
f1 score: 0.48453320081992673
accuracy score: 0.5751439907845898

Liar dataset:
```

In [ ]:
```
from scipy.sparse import hstack
content_test_bow = pipeline_bow.fit_transform(content_test, y_test)
```

```
title_test_bow = pipeline_bow.fit_transform(title_test, y_test)
combined_test_bow = hstack((content_test_bow, title_test_bow))

# need to be 10000 features, change max_features to 10000
# liar_bow = pipeline_bow.fit_transform(liar_cpy[2], liar_cpy[1])
# liar_tfidf = pipeline_tfidf.fit_transform(liar_cpy[2], liar_cpy[1])

content_test_tfidf = pipeline_bow.fit_transform(content_test, y_test)
title_test_tfidf = pipeline_bow.fit_transform(title_test,y_test)
combined_test_tfidf = hstack((content_test_tfidf, title_test_tfidf))


simple_model = load('models/simple_model_combined.joblib')
advanced_model = load('models/svm_1gram_combined.joblib')


simple_pred_test = simple_model.predict(combined_test_bow)
advanced_pred_test = advanced_model.predict(combined_test_tfidf)
# simple_pred_liar = simple_model.predict(liar_bow)
# advanced_pred_liar = advanced_model.predict(liar_tfidf)

accuracy_simple = metrics.accuracy_score(y_test, simple_pred_test)
f1_simple = metrics.f1_score(y_test, simple_pred_test)
accuracy_advanced = metrics.accuracy_score(y_test, advanced_pred_test)
f1_advanced = metrics.f1_score(y_test, advanced_pred_test)

# accuracy_advanced_liar = metrics.accuracy_score(liar_cpy[1], advanced_p
# accuracy_simple_liar = metrics.accuracy_score(liar_cpy[1], simple_pred_
# f1_simple_liar = metrics.f1_score(liar_cpy[1], simple_pred_liar)
# f1_advanced_liar = metrics.f1_score(liar_cpy[1], advanced_pred_liar)

print("Simple model:")
print("Test set:")
print("f1 score:", f1_simple)
print("accuracy score:", accuracy_simple)
# print("\nLiar dataset:")
# print("f1 score:", f1_simple_liar)
# print("accuracy score:", accuracy_simple_liar)

print("\nAdvanced model:")
print("Test set:")
print("f1 score:", f1_advanced)
print("accuracy score:", accuracy_advanced)
# print("\nLiar dataset:")
# print("f1 score:", f1_advanced_liar)
# print("accuracy score:", accuracy_advanced_liar)
```

```
Simple model:
Test set:
f1 score: 0.47132944284760325
accuracy score: 0.5338794317163702

Advanced model:
Test set:
f1 score: 0.43051105018004027
accuracy score: 0.5587098425700755
```