Machine Learning Final Exam 2020

Ciarán Donegan
16322253

January 2021

1. Game review text analysis.

Text preprocessing

The dataset contains 5000 reviews of a game and we are interested in predicting, based on the review text, the review polarity (up voted or not) and if the review is for an early access version of the game.

The first problem to address is that much of the review text is not written in English. While it may still be possible to tokenize the text (split up on white spaces, puncuation, etc) and let a machine learning model blindly learn associations between whatever language the review is in and the output prediction (review polarity and early access), this will lead to a huge number of input features and a very small number of examples of each feature. Furthermore, languages like Russian and Chinese, don't use white space and punctuation the way we do in English, so this would be a very bad approach to take.

Therefore it was necessary to convert all the text to English. This is very simple to achieve using Google Translator API in python. There is a the fear that Google translate will mess up the meaning of the review through a bad translation, but there's not much that can be done to avoid this.

```
from google_trans_new import google_translator

with open(file, "rb") as f:
    for item in json_lines.reader(f):
            english = translator.translate(item['text'])
```

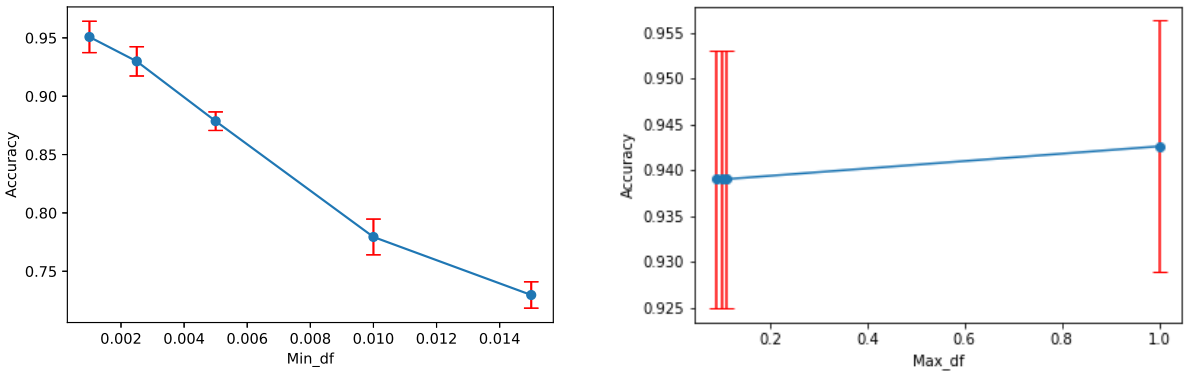
The next problem to address is that some of the reviews aren't at all useful. Reviews like "¡3", "...." and "." which contain no words, only non alphanumeric characters should be ignored. By ignoring these reviews, I reduce the size of my dataset from 5000 down to 4902.

Next, I want to tokenize the review text into words and these words will be treated as the input features in to my model. This results in 15541 individual features (unique words). This is too many so I want to remove redundant features that carry no information.

Some words, called stop words, are very common and contain little information (e.g. I, me, at, the, on, etc.). By ignoring these words I reduce the number of features down to 15251 i.e. I remove 290 features.

Other words in the dataset may be very rare and occur only once in total. These rare words should also be removed because they have little predictive power. To do this I set a minimum frequency threshold (`min_df`) when vectorizing the words. E.g. If I set `min_df` to be 0.01 it will ignore any words that occur in last than 1% of the reviews. I use cross validation, with an SVM model (I compare three models, logistic regression, SVM and fully connected NN which I discuss later) to choose the best parameter to use here. See figure1a. It turns out that selecting a very low `min_df` works best. In fact, ignoring the data frequency and including all words performs even better, but I aim to reduce the number of features in my model so I may have to tradeoff accuracy very slightly. I choose a `min_df` of 0.001 which leaves just 3570 features.

