Project Title	**Twitter Financial News**
Tools	Visual Studio code / jupyter notebook
Domain	Finance Analyst
Project Difficulties level	intermediate

Dataset : Dataset is available in the given link. You can download it at your convenience.

Click here to download data set

About the Data

The Twitter Financial News dataset is an English-language dataset containing an annotated corpus of finance-related tweets. This dataset is used to classify finance-related tweets for their topic.

Featured Notebooks:

Ideas:

1. The data is a multi-label text classification problem with imbalanced data.
2. The links within the texts could be extracted.

3. EDA

The dataset holds 21,107 documents annotated with 20 labels:

```
"LABEL_0": "Analyst Update",

 "LABEL_1": "Fed | Central Banks",

 "LABEL_2": "Company | Product News",

 "LABEL_3": "Treasuries | Corporate Debt",

 "LABEL_4": "Dividend",

 "LABEL_5": "Earnings",

 "LABEL_6": "Energy | Oil",

 "LABEL_7": "Financials",

 "LABEL_8": "Currencies",

 "LABEL_9": "General News | Opinion",

 "LABEL_10": "Gold | Metals | Materials",

 "LABEL_11": "IPO",

 "LABEL_12": "Legal | Regulation",

 "LABEL_13": "M&A | Investments",

 "LABEL_14": "Macro",

 "LABEL_15": "Markets",
```

```
"LABEL_16": "Politics",

"LABEL_17": "Personnel Change",

"LABEL_18": "Stock Commentary",

"LABEL_19": "Stock Movement"
```

Example: You can get the basic idea how you can create a project from here

Creating a **Twitter Financial News Analysis** project involves building a pipeline that processes, analyzes, and models financial sentiments. Here's a step-by-step explanation tailored for your experience level.

Step 1: Problem Definition

- **Objective**: Analyze Twitter financial news and classify sentiments (e.g., `Positive`, `Neutral`, `Negative`) or financial impacts (e.g., `Bullish`, `Bearish`).
- **Data**:
  - `text`: Tweets or news text related to financial markets.
  - `label`: The sentiment or impact associated with each text (categorical values).

Step 2: Data Collection

Assume we have a dataset, or use the **Tweepy API** to collect financial tweets.

Code for Collecting Data:

python

code

```
import tweepy
import pandas as pd


# Twitter API credentials
```

```
api_key = 'your_api_key'
api_secret = 'your_api_secret'
access_token = 'your_access_token'
access_token_secret = 'your_access_token_secret'

# Authenticate
auth = tweepy.OAuth1UserHandler(api_key, api_secret,
access_token, access_token_secret)
api = tweepy.API(auth)

# Fetch tweets
query = 'stock market'  # Financial keyword
tweets = tweepy.Cursor(api.search_tweets, q=query, lang="en",
tweet_mode='extended').items(1000)

# Store tweets in DataFrame
data = {'text': [], 'label': []}  # Add labels manually or use
pre-annotated data
for tweet in tweets:
    data['text'].append(tweet.full_text)

df = pd.DataFrame(data)
df.to_csv('financial_tweets.csv', index=False)
```

Step 3: Data Cleaning

Clean the tweets to remove irrelevant information like URLs, hashtags, and mentions.

Code:

python

code

```
import re

def clean_text(text):
    text = re.sub(r'http\S+|www\S+|https\S+', '', text,
flags=re.MULTILINE)  # Remove URLs
    text = re.sub(r'\@\w+|\#', '', text)  # Remove mentions and
hashtags
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = re.sub(r'\d+', '', text)  # Remove numbers
    text = text.lower()  # Convert to lowercase
    return text

df['clean_text'] = df['text'].apply(clean_text)
print(df.head())
```

Step 4: Exploratory Data Analysis (EDA)

4.1 Sentiment Distribution

Visualize the distribution of sentiment labels.

Code:

python

code

```
import matplotlib.pyplot as plt
import seaborn as sns

# Sentiment distribution
sns.countplot(data=df, x='label', palette='viridis')
plt.title('Sentiment Distribution')
plt.xlabel('Sentiment')
plt.ylabel('Count')
plt.show()
```

4.2 Word Cloud

Visualize frequent words in tweets.

python

code

```
from wordcloud import WordCloud

# Generate Word Cloud
text = ' '.join(df['clean_text'])
wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text)
```

```
# Display Word Cloud
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Step 5: Feature Engineering

5.1 Text Vectorization

Use **TF-IDF** to convert text into numerical features.

Code:

python

code

```
from sklearn.feature_extraction.text import TfidfVectorizer

# TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=5000)  # Top 5000 words
X = vectorizer.fit_transform(df['clean_text']).toarray()
y = df['label']  # Target variable
```

5.2 Split Data

python

code

```
from sklearn.model_selection import train_test_split

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Step 6: Model Training

6.1 Use Logistic Regression

Logistic Regression is a good starting point for classification problems.

Code:

python

code

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report,
confusion_matrix

# Train the model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)
```

```
# Evaluate the model
print(classification_report(y_test, y_pred))
```

Step 7: Data Visualization

7.1 Confusion Matrix

python

code

```
import seaborn as sns
import matplotlib.pyplot as plt

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=model.classes_, yticklabels=model.classes_)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

Step 8: Advanced Modeling

Experiment with more sophisticated models like **BERT** or **LSTM**.

Using Hugging Face's BERT:

python

code

```
from transformers import BertTokenizer,
BertForSequenceClassification
from torch.utils.data import DataLoader, Dataset

# Tokenize text using BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
tokens = tokenizer(list(df['clean_text']), padding=True,
truncation=True, return_tensors="pt")

# Use BERT model for classification
model =
BertForSequenceClassification.from_pretrained('bert-base-uncase
d', num_labels=len(df['label'].unique()))
```

Step 9: Deploy the Model

Deploy using **Flask** or **Streamlit** for real-time predictions.

Streamlit Example:

python

code

```
import streamlit as st
```

```
st.title('Twitter Financial News Sentiment Analysis')
user_input = st.text_input("Enter a financial tweet:")
if st.button("Predict"):
    clean_input = clean_text(user_input)
    input_vector =
vectorizer.transform([clean_input]).toarray()
    prediction = model.predict(input_vector)
    st.write(f"Predicted Sentiment: {prediction[0]}")
```

Step 10: Statistical Insights

Analyze relationships between sentiment and financial trends using external data like stock prices.

Example: You can get the basic idea how you can create a project from here

Sample code with output

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from nltk.corpus import stopwords
from nltk import WordNetLemmatizer
from nltk import word_tokenize
import nltk
from wordcloud import WordCloud

import re
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import
pad_sequences
from tensorflow.keras.layers import Input, Dense, Embedding,
LSTM, Bidirectional, GRU, GlobalMaxPooling1D, Dropout,
SimpleRNN, Conv1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.losses import
```

```
SparseCategoricalCrossentropy, categorical_crossentropy

from sklearn.feature_extraction.text import CountVectorizer
```

1. Preprocessing

In [46]:

```
# nltk.download("stopwords")
# nltk.download("punkt")
# nltk.download("wordnet")
```

In [47]:

```
# get stopwords
stops = set(stopwords.words("english"))
```

In [48]:

```
df_train =
pd.read_csv("/kaggle/input/twitter-financial-news/train_data.cs
v")
df_test =
pd.read_csv("/kaggle/input/twitter-financial-news/valid_data.cs
v")
```

In [49]:

```
df_train.head()
```

Out[49]:

	text	label
0	Here are Thursday's biggest analyst calls: App...	0
1	Buy Las Vegas Sands as travel to Singapore bui...	0
2	Piper Sandler downgrades DocuSign to sell, cit...	0
3	Analysts react to Tesla's latest earnings, bre...	0

4	Netflix and its peers are set for a ‘return to...	0

In [50]:

```
train_labels = df_train["label"]
train_corpus = df_train["text"]

test_labels = df_test["label"]
test_corpus = df_test["text"]
```

In [51]:

```
# get correct mapping of ordinal encoded target variables

label_mapping = {"LABEL_0": "Analyst Update",

    "LABEL_1": "Fed | Central Banks",

    "LABEL_2": "Company | Product News",

    "LABEL_3": "Treasuries | Corporate Debt",

    "LABEL_4": "Dividend",
```

```
"LABEL_5": "Earnings",

"LABEL_6": "Energy | Oil",

"LABEL_7": "Financials",

"LABEL_8": "Currencies",

"LABEL_9": "General News | Opinion",

"LABEL_10": "Gold | Metals | Materials",

"LABEL_11": "IPO",

"LABEL_12": "Legal | Regulation",

"LABEL_13": "M&A | Investments",

"LABEL_14": "Macro",

"LABEL_15": "Markets",

"LABEL_16": "Politics",

"LABEL_17": "Personnel Change",
```

```
    "LABEL_18": "Stock Commentary",

    "LABEL_19": "Stock Movement"
}
```

In [52]:

```
label_mapping = {k:v for k,v in zip(range(20),
label_mapping.values())}
```

In [53]:

```
label_mapping
```

Out[53]:

```
{0: 'Analyst Update',
 1: 'Fed | Central Banks',
 2: 'Company | Product News',
 3: 'Treasuries | Corporate Debt',
 4: 'Dividend',
 5: 'Earnings',
 6: 'Energy | Oil',
 7: 'Financials',
```

```
 8: 'Currencies',
 9: 'General News | Opinion',
 10: 'Gold | Metals | Materials',
 11: 'IPO',
 12: 'Legal | Regulation',
 13: 'M&A | Investments',
 14: 'Macro',
 15: 'Markets',
 16: 'Politics',
 17: 'Personnel Change',
 18: 'Stock Commentary',
 19: 'Stock Movement'}
```

In [54]:

```
# get understanding of the text data

# number of the random articles
S = 5
inds = np.random.choice(train_corpus.index, 5)

for i in inds:
  print(train_corpus[i])
```

```
Klarna's valuation slumps to $6.7 billion with $800 million
```

```
raise  https://t.co/lrNWSfKGgF  https://t.co/G0tE1Sv2n9
WATCH: SAS said that a pilot strike could keep the company from
accessing long-term capital it needs for reorganization and
thus the airline could collapse  https://t.co/m08vgf3CSJ
https://t.co/7Y4YMZkXSc
SILVERSTONE MASTER UK Regulatory Announcement: FRN Variable
Rate Fix  https://t.co/a5eW1kIo89  https://t.co/K8UEJVjoVX
Consumer Cable and Internet Spending Plateaus Despite
Cross-category Surge in U.S. Household Expenses
https://t.co/AZaOTUJ3a6  https://t.co/eMdIL0xjSU
RBN Energy’s Refined Fuels Analytics Team Welcomes John Auers
https://t.co/mbbCkBR4fG  https://t.co/XA1isVTb3Y
```

In [55]:

