**GitHub Link**: <https://github.com/NITIN0601/FeatureEngineering-ICE1-Assignemnt_Data>

**Wiki Content**

**# Data Creation**

# Importing libraries

!pip install GoogleNews

!pip install newspaper3k

from GoogleNews import GoogleNews

from newspaper import Article

from newspaper import Config

import pandas as pd

import nltk

nltk.download('punkt')

# Configuring browser

user\_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36'

config = Config()

config.browser\_user\_agent = user\_agent

# Searching the autonomous articles

googlenews=GoogleNews(start='01/01/2012',end='04/09/2018')

googlenews.search('Autonomous Cars')

result=googlenews.result()

df=pd.DataFrame(result)

# Fetching the articles

for i in range(8,19):

googlenews.getpage(i)

result=googlenews.result()

df=pd.DataFrame(result)

# Removing unavailable articles

update\_df = df.drop([8,47,50,117,200,221])

# getting the full article and storing in excel

listd = [] # Storing the data into a list

for i in update\_df.index:

d={}

article = Article(update\_df['link'][i],config=config)

article.download()

article.parse()

article.nlp()

d['Date']=update\_df['date'][i]

d['Media']=update\_df['media'][i]

d['Title']=article.title

d['Article']=article.text

d['Summary']=article.summary

listd.append(d)

df\_news=pd.DataFrame(listd)

df\_news.to\_excel("articles.xlsx")

# Mounting the Google drive

from google.colab import drive

drive.mount('/content/drive')

# Reading dataset

df\_bbc = pd.read\_csv('/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/News\_dataset.csv', sep=';')

# Creating autonomous category

df\_news['Category'] = 'autonomous car'

# Extracting the required columns and creating a new dataframe

df\_auto = df\_news[['Article','Summary','Category']]

df\_auto.rename(columns = {'Article':'File\_Name','Summary':'Content'},inplace = True)

# Removing tech articles

df\_bbc = df\_bbc[df\_bbc['Category'] != 'tech'].reset\_index(drop = True)

print(df\_bbc.head())

# Concatenating bbc news with autonomous articles

frames = [df\_auto,df\_bbc]

df\_final\_news = pd.DataFrame(pd.concat(frames,ignore\_index = True))

print(df\_final\_data.head())

# Saving the dataframe to csv

df\_final\_data = df\_final\_data.to\_csv('/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/dataset.csv')

**# Exploratory Data Analysis**

# Importing libraries

import pandas as pd

import matplotlib.pyplot as plt

import pickle

import seaborn as sns

sns.set\_style("whitegrid")

import altair as alt

#Code for hiding warnings

import warnings

warnings.filterwarnings("ignore")

# Mounting the Google drive

from google.colab import drive

drive.mount('/content/drive')

# Loading dataset

df\_path = "/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/"

df\_path2 = df\_path + 'dataset.csv'

df = pd.read\_csv(df\_path2)

# Number of articles in each category

bars = alt.Chart(df).mark\_bar(size=50).encode(

x=alt.X("Category"),

y=alt.Y("count():Q", axis=alt.Axis(title='Number of articles')),

tooltip=[alt.Tooltip('count()', title='Number of articles'), 'Category'],

color='Category'

)

text = bars.mark\_text(

align='center',

baseline='bottom',

).encode(

text='count()'

)

(df['id'] = 1

df2 = pd.DataFrame(df.groupby('Category').count()['id']).reset\_index()

bars = alt.Chart(df2).mark\_bar(size=50).encode(

x=alt.X('Category'),

y=alt.Y('PercentOfTotal:Q', axis=alt.Axis(format='.0%', title='% of Articles')),

color='Category'

).transform\_window(

TotalArticles='sum(id)',

frame=[None, None]

).transform\_calculate(

PercentOfTotal="datum.id / datum.TotalArticles"

)

text = bars.mark\_text(

align='center',

baseline='bottom',

#dx=5 # Nudges text to right so it doesn't appear on top of the bar

).encode(

text=alt.Text('PercentOfTotal:Q', format='.1%')

)

(bars + text).interactive().properties(

height=300,

width=700,

title = "% of articles in each category",

)bars + text).interactive().properties(

height=300,

width=700,

title = "Number of articles in each category",

)