I also want to ignore very common words. Words like "game" which may occur very often, provide little useful information. But through cross validation I found this to have little effect on the perfromance (figure 1b). Setting a `max_df` of 0.1, which would ignore words that occur in more than 10% of all reviews results in just 7 additional words being ignored but also degrades performance a little, so it is not worth it. I choose not to ignore any overly common words by setting `max_df` to 1.0 i.e. 100%



(a) Minimum data frequency cross validation

(b) Maximum data frequency cross validation

Figure 1: Selecting a low `min_df` and a high `max_df` results in better performance. This corresponds to ignoring the least number of features possible but will also increase the number of unncessary features.

Instead of just using single words as tokens, I can also use pairs of words. This will preserve the context of the words. For example "not", "good" and "not good" provide very different meanings. This would increase the number of features by n^2, but I again apply `min_df`=0.001 to remove very rare words and pairs of words. The results with 4272 features (single words and pairs of words) in total. The increased performance of using pairs of tokens is displayed in the cross validation results in figure 2.

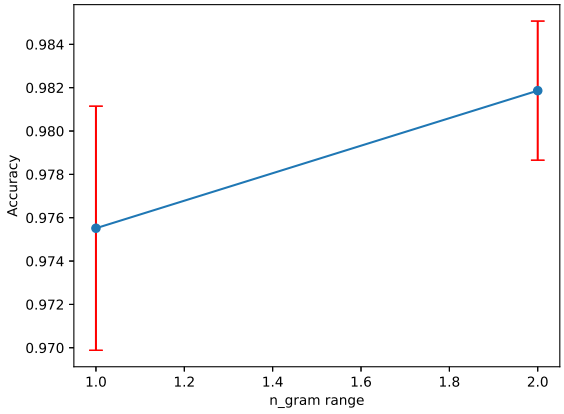


Figure 2: Cross validation results for single tokens `n_gram`=1 versus pairs of tokens `n_gram`=2.

All the tokens have to be encoded in some way, so that the model can be fed as input a fixed feature. One way of doing this is with a bag of words representation. With this technique, you have a feature vector and each index corresponds to a unique token (work or pair of words). At each index, you insert the frequency of that token in the feature vector.

A better alternative to bag of words is TF-IDF representation which expresses the importance of each token by looking at its frequency in the entire dataset. Term frequency is the number of times the token occurs in a given review. And inverse document frequency normalizes the term frequency by discounting very common words that occur across many reviews.

All these prepossessing steps are done very quickly with the sklearn function `TfidfVectorizer()`.

```
vectorizer = TfidfVectorizer(stop_words="english",  min_df=0.001,
             max_df=1.0, ngram_range=(1,2))
```

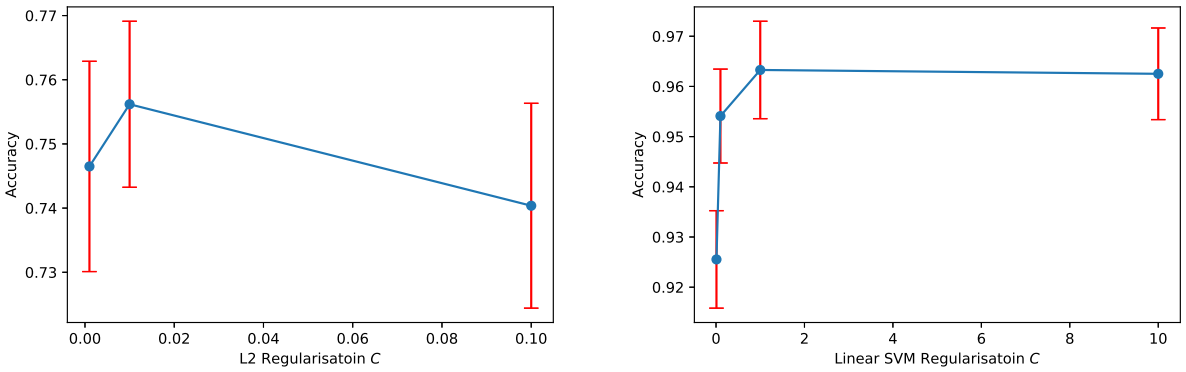
So if you give an input review of "First sell for and then do FreePlay I will definitely never buy anything from them again at least I could have offered a compensation" the corresponding feature vector will look like

```
[[0. 0. 0. ... 0. 0. 0.]]
```

Where each index corresponds to the TF-IDF of the given token.

