Amazon Food Review Rating Prediction

This file experiments with different neural network based models to come up with a great one to submit to the class Kaggle Competition

The pretrained GloVe models used to produce meaningful embeddings were obtained from [https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/)](https://nlp.stanford.edu/projects/glove/). (Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.)

In [2]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
plt.style.use("seaborn")
import seaborn as sns
```

Get Data

In [12]:
```
# train data
df_train = pd.read_csv("./data/train.csv")
display(df_train.head(3))
print(df_train.shape)
```

	Score	Review_text
0	5	I received this product early from the seller!...
1	5	***** Numi's Collection Assortment Melang...
2	5	I was very careful not to overcook this pasta,...

```
(426340, 2)
```

In [167]:
```
# test data
df_test = pd.read_csv("./data/test_new.csv")
display(df_test.head(3))
print(df_test.shape)
```

	Id	Review_text
0	1	I have a very picky German Shephard mix and sh...
1	2	It is hard to believe that this candy is sugar...
2	3	These are delicious cookies but I just cancell...

```
(142114, 2)
```

Clean Data

In [168]:
```
import re
from emot.emo_unicode import EMOTICONS_EMO # for emoticon expansion
```

In [169]:

```
CONTRACTIONS = {
    "ain't": "is not",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
    "'cause": "because",
    "could've": "could have",
    "couldn't": "could not",
    "couldn't've": "could not have",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hadn't've": "had not have",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd": "he would",
    "he'd've": "he would have",
    "he'll": "he will",
    "he'll've": "he will have",
    "he's": "he is",
    "how'd": "how did",
    "how'd'y": "how do you",
    "how'll": "how will",
    "how's": "how is",
    "i'd": "i would",
    "i'd've": "i would have",
    "i'll": "i will",
    "i'll've": "i will have",
    "i'm": "i am",
    "i've": "i have",
    "isn't": "is not",
    "it'd": "it would",
    "it'd've": "it would have",
    "it'll": "it will",
    "it'll've": "it will have",
    "it's": "it is",
    "let's": "let us",
    "ma'am": "madam",
    "mayn't": "may not",
    "might've": "might have",
    "mightn't": "might not",
    "mightn't've": "might not have",
    "must've": "must have",
    "mustn't": "must not",
    "mustn't've": "must not have",
    "needn't": "need not",
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",
    "she'll": "she will",
    "she'll've": "she will have",
    "she's": "she is",
```

```
    "should've": "should have",
    "shouldn't": "should not",
    "shouldn't've": "should not have",
    "so've": "so have",
    "so's": "so as",
    "that'd": "that would",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    "we'd": "we would",
    "we'd've": "we would have",
    "we'll": "we will",
    "we'll've": "we will have",
    "we're": "we are",
    "we've": "we have",
    "weren't": "were not",
    "what'll": "what will",
    "what'll've": "what will have",
    "what're": "what are",
    "what's": "what is",
    "what've": "what have",
    "when's": "when is",
    "when've": "when have",
    "where'd": "where did",
    "where's": "where is",
    "where've": "where have",
    "who'll": "who will",
    "who'll've": "who will have",
    "who's": "who is",
    "who've": "who have",
    "why's": "why is",
    "why've": "why have",
    "will've": "will have",
    "won't": "will not",
    "won't've": "will not have",
    "would've": "would have",
    "wouldn't": "would not",
    "wouldn't've": "would not have",
    "y'all": "you all",
    "y'all'd": "you all would",
    "y'all'd've": "you all would have",
    "y'all're": "you all are",
    "y'all've": "you all have",
    "you'd": "you would",
    "you'd've": "you would have",
    "you'll": "you will",
    "you'll've": "you will have",
    "you're": "you are",
    "you've": "you have"
}
```

```python

def clean(text):
    ''' Function that returns a given piece of text after cleaning it.
    # check for html embeddings and sub them out with a space
    text = re.sub("<[^<>]*>"," ",text)

    # if an emoticon is encountered, replace with word equivalent
    for emote in EMOTICONS_EMO:
        text = text.replace(emote, " "+EMOTICONS_EMO[emote])

    # make lowercase
    text = text.lower().strip()

    # expand contractions
    for key, val in CONTRACTIONS.items():
        text = re.sub(key, val, text)

    # discard 's
    text = re.sub("'s","",text)

    # discard if it is not a-z or space
    text = re.sub('-'," ",text)

    # discard if it is not a-z or space
    text = re.sub('[^a-z ]*',"",text)

    # discard urls
    text = re.sub('(https?\w*)|(www\w*)',"",text)

    # discard extra spaces
    text = re.sub('\s\s+'," ",text)

    # check if final text is empty or not
    if re.search("[a-z]", text): return text
    else: return ""
```

WARNING: Time consuming cell ahead! (upto 3 mins)

In [20]:

