Imports

In [1]:

```
import pandas as pd
import numpy as np
import re
import string
import json
import os
import pickle

import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline

from langdetect import detect
import spacy

import nltk
from nltk import pos_tag
from nltk.corpus import wordnet
from nltk.probability import FreqDist
from nltk.corpus import stopwords
from nltk.tokenize import regexp_tokenize, word_tokenize, RegexpTokenizer

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,\
classification_report, accuracy_score, precision_score

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer,\
HashingVectorizer

from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC

from xgboost import XGBClassifier

import pendulum

import tensorflow as tf
from transformers import TFAutoModel
```

In [2]:

```
# helper function to print the classification report and confusion matrix
def report(y_true, y_pred, class_names=['no_spoiler', 'spoiler']):
    print(classification_report(y_true, y_pred, target_names=class_names))
    confusion_matrix_plot(y_true, y_pred, class_names)

# helper function to plot the confusion matrix
def confusion_matrix_plot(y_true, y_pred, class_names):
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
    disp.plot(cmap=plt.cm.Blues)
    return plt.show()
```

Load Data

We performed our lemmatization in the EDA notebook and have full and partial dataframes saved on disk to load. Here we are loading from the reviews and synopsis lemmed data just the target and the review text. We have acccess to both the original text and the cleaned/lemmatized text depending on which modeling technique we're using.

For bag of words modeling use the 'review_text_lemmed' feature. For BERT use the 'review_text' feature

In [3]:

```
# loading the review data
large_df = pd.read_parquet('./data/reviews_lemmed.parquet', columns=['is_spoiler',
                                                                      'review_text_lemme
large_df
```

Out[3]:

	is_spoiler	review_text_lemmed
0	1	oscar year shawshank redemption write direct f...
1	1	shawshank redemption without doubt one brillia...
2	1	believe film best story ever tell film i'm tel...
3	1	yes spoiler film emotional impact find hard wr...
4	1	heart extraordinary movie brilliant indelible ...
...	...	...
573855	0	go wise fast pure entertainment assemble excep...
573856	0	well shall say one fun rate three plotlines or...
573857	0	go best movie ever see i've see lot movie read...
573858	0	call teenage version pulp fiction whatever wan...
573859	0	movie make doubt sucker family rebel mtv faith...

573860 rows × 2 columns

train/val/test split

We are splitting our data into train/validation/test datasets, with the validation and testing sets at 10% of our total data.

In [4]:

```
# set predictor to the review text, target to is_spoiler
predictor = large_df.review_text_lemmed
target = large_df.is_spoiler

# We want 10% of our data for test and 10% for validation, generate our holdout number
holdout = round(len(predictor) * 0.1)

# do first train/test split for train/val set and test set
```

```
X_trainval, X_test, y_trainval, y_test = train_test_split(predictor, target, random_sta
                                                          test_size=holdout, stratify=t

# perform 2nd train/test split (on train/val) for train and val sets
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, random_state=
                                                  test_size=holdout, stratify=y_trainva

# delete the large dataframe, predictor, target and trainval variables (no longer neede
del large_df, predictor, target, X_trainval, y_trainval

# confirm shapes
print(f"X_train / y_train shapes: {X_train.shape}, {y_train.shape}")
print(f"X_val / y_val shapes: {X_val.shape}, {y_val.shape}")
print(f"X_test / y_test shapes: {X_test.shape}, {y_test.shape}")
```

```
X_train / y_train shapes: (459088,), (459088,)
X_val / y_val shapes: (57386,), (57386,)
X_test / y_test shapes: (57386,), (57386,)
```

Baseline Model

For the baseline model we decided to keep it extremely simple and say that any review that contains the word 'spoiler' was, in fact, a spoiler.

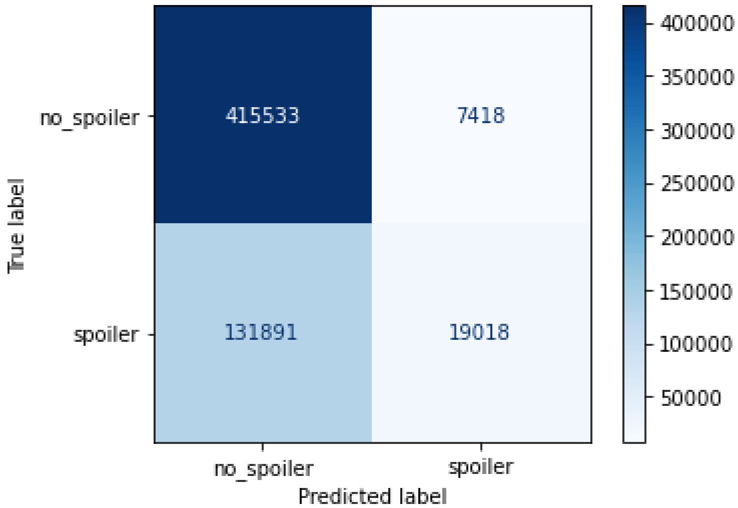
In [6]:

```
# make a baseline dataframe with just the target and the review_text
baseline_df = large_df[['is_spoiler', 'review_text_lemmed']].copy()

# create a new boolean value for reviews that contain the word 'spoiler'
baseline_df['contains_spoiler'] = baseline_df.review_text_lemmed.str.contains('spoiler'
baseline_df['contains_spoiler'] = baseline_df.contains_spoiler.astype(int)
```

In [7]:

```
report(baseline_df.is_spoiler, baseline_df.contains_spoiler)
```

```
              precision    recall  f1-score   support

  no_spoiler       0.76      0.98      0.86    422951
     spoiler       0.72      0.13      0.21    150909

    accuracy                           0.76    573860
   macro avg       0.74      0.55      0.54    573860
weighted avg       0.75      0.76      0.69    573860
```



The idea for basleline modeling does catch some spoilers correctly, but the recall on spoilers is only at .13. Interestingly, almost 30 percent of the reviews that contained the word spoiler were not actually spoilers. It's likely that the text says something like 'no spoiler' or 'spoiler free' which this baseline model would not pick up on. But this gives us some numbers to beat in iterative modeling efforts using bag of words.

Bag of Words

We wanted to make it easy to vectorize with different parameters and test the resulting data with a variety of models. The cell below contains two helper functions. The first transforms the X_train and X_val sets with the vectorizer we choose. The second function utilizes the first, and then feeds the resulting transformed data into the fitpredreport function defined in our imports. This way we can run a text on these three modeling techniques efficiently.

During modeling, we attempted to use RandomForest and XGBoost but those modeling techniques a) did not see much success or b) were unable to run due to memory overloading (see 4.3.1). So our helper function for testing the models will just run 3: Multinomial Naive Bayes, Logistic Regression, and Support Vector Machine.

