Programming assignment 3c: Text Classification system, DAT340/DIT867, Applied ML

- Alfredo Serafini
- Erling Hjermstad
  - Group PA3c 8


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

# for bundling the vectorizer and the classifier as a single "package"
from sklearn.pipeline import make_pipeline
# for evaluating the quality of the classifier
from sklearn.metrics import accuracy_score

%config InlineBackend.figure_format = 'svg'
plt.style.use('seaborn')
%matplotlib inline
```

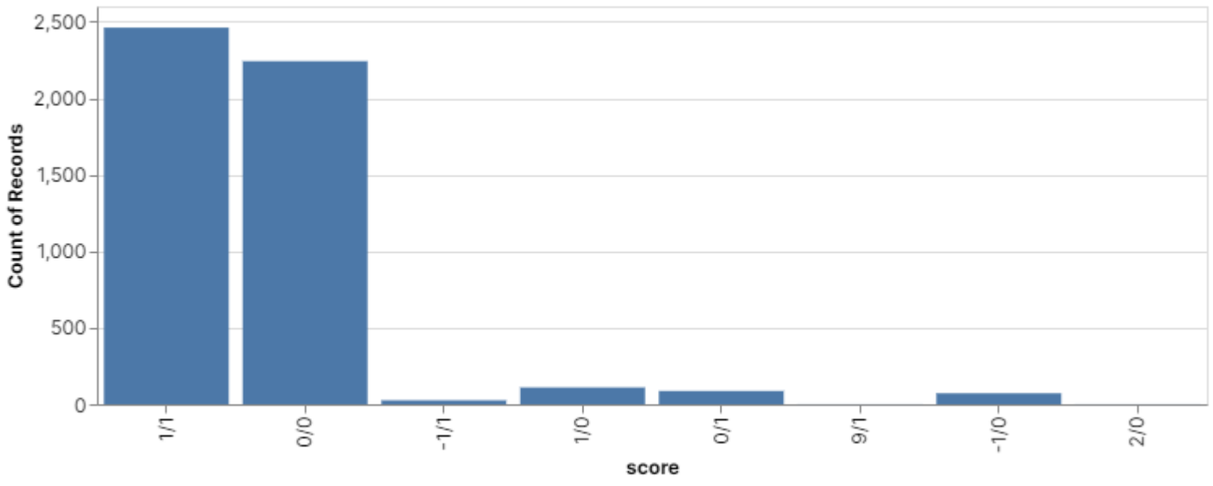
Import training data

```
train = pd.read_csv('PA3_train.tsv', names=['score', 'comment'], sep='\t')
train
```

	score object	comment object
	1/1 ..... 49.8% 0/0 ..... 44.5% 8 others 5.6%	We went wh… ..... 0% We arrived… ..... 0% 7007 others 99.9%
7005	9/1	Staff Service is good, sushi…
146	2/1	We were able to

		get a table…	
3374	2/1	Lovely staff so friendly all…	
4004	2/0	After a horrible…	
5147	2/0	All food tasted like the…	
3509	1/1	Very nice breakfast spot…	
3893	1/1	Have visited other Atul…	
3908	1/1	Simply love it. Good food,…	
3907	1/1	The service and ambience at th…	
3906	1/1	Great lunch and a must. We liv…	

Visualization of `train`



Clean training data and give human readable labels

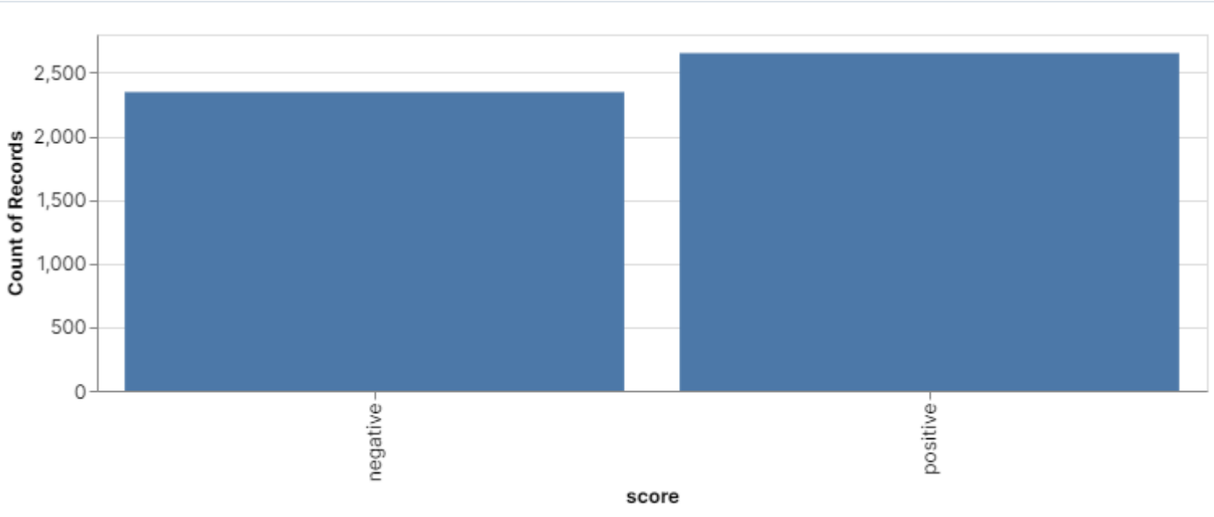
Only keeping data with agreeing annotation, i.e. 1/1 or 0/0

