Management and Computer Science

AY 2019/2020

Artificial Intelligence and Machine Learning - Team Project

MBTI Personality Type Prediction

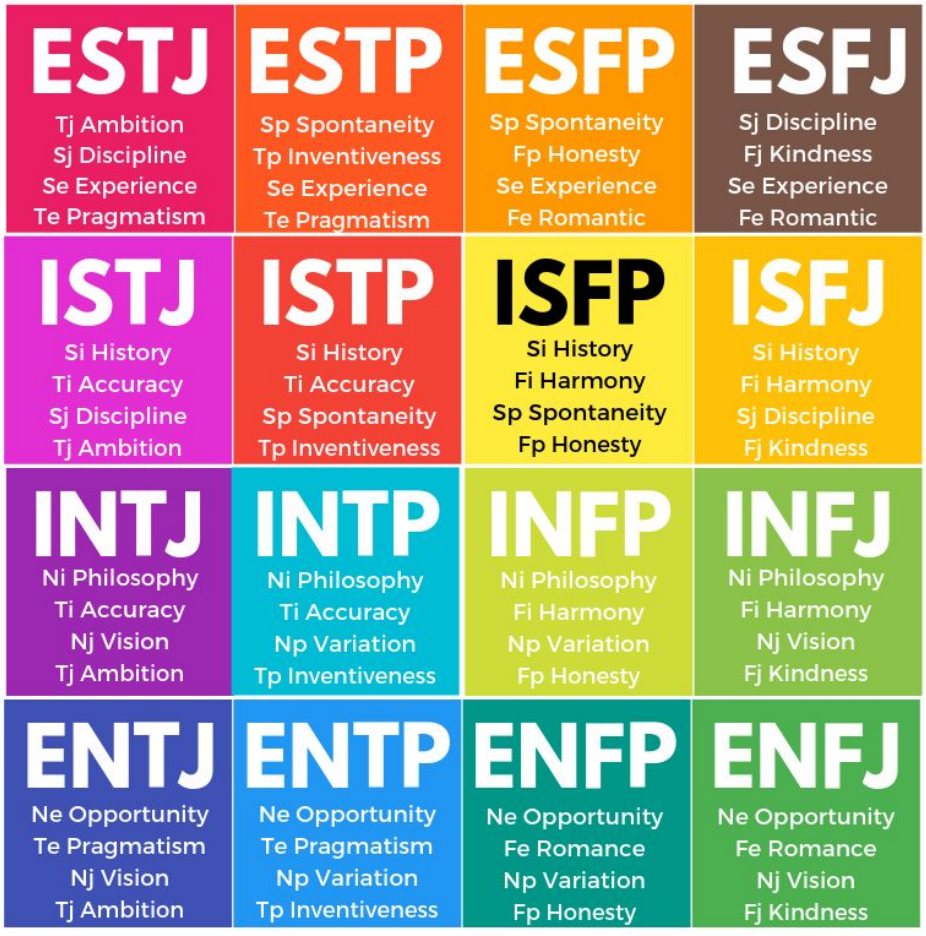
Dario Moceri, Nicolò Pagliari, Aurora Spagnol

The Myers-Briggs Type Indicator (MBTI)

The **Myers-Briggs Type Indicator** (**MBTI**) is a self-report questionnaire designed to identify a person's personality type. There are 16 different personality types composed by four binary categories:

1) **Energy:** **E**xtrovert - **I**ntrovert
2) **Information:** **S**ensing - I**N**tuition
3) **Decision:** **T**hinking - **F**eeling
4) **Lifestyle:** **J**udging - **P**erceiving

Each person's MBTI personality type is defined as the collection of their four types for the four categories.
For example: a personality INTP identifies one who is introverted (I), uses intuition (N) to get and interpret information in the world, takes decisions by thinking rationally (T), and lives following his perception (P).



Our goal

Our goal is to develop a machine learning model capable of predicting a person's personality type by analyzing their social media posts (supervised classification task).
In order to develop our model, we are going to use Kaggle's *MBTI Personality Dataset*.