In [5]:

```
# helper function to transform the training and validation data
def transformX(vectorizer, train=X_train, val=X_val, train_target=y_train, val_target=y
    # fit/transform training data
    train_vec = vectorizer.fit_transform(train)
    train_vec = pd.DataFrame.sparse.from_spmatrix(train_vec)
    train_vec.columns = sorted(vectorizer.vocabulary_)
    train_vec.set_index(train_target.index, inplace=True)
    # transform validation data
    val_vec = vectorizer.transform(val)
    val_vec = pd.DataFrame.sparse.from_spmatrix(val_vec)
    val_vec.columns = sorted(vectorizer.vocabulary_)
    val_vec.set_index(val_target.index, inplace=True)
    # return both dataframes
    return train_vec, val_vec

# helper function to test the models with our vectorized data
def test_models(vectorizer):
```

```
    # transfrom X_train and X_val with helper function
    X_train_tfidf, X_val_tfidf = transformX(vectorizer)

    # helper function to make fitting, predicting and reporting easier
    def fitpredreport(model):
        model.fit(X_train_tfidf, y_train)
        y_pred = model.predict(X_val_tfidf)
        report(y_val, y_pred)

    nb = MultinomialNB()
    lr = LogisticRegression(verbose=1, solver='liblinear', random_state=42, C=5, max_it
    svm = LinearSVC(random_state=42)

    print('Multinomial Naive Bayes')
    fitpredreport(nb)
    print('--------------------------------------------')
    print('Logistic Regression')
    fitpredreport(lr)
    print('--------------------------------------------')
    print('Support Vector')
    fitpredreport(svm)
```

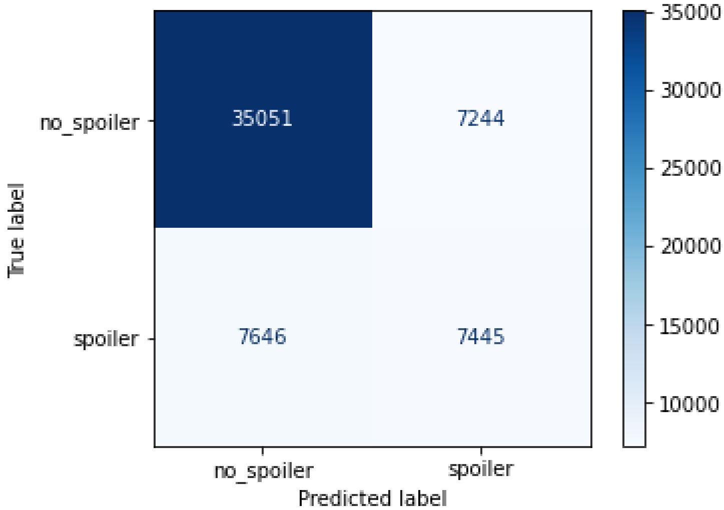
base count vectorizer

Using a basic count vectorizer. This takes about one and a quarter hours and generates a 230,366 word vocabulary.

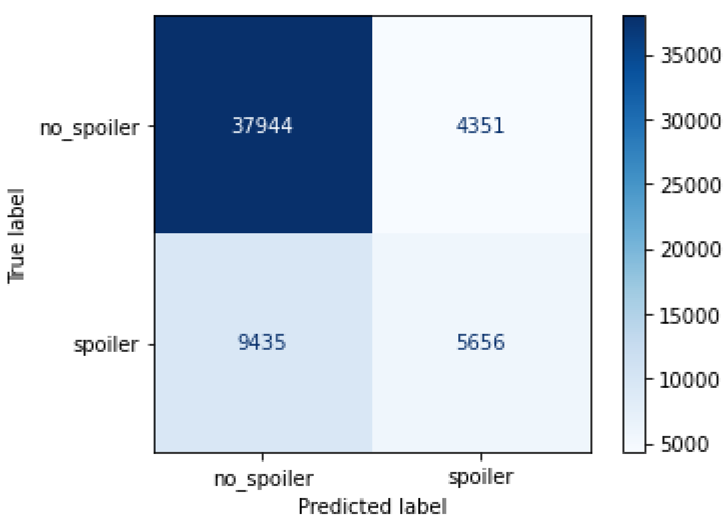
In [7]:

```
base_countvec = CountVectorizer()
test_models(base_countvec)
```

```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.82      0.83      0.82     42295
     spoiler       0.51      0.49      0.50     15091

    accuracy                           0.74     57386
   macro avg       0.66      0.66      0.66     57386
weighted avg       0.74      0.74      0.74     57386
```

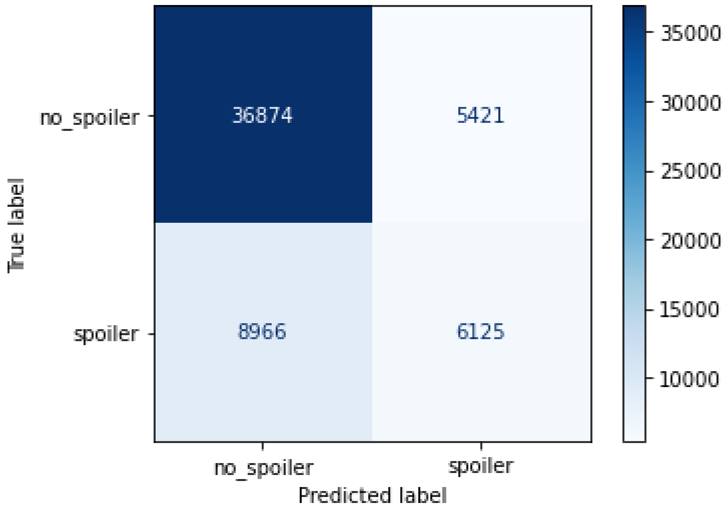



```
-------------------------------------------
Logistic Regression
[LibLinear]
C:\Users\brtra\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarnin
g: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
              precision    recall  f1-score   support

  no_spoiler       0.80      0.90      0.85     42295
     spoiler       0.57      0.37      0.45     15091

    accuracy                           0.76     57386
   macro avg       0.68      0.64      0.65     57386
weighted avg       0.74      0.76      0.74     57386
```




```
-------------------------------------------
Support Vector
C:\Users\brtra\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarnin
g: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
              precision    recall  f1-score   support

  no_spoiler       0.80      0.87      0.84     42295
     spoiler       0.53      0.41      0.46     15091

    accuracy                           0.75     57386
```

```
   macro avg       0.67      0.64      0.65     57386
weighted avg       0.73      0.75      0.74     57386
```



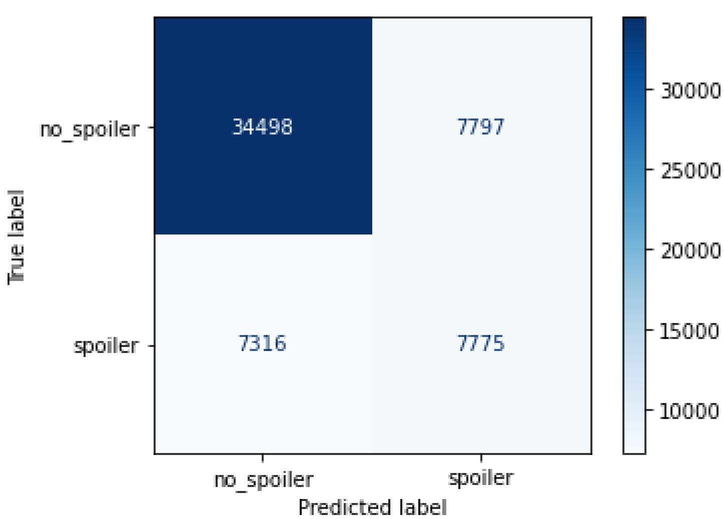
count vectorizer, min 5

Using a count vectorizer with min_df of 5. This takes about an hour and 15 minutes and generates a 65,206 word vocabulary.