Machine learning models

I compare the performance of three different machine learning models to the review data, namely logistic regression, SVM with linear kernel, and fully-connected neural network. I use cross validation to choose the regularisation hyperparameter, C, in Logistic regression and SVM in the usual manner. See figure 3. I choose $C = 0.01$ for the logistic model and $C = 1$ for the SVM model.



(a) Logistic regression model

(b) SVM

Figure 3: Cross validation analysis to choose regularisation hyperparameter in logistic regression and SVM

Recall that a logisitc model is described as a linear combination of features x (text tokens in this case) and parameters θ.

$$\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

The goal of training is to learn the weight parameters θs to predict the output, either review polarity or early access game.

Recall that the SVM cost function is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \max(0, 1 - y^{(i)} \theta^T x^{(i)}) + \frac{\theta^T \theta}{C}$$

C is a regularisation parameter in which the regularisation strength is inversely proportional to C. This means that a large C value will apply low regularisation and may lead to overfitting of the data and vice versa.

I designed a simple fully connected neural network with two hidden layers (see summary below). The first hidden layer had 500 neurons and the second has 50. Both layers use ReLU activation units and the final, output layer, uses a sigmoid activation function. This outputs probabilities (in the range [0,1]) of a review belonging to the class. I used the Adam optimization algorithm as opposed to vanilla stochastic gradient descent. The beneefit of Adam is that it uses adaptive learning rates at each itteration of training, so you don't have to manually set learning rates.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4272)]            0
_________________________________________________________________
dense (Dense)                (None, 500)               2136500
_________________________________________________________________
dense_1 (Dense)              (None, 50)                25050
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 2,161,601
Trainable params: 2,161,601
Non-trainable params: 0
```

I show the training curve of the neural network in figure 4. I train for 40 epochs in total using batch size 256. As expected, the loss decreases rapidly at the start and tapers off towards the end of the training. This is because most gains in performance happen early in the training and the more trained the network is, the harder it becomes to improve it further.

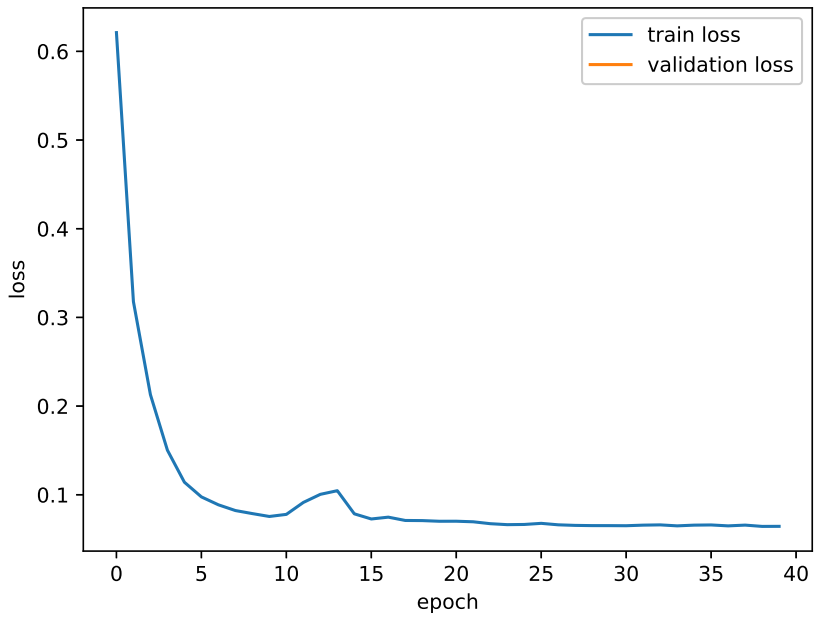
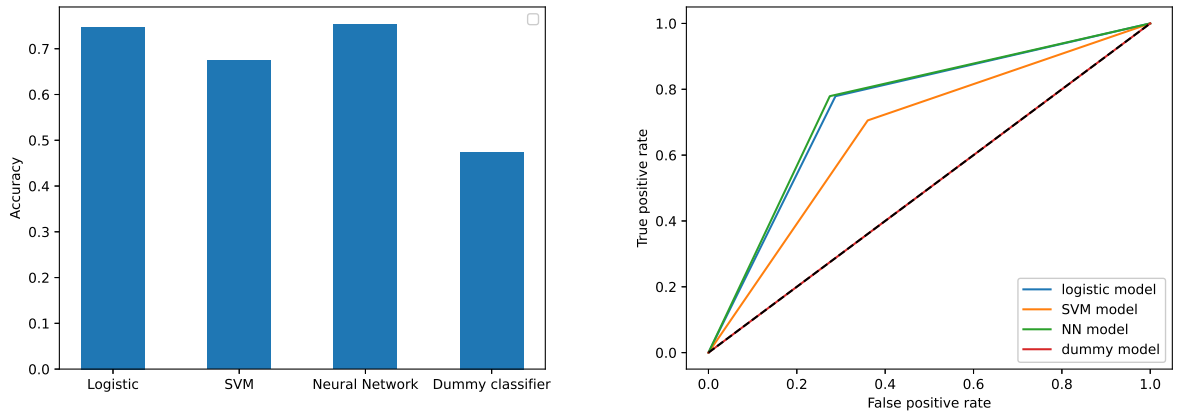


Figure 4: Neural network learning curve (showing loss)

Results

Predicting review polarity

I primarily use accuracy as a metric to compare classifiers for this task (predicting review polarity). Accuracy is suitable in this case because it is a classification task (not regression) and the two classes are evenly represented in the dataset (50:50). If one class was much more frequent than the other, accuracy would not be useful because it would be easy to get a high score by just predicting the most common class. I have also shown the confusion matrix for each model to take a closer look at what is being classified right and wrong. All the results below are from the unseen, test dataset (20% of the entire dataset).



(a) Accuracy

(b) ROC curve

Figure 5: Comparing model performances on predicting review polarity.

	Accuracy	Confusion Matrix
Logistic Regression	74.7%	[[166 67] [57 201]]
SVM	67.4%	[[149 84] [76 182]]
Neural Network	75.4%	[[169 64] [57 201]]
Dummy (most common class)	47.5%	[[233 0] [258 0]]

Table 1: Results for predicting review polarity.

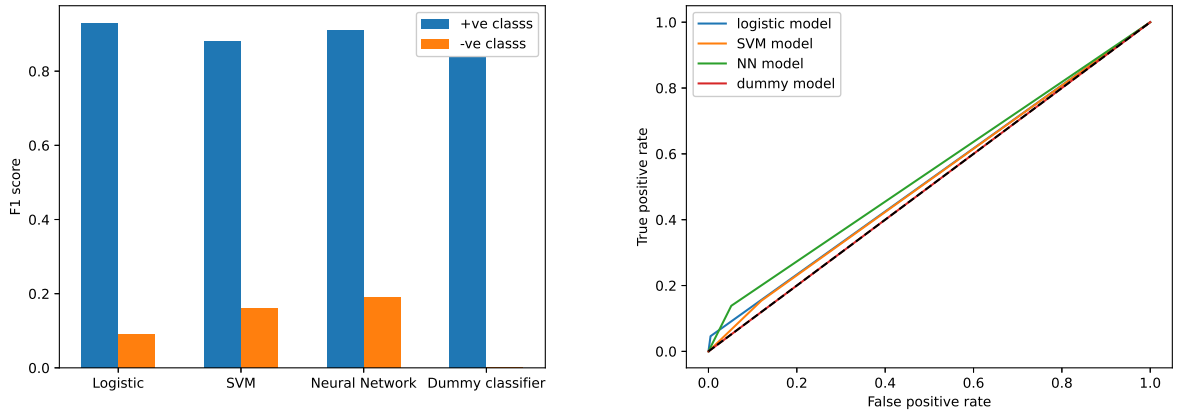