The MBTI Personality Dataset

The MBTI Dataset is available at: https://www.kaggle.com/datasnaek/mbti-type
This dataset contains over **8,600 rows of data**. Each row represents a person, and contains two columns:

- **type**: the four letters identifying the MBTI's personality type. This is our *response variable,* which is the value we want to predict.
- **posts**: a section of at last 50 posts of the person with that personality type. Each post is separated by "|||".

type	posts
INFJ	'http://www.youtube.com/watch?v=qsXHcwe3krw|||http://41.media.tumblr.com/tumblr_lfouy03PMA1qa1rc
ENTP	'I'm finding the lack of me in these posts very alarming.|||Sex can be boring if it's in the same position ofte
INTP	'Good one _____ https://www.youtube.com/watch?v=fHiGbolFFGw|||Of course, to which I say I know; th
INTJ	'Dear INTP, I enjoyed our conversation the other day. Esoteric gabbing about the nature of the universe a
ENTJ	'You're fired.|||That's another silly misconception. That approaching is logically is going to be the key to un
INTJ	'18/37 @.@|||Science is not perfect. No scientist claims that it is, or that scientific information will not be r unquestionably). Yes, most useful when thought of as a text by man about God. ...|||Apparently, you need

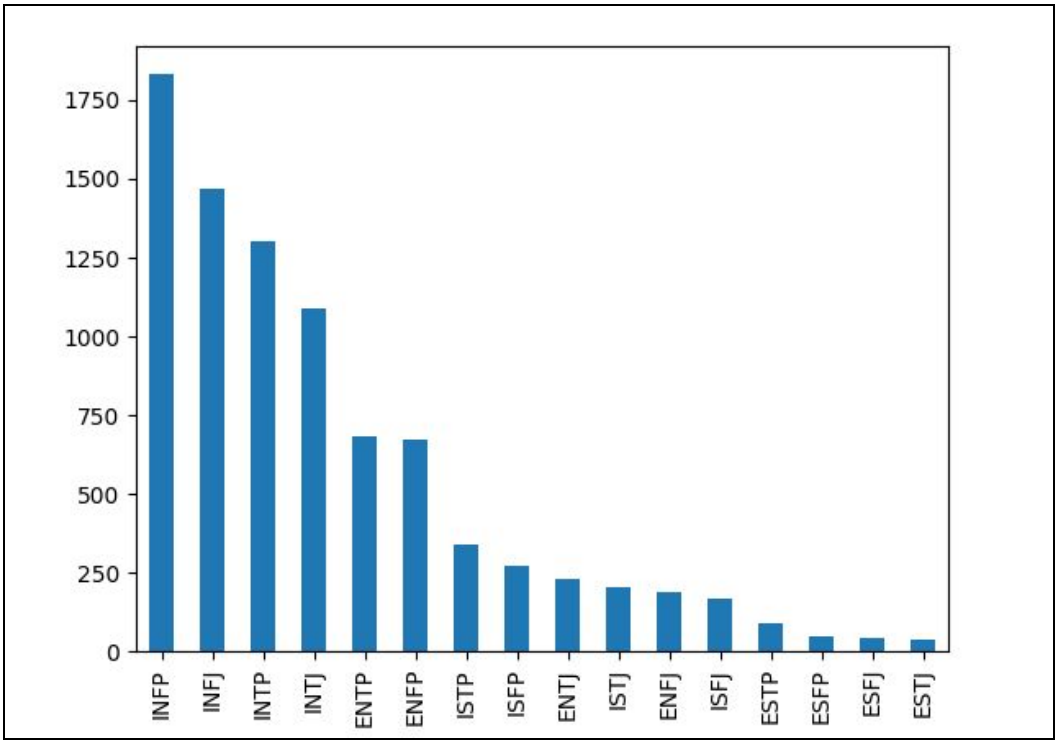
This data was collected from the *PersonalityCafe* forum.