In [6]:

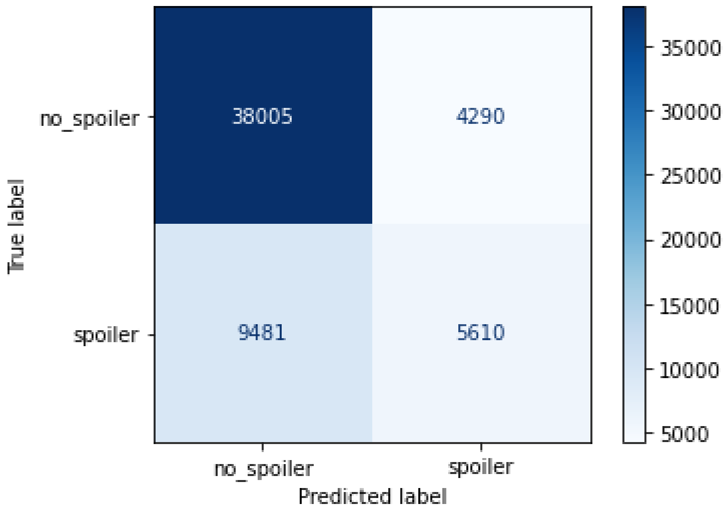
```
countvec_min_five = CountVectorizer(min_df=5)
test_models(countvec_min_five)
```

```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.83      0.82      0.82     42295
     spoiler       0.50      0.52      0.51     15091

    accuracy                           0.74     57386
   macro avg       0.66      0.67      0.66     57386
weighted avg       0.74      0.74      0.74     57386
```

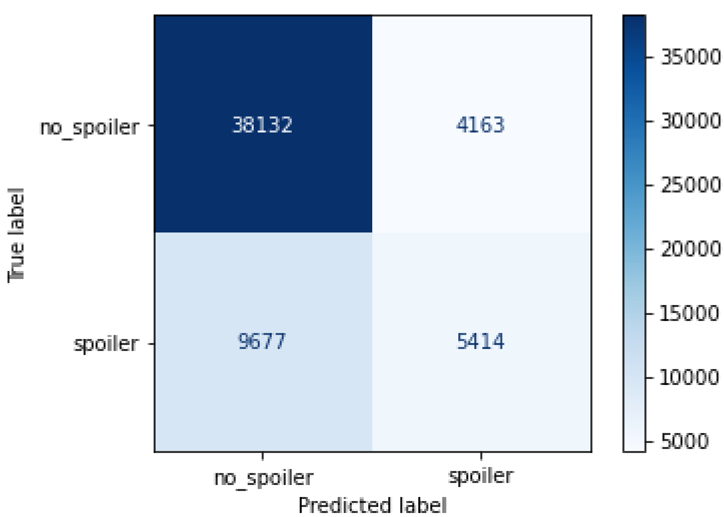



```
--------------------------------------------
Logistic Regression
[LibLinear]              precision    recall  f1-score   support
```

```
  no_spoiler       0.80      0.90      0.85     42295
     spoiler       0.57      0.37      0.45     15091

    accuracy                           0.76     57386
   macro avg       0.68      0.64      0.65     57386
weighted avg       0.74      0.76      0.74     57386
```




```
--------------------------------------------
Support Vector
C:\Users\brtra\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarnin
g: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
              precision    recall  f1-score   support

  no_spoiler       0.80      0.90      0.85     42295
     spoiler       0.57      0.36      0.44     15091

    accuracy                           0.76     57386
   macro avg       0.68      0.63      0.64     57386
weighted avg       0.74      0.76      0.74     57386
```



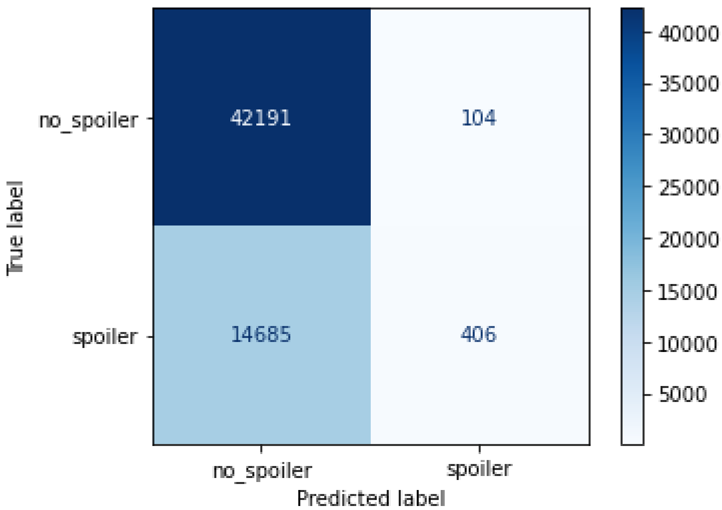
base tfidf vectorizer

Vectorizing tfidf with no parameter changes. Takes about 3 minutes and generates a vocabulary of

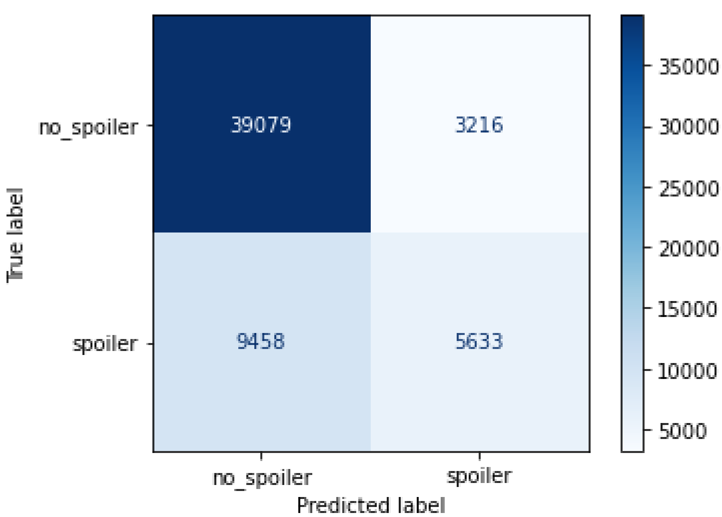
230,366

In [6]:
```
base_tfidf = TfidfVectorizer()
test_models(base_tfidf)
```

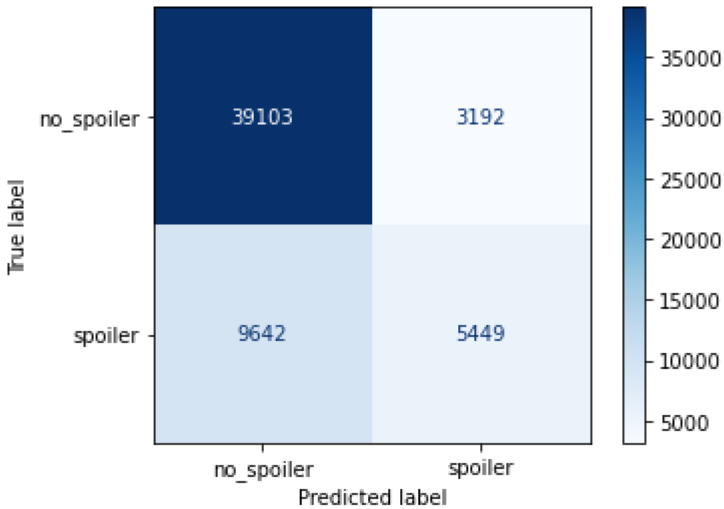
```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.74      1.00      0.85     42295
     spoiler       0.80      0.03      0.05     15091

    accuracy                           0.74     57386
   macro avg       0.77      0.51      0.45     57386
weighted avg       0.76      0.74      0.64     57386
```




```
--------------------------------------------
Logistic Regression
[LibLinear]              precision    recall  f1-score   support

  no_spoiler       0.81      0.92      0.86     42295
     spoiler       0.64      0.37      0.47     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.65      0.67     57386
weighted avg       0.76      0.78      0.76     57386
```



```
-------------------------------------------
Support Vector
              precision    recall  f1-score   support

  no_spoiler       0.80      0.92      0.86     42295
     spoiler       0.63      0.36      0.46     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.64      0.66     57386
weighted avg       0.76      0.78      0.75     57386
```



Of the three, the naive bayes model performs the worst, while the logistic regression and the support vector machine models perform similarly, with logistic regression just edging out the SVM model. The model definitely performs better than the simple baseline, but it's not super impressive, barely cracking 36% recall on the spoiler label.

We wanted to try to additional modeling techniques to see how they fared with the data. Knowing they tend to take longer to train we did not include them in the helper function, but will try them now.