As is expected, the dummy classifier preforms the worst. It is interesting to note how the logistic regression model and neural network preform very comparably. In figure 5a and figure 5b, you can see how the test accuracy are very similar as well as their ROC curves. This is despite the fact the the neural network has vastly more parameters to learn. The logistic model only needs to learn as many parameters as there are features (i.e. 4272) whereas the neural network needs to learn (2,161,601) parameters in total. This can be calculated my multiplying the number of connections between neurons and adding on the number of neurons i.e. $(4272 \times 500) + (500 \times 50) + 500 + 50 + 1$.

Because of it's more complicated structure, the neural network takes longer to train than the logistic model. It takes 21 seconds while the logistic model takes only 0.7 seconds.

I was surprised as to how badly the SVM model with linear kernel performed relative to logistic regression, seeing as they are similar under the hood. The SVM model also took longer to train than both the neural network and logistic model. It took 68 seconds in total. I suspect it's poor performance here may be as a result of the large number of features compared to number of training examples. There are 4272 features and just 4411 training examples (reviews).

Predicting if review is for early access game

Comparing model performance by looking at accuracy for this task is not helpful. This is because the classes are hugely imbalanced for the early access label. Out of the 4902 reviews (ignoring the ones with only non-alphanumeric characters), 4362 belong to the negative class (not early access), leaving only 540 belong to the positive class (is early access). This is about a 90:10 ratio of negative to positive examples. If you predict "not early access" for every review, you would have 90% accuracy. So to get a picture of what's going on, I will look at F1 scores as well as the confusion matrices for each model. See table 2



(a) F1 score per class

(b) ROC curve

Figure 6: Comparing model performances on predicting review polarity.

	F1 score (+ve class)	F1 score (-ve class)	Confusion Matrix
Logistic Regression	93%	9%	[[424 2] [62 3]]
SVM	88%	16%	[[375 51] [55 10]]
Neural Network	91%	19%	[[404 22] [56 9]]
Dummy (most common class)	93%	0%	[[426 0] [65 0]]

Table 2: Results for predicting earlly access or not.

From the bar plots in figure 6a and the ROC curves in figure 6b, it is clear that no model is outperforming the baseline (which just predicts the most common class) and that they are all as bad as each other. The bad performance is due to the class imbalance, and until that is addressed, no model will perform well.

Conclusions

From my discussions above it is clear that the review text can be used to predict the review polarity to a reasonable accuracy (75%). The neural network preforms the best, but it is not significantly better than the logistic model. If the computational cost of the model is important, I would choose to use the logistic model over the neural network, otherwise I would go with the neural network.

Based on my cross validation results in figures 1a, 1b, 3, you may be surprised to see that the cross validation accuracy was over 90% while the test accuracy was only 75%. This is because the text vectorizer is fit to the entire training set so the `min_df` parameter and the IDF value in TFIDF has seen all of the training data. Therefore, when you do cross validation, the fold of data that you evaluate your model on isn't truly unseen data because the vectorizer has already looked at the frequency of tokens in it. However, since the vectorizer never looked at the test set, it only transforms it according to what it saw in the training set, it represents truly unseen data and this is why the test accuracy is lower than the cross validation.