Exploratory Data Analysis

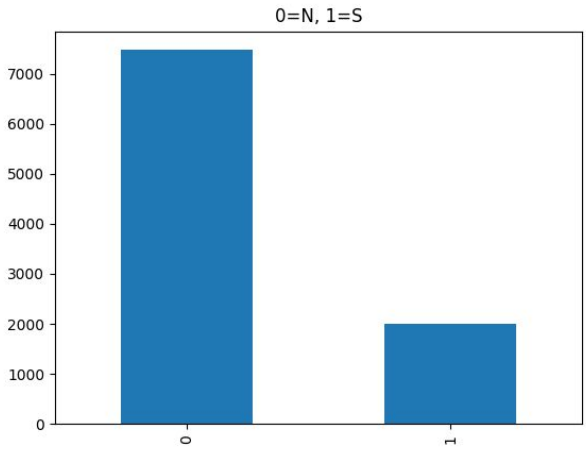
First of all, we can start with some **Exploratory Data Analysis** (**EDA**).

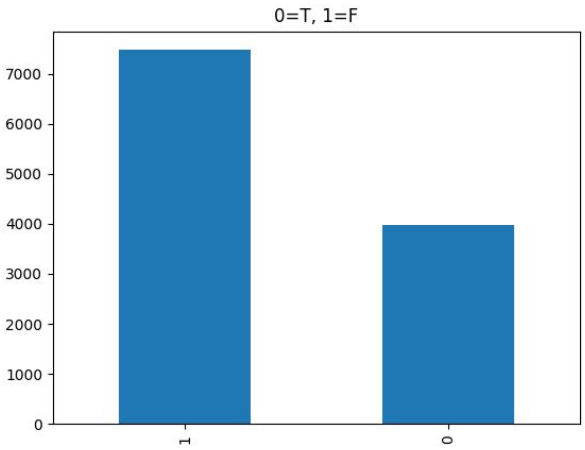
We plot a graph that counts the occurrences of each of the 16 personality types.

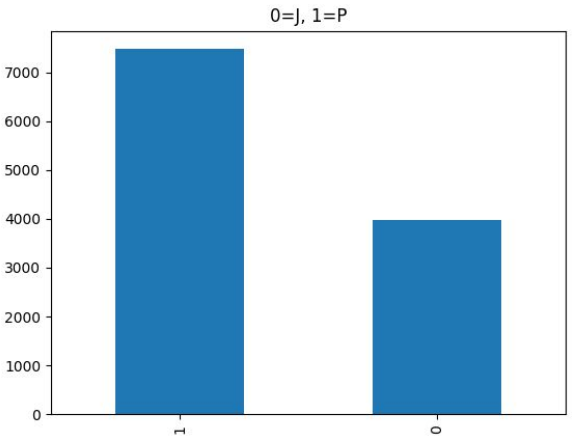
```
types = dataset.type.tolist()
pd.Series(types).value_counts().plot(kind="bar")
```

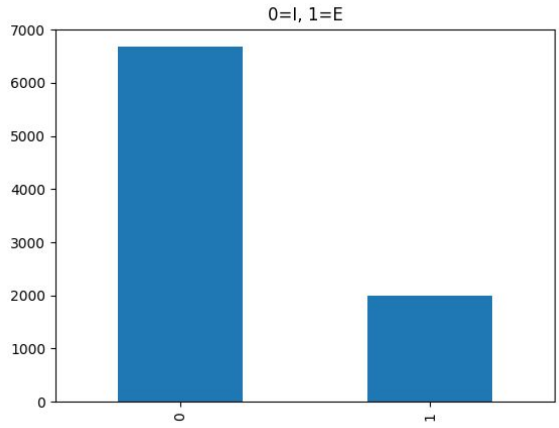


As we can easily see, in this dataset some personality types are much more frequent than others.
We now plot each category to see if they are balanced.









We observe that IE and NS are fairly unbalanced.

Using the *info()* method, we get some info about our dataset. As we can see, **we do not have any NULL value**. Good news! One thing less to think about.

```
++++++++++ dataset.info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8675 entries, 0 to 8674
Data columns (total 2 columns):
type     8675 non-null object
posts    8675 non-null object
dtypes: object(2)
memory usage: 135.7+ KB
None
++++++++++++++++++++++++
```

Since distinguishing between 16 different personality types may be very difficult, **we decided to perform four different binary classifications**; one for each MBTI category. This means that we are going to train four different classifiers.

This reasoning is enforced by the fact that some personality types are very much frequent than others. Thus, we may not have enough data to train the model to classify, for example, an ESTJ or an ESFJ personality type. In others words, our model would be *biased* towards the most frequent personality types, namely INFP, INFJ, INTP.