RandomForest / XGBoost

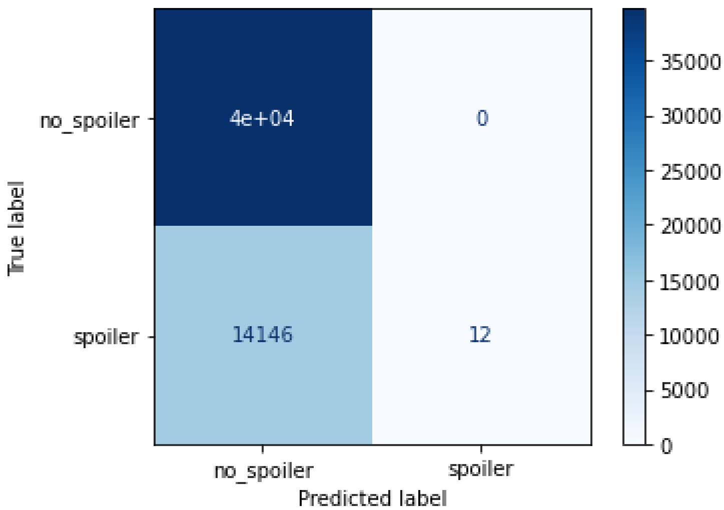
Here we have to break apart the helper function to generate the data to feed into these models to see how the results look.

In [12]:

```
# set tfidf vectorizer, default parameters
base_tfidf = TfidfVectorizer()

X_train_tfidf, X_val_tfidf = transformX(base_tfidf)

# helper function to make fitting, predicting and reporting easier
def fitpredreport(model):
    model.fit(X_train_tfidf, y_train)
    y_pred = model.predict(X_val_tfidf)
    report(y_val, y_pred)

# instantiate random forest classifier
rf_v1 = RandomForestClassifier(max_depth=20, random_state=42, n_jobs=-1)
```

```
# fit, predict and report with helper function
fitpredreport(rf_v1)
```

```
              precision    recall  f1-score   support

  no_spoiler       0.74      1.00      0.85     39720
     spoiler       1.00      0.00      0.00     14158

    accuracy                           0.74     53878
   macro avg       0.87      0.50      0.43     53878
weighted avg       0.81      0.74      0.63     53878
```



In [15]:

```
xgb_v1 = XGBClassifier(random_state=42, n_jobs=-1)
fitpredreport(xgb_v1)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_10548/2058205553.py in <module>
      1 xgb_v1 = XGBClassifier(random_state=42, n_jobs=-1)
----> 2 fitpredreport(xgb_v1)

~\AppData\Local\Temp/ipykernel_10548/2006769416.py in fitpredreport(model)
      6 # helper function to make fitting, predicting and reporting easier
      7 def fitpredreport(model):
----> 8     model.fit(X_train_tfidf, y_train)
      9     y_pred = model.predict(X_val_tfidf)
     10     report(y_val, y_pred)

~\anaconda3\lib\site-packages\xgboost\core.py in inner_f(*args, **kwargs)
    530         for k, arg in zip(sig.parameters, args):
    531             kwargs[k] = arg
--> 532         return f(**kwargs)
    533 
    534     return inner_f

~\anaconda3\lib\site-packages\xgboost\sklearn.py in fit(self, X, y, sample_weight, base_
margin, eval_set, eval_metric, early_stopping_rounds, verbose, xgb_model, sample_weight_
eval_set, base_margin_eval_set, feature_weights, callbacks)
   1380             xgb_model, eval_metric, params, early_stopping_rounds, callbacks
   1381         )
-> 1382         train_dmatrix, evals = _wrap_evaluation_matrices(
```

```
   1383             missing=self.missing,
   1384             X=X,

~\anaconda3\lib\site-packages\xgboost\sklearn.py in _wrap_evaluation_matrices(missing,
 X, y, group, qid, sample_weight, base_margin, feature_weights, eval_set, sample_weight_
eval_set, base_margin_eval_set, eval_group, eval_qid, create_dmatrix, enable_categorica
l)
    399
    400     """
--> 401     train_dmatrix = create_dmatrix(
    402         data=X,
    403         label=y,

~\anaconda3\lib\site-packages\xgboost\sklearn.py in <lambda>(**kwargs)
   1394             eval_group=None,
   1395             eval_qid=None,
-> 1396             create_dmatrix=lambda **kwargs: DMatrix(nthread=self.n_jobs, **kwar
gs),
   1397             enable_categorical=self.enable_categorical,
   1398         )

~\anaconda3\lib\site-packages\xgboost\core.py in inner_f(*args, **kwargs)
    530         for k, arg in zip(sig.parameters, args):
    531             kwargs[k] = arg
--> 532         return f(**kwargs)
    533
    534     return inner_f

~\anaconda3\lib\site-packages\xgboost\core.py in __init__(self, data, label, weight, bas
e_margin, missing, silent, feature_names, feature_types, nthread, group, qid, label_lowe
r_bound, label_upper_bound, feature_weights, enable_categorical)
    641             return
    642
--> 643         handle, feature_names, feature_types = dispatch_data_backend(
    644             data,
    645             missing=self.missing,

~\anaconda3\lib\site-packages\xgboost\data.py in dispatch_data_backend(data, missing, th
reads, feature_names, feature_types, enable_categorical)
    894         return _from_tuple(data, missing, threads, feature_names, feature_types
)
    895     if _is_pandas_df(data):
--> 896         return _from_pandas_df(data, enable_categorical, missing, threads,
    897                                feature_names, feature_types)
    898     if _is_pandas_series(data):

~\anaconda3\lib\site-packages\xgboost\data.py in _from_pandas_df(data, enable_categorica
l, missing, nthread, feature_names, feature_types)
    343     feature_types: Optional[List[str]],
    344 ) -> Tuple[ctypes.c_void_p, FeatureNames, Optional[List[str]]]:
--> 345     data, feature_names, feature_types = _transform_pandas_df(
    346         data, enable_categorical, feature_names, feature_types
    347     )

~\anaconda3\lib\site-packages\xgboost\data.py in _transform_pandas_df(data, enable_categ
orical, feature_names, feature_types, meta, meta_type)
    329
    330     dtype = meta_type if meta_type else np.float32
--> 331     arr = transformed.values
    332     if meta_type:
```

```
    333         arr = arr.astype(meta_type)

~\anaconda3\lib\site-packages\pandas\core\frame.py in values(self)
  10662         """
  10663         self._consolidate_inplace()
> 10664         return self._mgr.as_array(transpose=True)
  10665 
  10666     @deprecate_nonkeyword_arguments(version=None, allowed_args=["self"])

~\anaconda3\lib\site-packages\pandas\core\internals\managers.py in as_array(self, transp
ose, dtype, copy, na_value)
   1464                 arr = arr.astype(dtype, copy=False)  # type: ignore[arg-typ
e]
   1465         else:
-> 1466             arr = self._interleave(dtype=dtype, na_value=na_value)
   1467             # The underlying data was copied within _interleave
   1468             copy = False

~\anaconda3\lib\site-packages\pandas\core\internals\managers.py in _interleave(self, dty
pe, na_value)
   1500         # Tuple[Any, Union[int, Sequence[int]]], List[Any], _DTypeDict,
   1501         # Tuple[Any, Any]]]"
-> 1502         result = np.empty(self.shape, dtype=dtype)  # type: ignore[arg-type]
   1503 
   1504         itemmask = np.zeros(self.shape[0])

MemoryError: Unable to allocate 713. GiB for an array with shape (221997, 431026) and da
ta type float64
```

The RandomForest model performed terribly, while the XGBoost classifer wouldn't run at all due to memory allocation. We are not going to attempt to model with these for further iteration of vectorizer.

