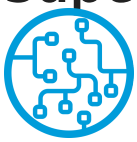
Supervised Machine Learning Pipeline



*In this project, we build a supervised learning pipeline to **classify political tweets based on their author**. The dataset consists of labeled tweets from political figures and organizations, and the goal is to predict whether a tweet belongs to a conservative (e.g., `realDonaldTrump`, `GOP`) or liberal (e.g., `JoeBiden`, `TheDemocrats`) source.*

The classification task is binary, and we apply several standard machine learning techniques including preprocessing, feature extraction using TF-IDF, and classification with Support Vector Machines (SVMs).

In [ ]:
```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
import sklearn
import string
import re # helps you filter urls
from IPython.display import display, Latex, Markdown
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import time
```

Classifying tweets

In this problem, you will be analyzing Twitter data related to the 2016 US Presidential election extracted using the Twitter API. The data contains tweets posted by the following six Twitter accounts: `realDonaldTrump, mike_pence, GOP, HillaryClinton, timkaine, TheDemocrats`

For every tweet, there are two pieces of information:

- `screen_name`: the Twitter handle of the user tweeting and
- `text`: the content of the tweet.

The tweets have been divided into two parts - train and test available as CSV files. For train, both the `screen_name` and `text` attributes were provided but for test, `screen_name` is **hidden**.

The overarching goal of the problem is to **"predict"** the political inclination (Republican/Democratic) of the Twitter user from one of his/her tweets. The ground truth (i.e., true class labels) is determined from the `screen_name` of the tweet as follows

- `realDonaldTrump, mike_pence, GOP` are Republicans
- `HillaryClinton, timkaine, TheDemocrats` are Democrats

Thus, this is a binary classification problem.

The problem proceeds in three stages:

- **A. Text processing**: We will clean up the raw tweet text using the various functions offered by the nltk package.
- **B. Feature construction**: In this part, we will construct bag-of-words feature vectors and training labels from the processed text of tweets and the `screen_name` columns respectively.
- **C. Classification**: Using the features derived, we will use sklearn package to learn a model which classifies the tweets as desired.

Install necessary `nltk` packages using:

In [2]:
```
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('punkt')
nltk.download('punkt_tab')
# Verify that the following commands work before moving on.

lemmatizer=nltk.stem.wordnet.WordNetLemmatizer()
stopwords=nltk.corpus.stopwords.words('english')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to
[nltk_data]     C:\Users\mnusa\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

A. Text Processing

First we will fill in the following function which processes and tokenizes raw text. The generated list of tokens should meet the following specifications:

1. The tokens must all be in lower case.
2. The tokens should appear in the same order as in the raw text.
3. The tokens must be in their lemmatized form. If a word cannot be lemmatized (i.e, you get an exception), simply catch it and ignore it. These words will not appear in the token list.
4. The tokens must not contain any punctuations. Punctuations should be handled as follows: (a) Apostrophe of the form `'s` must be ignored. e.g., `She's` becomes `she`. (b) Other apostrophes should be omitted. e.g, `don't` becomes `dont`. (c) Words must be broken at the hyphen and other punctuations.
5. The tokens must not contain any part of a url.

Part 1:

In [3]:
```
# Convert part of speech tag from nltk.pos_tag to word net compatible format
# Simple mapping based on first letter of return tag to make grading consistent
# Everything else will be considered noun 'n'
posMapping = {
# "First_Letter by nltk.pos_tag":"POS_for_lemmatizer"
    "N":'n',
    "V":'v',
    "J":'a',
    "R":'r'
}

def process(text, lemmatizer=nltk.stem.wordnet.WordNetLemmatizer()):
    """ Normalizes case and handles punctuation
    Inputs:
        text: str: raw text
        lemmatizer: an instance of a class implementing the lemmatize() method
                    (the default argument is of type nltk.stem.wordnet.WordNetLemma
    Outputs:
        list(str): tokenized text
    """
    # Step 1: Convert to lower case
    text = text.lower()

    # Step 2: Remove URLs
```

```
    text = re.sub(r'http:/\S+|www\S+|https:/\S+', '', text, flags=re.MULTILINE)