By reducing our problem to four binary classifiers, we have more data to train our model to classify a single category.

However, as we have seen, the categories IE and NS are unbalanced. So, it may be a good idea to perform a *stratified sampling* when dividing the train and the test set.

The fact that we have no NULL values means that our *data cleaning* will be much more easier.

Now, what is our next step? In order to perform the most precise classification, **we may add new features to our dataset**.

Adding new features

The new features that we added to our dataset are the number of smiles per comment, the number of words per comment, the number of links and so on…
Below, we define the function that adds these new feature in our dataset.

```
def add_features(self):
    # Add new features, such as words per comment, links per comment, images per comment...
    self.df['ellipsis_per_comment'] = self.df['posts'].apply(lambda x: x.count('...') / (x.count("|||") + 1))
    self.df['words_per_comment'] = self.df['posts'].apply(lambda x: x.count(' ') / (x.count("|||") + 1))
    self.df['words'] = self.df['posts'].apply(lambda x: x.count(' '))
    self.df['link_per_comment'] = self.df['posts'].apply(lambda x: x.count('http') / (x.count("|||") + 1))
    self.df['smiles_per_comment'] = self.df['posts'].apply(lambda x: (x.count(':-)') + x.count(':)') + x.count(':-D') + x.count(':D')) / (x.count("|||") + 1))
    self.df['sad'] = self.df['posts'].apply(lambda x: (x.count(':(') + x.count('):') ) / (x.count("|||") + 1))
    self.df['heart'] = self.df['posts'].apply(lambda x: x.count('<3') / (x.count("|||") + 1))
    self.df['smiling'] = self.df['posts'].apply(lambda x: x.count(';)') / (x.count("|||") + 1))
    self.df['exclamation_mark_per_comment'] = self.df['posts'].apply(lambda x: x.count("!") / (x.count("|||") + 1))
    self.df['question_mark_per_comment'] = self.df['posts'].apply(lambda x: x.count("?") / (x.count("|||") + 1))
    self.df['polarity'] = self.df['posts'].apply(lambda x: TextBlob(x).sentiment.polarity)
```

For example, we may expect that an Extrovert (E) person may use more question marks or more smiles.

Now, in order to perform our binary classification, we must also create 4 new columns that identify each personality type. Thus, we created 4 simple (binary) *dummy variables*.

```
# Create 4 more columns for binary classification - LABEL ENCODING, ONE-HOT ENCODING
map1 = {"I": 0, "E": 1}
map2 = {"N": 0, "S": 1}
map3 = {"T": 0, "F": 1}
map4 = {"J": 0, "P": 1}
self.df['IE'] = self.df['type'].astype(str).str[0]
self.df['IE'] = self.df['IE'].map(map1)
self.df['NS'] = self.df['type'].astype(str).str[1]
self.df['NS'] = self.df['NS'].map(map2)
self.df['TF'] = self.df['type'].astype(str).str[2]
self.df['TF'] = self.df['TF'].map(map3)
self.df['JP'] = self.df['type'].astype(str).str[3]
self.df['JP'] = self.df['JP'].map(map4)
```

So, if in the column "NS" we have a "0", this means that we are dealing with an N type.

Post cleaner

Now, we should "clean" our posts. This means that we remove some characters such as @, -, <, > and so on. We also used *NLTK lemmatizer* function

This process is performed by the *post_cleaner()* function.

This function also puts every letter in lowercase.

```
def post_cleaner(self, post):
    post = post.lower()
    post = re.sub(
        r'''(?i)\b((?:https?://|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}/)(?:[^\s()<>]+|\
        '', post, flags=re.MULTILINE)
    puncs1 = ['@', '#', '$', '%', '^', '&', '*', '(', ')', '-', '_', '+', '=', '{', '}',
              "'", ';', ':', '<', '>', '/']

    for punc in puncs1:
        post = post.replace(punc, '')

    puncs2 = [',', '.', '?', '!', '\n']
    for punc in puncs2:
        post = post.replace(punc, ' ')

    post = re.sub('\s+', ' ', post).strip()

    return post
```

Of course, *post_cleaner()* function is executed after *add_features()*, otherwise we would have deleted crucial characters that would not be counted. We can also say that *post_cleaner()* has eliminated a lot of redundancy added with *add_features()*.