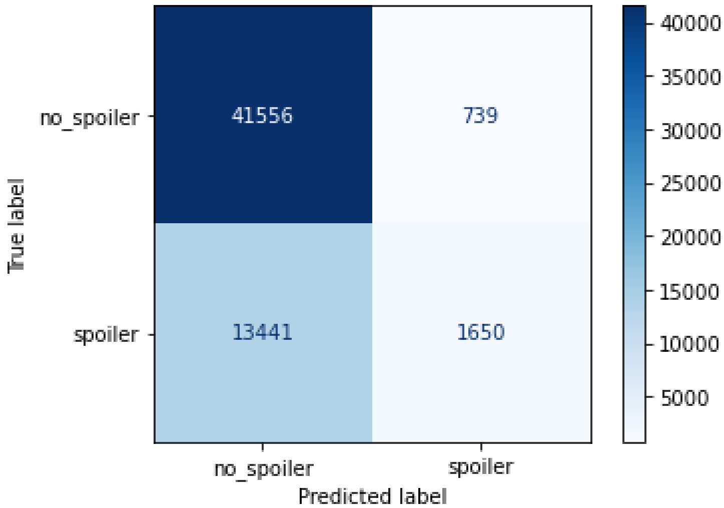
tfidf vectorizer, min 5

Vectorizing with min_df of 5. Takes about 3 minutes and generates a vocabulary of 65,206.

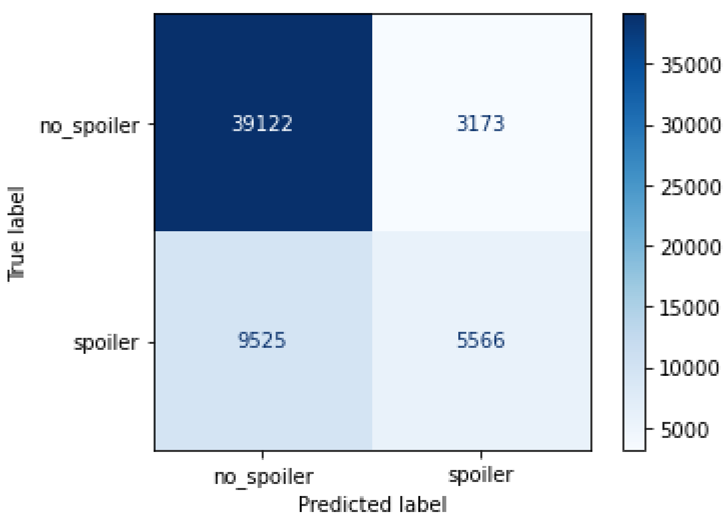
In [8]:

```python
tfidf_min_five = TfidfVectorizer(min_df=5)

test_models(tfidf_min_five)
```

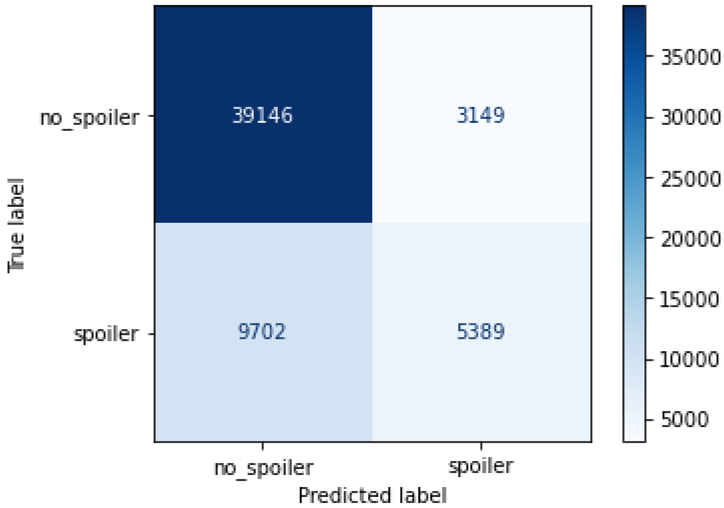
```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.76      0.98      0.85     42295
     spoiler       0.69      0.11      0.19     15091

    accuracy                           0.75     57386
   macro avg       0.72      0.55      0.52     57386
weighted avg       0.74      0.75      0.68     57386
```




```
-------------------------------------------
Logistic Regression
[LibLinear]              precision    recall  f1-score   support

  no_spoiler       0.80      0.92      0.86     42295
     spoiler       0.64      0.37      0.47     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.65      0.66     57386
weighted avg       0.76      0.78      0.76     57386
```




```
-------------------------------------------
Support Vector
              precision    recall  f1-score   support

  no_spoiler       0.80      0.93      0.86     42295
     spoiler       0.63      0.36      0.46     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.64      0.66     57386
weighted avg       0.76      0.78      0.75     57386
```



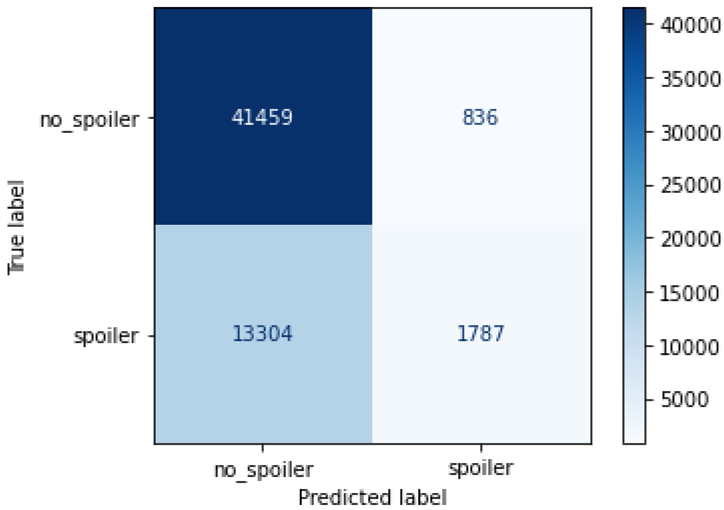
We certainly see some improvement by restricting the occurance of the words, i.e getting rid of words only used a few times. Lets up the min_df to 8 and see if we can maintain the models performance with an even smaller library

tfidf vectorizer, min 8

Vectorizing with min_df of 8. Takes about three minutes and generates a vocabulary of 53,190

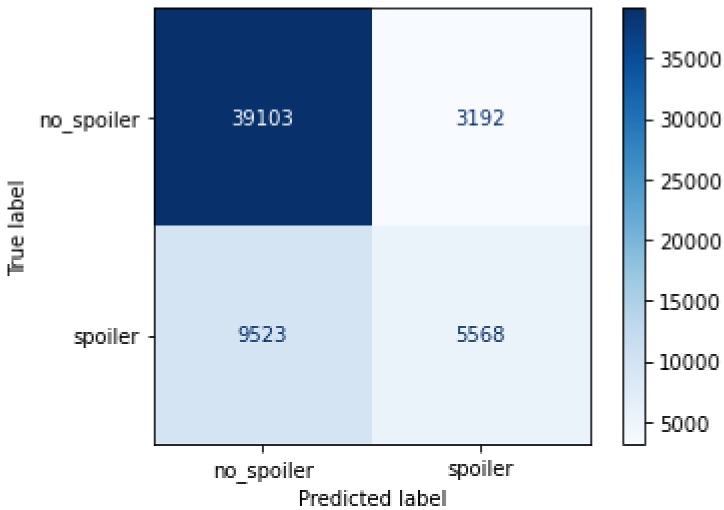
In [10]:
```
tfidf_min_eight = TfidfVectorizer(min_df=8)

test_models(tfidf_min_eight)
```

```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.76      0.98      0.85     42295
     spoiler       0.68      0.12      0.20     15091

    accuracy                           0.75     57386
   macro avg       0.72      0.55      0.53     57386
weighted avg       0.74      0.75      0.68     57386
```