    # Step 3: Remove or handle punctuation
    # Handle specific punctuation cases and remove others
    text = re.sub(r"'s", '', text)  # Remove 's
    text = re.sub(r"'", '', text)  # Replace other apostrophes with 'dont'
    text = re.sub(r'[%s]' % re.escape(string.punctuation), ' ', text) # Replace any

    # Step 4: Tokenize the text
    tokens = nltk.word_tokenize(text)

    # Step 5: Lemmatize tokens based on POS
    lemmatized_tokens = []
    pos_tags = nltk.pos_tag(tokens)  # Get POS tags

    for word, tag in pos_tags:
        # Get the first letter of the POS tag
        first_letter = tag[0]
        pos = posMapping.get(first_letter, 'n')  # Default to noun if not found
        try:
            lemmatized_word = lemmatizer.lemmatize(word, pos=pos)
            lemmatized_tokens.append(lemmatized_word)
        except Exception:
            continue  # Ignore words that cannot be lemmatized

    return lemmatized_tokens
```

Test the above function as follows; we want to make the test strings as exhaustive as possible.

In [4]:
```
print(process("I'm doing well! How about you?"))
# ['im', 'do', 'well', 'how', 'about', 'you']

print(process("Education is the ability to listen to almost anything without losing
# ['education', 'be', 'the', 'ability', 'to', 'listen', 'to', 'almost', 'anything',

print(process("been had done languages cities mice"))
# ['be', 'have', 'do', 'language', 'city', 'mice']

print(process("It's hilarious. Check it out http://t.co/dummyurl"))
# ['it', 'hilarious', 'check', 'it', 'out']

print(process("See it Sunday morning at 8:30a on RTV6 and our RTV6 app. http:…"))
# ['see', 'it', 'sunday', 'morning', 'at', '8', '30a', 'on', 'rtv6', 'and', 'our',
# Here '…' is a special unicode character not in string.punctuation and it is still
```

```
['im', 'do', 'well', 'how', 'about', 'you']
['education', 'be', 'the', 'ability', 'to', 'listen', 'to', 'almost', 'anything', 'w
ithout', 'lose', 'your', 'temper', 'or', 'your', 'self', 'confidence']
['be', 'have', 'do', 'language', 'city', 'mice']
['it', 'hilarious', 'check', 'it', 'out']
['see', 'it', 'sunday', 'morning', 'at', '8', '30a', 'on', 'rtv6', 'and', 'our', 'rt
v6', 'app', 'http', '…']
```

Part 2:

We will now use the `process()` function we implemented to convert the pandas dataframe we just loaded from tweets_train.csv file. Our function should be able to handle any data frame which contains a column called `text`. The data frame we return should replace every string in `text` with the result of `process()` and retain all other columns as such. We do not change the order of rows/columns.

Before writing `process_all()`, we will load the data into a DataFrame and look at its format:

In [5]:
```
tweets = pd.read_csv("tweets_train.csv", na_filter=False)
display(tweets.head())
```

	screen_name	text
0	GOP	RT @GOPconvention: #Oregon votes today. That m...
1	TheDemocrats	RT @DWStweets: The choice for 2016 is clear: W...
2	HillaryClinton	Trump's calling for trillion dollar tax cuts f...
3	HillaryClinton	.@TimKaine's guiding principle: the belief tha...
4	timkaine	Glad the Senate could pass a #THUD / MilCon / ...

In [6]:
```
def process_all(df, lemmatizer=nltk.stem.wordnet.WordNetLemmatizer()):
    """ process all text in the dataframe using process() function.
    Inputs
        df: pd.DataFrame: dataframe containing a column 'text' loaded from the CSV
        lemmatizer: an instance of a class implementing the lemmatize() method
                    (the default argument is of type nltk.stem.wordnet.WordNetLemma
    Outputs
        pd.DataFrame: dataframe in which the values of text column have been change
                      the output from process() function. Other columns are unaff
    """
    # Apply the process function to the 'text' column
    df['text'] = df['text'].apply(lambda x: process(x, lemmatizer))
    return df
```

In [7]:
```
# test your code
processed_tweets = process_all(tweets)
print(processed_tweets.head())