```
vectorizer, X_vectors_train = text_pre_processing(X_train, min_df=0.001,
                                                  max_df=1.0, n_gram=2)
X_vectors_test = vectorizer.transform(X_test).toarray()
```

I have also show how the given dataset is not able to predict whether the review was for an early access version of the game or not. None of the models evaluated are able to outperform the baseline classifier. As stated before, this is due to the large class imbalance. There are methods of working with imbalanced data such as under-sampling the dominant class and oversampling the minor class or applying weight to the classes so that the smaller class examples have a greater weight. However, I would only use the technique in cases where it is absolutely impossible to collect more data. For example if you were training a model to predict a rare disease where there are very few example of that disease. In this case, I suspect that it would be possible to collect a more balance dataset, and that is definitely the approach I would take

2. (i) Over-fitting occurs when a machine learning model is too complex and fits the training data too well. It captures the noise of the training data and will fail to generalise to unseen data. An example of this would be trying to model features of a quadratic relationship using a cubic model. Over-fit models are said to display low bias but high variance. The performance of an over-fit model appears to be good on the training data, but it will preform poorly on unseen data.

   Under-fitting occurs when a model is too simple and fails to capture the underlying trend of the data. This model exhibits low variance but high bias. It will perform poorly on both the training data and unseen data. An example of under-fitting is trying to fit a quadratic relationship with a linear model.

   (ii) K-folds cross validation pseudo code.