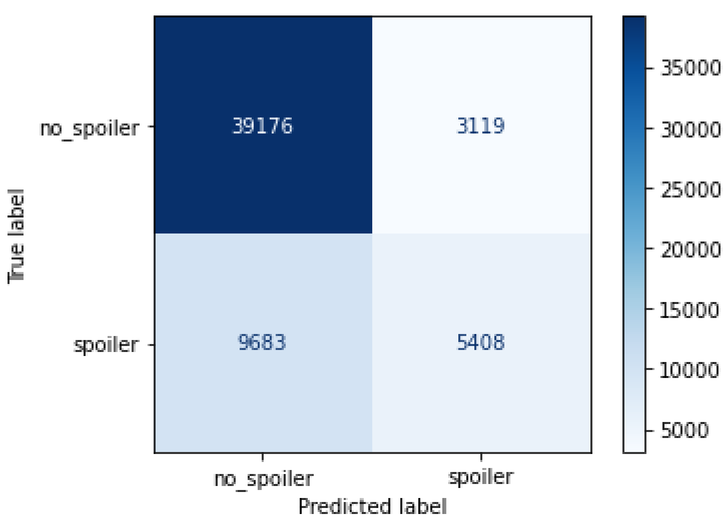



```
--------------------------------------------
Logistic Regression
```

```
[LibLinear]              precision    recall  f1-score   support

  no_spoiler       0.80      0.92      0.86     42295
     spoiler       0.64      0.37      0.47     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.65      0.66     57386
weighted avg       0.76      0.78      0.76     57386
```




```
--------------------------------------------
Support Vector
              precision    recall  f1-score   support

  no_spoiler       0.80      0.93      0.86     42295
     spoiler       0.63      0.36      0.46     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.64      0.66     57386
weighted avg       0.76      0.78      0.75     57386
```



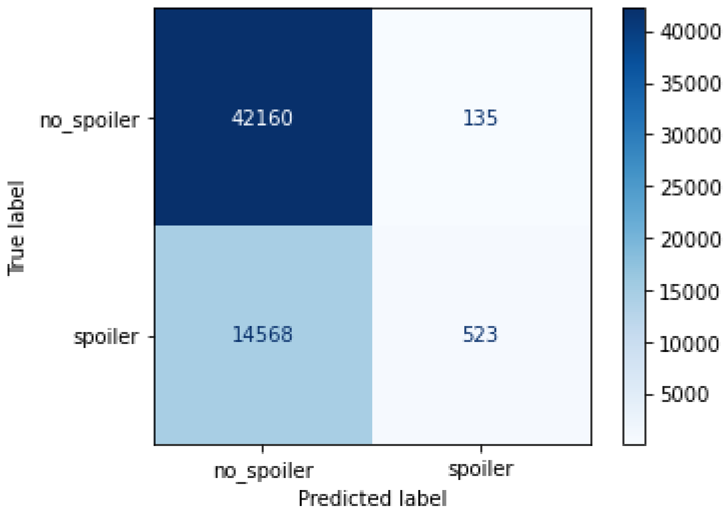
Increasing the minimum did improve our model much, save for the naive bayes model (but that 'improvement' is minimal). Lets add in bigrams, keeping our min_df at 8 and see if there can be any more improvement.

tfidf vectorizer, min 8 adding bigrams

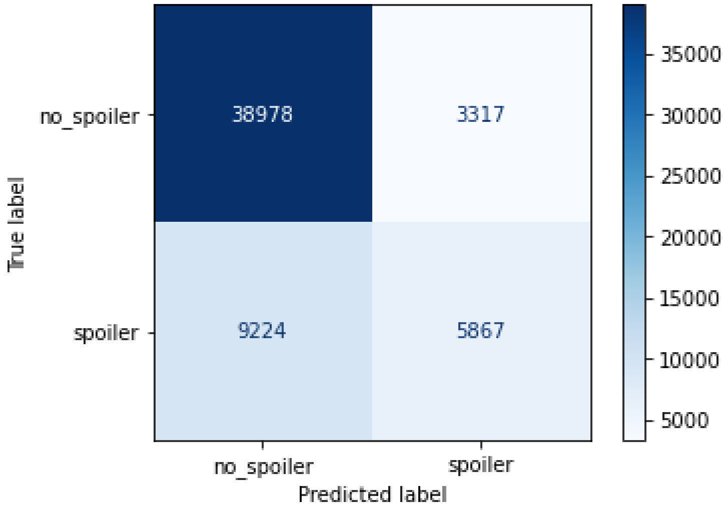
Vectorizing with bigrams and a min_df of 8. Takes about 12 minutes and generates a vocabulary of 967,418

In [12]:
```
tfidf_min_eight_bi = TfidfVectorizer(min_df=8, ngram_range=(1,2))

test_models(tfidf_min_eight_bi)
```

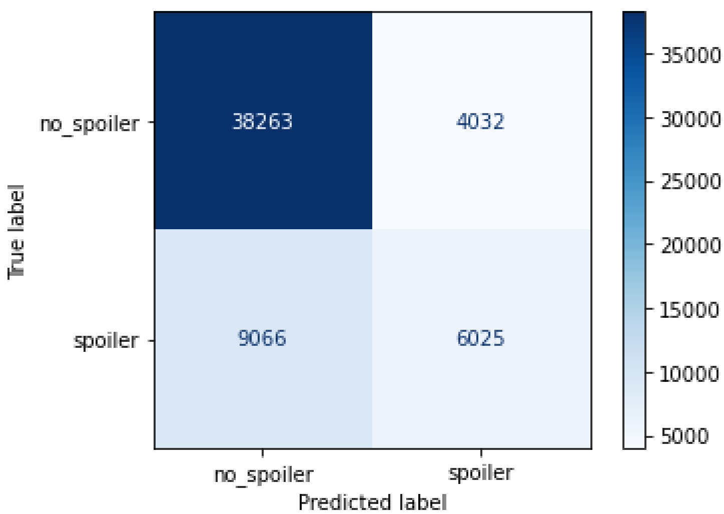
```
Multinomial Naive Bayes
              precision    recall  f1-score   support

  no_spoiler       0.74      1.00      0.85     42295
     spoiler       0.79      0.03      0.07     15091

    accuracy                           0.74     57386
   macro avg       0.77      0.52      0.46     57386
weighted avg       0.76      0.74      0.65     57386
```




```
--------------------------------------------
Logistic Regression
[LibLinear]              precision    recall  f1-score   support

  no_spoiler       0.81      0.92      0.86     42295
     spoiler       0.64      0.39      0.48     15091

    accuracy                           0.78     57386
   macro avg       0.72      0.66      0.67     57386
weighted avg       0.76      0.78      0.76     57386
```




```
------------------------------------------
Support Vector
              precision    recall  f1-score   support

  no_spoiler       0.81      0.90      0.85     42295
     spoiler       0.60      0.40      0.48     15091

    accuracy                           0.77     57386
   macro avg       0.70      0.65      0.67     57386
weighted avg       0.75      0.77      0.76     57386
```



Still seeing improvment, but it's kind of plateauing. Lets add in trigrams and see how it looks.