```
train_equal_annot = train.query('score == "1/1" or score == "0/0"').copy()
train_equal_annot['score'] = train_equal_annot['score'].apply(lambda x: ['negative', 'positive'][int(x[0])]) #convert each score to integer 0 or 1
train_equal_annot
```

	score object	comment object
		We arrived… …… 0%

	positive ..... 52.8% negative 47.2%	Ordered my… ..... 0% 6619 others … 100%	
0	negative	Ordered my food the hole meal…	
1	positive	We stopped her whilst walking…	
2	negative	Bad experience, On 23/03/19…	
3	negative	Extremely underwhelming…	
4	negative	Waited 30 minutes to get…	
5	negative	A mediocre burger, not…	
7	negative	If you go in here, don't go…	
8	negative	It's in a great location.…	
9	positive	About 200 people queuing…	
10	positive	Smaka is a great place to…	

Visualization of `train_equal_annot`

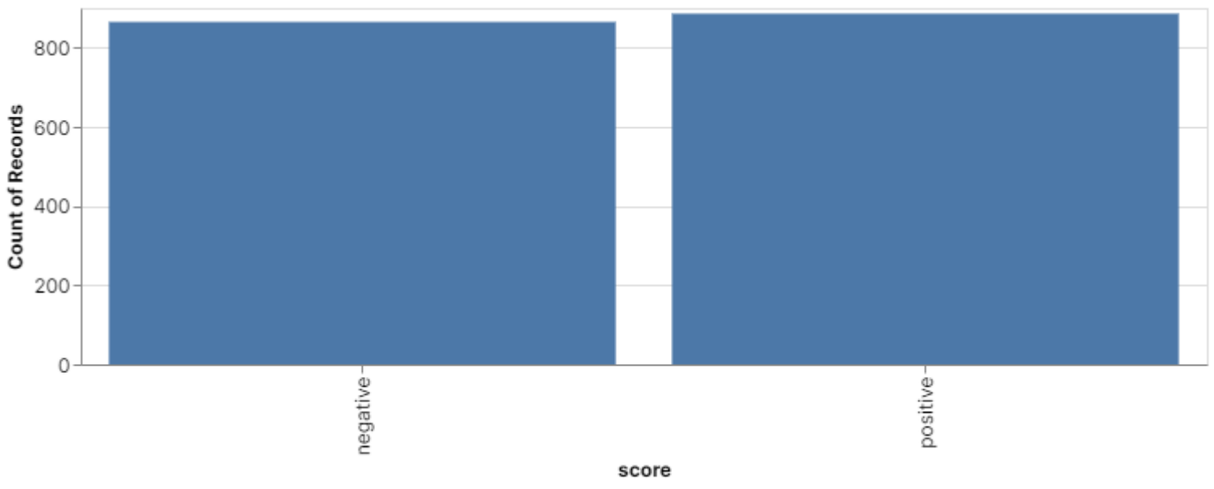


Import test data and give human readable labels