```python
# there is a need to remove all hyperlinks, since they do not
contain any contextual text data

def remove_hyperlinks_and_punctuation(text):
  pattern = r'\bhttps?:\/\/\S+|[^\w\s]'
  new_text = re.sub(pattern, "", text)

  return new_text
```

In [56]:

```
train_corpus_cleaned = train_corpus.apply(lambda x:
remove_hyperlinks_and_punctuation(x))
test_corpus_cleaned = test_corpus.apply(lambda x:
remove_hyperlinks_and_punctuation(x))
```

In [57]:

```
for i in inds:
  print(train_corpus_cleaned[i])
```

```
Klarnas valuation slumps to 67 billion with 800 million raise
WATCH SAS said that a pilot strike could keep the company from
accessing longterm capital it needs for reorganization and thus
the airline could collapse
SILVERSTONE MASTER UK Regulatory Announcement FRN Variable Rate
Fix
Consumer Cable and Internet Spending Plateaus Despite
Crosscategory Surge in US Household Expenses
RBN Energys Refined Fuels Analytics Team Welcomes John Auers
```

2. Tokenize corpus

In [58]:

```
# use count vectorizer to feed in ANN without paying attention
to the sequence
# Could make sense, because classifcation of the source of the
news may not be dependent on the sequence of the words, but
rather on some certain important words which are unique for the
source

class LemmaTokenizer(object):
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, articles):
        return [self.wnl.lemmatize(t) for t in
word_tokenize(articles)]

MAX_VOCAB_SIZE_ANN = 18000

vectorizer = CountVectorizer(tokenizer=LemmaTokenizer(),
stop_words=list(stops), max_features=MAX_VOCAB_SIZE_ANN)
vectorizer.fit(train_corpus_cleaned)

train_data_ann = vectorizer.transform(train_corpus_cleaned)
test_data_ann = vectorizer.transform(test_corpus_cleaned)
```

```
/opt/conda/lib/python3.7/site-packages/sklearn/feature_extracti
```

```
on/text.py:517: UserWarning: The parameter 'token_pattern' will
not be used since 'tokenizer' is not None'
  "The parameter 'token_pattern' will not be used"
/opt/conda/lib/python3.7/site-packages/sklearn/feature_extracti
on/text.py:401: UserWarning: Your stop_words may be
inconsistent with your preprocessing. Tokenizing the stop words
generated tokens ["'d", "'ll", "'re", "'s", "'ve", 'could',
'doe', 'ha', 'might', 'must', "n't", 'need', 'sha', 'wa', 'wo',
'would'] not in stop_words.
  % sorted(inconsistent)
```

In [59]:

```
# create batch generator to feed sparse matrix into keras fit
method

def batch_generator(X, y, batch_size):
    number_of_batches = X.shape[0]/batch_size
    counter=0
    shuffle_index = np.arange(np.shape(y)[0])
    np.random.shuffle(shuffle_index)
    X =  X[shuffle_index, :]
    y =  y[shuffle_index]
    while 1:
        # each batch contains all the shuffled indices
```

```
        index_batch = shuffle_index[batch_size*counter :
batch_size*(counter+1)]
        X_batch = X[index_batch,:].todense()
        y_batch = y[index_batch]
        counter += 1
        yield(np.array(X_batch),y_batch)
        if (counter < number_of_batches):
            np.random.shuffle(shuffle_index)
            counter=0
```

In [60]:

```
# Apply lemmatization to corpus - remove maybe later in case of
poor performance
# used for sequence models
wnl = WordNetLemmatizer()

def lemmatize_sentence(sentence):
  tokens = [wnl.lemmatize(t) for t in word_tokenize(sentence)]
  return (" ").join(tokens)

train_corpus_cleaned = train_corpus_cleaned.apply(lambda x:
lemmatize_sentence(x))
test_corpus_cleaned = test_corpus_cleaned.apply(lambda x:
lemmatize_sentence(x))
```

In [61]:

```
for i in inds:
  print(train_corpus_cleaned[i])
```

```
Klarnas valuation slump to 67 billion with 800 million raise
WATCH SAS said that a pilot strike could keep the company from
accessing longterm capital it need for reorganization and thus
the airline could collapse
SILVERSTONE MASTER UK Regulatory Announcement FRN Variable Rate
Fix
Consumer Cable and Internet Spending Plateaus Despite
Crosscategory Surge in US Household Expenses
RBN Energys Refined Fuels Analytics Team Welcomes John Auers
```

In [62]:

```
# transform the corpus to get training and test data

MAX_VOCAB_SIZE = 30000

tokenizer = Tokenizer(num_words=MAX_VOCAB_SIZE,
                      oov_token="OOV",
```

```
                      lower=True)

tokenizer.fit_on_texts(train_corpus_cleaned)
train_data = tokenizer.texts_to_sequences(train_corpus_cleaned)
test_data = tokenizer.texts_to_sequences(test_corpus_cleaned)
```

In [63]:

```
# get max len of sentences for padding

maxlen1 = max(len(sent) for sent in train_data)
maxlen2 = max(len(sent) for sent in test_data)

T = max(maxlen1, maxlen2)
print(T)
```

```
64
```

In [64]:

```
word2idx = tokenizer.word_index

V = len(word2idx)
```

```
print("The corpus contains %s words!" % V)
```

```
The corpus contains 25558 words!
```

In [65]:

```
# pad sequences

train_data_padded = pad_sequences(train_data, maxlen=T)
test_data_padded = pad_sequences(test_data, maxlen=T)
```

In [66]:

```
print("Train tensor shape:", train_data_padded.shape)
print("Test tensor shape:", test_data_padded.shape)
```

```
Train tensor shape: (16990, 64)
Test tensor shape: (4117, 64)
```

In [67]:

```
# get class number

K = len(set(train_labels))
```

```
print(f"We have %s unique labels!" % K)
```

```
We have 20 unique labels!
```

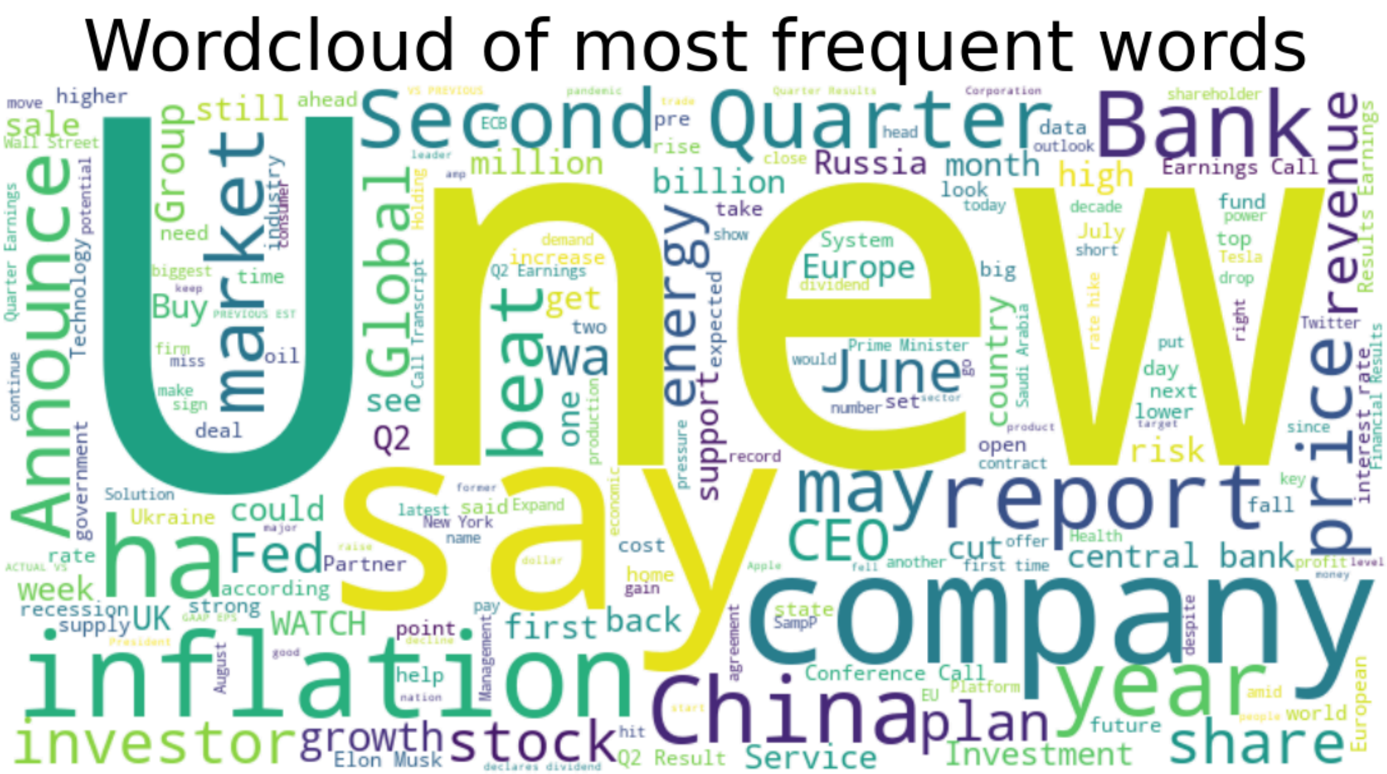
3. Data overview and visualization

In [68]:

```
# show word overview with wordcloud

# not using all words, but rather subsamples
random_subsample_word_wordcloud_size =
round(len(train_corpus_cleaned) * 0.33)
wordcloud_ids = np.random.choice(len(train_corpus_cleaned),
random_subsample_word_wordcloud_size)

concat_text = ""
for i in wordcloud_ids:
  concat_text += " " + train_corpus_cleaned[i]

wordcloud = WordCloud(background_color="white", width=800,
height=400, stopwords=stops).generate(concat_text)
```

In [69]:

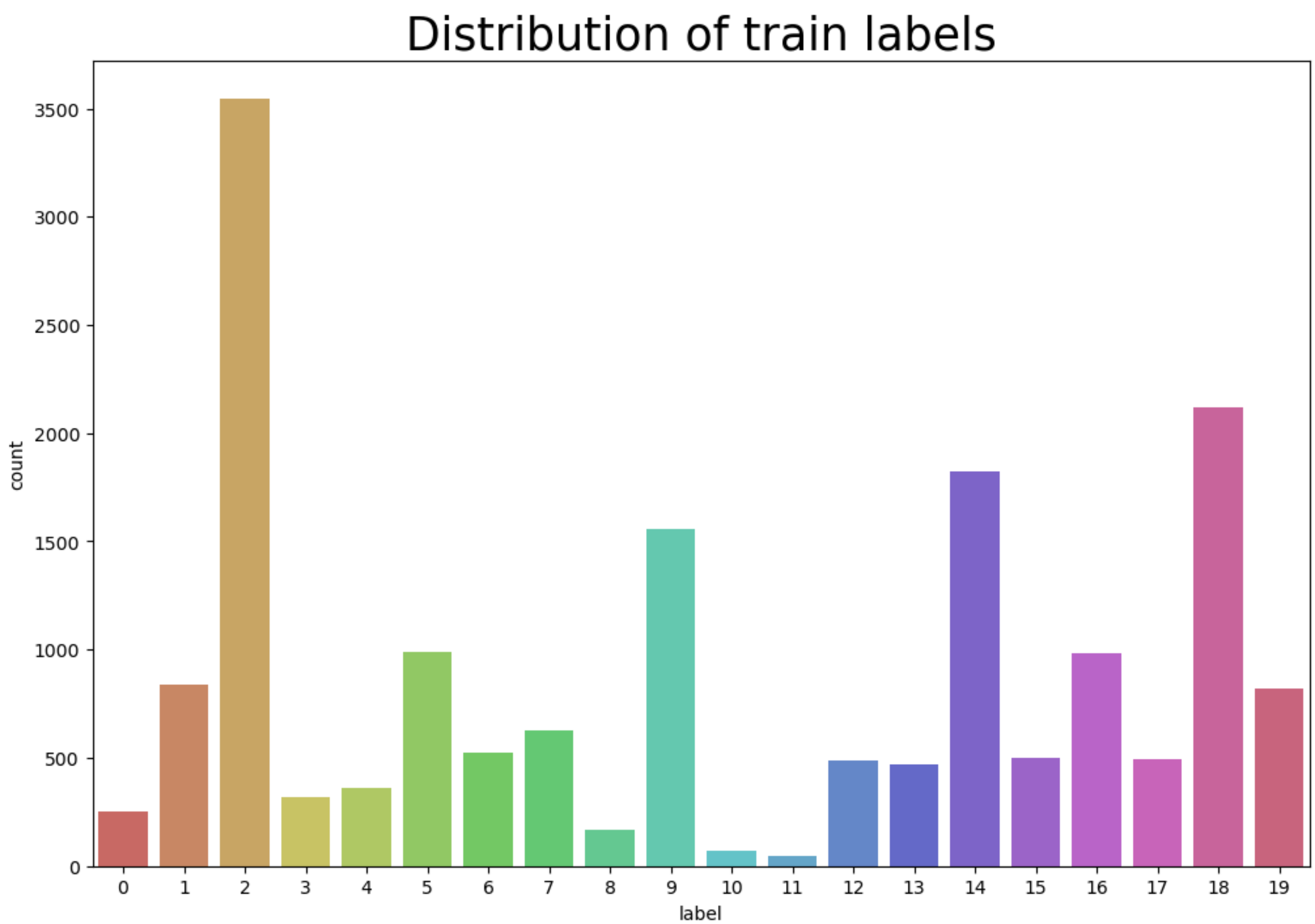
```python
fig, ax = plt.subplots(figsize=(14,10))
ax.imshow(wordcloud, interpolation="bilinear")
ax.axis("off")
plt.title("Wordcloud of most frequent words", fontsize=40)
plt.show()
```



In [70]:

```python
# distribution of train targets


fig, ax = plt.subplots(figsize=(12,8))
sns.countplot(x=train_labels.index, data=train_labels, ax=ax,
```

```
palette="hls")
plt.title("Distribution of train labels", fontsize=25)
plt.show()
print("Highly imbalanced data!!")
```

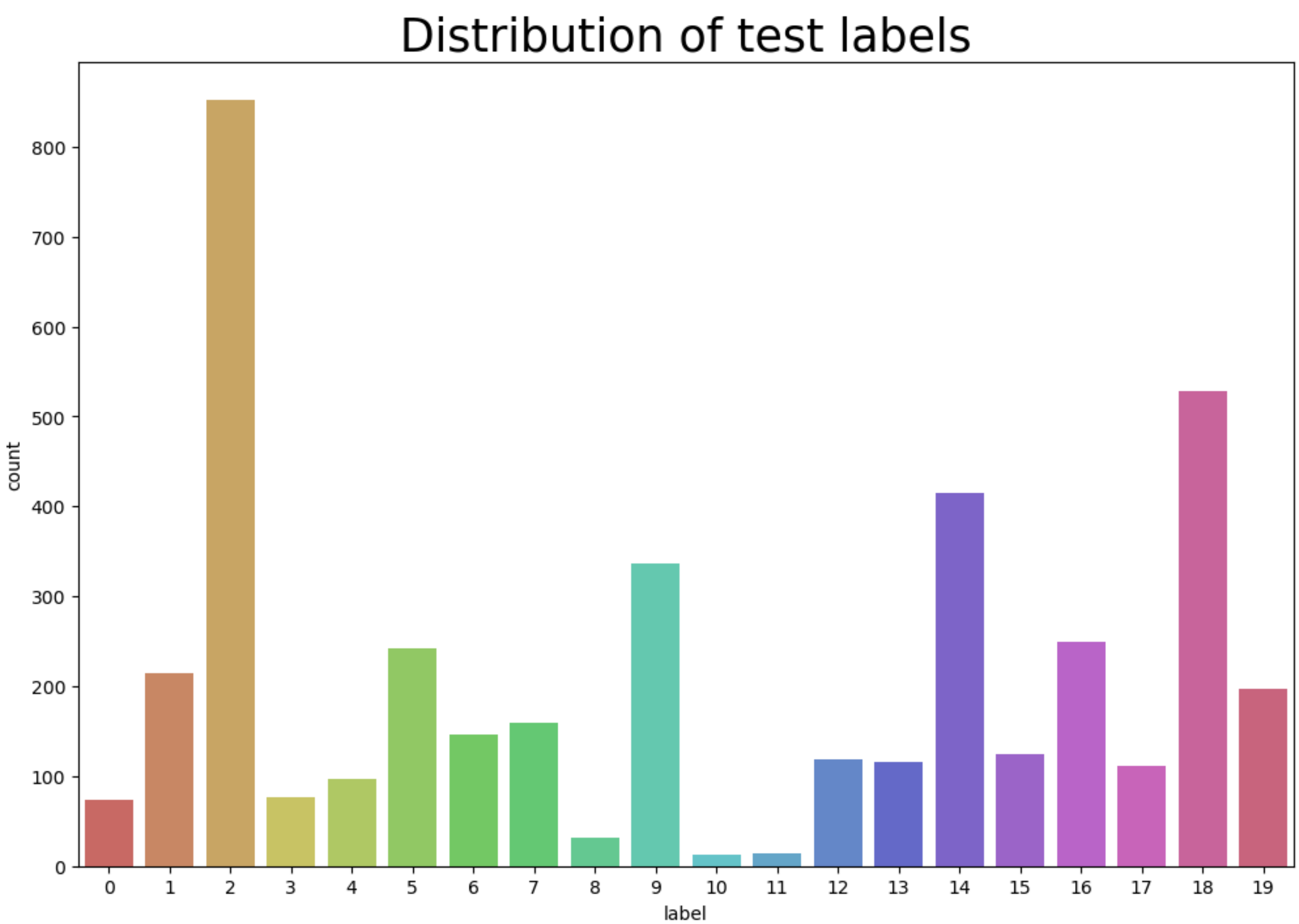



```
Highly imbalanced data!!
```

In [71]:

```
# distribution of train targets
```

```
fig, ax = plt.subplots(figsize=(12,8))
sns.countplot(x=test_labels.index, data=test_labels, ax=ax,
palette="hls")
plt.title("Distribution of test labels", fontsize=25)
plt.show()
print("Highly imbalanced data!!")
```




```
Highly imbalanced data!!
```

In [72]:

```
# highly imblanced data, so inverse class weights are computed for the loss function

class_weights = train_labels.value_counts(normalize=True).sort_index()
inverse_class_weights = class_weights.apply(lambda x: 1 / x)

inverse_class_weights
```

Out[72]:

```
0      66.627451
1      20.298686
2       4.792666
3      52.928349
4      47.325905
5      17.213779
6      32.423664
7      27.227564
8     102.349398
9      10.912010
10    246.231884
```

```
11    386.136364
12     34.887064
13     36.072187
14      9.324918
15     33.912176
16     17.248731
17     34.323232
18      8.021719
19     20.643985

Name: label, dtype: float64
```

4. Build the model

```
In [73]:
# Embedding dimension
D = 64
hidden_states1 = 12



def make_model_rnn():
  with tf.device("/GPU:0"):
    i = Input(shape=(T,))
    x = Embedding(V + 1, D)(i)
    x = SimpleRNN(hidden_states1, return_sequences=True)(x)
    x = GlobalMaxPooling1D()(x)
    x = Dense(64, activation="relu")(x)
```

```
    x = Dropout(0.5)(x)
    x = Dense(K, activation="softmax")(x)

    model = Model(i,x)
    model.compile(loss=SparseCategoricalCrossentropy(),
optimizer=Adam(learning_rate=0.001), metrics=["accuracy"])

    return model
```

In [74]:

```
def make_model_cnn():
  with tf.device("/GPU:0"):
    i = Input(shape=(T,))
    x = Embedding(V + 1, D)(i)
    x = Conv1D(32,3)(x)
    x = GlobalMaxPooling1D()(x)
    x = Dense(64, activation="relu")(x)
    x = Dropout(0.3)(x)
    x = Dense(K, activation="softmax")(x)

    model = Model(i,x)
    model.compile(loss=SparseCategoricalCrossentropy(),
optimizer=SGD(learning_rate=0.0001, momentum=0.9),
metrics=["accuracy"])
```

```
    return model
```

In [75]:

```
def make_model_ann():
  with tf.device("/GPU:0"):
    i = Input(shape=(MAX_VOCAB_SIZE_ANN,))
    x = Dense(1028, activation="relu")(i)
    x = Dropout(0.2)(x)
    x = Dense(512, activation="relu")(x)
    x = Dropout(0.2)(x)
    x = Dense(256, activation="relu")(x)
    x = Dense(K, activation="softmax")(x)

    model = Model(i,x)
    model.compile(loss=SparseCategoricalCrossentropy(),
optimizer=SGD(learning_rate=0.0001, momentum=0.9),
metrics=["accuracy"])
    # maybe focal loss

    return model
```

5 Fit the model

In [76]:

```
model = make_model_cnn()
print(model.summary())
```

```
Model: "model_1"
________________________________________________________________________
__
 Layer (type)                Output Shape              Param #
========================================================================
==
 input_2 (InputLayer)        [(None, 64)]              0

 embedding_1 (Embedding)     (None, 64, 64)            1635776

 conv1d_1 (Conv1D)           (None, 62, 32)            6176

 global_max_pooling1d_1 (Glo  (None, 32)               0
 balMaxPooling1D)

 dense_2 (Dense)             (None, 64)                2112

 dropout_1 (Dropout)         (None, 64)                0

 dense_3 (Dense)             (None, 20)                1300
```

```
=================================================================
==
Total params: 1,645,364
Trainable params: 1,645,364
Non-trainable params: 0
-----------------------------------------------------------------
--
None
```

In [77]:

```
r1 = model.fit(train_data_padded, train_labels, epochs=100,
batch_size=32, validation_data=(test_data_padded, test_labels),
class_weight=dict(inverse_class_weights))
```

```
Epoch 1/100
531/531 [==============================] - 11s 20ms/step -
loss: 59.9142 - accuracy: 0.0312 - val_loss: 2.9933 -
val_accuracy: 0.0554
Epoch 2/100
531/531 [==============================] - 4s 7ms/step - loss:
59.7056 - accuracy: 0.0884 - val_loss: 2.9779 - val_accuracy:
0.1603
```

```
Epoch 3/100
531/531 [==============================] - 4s 7ms/step - loss:
59.3280 - accuracy: 0.0979 - val_loss: 2.9658 - val_accuracy:
0.1438
Epoch 4/100
531/531 [==============================] - 3s 6ms/step - loss:
58.3765 - accuracy: 0.1104 - val_loss: 2.9028 - val_accuracy:
0.1219
Epoch 5/100
531/531 [==============================] - 3s 5ms/step - loss:
56.0483 - accuracy: 0.1399 - val_loss: 2.7866 - val_accuracy:
0.1231
Epoch 6/100
531/531 [==============================] - 3s 5ms/step - loss:
53.0405 - accuracy: 0.1491 - val_loss: 2.6679 - val_accuracy:
0.1681
Epoch 7/100
531/531 [==============================] - 3s 5ms/step - loss:
49.8193 - accuracy: 0.1674 - val_loss: 2.5738 - val_accuracy:
0.1953
Epoch 8/100
531/531 [==============================] - 4s 7ms/step - loss:
46.3996 - accuracy: 0.1842 - val_loss: 2.4852 - val_accuracy:
0.1870
Epoch 9/100
```

```
531/531 [==============================] - 3s 5ms/step - loss:
43.9879 - accuracy: 0.1951 - val_loss: 2.4221 - val_accuracy:
0.2154
Epoch 10/100
531/531 [==============================] - 3s 5ms/step - loss:
42.0127 - accuracy: 0.2095 - val_loss: 2.3389 - val_accuracy:
0.2351
Epoch 11/100
531/531 [==============================] - 3s 5ms/step - loss:
40.6706 - accuracy: 0.2182 - val_loss: 2.2905 - val_accuracy:
0.2577
Epoch 12/100
531/531 [==============================] - 3s 5ms/step - loss:
38.9891 - accuracy: 0.2344 - val_loss: 2.2401 - val_accuracy:
0.2723
Epoch 13/100
531/531 [==============================] - 3s 5ms/step - loss:
37.4722 - accuracy: 0.2553 - val_loss: 2.1870 - val_accuracy:
0.2781
Epoch 14/100
531/531 [==============================] - 3s 6ms/step - loss:
35.8749 - accuracy: 0.2728 - val_loss: 2.0974 - val_accuracy:
0.3082
Epoch 15/100
531/531 [==============================] - 3s 5ms/step - loss:
```

```
33.9444 - accuracy: 0.3010 - val_loss: 2.0349 - val_accuracy:
0.3442
Epoch 16/100
531/531 [==============================] - 3s 5ms/step - loss:
31.9808 - accuracy: 0.3286 - val_loss: 1.9389 - val_accuracy:
0.3777
Epoch 17/100
531/531 [==============================] - 3s 5ms/step - loss:
29.8246 - accuracy: 0.3577 - val_loss: 1.8335 - val_accuracy:
0.4015
Epoch 18/100
531/531 [==============================] - 3s 6ms/step - loss:
27.7647 - accuracy: 0.3832 - val_loss: 1.7307 - val_accuracy:
0.4282
Epoch 19/100
531/531 [==============================] - 3s 6ms/step - loss:
25.5822 - accuracy: 0.4215 - val_loss: 1.6495 - val_accuracy:
0.4608
Epoch 20/100
531/531 [==============================] - 3s 5ms/step - loss:
23.6537 - accuracy: 0.4559 - val_loss: 1.5950 - val_accuracy:
0.4892
Epoch 21/100
531/531 [==============================] - 3s 5ms/step - loss:
21.8214 - accuracy: 0.4775 - val_loss: 1.5225 - val_accuracy:
```

```
0.5179
Epoch 22/100
531/531 [==============================] - 3s 6ms/step - loss:
20.4309 - accuracy: 0.5091 - val_loss: 1.4603 - val_accuracy:
0.5239
Epoch 23/100
531/531 [==============================] - 3s 5ms/step - loss:
18.7395 - accuracy: 0.5311 - val_loss: 1.4067 - val_accuracy:
0.5397
Epoch 24/100
531/531 [==============================] - 3s 5ms/step - loss:
17.5805 - accuracy: 0.5568 - val_loss: 1.3477 - val_accuracy:
0.5669
Epoch 25/100
531/531 [==============================] - 3s 5ms/step - loss:
16.2075 - accuracy: 0.5777 - val_loss: 1.2764 - val_accuracy:
0.6036
Epoch 26/100
531/531 [==============================] - 3s 5ms/step - loss:
15.2806 - accuracy: 0.6019 - val_loss: 1.2530 - val_accuracy:
0.6148
Epoch 27/100
531/531 [==============================] - 2s 5ms/step - loss:
14.0660 - accuracy: 0.6257 - val_loss: 1.1998 - val_accuracy:
0.6369
```

```
Epoch 28/100
531/531 [==============================] - 3s 5ms/step - loss:
13.0900 - accuracy: 0.6486 - val_loss: 1.1602 - val_accuracy:
0.6476
Epoch 29/100
531/531 [==============================] - 3s 5ms/step - loss:
12.3568 - accuracy: 0.6676 - val_loss: 1.1411 - val_accuracy:
0.6621
Epoch 30/100
531/531 [==============================] - 3s 5ms/step - loss:
11.3669 - accuracy: 0.6905 - val_loss: 1.1007 - val_accuracy:
0.6680
Epoch 31/100
531/531 [==============================] - 3s 6ms/step - loss:
10.6286 - accuracy: 0.7146 - val_loss: 1.0702 - val_accuracy:
0.6779
Epoch 32/100
531/531 [==============================] - 3s 5ms/step - loss:
9.8552 - accuracy: 0.7281 - val_loss: 1.0374 - val_accuracy:
0.6944
Epoch 33/100
531/531 [==============================] - 3s 5ms/step - loss:
9.3955 - accuracy: 0.7398 - val_loss: 1.0416 - val_accuracy:
0.6906
Epoch 34/100
```

```
531/531 [==============================] - 3s 5ms/step - loss:
8.8528 - accuracy: 0.7568 - val_loss: 1.0104 - val_accuracy:
0.7066
Epoch 35/100
531/531 [==============================] - 3s 5ms/step - loss:
8.2548 - accuracy: 0.7676 - val_loss: 0.9962 - val_accuracy:
0.7095
Epoch 36/100
531/531 [==============================] - 3s 5ms/step - loss:
7.7419 - accuracy: 0.7846 - val_loss: 0.9886 - val_accuracy:
0.7127
Epoch 37/100
531/531 [==============================] - 2s 5ms/step - loss:
7.3814 - accuracy: 0.7893 - val_loss: 0.9607 - val_accuracy:
0.7233
Epoch 38/100
531/531 [==============================] - 3s 5ms/step - loss:
6.9352 - accuracy: 0.8069 - val_loss: 0.9699 - val_accuracy:
0.7280
Epoch 39/100
531/531 [==============================] - 3s 5ms/step - loss:
6.3897 - accuracy: 0.8205 - val_loss: 0.9449 - val_accuracy:
0.7323
Epoch 40/100
531/531 [==============================] - 3s 5ms/step - loss:
```

```
6.0012 - accuracy: 0.8366 - val_loss: 0.9583 - val_accuracy:
0.7294
Epoch 41/100
531/531 [==============================] - 2s 5ms/step - loss:
5.6445 - accuracy: 0.8424 - val_loss: 0.9268 - val_accuracy:
0.7377
Epoch 42/100
531/531 [==============================] - 3s 5ms/step - loss:
5.3952 - accuracy: 0.8474 - val_loss: 0.9267 - val_accuracy:
0.7457
Epoch 43/100
531/531 [==============================] - 3s 7ms/step - loss:
5.2183 - accuracy: 0.8563 - val_loss: 0.9231 - val_accuracy:
0.7433
Epoch 44/100
531/531 [==============================] - 3s 5ms/step - loss:
4.7608 - accuracy: 0.8657 - val_loss: 0.9196 - val_accuracy:
0.7491
Epoch 45/100
531/531 [==============================] - 3s 5ms/step - loss:
4.4999 - accuracy: 0.8742 - val_loss: 0.9278 - val_accuracy:
0.7508
Epoch 46/100
531/531 [==============================] - 2s 5ms/step - loss:
4.3075 - accuracy: 0.8802 - val_loss: 0.9163 - val_accuracy:
```

```
0.7522
Epoch 47/100
531/531 [==============================] - 3s 5ms/step - loss:
4.3098 - accuracy: 0.8828 - val_loss: 0.9159 - val_accuracy:
0.7583
Epoch 48/100
531/531 [==============================] - 3s 5ms/step - loss:
3.9671 - accuracy: 0.8904 - val_loss: 0.9177 - val_accuracy:
0.7549
Epoch 49/100
531/531 [==============================] - 3s 5ms/step - loss:
3.7508 - accuracy: 0.8999 - val_loss: 0.9170 - val_accuracy:
0.7578
Epoch 50/100
531/531 [==============================] - 3s 5ms/step - loss:
3.4167 - accuracy: 0.9050 - val_loss: 0.9120 - val_accuracy:
0.7654
Epoch 51/100
531/531 [==============================] - 3s 5ms/step - loss:
3.3205 - accuracy: 0.9131 - val_loss: 0.9148 - val_accuracy:
0.7656
Epoch 52/100
531/531 [==============================] - 2s 5ms/step - loss:
3.2107 - accuracy: 0.9130 - val_loss: 0.9166 - val_accuracy:
0.7675
```

```
Epoch 53/100
531/531 [==============================] - 3s 5ms/step - loss:
3.0907 - accuracy: 0.9195 - val_loss: 0.9169 - val_accuracy:
0.7700
Epoch 54/100
531/531 [==============================] - 2s 5ms/step - loss:
2.8278 - accuracy: 0.9243 - val_loss: 0.9265 - val_accuracy:
0.7685
Epoch 55/100
531/531 [==============================] - 3s 6ms/step - loss:
2.7340 - accuracy: 0.9261 - val_loss: 0.9192 - val_accuracy:
0.7714
Epoch 56/100
531/531 [==============================] - 2s 5ms/step - loss:
2.6732 - accuracy: 0.9307 - val_loss: 0.9404 - val_accuracy:
0.7724
Epoch 57/100
531/531 [==============================] - 2s 5ms/step - loss:
2.5135 - accuracy: 0.9361 - val_loss: 0.9273 - val_accuracy:
0.7727
Epoch 58/100
531/531 [==============================] - 3s 5ms/step - loss:
2.3983 - accuracy: 0.9387 - val_loss: 0.9354 - val_accuracy:
0.7722
Epoch 59/100
```

```
531/531 [==============================] - 3s 5ms/step - loss:
2.2742 - accuracy: 0.9404 - val_loss: 0.9333 - val_accuracy:
0.7758
Epoch 60/100
531/531 [==============================] - 3s 5ms/step - loss:
2.1182 - accuracy: 0.9481 - val_loss: 0.9434 - val_accuracy:
0.7795
Epoch 61/100
531/531 [==============================] - 3s 5ms/step - loss:
2.0637 - accuracy: 0.9483 - val_loss: 0.9443 - val_accuracy:
0.7787
Epoch 62/100
531/531 [==============================] - 3s 5ms/step - loss:
1.9177 - accuracy: 0.9512 - val_loss: 0.9614 - val_accuracy:
0.7780
Epoch 63/100
531/531 [==============================] - 2s 5ms/step - loss:
1.9059 - accuracy: 0.9514 - val_loss: 0.9559 - val_accuracy:
0.7799
Epoch 64/100
531/531 [==============================] - 3s 6ms/step - loss:
1.7550 - accuracy: 0.9569 - val_loss: 0.9626 - val_accuracy:
0.7833
Epoch 65/100
531/531 [==============================] - 3s 6ms/step - loss:
```

```
1.7220 - accuracy: 0.9592 - val_loss: 0.9832 - val_accuracy:
0.7846
Epoch 66/100
531/531 [==============================] - 3s 5ms/step - loss:
1.7179 - accuracy: 0.9596 - val_loss: 0.9819 - val_accuracy:
0.7833
Epoch 67/100
531/531 [==============================] - 3s 6ms/step - loss:
1.5851 - accuracy: 0.9616 - val_loss: 0.9849 - val_accuracy:
0.7833
Epoch 68/100
531/531 [==============================] - 3s 5ms/step - loss:
1.4692 - accuracy: 0.9647 - val_loss: 0.9851 - val_accuracy:
0.7829
Epoch 69/100
531/531 [==============================] - 2s 5ms/step - loss:
1.5336 - accuracy: 0.9640 - val_loss: 0.9978 - val_accuracy:
0.7841
Epoch 70/100
531/531 [==============================] - 2s 5ms/step - loss:
1.7107 - accuracy: 0.9606 - val_loss: 0.9974 - val_accuracy:
0.7831
Epoch 71/100
531/531 [==============================] - 3s 5ms/step - loss:
1.5381 - accuracy: 0.9666 - val_loss: 1.0488 - val_accuracy:
```

```
0.7785
Epoch 72/100
531/531 [==============================] - 3s 5ms/step - loss:
1.4126 - accuracy: 0.9662 - val_loss: 1.0092 - val_accuracy:
0.7858
Epoch 73/100
531/531 [==============================] - 3s 5ms/step - loss:
1.3388 - accuracy: 0.9694 - val_loss: 1.0167 - val_accuracy:
0.7848
Epoch 74/100
531/531 [==============================] - 3s 5ms/step - loss:
1.2163 - accuracy: 0.9700 - val_loss: 1.0332 - val_accuracy:
0.7848
Epoch 75/100
531/531 [==============================] - 2s 5ms/step - loss:
1.1791 - accuracy: 0.9726 - val_loss: 1.0386 - val_accuracy:
0.7863
Epoch 76/100
531/531 [==============================] - 3s 5ms/step - loss:
1.2157 - accuracy: 0.9727 - val_loss: 1.0261 - val_accuracy:
0.7875
Epoch 77/100
531/531 [==============================] - 3s 5ms/step - loss:
1.2254 - accuracy: 0.9736 - val_loss: 1.0459 - val_accuracy:
0.7858
```

```
Epoch 78/100
531/531 [==============================] - 3s 5ms/step - loss:
1.2307 - accuracy: 0.9706 - val_loss: 1.0429 - val_accuracy:
0.7858
Epoch 79/100
531/531 [==============================] - 3s 6ms/step - loss:
1.2113 - accuracy: 0.9738 - val_loss: 1.0581 - val_accuracy:
0.7863
Epoch 80/100
531/531 [==============================] - 3s 5ms/step - loss:
1.0248 - accuracy: 0.9775 - val_loss: 1.0508 - val_accuracy:
0.7872
Epoch 81/100
531/531 [==============================] - 3s 5ms/step - loss:
1.0278 - accuracy: 0.9756 - val_loss: 1.0654 - val_accuracy:
0.7884
Epoch 82/100
531/531 [==============================] - 2s 5ms/step - loss:
0.9587 - accuracy: 0.9783 - val_loss: 1.0769 - val_accuracy:
0.7901
Epoch 83/100
531/531 [==============================] - 2s 5ms/step - loss:
0.9815 - accuracy: 0.9786 - val_loss: 1.0817 - val_accuracy:
0.7867
Epoch 84/100
```

```
531/531 [==============================] - 3s 5ms/step - loss:
0.9486 - accuracy: 0.9788 - val_loss: 1.0876 - val_accuracy:
0.7884
Epoch 85/100
531/531 [==============================] - 3s 5ms/step - loss:
0.8395 - accuracy: 0.9823 - val_loss: 1.0883 - val_accuracy:
0.7880
Epoch 86/100
531/531 [==============================] - 3s 5ms/step - loss:
0.8297 - accuracy: 0.9818 - val_loss: 1.1006 - val_accuracy:
0.7853
Epoch 87/100
531/531 [==============================] - 2s 5ms/step - loss:
0.8609 - accuracy: 0.9808 - val_loss: 1.0987 - val_accuracy:
0.7860
Epoch 88/100
531/531 [==============================] - 3s 5ms/step - loss:
0.8015 - accuracy: 0.9838 - val_loss: 1.1071 - val_accuracy:
0.7875
Epoch 89/100
531/531 [==============================] - 3s 5ms/step - loss:
0.7211 - accuracy: 0.9834 - val_loss: 1.1173 - val_accuracy:
0.7892
Epoch 90/100
531/531 [==============================] - 2s 5ms/step - loss:
```

```
0.8440 - accuracy: 0.9818 - val_loss: 1.1224 - val_accuracy:
0.7892
Epoch 91/100
531/531 [==============================] - 3s 6ms/step - loss:
0.7433 - accuracy: 0.9836 - val_loss: 1.1264 - val_accuracy:
0.7870
Epoch 92/100
531/531 [==============================] - 2s 5ms/step - loss:
0.7630 - accuracy: 0.9842 - val_loss: 1.1179 - val_accuracy:
0.7880
Epoch 93/100
531/531 [==============================] - 3s 5ms/step - loss:
0.6864 - accuracy: 0.9861 - val_loss: 1.1443 - val_accuracy:
0.7875
Epoch 94/100
531/531 [==============================] - 2s 5ms/step - loss:
0.7292 - accuracy: 0.9845 - val_loss: 1.1516 - val_accuracy:
0.7865
Epoch 95/100
531/531 [==============================] - 3s 5ms/step - loss:
0.6997 - accuracy: 0.9850 - val_loss: 1.1412 - val_accuracy:
0.7870
Epoch 96/100
531/531 [==============================] - 2s 5ms/step - loss:
0.6978 - accuracy: 0.9852 - val_loss: 1.1373 - val_accuracy:
```

```
0.7889
Epoch 97/100
531/531 [==============================] - 3s 5ms/step - loss:
0.6556 - accuracy: 0.9852 - val_loss: 1.1587 - val_accuracy:
0.7858
Epoch 98/100
531/531 [==============================] - 2s 5ms/step - loss:
0.6145 - accuracy: 0.9880 - val_loss: 1.1610 - val_accuracy:
0.7894
Epoch 99/100
531/531 [==============================] - 2s 5ms/step - loss:
0.6179 - accuracy: 0.9866 - val_loss: 1.1715 - val_accuracy:
0.7860
Epoch 100/100
531/531 [==============================] - 3s 5ms/step - loss:
0.6324 - accuracy: 0.9862 - val_loss: 1.1672 - val_accuracy:
0.7870
```

In [78]:

```
# train also a simple ANN
model2 = make_model_ann()
print(model2.summary())
```

```
Model: "model_2"

-----------------------------------------------------------------
--
 Layer (type)                Output Shape              Param #
=================================================================
==
 input_3 (InputLayer)        [(None, 18000)]           0


 dense_4 (Dense)             (None, 1028)              18505028


 dropout_2 (Dropout)         (None, 1028)              0


 dense_5 (Dense)             (None, 512)               526848


 dropout_3 (Dropout)         (None, 512)               0


 dense_6 (Dense)             (None, 256)               131328


 dense_7 (Dense)             (None, 20)                5140


=================================================================
==
Total params: 19,168,344
Trainable params: 19,168,344
Non-trainable params: 0
```

```
-----------------------------------------------------------------
--
None
```

In [79]:

```
r2 = model2.fit(train_data_ann.toarray(), train_labels,
epochs=100, batch_size=32,
validation_data=(test_data_ann.toarray(), test_labels),
class_weight=dict(inverse_class_weights), shuffle=True)
```

```
Epoch 1/100
531/531 [==============================] - 7s 11ms/step - loss:
59.3496 - accuracy: 0.1099 - val_loss: 2.9208 - val_accuracy:
0.2652
Epoch 2/100
531/531 [==============================] - 5s 8ms/step - loss:
53.0952 - accuracy: 0.2433 - val_loss: 2.4742 - val_accuracy:
0.3384
Epoch 3/100
531/531 [==============================] - 4s 8ms/step - loss:
33.7288 - accuracy: 0.4246 - val_loss: 1.6334 - val_accuracy:
0.5373
Epoch 4/100
```

```
531/531 [==============================] - 4s 8ms/step - loss:
19.9803 - accuracy: 0.5876 - val_loss: 1.2084 - val_accuracy:
0.6738
Epoch 5/100
531/531 [==============================] - 4s 8ms/step - loss:
13.7646 - accuracy: 0.6871 - val_loss: 1.0191 - val_accuracy:
0.7037
Epoch 6/100
531/531 [==============================] - 4s 8ms/step - loss:
10.4509 - accuracy: 0.7469 - val_loss: 0.8996 - val_accuracy:
0.7345
Epoch 7/100
531/531 [==============================] - 5s 9ms/step - loss:
8.1547 - accuracy: 0.7928 - val_loss: 0.8309 - val_accuracy:
0.7561
Epoch 8/100
531/531 [==============================] - 4s 8ms/step - loss:
6.6510 - accuracy: 0.8230 - val_loss: 0.7322 - val_accuracy:
0.7802
Epoch 9/100
531/531 [==============================] - 4s 8ms/step - loss:
5.4875 - accuracy: 0.8501 - val_loss: 0.6871 - val_accuracy:
0.7928
Epoch 10/100
531/531 [==============================] - 4s 8ms/step - loss:
```

```
4.5912 - accuracy: 0.8733 - val_loss: 0.6747 - val_accuracy:
0.7962
Epoch 11/100
531/531 [==============================] - 4s 8ms/step - loss:
3.8537 - accuracy: 0.8879 - val_loss: 0.6750 - val_accuracy:
0.8006
Epoch 12/100
531/531 [==============================] - 5s 9ms/step - loss:
3.2754 - accuracy: 0.9083 - val_loss: 0.6381 - val_accuracy:
0.8115
Epoch 13/100
531/531 [==============================] - 4s 8ms/step - loss:
2.8264 - accuracy: 0.9181 - val_loss: 0.6324 - val_accuracy:
0.8190
Epoch 14/100
531/531 [==============================] - 5s 9ms/step - loss:
2.4749 - accuracy: 0.9306 - val_loss: 0.6374 - val_accuracy:
0.8220
Epoch 15/100
531/531 [==============================] - 5s 9ms/step - loss:
2.1491 - accuracy: 0.9381 - val_loss: 0.6302 - val_accuracy:
0.8229
Epoch 16/100
531/531 [==============================] - 4s 8ms/step - loss:
1.9063 - accuracy: 0.9468 - val_loss: 0.6319 - val_accuracy:
```

```
0.8278
Epoch 17/100
531/531 [==============================] - 4s 8ms/step - loss:
1.6878 - accuracy: 0.9530 - val_loss: 0.6385 - val_accuracy:
0.8283
Epoch 18/100
531/531 [==============================] - 5s 9ms/step - loss:
1.5010 - accuracy: 0.9592 - val_loss: 0.6416 - val_accuracy:
0.8297
Epoch 19/100
531/531 [==============================] - 4s 8ms/step - loss:
1.3219 - accuracy: 0.9650 - val_loss: 0.6553 - val_accuracy:
0.8285
Epoch 20/100
531/531 [==============================] - 4s 8ms/step - loss:
1.1601 - accuracy: 0.9689 - val_loss: 0.6488 - val_accuracy:
0.8317
Epoch 21/100
531/531 [==============================] - 4s 8ms/step - loss:
1.0776 - accuracy: 0.9712 - val_loss: 0.6625 - val_accuracy:
0.8271
Epoch 22/100
531/531 [==============================] - 5s 9ms/step - loss:
0.9872 - accuracy: 0.9752 - val_loss: 0.6662 - val_accuracy:
0.8302
```

```
Epoch 23/100
531/531 [==============================] - 4s 8ms/step - loss:
0.8918 - accuracy: 0.9770 - val_loss: 0.6763 - val_accuracy:
0.8288
Epoch 24/100
531/531 [==============================] - 5s 8ms/step - loss:
0.8101 - accuracy: 0.9812 - val_loss: 0.6719 - val_accuracy:
0.8314
Epoch 25/100
531/531 [==============================] - 4s 8ms/step - loss:
0.7049 - accuracy: 0.9839 - val_loss: 0.6813 - val_accuracy:
0.8295
Epoch 26/100
531/531 [==============================] - 4s 8ms/step - loss:
0.6768 - accuracy: 0.9841 - val_loss: 0.6940 - val_accuracy:
0.8297
Epoch 27/100
531/531 [==============================] - 4s 8ms/step - loss:
0.6347 - accuracy: 0.9855 - val_loss: 0.7071 - val_accuracy:
0.8297
Epoch 28/100
531/531 [==============================] - 4s 8ms/step - loss:
0.5714 - accuracy: 0.9881 - val_loss: 0.7053 - val_accuracy:
0.8297
Epoch 29/100
```

```
531/531 [==============================] - 5s 9ms/step - loss:
0.5252 - accuracy: 0.9892 - val_loss: 0.7156 - val_accuracy:
0.8297
Epoch 30/100
531/531 [==============================] - 4s 8ms/step - loss:
0.5076 - accuracy: 0.9895 - val_loss: 0.7211 - val_accuracy:
0.8297
Epoch 31/100
531/531 [==============================] - 4s 8ms/step - loss:
0.4791 - accuracy: 0.9906 - val_loss: 0.7302 - val_accuracy:
0.8317
Epoch 32/100
531/531 [==============================] - 4s 8ms/step - loss:
0.4032 - accuracy: 0.9917 - val_loss: 0.7361 - val_accuracy:
0.8317
Epoch 33/100
531/531 [==============================] - 4s 8ms/step - loss:
0.4039 - accuracy: 0.9919 - val_loss: 0.7441 - val_accuracy:
0.8261
Epoch 34/100
531/531 [==============================] - 4s 8ms/step - loss:
0.3883 - accuracy: 0.9923 - val_loss: 0.7457 - val_accuracy:
0.8292
Epoch 35/100
531/531 [==============================] - 4s 8ms/step - loss:
```

```
0.3482 - accuracy: 0.9942 - val_loss: 0.7600 - val_accuracy:
0.8275
Epoch 36/100
531/531 [==============================] - 5s 9ms/step - loss:
0.3278 - accuracy: 0.9940 - val_loss: 0.7609 - val_accuracy:
0.8295
Epoch 37/100
531/531 [==============================] - 4s 8ms/step - loss:
0.3191 - accuracy: 0.9942 - val_loss: 0.7683 - val_accuracy:
0.8290
Epoch 38/100
531/531 [==============================] - 4s 8ms/step - loss:
0.3362 - accuracy: 0.9943 - val_loss: 0.7717 - val_accuracy:
0.8288
Epoch 39/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2956 - accuracy: 0.9947 - val_loss: 0.7717 - val_accuracy:
0.8302
Epoch 40/100
531/531 [==============================] - 4s 8ms/step - loss:
0.3042 - accuracy: 0.9942 - val_loss: 0.7848 - val_accuracy:
0.8326
Epoch 41/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2688 - accuracy: 0.9954 - val_loss: 0.7873 - val_accuracy:
```

```
0.8317
Epoch 42/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2625 - accuracy: 0.9953 - val_loss: 0.7924 - val_accuracy:
0.8302
Epoch 43/100
531/531 [==============================] - 5s 8ms/step - loss:
0.2561 - accuracy: 0.9954 - val_loss: 0.7943 - val_accuracy:
0.8297
Epoch 44/100
531/531 [==============================] - 5s 9ms/step - loss:
0.2245 - accuracy: 0.9961 - val_loss: 0.7995 - val_accuracy:
0.8317
Epoch 45/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2569 - accuracy: 0.9954 - val_loss: 0.8092 - val_accuracy:
0.8292
Epoch 46/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2410 - accuracy: 0.9959 - val_loss: 0.8092 - val_accuracy:
0.8314
Epoch 47/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2194 - accuracy: 0.9959 - val_loss: 0.8133 - val_accuracy:
0.8275
```

```
Epoch 48/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2042 - accuracy: 0.9966 - val_loss: 0.8152 - val_accuracy:
0.8297
Epoch 49/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1949 - accuracy: 0.9964 - val_loss: 0.8224 - val_accuracy:
0.8288
Epoch 50/100
531/531 [==============================] - 5s 9ms/step - loss:
0.2053 - accuracy: 0.9964 - val_loss: 0.8172 - val_accuracy:
0.8300
Epoch 51/100
531/531 [==============================] - 4s 8ms/step - loss:
0.2076 - accuracy: 0.9961 - val_loss: 0.8298 - val_accuracy:
0.8307
Epoch 52/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1786 - accuracy: 0.9970 - val_loss: 0.8308 - val_accuracy:
0.8309
Epoch 53/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1694 - accuracy: 0.9974 - val_loss: 0.8380 - val_accuracy:
0.8319
Epoch 54/100
```

```
531/531 [==============================] - 5s 9ms/step - loss:
0.1737 - accuracy: 0.9966 - val_loss: 0.8390 - val_accuracy:
0.8300
Epoch 55/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1752 - accuracy: 0.9969 - val_loss: 0.8389 - val_accuracy:
0.8271
Epoch 56/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1638 - accuracy: 0.9973 - val_loss: 0.8438 - val_accuracy:
0.8307
Epoch 57/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1627 - accuracy: 0.9970 - val_loss: 0.8474 - val_accuracy:
0.8309
Epoch 58/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1621 - accuracy: 0.9972 - val_loss: 0.8518 - val_accuracy:
0.8329
Epoch 59/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1530 - accuracy: 0.9974 - val_loss: 0.8524 - val_accuracy:
0.8319
Epoch 60/100
531/531 [==============================] - 4s 8ms/step - loss:
```

```
0.1619 - accuracy: 0.9968 - val_loss: 0.8556 - val_accuracy:
0.8317
Epoch 61/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1450 - accuracy: 0.9976 - val_loss: 0.8704 - val_accuracy:
0.8300
Epoch 62/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1656 - accuracy: 0.9966 - val_loss: 0.8672 - val_accuracy:
0.8275
Epoch 63/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1386 - accuracy: 0.9976 - val_loss: 0.8645 - val_accuracy:
0.8324
Epoch 64/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1377 - accuracy: 0.9975 - val_loss: 0.8697 - val_accuracy:
0.8285
Epoch 65/100
531/531 [==============================] - 5s 8ms/step - loss:
0.1511 - accuracy: 0.9972 - val_loss: 0.8693 - val_accuracy:
0.8331
Epoch 66/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1450 - accuracy: 0.9975 - val_loss: 0.8744 - val_accuracy:
```

```
0.8314
Epoch 67/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1413 - accuracy: 0.9976 - val_loss: 0.8756 - val_accuracy:
0.8302
Epoch 68/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1326 - accuracy: 0.9974 - val_loss: 0.8799 - val_accuracy:
0.8273
Epoch 69/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1425 - accuracy: 0.9975 - val_loss: 0.8755 - val_accuracy:
0.8317
Epoch 70/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1448 - accuracy: 0.9972 - val_loss: 0.8793 - val_accuracy:
0.8305
Epoch 71/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1185 - accuracy: 0.9975 - val_loss: 0.8894 - val_accuracy:
0.8292
Epoch 72/100
531/531 [==============================] - 5s 10ms/step - loss:
0.1233 - accuracy: 0.9979 - val_loss: 0.8914 - val_accuracy:
0.8288
```

```
Epoch 73/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1395 - accuracy: 0.9978 - val_loss: 0.8879 - val_accuracy:
0.8319
Epoch 74/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1126 - accuracy: 0.9981 - val_loss: 0.8919 - val_accuracy:
0.8300
Epoch 75/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1303 - accuracy: 0.9977 - val_loss: 0.8992 - val_accuracy:
0.8324
Epoch 76/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1247 - accuracy: 0.9975 - val_loss: 0.9046 - val_accuracy:
0.8288
Epoch 77/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1217 - accuracy: 0.9978 - val_loss: 0.8998 - val_accuracy:
0.8297
Epoch 78/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1256 - accuracy: 0.9976 - val_loss: 0.9039 - val_accuracy:
0.8319
Epoch 79/100
```

```
531/531 [==============================] - 5s 9ms/step - loss:
0.1284 - accuracy: 0.9974 - val_loss: 0.9012 - val_accuracy:
0.8305
Epoch 80/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1204 - accuracy: 0.9979 - val_loss: 0.9025 - val_accuracy:
0.8297
Epoch 81/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1121 - accuracy: 0.9980 - val_loss: 0.9131 - val_accuracy:
0.8275
Epoch 82/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1153 - accuracy: 0.9980 - val_loss: 0.9135 - val_accuracy:
0.8309
Epoch 83/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1092 - accuracy: 0.9981 - val_loss: 0.9152 - val_accuracy:
0.8312
Epoch 84/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0935 - accuracy: 0.9984 - val_loss: 0.9175 - val_accuracy:
0.8290
Epoch 85/100
531/531 [==============================] - 4s 8ms/step - loss:
```

```
0.1042 - accuracy: 0.9979 - val_loss: 0.9181 - val_accuracy:
0.8295
Epoch 86/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1098 - accuracy: 0.9981 - val_loss: 0.9175 - val_accuracy:
0.8302
Epoch 87/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1022 - accuracy: 0.9981 - val_loss: 0.9198 - val_accuracy:
0.8319
Epoch 88/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1036 - accuracy: 0.9980 - val_loss: 0.9220 - val_accuracy:
0.8324
Epoch 89/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1143 - accuracy: 0.9977 - val_loss: 0.9240 - val_accuracy:
0.8307
Epoch 90/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1047 - accuracy: 0.9982 - val_loss: 0.9324 - val_accuracy:
0.8285
Epoch 91/100
531/531 [==============================] - 4s 8ms/step - loss:
0.1002 - accuracy: 0.9982 - val_loss: 0.9309 - val_accuracy:
```

```
0.8268
Epoch 92/100
531/531 [==============================] - 5s 9ms/step - loss:
0.1052 - accuracy: 0.9979 - val_loss: 0.9309 - val_accuracy:
0.8273
Epoch 93/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0919 - accuracy: 0.9979 - val_loss: 0.9342 - val_accuracy:
0.8317
Epoch 94/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0916 - accuracy: 0.9981 - val_loss: 0.9303 - val_accuracy:
0.8307
Epoch 95/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0977 - accuracy: 0.9982 - val_loss: 0.9397 - val_accuracy:
0.8275
Epoch 96/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0966 - accuracy: 0.9981 - val_loss: 0.9381 - val_accuracy:
0.8275
Epoch 97/100
531/531 [==============================] - 4s 8ms/step - loss:
0.0846 - accuracy: 0.9984 - val_loss: 0.9384 - val_accuracy:
0.8283
```

```
Epoch 98/100
531/531 [==============================] - 4s 8ms/step - loss: 0.0956 - accuracy: 0.9980 - val_loss: 0.9400 - val_accuracy: 0.8305
Epoch 99/100
531/531 [==============================] - 4s 8ms/step - loss: 0.0883 - accuracy: 0.9982 - val_loss: 0.9500 - val_accuracy: 0.8278
Epoch 100/100
531/531 [==============================] - 4s 8ms/step - loss: 0.0941 - accuracy: 0.9982 - val_loss: 0.9412 - val_accuracy: 0.8288
```

6. Evaluation

Note: A sequence model like RNN or LSTM was not considered because it was overfitting strongly

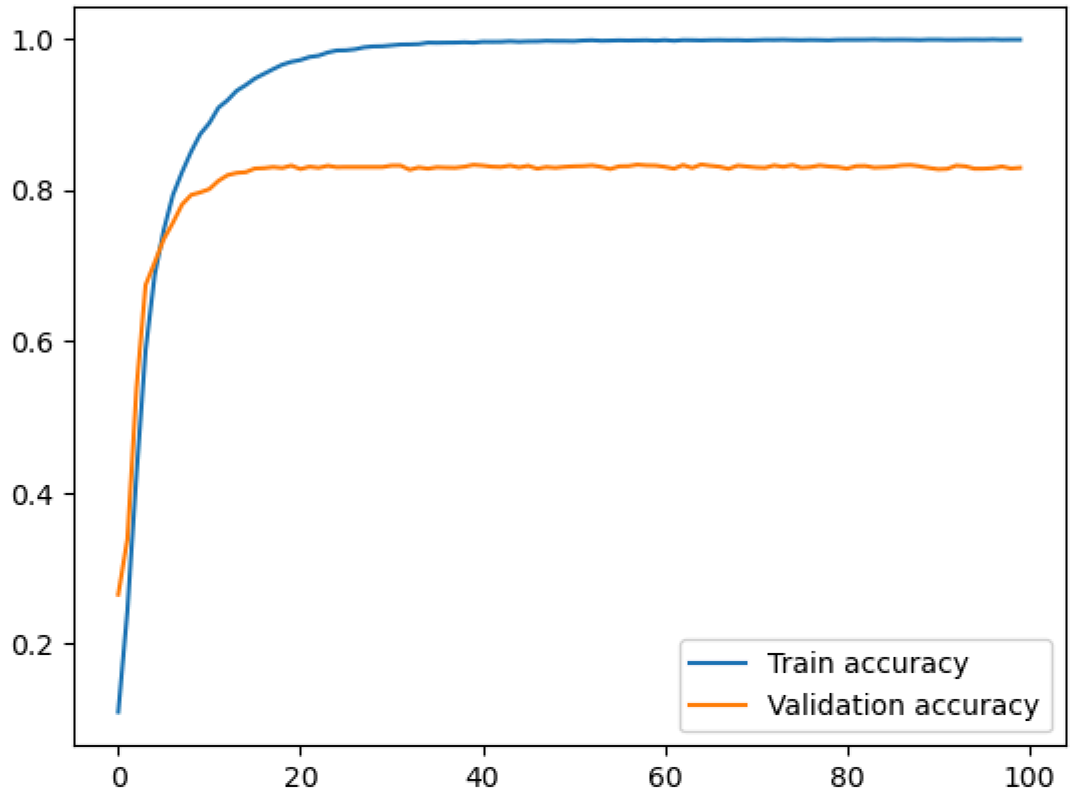
The CNN performed much better for the sequence and Embedding data

6.1 Plain Feedforward Network Evaluation

In [80]:

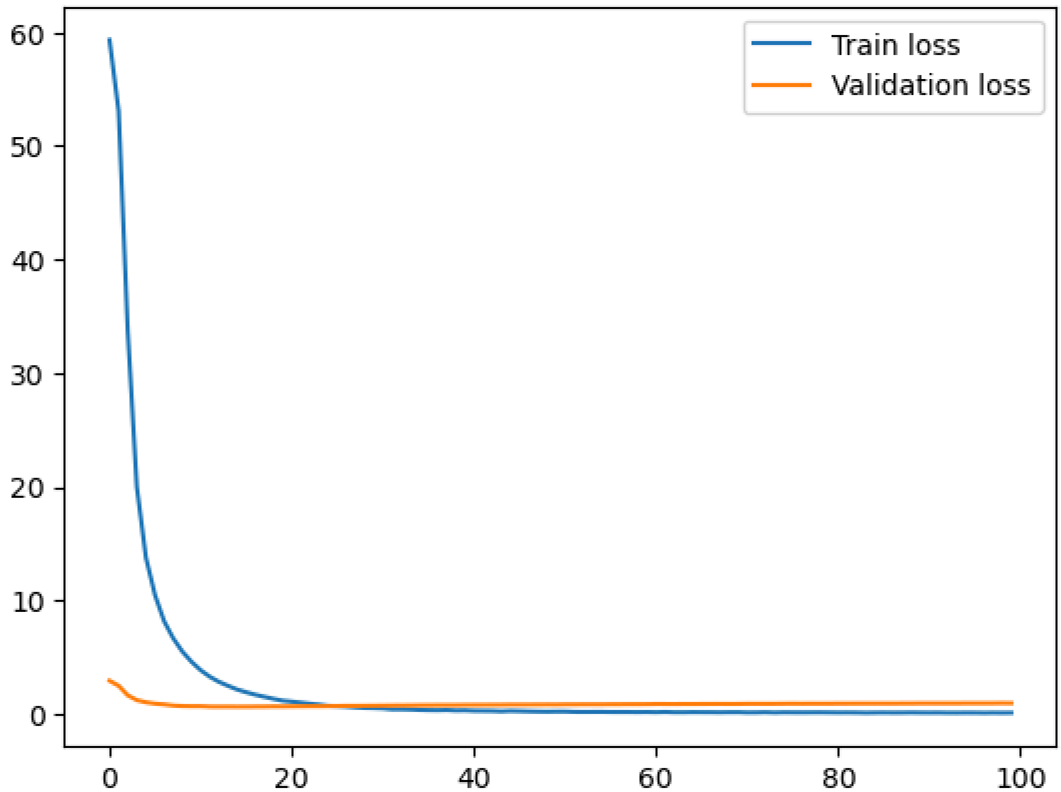
```python
plt.plot(r2.history["accuracy"], label="Train accuracy")
plt.plot(r2.history["val_accuracy"], label="Validation accuracy")
plt.legend()
```

```
plt.show()
```



In [81]:

```
plt.plot(r2.history["loss"], label="Train loss")
plt.plot(r2.history["val_loss"], label="Validation loss")
plt.legend()
plt.show()
```



In [82]:

```
from sklearn.metrics import f1_score, classification_report
preds = np.argmax(model2.predict(test_data_ann), axis=1)
```

```
129/129 [==============================] - 0s 3ms/step
```

In [83]:

```
print("Classification report for test data")
print(classification_report(preds,test_labels))
```

Classification report for test data

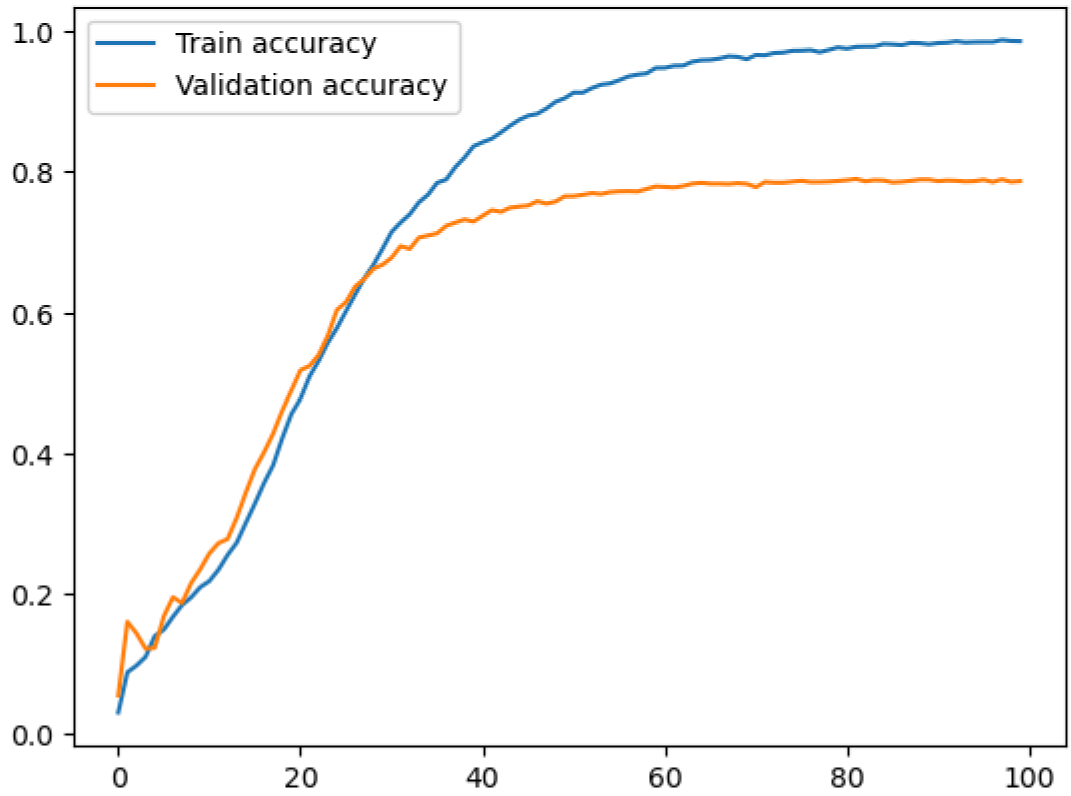
	precision	recall	f1-score	support
0	0.79	0.82	0.81	71
1	0.86	0.86	0.86	215
2	0.86	0.82	0.84	888
3	0.77	0.88	0.82	67
4	0.99	0.95	0.97	101
5	0.94	0.91	0.92	250
6	0.82	0.82	0.82	147
7	0.85	0.86	0.85	159
8	0.84	0.77	0.81	35
9	0.76	0.71	0.73	363
10	0.69	0.41	0.51	22
11	0.93	0.87	0.90	15
12	0.81	0.88	0.84	109
13	0.72	0.74	0.73	113
14	0.82	0.82	0.82	416
15	0.78	0.76	0.77	129
16	0.88	0.92	0.90	239
17	0.81	0.88	0.85	103
18	0.81	0.87	0.84	487
19	0.71	0.74	0.73	188
accuracy			0.83	4117

```
   macro avg       0.82      0.81      0.82      4117
weighted avg       0.83      0.83      0.83      4117
```

6.2 Plain Feedforward Network Evaluation

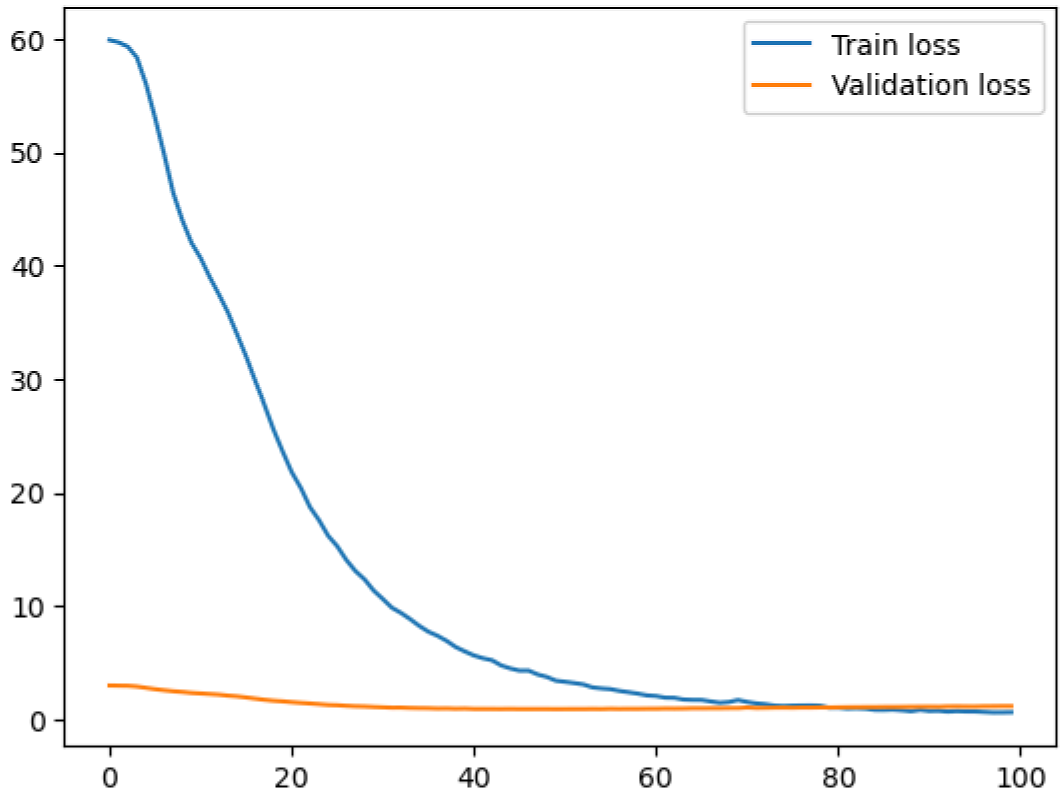
In [84]:

```
plt.plot(r1.history["accuracy"], label="Train accuracy")
plt.plot(r1.history["val_accuracy"], label="Validation 
accuracy")
plt.legend()
plt.show()
```



In [85]:

```
plt.plot(r1.history["loss"], label="Train loss")
plt.plot(r1.history["val_loss"], label="Validation loss")
plt.legend()
plt.show()
```



In [86]:

```
preds = np.argmax(model .predict(test_data_padded), axis=1)
print("Classification report for test data")
print(classification_report(preds,test_labels))
```

```
129/129 [==============================] - 0s 2ms/step
Classification report for test data
              precision    recall  f1-score   support


           0       0.41      0.88      0.56        34
           1       0.79      0.87      0.82       194
           2       0.84      0.74      0.79       962
```

3	0.74	0.79	0.77	72
4	0.96	0.97	0.96	96
5	0.93	0.89	0.91	254
6	0.79	0.80	0.79	144
7	0.82	0.82	0.82	161
8	0.88	0.76	0.81	37
9	0.72	0.68	0.70	360
10	0.77	0.48	0.59	21
11	0.93	0.81	0.87	16
12	0.73	0.75	0.74	116
13	0.62	0.88	0.73	82
14	0.78	0.76	0.77	425
15	0.75	0.79	0.77	119
16	0.83	0.84	0.84	246
17	0.76	0.93	0.84	91
18	0.76	0.78	0.77	515
19	0.72	0.83	0.77	172
accuracy			0.79	4117
macro avg	0.78	0.80	0.78	4117
weighted avg	0.79	0.79	0.79	4117

[Reference link]