```python
# clean train data
df_train["review"] = df_train["Review_text"].apply(lambda review: clean
df_train.drop(["Review_text"], axis=1)
df_train = df_train[df_train["review"] != ""]
```

In [170]:

```python
# clean test data
df_test["review"] = df_test["Review_text"].apply(lambda review: clean(r
df_test.drop(["Review_text"], axis=1)
df_test = df_test[df_test["review"] != ""]
```

Feature Engineering

In [171]:
```
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
sid = SentimentIntensityAnalyzer()
import math
```

```
[nltk_data] Downloading package vader_lexicon to C:\Users\Gayathri
[nltk_data]     Girish Nair\AppData\Roaming\nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
```

WARNING: Time consuming cell ahead! (upto 5 mins)

In [27]:
```
# adding features "polarity", "word_count" and "longest_word_length" to
df_train = pd.DataFrame({
    "review": df_train.review,
    "word_count": df_train["review"].apply(lambda review: math.log(len(
    "longest_word_length": df_train["review"].apply(
        lambda review: math.log(max([len(word) for word in review.split
    ),
    "polarity": df_train["review"].apply(lambda review: sid.polarity_sc
    "score": df_train.Score
})
display(df_train.head(3))
```

	review	word_count	longest_word_length	polarity	score
0	i received this product early from the seller ...	3.637586	2.079442	0.9488	5
1	numi collection assortment melange includes h...	5.755742	2.484907	0.9921	5
2	i was very careful not to overcook this pasta ...	5.323010	2.639057	0.9980	5

WARNING: Time consuming cell ahead! (upto 3 mins)

In [173]:
```
# adding features "polarity", "word_count" and "longest_word_length" to
df_test = pd.DataFrame({
    "Id": df_test.Id,
    "review": df_test.review,
    "word_count": df_test["review"].apply(lambda review: math.log(len(r
    "longest_word_length": df_test["review"].apply(
        lambda review: math.log(max([len(word) for word in review.split
    ),
    "polarity": df_test["review"].apply(lambda review: sid.polarity_sco
})
display(df_test.head(3))
```

	Id	review	word_count	longest_word_length	polarity
0	1	i have a very picky german shephard mix and sh...	5.099866	2.302585	0.9584
1	2	it is hard to believe that this candy is sugar...	4.110874	2.079442	0.8720
2	3	these are delicious cookies but i just cancell...	3.970292	2.484907	0.8663

In [174]:
```python
# standardize "word_count" and "longest_word_length"
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_train = scaler.fit_transform(df_train[["word_count", "longest_wo
scaled_test = scaler.transform(df_test[["word_count", "longest_word_len

df_train["word_count"] = scaled_train[:, 0]
df_train["longest_word_length"] = scaled_train[:, 1]

df_test["word_count"] = scaled_test[:, 0]
df_test["longest_word_length"] = scaled_test[:, 1]

del scaled_train
del scaled_test
```

Text Processing And Normalization

In [175]:
```python
# stop word removal
from nltk.corpus import stopwords
import nltk

STOP_WORDS = set(stopwords.words('english'))
def remove_stopwords(text): # (M.D. Pietro, 2017)
    ''' Function that removes stopwords from a given list of word
        and returns this new possibly smaller list.
        @param text: The list of words from which to remove stopwords.
    '''
    return " ".join([word for word in text.split(" ") if not word in ST

df_train["review"] = df_train.review.apply(lambda review: remove_stopwo
df_test["review"] = df_test.review.apply(lambda review: remove_stopword
```

WARNING: Time consuming cell ahead! (upto 30 mins)

In [176]:
```python
# lemmatization
import spacy
nlp = spacy.load('en_core_web_sm', disable=['ner', 'parser']) # !python
nlp.add_pipe('sentencizer')
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"us"}]],{"LEMMA":"us"})
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"them"}]],{"LEMMA":"them"
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"nt"}]],{"LEMMA":"not"})

def lemmatize(text_list):
    doc = nlp(text_list)
    return " ".join([token.lemma_ for token in doc])