```
test = pd.read_csv('PA3_test_clean.tsv', names=['score', 'comment'], sep='\t')
test['score'] = test['score'].apply(lambda x: ['negative', 'positive'][x])
test
```

	score object	comment object
	positive ..... 50.6% negative 49.4%	Always ver… 0.1% Bigged up … 0.1% 1747 others 99.8%
0	negative	Over all I felt a bit…
1	positive	A wonderful experience!
2	positive	Always very delicious…
3	positive	Amazing as always
4	positive	Amazing food, the aubergine…
5	negative	Awful experience
6	negative	Bad food!
7	negative	Bigged up on reviews, overl…
8	positive	Brilliant. Lovely staff, …
9	positive	Delicious bake goods and…

Visualization of `test`



Give appropriate names

```
X_train, X_test, Y_train, Y_test = train_equal_annot['comment'], test['comment'], train_equal_annot['score'], test['score']
```

TfidfVectorizer + Dummy classifier as baseline

Recieve the expected ≈ 50 % accuracy. The classifier consistently choose **`positive`** as this was the most common label in our training data.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.dummy import DummyClassifier

dummy = make_pipeline(TfidfVectorizer(), DummyClassifier()).fit(X_train, Y_train)
dummy_score = accuracy_score(Y_test, dummy.predict(X_test))*100
print('TfidfVectorizer + Dummy classifier accuracy:', dummy_score,  "%")
```

```
TfidfVectorizer + Dummy classifier accuracy: 50.59965733866362 %
```

First try: TfidfVectorizer and linearSVC.

```
from sklearn.svm import LinearSVC

clf = make_pipeline(TfidfVectorizer(), LinearSVC())
clf.fit(X_train, Y_train)

print('Vocabulary size', len(clf.steps[0][1].vocabulary_))
print('TfidfVectorizer + Linear SVC accuracy:', accuracy_score(Y_test, clf.predict(X_test))*100, "%")
```

```
Vocabulary size 11331
TfidfVectorizer + Linear SVC accuracy: 96.68760708166761 %
```

Hyperparameter tuning

We set **`min_df=3`** so if a word occurs less then 3 times then the **`CountVectorizer`** will not inlcude it. Dramatically reduces vocabulary size (more than 60% smaller). Also tried ngram_range(1,2), (2,2), to include 'not great' etc as distinct features to caputure this neagtive intent instead of only 'great' as positive. Did not improve accuracy, seems that **`not`** by itself is sufficient to catch most of these cases, but also error prone as it is a common word.

```
clf = make_pipeline(TfidfVectorizer(ngram_range=(1, 1), max_df=0.33, min_df=3, strip_accents='ascii'), LinearSVC())
clf.fit(X_train, Y_train)
```

```
print('Vocabulary size', len(clf.steps[0][1].vocabulary_))
print('TfidfVectorizer + Linear SVC accuracy:', accuracy_score(Y_test, clf.predict(X_test))*100, "%")
```

```
Vocabulary size 4408
TfidfVectorizer + Linear SVC accuracy: 96.74471730439748 %
```

What does the model consider important?

Note that `not` is the most important word for classification

```
order = np.argsort(clf.steps[1][1].coef_[0])
a = np.asarray(list(zip(clf.steps[0][1].get_feature_names_out(), clf.steps[1][1].coef_[0])))

print("Top 10 decisive words for negative classification:\n", a[order][:10])
print("Top 10 decisive words for positive classification:\n", a[order][-10:][::-1])

idx = np.argsort(np.abs(clf.steps[1][1].coef_[0]))
print("Top 10 useless words for classification:\n", a[idx][:10])
```

```
 -                          -
 ['disappointing' '-2.6149190484575513']
 ['rude' '-2.443034368176764']
 ['poor' '-2.285121134107458']
 ['terrible' '-2.2573822971806226']
 ['dry' '-2.236676709141419']
 ['overpriced' '-2.13869615203587']
 ['average' '-2.0666854678768765']
 ['mediocre' '-2.041328887242617']]
Top 10 decisive words for positive classification:
 [['delicious' '3.8831189825749886']
 ['excellent' '3.4791862002509575']
 ['great' '3.358776284026078']
 ['amazing' '3.143825580974346']
 ['best' '2.508210000776978']
 ['good' '2.4926066212878113']
 ['loved' '2.245851078589634']
 ['love' '2.20645967743107']
 ['perfect' '2.1803300108662773']
 ['fantastic' '2.0796880608342527']]
Top 10 useless words for classification:
 [['jone' '0.0']
 ['pleasing' '0.0']
 ['acknowledgement' '0.0']
 ['cider' '0.0']
 ['elite' '0.0']
 ['law' '0.0']
 ['cigarette' '0.0']
```

```
['enemy' '0.0']
['leaf' '0.0']
['claire' '0.0']]
```