#       screen_name                                               text
# 0             GOP  [rt, gopconvention, oregon, vote, today, that,...
# 1    TheDemocrats  [rt, dwstweets, the, choice, for, 2016, be, cl...
```

```
# 2  HillaryClinton  [trump, call, for, trillion, dollar, tax, cut,...
# 3  HillaryClinton  [timkaine, guide, principle, the, belief, that...
# 4         timkaine  [glad, the, senate, could, pass, a, thud, milc...
```

	screen_name	text
0	GOP	[rt, gopconvention, oregon, vote, today, that,...
1	TheDemocrats	[rt, dwstweets, the, choice, for, 2016, be, cl...
2	HillaryClinton	[trump, call, for, trillion, dollar, tax, cut,...
3	HillaryClinton	[timkaine, guide, principle, the, belief, that...
4	timkaine	[glad, the, senate, could, pass, a, thud, milc...

B. Feature Construction

The next step is to derive feature vectors from the tokenized tweets. In this section, we will be constructing a bag-of-words TF-IDF feature vector. But before that, the number of possible words is prohibitively large and not all of them may be useful for our classification task. We need to determine which words to retain, and which to omit. A common heuristic is to construct a frequency distribution of words in the corpus and prune out the head and tail of the distribution. The intuition of the above operation is as follows.

Very common words (i.e. stopwords) add almost no information regarding similarity of two pieces of text. Similarly with very rare words. NLTK has a list of in-built stop words which is a good substitute for head of the distribution. We will consider a word rare if it occurs only in a single document (row) in whole of `tweets_train.csv`.

Part 3:

Construct a sparse matrix of features for each tweet with the help of `sklearn.feature_extraction.text.TfidfVectorizer` (documentation here). We will need to pass a parameter `min_df=2` to filter out the words occuring only in one document in the whole training set. Remember to ignore the stop words as well. We must leave other optional parameters (e.g., `vocab`, `norm`, etc) at their default values; but we may need to use parameters like `lowercase` and `tokenizer` to handle `processed_tweets` that is a `list` of tokens (not raw text).

In [8]:
```
import sklearn.feature_extraction

def create_features(processed_tweets, stop_words):
    """ creates the feature matrix using the processed tweet text
    Inputs:
        processed_tweets: pd.DataFrame: processed tweets read from train/test csv f
```

```
        stop_words: list(str): stop_words by nltk stopwords (after processing)
    Outputs:
        sklearn.feature_extraction.text.TfidfVectorizer: the TfidfVectorizer object
            we need this to tranform test tweets in the same way as train tweets
        scipy.sparse.csr.csr_matrix: sparse bag-of-words TF-IDF feature matrix
    """
    # sort stop_words set first

    # Initialize the TfidfVectorizer with the required parameters
    vectorizer = sklearn.feature_extraction.text.TfidfVectorizer(
        lowercase=False,  # Convert to lowercase
        tokenizer=lambda x: x,  # The input is already tokenized
        stop_words=list(sorted(stop_words)),  # Ignore the stop words
        min_df=2  # Filter out words that occur in less than 2 documents
    )

    # Fit the vectorizer to the processed tweets and transform the text into a spar
    feature_matrix = vectorizer.fit_transform(processed_tweets['text'])

    return vectorizer, feature_matrix
```

In [9]:

```
# execute this code

# It is recommended to process stopwords according to our data cleaning rules
processed_stopwords = set(np.concatenate([process(word) for word in stopwords]))
(tfidf, X) = create_features(processed_tweets, processed_stopwords)
# Ignore warning
tfidf, X
# Output (should be similar):
# (TfidfVectorizer(lowercase=False, min_df=2,
#                  stop_words={'a', 'about', 'above', 'after', 'again', 'against',
#                              'ain', 'all', 'an', 'and', 'any', 'aren', 'arent',
#                              'at', 'be', 'because', 'before', 'below', 'between',
#                              'both', 'but', 'by', 'can', 'couldn', 'couldnt',
#                              'd', 'didn', 'didnt', 'do', 'doesn', ...},
#                  tokenizer=<function create_features.<locals>.<lambda> at 0x7fd40
#  <17298x8114 sparse matrix of type '<class 'numpy.float64'>'
#       with 170355 stored elements in Compressed Sparse Row format>)
```