Chart, bar chart

Description automatically generated

# % of articles in each category

df['id'] = 1

df2 = pd.DataFrame(df.groupby('Category').count()['id']).reset\_index()

bars = alt.Chart(df2).mark\_bar(size=50).encode(

x=alt.X('Category'),

y=alt.Y('PercentOfTotal:Q', axis=alt.Axis(format='.0%', title='% of Articles')),

color='Category'

).transform\_window(

TotalArticles='sum(id)',

frame=[None, None]

).transform\_calculate(

PercentOfTotal="datum.id / datum.TotalArticles"

)

text = bars.mark\_text(

align='center',

baseline='bottom',

#dx=5 # Nudges text to right so it doesn't appear on top of the bar

).encode(

text=alt.Text('PercentOfTotal:Q', format='.1%')

)

(bars + text).interactive().properties(

height=300,

width=700,

title = "% of articles in each category",)

Chart, bar chart

Description automatically generated

# News length by category

df['News\_length'] = df['Content'].str.len()

plt.figure(figsize=(13,8))

sns.distplot(df['News\_length']).set\_title('News length distribution');

Chart

Description automatically generated

df['News\_length'].describe()  
  
# Let's remove from the 95% percentile onwards to better appreciate the histogram:

quant\_95 = df['News\_length'].quantile(0.95)

df95 = df[df['News\_length'] < quant\_95]

plt.figure(figsize=(13,8))

sns.distplot(df95['News\_length']).set\_title('News length distribution');

We can get the number of news articles with more than 10,000 characters:

more10k = df[df['News\_length'] > 10000]

len(more10k)

Let's see one:

more10k['Content'].iloc[0]

Let's now plot a boxplot:

plt.figure(figsize=(13,8))

sns.boxplot(data=df, x='Category', y='News\_length', width=.8);

Chart, box and whisker chart

Description automatically generated

Now, let's remove the larger documents for better comprehension:

plt.figure(figsize=(13,8))

sns.boxplot(data=df95, x='Category', y='News\_length',width=.8);

Chart, box and whisker chart

Description automatically generated

We can see that, although the length distribution is different for every category, the difference is not too big. If we had way too different lengths between categories we would have a problem since the feature creation process may take into account counts of words. However, when creating the features with TF-IDF scoring, we will normalize the features just to avoid this.

At this point, we cannot do further Exploratory Data Analysis. We'll turn onto the Feature Engineering section.

We'll save the dataset:

with open('/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/News\_dataset.pickle', 'wb') as output:

pickle.dump(df, output)

**# Feature Engineering**

The next step is to create features from the raw text so we can train the machine learning models. The steps followed are:

1. \*\*Text Cleaning and Preparation\*\*: cleaning of special characters, downcasing, punctuation signs. possessive pronouns and stop words removal and lemmatization.

2. \*\*Label coding\*\*: creation of a dictionary to map each category to a code.

3. \*\*Train-test split\*\*: to test the models on unseen data.

4. \*\*Text representation\*\*: use of TF-IDF scores to represent text.

# Importing libraries

import pickle

import pandas as pd

import re

import nltk

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_selection import chi2

import numpy as np

First of all we'll load the dataset:

Mounting Drive:

from google.colab import drive

drive.mount('/content/drive')

pathcolab='/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Already\_Created\_Set/Latest-News-Classifier/0. Latest News Classifier/02. Exploratory Data Analysis/'

path\_df = 'News\_dataset.pickle'

with open(pathcolab+path\_df, 'rb') as data:

df = pickle.load(data)

df.head()

And visualize one sample news content:

df.loc[1]['Content']

1. Text cleaning and preparation

1.1. Special character cleaning

We can see the following special characters:

\* ``\r``

\* ``\n``

\* ``\`` before possessive pronouns (`government's = government\'s`)

\* ``\`` before possessive pronouns 2 (`Yukos'` = `Yukos\'`)

\* ``"`` when quoting text

# \r and \n

df['Content\_Parsed\_1'] = df['Content'].str.replace("\r", " ")

df['Content\_Parsed\_1'] = df['Content\_Parsed\_1'].str.replace("\n", " ")

df['Content\_Parsed\_1'] = df['Content\_Parsed\_1'].str.replace(" ", " ")

Regarding 3rd and 4th bullet, although it seems there is a special character, it won't affect us since it is not a \*real\* character:

text = "Mr Greenspan\'s"

text

# " when quoting text

df['Content\_Parsed\_1'] = df['Content\_Parsed\_1'].str.replace('"', '')

#1.2. Upcase/downcase

We'll downcase the texts because we want, for example, `Football` and `football` to be the same word.

