We will be importing the following packages to perform the analysis.

In [1]:
```
import numpy as np
import pandas as pd
import itertools
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt
```

Steps

1. Reading and Characterizing the Data `(ACQUIRE)`
2. Exploration `(PREPARE)`
3. Cleaning and Filtering the Data for our requirements `(PREPARE)`
4. Analysis `(ANALYZE)`
5. Results `(REPORT)`

I. Characterizing the data (ACQUIRE)

Code below is to `read` the data file into a pandas dataframe.

In [2]:
```
df=pd.read_csv('news.csv')
```

Before we start exploring the data, we want a clearer picture of the data. Hence we ask a few questions.

Q1. How many news records do we have?

Code below is to see the `number of rows and columns` in the data.

In [3]:
```
df.shape
```
Out[3]:
```
(6335, 4)
```

We can see that the dataframe has `4 columns` and `6335 records`.

Q2. What are the column names?

Code below is to see the `column` names of the dataframe

In [4]:
```
df.columns
```
Out[4]:
```
Index(['Unnamed: 0', 'title', 'text', 'label'], dtype='object')
```

We can see that only 3 out of 4 columns have names. They are `title`, `text`, `label`.

Q3. What does the data look like?

Code below is to see a `sample` of the data.

In [5]:
```
#df.head()
df.tail()
```
Out[5]:

	Unnamed: 0	title	text	label
6330	4490	State Department says it can't find emails fro...	The State Department told the Republican Natio...	REAL
6331	8062	The ‘P’ in PBS Should Stand for ‘Plutocratic’ ...	The ‘P’ in PBS Should Stand for ‘Plutocratic’ ...	FAKE
6332	8622	Anti-Trump Protesters Are Tools of the Oligarc...	Anti-Trump Protesters Are Tools of the Oligar...	FAKE
6333	4021	In Ethiopia, Obama seeks progress on peace, se...	ADDIS ABABA, Ethiopia —President Obama convene...	REAL
6334	4330	Jeb Bush Is Suddenly Attacking Trump. Here's W...	Jeb Bush Is Suddenly Attacking Trump. Here's W...	REAL

Now we have understood the `basic characteristics` of the data. Let's proceed with the next step.

II. Exploration (PREPARE)

Code below is to `create` a new dataframe that contains only the labels(REAL/FAKE).

In [6]:
```
labels = df.label
```

Code below is to `see` a sample of the newly created `labels` dataframe.

In [7]:
```
labels.head()
```

Out[7]:
```
0    FAKE
1    FAKE
2    REAL
3    FAKE
4    REAL
Name: label, dtype: object
```

Code below is to `split` the dataset into training and testing sets using the `train_test_split` function.

In [8]:
```
x_train, x_test, y_train, y_test = train_test_split(df['text'], labels, test_size=0.3, random_state=7)
```

Code below is to see the `training` datasets created using the `train_test_split` function.

In [9]:
```
x_train
#y_train
```

Out[9]:
```
4274    Home / Be The Change / Government Corruption /...
4310    At least a half-dozen attendees shoved and tac...
2050    As soon as Rep. Kevin McCarthy (R-Calif.) shoc...
4410    NOT ON THE SHORT LIST\n\nFormer Vice President...
3106    Eric Liu is the founder of Citizen University ...
                              ...
5699
2550    It's not that Americans won't elect wealthy pr...
537     Anyone writing sentences like 'nevertheless fu...
1220    More Catholics are in Congress than ever befor...
4271    It was hosted by CNN, and the presentation was...
Name: text, Length: 4434, dtype: object
```

Code below is to see the `testing` datasets created using the `train_test_split` function.

In [10]:
```
#x_test
y_test
```

Out[10]:
```
3534    REAL
6265    FAKE
3123    REAL
3940    REAL
2856    REAL
        ...
118     FAKE
3258    REAL
4521    FAKE
5926    FAKE
89      REAL
Name: label, Length: 1901, dtype: object
```

III. Cleaning and Filtering the Data for our requirements (PREPARE)

Stop words are the most common words in a language that are considered to be useless and is mostly filtered out before processing the natural language data. A simple way to `filter` out stopwords is to just use the `corpus` from `nltk` that we can download easily and `remove` the `stopwords` from our text using a `loop`. Let's use a different approach this time.

Term Frequency – Inverse Document Frequency

An alternative is to calculate `word frequencies`, is called `TF-IDF`.