```
c:\Users\mnusa\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\fea
ture_extraction\text.py:521: UserWarning: The parameter 'token_pattern' will not be
used since 'tokenizer' is not None'
  warnings.warn(
c:\Users\mnusa\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\fea
ture_extraction\text.py:406: UserWarning: Your stop_words may be inconsistent with y
our preprocessing. Tokenizing the stop words generated tokens ['b', 'c', 'e', 'f',
'g', 'h', 'j', 'l', 'n', 'p', 'r', 'u', 'v', 'w'] not in stop_words.
  warnings.warn(
```

Out[9]:
```
(TfidfVectorizer(lowercase=False, min_df=2,
                 stop_words=[np.str_('a'), np.str_('about'), np.str_('above'),
                             np.str_('after'), np.str_('again'),
                             np.str_('against'), np.str_('ain'), np.str_('all'),
                             np.str_('an'), np.str_('and'), np.str_('any'),
                             np.str_('aren'), np.str_('arent'), np.str_('at'),
                             np.str_('be'), np.str_('because'),
                             np.str_('before'), np.str_('below'),
                             np.str_('between'), np.str_('both'), np.str_('but'),
                             np.str_('by'), np.str_('can'), np.str_('couldn'),
                             np.str_('couldnt'), np.str_('d'), np.str_('didn'),
                             np.str_('didnt'), np.str_('do'), np.str_('doesn'),
...],
                 tokenizer=<function create_features.<locals>.<lambda> at 0x000001
E4D620D3A0>),
 <Compressed Sparse Row sparse matrix of dtype 'float64'
        with 168110 stored elements and shape (17298, 8101)>)
```

Part 4:

Also for each tweet, assign a class label (0 or 1) using its `screen_name`. Use 0 for realDonaldTrump, mike_pence, GOP and 1 for the rest.

In [10]:
```
def create_labels(processed_tweets):
    """ creates the class labels from screen_name
    Inputs:
        processed_tweets: pd.DataFrame: tweets read from train file, containing the
    Outputs:
        numpy.ndarray(int): dense binary numpy array of class labels
    """
    # Define the screen names that should be labeled as 0
    label_0_names = {'realDonaldTrump', 'mike_pence', 'GOP'}

    # Create the labels by checking if each screen_name is in the label_0_names set
    labels = processed_tweets['screen_name'].apply(lambda name: 0 if name in label_

    # Convert the result to a numpy array
    return labels.to_numpy(dtype=int)
```

In [11]:
```
# execute this code

y = create_labels(processed_tweets)
y
# 0       0
# 1       1
# 2       1
# 3       1
# 4       1
```

```
#         ..
# 17293    0
# 17294    0
# 17295    0
# 17296    1
# 17297    0
# Name: screen_name, Length: 17298, dtype: int32
```

Out[11]: `array([0, 1, 1, ..., 0, 1, 0])`

C. Classification

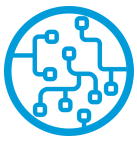
And finally, we are ready to put things together and learn a model for the classification of tweets. The classifier you will be using is `sklearn.svm.SVC` (Support Vector Machine).

At the heart of SVMs is the concept of kernel functions, which determines how the similarity/distance between two data points in computed. `sklearn` 's SVM provides four kernel functions: `linear` , `poly` , `rbf` , `sigmoid` (details here) but we can also implement our own distance function and pass it as an argument to the classifier.

Through the various functions we implement in this part, we will be able to learn a classifier, score a classifier based on how well it performs, use it for prediction tasks and compare it to a baseline.

Specifically, we will carry out the following tasks (Parts 5 - 9) in order:

1. Implement and evaluate a simple baseline classifier MajorityLabelClassifier.
2. Implement the `learn_classifier()` function assuming `kernel` is always one of { `linear` , `poly` , `rbf` , `sigmoid` }.
3. Implement the `evaluate_classifier()` function which scores a classifier based on accuracy of a given dataset.
4. Implement `best_model_selection()` to perform cross-validation by calling `learn_classifier()` and `evaluate_classifier()` for different folds and determine which of the four kernels performs the best.
5. Go back to `learn_classifier()` and fill in the best kernel.



Part 5:

To determine whether our classifier is performing well, we need to compare it to a baseline classifier. A baseline is generally a simple or trivial classifier and our classifier should beat the

baseline in terms of a performance measure such as accuracy. Therefore, we will implement a classifier called `MajorityLabelClassifier` that always predicts the class equal to **mode** of the labels (i.e., the most frequent label) in training data.

In [12]:
```
# Skeleton of MajorityLabelClassifier is consistent with other sklearn classifiers
class MajorityLabelClassifier():
    """
    A classifier that predicts the mode of training labels
    """
    def __init__(self):
        """
        Initialize your parameter here
        """
        self.mode_label = None  # To store the mode of labels