# Lowercasing the text

df['Content\_Parsed\_2'] = df['Content\_Parsed\_1'].str.lower()

# 1.3. Punctuation signs

Punctuation signs won't have any predicting power, so we'll just get rid of them.

punctuation\_signs = list("?:!.,;")

df['Content\_Parsed\_3'] = df['Content\_Parsed\_2']

for punct\_sign in punctuation\_signs:

df['Content\_Parsed\_3'] = df['Content\_Parsed\_3'].str.replace(punct\_sign, '')

By doing this we are messing up with some numbers, but it's no problem since we aren't expecting any predicting power from them.

### 1.4. Possessive pronouns

We'll also remove possessive pronoun terminations:

df['Content\_Parsed\_4'] = df['Content\_Parsed\_3'].str.replace("'s", "")

### 1.5. Stemming and Lemmatization

Since stemming can produce output words that don't exist, we'll only use a lemmatization process at this moment. Lemmatization takes into consideration the morphological analysis of the words and returns words that do exist, so it will be more useful for us.

# Downloading punkt and wordnet from NLTK

nltk.download('punkt')

print("------------------------------------------------------------")

nltk.download('wordnet')

# Saving the lemmatizer into an object

wordnet\_lemmatizer = WordNetLemmatizer()

In order to lemmatize, we have to iterate through every word:

nrows = len(df)

lemmatized\_text\_list = []

for row in range(0, nrows):

# Create an empty list containing lemmatized words

lemmatized\_list = []

# Save the text and its words into an object

text = df.loc[row]['Content\_Parsed\_4']

text\_words = text.split(" ")

# Iterate through every word to lemmatize

for word in text\_words:

lemmatized\_list.append(wordnet\_lemmatizer.lemmatize(word, pos="v"))

# Join the list

lemmatized\_text = " ".join(lemmatized\_list)

# Append to the list containing the texts

lemmatized\_text\_list.append(lemmatized\_text)

df['Content\_Parsed\_5'] = lemmatized\_text\_list

Although lemmatization doesn't work perfectly in all cases (as can be seen in the example below), it can be useful.

### 1.6. Stop words

# Downloading the stop words list

nltk.download('stopwords')

# Loading the stop words in english

stop\_words = list(stopwords.words('english'))

stop\_words[0:10]

To remove the stop words, we'll handle a regular expression only detecting whole words, as seen in the following example:

example = "me eating a meal"

word = "me"

# The regular expression is:

regex = r"\b" + word + r"\b" # we need to build it like that to work properly

re.sub(regex, "StopWord", example)

We can now loop through all the stop words:

df['Content\_Parsed\_6'] = df['Content\_Parsed\_5']

for stop\_word in stop\_words:

regex\_stopword = r"\b" + stop\_word + r"\b"

df['Content\_Parsed\_6'] = df['Content\_Parsed\_6'].str.replace(regex\_stopword, '')

We have some dobule/triple spaces between words because of the replacements. However, it's not a problem because we'll tokenize by the spaces later.

As an example, we'll show an original news article and its modifications throughout the process:

df.loc[5]['Content']

1. Special character cleaning

df.loc[5]['Content\_Parsed\_1']

2. Upcase/downcase

df.loc[5]['Content\_Parsed\_2']

3. Punctuation signs

df.loc[5]['Content\_Parsed\_3']

4. Possessive pronouns

df.loc[5]['Content\_Parsed\_4']

5. Stemming and Lemmatization

df.loc[5]['Content\_Parsed\_5']

6. Stop words

df.loc[5]['Content\_Parsed\_6']

Finally, we can delete the intermediate columns:

df.head(1)

list\_columns = ["File\_Name", "Category", "Complete\_Filename", "Content", "Content\_Parsed\_6"]

df = df[list\_columns]

df = df.rename(columns={'Content\_Parsed\_6': 'Content\_Parsed'})

df.head()

\*\*IMPORTANT:\*\*

We need to remember that our model will gather the latest news articles from different newspapers every time we want. For that reason, we not only need to take into account the peculiarities of the training set articles, but also possible ones that are present in the gathered news articles.

For this reason, possible peculiarities have been studied in the \*05. News Scraping\* folder.