# df_train["review"] = df_train.review.apply(lambda review: lemmatize(r
df_test["review"] = df_test.review.apply(lambda review: lemmatize(revie
```

In [177]:
```python
# save processed data
# df_train.to_csv("data/train_processed_full.csv", index=False)
df_test.to_csv("data/test_processed_full.csv", index=False)
```

Review Vectorization

In [178]:
```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import label_binarize
from imblearn.under_sampling import NearMiss
import json
```

In [179]:
```python
# load stored processed data
df_train = pd.read_csv("data/train_processed_full.csv")
df_test = pd.read_csv("data/test_processed_full.csv")
```

In [181]:
```python
# for resampling
def strategy(x, y, threshold, t='majority'):
    ''' Function that aids in doing over/under sampling. '''
    targets = ''
    if t == 'majority': targets = y.value_counts() > threshold
    elif t == 'minority': targets = y.value_counts() < threshold
    tc = targets[targets == True].index
    strategy = {}
    for target in tc: strategy[target] = threshold
    return strategy

def uo_resample(x, y, type):
    if not type in ["under", "over"]: raise Exception("resample: invali

    if type == "under": # undersampling
        resample = NearMiss(version=1, n_neighbors=3, sampling_strategy
        x, y = resample.fit_resample(x, y)

    else: # oversampling
        resample = SMOTE(random_state=30, sampling_strategy=strategy(x,
        x, y = resample.fit_resample(x, y)

    return x, y
```

WARNING: Time consuming cell ahead! (upto 5 mins)

In [24]:
```python
# get bag of words vectors
vectorizer = TfidfVectorizer(ngram_range=(2,2), max_features=1000)
tfidf_train = vectorizer.fit_transform(df_train['review'])
tfidf_test = vectorizer.transform(df_test['review'])

# get TF-IDF scores
tfidf_df_train = pd.DataFrame(data=tfidf_train.toarray(), columns=vecto
tfidf_df_test = pd.DataFrame(data=tfidf_test.toarray(), columns=vectori

# concatenate added features
bow_train = pd.concat([tfidf_df_train, df_train[["polarity", "word_coun
bow_labels_train = df_train["score"]
bow_test = pd.concat([tfidf_df_test, df_test[["polarity", "word_count",

# resampling
bow_train, bow_labels_train = uo_resample(x=bow_train, y=bow_labels_tra
bow_train, bow_labels_train = uo_resample(x=bow_train, y=bow_labels_tra
```

WARNING: Time consuming cell ahead! (upto 3 mins)

In [73]:

```
# get global vectors
from sklearn import preprocessing

# load pretrained embeddings
GLOVE_MODEL_100 = {}
with open("./data/glove.6B.100d.txt", "r", encoding="utf8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], 'float32')
        GLOVE_MODEL_100[word] = vector
f.close()

# get GloVe embeddings
def glove_embed(docs):
    ''' Function that converts given documents
        into vectors using a GloVe model.
        @param docs: Documents to embed as an iterable.
        @return docs: Embedded documents as np array.
    '''
    docs_embedded = []
    for doc_words in [doc.split(" ") for doc in docs]:
        doc_vector = np.array([0.0]*100)
        for word in doc_words:
            try: word_vec = GLOVE_MODEL_100[word]
            except: word_vec = np.array([0.0]*100)
            doc_vector += word_vec
        docs_embedded.append(doc_vector)
    docs_embedded = preprocessing.normalize(docs_embedded, axis=1)
    return np.array(docs_embedded)

glove_df_train = pd.DataFrame(data=glove_embed(df_train["review"]),colu
glove_df_test = pd.DataFrame(data=glove_embed(df_test["review"]),column

# concatenate added features
we_train = pd.concat([glove_df_train, df_train[["polarity", "word_count
we_labels_train = df_train["score"]
we_test = pd.concat([glove_df_test, df_test[["polarity", "word_count",

# resampling
we_train, we_labels_train = resample(x=we_train, y=we_labels_train, typ
we_train, we_labels_train = resample(x=we_train, y=we_labels_train, typ
```

WARNING: Time consuming cell ahead! (upto 4 mins)

In [250]:

```
# save vectorized data
bow_train["score"] = bow_labels_train
we_train["score"] = we_labels_train

bow_train.to_csv("./data/bow_train_full.csv", index=False)
bow_test.to_csv("./data/bow_test_full.csv", index=False)
we_train.to_csv("./data/we_train_full.csv", index=False)
we_test.to_csv("./data/we_test_full.csv", index=False)
```

In [ ]:
```python
sample = pd.read_csv("./SampleSubmission.csv")
```

Model: LSTM RNN

OBSERVATION: The LSTM model whose predictions are to be entered in the Kaggle competition was build based on learnings obtained from experiments conducted as part of CW2.

- An LSTM variation of RNN models was chosen because LSTM was the model that did best with train and test data among CW2 experiments.
- Additional dropout layers were added to this model since it was observed in CW2 that the LSTM model tends to overfit.
- The embedding layer outputs embeddings as generated by a pre-trained GloVe model since this was seen to produce most meaningful embeddings in CW2 experiments with LSTM and GRU RNN models.
- Since the added feature "polarity" had high correlation with ratings, this feature value shall be concatenated with the output of the LSTM layers and fed to dense layers to produce more fine tuned predictions.
- Activation function chosen for dense layers were "relu" ensuring that this model performs well even when data is not linearly separable as is most likely the case here.

In [13]:
```python
import tensorflow as tf
import keras
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.initializers import Constant
from tensorflow.keras.optimizers import Adam
from keras.preprocessing.text import Tokenizer
import json
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.initializers import Constant
from tensorflow.keras.optimizers import Adam
import seaborn as sns
```

In [17]:
```python
import gc