Counting words

Here is one of the most crucial parts. Some models may have difficulties in detecting patterns when dealing with words. Thus, we decided to assign each column to a word, so that we can count its occurrences in a person's posts.

In order to do this, we have developed a new dataset composed of 5,000 more columns, each of them representing the occurrences of the most frequent words.

t	NS	i	the	to	a	and	it	of	you	that	is	in	my	be	but	for	have	with	me	im	like	thi	not	are	on	an	your	as	wa	so	do	think	just	if	wl
	0	11	20	11	17	12	4	15	6	7	3	15	5	6	0	9	0	4	2	1	4	2	4	2	5	3	9	4	2	0	2	1	3	1	
	0	56	40	30	24	33	19	18	23	15	7	13	17	9	4	9	11	12	18	16	16	4	3	8	8	9	9	10	10	5	5	2	5	6	
	0	27	19	23	8	18	23	18	13	17	10	14	8	17	5	8	3	4	3	3	4	11	4	2	6	5	10	2	2	4	3	5	2	4	
	0	45	37	29	23	16	13	30	17	20	16	16	10	13	7	3	9	10	8	11	6	5	9	6	4	3	12	10	6	9	2	4	3	8	
	0	31	28	31	22	21	11	16	16	16	17	7	7	15	6	7	12	9	4	0	5	4	8	17	11	3	3	3	11	5	6	8	12	4	
	0	53	35	47	37	28	26	23	26	26	31	22	8	18	11	9	19	9	3	2	7	12	15	17	5	10	5	18	6	4	12	9	3	14	
	0	53	33	22	39	29	13	21	25	16	19	16	21	7	11	11	9	14	10	10	7	16	7	11	12	4	6	6	6	6	7	10	6	6	
	0	64	34	50	26	22	32	25	14	14	18	12	12	10	6	13	5	8	14	3	13	13	12	8	10	3	7	8	12	5	14	10	5	4	
	0	12	25	23	26	24	19	19	3	9	7	11	6	6	7	11	4	5	4	10	2	9	7	2	10	1	0	2	3	4	2	3	3	0	

This process is performed by *transform_df()* along with the *NLTK.tokenize* package.
The output of this function are 4 datasets, one for each of the 4 binary categories. Now, we just need to develop our models and to evaluate them. We start with a SVM model.

SVM Model

To develop this model, we have defined a function *perform_svm()*.

```
315     def perform_svm(x_train, x_test, y_train, y_test):
316         scaling = MinMaxScaler(feature_range=(-1, 1)).fit(x_train)
317         x_train = scaling.transform(x_train)
318         x_test = scaling.transform(x_test)
319         clf = svm.SVC(C=10, kernel='linear', degree=1, gamma='auto')
320         clf.fit(x_train, y_train)
321         #scores = cross_val_score(clf, x_train, x_test, cv=10)
322         score = clf.score(x_test, y_test)
323         return score
```

This function takes as input the predictors and the responses of both the train and the test set.
This is how we call this function:

```
IE_df = pd.read_csv('IE_df.csv')
y = IE_df['IE']
del IE_df['IE']
x_train, x_test, y_train, y_test = train_test_split(IE_df, y, test_size=0.20, random_state=1, stratify=y)
IE_accuracy = perform_svm(x_train, x_test, y_train, y_test)
```

As we can see, the test and the train set are divided in a stratified way, so that we can maintain the right proportions among the responses. The features of our datasets are scaled with the MinMaxScaler function().
The model is fit four times, one for each MBTI type, and it is evaluated using the cross_val_score() function with k=10.
The resulting final accuracies are:

- **IE**: 0.7596541786743516
- **JP**: 0.6737752161383286
- **TF**: 0.7717579250720461
- **NS**: 0.8334293948126801

Which means that we have approximately a 32% overall accuracy.