## 2. Label coding

We'll create a dictionary with the label codification:

category\_codes = {

'business': 0,

'entertainment': 1,

'politics': 2,

'sport': 3,

'autonomus cars: 4

}

# Category mapping

df['Category\_Code'] = df['Category']

df = df.replace({'Category\_Code':category\_codes})

## 3. Train - test split

We'll set apart a test set to prove the quality of our models. We'll do Cross Validation in the train set in order to tune the hyperparameters and then test performance on the unseen data of the test set.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Content\_Parsed'],

df['Category\_Code'],

test\_size=0.15,

random\_state=8)

Since we don't have much observations (only 2.225), we'll choose a test set size of 15% of the full dataset.

## 4. Text representation

We have various options:

\* Count Vectors as features

\* TF-IDF Vectors as features

\* Word Embeddings as features

\* Text / NLP based features

\* Topic Models as features

We'll use \*\*TF-IDF Vectors\*\* as features.

We have to define the different parameters:

\* `ngram\_range`: We want to consider both unigrams and bigrams.

\* `max\_df`: When building the vocabulary ignore terms that have a document

frequency strictly higher than the given threshold

\* `min\_df`: When building the vocabulary ignore terms that have a document

frequency strictly lower than the given threshold.

\* `max\_features`: If not None, build a vocabulary that only consider the top

max\_features ordered by term frequency across the corpus.

See `TfidfVectorizer?` for further detail.

It needs to be mentioned that we are implicitly scaling our data when representing it as TF-IDF features with the argument `norm`.

# Parameter election

ngram\_range = (1,2)

min\_df = 10

max\_df = 1.

max\_features = 300

We have chosen these values as a first approximation. Since the models that we develop later have a very good predictive power, we'll stick to these values. But it has to be mentioned that different combinations could be tried in order to improve even more the accuracy of the models.

tfidf = TfidfVectorizer(encoding='utf-8',

ngram\_range=ngram\_range,

stop\_words=None,

lowercase=False,

max\_df=max\_df,

min\_df=min\_df,

max\_features=max\_features,

norm='l2',

sublinear\_tf=True)

features\_train = tfidf.fit\_transform(X\_train).toarray()

labels\_train = y\_train

print(features\_train.shape)

features\_test = tfidf.transform(X\_test).toarray()

labels\_test = y\_test

print(features\_test.shape)

Please note that we have fitted and then transformed the training set, but we have \*\*only transformed\*\* the \*\*test set\*\*.

We can use the Chi squared test in order to see what unigrams and bigrams are most correlated with each category:

from sklearn.feature\_selection import chi2

import numpy as np

for Product, category\_id in sorted(category\_codes.items()):

features\_chi2 = chi2(features\_train, labels\_train == category\_id)

indices = np.argsort(features\_chi2[0])

feature\_names = np.array(tfidf.get\_feature\_names())[indices]

unigrams = [v for v in feature\_names if len(v.split(' ')) == 1]

bigrams = [v for v in feature\_names if len(v.split(' ')) == 2]

print("# '{}' category:".format(Product))

print(" . Most correlated unigrams:\n. {}".format('\n. '.join(unigrams[-5:])))

print(" . Most correlated bigrams:\n. {}".format('\n. '.join(bigrams[-2:])))

print("")

As we can see, the unigrams correspond well to their category. However, bigrams do not. If we get the bigrams in our features:

bigrams

We can see there are only six. This means the unigrams have more correlation with the category than the bigrams, and since we're restricting the number of features to the most representative 300, only a few bigrams are being considered.

Let's save the files we'll need in the next steps:

PathColab='/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/Pickles/'

# X\_train

with open(PathColab+'X\_test.pickle', 'wb') as output:

pickle.dump(X\_train, output)

# X\_test

with open(PathColab+'X\_train.pickle', 'wb') as output:

pickle.dump(X\_test, output)

# y\_train

with open(PathColab+'y\_train.pickle', 'wb') as output:

pickle.dump(y\_train, output)

# y\_test

with open(PathColab+'y\_test.pickle', 'wb') as output:

pickle.dump(y\_test, output)

# df

with open(PathColab+'df.pickle', 'wb') as output:

pickle.dump(df, output)

# features\_train

with open(PathColab+'features\_train.pickle', 'wb') as output:

pickle.dump(features\_train, output)

# labels\_train

with open(PathColab+'labels\_train.pickle', 'wb') as output:

pickle.dump(labels\_train, output)

# features\_test

with open(PathColab+'features\_test.pickle', 'wb') as output:

pickle.dump(features\_test, output)

# labels\_test

with open(PathColab+'labels\_test.pickle', 'wb') as output:

pickle.dump(labels\_test, output)

# TF-IDF object

with open(PathColab+'tfidf.pickle', 'wb') as output:

pickle.dump(tfidf, output)

**# Model Training**

**# Random Forest**

# Importing libraries

import pickle

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.ensemble import RandomForestClassifier

from pprint import pprint

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

from sklearn.model\_selection import ShuffleSplit

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

First, we load the data:

Mounting Drive:

from google.colab import drive

drive.mount('/content/drive')

pathcolab = '/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/dataset.csv/Pickles/'

# Dataframe

path\_df = "df.pickle"

with open(pathcolab+path\_df, 'rb') as data:

df = pickle.load(data)

# features\_train

path\_features\_train = "features\_train.pickle"

with open(pathcolab+path\_features\_train, 'rb') as data:

features\_train = pickle.load(data)

# labels\_train

path\_labels\_train = "labels\_train.pickle"

with open(pathcolab+path\_labels\_train, 'rb') as data:

labels\_train = pickle.load(data)

# features\_test

path\_features\_test = "features\_test.pickle"

with open(pathcolab+path\_features\_test, 'rb') as data:

features\_test = pickle.load(data)

# labels\_test

path\_labels\_test = "labels\_test.pickle"

with open(pathcolab+path\_labels\_test, 'rb') as data:

labels\_test = pickle.load(data)

Let's check the dimension of our feature vectors:

print(features\_train.shape)

print(features\_test.shape)

Cross-Validation for Hyperparameter tuning

rf\_0 = RandomForestClassifier(random\_state = 8)

print('Parameters currently in use:\n')

print(rf\_0.get\_params())

Randomized Search Cross Validation

# n\_estimators

n\_estimators = [int(x) for x in np.linspace(start = 200, stop = 1000, num = 5)]

# max\_features

max\_features = ['auto', 'sqrt']

# max\_depth

max\_depth = [int(x) for x in np.linspace(20, 100, num = 5)]

max\_depth.append(None)

# min\_samples\_split

min\_samples\_split = [2, 5, 10]

# min\_samples\_leaf

min\_samples\_leaf = [1, 2, 4]

# bootstrap

bootstrap = [True, False]

# Create the random grid

random\_grid = {'n\_estimators': n\_estimators,

'max\_features': max\_features,

'max\_depth': max\_depth,

'min\_samples\_split': min\_samples\_split,

'min\_samples\_leaf': min\_samples\_leaf,

'bootstrap': bootstrap}

pprint(random\_grid)

# First create the base model to tune

rfc = RandomForestClassifier(random\_state=8)

# Definition of the random search

random\_search = RandomizedSearchCV(estimator=rfc,

param\_distributions=random\_grid,

n\_iter=50,

scoring='accuracy',

cv=3,

verbose=1,

random\_state=8)

# Fit the random search model

random\_search.fit(features\_train, labels\_train)

print("The best hyperparameters from Random Search are:")

print(random\_search.best\_params\_)

print("")

print("The mean accuracy of a model with these hyperparameters is:")

print(random\_search.best\_score\_)

### Grid Search Cross Validation

# Create the parameter grid based on the results of random search

bootstrap = [False]

max\_depth = [30, 40, 50]

max\_features = ['sqrt']

min\_samples\_leaf = [1, 2, 4]

min\_samples\_split = [5, 10, 15]

n\_estimators = [800]

param\_grid = {

'bootstrap': bootstrap,

'max\_depth': max\_depth,

'max\_features': max\_features,

'min\_samples\_leaf': min\_samples\_leaf,

'min\_samples\_split': min\_samples\_split,

'n\_estimators': n\_estimators

}

# Create a base model

rfc = RandomForestClassifier(random\_state=8)

# Manually create the splits in CV in order to be able to fix a random\_state (GridSearchCV doesn't have that argument)

cv\_sets = ShuffleSplit(n\_splits = 3, test\_size = .33, random\_state = 8)

# Instantiate the grid search model

grid\_search = GridSearchCV(estimator=rfc,

param\_grid=param\_grid,

scoring='accuracy',

cv=cv\_sets,

verbose=1)

# Fit the grid search to the data

grid\_search.fit(features\_train, labels\_train)

print("The best hyperparameters from Grid Search are:")

print(grid\_search.best\_params\_)

print("")

print("The mean accuracy of a model with these hyperparameters is:")

print(grid\_search.best\_score\_)

best\_rfc = grid\_search.best\_estimator\_

best\_rfc

### Model fit and performance

best\_rfc.fit(features\_train, labels\_train)

rfc\_pred = best\_rfc.predict(features\_test)

#### Training accuracy

# Training accuracy

print("The training accuracy is: ")

print(accuracy\_score(labels\_train, best\_rfc.predict(features\_train)))

#Classification report

print("Classification Report : ")

print(Classification Report (labels\_test, rfc\_pred))

![Table

Description automatically generated]()

#Confusion matrix

aux\_df = df[['Category', 'Category\_Code']].drop\_duplicates().sort\_values('Category\_Code')

conf\_matrix = confusion\_matrix(labels\_test, rfc\_pred)

plt.figure(figsize=(12.8,6))

sns.heatmap(conf\_matrix,

annot=True,

xticklabels=aux\_df['Category'].values,

yticklabels=aux\_df['Category'].values,

cmap="Blues")

plt.ylabel('Predicted')

plt.xlabel('Actual')

plt.title('Confusion matrix')

plt.show()

A picture containing chart

Description automatically generated

#### At this point, we could get the average time the model takes to get predictions. We want the algorithm to be fast since we are creating an app which will gather data from the internet and get the predicted categories. However, since the difference when predicting 10-20 observations will be very little, we won't take this into account.

import time

features\_time = features\_train

elapsed\_list = []

for i in range(0,10):

start = time.time()

predictions = best\_rfc.predict(features\_time)

end = time.time()

elapsed = end - start

elapsed\_list.append(elapsed)

mean\_time\_elapsed = np.mean(elapsed\_list)

base\_model = RandomForestClassifier(random\_state = 8)

base\_model.fit(features\_train, labels\_train)

accuracy\_score(labels\_test, base\_model.predict(features\_test))

best\_rfc.fit(features\_train, labels\_train)

accuracy\_score(labels\_test, best\_rfc.predict(features\_test))

d = {

'Model': 'Random Forest',

'Training Set Accuracy': accuracy\_score(labels\_train, best\_rfc.predict(features\_train)),

'Test Set Accuracy': accuracy\_score(labels\_test, rfc\_pred)

}

df\_models\_rfc = pd.DataFrame(d, index=[0])

df\_models\_rfc

Let's save the model and this dataset:

pathcolab='/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/Models/'

with open(pathcolab+'best\_rfc.pickle', 'wb') as output:

pickle.dump(best\_rfc, output)

with open(pathcolab+'df\_models\_rfc.pickle', 'wb') as output:

pickle.dump(df\_models\_rfc, output)

# Support Vector Machine

#Importing library

import pickle

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn import svm

from pprint import pprint

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

from sklearn.model\_selection import ShuffleSplit

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

First, we load the data:

Mounting Drive:

from google.colab import drive

drive.mount('/content/drive')

pathcolab = '/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/dataset.csv/Pickles/'

# Dataframe

path\_df = "df.pickle"

with open(pathcolab+path\_df, 'rb') as data:

df = pickle.load(data)

# features\_train

path\_features\_train = "features\_train.pickle"

with open(pathcolab+path\_features\_train, 'rb') as data:

features\_train = pickle.load(data)

# labels\_train

path\_labels\_train = "labels\_train.pickle"

with open(pathcolab+path\_labels\_train, 'rb') as data:

labels\_train = pickle.load(data)

# features\_test

path\_features\_test = "features\_test.pickle"

with open(pathcolab+path\_features\_test, 'rb') as data:

features\_test = pickle.load(data)

# labels\_test

path\_labels\_test = "labels\_test.pickle"

with open(pathcolab+path\_labels\_test, 'rb') as data:

labels\_test = pickle.load(data)

Let's check the dimension of our feature vectors:

print(features\_train.shape)

print(features\_test.shape)

## Cross-Validation for Hyperparameter tuning

First, we can see what hyperparameters the model has:

svc\_0 =svm.SVC(random\_state=8)

print('Parameters currently in use:\n')

print(svc\_0.get\_params())

We'll tune the following ones:

\* `C`: Penalty parameter C of the error term.

\* `kernel`: Specifies the kernel type to be used in the algorithm.

\* `gamma`: Kernel coefficient.

\* `degree`: Degree of the polynomial kernel function.