Try the `CountVectorizer`

```
from sklearn.feature_extraction.text import CountVectorizer

clf = make_pipeline(CountVectorizer(min_df=3, max_df=0.33, strip_accents='ascii'), LinearSVC())
clf.fit(X_train, Y_train)

print("Vocabulary size:", len(clf.steps[0][1].vocabulary_))
print("CountVectorizer + LinearSVC accuracy:", accuracy_score(Y_test, clf.predict(X_test))*100, "%")
```

```
Vocabulary size: 3128
CountVectorizer + LinearSVC accuracy: 95.20274129069102 %
```

Are most words useless?

If so perhaps stricter feature extraction can be implemented. 1631 of 4408 features have impacted less than 1/100 of the most important features.

```
max_impact = np.max(np.abs(clf.steps[1][1].coef_[0]))
i = sum(np.abs(clf.steps[1][1].coef_[0])<max_impact*1e-2)
print(round(i/len(clf.steps[0][1].vocabulary_)*100,1), "% of features have little impact on classification")
```

```
26.5 % of features have little impact on classification
```

Comparing different vectorizer and linear classifier combinations

```
def linearClassification(vectorizer, classifier, verbose=True):

    pipe = make_pipeline(vectorizer, classifier)
    pipe.fit(X_train, Y_train)
    print("--------" + pipe.steps[0][0] + " + " + pipe.steps[1][0] + "--------")
    print("Vocabulary size:", len(pipe.steps[0][1].vocabulary_))
    score = accuracy_score(Y_test, pipe.predict(X_test))*100
    print("Accuracy:", score, "%")

    if verbose:
```

```
        order = np.argsort(pipe.steps[1][1].coef_[0])
        feature_value = np.asarray(list(zip(pipe.steps[0][1].get_feature_names_out(), pipe.steps[1][1].coef_[0])))

        print("Top 10 decisive words for negative classification:\n", feature_value[order][:10])
        print("Top 10 decisive words for postive classification:\n", feature_value[order][-10:][::-1])

        abs_order = np.argsort(np.abs(pipe.steps[1][1].coef_[0]))
        print("Top 10 useless words for classification:\n", feature_value[abs_order][:10])
    print("")
    return score, pipe.steps[0][0] + " + " + pipe.steps[1][0]
```

```
from sklearn.linear_model import LogisticRegression, Perceptron, PassiveAggressiveClassifier, RidgeClassifier

classifiers = [LinearSVC(), LogisticRegression(), Perceptron(),
               PassiveAggressiveClassifier(), RidgeClassifier()]
vectorizers = [CountVectorizer(min_df=3), TfidfVectorizer(min_df=3)]

scores = []
names = []

for c in classifiers:
    for v in vectorizers:
        score, name = linearClassification(v, c, verbose=False)
        scores.append(score)
        names.append(name)
```

```
Accuracy: 96.28783552255854 %

--------tfidfvectorizer + logisticregression--------
Vocabulary size: 4397
Accuracy: 96.0022844089092 %


--------countvectorizer + perceptron--------
Vocabulary size: 4397
Accuracy: 95.03141062250144 %

--------tfidfvectorizer + perceptron--------
Vocabulary size: 4397
Accuracy: 94.74585950885208 %

--------countvectorizer + passiveaggressiveclassifier--------
Vocabulary size: 4397
Accuracy: 95.14563106796116 %

--------tfidfvectorizer + passiveaggressiveclassifier--------
Vocabulary size: 4397
Accuracy: 95.88806396344945 %

--------countvectorizer + ridgeclassifier--------
```

```
Vocabulary size: 4397
Accuracy: 91.37635636778984 %

--------tfidfvectorizer + ridgeclassifier--------
Vocabulary size: 4397
Accuracy: 96.63049685893775 %
```