```
k = # folds to use
divide training data into k folds

for each fold i:
    hold out fold i of training data and train model on remaining folds
    evaluate model performance on fold i

calculate average performance across all folds
```

   (iii) K-fold cross validation allows you to get the most out of your training data by showing you how a model will perform on unseen data. It prevents under-fitting of the training data because it gives you the opportunity to tune the model hyperparametrs to best preform on the training data.

   It prevents over-fitting because, by using cross validation, you can have a completely untouched test dataset. Without using cross-validation, you will likely (be it intentionally or not) overfit to the test dataset because you will tune model parameters to give you the best result on it.

   (iv) Logistic Regression vs kNN:

   - Logistic regression is a parametric model while kNN is a non-parametric model. This is an advantage of kNN because non-parametric models make no assumptions about the relationship between the input and output (e.g. that it is a linear or quadratic relationship). Instead, kNN just predicts an output based on the labels its nearest neighbours.
   - kNN being a non-parametric model can also be a disadvantage because you learn little about the features themselves. With a parametric model like logistic regression, learning the parameters (weights) of a model tells you about the role each feature plays in predicting the output. kNN gives you no real insights about the data because it just predicts outputs based on neighbouring points.
   - kNN is slower to train because finding nearest neighbours for each data point is computationally expensive. Furthermore, this causes kNN to scale poorly as you increase the dataset size.

   (v)
   - kNN can fail if the classes are imbalanced. If one class is much more common, the nearest neighbours will be biased towards the majority class.
   - kNN doesn't work well for high dimensional data. As the number of dimensions (number of features) increases, the nearest neighbour has to be the nearest in *every* dimension. This is know as the curse of dimensionality.

A Code to translate data to English

```
import json_lines
import pickle
from google_trans_new import google_translator
import re
import time

def load_and_translate(path):
    X = []
    y = []
    z = []
    translator = google_translator()
    start = time.time()
    with open(path, "rb") as f:
        for item in json_lines.reader(f):
            english = translator.translate(item['text'])
            english = re.sub("[^0-9a-zA-Z ]+", "", english)
            if not(english=="" and english.isspace):
                X.append(english)
                y.append(item['voted_up'])
                z.append(item['early_access'])
            else:
                print(english)


    print(len(X))
    with open("translated.pickle", "wb") as f:
        pickle.dump([X, y, z], f)

    print (time.time() - start)
    return


load_and_translate("data.json")
```

B Text analysis

```
import nltk
import pickle
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report,
                                                roc_curve
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.dummy import DummyClassifier

import keras
from keras.models import Sequential
from keras.layers import Dense
from matplotlib.ticker import MaxNLocator

def text_pre_processing(X, min_df, max_df, n_gram=1):
    vectorizer = TfidfVectorizer(norm=None, stop_words="english",  min_df=min_df, max_df=
                                                    max_df, ngram_range=(1,n_gram))
    X_vectors = vectorizer.fit_transform(X)
    print("vectorizer.stop_words_")
    print(len(vectorizer.stop_words_))
    print("vectorizer.get_feature_names()")
    print(len(vectorizer.get_feature_names()))
    return vectorizer, X_vectors.toarray()


def do_kfold(X, y,  method, order=None, C=None, kfolds=5):
    valid_methods = ["logistic", "svm", "bayes"]
    if method not in valid_methods:
        raise ValueError("Invalid type")