### Randomized Search Cross Validation

We first need to define the grid:

# C

C = [.0001, .001, .01]

# gamma

gamma = [.0001, .001, .01, .1, 1, 10, 100]

# degree

degree = [1, 2, 3, 4, 5]

# kernel

kernel = ['linear', 'rbf', 'poly']

# probability

probability = [True]

# Create the random grid

random\_grid = {'C': C,

'kernel': kernel,

'gamma': gamma,

'degree': degree,

'probability': probability

}

print(random\_grid)

Then, we'll perform the Random Search:

# First create the base model to tune

svc = svm.SVC(random\_state=8)

# Definition of the random search

random\_search = RandomizedSearchCV(estimator=svc,

param\_distributions=random\_grid,

n\_iter=50,

scoring='accuracy',

cv=3,

verbose=1,

random\_state=8)

# Fit the random search model

random\_search.fit(features\_train, labels\_train)

We can see the best hyperparameters resulting from the Random Search:

print("The best hyperparameters from Random Search are:")

print(random\_search.best\_params\_)

print("")

print("The mean accuracy of a model with these hyperparameters is:")

print(random\_search.best\_score\_)

After that, we can do a more exhaustive search centered in those values:

### Grid Search Cross Validation

# Create the parameter grid based on the results of random search

C = [.0001, .001, .01, .1]

degree = [3, 4, 5]

gamma = [1, 10, 100]

probability = [True]

param\_grid = [

{'C': C, 'kernel':['linear'], 'probability':probability},

{'C': C, 'kernel':['poly'], 'degree':degree, 'probability':probability},

{'C': C, 'kernel':['rbf'], 'gamma':gamma, 'probability':probability}

]

# Create a base model

svc = svm.SVC(random\_state=8)

# Manually create the splits in CV in order to be able to fix a random\_state (GridSearchCV doesn't have that argument)

cv\_sets = ShuffleSplit(n\_splits = 3, test\_size = .33, random\_state = 8)

# Instantiate the grid search model

grid\_search = GridSearchCV(estimator=svc,

param\_grid=param\_grid,

scoring='accuracy',

cv=cv\_sets,

verbose=1)

# Fit the grid search to the data

grid\_search.fit(features\_train, labels\_train)

The best hyperparameters turn out to be:

print("The best hyperparameters from Grid Search are:")

print(grid\_search.best\_params\_)

print("")

print("The mean accuracy of a model with these hyperparameters is:")

print(grid\_search.best\_score\_)

Let's save the model in `best\_svc`:

best\_svc = grid\_search.best\_estimator\_

best\_svc

We now know the best SVM model. Let's fit it and see how it performs:

## Model fit and performance

Now, we can fit the model to our training data:

best\_svc.fit(features\_train, labels\_train)

And get the predictions:

svc\_pred = best\_svc.predict(features\_test)

The conditional class probabilities can be obtained by typing:

`svc\_pred = best\_svc.predict\_proba(features\_test)`

For performance analysis, we will use the confusion matrix, the classification report and the accuracy on both training and test data:

#### Training accuracy

# Training accuracy

print("The training accuracy is: ")

print(accuracy\_score(labels\_train, best\_svc.predict(features\_train)))

#### Test accuracy

# Test accuracy

print("The test accuracy is: ")

print(accuracy\_score(labels\_test, svc\_pred))

#### Classification report

# Classification report

print("Classification report")

print(classification\_report(labels\_test,svc\_pred))

![A picture containing text, receipt, screenshot

Description automatically generated]()

#### Confusion matrix

aux\_df = df[['Category', 'Category\_Code']].drop\_duplicates().sort\_values('Category\_Code')

conf\_matrix = confusion\_matrix(labels\_test, svc\_pred)

plt.figure(figsize=(12.8,6))

sns.heatmap(conf\_matrix,

annot=True,

xticklabels=aux\_df['Category'].values,

yticklabels=aux\_df['Category'].values,

cmap="Blues")

plt.ylabel('Predicted')

plt.xlabel('Actual')

plt.title('Confusion matrix')

plt.show()

Chart

Description automatically generated

At this point, we could get the average time the model takes to get predictions. We want the algorithm to be fast since we are creating an app which will gather data from the internet and get the predicted categories. However, since the difference when predicting 10-20 observations will be very little, we won't take this into account.