Tree-based methods

`GradientBoostingClassifier` and **`DecisionTree`** combined with two distinct vectorizers.

```
from sklearn.ensemble import GradientBoostingClassifier

clf_gd = make_pipeline(CountVectorizer(min_df=3, max_df=1.0, strip_accents='ascii'), GradientBoostingClassifier())
clf_gd.fit(X_train, Y_train)

print('Vocabulary size: ', len(clf_gd.steps[0][1].vocabulary_))
cv_gbc_score = accuracy_score(Y_test, clf_gd.predict(X_test))*100
print("CountVectorizer + GradientBoosting accuracy: ",  cv_gbc_score, "%")


clf_gd = make_pipeline(TfidfVectorizer(min_df=3, max_df=1.0, strip_accents='ascii'), GradientBoostingClassifier())
clf_gd.fit(X_train, Y_train)

print('Vocabulary size: ', len(clf_gd.steps[0][1].vocabulary_))
tv_gbc_score = accuracy_score(Y_test, clf_gd.predict(X_test))*100
print("TdidfVectorizer + GradientBoosting accuracy: ",  tv_gbc_score, "%")
```

```
Vocabulary size:  4415
CountVectorizer + GradientBoosting accuracy:  91.2621359223301 %
Vocabulary size:  4415
TdidfVectorizer + GradientBoosting accuracy:  91.54768703597944 %
```

```
from sklearn.tree import DecisionTreeClassifier

clf_dt = make_pipeline(TfidfVectorizer(min_df=3, max_df=1.0, strip_accents='unicode'), DecisionTreeClassifier(max_depth=6)).fit(X_train, Y_train)
tv_dt_score = accuracy_score(Y_test, clf_dt.predict(X_test))*100
print('Vocabulary size: ', len(clf_gd.steps[0][1].vocabulary_))
print("TfidfVectorizer + DecisionTree accuracy: ", tv_dt_score, "%")

clf_dt = make_pipeline(CountVectorizer(min_df=3, max_df=1.0, strip_accents='unicode'), DecisionTreeClassifier(max_depth=6)).fit(X_train, Y_train)
cv_dt_score = accuracy_score(Y_test, clf_dt.predict(X_test))*100
print('Vocabulary size: ', len(clf_gd.steps[0][1].vocabulary_))
print("CountVectorizer + DecisionTree accuracy: ", cv_dt_score, "%")
```

```
Vocabulary size:  4415
```

```
TfidfVectorizer + DecisionTree accuracy:  80.01142204454598 %
Vocabulary size:  4415
CountVectorizer + DecisionTree accuracy:  80.01142204454598 %
```

Decision tree feature visualization

We can visualize the decision process in a decision tree, but since the gradient boosting classifier consists of many randomly perturbed decision trees we cannot visualize its decision process easily. For the count vectorizer + decision tree most splits consider if some value is >0.5 or not, which means if that feature occures.