def clean_memory():
    ''' Function that cleans unused memory. '''
    gc.collect()
    tf.keras.backend.clear_session()
```

In [234]:
```python
df_train = pd.read_csv("./data/train_processed_full.csv")
df_test = pd.read_csv("./data/test_processed_full.csv")
```

In [9]:
```
def get_padded_seqmod_input(train_sentences, test_sentences, max_len):
    # map words in vocabulary to index of corresponding token after tok
    tokenizer = Tokenizer(oov_token="[UNK]") # replace out of vocabular
    tokenizer.fit_on_texts(train_sentences)
    word_index = tokenizer.word_index

    # replace words in each document with its corresponding index in wo
    train_sequences = tokenizer.texts_to_sequences(train_sentences)
    test_sequences = tokenizer.texts_to_sequences(test_sentences)

    # pad all document sequences to have max_len words.
    train_padded = pad_sequences(train_sequences, maxlen=max_len, paddi
    test_padded = pad_sequences(test_sequences, maxlen=max_len, padding

    return train_padded, test_padded, tokenizer, word_index
```

In [10]:
```
def get_embedding_matrix(word_index, glove_dim):

    embedding_dict = {}
    with open(f"./data/glove.6B.{glove_dim}d.txt", "r", encoding="utf8"
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], 'float32')
            embedding_dict[word] = vector
    f.close()

    # create GloVe embedding matrix which is a matrix of GloVe embeddin
    # such that each embedding has index equal to the index of correspo
    num_words = len(word_index) + 1
    embedding_matrix = np.zeros((num_words, glove_dim)) # each word vec
    for word, i in word_index.items():
        if i < num_words:
            emb_vec = embedding_dict.get(word)
            if emb_vec is not None: embedding_matrix[i] = emb_vec

    return embedding_matrix, num_words
```

In [237]:
```
train_sentences = df_train["review"]
train_labels = pd.get_dummies(df_train["score"]).values
test_sentences = df_test["review"]
```

In [238]:
```
max_len = 100
glove_dim = 100
train_padded, test_padded, tokenizer, word_index = get_padded_seqmod_in
embedding_matrix, num_words = get_embedding_matrix(word_index, glove_di
```

In [266]:

```
# make model
input1 = keras.Input(shape=(max_len,)) # documents
input2 = keras.Input(shape=(1,)) # polarity

# layers
embedding = Embedding(
    name="Embedding_Layer", input_dim = num_words, output_dim = glove_d
    input_length = max_len, trainable = False,
    embeddings_initializer = Constant(embedding_matrix)
)

lstm1 = LSTM(name="LSTM_Layer1", units=100, return_sequences=True, drop
lstm2 = LSTM(name="LSTM_Layer2", units=100, return_sequences=False, dro

dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)

dense1 = Dense(name="Dense_Layer1", units=101, activation='relu')
dense2 = Dense(name="Dense_Layer2", units=101, activation='relu')
dense3 = Dense(name="Dense_Layer3", units=5, activation='softmax')

concatenate = keras.layers.Concatenate(axis=1)

# functional API
x = embedding(input1)
x = dropout1(x)
x = lstm1(x)
x = lstm2(x)
x = concatenate([x, input2])
x = dense1(x)
x = dropout2(x)
x = dense2(x)
outputs = dense3(x)

# make model
model = keras.Model(inputs=[input1, input2], outputs=outputs, name="LST

# compile model
model.compile(loss="categorical_crossentropy", optimizer='adam', metric
```

In [4]:
```
import keras
model = keras.models.load_model('models/100dlstm80epochs.h5')
model.summary()
```

```
Model: "LSTM_Functional"
______________________________________________________________________________
________________________
 Layer (type)                   Output Shape         Param #     Connected
to
==============================================================================
========================
 input_1 (InputLayer)           [(None, 100)]        0           []

 Embedding_Layer (Embedding)    (None, 100, 100)     13715300    ['input_1
[0][0]']

 dropout (Dropout)              (None, 100, 100)     0           ['Embeddi
ng_Layer[0][0]']