    def fit(self, X, y):
        """
        Implement fit by taking training data X and their labels y and finding the
        i.e. store your learned parameter
        """
        # Count occurrences of each label in y
        label_counts = {}
        for label in y:
            if label in label_counts:
                label_counts[label] += 1
            else:
                label_counts[label] = 1

        # Find the label with the maximum count (the mode)
        self.mode_label = max(label_counts, key=label_counts.get)

    def predict(self, X):
        """
        Implement to give the mode of training labels as a prediction for each data
        return labels
        """
        # Create an array with the same length as X, filled with the mode label
        return np.full(X.shape[0], self.mode_label)

# Report the accuracy of your classifier by comparing the predicted label of each e
baselineClf = MajorityLabelClassifier()
# Use fit and predict methods to get predictions and compare it with the true label
baselineClf.fit(X, y) # Fit the classifier on training data
y_pred = baselineClf.predict(X) # Predict on training data
accuracy = np.mean(y_pred == y) # Calculate accuracy by comparing predictions with
# print(training accuracy) should give 0.5001734304543878
print("Training Accuracy: ", accuracy)
```

```
Training Accuracy:  0.5001734304543878
```

Part 6:

Implement the `learn_classifier()` function assuming `kernel` is always one of { `linear` , `poly` , `rbf` , `sigmoid` }. Stick to default values for any other optional parameters.

In [13]:
```
def learn_classifier(X_train, y_train, kernel):
    """ learns a classifier from the input features and labels using the kernel fun
    Inputs:
        X_train: scipy.sparse.csr.csr_matrix: sparse matrix of features, output of
        y_train: numpy.ndarray(int): dense binary vector of class labels, output of
        kernel: str: kernel function to be used with classifier. [linear|poly|rbf|s
    Outputs:
        sklearn.svm.SVC: classifier learnt from data
    """

    # Initialize SVC with specified kernel
    classifier = SVC(kernel=kernel)

    # Fit the classifier on the training data
    classifier.fit(X_train, y_train)

    return classifier
```

In [14]:
```
# execute code
classifier = learn_classifier(X, y, 'linear')
```

Part 7:

Now that we know how to learn a classifier, the next step is to evaluate it, ie., characterize how good its classification performance is. This step is necessary to select the best model among a given set of models, or even tune hyperparameters for a given model.

There are two questions that come to mind:

1. **What data to use?**

    - **Validation Data**: The data used to evaluate a classifier is called **validation data** (or hold-out data), and it is usually different from the data used for training. The model or hyperparameter with the best performance in the held out data is chosen. This approach is relatively fast and simple but vulnerable to biases found in validation set.
    - **Cross-validation**: This approach divides the dataset in k groups (so, called k-fold cross-validation). One of group is used as test set for evaluation and other groups

as training set. The model or hyperparameter with the best average performance across all k folds is chosen. For this question you will perform 4-fold cross validation to determine the best kernel. We will keep all other hyperparameters default for now. This approach provides robustness toward biasness in validation set. However, it takes more time.

2. **And what metric?** There are several evaluation measures available in the literature (e.g., accuracy, precision, recall, F-1,etc) and different fields have different preferences for specific metrics due to different goals. We will go with accuracy. According to wiki, **accuracy** of a classifier measures the fraction of all data points that are correctly classified by it; it is the ratio of the number of correct classifications to the total number of (correct and incorrect) classifications. `sklearn.metrics` provides a number of performance metrics.

Now, we will implement the following function.

In [15]:
```
def evaluate_classifier(classifier, X_validation, y_validation):
    """ evaluates a classifier based on a supplied validation data
    Inputs:
        classifier: sklearn.svm.classes.SVC: classifer to evaluate
        X_validation: scipy.sparse.csr.csr_matrix: sparse matrix of features
        y_validation: numpy.ndarray(int): dense binary vector of class labels
    Outputs:
        double: accuracy of classifier on the validation data
    """
    # Use the classifier to predict labels on the validation set
    y_pred = classifier.predict(X_validation)

    # Calculate and return accuracy
    accuracy = accuracy_score(y_validation, y_pred)
    return accuracy
```

In [16]:
```
# test your code by evaluating the accuracy on the training data
accuracy = evaluate_classifier(classifier, X, y)
print(accuracy)
# should give around 0.9545034107989363
```

```
0.9543877904960111
```

Part 8:

Now it is time to decide which kernel works best by using the cross-validation technique. We will split the training data into 4-folds (75% training and 25% validation) by shuffling randomly. For each kernel, record the average accuracy for all folds and determine the best

classifier. Since our dataset is balanced (both classes are in almost equal propertion), `sklearn.model_selection.KFold` doc can be used for cross-validation.

In [17]:
```
kf = sklearn.model_selection.KFold(n_splits=4, random_state=1, shuffle=True)
kf
```

Out[17]:
```
KFold(n_splits=4, random_state=1, shuffle=True)
```

Then use the following code to determine which classifier is the best.

In [18]:
```
def best_model_selection(kf, X, y):
    """
    Select the kernel giving best results using k-fold cross-validation.
    Other parameters should be left default.
    Input:
    kf (sklearn.model_selection.KFold): kf object defined above
    X (scipy.sparse.csr.csr_matrix): training data
    y (array(int)): training labels
    Return:
    best_kernel (string)
    """
    # Dictionary to store average accuracies for each kernel
    kernel_accuracies = {}

    for kernel in ['linear', 'rbf', 'poly', 'sigmoid']:
        accuracies = []
        # Perform cross-validation
        for train_index, val_index in kf.split(X):
            X_train, X_val = X[train_index], X[val_index]
            y_train, y_val = y[train_index], y[val_index]

            # Timing the classifier training
            start_time = time.time()
            classifier = learn_classifier(X_train, y_train, kernel)
            train_time = time.time() - start_time

            # Timing the evaluation
            start_time = time.time()
            accuracy = evaluate_classifier(classifier, X_val, y_val)
            eval_time = time.time() - start_time

            accuracies.append(accuracy)

            # Print timings for this fold
            print(f"Kernel: {kernel}, Train Time: {train_time:.4f}s, Eval Time: {ev

        # Calculate and store the average accuracy for the kernel
        avg_accuracy = sum(accuracies) / len(accuracies)
        kernel_accuracies[kernel] = avg_accuracy

    # Determine the kernel with the highest average accuracy
    best_kernel = max(kernel_accuracies, key=kernel_accuracies.get)

    return best_kernel
```

```
# small_X = X[:1000]  # Take the first 1000 samples
# small_y = y[:1000]
# best_kernel = best_model_selection(kf, small_X, small_y)
# print("Best kernel for small dataset:", best_kernel)
# print(X.shape[0])