    kf = KFold(n_splits=kfolds)
    mean_error = []
    for train, test in kf.split(X):
        if method == "logistic":
            model = train_logistic_with_l2(X[train], y[train], C)
        elif method =="bayes":
            model = train_naive_bayes(X, y)
        else:
            model = train_svm(X, y, C)
        y_pred = model.predict(X[test])
        mean_error.append(accuracy_score(y[test], y_pred))
    acc = np.array(mean_error).mean()
    std = np.array(mean_error).std()
    return model, acc, std

def train_logistic_with_l2(X, y, C):
    clf = LogisticRegression(C=C, random_state=0)
    clf.fit(X,y)
    return clf

def train_svm(X, y, C):
    clf = SVC(kernel="linear", random_state=0, C=C)
    clf.fit(X, y)
    return clf

def train_naive_bayes(X, y):
    clf = GaussianNB()
    clf.fit(X, y)
    return clf


# Define some useful functions
```

```
class PlotLossAccuracy(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.acc = []
        self.losses = []
        self.val_losses = []
        self.val_acc = []
        self.logs = []

    def on_epoch_end(self, epoch, logs={}):
        self.logs.append(logs)
        self.x.append(int(self.i))
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.acc.append(logs.get('acc'))
        self.val_acc.append(logs.get('val_acc'))
        self.i += 1

        plt.figure()
        plt.plot(self.x, self.losses, label="train loss")
        plt.plot(self.x, self.val_losses, label="validation loss")
        plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))
        plt.ylabel('loss')
        plt.xlabel('epoch')
        plt.title('Model Loss')
        plt.legend()
        plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))
        plt.savefig("training_curve.eps", format="eps")


def full_pipeline(filename):

    with open(filename, "rb") as f:
        X,y,z = pickle.load(f)

    X = np.asarray(X)
    y = np.asarray(y)
    z = np.asarray(z)

    print(sum(z==True))

    # Split into 80% train and 20% test sets
    X_train, X_test, y_train, y_test = train_test_split(X, z, test_size=0.1, random_state=10
                                                        )

    # Use cross val to select preprocessing parameters
    # Select min_df (very rare terms)
    min_df_list = [0.0025, 0.005, 0.01, 0.015]
    acc_list = []
    std_list = []
    for min_df in min_df_list:
        _, X_vectors= text_pre_processing(X_train, min_df, max_df=1.0)
        model, acc, std = do_kfold(X_vectors, y_train, method="svm", order=1, C=1)
        acc_list.append(acc)
        std_list.append(std)
    plt.figure()
    plt.xlabel("Min_df")
    plt.ylabel("Accuracy")
    plt.errorbar(min_df_list, acc_list, yerr=std_list, fmt="-o", ecolor="r", capsize=5)
    plt.show()

    # Select max_df (very common terms)
    max_df_list = [0.09, 0.1, 0.11, 1.0]
    acc_list = []
    std_list = []
    for max_df in max_df_list:
        _, X_vectors= text_pre_processing(X_train, min_df=0.0025, max_df=max_df)
```

```
    model, acc, std = do_kfold(X_vectors, y_train, method="svm", order=1, C=1)
    acc_list.append(acc)
    std_list.append(std)
plt.figure()
plt.xlabel("Max_df")
plt.ylabel("Accuracy")
plt.errorbar(max_df_list, acc_list, yerr=std_list, fmt="-o", ecolor="r", capsize=5)
plt.show()

# Use cross val to select ngram parameter
n_gram_list = [1,2]
acc_list = []
std_list = []
for n in n_gram_list:
    _, X_vectors= text_pre_processing(X_train, 0.001, max_df=1.0, n_gram=n)
    model, acc, std = do_kfold(X_vectors, y_train, method="svm", order=1, C=1)
    acc_list.append(acc)
    std_list.append(std)
plt.figure()
plt.xlabel("n_gram range")
plt.ylabel("Accuracy")
plt.errorbar(n_gram_list, acc_list, yerr=std_list, fmt="-o", ecolor="r", capsize=5)
plt.savefig("n_gram_cross_val.eps", format='eps')

# Use cross validation to select C regularisatoin in logistic regression
c_vals = [ 0.001, 0.01, 0.1]
acc_list = []
std_list = []
_, X_vectors_train = text_pre_processing(X_train, min_df=0.01, max_df=1.0)
for c in c_vals:

    model, acc, std = do_kfold(X_vectors_train, y_train, method="logistic", order=2, C=c
                                                        )
    acc_list.append(acc)
    std_list.append(std)

plt.figure()
plt.xlabel("L2 Regularisatoin $C$")
plt.ylabel("Accuracy")
plt.errorbar(c_vals, acc_list, yerr=std_list, fmt="-o", ecolor="r", capsize=5)
plt.show()

# Use cross validation to select C in SVM
c_vals = [0.01, 0.1, 1]
acc_list = []
std_list = []
_, X_vectors_train = text_pre_processing(X_train, min_df=0.0025, max_df=1.0)
for c in c_vals:

    model, acc, std = do_kfold(X_vectors_train, y_train, method="svm", C=c)
    print(acc )
    acc_list.append(acc)
    std_list.append(std)

plt.figure()
plt.xlabel("Linear SVM Regularisatoin $C$")
plt.ylabel("Accuracy")
plt.errorbar(c_vals, acc_list, yerr=std_list, fmt="-o", ecolor="r", capsize=5)
# plt.show()

vectorizer, X_vectors_train = text_pre_processing(X_train, min_df=0.001, max_df=1.0,
                                                  n_gram=2)
X_vectors_test = vectorizer.transform(X_test).toarray()
print(X_vectors_train.shape)
print(X_vectors_test.shape)

# Logistic model classifier
model = train_logistic_with_l2(X_vectors_train, y_train, C=0.01)
```

```
    y_pred = model.predict(X_vectors_test)
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    log_fpr, log_tpr, _ = roc_curve(y_test, y_pred)


    # SVM model classifier
    model = train_svm(X_vectors_train, y_train, C=1)
    y_pred = model.predict(X_vectors_test)
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    svm_fpr, svm_tpr, _ = roc_curve(y_test, y_pred)

    # fully connected NN
    inputs = keras.layers.Input(shape=len(vectorizer.get_feature_names()))
    x = inputs
    x = Dense(500, activation='relu', )(x)
    x = Dense(50, activation='relu')(x)
    predictions = Dense(1, activation='sigmoid')(x)
    # we create the model
    model = keras.models.Model(inputs=inputs, outputs=predictions)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.summary()
    # we create a callback function to plot our loss function and accuracy
    pltCallBack = PlotLossAccuracy()
    # and train
    model.fit(X_vectors_train, y_train,
            batch_size=256, epochs=40,
            callbacks=[pltCallBack])
    y_pred = (model.predict(X_vectors_test) > 0.5).astype("int32")
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    nn_fpr, nn_tpr, _ = roc_curve(y_test, y_pred)

    # Dummy classifier
    dummy = DummyClassifier(strategy="most_frequent").fit(X_vectors_train, y_train)
    y_dummy = dummy.predict(X_vectors_test)
    dummy_fpr, dummy_tpr, _ = roc_curve(y_test, y_dummy)
    print("Dummy Model")
    print(confusion_matrix(y_test, y_dummy))
    print(classification_report(y_test, y_dummy))
    dummy_fpr, dummy_tpr, _ = roc_curve(y_test, y_dummy)

    # plot ROC curve
    plt.figure()
    plt.plot(log_fpr, log_tpr, label="logistic model")
    plt.plot(svm_fpr, svm_tpr, label="SVM model")
    plt.plot(nn_fpr, nn_tpr, label="NN model")
    plt.plot(dummy_fpr, dummy_tpr, label="dummy model")
    plt.plot([0, 1], [0, 1], color='k',linestyle='--')
    plt.xlabel("False positive rate")
    plt.ylabel("True positive rate")
    plt.legend()
    plt.savefig("ROC.eps", format="eps")


if __name__ == "__main__":

    full_pipeline("cleaned.pickle")
```