 LSTM_Layer1 (LSTM)             (None, 100, 100)     80400       ['dropout
[0][0]']

 LSTM_Layer2 (LSTM)             (None, 100)          80400       ['LSTM_La
yer1[0][0]']

 input_2 (InputLayer)           [(None, 1)]          0           []

 concatenate (Concatenate)      (None, 101)          0           ['LSTM_La
yer2[0][0]',
                                                                  'input_2
[0][0]']

 Dense_Layer1 (Dense)           (None, 101)          10302       ['concate
nate[0][0]']

 dropout_1 (Dropout)            (None, 101)          0           ['Dense_L
ayer1[0][0]']

 Dense_Layer2 (Dense)           (None, 101)          10302       ['dropout
_1[0][0]']

 Dense_Layer3 (Dense)           (None, 5)            510         ['Dense_L
ayer2[0][0]']

==============================================================================
========================
Total params: 13,897,214
Trainable params: 181,914
Non-trainable params: 13,715,300
______________________________________________________________________________
________________________
```

In [269]:

```
# train model (this was ran 8 times)
clean_memory()
history = model.fit(
    x=[train_padded, df_train.polarity],
    y=train_labels,
    epochs=10,
    verbose=1,
    shuffle=True
)
```

```
Epoch 1/10
13324/13324 [==============================] - 184s 14ms/step - loss: 0.61
67 - accuracy: 0.7695
Epoch 2/10
13324/13324 [==============================] - 180s 13ms/step - loss: 0.61
62 - accuracy: 0.7693
Epoch 3/10
13324/13324 [==============================] - 179s 13ms/step - loss: 0.61
56 - accuracy: 0.7692
Epoch 4/10
13324/13324 [==============================] - 179s 13ms/step - loss: 0.61
49 - accuracy: 0.7698
Epoch 5/10
13324/13324 [==============================] - 181s 14ms/step - loss: 0.61
47 - accuracy: 0.7697
Epoch 6/10
13324/13324 [==============================] - 189s 14ms/step - loss: 0.61
47 - accuracy: 0.7701
Epoch 7/10
13324/13324 [==============================] - 186s 14ms/step - loss: 0.61
70 - accuracy: 0.7690
Epoch 8/10
13324/13324 [==============================] - 183s 14ms/step - loss: 0.61
48 - accuracy: 0.7697
Epoch 9/10
13324/13324 [==============================] - 183s 14ms/step - loss: 0.61
49 - accuracy: 0.7701
Epoch 10/10
13324/13324 [==============================] - 185s 14ms/step - loss: 0.61
55 - accuracy: 0.7692
```

In [270]:

```
# save model
model.save("models/100dlstm80epochs.h5")
```

In [276]:
```
1 model.summary()
```

```
Model: "LSTM_Functional"
_________________________________________________________________________
_________________________
 Layer (type)                   Output Shape         Param #     Connected
to
=========================================================================
=========================
 input_1 (InputLayer)           [(None, 100)]        0           []

 Embedding_Layer (Embedding)    (None, 100, 100)     13715300    ['input_1
[0][0]']

 dropout (Dropout)              (None, 100, 100)     0           ['Embeddi
ng_Layer[0][0]']

 LSTM_Layer1 (LSTM)             (None, 100, 100)     80400       ['dropout
[0][0]']

 LSTM_Layer2 (LSTM)             (None, 100)          80400       ['LSTM_La
yer1[0][0]']

 input_2 (InputLayer)           [(None, 1)]          0           []

 concatenate (Concatenate)      (None, 101)          0           ['LSTM_La
yer2[0][0]',
                                                                  'input_2
[0][0]']

 Dense_Layer1 (Dense)           (None, 101)          10302       ['concate
nate[0][0]']

 dropout_1 (Dropout)            (None, 101)          0           ['Dense_L
ayer1[0][0]']

 Dense_Layer2 (Dense)           (None, 101)          10302       ['dropout
_1[0][0]']

 Dense_Layer3 (Dense)           (None, 5)            510         ['Dense_L
ayer2[0][0]']

=========================================================================
=========================
Total params: 13,897,214
Trainable params: 181,914
Non-trainable params: 13,715,300
_________________________________________________________________________
_________________________
```

In [21]:

```
def plot_train_val_curves(history):
    # plot training and validation results
    plt.figure(figsize=(18,5))

    plt.subplot(121)
    plt.title("Loss")
    plt.plot(history.history["loss"], label="train")
    plt.plot(history.history["val_loss"], label="validate")
    plt.legend()

    plt.subplot(122)
    plt.title("Accuracy")
    plt.plot(history.history["accuracy"], label="train")
    plt.plot(history.history["val_accuracy"], label="validate")
    plt.legend()

    plt.show()