We can use the `DecisionTreeClassifier` as initial visualisation to have an overview, however, the linear classifiers are much better in performance compared to the `DecisionTree` and in addition they give more precise information with an equal level of visualisation. We recommend to use a linear classifier for performance, reliability and visualisation.

```
from sklearn.tree import export_text
print(export_text(clf_dt.steps[1][1], feature_names=list(clf_dt.steps[0][1].get_feature_names_out()), max_depth=5))
```

```
|   |   |   |   |--- sat >  0.50
|   |   |   |   |   |--- of <= 0.50
|   |   |   |   |   |   |--- class: positive
|   |   |   |   |   |--- of >  0.50
|   |   |   |   |   |   |--- class: negative
|   |   |   |--- had >  1.50
|   |   |   |   |--- perfectly <= 0.50
|   |   |   |   |   |--- class: negative
|   |   |   |   |--- perfectly >  0.50
|   |   |   |   |   |--- class: positive
|   |--- delicious >  0.50
|   |   |--- to <= 4.50
|   |   |   |--- attention <= 0.50
|   |   |   |   |--- returning <= 0.50
|   |   |   |   |   |--- understand <= 0.50
|   |   |   |   |   |   |--- class: positive
|   |   |   |   |   |--- understand >  0.50
|   |   |   |   |   |   |--- class: negative
|   |   |   |   |--- returning >  0.50
|   |   |   |   |   |--- class: negative
|   |   |   |--- attention >  0.50
|   |   |   |   |--- paris <= 0.50
|   |   |   |   |   |--- class: negative
|   |   |   |   |--- paris >  0.50
|   |   |   |   |   |--- class: positive
|   |   |--- to >  4.50
|   |   |   |--- everything <= 0.50
|   |   |   |   |--- class: negative
|   |   |   |--- everything >  0.50
|   |   |   |   |--- class: positive
```

Neural network

Outperformed by linear methods. Performs almost at its best with a single hidden layer with a single neuron, which is very similar to a perceptron. We tried NN architectecture with `500, 200, 10` (3 layers) then `10, 10, 10` then `10, 10` then `10` neurons. We conclude that for this purpose a single layer with a single neuron is sufficient to classify the data.

```
from sklearn.neural_network import MLPClassifier

net = make_pipeline(TfidfVectorizer(min_df=3, strip_accents='ascii'), MLPClassifier(hidden_layer_sizes = (1))).fit(X_train, Y_train)
tv_net_score = net.score(X_test, Y_test)*100
print("Neural network accuracy:", tv_net_score, "%")
net = make_pipeline(CountVectorizer(min_df=3, strip_accents='ascii'), MLPClassifier(hidden_layer_sizes = (1))).fit(X_train, Y_train)
cv_net_score = net.score(X_test, Y_test)*100
print("Neural network accuracy:", cv_net_score, "%")
```

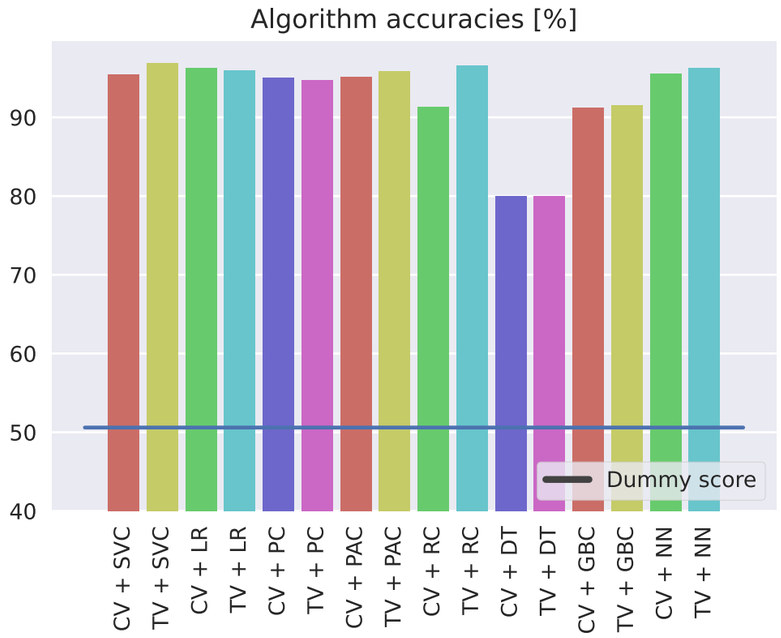
```
/shared-libs/python3.7/py/lib/python3.7/site-packages/sklearn/neural_network/_multilayer_perceptron.py:696: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet
  ConvergenceWarning,
Neural network accuracy: 96.23072529982866 %
Neural network accuracy: 95.54540262707025 %
/shared-libs/python3.7/py/lib/python3.7/site-packages/sklearn/neural_network/_multilayer_perceptron.py:696: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet
  ConvergenceWarning,
```