This is an acronym than stands for `“Term Frequency – Inverse Document Frequency"` which are the components of the resulting scores assigned to each word.

- `Term Frequency`: This summarizes `how often` a given word appears within a document.
- `Inverse Document Frequency`: This `downscales` words that appear a lot across documents.

To simply state, `TF-IDF` are `word frequency scores` that try to `highlight` words that are `interesting`, rather than merely highlighting the most frequent words. We can `omit` words which `cross` a certain `threshold(frequency)`.

- A `TfidfVectorizer` turns a collection of raw documents into a `matrix` of `TF-IDF features`

- `max_df` is the `parameter` for `threshold` that we give as an input for the function.
- `max_df` can be set to a value in the range `[0.7, 1.0]` to `automatically` detect and `filter` stop words based on intra corpus document frequency of terms

Example: Stopwords like `the` is very commonly used and will most likely have document frequency higher than `0.7` in a given sample news text.

Let's initialize a `TfidfVectorizer` with maximum document frequency of `0.7` (terms with a higher document frequency will be discarded).

In [11]:
```
vectorizer = TfidfVectorizer(max_df=0.7)
```

The `fit` method, when applied to the `training` dataset, `learns` the `model parameters` . We should then apply the `transform` method on the training dataset to get the `transformed (scaled)` training dataset. Instead of performing them individually, we perform both of these steps in one step by applying `fit_transform` on the training dataset

Code below is to `fit` and `transform` the vectorizer on the `training` set.

In [12]:
```
tfidf_train = vectorizer.fit_transform(x_train)
```

But for `testing` set, Machine Learning applies `prediction` based on what was `learned` during the fitting of the training set and so it doesn't need to learn the models parameters, it directly performs the `transformation` .

Code below is to `transform` the vectorizer on the `test` set.

In [13]:
```
tfidf_test = vectorizer.transform(x_test)
```

Code below to is to see the transformed training and test datasets.

In [14]:

```
#print(tfidf_train)
print(tfidf_test)
```

```
  (0, 57224)	0.05519528105096844
  (0, 57178)	0.031192704431556396
  (0, 56860)	0.023712653791672828
  (0, 56793)	0.04250547812724162
  (0, 56783)	0.07367402497455956
  (0, 56305)	0.0977655117405021
  (0, 56197)	0.029044530183008413
  (0, 56172)	0.024490916832987555
  (0, 56150)	0.023453885650831386
  (0, 56080)	0.05048930873591531
  (0, 55877)	0.1127383261667559
  (0, 55672)	0.10262396941875963
  (0, 55601)	0.07458532488029349
  (0, 55239)	0.06591601888097635
  (0, 54939)	0.06283196078414083
  (0, 54424)	0.030434000427700194
  (0, 54292)	0.02428650983920786
  (0, 54226)	0.04242613384089784
  (0, 53748)	0.11573181749112645
  (0, 51943)	0.19609860066137222
  (0, 51623)	0.03358531291902588
  (0, 51592)	0.038177147673052726
  (0, 51497)	0.03312197768596556
  (0, 51470)	0.02148967190615234
  (0, 51426)	0.02362395217837374
  :	:
  (1900, 4928)	0.019341505823414613
  (1900, 3804)	0.014078619628178721
  (1900, 3767)	0.02002705524036257
  (1900, 3495)	0.015979793733507684
  (1900, 3433)	0.02987833791332498
  (1900, 3290)	0.01008872712538997
  (1900, 3267)	0.045897743854312854
  (1900, 3259)	0.018458331365985684
  (1900, 3213)	0.06690993887860666
  (1900, 3168)	0.009454303787877413
  (1900, 2887)	0.027246814620999194
  (1900, 2867)	0.05397074111861263
```

```
  (1900, 2838)	0.02148166745946232
  (1900, 2815)	0.034435373095004655
  (1900, 2778)	0.0323437430969293
  (1900, 2756)	0.024703404395062703
  (1900, 2737)	0.01624512944973044
  (1900, 2735)	0.037969113344949915
  (1900, 2721)	0.01048730126970561
  (1900, 2317)	0.01735203645639521
  (1900, 1970)	0.07328349111234908
  (1900, 1020)	0.04744390200517566
  (1900, 631)	0.021224832383165613
  (1900, 273)	0.019881514371597356
  (1900, 1)	0.017138061540192425
```

Code below is to see the words which had a document frequency `less than 0.7`.