# plot_train_val_curves(history)
```

In [271]:

```
# make prediction
pred_prob = model.predict([test_padded, df_test.polarity])
pred = np.array([np.argmax(p)+1 for p in pred_prob])
```

In [273]:

```
submission_path = "kaggle_submissions/submission16.csv"
```

In [274]:

```
# save predictions
submission = pd.DataFrame({"Id": df_test.Id, "Score": pred})
submission.to_csv(submission_path, index=False) # submission to be ente
```

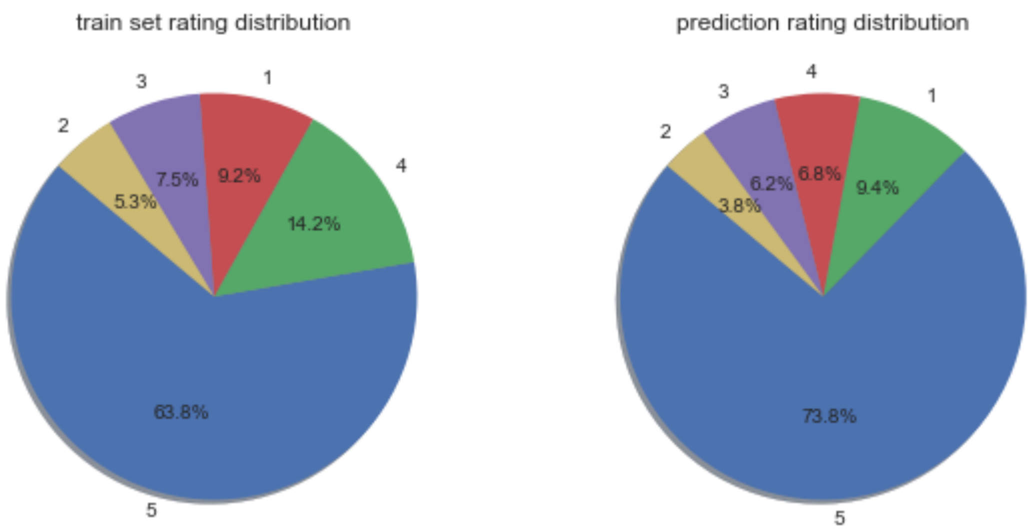
In [203]:

```
def plot_distributions(train_df, pred_df):
    ''' Plot pie charts to view distribution comparison
        between training set labels and predicted values '''

    train_rating_data = list(train_df.score.value_counts().values)
    train_rating_labels = list(train_df.score.value_counts().keys())
    pred_data = list(pred_df.Score.value_counts().values)
    pred_labels = list(pred_df.Score.value_counts().keys())

    plt.figure(figsize=(10,5))
    plt.subplot(121)
    plt.title("train set rating distribution")
    plt.pie(x=train_rating_data, labels=train_rating_labels, autopct='%
    plt.subplot(122)
    plt.title("prediction rating distribution")
    plt.pie(x=pred_data, labels=pred_labels, autopct='%1.1f%%', shadow=
    plt.show()
```

In [275]:

```python
# Load and examine prediction distribution
sub = pd.read_csv(submission_path)
display(sub.describe())

# plot pie charts to view distribution comparison between training set
plot_distributions(df_train, sub)
```

	Id	Score
count	142114.000000	142114.000000
mean	71057.500000	4.318519
std	41024.922415	1.302047
min	1.000000	1.000000
25%	35529.250000	4.000000
50%	71057.500000	5.000000
75%	106585.750000	5.000000
max	142114.000000	5.000000



OBSERVATION: Viewing the distribution of predictions on the test set vs train set labels was found to be a great way to make an educated guess as to whether the model's predictions seem sound.

Experimentation Zone

In [5]:

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from keras.preprocessing.text import Tokenizer
```

OBSERVATION: Since the test set truth values were not available, experiments had to be conducted by splitting the train set into stratified train, test and validation sets such that a large enough portion of the entire dataset was set aside for testing and validation so as to mimic a larger amount of unseen data.