Evaluation of the algorithms

All tried algorithms of course outperform the baseline. The linear models seem to have an extra edge, with the linearSVC performing the best at 97% accuracy. Observed accuracies range from 82-97%. Also the vocabulary size or how computational expensive might be a relevant factor when comparing the approaches. However, all of them were fast for this task. Also the ability to extrapolate the important features is practical.

```
names = ["CV + SVC", "TV + SVC", "CV + LR", "TV + LR", "CV + PC", "TV + PC", "CV + PAC", "TV + PAC", "CV + RC", "TV + RC","CV + DT", "TV + DT", "CV + GBC", "TV + GBC", "CV + NN", "TV + NN"]
[scores.append(s) for s in [cv_dt_score,tv_dt_score, cv_gbc_score, tv_gbc_score, cv_net_score, tv_net_score]]

fig=plt.figure(figsize=(6.4,4.8))
sns.barplot(x = names, y = [s-40 for s in scores], bottom = 40, palette = sns.color_palette("hls"))
plt.xticks(list(range(len(names))), labels = names, rotation=90)
plt.title('Algorithm accuracies [%]')
sns.lineplot(x = [-1, len(names)], y = [dummy_score, dummy_score])
plt.legend(['Dummy score'], frameon=True, loc = 'lower right')
# plt.savefig('accuracy_barplot.png', bbox_inches='tight', dpi=300)
plt.tight_layout()
fig.get_size_inches()
plt.show()
```



Error inspection

Moving forward with the best performing model: TfidfVectorizer + LinearSVC

```
model = make_pipeline(TfidfVectorizer(min_df = 3, max_df=0.3, strip_accents='ascii'), LinearSVC()).fit(X_train, Y_train)
score = accuracy_score(Y_test, model.predict(X_test))
print("Best model accuracy:", round(100*score,1),"%")
```

```
Best model accuracy: 96.8 %
```

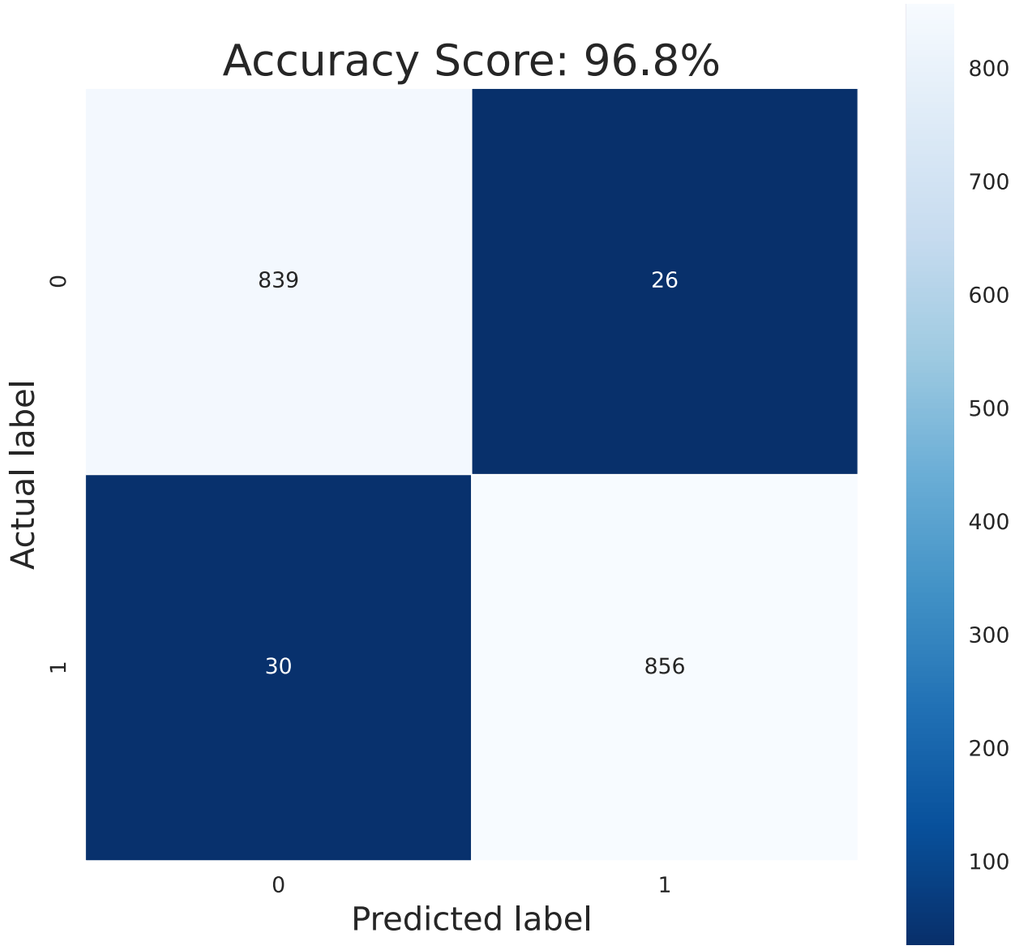
Equal errror distribution

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

conf_matrix  = confusion_matrix(Y_test, model.predict(X_test))

def plot_confusion_matrix(conf_matrix, plot = True, save = False, filename = 'confusion_matrix'):
    plt.figure(figsize=(8,8)) #size in inches
    sns.heatmap(conf_matrix, annot=True, fmt='g', linewidths=.5, square = True, cmap = 'Blues_r', xticklabels=['negative', 'positive'], yticklabels=['negative', 'positive'])
    plt.ylabel('Actual label', size=15)
    plt.xlabel('Predicted label', size=15)
    plt.title('Accuracy Score: {fscore:.{precision}f}%'.format(fscore = score*100, precision = 1), size = 20)
    if save:
        plt.savefig(filename)
    if plot:
        plt.show()
    plt.close()

plot_confusion_matrix(conf_matrix)
```




```
i = 0
errors = []

for comment, score in zip(X_test, Y_test):
    if score != model.predict([comment]):
        i += 1
        errors.append(comment)
        # print(comment, "----- should be", score,"predicted as", model.predict([comment]), "\n")

print(i, "erronous predictions found")

transformed_errors = model.steps[0][1].transform(errors)
errors_important_words = model.steps[0][1].inverse_transform(transformed_errors)

all_words = [w for sublist in errors_important_words for w in sublist]

cv = CountVectorizer()
occur = cv.fit_transform(all_words).toarray().sum(axis=0)
pos = np.argsort(occur)[-20:]
words = cv.get_feature_names_out()[pos][::-1]
print("20 most frequent words in errounous classifications along with their values for the tdidf + linearsvc and occurences:")
indices = [np.where(model.steps[0][1].get_feature_names_out() == w)[0][0] for w in words]
```

```
for val in list(zip(words, model.steps[1][1].coef_[0][indices], occur[pos][::-1])):
    print(val)
```

```
56 erronous predictions found
20 most frequent words in errounous classifications along with their values for the tdidf + linearsvc and occurences:
('but', -0.6633758087439657, 19)
('this', -0.07240244886391894, 17)
('with', 0.10621183151125504, 16)
('we', 0.13747938705478044, 16)
('good', 2.510469363939202, 15)
('not', -4.350011376802106, 15)
('my', 0.1695962229184103, 14)
('have', 0.2964235917165353, 13)
('were', -0.24060285217216346, 13)
('they', -0.4414075984729, 11)
('that', -0.16040917820960793, 11)
('restaurant', 0.2519013265748942, 11)
('nice', 1.3818351495003467, 11)
('had', 0.15216613135171758, 10)
('you', 0.3312418306139829, 9)
('on', -0.31897739360175176, 9)
('or', -0.37235836357601254, 9)
('very', 0.2735096768920818, 9)
('our', 0.46553073636960945, 9)
('just', -0.3025908503582582, 9)
```

We hypothesize that `not` is a key factor in giving erroneous classification. `but` tops the above list. We suspect this comes from messages in the form "something positive, *but* something negative" or vice-versa, which are hard to classify.

Perhaps most of the words on the list should not be included as features as they are not opinionated words. However, removing these common words without removing common useful words is difficult. E.g. `not` is common but important for sentiment. Perhaps some approach which specifically includes a chosen few common words, but ignores most could be used. On the other hand, maybe our algorithms pickup some common phrasings or sentence-builds of common words which are typical of negative/positive reviews?