In [15]:

```python
print(vectorizer.vocabulary_)
```

```
40214, 'foreign': 20618, 'until': 54226, 'proven': 40816, 'otherwise': 36924, 'bad': 5454, 'none': 35649, 'neat': 350
07, 'colorblind': 11125, 'ultimately': 53270, 'deepest': 14027, 'racial': 41541, 'start': 48864, 'lazy': 29906, 'mess
y': 32888, 'historical': 24541, 'truths': 52892, 'stories': 49272, 'story': 49281, 'endings': 17683, 'cheap': 9939,
'point': 39413, 'mlk': 33631, 'redemption': 42440, 'easy': 16887, 'honor': 24819, 'official': 36374, 'commissions': 1
1287, 'conversations': 12159, 'ourselves': 36960, 'washington': 55765, 'continues': 12057, 'force': 20577, 'heading':
23902, 'election': 17222, 'orc': 36737, 'poll': 39516, 'broad': 8087, 'lead': 29925, 'field': 19839, 'democratic': 14
353, 'challengers': 9752, 'nomination': 35615, 'contest': 12031, 'sizable': 47258, 'contenders': 12020, 'side': 4694
7, 'general': 21742, 'match': 32175, 'ups': 54352, 'candidates': 8909, 'gets': 21924, 'within': 56609, '10': 109, 'po
ints': 39422, 'hypothetical': 25393, 'matchups': 32180, 'rand': 41778, 'closest': 10752, '43': 1100, 'likely': 30474,
'54': 1266, 'walker': 55602, 'equally': 18053, 'carrying': 9218, '55': 1275, 'huckabee': 25115, '41': 1080, 'carson':
9221, '56': 1282, 'warren': 55740, 'decide': 13891, 'stands': 48811, 'benefit': 6429, 'gaining': 21436, 'holding': 24
674, '67': 1410, '16': 312, 'advantage': 2557, 'biden': 6708, 'backers': 5400, 'allocated': 3219, 'choice': 10196, 'n
otably': 35808, 'surges': 50065, '74': 1485, 'broadly': 8100, 'believe': 6339, 'chances': 9778, 'hold': 24670, 'stron
gest': 49466, '68': 1415, 'better': 6609, 'shot': 46772, 'ticket': 51714, 'favorability': 19470, 'rating': 41917, 're
cently': 42244, 'prospects': 40741, 'appear': 4059, 'unchanged': 53423, 'compared': 11357, 'polls': 39529, 'conducte
d': 11624, 'broke': 8123, 'personal': 38528, 'email': 17389, 'address': 2391, 'based': 5907, 'server': 46179, 'servin
g': 46192, 'leads': 29935, 'pack': 37405, 'follows': 20502, '13': 226, 'nearly': 35004, 'matches': 32177, '12': 199,
'holds': 24680, 'backing': 5410, 'dipped': 15261, 'significantly': 47027, 'february': 19563, 'generally': 21750, 'ste
ady': 48974, 'single': 47163, 'digits': 15154, 'jersey': 27881, 'gov': 22489, 'christie': 10268, 'sen': 46034, 'marc
```

If we explore the words printed above, we can notice that `stopwords` have been `removed` automatically.

IV. Analysis (ANALYZE)

Passive Aggresive Classifier

This type of classifier is generally used for large-scale learning. It is one of the few `online-learning algorithms` . In online machine learning algorithms, the input data comes in sequential order and the machine learning model is `updated step-by-step` , as opposed to batch learning, where the entire training dataset is used at once.

- `Passive` : If the prediction is correct, keep the model and do not make any changes. i.e., the data in the example is not enough to cause any changes in the model.
- `Aggressive` : If the prediction is incorrect, make changes to the model. i.e., some change to the model may correct it.
- The parameter `max_iter` denotes the maximum number of iterations the model makes over the training data.

This is very useful in situations where there is a huge amount of data and it is `computationally infeasible` to train the entire dataset because of the sheer size of the data. We can simply say that an online-learning algorithm will `get` a training example, `update` the classifier, and then `throw away the example` .

A very good example of this would be to detect fake news on a social media website like Twitter, where new data is being added every second. To dynamically read data from Twitter continuously, the data would be huge, and using an online-learning algorithm would be the ideal choice.

First, we Initialize a `PassiveAggressiveClassifier` . Then we `fit` this on `tfidf_train` and `y_train` . Post fitting, we predict on the test set.

Code below to `initialize` a Passive Aggressive Classifier.

In [16]:
```
pac = PassiveAggressiveClassifier(max_iter=50)
```

Code below to `fit` the classifier using the training datasets.

In [17]:
```
pac.fit(tfidf_train,y_train)
```

Out[17]:
```
PassiveAggressiveClassifier(C=1.0, average=False, class_weight=None,
                            early_stopping=False, fit_intercept=True,
                            loss='hinge', max_iter=50, n_iter_no_change=5,
                            n_jobs=None, random_state=None, shuffle=True,
                            tol=0.001, validation_fraction=0.1, verbose=0,
                            warm_start=False)
```

We can see above that we have created a `Passive Aggressive Classifier` .

Code below to `predict` on the test set from the TfidfVectorizer

In [18]:
```
y_pred = pac.predict(tfidf_test)
```

Calculate the accuracy with `accuracy_score()` from `sklearn.metrics` .

In [19]:
```
score=accuracy_score(y_test,y_pred)
```

V. Results (REPORT)

Code below to assign the accuracy value to a variable for convenience.

In [20]:
```
d = round(score*100,2)
```

Code below to print the accuracy of the model.

In [21]:
```
print("Accuracy: ",d,"%")
```

```
Accuracy:  92.9 %
```

Confusion matrix is used to display the exact predictions of a classifier model.

Code below to build a confusion matrix

In [22]:
```
cmatrix = confusion_matrix(y_test,y_pred, labels=['FAKE','REAL'])
```

Code below to print the confusion matrix.

In [23]:
```
print(cmatrix)
```

```
[[904  70]
 [ 65 862]]
```

To better understand the confusion matrix, we will access the matrix and display the data in a easily understandable manner.

Code below to do the above mentioned actions.

In [24]:
```
w = cmatrix[0][0]
x = cmatrix[0][1]
y = cmatrix[1][1]
z = cmatrix[1][0]
print("True Positives =",w)
print("False Positives =",x)
print("True Negatives =",y)
print("False Negatives =",z)
```

```
True Positives = 904
False Positives = 70
True Negatives = 862
False Negatives = 65
```

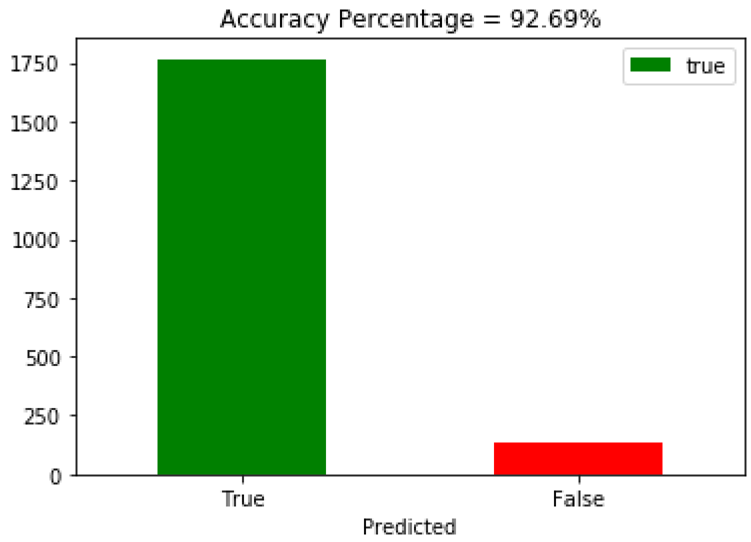
In order to further understand our findings, we can vizualise it.

Code below is to vizualise our findings.

In [25]:

```python
a = w + y
b = x + z
df = pd.DataFrame({'Predicted':['True', 'False'], 'true':[a, b]})

ax = df.plot.bar(x='Predicted', y='true', rot=0, title="Accuracy Percentage = 92.69%", color=['green', 'red'])
```



Code below to print the conclusion.

In [26]:

```
print("So with this model, we have ",w+y," correctly predicted labels, and ",x+z," wrongly predicted labels.")
```

```
So with this model, we have  1766  correctly predicted labels, and  135  wrongly predicted labels.
```

In [ ]:

```

```