In [2]:

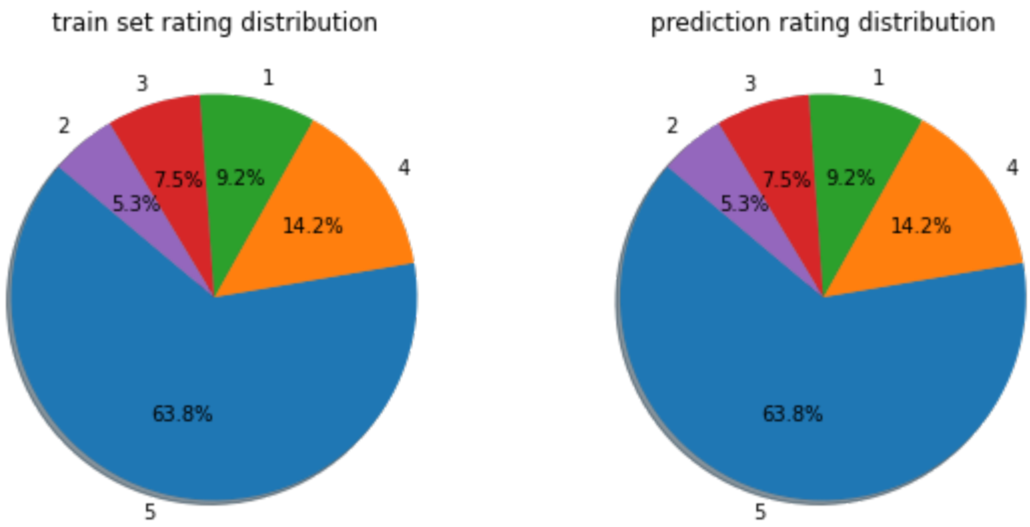
```
data_full = pd.read_csv("./data/train_processed_full.csv")
x = data_full.drop(["score"], axis=1)
y = data_full.score
```

In [3]:

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4
```

In [11]:

```
x_score = pd.DataFrame({"score":y_train})
y_score = pd.DataFrame({"score":y_test})

train_rating_data = list(x_score.score.value_counts().values)
train_rating_labels = list(x_score.score.value_counts().keys())
pred_data = list(y_score.score.value_counts().values)
pred_labels = list(y_score.score.value_counts().keys())

plt.figure(figsize=(10,5))
plt.subplot(121)
plt.title("train set rating distribution")
plt.pie(x=train_rating_data, labels=train_rating_labels, autopct='%1.1f
plt.subplot(122)
plt.title("prediction rating distribution")
plt.pie(x=pred_data, labels=pred_labels, autopct='%1.1f%%', shadow=True
plt.show()
```



In [14]:

```
max_len = 100
glove_dim = 100
train_padded, test_padded, tokenizer, word_index = get_padded_seqmod_in
embedding_matrix, num_words = get_embedding_matrix(word_index, glove_di
train_labels = pd.get_dummies(y_train).values
test_labels = pd.get_dummies(y_test).values
```

In [19]:

```
# make model
input1 = keras.Input(shape=(max_len,)) # documents
input2 = keras.Input(shape=(1,)) # polarity

# layers
embedding = Embedding(
    name="Embedding_Layer", input_dim = num_words, output_dim = glove_d
    input_length = max_len, trainable = False,
    embeddings_initializer = Constant(embedding_matrix)
)

lstm1 = LSTM(name="LSTM_Layer1", units=100, return_sequences=True, drop
lstm2 = LSTM(name="LSTM_Layer2", units=100, return_sequences=False, dro

dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)

dense1 = Dense(name="Dense_Layer1", units=101, activation='relu')
dense2 = Dense(name="Dense_Layer2", units=101, activation='relu')
dense3 = Dense(name="Dense_Layer3", units=5, activation='softmax')

concatenate = keras.layers.Concatenate(axis=1)

# functional API
x = embedding(input1)
x = dropout1(x)
x = lstm1(x)
x = lstm2(x)
x = concatenate([x, input2])
x = dense1(x)
x = dropout2(x)
x = dense2(x)
outputs = dense3(x)

# make model
model = keras.Model(inputs=[input1, input2], outputs=outputs, name="LST

# compile model
model.compile(loss="categorical_crossentropy", optimizer='adam', metric
```

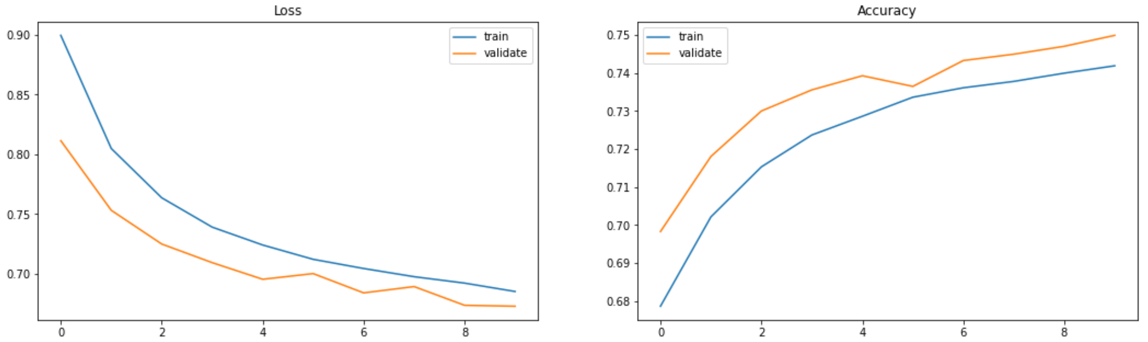
In [20]:

```
# train model
clean_memory()
history = model.fit(
    x=[train_padded, x_train.polarity],
    y=train_labels,
    epochs=10,
    validation_split=0.3,
    verbose=1,
    shuffle=True
)
```

```
Epoch 1/10
5596/5596 [==============================] - 98s 17ms/step - loss: 0.8996
- accuracy: 0.6786 - val_loss: 0.8115 - val_accuracy: 0.6983
Epoch 2/10
5596/5596 [==============================] - 82s 15ms/step - loss: 0.8049
- accuracy: 0.7021 - val_loss: 0.7532 - val_accuracy: 0.7181
Epoch 3/10
5596/5596 [==============================] - 82s 15ms/step - loss: 0.7637
- accuracy: 0.7153 - val_loss: 0.7250 - val_accuracy: 0.7300
Epoch 4/10
5596/5596 [==============================] - 81s 14ms/step - loss: 0.7391
- accuracy: 0.7237 - val_loss: 0.7094 - val_accuracy: 0.7356
Epoch 5/10
5596/5596 [==============================] - 81s 14ms/step - loss: 0.7242
- accuracy: 0.7286 - val_loss: 0.6955 - val_accuracy: 0.7393
Epoch 6/10
5596/5596 [==============================] - 81s 14ms/step - loss: 0.7122
- accuracy: 0.7337 - val_loss: 0.7003 - val_accuracy: 0.7365
Epoch 7/10
5596/5596 [==============================] - 81s 15ms/step - loss: 0.7045
- accuracy: 0.7362 - val_loss: 0.6841 - val_accuracy: 0.7433
Epoch 8/10
5596/5596 [==============================] - 81s 15ms/step - loss: 0.6977
- accuracy: 0.7378 - val_loss: 0.6894 - val_accuracy: 0.7450
Epoch 9/10
5596/5596 [==============================] - 81s 14ms/step - loss: 0.6923
- accuracy: 0.7400 - val_loss: 0.6737 - val_accuracy: 0.7471
Epoch 10/10
5596/5596 [==============================] - 81s 14ms/step - loss: 0.6853
- accuracy: 0.7419 - val_loss: 0.6730 - val_accuracy: 0.7500
```

In [22]:

```
plot_train_val_curves(history)
```



In [23]:

```python
# test model
pred_prob = model.predict([test_padded, x_test.polarity])
pred = np.array([np.argmax(p)+1 for p in pred_prob])
print("\ntrain results")
train_loss, train_acc = model.evaluate([train_padded, x_train.polarity]
print("test results")
test_loss, test_acc = model.evaluate([test_padded, x_test.polarity], te
print(f"Train Accuracy = {train_acc}")
print(f"Test Accuracy = {test_acc}")
```

```
train results
7994/7994 [==============================] - 46s 6ms/step - loss: 0.6348 -
accuracy: 0.7603
test results
5330/5330 [==============================] - 31s 6ms/step - loss: 0.6708 -
accuracy: 0.7494
Train Accuracy = 0.760260820388794
Test Accuracy = 0.7493549585342407
```

The above experiment was the last one among various experiments that were conducted. The model obtained above was put to the test on Kaggle because even after 20 epochs, the model continued learn without convergence or overfitting. It is to be noted that while adding more layers or units may have led to better results, this could not be done here since trying it for just a few epochs made it clear that the exponential increase in computing time and resources meant that such a model was not practical with the machines at hand.

Kaggle Results

OBSERVATION: This model performed as expected and produced similar accuracy as obtained here on Kaggle as well. This model was ranked 3rd by kaggle with a provincial score of 0.777 (77.7% accuracy) and a final score of 0.778 (77.8% accuracy). The model submitted on Kaggle had trained for 90 epochs. Even though it continued to learn in the 91st epoch as well, the learning was very slow and thus it was considered to have converged. The predictions of the model after 90 epochs of training was the final Kaggle submission. It is interesting to note that the score when tested on more data (Kaggle final score) was higher than when tested on lesser data (Kaggle provincial score) further indicating this model's good reliability and resistance to overfitting.