# Test your code
best_kernel = best_model_selection(kf, X, y)
print("Best kernel:", best_kernel)
```

```
Kernel: linear, Train Time: 5.4251s, Eval Time: 1.2519s
Kernel: linear, Train Time: 5.3891s, Eval Time: 1.2485s
Kernel: linear, Train Time: 5.3234s, Eval Time: 1.2482s
Kernel: linear, Train Time: 5.3654s, Eval Time: 1.2332s
Kernel: rbf, Train Time: 18.6579s, Eval Time: 2.4710s
Kernel: rbf, Train Time: 18.4523s, Eval Time: 2.4286s
Kernel: rbf, Train Time: 18.3242s, Eval Time: 2.4637s
Kernel: rbf, Train Time: 18.4943s, Eval Time: 2.4218s
Kernel: poly, Train Time: 28.7439s, Eval Time: 2.8806s
Kernel: poly, Train Time: 29.1562s, Eval Time: 2.9194s
Kernel: poly, Train Time: 28.7404s, Eval Time: 2.9380s
Kernel: poly, Train Time: 29.3061s, Eval Time: 2.8958s
Kernel: sigmoid, Train Time: 5.6008s, Eval Time: 1.2325s
Kernel: sigmoid, Train Time: 5.5321s, Eval Time: 1.2196s
Kernel: sigmoid, Train Time: 5.7141s, Eval Time: 1.2219s
Kernel: sigmoid, Train Time: 5.5497s, Eval Time: 1.2323s
Best kernel: poly
```

Results

4 mins 25.1 sec runtime

Kernel: linear, Train Time: 5.4251s, Eval Time: 1.2519s
Kernel: linear, Train Time: 5.3891s, Eval Time: 1.2485s
Kernel: linear, Train Time: 5.3234s, Eval Time: 1.2482s
Kernel: linear, Train Time: 5.3654s, Eval Time: 1.2332s

Kernel: rbf, Train Time: 18.6579s, Eval Time: 2.4710s
Kernel: rbf, Train Time: 18.4523s, Eval Time: 2.4286s
Kernel: rbf, Train Time: 18.3242s, Eval Time: 2.4637s
Kernel: rbf, Train Time: 18.4943s, Eval Time: 2.4218s

Kernel: poly, Train Time: 28.7439s, Eval Time: 2.8806s
Kernel: poly, Train Time: 29.1562s, Eval Time: 2.9194s
Kernel: poly, Train Time: 28.7404s, Eval Time: 2.9380s
Kernel: poly, Train Time: 29.3061s, Eval Time: 2.8958s

Kernel: sigmoid, Train Time: 5.6008s, Eval Time: 1.2325s
Kernel: sigmoid, Train Time: 5.5321s, Eval Time: 1.2196s

Kernel: sigmoid, Train Time: 5.7141s, Eval Time: 1.2219s
Kernel: sigmoid, Train Time: 5.5497s, Eval Time: 1.2323s\

Best kernel: poly

Part 9

We're almost done! Now it's time to write a nice little wrapper function that will use our model to classify unlabeled tweets from tweets_test.csv file.

In [19]:
```
def classify_tweets(tfidf, classifier, unlabeled_tweets):
    """ predicts class labels for raw tweet text
    Inputs:
        tfidf: sklearn.feature_extraction.text.TfidfVectorizer: the TfidfVectorizer
        classifier: sklearn.svm.SVC: classifier learned
        unlabeled_tweets: pd.DataFrame: tweets read from tweets_test.csv
    Outputs:
        numpy.ndarray(int): dense binary vector of class labels for unlabeled tweet
    """
    # Process the unlabeled tweets
    unlabeled_tweets_processed = process_all(unlabeled_tweets)

    # Use the tfidf vectorizer to transform the raw text of the unlabeled tweets
    X_unlabeled = tfidf.transform(unlabeled_tweets_processed['text'])

    # Use the trained classifier to predict class labels for the unlabeled tweets
```

```
    y_pred = classifier.predict(X_unlabeled)

    return y_pred
```

In [20]:
```
# Fill in best classifier in your function and re-trian your classifier using all t
# Get predictions for unlabelled test data

classifier = learn_classifier(X, y, best_kernel)
unlabeled_tweets = pd.read_csv("tweets_test.csv", na_filter=False)
y_pred = classify_tweets(tfidf, classifier, unlabeled_tweets)
```

In [21]:
```
# Retrain the classifier on the entire training dataset
classifier = learn_classifier(X, y, best_kernel)

# Calculate training accuracy to assess the model's performance on the training dat
y_train_pred = classifier.predict(X)
training_accuracy = np.mean(y_train_pred == y)  # Compare predictions on training d
print("Training Accuracy: ", training_accuracy)
```

```
Training Accuracy:  0.9968782518210197
```

Final SVM Accuracy

Our SVM classifier performed better than the baseline classifier; the baseline classifier achieved a training accuracy of approximately **50.02%**, while our SVM classifier achieved a training accuracy of **99.69%**. This indicates an improvement in predictive performance on the training data, showing that the SVM classifier is better at distinguishing between classes in the dataset compared to the baseline model.

Image Credit: Oksana Latysheva, CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0, via Wikimedia Commons