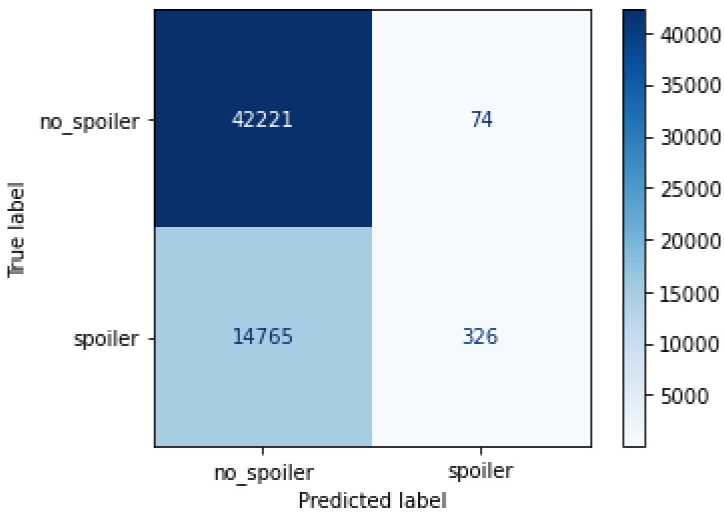
tfidf vectorizer, min 8 adding trigrams

Vectorizing with trigrams and a min_df of 8. Takes about 40 minutes and generates a vocabulary of 1,358,109

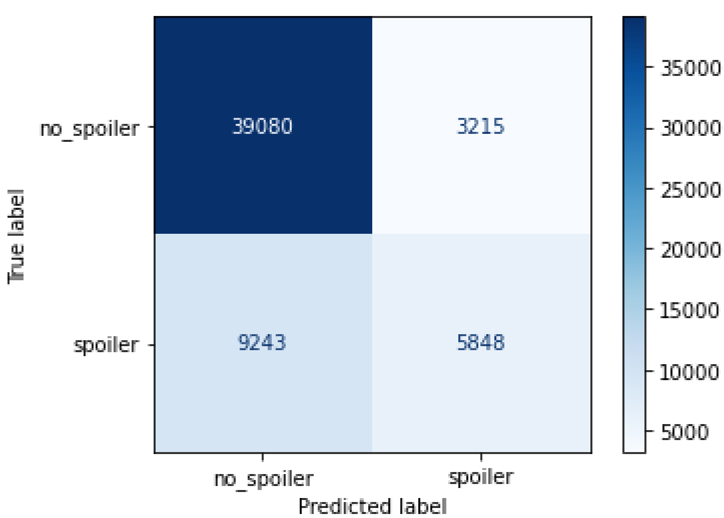
In [14]:
```
tfidf_min_eight_tri = TfidfVectorizer(min_df=8, ngram_range=(1,3))

test_models(tfidf_min_eight_tri)
```

```
Multinomial Naive Bayes
              precision    recall  f1-score   support
```

```
  no_spoiler       0.74      1.00      0.85     42295
     spoiler       0.81      0.02      0.04     15091

    accuracy                           0.74     57386
   macro avg       0.78      0.51      0.45     57386
weighted avg       0.76      0.74      0.64     57386
```

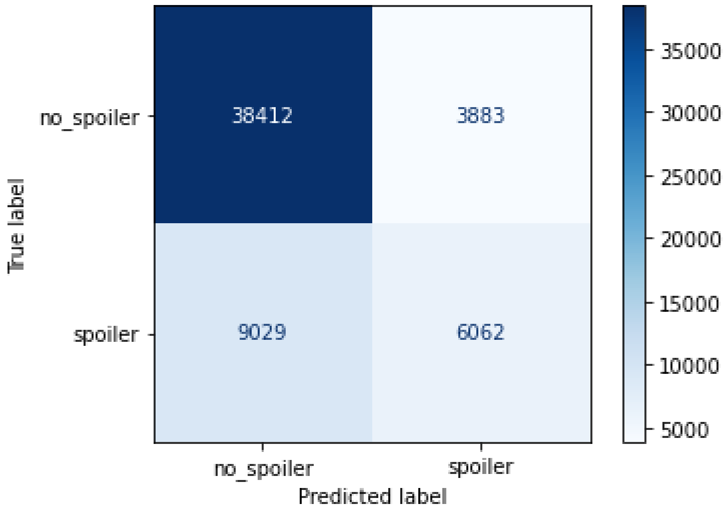



```
--------------------------------------------
Logistic Regression
[LibLinear]              precision    recall  f1-score   support

  no_spoiler       0.81      0.92      0.86     42295
     spoiler       0.65      0.39      0.48     15091

    accuracy                           0.78     57386
   macro avg       0.73      0.66      0.67     57386
weighted avg       0.77      0.78      0.76     57386
```




```
--------------------------------------------
Support Vector
              precision    recall  f1-score   support

  no_spoiler       0.81      0.91      0.86     42295
     spoiler       0.61      0.40      0.48     15091

    accuracy                           0.77     57386
```

```
   macro avg       0.71      0.65      0.67     57386
weighted avg       0.76      0.77      0.76     57386
```



Evaluating selected model on test set

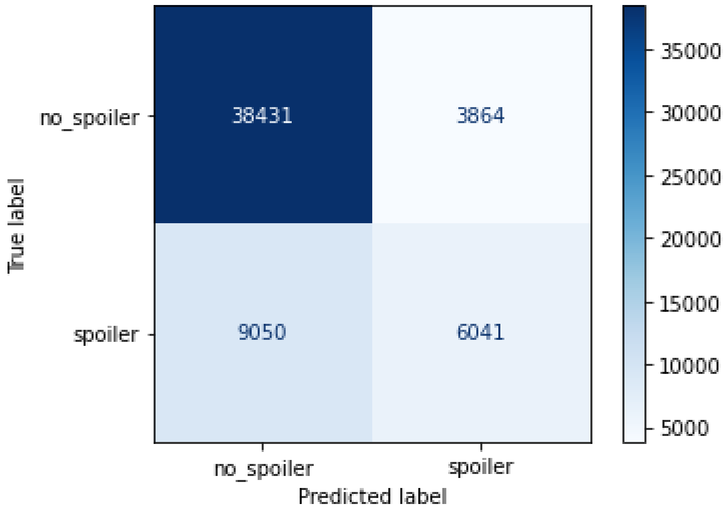
In [18]:

```
# tfidf_min_eight_tri = TfidfVectorizer(min_df=8, ngram_range=(1,3))

# using the transform helper function but setting the validation to X and y_test
X_train_tfidf, X_test_tfidf = transformX(tfidf_min_eight_tri, val=X_test, val_target=y_
```

In [19]:

```
svm = LinearSVC(random_state=42)
svm.fit(X_train_tfidf, y_train)

y_pred = svm.predict(X_test_tfidf)
report(y_test, y_pred)
```

```
              precision    recall  f1-score   support

  no_spoiler       0.81      0.91      0.86     42295
     spoiler       0.61      0.40      0.48     15091

    accuracy                           0.77     57386
   macro avg       0.71      0.65      0.67     57386
weighted avg       0.76      0.77      0.76     57386
```



BERT

Inspiration

preprocessing

In [23]:

```
large_df
```

Out[23]:

	is_spoiler	review_text
0	1	The second Tom Clancy novel made into a film (...
1	1	The second in what looks like becoming the 'Ja...
2	1	I was not a fan of The Hunt For Red October. I...
3	1	Jack Ryan (Harrison Ford) is a CIA analyst who...
4	1	This was one of the big summer movies of 1992....
...	...	...
538777	0	Dunkirk is a beautifully done movie that has h...
538778	0	Dunkirk is one of the rare cases a film receiv...
538779	0	Film gave insufficient background on what was ...
538780	0	In screen writing, a shot is an image captured...
538781	0	In a movie that entirely engulfs you it's rath...