However, the code below could do this task:

```python

features\_time = features\_train

elapsed\_list = []

for i in range(0,10):

start = time.time()

predictions = best\_lrc.predict(features\_time)

end = time.time()

elapsed = end - start

elapsed\_list.append(elapsed)

mean\_time\_elapsed = np.mean(elapsed\_list)

```

Let's see if the hyperparameter tuning process has returned a better model:

base\_model = svm.SVC(random\_state = 8)

base\_model.fit(features\_train, labels\_train)

accuracy\_score(labels\_test, base\_model.predict(features\_test))

best\_svc.fit(features\_train, labels\_train)

accuracy\_score(labels\_test, best\_svc.predict(features\_test))

We'll create a dataset with a model summary to compare models:

d = {

'Model': 'SVM',

'Training Set Accuracy': accuracy\_score(labels\_train, best\_svc.predict(features\_train)),

'Test Set Accuracy': accuracy\_score(labels\_test, svc\_pred)

}

df\_models\_svc = pd.DataFrame(d, index=[0])

df\_models\_svc

Let's save the model and this dataset:

pathcolab='/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data//Models/'

with open('best\_svc.pickle', 'wb') as output:

pickle.dump(best\_svc, output)

with open('df\_models\_svc.pickle', 'wb') as output:

pickle.dump(df\_models\_svc, output)

## Multinomial Naïve Bayes

# Importing libraries

import pickle

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

from pprint import pprint

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

from sklearn.model\_selection import ShuffleSplit

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

First, we load the data:

Mounting Drive:

from google.colab import drive

drive.mount('/content/drive')

pathcolab = '/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/dataset.csv/Pickles/'

# Dataframe

path\_df = "df.pickle"

with open(pathcolab+path\_df, 'rb') as data:

df = pickle.load(data)

# features\_train

path\_features\_train = "features\_train.pickle"

with open(pathcolab+path\_features\_train, 'rb') as data:

features\_train = pickle.load(data)

# labels\_train

path\_labels\_train = "labels\_train.pickle"

with open(pathcolab+path\_labels\_train, 'rb') as data:

labels\_train = pickle.load(data)

# features\_test

path\_features\_test = "features\_test.pickle"

with open(pathcolab+path\_features\_test, 'rb') as data:

features\_test = pickle.load(data)

# labels\_test

path\_labels\_test = "labels\_test.pickle"

with open(pathcolab+path\_labels\_test, 'rb') as data:

labels\_test = pickle.load(data)

##### Let's check the dimension of our feature vectors:

print(features\_train.shape)

print(features\_test.shape)

##### Cross-Validation for Hyperparameter tuning

mnbc = MultinomialNB()

mnbc

### Model fit and performance

mnbc.fit(features\_train, labels\_train)

mnbc\_pred = mnbc.predict(features\_test)

#### Training accuracy

# Training accuracy

print("The training accuracy is: ")

print(accuracy\_score(labels\_train, mnbc.predict(features\_train)))

# Test accuracy

print("The test accuracy is: ")

print(accuracy\_score(labels\_test, mnbc\_pred))

#Classification report

# Classification report

print("Classification report")

print(classification\_report(labels\_test,mnbc\_pred))

![Table

Description automatically generated]()

#Confusion matrix

aux\_df = df[['Category', 'Category\_Code']].drop\_duplicates().sort\_values('Category\_Code')

conf\_matrix = confusion\_matrix(labels\_test, mnbc\_pred)

plt.figure(figsize=(13,8))

sns.heatmap(conf\_matrix,

annot=True,

xticklabels=aux\_df['Category'].values,

yticklabels=aux\_df['Category'].values,

cmap="Blues")

plt.ylabel('Predicted')

plt.xlabel('Actual')

plt.title('Confusion matrix')

plt.show()

**Chart

Description automatically generated**

d = {

'Model': 'Multinomial Naïve Bayes',

'Training Set Accuracy': accuracy\_score(labels\_train, mnbc.predict(features\_train)),

'Test Set Accuracy': accuracy\_score(labels\_test, mnbc\_pred)

}

df\_models\_mnbc = pd.DataFrame(d, index=[0])

df\_models\_mnbc

##### Let's save the model and this dataset:

pathcolab='/content/drive/MyDrive/AI Projects/Feature Engineering/ICE 1/Assignemnt\_data/Models/'

with open(pathcolab+'best\_mnbc.pickle', 'wb') as output:

pickle.dump(mnbc, output)

with open(pathcolab+'df\_models\_mnbc.pickle', 'wb') as output:

pickle.dump(df\_models\_mnbc, output)