LSTM

We have also developed an LSTM neural network, executed by the *lstm()* function.

```
271     def lstm(self, type, dropout, max_words, max_len, neurons):
272         self.load_clean_df()
273         X = self.df['posts']
274         Y = self.df[type]
275         le = LabelEncoder()
276         Y = le.fit_transform(Y)
277         Y = Y.reshape(-1, 1)
278         X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15, random_state=1, stratify=Y)
279         tok = Tokenizer(num_words=max_words)
280         tok.fit_on_texts(X_train)
281         sequences = tok.texts_to_sequences(X_train)
282         sequences_matrix = sequence.pad_sequences(sequences, maxlen=max_len)
283         inputs = Input(name='inputs', shape=[max_len])
284         layer = Embedding(max_words, 50, input_length=max_len)(inputs)
285         layer = LSTM(64)(layer)
286         layer = Dense(neurons, name='FC1')(layer)
287         layer = Dense(neurons, name='FC2')(layer)
288         layer = Dense(neurons, name='FC3')(layer)
289         layer = LeakyReLU(alpha=0.1)(layer)
290         layer = Dropout(dropout)(layer)
291         layer = Dense(1, name='out_layer')(layer)
292         layer = Activation('sigmoid')(layer)
293         model = Model(inputs=inputs, outputs=layer)
294         model.summary()
295         model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])
296         model.fit(sequences_matrix, Y_train, batch_size=128, epochs=10,
297                   validation_split=0.2, callbacks=[EarlyStopping(monitor='val_loss', min_delta=0.0001)])
298         test_sequences = tok.texts_to_sequences(X_test)
299         test_sequences_matrix = sequence.pad_sequences(test_sequences, maxlen=max_len)
300         accr = model.evaluate(test_sequences_matrix, Y_test)
301         with open('report.txt', 'a') as file:
302             file.write(f'Type: {type}\nDropout: {dropout} \nMax words: {max_words} \nMax len: {max_len} \nNeurons: {neurons}\n'
303                        f'Loss: {accr[0]}\n  Accuracy: {accr[1]}\n\n\n\n')
```

The neural network is composed by one LSTM layer followed by three dense layers. The last layer, named "out_layer", contains just one neuron.

We used a tokenizer to turn the text into a sequence, *Adam* optimizer, *Leaky ReLU* and *Sigmoid* activation function. Our loss function is *binary cross-entropy*.

We have tried several combinations of the parameters *max_len*, *max_words* and for the *dropout*. The best combination is the one that results in the highest accuracy. Each binary classification resulted in different configurations.

For instance, the best parameters for the IE prediction are:

- Dropout: 0.5
- Max words: 1600
- Max len: 50
- **Accuracy: 0.7718893885612488**

The resulting final accuracies are:

- **IE**: 0.7718893885612488
- **JP**: 0.6044546961784363
- **TF**: 0.6666666865348816
- **NS**: 0.861751139163971

Which means that we have approximately a 27% overall accuracy.

This is the summary of the neural network created for the NS prediction:

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inputs (InputLayer)          (None, 50)                0
_________________________________________________________________
embedding_1 (Embedding)      (None, 50, 50)            95000
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                29440
_________________________________________________________________
FC1 (Dense)                  (None, 400)               26000
_________________________________________________________________
FC2 (Dense)                  (None, 400)               160400
_________________________________________________________________
FC3 (Dense)                  (None, 400)               160400
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 400)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 400)               0
_________________________________________________________________
out_layer (Dense)            (None, 1)                 401
_________________________________________________________________
activation_1 (Activation)    (None, 1)                 0
=================================================================
Total params: 471,641
Trainable params: 471,641
Non-trainable params: 0
```

21:49

Conclusions

An 32% overall accuracy in the SVM model, calculated multiplying the four accuracies, may appear low; but in reality, given the complexity of our task, it is a discrete result. Considering that the most common personality in the world is “ISFJ” (14% of population), using our algorithm is way more efficient than guessing. Moreover, if only one personality trait is involved, the accuracy is good. We were genuinely surprised that SVMs could be so performing for this task.

On the other hand, the RNN model only gave us a 27% accuracy, which is a bit disappointing, given the popularity this model has for text classification tasks.
Anyway, we need to consider that the exploration of all possible hyperparameter combinations takes a long time, and we were not able to explore everything.

This last model can be considerably improved, probably up to a 40% accuracy, as stated in some more complicated models involving RNNs we found during our online research.