538782 rows × 2 columns

In [22]:

```
seq_len = 512
num_samples = len(large_df)

num_samples, seq_len
```

Out[22]:

```
(538782, 512)
```

In [24]:
```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-cased')

tokens = tokenizer(large_df['review_text'].tolist(),
                   max_length=seq_len,
                   truncation=True,
                   padding='max_length',
                   add_special_tokens=True,
                   return_tensors='np')
```

In [25]:
```
with open('review-xids.npy', 'wb') as f:
    np.save(f, tokens['input_ids'])

with open('review-xmask.npy', 'wb') as f:
    np.save(f, tokens['attention_mask'])
```

In [ ]:
```
del tokens
```

In [26]:
```
arr = large_df['is_spoiler'].values
```

In [27]:
```
labels = np.zeros((num_samples, arr.max()+1))
labels.shape
```

Out[27]:
```
(538782, 2)
```

In [29]:
```
labels[np.arange(num_samples), arr] = 1
labels
```

Out[29]:
```
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       ...,
       [1., 0.],
       [1., 0.],
       [1., 0.]])
```

In [30]:
```
with open('review-labels.npy', 'wb') as f:
    np.save(f, labels)
```

input pipeline

In [3]:
```
with open('review-xids.npy', 'rb') as f:
    Xids = np.load(f, allow_pickle=True)
with open('review-xmask.npy', 'rb') as f:
    Xmask = np.load(f, allow_pickle=True)
```

```
with open('review-labels.npy', 'rb') as f:
    labels = np.load(f, allow_pickle=True)
```

In [4]:

```
dataset = tf.data.Dataset.from_tensor_slices((Xids, Xmask, labels))
```

In [5]:

```
dataset.take(1)
```

Out[5]:

```
<TakeDataset element_spec=(TensorSpec(shape=(512,), dtype=tf.int32, name=None), TensorSpec(shape=(512,), dtype=tf.int32, name=None), TensorSpec(shape=(2,), dtype=tf.float64, name=None))>
```

In [6]:

```
def map_func(input_ids, masks, labels):
    # we convert our three-item tuple into a two-item tuple where the input item is a d
    return {'input_ids': input_ids, 'attention_mask': masks}, labels

# then we use the dataset map method to apply this transformation
dataset = dataset.map(map_func)

dataset.take(1)
```

Out[6]:

```
<TakeDataset element_spec=({'input_ids': TensorSpec(shape=(512,), dtype=tf.int32, name=None), 'attention_mask': TensorSpec(shape=(512,), dtype=tf.int32, name=None)}, TensorSpec(shape=(2,), dtype=tf.float64, name=None))>
```

In [7]:

```
# shuffle data and batch it with batch size 256, dropping the remainder that don't fit

batch_size = 256

dataset = dataset.shuffle(10000).batch(batch_size, drop_remainder=True)

dataset.take(1)
```

Out[7]:

```
<TakeDataset element_spec=({'input_ids': TensorSpec(shape=(256, 512), dtype=tf.int32, name=None), 'attention_mask': TensorSpec(shape=(256, 512), dtype=tf.int32, name=None)}, TensorSpec(shape=(256, 2), dtype=tf.float64, name=None))>
```

In [8]:

```
# split data into train and validation, 80/20 split

split = 0.8

# we need to calculate how many batches must be taken to create 90% training set
size = int((Xids.shape[0] / batch_size) * split)

size
```

Out[8]:

```
1683
```

In [9]:

```
train_ds = dataset.take(size)
val_ds = dataset.skip(size)

# free up memory
del dataset
```

In [10]:
```
tf.data.experimental.save(train_ds, 'train')
tf.data.experimental.save(val_ds, 'val')
```

In [11]:
```
train_ds.element_spec
```

Out[11]:
```
({'input_ids': TensorSpec(shape=(256, 512), dtype=tf.int32, name=None),
  'attention_mask': TensorSpec(shape=(256, 512), dtype=tf.int32, name=None)},
 TensorSpec(shape=(256, 2), dtype=tf.float64, name=None))
```

In [12]:
```
val_ds.element_spec == train_ds.element_spec
```

Out[12]:
```
True
```

build and train

In [13]:
```
ds = tf.data.experimental.load('train', element_spec=train_ds.element_spec)
```

In [14]:
```
bert = TFAutoModel.from_pretrained('bert-base-cased')
```

```
Some layers from the model checkpoint at bert-base-cased were not used when initializing
TFBertModel: ['mlm___cls', 'nsp___cls']
- This IS expected if you are initializing TFBertModel from the checkpoint of a model tr
ained on another task or with another architecture (e.g. initializing a BertForSequenceC
lassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFBertModel from the checkpoint of a mode
l that you expect to be exactly identical (initializing a BertForSequenceClassification
model from a BertForSequenceClassification model).
All the layers of TFBertModel were initialized from the model checkpoint at bert-base-ca
sed.
If your task is similar to the task the model of the checkpoint was trained on, you can
already use TFBertModel for predictions without further training.
```

In [15]:
```
bert.summary()
```

```
Model: "tf_bert_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bert (TFBertMainLayer)      multiple                  108310272

=================================================================
Total params: 108,310,272
Trainable params: 108,310,272
Non-trainable params: 0
_________________________________________________________________
```

In [16]:
```
# two input layers, we ensure layer name variables match to dictionary keys in TF datas
input_ids = tf.keras.layers.Input(shape=(512,), name='input_ids', dtype='int32')
mask = tf.keras.layers.Input(shape=(512,), name='attention_mask', dtype='int32')

# we access the transformer model within our bert object using the bert attribute (eg b
embeddings = bert.bert(input_ids, attention_mask=mask)[1]  # access final activations (
```

```
# convert bert embeddings into binary output
x = tf.keras.layers.Dense(1024, activation='relu')(embeddings)
y = tf.keras.layers.Dense(2, activation='sigmoid', name='outputs')(x)
```

In [17]:

```
# initialize model
model = tf.keras.Model(inputs=[input_ids, mask], outputs=y)

# (optional) freeze bert layer
model.layers[2].trainable = False

# print out model summary
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to                     
==================================================================================================
 input_ids (InputLayer)         [(None, 512)]        0           []                               

 attention_mask (InputLayer)    [(None, 512)]        0           []                               

 bert (TFBertMainLayer)         TFBaseModelOutputWi  108310272   ['input_ids[0][0]',              
                                thPoolingAndCrossAt               'attention_mask[0]
[0]']                                                                                            
                                tentions(last_hidde                                               
                                n_state=(None, 512,                                               
                                 768),                                                            
                                 pooler_output=(Non                                               
                                e, 768),                                                          
                                 past_key_values=No                                               
                                ne, hidden_states=N                                               
                                one, attentions=Non                                               
                                e, cross_attentions                                               
                                =None)                                                            

 dense (Dense)                  (None, 1024)         787456      ['bert[0][1]']                   

 outputs (Dense)                (None, 2)            2050        ['dense[0][0]']                  

==================================================================================================
Total params: 109,099,778
Trainable params: 789,506
Non-trainable params: 108,310,272
__________________________________________________________________________________________________
```

In [18]:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-5, decay=1e-6)
loss = tf.keras.losses.BinaryCrossentropy()
acc = tf.keras.metrics.CategoricalAccuracy('accuracy')

model.compile(optimizer=optimizer, loss=loss, metrics=[acc])
```

In [19]:

```
element_spec = ({'input_ids': tf.TensorSpec(shape=(256, 512), dtype=tf.int32, name=None
                 'attention_mask': tf.TensorSpec(shape=(256, 512), dtype=tf.int32, name
                tf.TensorSpec(shape=(256, 2), dtype=tf.float64, name=None))

# load the training and validation sets
train_ds = tf.data.experimental.load('train', element_spec=element_spec)
val_ds = tf.data.experimental.load('val', element_spec=element_spec)

# view the input format
train_ds.take(1)
```

Out[19]:

```
<TakeDataset element_spec=({'input_ids': TensorSpec(shape=(256, 512), dtype=tf.int32, na
me=None), 'attention_mask': TensorSpec(shape=(256, 512), dtype=tf.int32, name=None)}, Te
nsorSpec(shape=(256, 2), dtype=tf.float64, name=None))>
```

In [ ]:

```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=3
)
```

```
Epoch 1/3
  14/1683 [..............................] - ETA: 196:35:07 - loss: 0.6148 - accuracy:
0.7355
```

In [ ]: