Milestone 2

CSML1010 Project, Dec 22, 2019

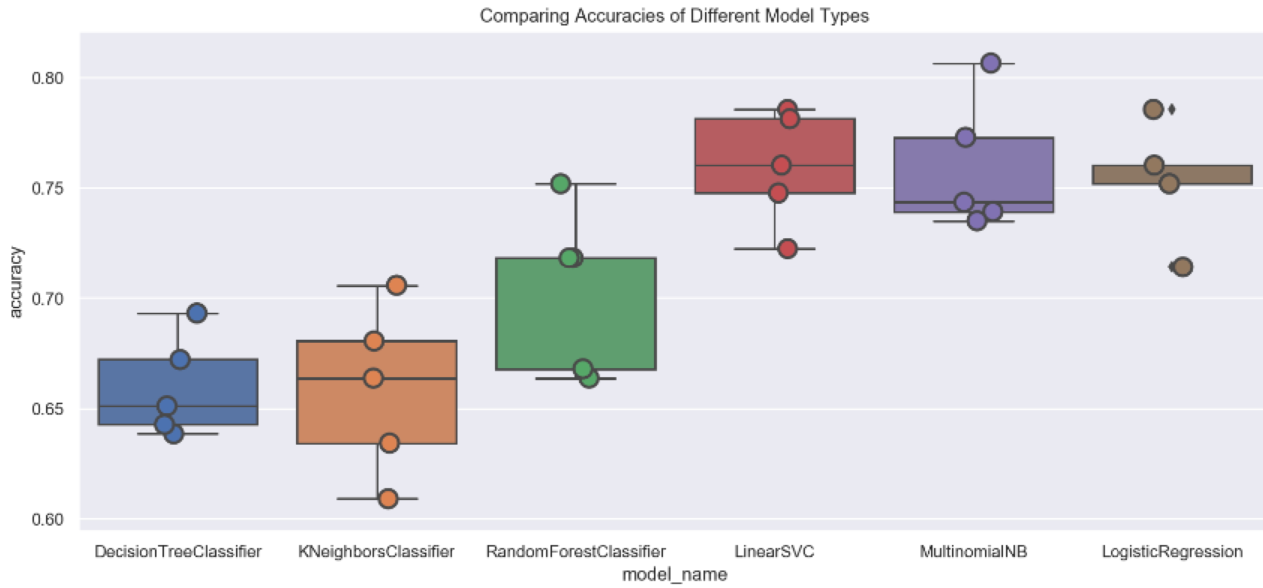
Pete Gray YorkU #217653247

Algorithm Implementation, Model Evaluation and Selection, plus Ensemble Methods

For ease of reading, summaries and samples of graphs will be presented first, followed by the complete code, output, and markdown for all Milestone 2 activities.

Comparing Model Accuracy

6 models were compared based on their accuracy with our dataset, Decision Tree, K-Neighbours, Random Forest, LinearSVC, MultinomialNB, and Logistic Regression. The comparison chart clearly shows superior performance by the latter 3, and we find in further explorations that these conclusions do clearly reflect the abilities of these classifiers in this context. We see in different evaluations that LinearSVC, MultinomialNB, and Logistic Regression perform roughly equally, but consistently superior to Decision Tree, K-Neighbours, and Random Forest.




```
<Figure size 1152x432 with 0 Axes>
```

```
: cv_df.groupby('model_name').accuracy.mean()
```

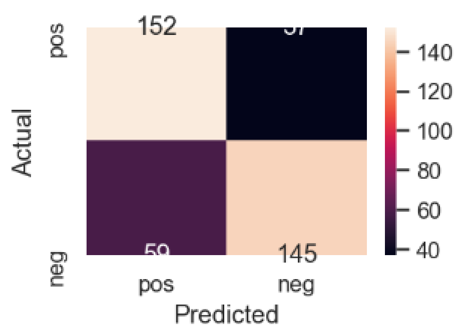
```
: model_name
  DecisionTreeClassifier   0.66
  KNeighborsClassifier     0.66
  LinearSVC                0.76
  LogisticRegression       0.75
  MultinomialNB            0.76
  RandomForestClassifier   0.70
  Name: accuracy, dtype: float64
```

Evaluation with Confusion Matrix, Precision, Recall, and F1-Score

These metrics are used on each of the higher performing models as well as a lower-performing model, for confirmation. The highest F1-scoring models were Logistic Regression and MultinomialNB, with LinearSVC a very close second. Random Forest did not perform as well. Please see full markdown below for all the numbers.

The heatmaps are a little trivial in the 2-labels context, but I kept little ones just as an extra little visual reference.

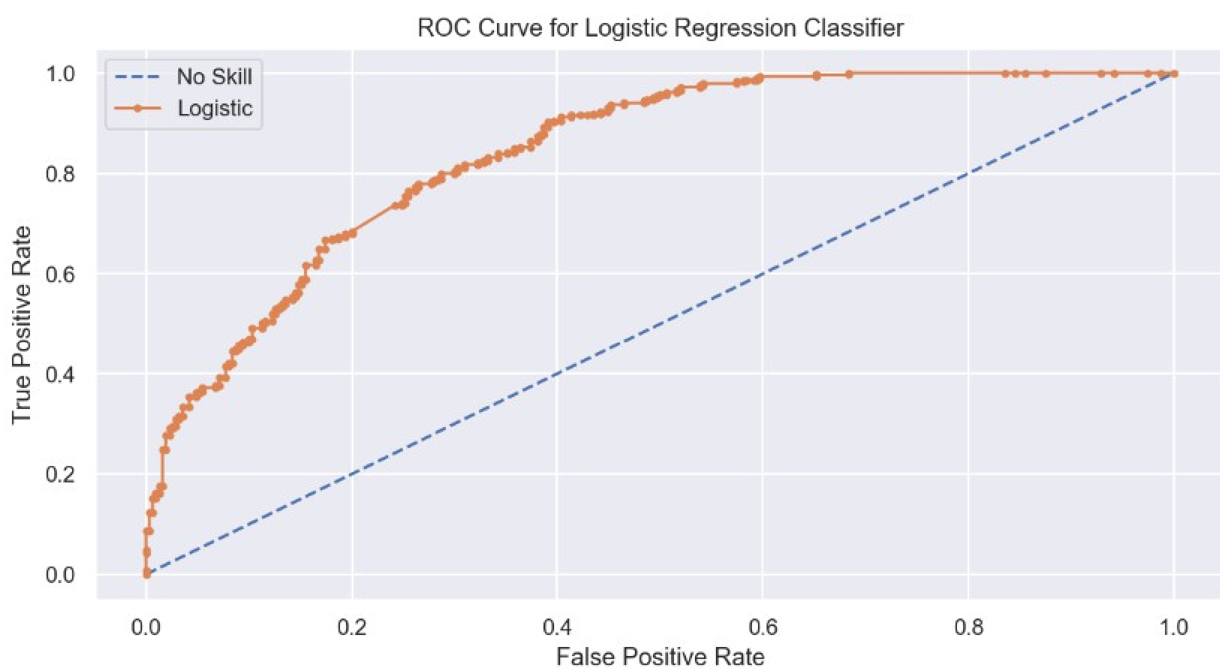
```
Confusion Matrix for Multinomial Naive Bayes Classifier:

[[152  37]
 [ 59 145]]

Confusion Matrix for Multinomial Naive Bayes Classifier as a little heatmap:
```




```
Precision, Recall, F1-scores for Multinomial Naive Bayes Classifier:

              precision    recall  f1-score   support

           0       0.72      0.80      0.76       189
           4       0.80      0.71      0.75       204

    accuracy                           0.76       393
   macro avg       0.76      0.76      0.76       393
weighted avg       0.76      0.76      0.76       393
```

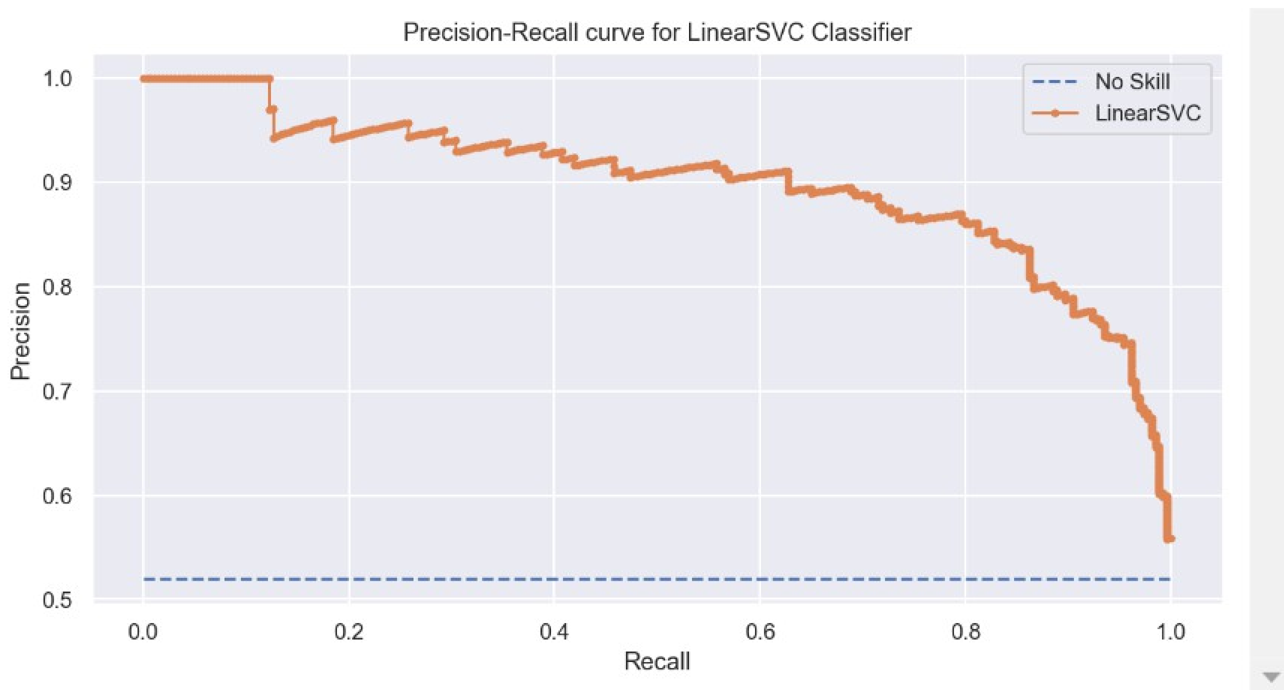
ROC Curves

ROC Curves were used to evaluate the skill of the models. Again, the highest AUC (Area Under the Curve) scores went to Regression, SVC, and MultinomialNB. Changes to feature selection or sample size resulted in different rankings between them, but through any changes their scores remain very close to one another.



Precision-Recall Curves

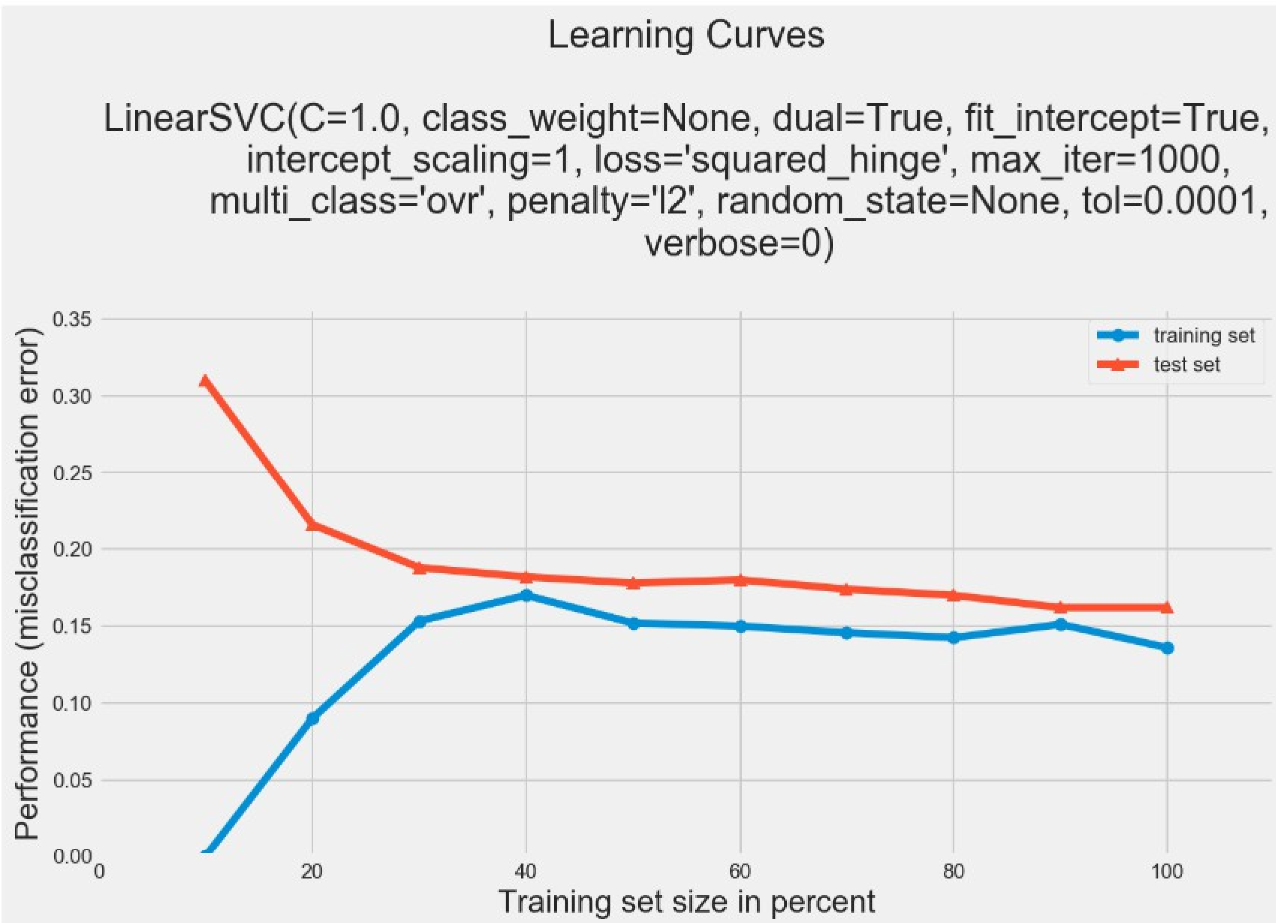
A small number of models were evaluated with Precision-Recall curves. As these are more suited to imbalanced datasets, and our data is as balanced as it can get, not many were made, and their scores will not figure into our model selection process.



Learning Curves

Learning Curves were plotted for each of the models. In some cased, as pictured, things looked good. For some models, erratic behavior or possibly overtraining could be seen.

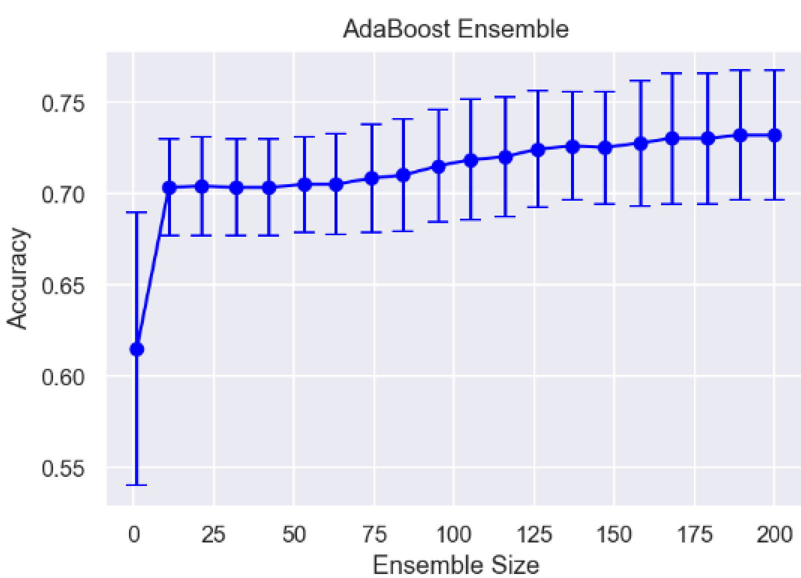
```
plt.show()
```



Ensemble Methods, and Ensemble Size Selection

A number of Ensemble Methods are attempted. None have performed as well as a standalone model at this point. In this graph, we can see an AdaBoost Ensemble's performance given various ensemble sizes. A reasonable level of accuracy is achieved with a small number, roughly a dozen, model in the ensemble. The accuracy appears to continue to rise up to 200, at which point it is comparable to a standalone model. Other ensemble methods yielded results even odder than that. Please see code, output, and markdown below for more details.

```
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('AdaBoost Ensemb
plt.show()
```



Please find the complete code, output, and markdown attached directly below.

Thank you.

In [ ]:

```

```

------> CUT

Code, output, and markdown for everything up to Milestone 1 has been removed for ease of Milestone 2

Beginning of Milestone 2

Working models.

Model Selection

In this section we will train and test models using 6 different algorithms, and then compare their accuracy.

This can help us understand which type of models works best with our data, and allow us to identify problems using certain types of models with our data.

In [81]:

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB

X_train, X_test, y_train, y_test = train_test_split(df_sm['text_nav'], df_sm['se
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X_train)
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)

#clf = MultinomialNB().fit(X_train_tfidf, y_train)
clf = MultinomialNB().fit(X_train_tfidf, y_train)
```

In [82]:

```
print(clf.predict(count_vect.transform(["AArgh, this is the silliest thing ever"
```

```
[4]
```

In [83]:

```
print(clf.predict(count_vect.transform(["I absolutely love this code."])))
```

```
[4]
```

In [84]:

```
print(clf.predict(count_vect.transform(["Not sure what I think of this one."])))
```

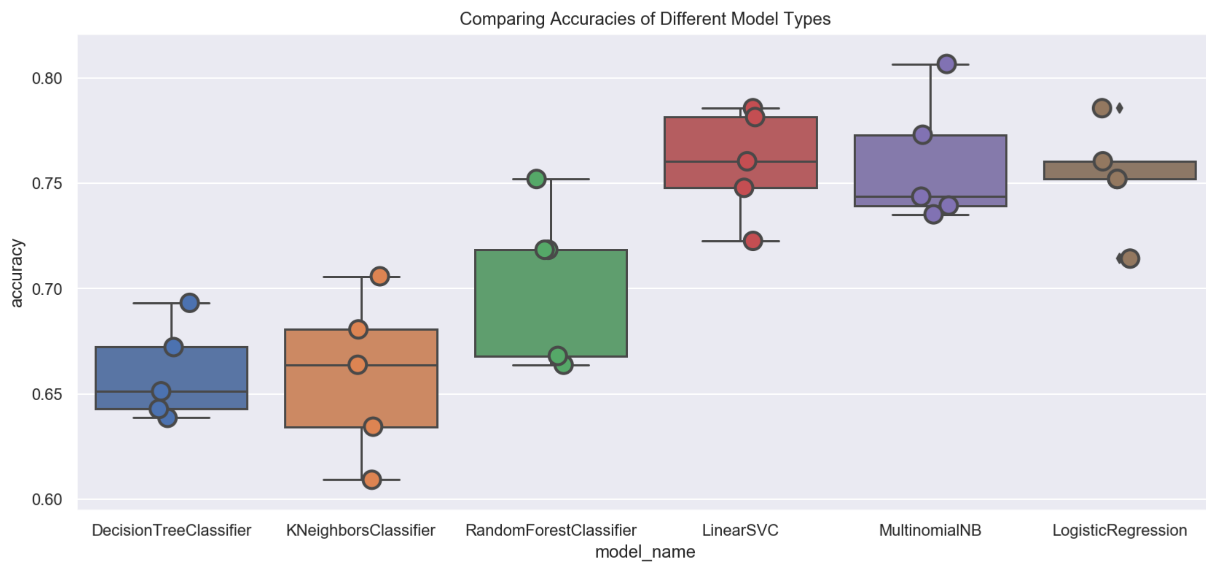
```
[4]
```

In [85]:
```
print(clf.predict(count_vect.transform(["Why would you do that?"])))
```

```
[0]
```

In [165]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import cross_val_score



models = [
    DecisionTreeClassifier(criterion='entropy', max_depth=2),
    KNeighborsClassifier(n_neighbors=1),
    RandomForestClassifier(n_estimators=200, max_depth=3, random_state=0),
    LinearSVC(),
    MultinomialNB(),
    LogisticRegression(random_state=0),

]
CV = 5
cv_df = pd.DataFrame(index=range(CV * len(models)))
entries = []
for model in models:
  model_name = model.__class__.__name__
  accuracies = cross_val_score(model, tv_matrix, df_sm['sentiment'], scoring='ac
  for fold_idx, accuracy in enumerate(accuracies):
    entries.append((model_name, fold_idx, accuracy))
cv_df = pd.DataFrame(entries, columns=['model_name', 'fold_idx', 'accuracy'])
```

In [169]:
```
import seaborn as sns

sns.set(rc={'figure.figsize':(14,6)})

sns.boxplot(x='model_name', y='accuracy', data=cv_df, width=0.8).set_title('Compa
sns.stripplot(x='model_name', y='accuracy', data=cv_df,
              size=12, jitter=True, edgecolor="gray", linewidth=2)
plt.figure(figsize=(16, 6))
#plt.set_size_inches(18.5, 10.5)
plt.savefig('test2png.png', dpi=100)
plt.show()
```




```
<Figure size 1152x432 with 0 Axes>
```

In [172]:
```
cv_df.groupby('model_name').accuracy.mean()
```

Out[172]:
```
model_name
DecisionTreeClassifier    0.66
KNeighborsClassifier      0.66
LinearSVC                 0.76
LogisticRegression        0.75
MultinomialNB             0.76
RandomForestClassifier    0.70
Name: accuracy, dtype: float64
```

Observation:

My first attempts at Ensembling yielded pretty dubious results. More dubious than some of these standalone models. Looking at this graphs helps explain why - the code I was originally using for Ensemble Methods used Decision Tree and K-Nearest-Neighbour. This graph suggests that these models perform poorly compared to SVC, NB, etc. Which gives us something to go on, in our quest to make the Ensemble Methods produce better results - Ensemble more powerful models!

===================================

LinearSVC (*Support Vector Classifier*) Model

Split into training and test sets, train model on training set, generate predictions on test set.

In [217]:
```
from sklearn.model_selection import train_test_split

model_svc = LinearSVC()

X_train, X_test, y_train, y_test, indices_train, indices_test = train_test_split
model_svc.fit(X_train, y_train)
y_pred = model_svc.predict(X_test)
```

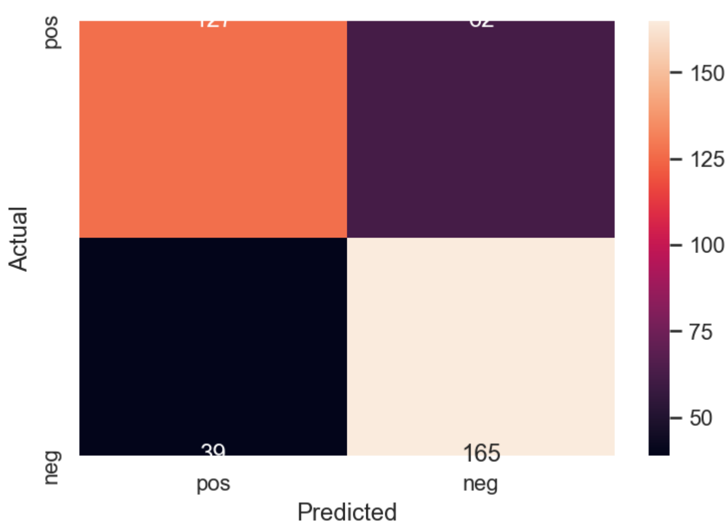
Confusion Matrix: LinearSVC

Calculate, and display, true positives, true negatives, false positives, and false negatives.

We'll show both the raw matrix, and a "Heatmap", which looks cool, but doesn't give too much insight with only two labels.

In [218]:
```
from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(y_test, y_pred)
print('RAW CONFUSION MATRIX', '\n')
print(conf_mat, '\n')
print('Because the heatmap may LOOK cool, but doesn\'t tell us much, with only tw
print('--------------------\n')
print('Heatmap of Consfusion Matrix:')
fig, ax = plt.subplots(figsize=(6,4))
sns.heatmap(conf_mat, annot=True, fmt='d',
            xticklabels=['pos','neg'], yticklabels=['pos','neg'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
RAW CONFUSION MATRIX

[[127  62]
 [ 39 165]]

Because the heatmap may LOOK cool, but doesn't tell us much, with only two labe
ls

--------------------

Heatmap of Consfusion Matrix:
```



In [219]:
```
model_svc.fit(tv_matrix, df_sm['sentiment'])
```

Out[219]:
```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)
```

Precision, Recall and f1-Score: LinearSVC

In [220]:
```
from sklearn import metrics
print(metrics.classification_report(y_test, y_pred,
                                    target_names=['0','4']))
```

```
              precision    recall  f1-score   support

           0       0.77      0.67      0.72       189
           4       0.73      0.81      0.77       204

    accuracy                           0.74       393
   macro avg       0.75      0.74      0.74       393
weighted avg       0.75      0.74      0.74       393
```

ROC Curve: LinearSVC

Using code and wisdom from https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/ (https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/)

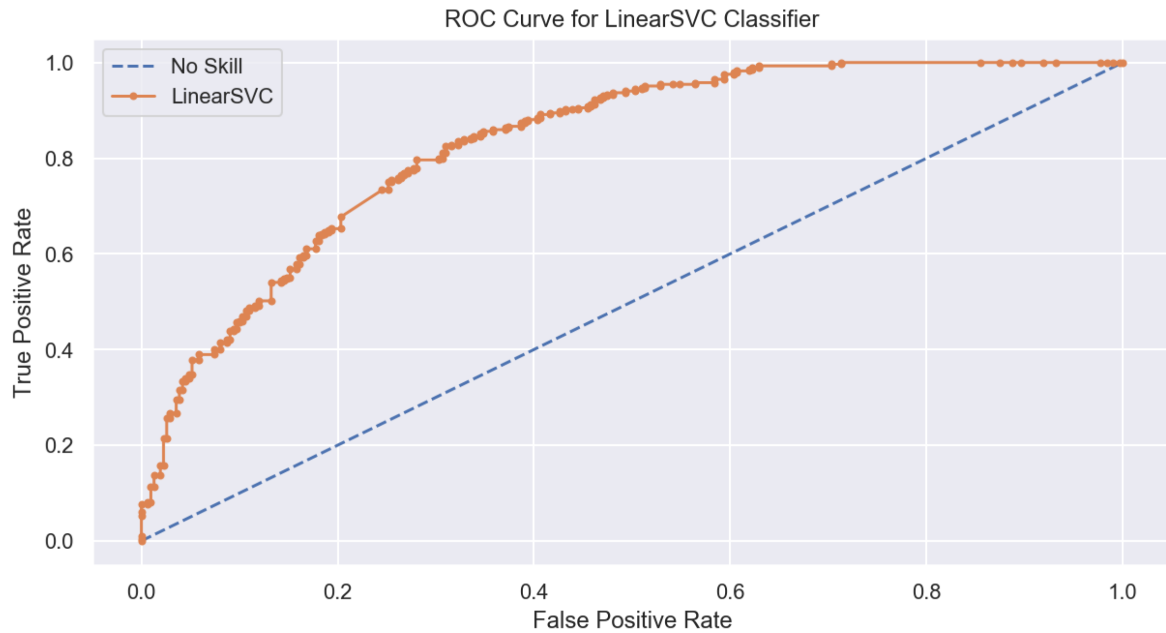
A useful tool when predicting the probability of a binary outcome is the **Receiver Operating Characteristic curve**, or ROC curve.

It is a plot of the false positive rate (x-axis) versus the true positive rate (y-axis) for a number of different candidate threshold values between 0.0 and 1.0. Put another way, it plots the false alarm rate versus the hit rate.

In [221]:

```
# roc curve and auc
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from matplotlib import pyplot

# Shrink back our Seaborn graph size - don't need it as big as we had for the acc
sns.set(rc={'figure.figsize':(10,5)})

# generate 2 class dataset
#X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
X = tv_matrix
y = df_sm['sentiment']
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state
# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(testy))]
# fit a model
# model = LogisticRegression(solver='lbfgs')

# We use our model as defined above
model_svc.fit(trainX, trainy)
# predict probabilities
# use _predict_proba_lr(testX) with SVC
lr_probs = model_svc._predict_proba_lr(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# calculate scores
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('LinearSVC: ROC AUC=%.3f' % (lr_auc))
print('\nROC Curve for LinearSVC Classifier:')
# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs, pos_label=4)
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs, pos_label=4)
# plot the roc curve for the model
pyplot.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
pyplot.plot(lr_fpr, lr_tpr, marker='.', label='LinearSVC')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.title('ROC Curve for LinearSVC Classifier')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

```
No Skill: ROC AUC=0.500
LinearSVC: ROC AUC=0.831

ROC Curve for LinearSVC Classifier:
```



This ROC curve shows considerably higher "skill" than the "no skill" curve, which represents the perfomance of a classifier that just guesses the most likely thing.

AUC - Area Under the Curve

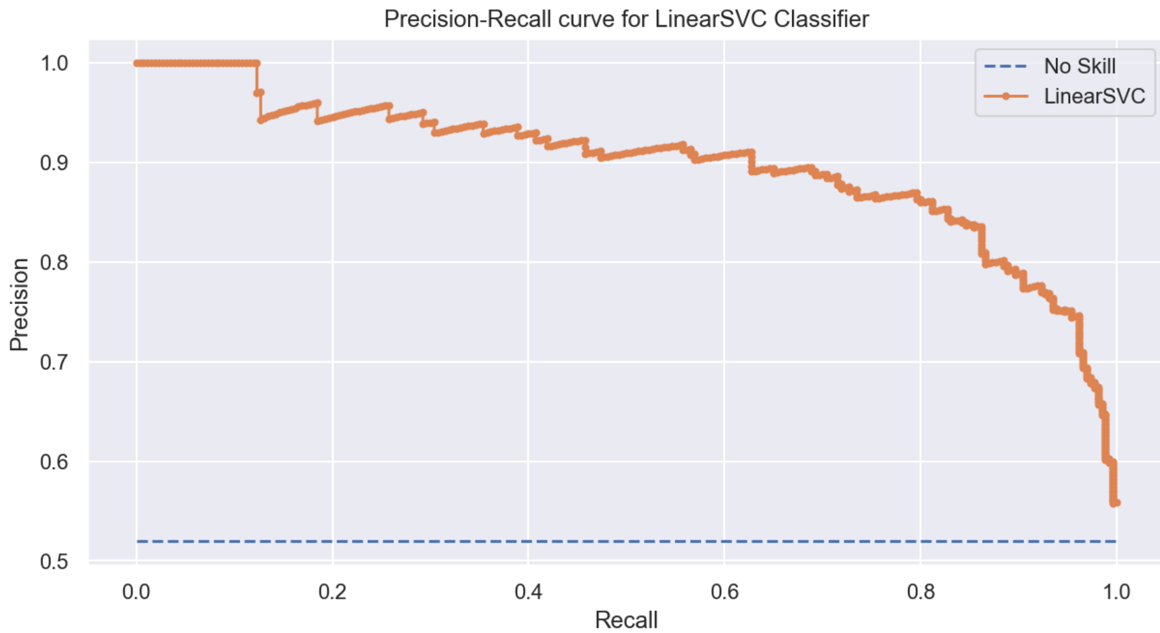
The "skill" of this model can be represented by the AUC score - the Area Under the Curve - and we do note that despite the LinearSVC model having the highest accuracy and f1-score of the models we tried, Logistic Regression appears to have a higher AUC, as we will see below.

Precision-Recall Curve: LinearSVC Model

In [222]:

```python
# precision-recall curve and f1
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_stat
# fit a model
# model = LogisticRegression(solver='lbfgs')

# use our model
model = model_svc

model.fit(trainX, trainy)
# predict probabilities
lr_probs = model._predict_proba_lr(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# predict class values
yhat = model.predict(testX)
lr_precision, lr_recall, _ = precision_recall_curve(testy, lr_probs)
lr_f1, lr_auc = f1_score(testy, yhat), auc(lr_recall, lr_precision)
# summarize scores
print('LinearSVC: f1=%.3f auc=%.3f' % (lr_f1, lr_auc))
# plot the precision-recall curves
no_skill = len(testy[testy==1]) / len(testy)
pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
pyplot.plot(lr_recall, lr_precision, marker='.', label='LinearSVC')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.title('Precision-Recall curve for LinearSVC Classifier')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

```
LinearSVC: f1=0.846 auc=0.899
```



The precision-recall curve plot show the precision/recall for each threshold for a Linear Support Vector Classifier model (orange) compared to a no skill model (blue).

When to Use ROC vs. Precision-Recall Curves?

Again from [https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/ (https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/)](https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/)

Generally, the use of ROC curves and precision-recall curves are as follows:

- ROC curves should be used when there are roughly equal numbers of observations for each class.
- Precision-Recall curves should be used when there is a moderate to large class imbalance.

The reason for this recommendation is that ROC curves present an optimistic picture of the model on datasets with a class imbalance.

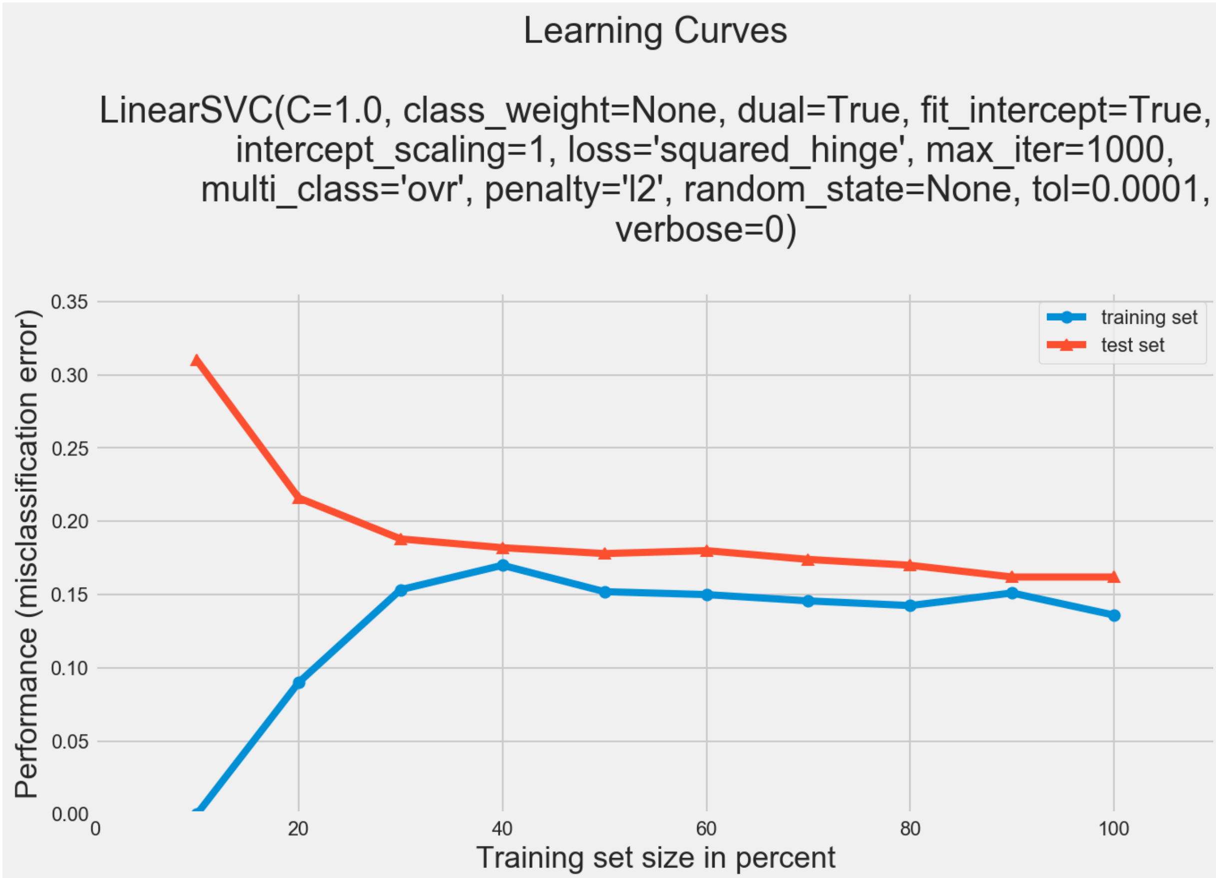
Some go further and suggest that using a ROC curve with an imbalanced dataset might be deceptive and lead to incorrect interpretations of the model skill.

The main reason for this optimistic picture is because of the use of true negatives in the False Positive Rate in the ROC Curve and the careful avoidance of this rate in the Precision-Recall curve.

As our dataset is very very balanced, we will lean towards ROC.

Learning Curves: LinearSVC

In [224]:
```
from mlxtend.plotting import plot_learning_curves
#plot_learning_curves(X_train, y_train, X_test, y_test, model_svc)
plot_learning_curves(trainX, trainy, testX, testy, model_svc)
plt.show()
```



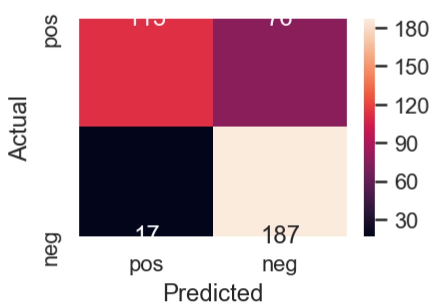
==================================

Logistic Regression

We'll do the same metrics for Logistic Regression as we did for LinearSVC

In [226]:
```
model_lr = LogisticRegression(random_state=0)
model_lr.fit(X_train, y_train)
y_pred = model_lr.predict(X_test)
conf_mat = confusion_matrix(y_test, y_pred)
print('\nConfusion Matrix for Logistic Regression Model: \n')
print(conf_mat , '\n')
print('Confusion Matrix for Logistic Regression Model as a little heatmap:')
fig, ax = plt.subplots(figsize=(3,2))
sns.heatmap(conf_mat, annot=True, fmt='d',
            xticklabels=['pos','neg'], yticklabels=['pos','neg'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
print('\nPrecision, Recall, F1-scores for Logistic Regression Model:', '\n')
print(metrics.classification_report(y_test, y_pred,
                                    target_names=['0','4']))
```

```
Confusion Matrix for Logistic Regression Model:

[[113  76]
 [ 17 187]]

Confusion Matrix for Logistic Regression Model as a little heatmap:
```



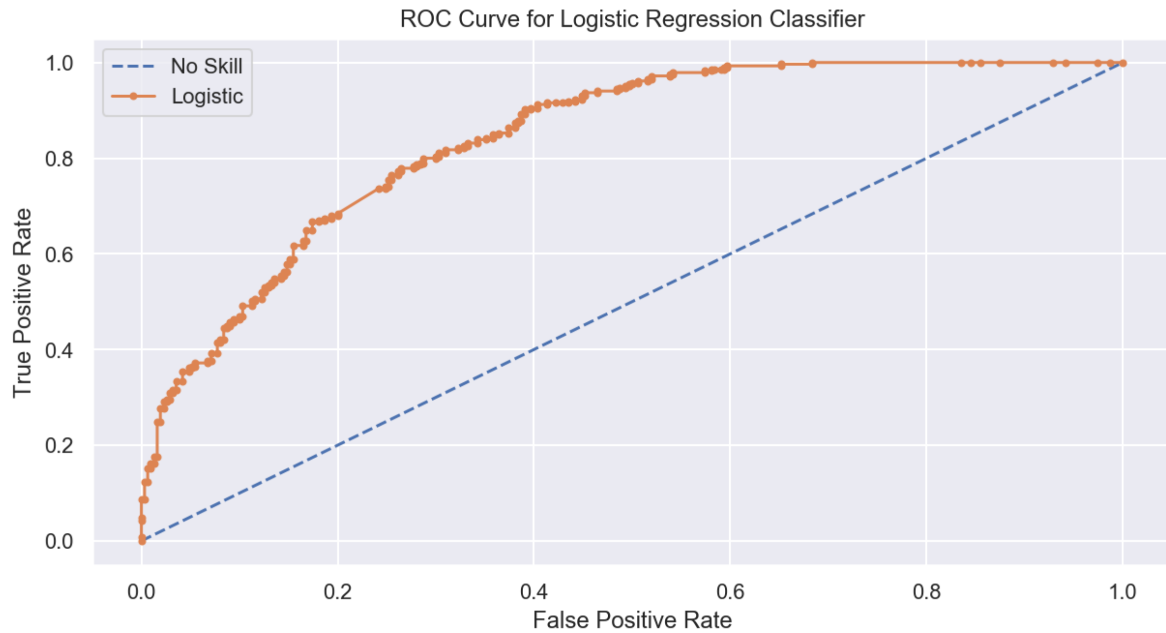

```
Precision, Recall, F1-scores for Logistic Regression Model:

              precision    recall  f1-score   support

           0       0.87      0.60      0.71       189
           4       0.71      0.92      0.80       204

    accuracy                           0.76       393
   macro avg       0.79      0.76      0.75       393
weighted avg       0.79      0.76      0.76       393
```

ROC Curve: Logistic Regression Classifier

In [227]:

```
# Use our vectorized and feature-selected data
X = tv_matrix
y = df_sm['sentiment']

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_stat
# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(testy))]

# use our model as defined above
model = model_lr

#LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# calculate scores
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('Logistic: ROC AUC=%.3f' % (lr_auc))
print('\nROC curve for Logistic Regression Classifier:')
# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs, pos_label=4)
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs, pos_label=4)
# plot the roc curve for the model
pyplot.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
pyplot.plot(lr_fpr, lr_tpr, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.title('ROC Curve for Logistic Regression Classifier')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

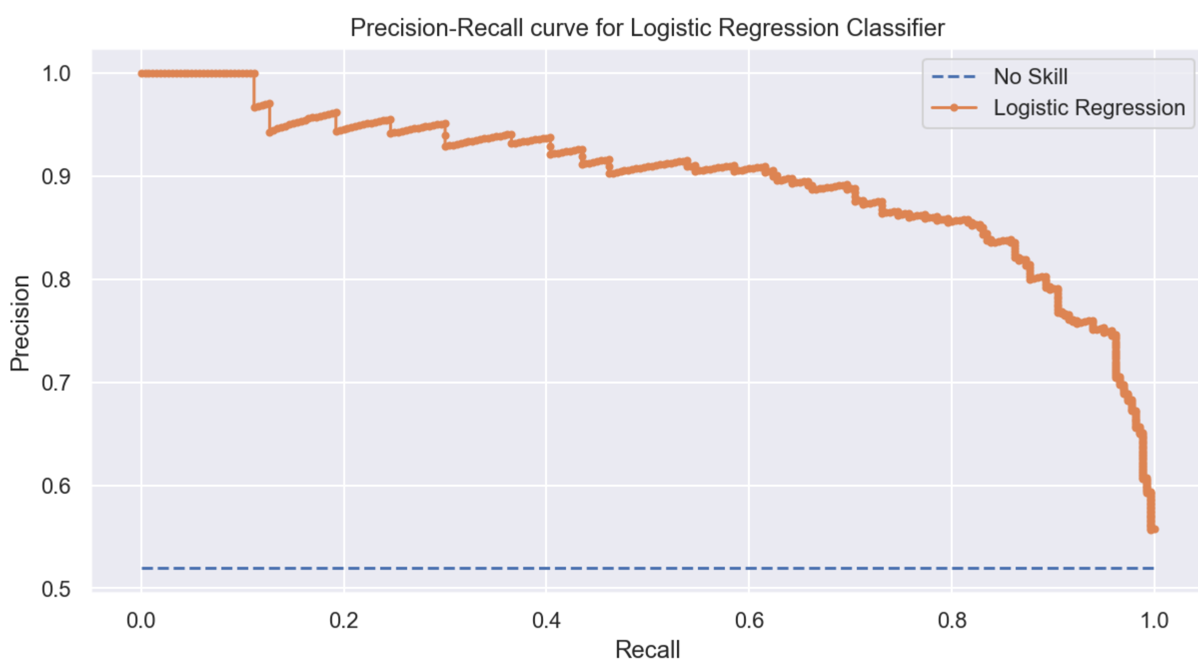
```
No Skill: ROC AUC=0.500
Logistic: ROC AUC=0.841

ROC curve for Logistic Regression Classifier:
```



Precision-Recall Curve: Logistic Regression

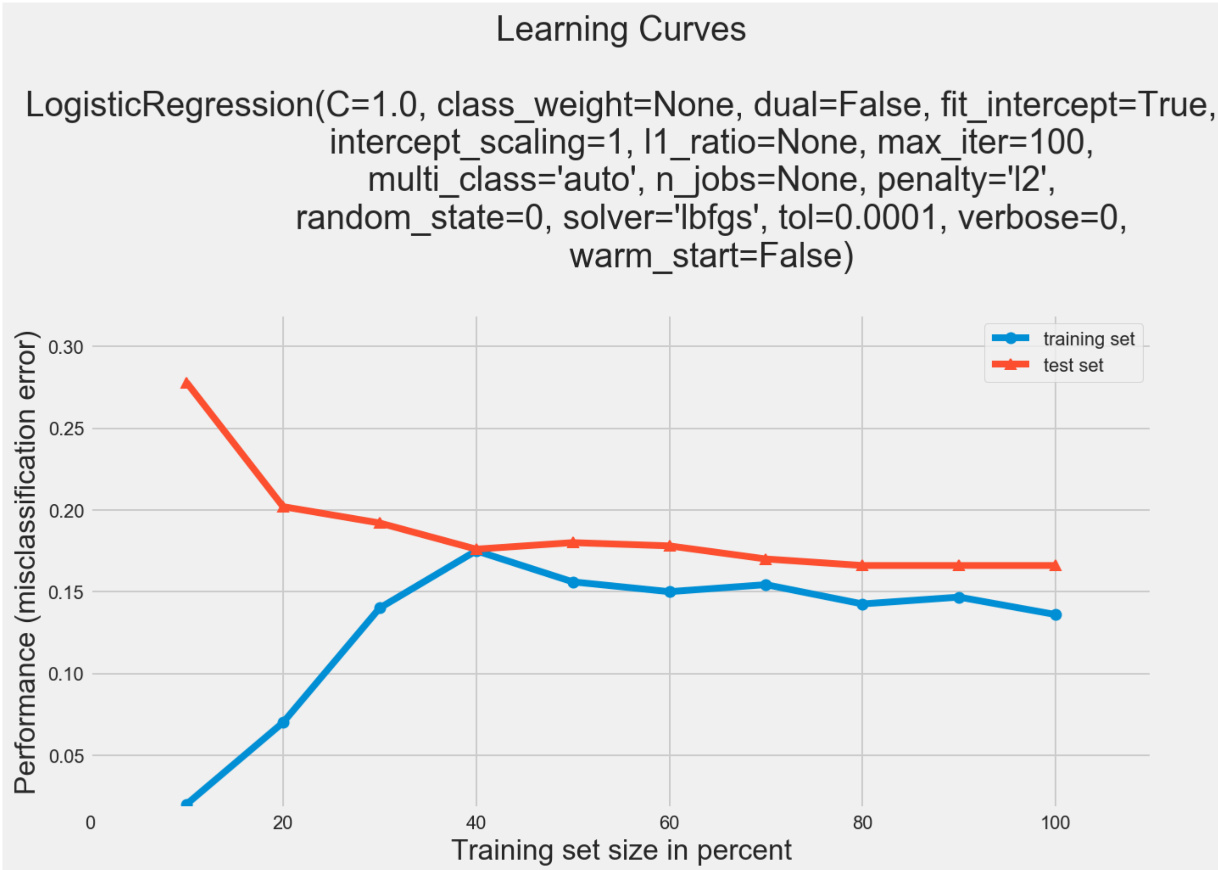
In [228]:
```
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_stat
# fit a model
# model = LogisticRegression(solver='lbfgs')

# use our model
model = model_lr

model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# predict class values
yhat = model.predict(testX)
lr_precision, lr_recall, _ = precision_recall_curve(testy, lr_probs)
lr_f1, lr_auc = f1_score(testy, yhat), auc(lr_recall, lr_precision)
# summarize scores
print('Logistic Regression: f1=%.3f auc=%.3f' % (lr_f1, lr_auc))
# plot the precision-recall curves
no_skill = len(testy[testy==1]) / len(testy)
pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
pyplot.plot(lr_recall, lr_precision, marker='.', label='Logistic Regression')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.title('Precision-Recall curve for Logistic Regression Classifier')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

```
Logistic Regression: f1=0.841 auc=0.898
```



Learning Curves: Logistic Regression

In [229]:
```
from mlxtend.plotting import plot_learning_curves
#plot_learning_curves(X_train, y_train, X_test, y_test, model_svc)
plot_learning_curves(trainX, trainy, testX, testy, model_lr)
plt.show()
```



================================

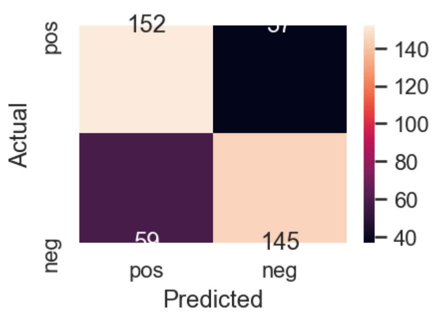
Multinomial NB Classifier

The **Multinomial Naive Bayes classifier** is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

In [231]:

```python
model_mnb = MultinomialNB()
model_mnb.fit(X_train, y_train)
y_pred = model_mnb.predict(X_test)
conf_mat = confusion_matrix(y_test, y_pred)
print('\nConfusion Matrix for Multinomial Naive Bayes Classifier: \n')
print(conf_mat , '\n')
print('Confusion Matrix for Multinomial Naive Bayes Classifier as a little heatma
fig, ax = plt.subplots(figsize=(3,2))
sns.heatmap(conf_mat, annot=True, fmt='d',
            xticklabels=['pos','neg'], yticklabels=['pos','neg'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
print('\nPrecision, Recall, F1-scores for Multinomial Naive Bayes Classifier:',
print(metrics.classification_report(y_test, y_pred,
                                    target_names=['0','4']))
```

```
Confusion Matrix for Multinomial Naive Bayes Classifier:

[[152  37]
 [ 59 145]]

Confusion Matrix for Multinomial Naive Bayes Classifier as a little heatmap:
```




```
Precision, Recall, F1-scores for Multinomial Naive Bayes Classifier:

              precision    recall  f1-score   support

           0       0.72      0.80      0.76       189
           4       0.80      0.71      0.75       204

    accuracy                           0.76       393
   macro avg       0.76      0.76      0.76       393
weighted avg       0.76      0.76      0.76       393
```

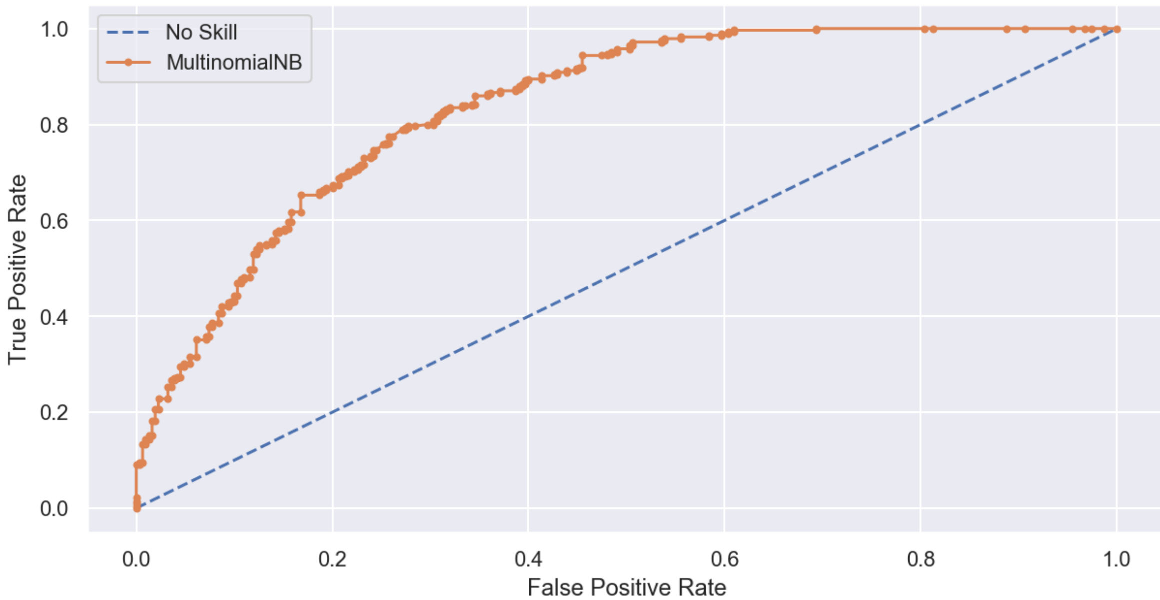
ROC and AUC: MultinomialNB

As with above.

In [232]:

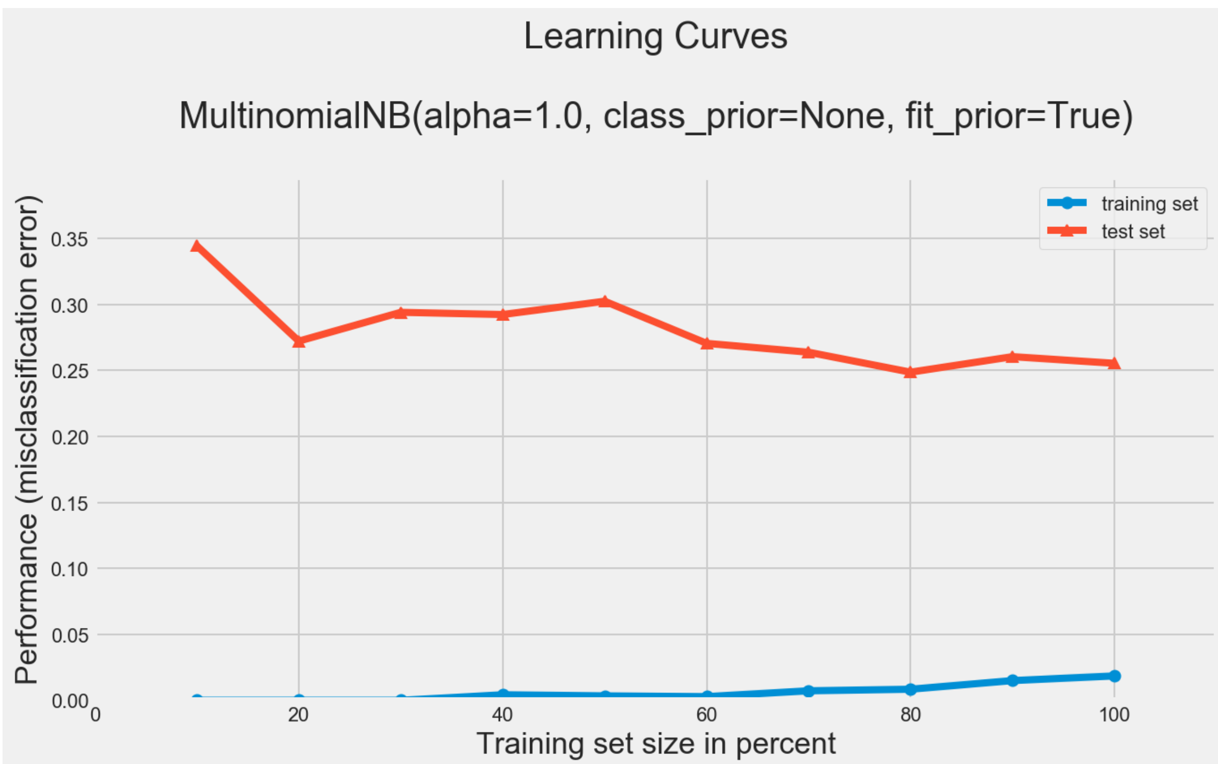
```python
# Use our vectorized and feature-selected data
X = tv_matrix
y = df_sm['sentiment']

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_stat
# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(testy))]

# use our model as defined above
model = model_mnb

#LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# calculate scores
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('MultinomialNB: ROC AUC=%.3f' % (lr_auc))
print('\nROC Curve for MultinomialNB:')
# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs, pos_label=4)
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs, pos_label=4)
# plot the roc curve for the model
pyplot.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
pyplot.plot(lr_fpr, lr_tpr, marker='.', label='MultinomialNB')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

```
No Skill: ROC AUC=0.500
MultinomialNB: ROC AUC=0.837

ROC Curve for MultinomialNB:
```



Learning Curves: MultinomialNB

In [233]:
```
plot_learning_curves(trainX, trainy, testX, testy, model_mnb)
plt.show()
```



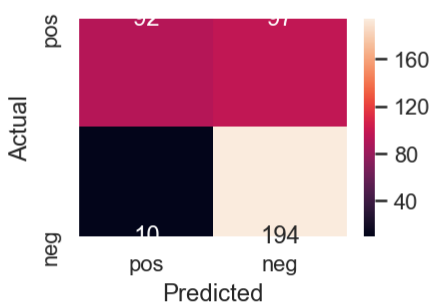
==

Random Forest Classifier

It did perform poorly on the "accuracy" comparison, especially with the really small dataset, but as we've seen above, accuracy is a poor predictor of other performance measures.

In [234]:
```
model_rfc = RandomForestClassifier(n_estimators=200, max_depth=20, random_state=(
#model_mnb = MultinomialNB()
model_rfc.fit(X_train, y_train)
y_pred = model_rfc.predict(X_test)
conf_mat = confusion_matrix(y_test, y_pred)
print('\nConfusion Matrix for Random Forest Classifier: \n')
print(conf_mat , '\n')
print('Confusion Matrix for Random Forest Classifier as a little heatmap:')
fig, ax = plt.subplots(figsize=(3,2))
sns.heatmap(conf_mat, annot=True, fmt='d',
            xticklabels=['pos','neg'], yticklabels=['pos','neg'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
print('\nPrecision, Recall, F1-scores for Random Forest Classifier Classifier:',
print(metrics.classification_report(y_test, y_pred,
                                    target_names=['0','4']))
```

```
Confusion Matrix for Random Forest Classifier:

[[ 92  97]
 [ 10 194]]

Confusion Matrix for Random Forest Classifier as a little heatmap:
```




```
Precision, Recall, F1-scores for Random Forest Classifier Classifier:

              precision    recall  f1-score   support

           0       0.90      0.49      0.63       189
           4       0.67      0.95      0.78       204

    accuracy                           0.73       393
   macro avg       0.78      0.72      0.71       393
weighted avg       0.78      0.73      0.71       393
```

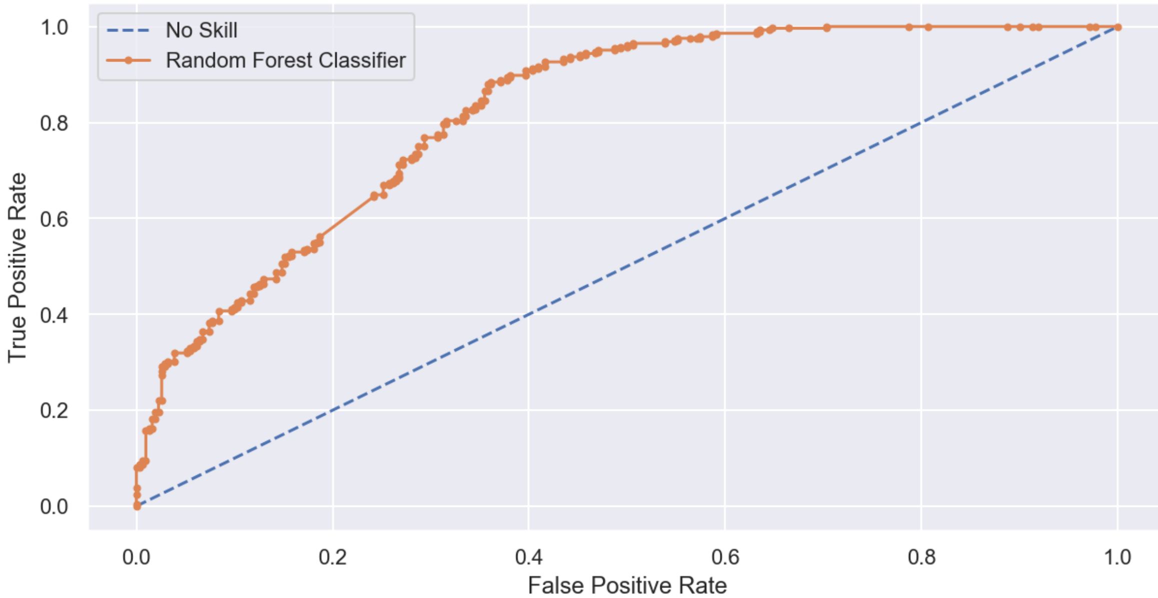
Okay! That is our worst one so far, both in terms of accuracy AND f1-score. Let's do the ROC curve!

ROC Curve and AUC metric: Random Forest

In [235]:

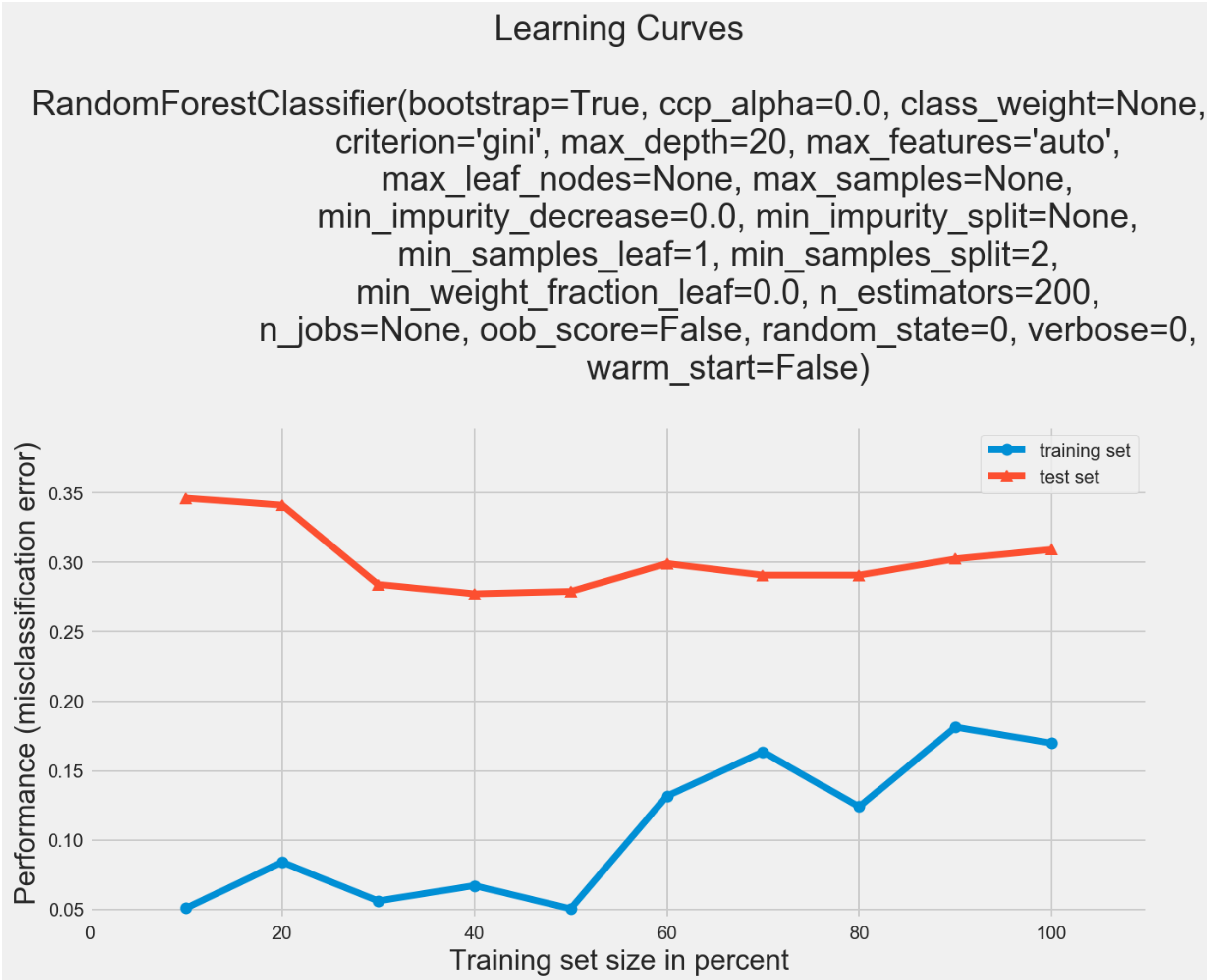
```
# Use our vectorized and feature-selected data
X = tv_matrix
y = df_sm['sentiment']

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_stat
# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(testy))]

# use our model as defined above
model = model_rfc

#LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# calculate scores
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('Random Forest Classifier: ROC AUC=%.3f' % (lr_auc))
print('\nROC Curve for Random Forest Classifier:')
# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs, pos_label=4)
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs, pos_label=4)
# plot the roc curve for the model
pyplot.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
pyplot.plot(lr_fpr, lr_tpr, marker='.', label='Random Forest Classifier')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

```
No Skill: ROC AUC=0.500
Random Forest Classifier: ROC AUC=0.821

ROC Curve for Random Forest Classifier:
```



Learning Curve: Random Forest

In [236]:
```
from mlxtend.plotting import plot_learning_curves
#plot_learning_curves(X_train, y_train, X_test, y_test, model_svc)
plot_learning_curves(trainX, trainy, testX, testy, model_rfc)
plt.show()
```



Interesting points of comparison!!

Early in this project, I was doing sentiment predictions with Afinn, to be used as a baseline later. At our first arrival at this point, things are looking better already!

Weighted Avg F1-Scores

0.63 - Afinn original values.

0.70 - Logistic Regression, 800 rows total, Tf-idf encoding

0.74 - LinearSVC, 800 rows total, Tf-idf encoding

===================================

Ensembling

================================

Ensemble methods are meta-algorithms that combine several machine learning techniques into one predictive model in order to decrease variance (bagging), bias (boosting), or improve predictions (stacking). Ensemble methods can be divided into two groups: sequential ensemble methods where the base learners are generated sequentially (e.g. AdaBoost) and parallel ensemble methods where the base learners are generated in parallel (e.g. Random Forest). The basic motivation of sequential methods is to exploit the dependence between the base learners since the overall performance can be boosted by weighing previously mislabeled examples with higher weight. The basic motivation of parallel methods is to exploit independence between the base learners since the error can be reduced dramatically by averaging.

Most ensemble methods use a single base learning algorithm to produce homogeneous base learners, i.e. learners of the same type leading to homogeneous ensembles. There are also some methods that use heterogeneous learners, i.e. learners of different types, leading to heterogeneous ensembles. In order for ensemble methods to be more accurate than any of its individual members the base learners have to be as accurate as possible and as diverse as possible.

Bagging

Bagging stands for **bootstrap aggregation**. One way to reduce the variance of an estimate is to average together multiple estimates. For example, we can train M different trees f_m on different subsets of the data (chosen randomly with replacement) and compute the ensemble:

$$f(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x)$$

Translation: *The final result is the average of the M sub-results.*

Bagging uses bootstrap sampling to obtain the data subsets for training the base learners. For aggregating the outputs of base learners, bagging uses voting for classification and averaging for regression.

In [237]:
```
%matplotlib inline

import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

np.random.seed(0)
```

In the coding example used for this section of the project, ensemble methods are used on Decision Tree Classifiers and K Nearest Neighbour Classifiers.

As we saw in our comparison on standalone models, these are the two worst, of the six we tried. We will instead use Logistic Regression and a Linear Support Vector Classifier for our Ensemble Methods.

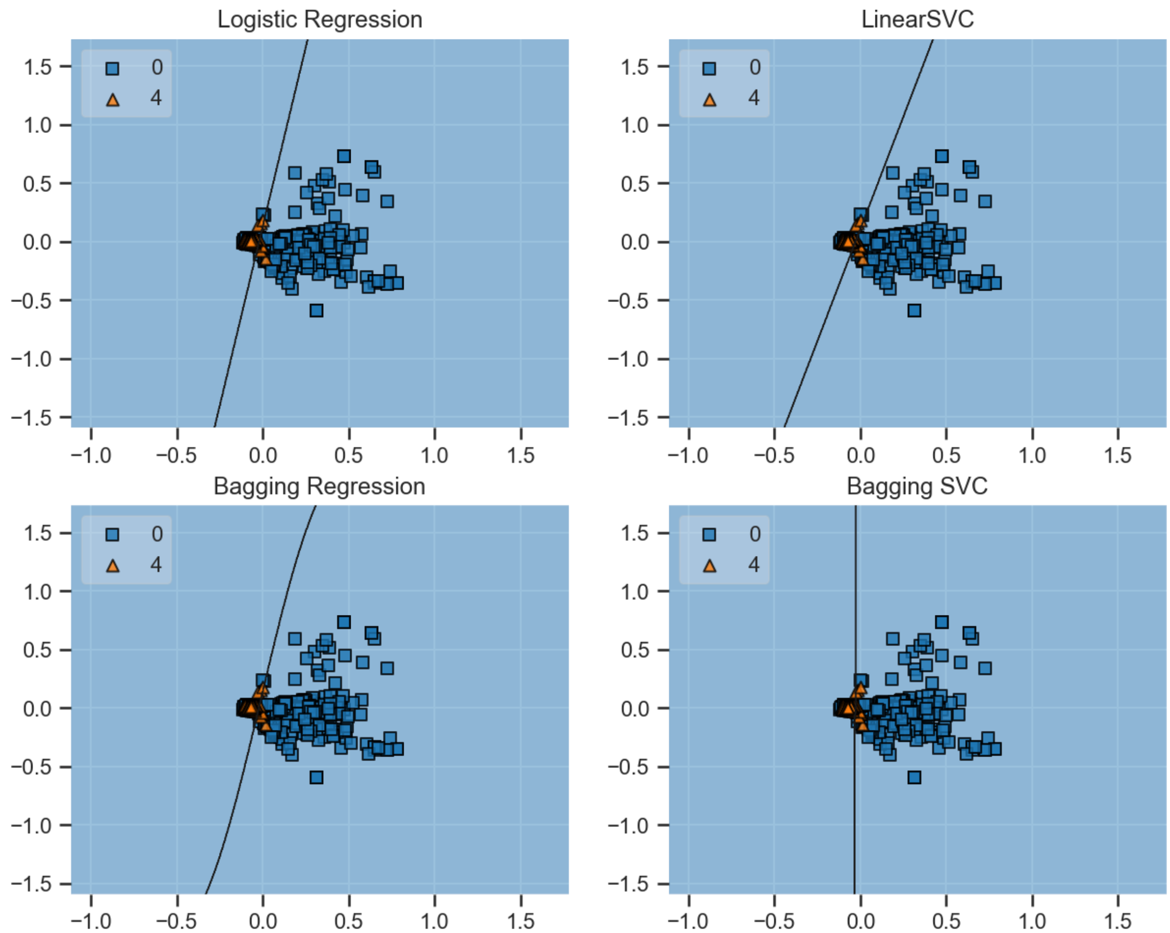
In [240]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
```

In [241]:
```
# X, y = iris.data[:, 0:2], iris.target

X = tv_matrix
y = np.array(df_sm['sentiment'])

clf1 = LogisticRegression(random_state=0)
clf2 = LinearSVC()

bagging1 = BaggingClassifier(base_estimator=clf1, n_estimators=10, max_samples=0
bagging2 = BaggingClassifier(base_estimator=clf2, n_estimators=10, max_samples=0
```

In [244]:

```python
from sklearn.decomposition import PCA

label = ['Logistic Regression', 'LinearSVC', 'Bagging Regression', 'Bagging SVC'
clf_list = [clf1, clf2, bagging1, bagging2]

fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

for clf, label, grd in zip(clf_list, label, grid):
    scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')


    print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.std(), label)

    #clf.fit(X, y)

    pca = PCA(n_components = 2)
    X_flattened = pca.fit_transform(X)
    #clf.fit(X_flattened, y)
    #clf.fit(X, y)
    clf.fit(X_flattened, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X_flattened, y=y, clf=clf, legend=2)
    plt.title(label)


    #plot_decision_regions(X_train2, y_train, clf=clf, legend=2)

plt.show()
```

```
Accuracy: 0.75 (+/- 0.02) [Logistic Regression]
Accuracy: 0.75 (+/- 0.03) [LinearSVC]
Accuracy: 0.74 (+/- 0.03) [Bagging Regression]
Accuracy: 0.74 (+/- 0.02) [Bagging SVC]
```

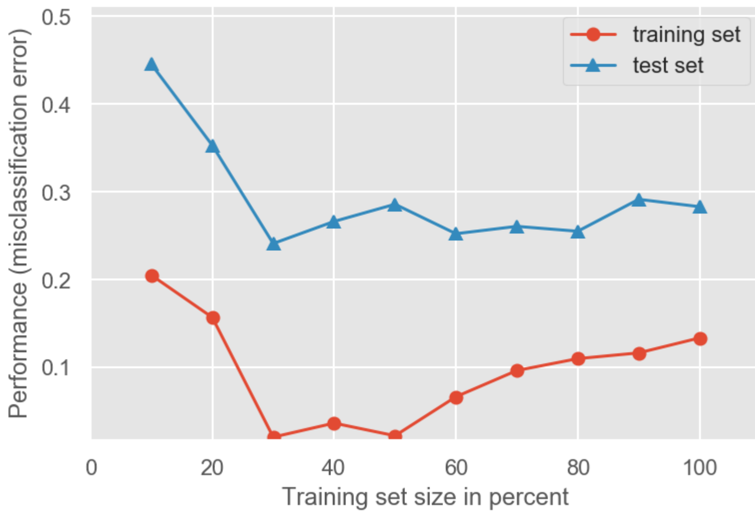


Much better accuracy with these algorithms! Moved us from the low 60s to the mid 70s.

The figure above shows the decision boundary of a decision tree and k-NN classifiers along with their bagging ensembles applied to the Iris dataset. The decision tree shows axes parallel boundaries while the $k = 1$ nearest neighbors fits closely to the data points. The bagging ensembles were trained using 10 base estimators with 0.8 subsampling of training data and 0.8 subsampling of features. The decision tree bagging ensemble achieved higher accuracy in comparison to k-NN bagging ensemble because k-NN are less sensitive to perturbation on training samples and therefore they are called *stable learners*. Combining stable learners is less advantageous since the ensemble will not help improve generalization performance.

In [245]:

```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_s

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, bagging1, print_model=Fals
plt.show()
```

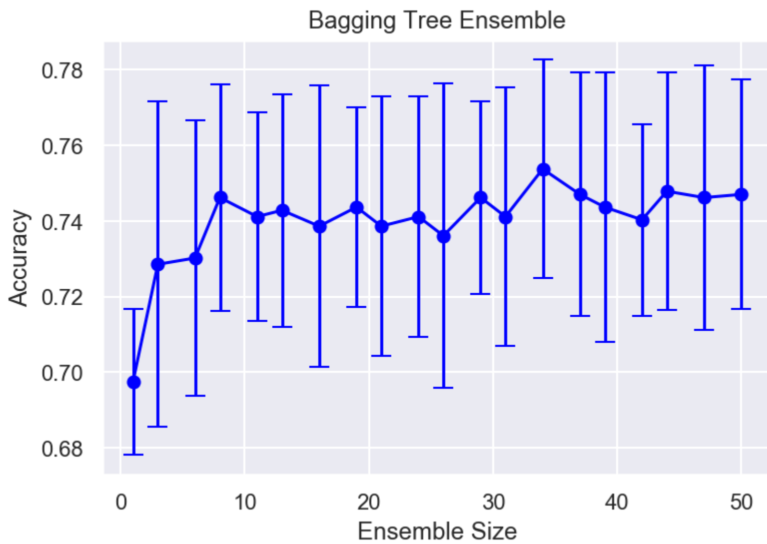


The figure above shows learning curves for the bagging tree ensemble. We can see an average error of 0.35 on the training data and a U-shaped error curve for the testing data. The smallest gap between training and test errors occurs at around 80% of the training set size.

In [264]:

```
#Ensemble Size
# num_est = map(int, np.linspace(1,100,20))
num_est = np.linspace(1,50,20).astype(int)
bg_clf_cv_mean = []
bg_clf_cv_std = []
for n_est in num_est:
    bg_clf = BaggingClassifier(base_estimator=clf1, n_estimators=n_est, max_sampl
    scores = cross_val_score(bg_clf, X, y, cv=3, scoring='accuracy')
    bg_clf_cv_mean.append(scores.mean())
    bg_clf_cv_std.append(scores.std())
```

In [265]:
```
plt.figure()
(_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_std, c='blue
# caps = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_std, c='blue', fmt=
for cap in caps:
    cap.set_markeredgewidth(1)
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('Bagging Tree Ense
plt.show()
```



The figure above shows how the test accuracy improves with the size of the ensemble. Based on cross-validation results, we can see the accuracy increases until approximately 20 base estimators and then plateaus afterwards. Thus, adding base estimators beyond 20 only increases computational complexity without accuracy gains for the Sentiment140 dataset.

Boosting

Boosting refers to a family of algorithms that are able to convert weak learners to strong learners. The main principle of boosting is to fit a sequence of weak learners (models that are only slightly better than random guessing, such as small decision trees) to weighted versions of the data, where more weight is given to examples that were mis-classified by earlier rounds. The predictions are then combined through a weighted majority vote (classification) or a weighted sum (regression) to produce the final prediction. The principal difference between boosting and the committee methods such as bagging is that base learners are trained in sequence on a weighted version of the data.

In [107]:
```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions
```

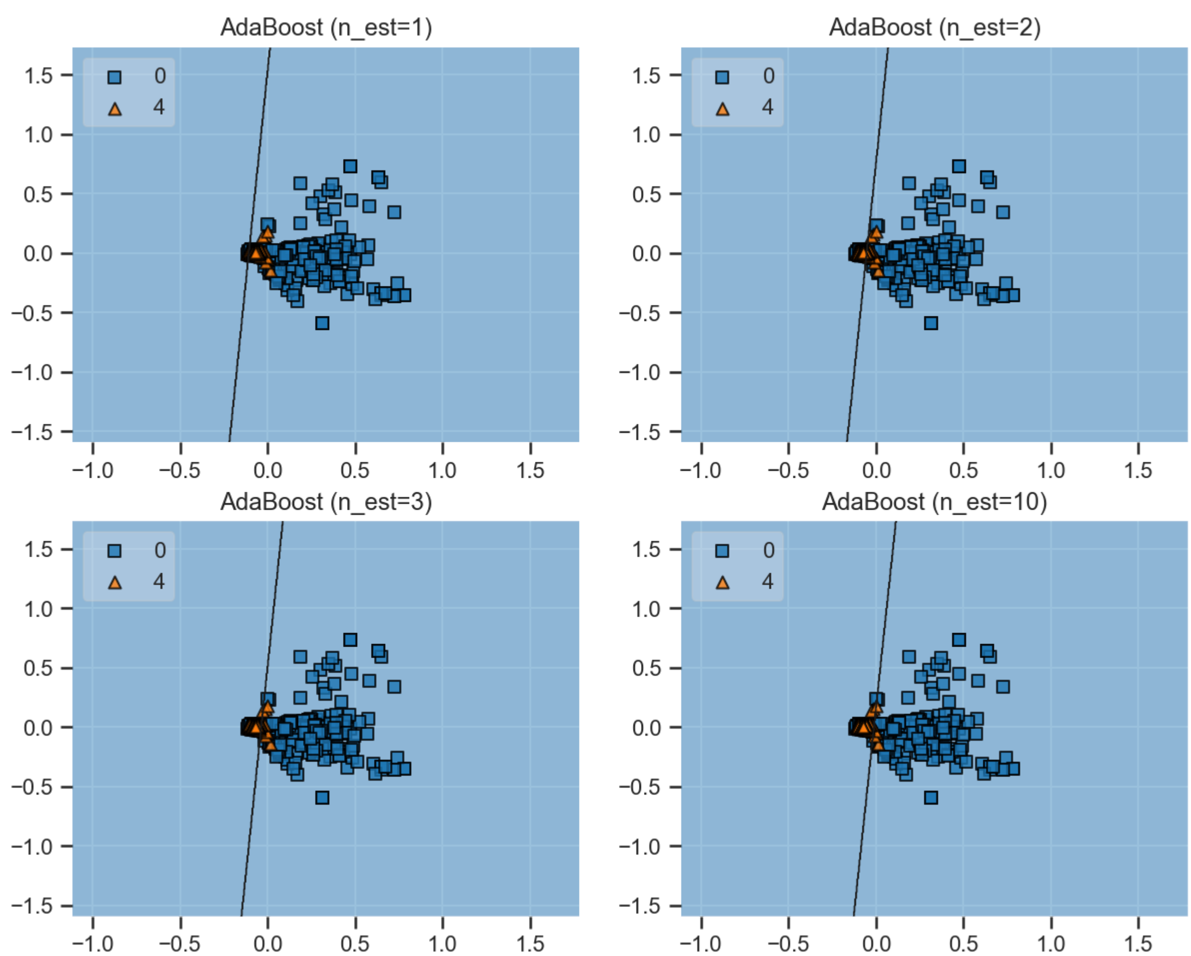
In [266]:
```
X = tv_matrix
y = np.array(df_sm['sentiment'])

clf = LogisticRegression(random_state=0)

num_est = [1, 2, 3, 10]
label = ['AdaBoost (n_est=1)', 'AdaBoost (n_est=2)', 'AdaBoost (n_est=3)', 'AdaBo
```

In [267]:
```
len(X)
```

Out[267]:
```
1190
```

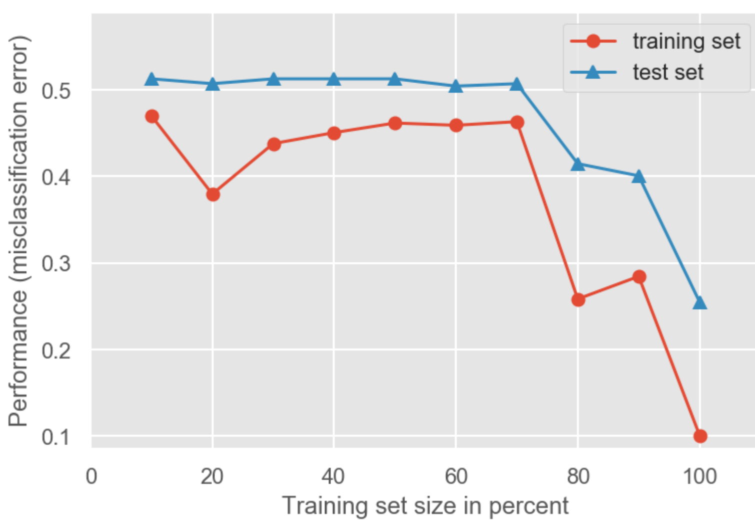
In [268]:
```
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

for n_est, label, grd in zip(num_est, label, grid):
    scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')


    print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.std(), label)
    boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
    pca = PCA(n_components = 2)
    X_flattened = pca.fit_transform(X)
    boosting.fit(X_flattened, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X_flattened, y=y, clf=boosting, legend=2)
    plt.title(label)

plt.show()
```

```
Accuracy: 0.75 (+/- 0.02) [AdaBoost (n_est=1)]
Accuracy: 0.75 (+/- 0.02) [AdaBoost (n_est=2)]
Accuracy: 0.75 (+/- 0.02) [AdaBoost (n_est=3)]
Accuracy: 0.75 (+/- 0.02) [AdaBoost (n_est=10)]
```



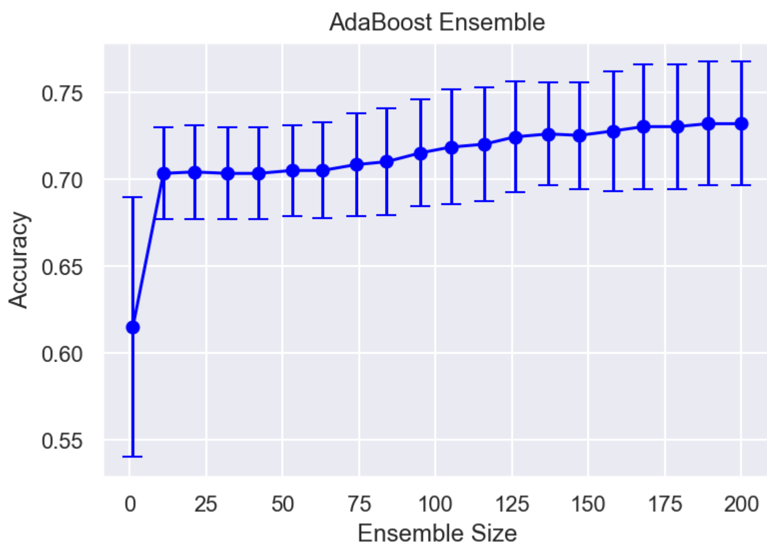
The AdaBoost algorithm is illustrated in the figure above. Each base learner consists of a decision tree with depth 1, thus classifying the data based on a feature threshold that partitions the space into two regions separated by a linear decision surface that is parallel to one of the axes.

In [269]:
```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_s

boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=20)

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, boosting, print_model=Fals
plt.show()
```



In [270]:
```
#Ensemble Size
#num_est = map(int, np.linspace(1,100,20))
num_est = np.linspace(1,200,20).astype(int)
bg_clf_cv_mean = []
bg_clf_cv_std = []
for n_est in num_est:
    ada_clf = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
    scores = cross_val_score(ada_clf, X, y, cv=3, scoring='accuracy')
    bg_clf_cv_mean.append(scores.mean())
    bg_clf_cv_std.append(scores.std())
```

In [271]:
```
plt.figure()
(_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_std, c='blue
for cap in caps:
    cap.set_markeredgewidth(1)
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('AdaBoost Ensembl
plt.show()
```



Stacking

Stacking is an ensemble learning technique that combines multiple classification or regression models via a meta-classifier or a meta-regressor. The base level models are trained based on complete training set then the meta-model is trained on the outputs of base level model as features. The base level often consists of different learning algorithms and therefore stacking ensembles are often heterogeneous.

In [257]:

```
import itertools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier

from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions
```

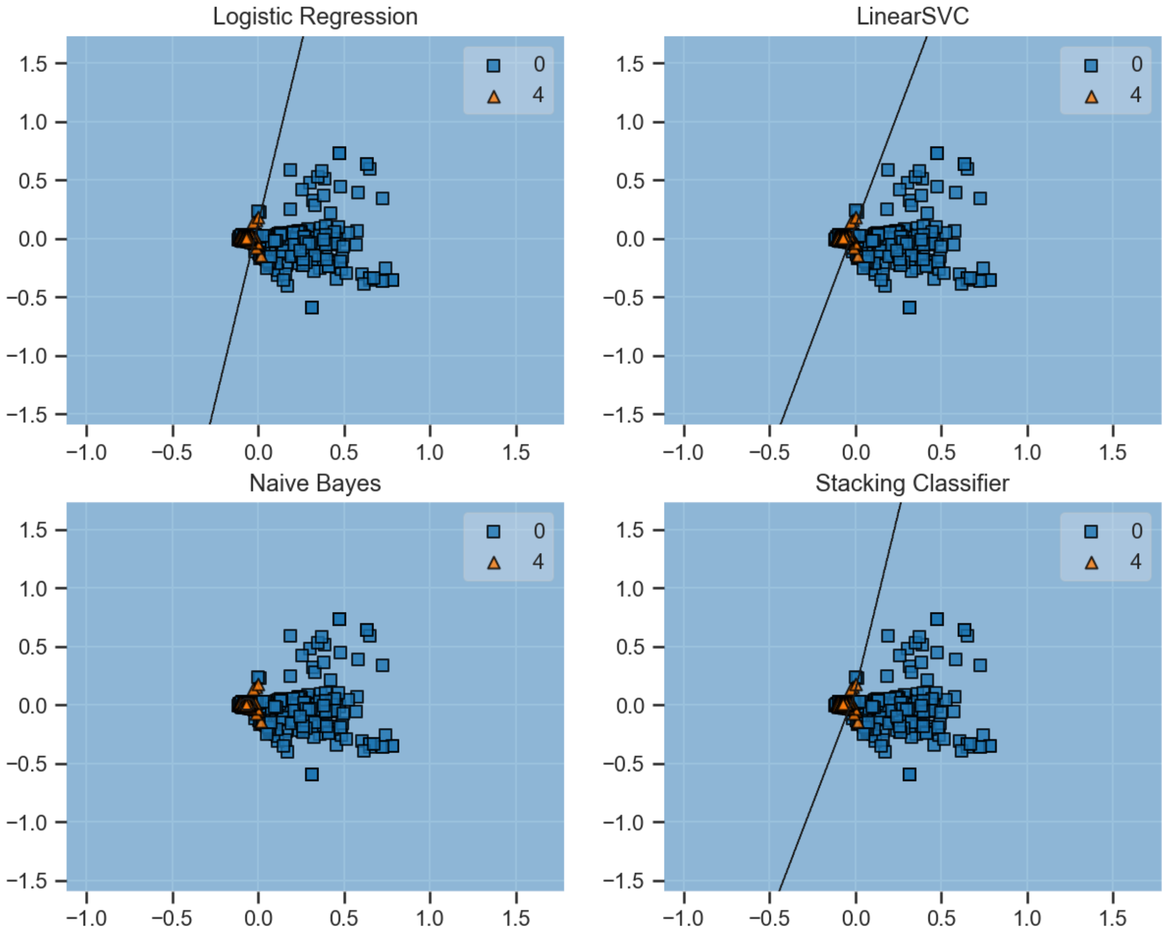
In [260]:

```
X = tv_matrix
y = np.array(df_sm['sentiment'])

clf1 = LogisticRegression(random_state=0)
clf2 = LinearSVC()
#clf1 = KNeighborsClassifier(n_neighbors=1)
#clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                          meta_classifier=lr)
```

In [261]:

```
label = ['Logistic Regression', 'LinearSVC', 'Naive Bayes', 'Stacking Classifier
clf_list = [clf1, clf2, clf3, sclf]

fig = plt.figure(figsize=(10,8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

clf_cv_mean = []
clf_cv_std = []
for clf, label, grd in zip(clf_list, label, grid):

    pca = PCA(n_components = 2)
    X_flattened = pca.fit_transform(X)

    scores = cross_val_score(clf, X_flattened, y, cv=3, scoring='accuracy')
    print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.std(), label)
    clf_cv_mean.append(scores.mean())
    clf_cv_std.append(scores.std())

    clf.fit(X_flattened, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X_flattened, y=y, clf=clf)
    plt.title(label)

plt.show()
```

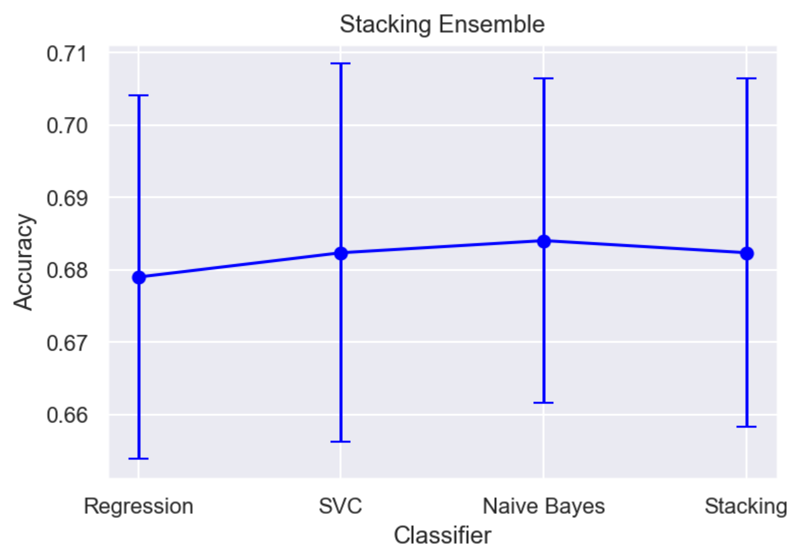
```
Accuracy: 0.68 (+/- 0.03) [Logistic Regression]
Accuracy: 0.68 (+/- 0.03) [LinearSVC]
Accuracy: 0.68 (+/- 0.02) [Naive Bayes]
Accuracy: 0.68 (+/- 0.02) [Stacking Classifier]
```



The stacking ensemble is illustrated in the figure above. It consists of k-NN, Random Forest and Naive Bayes base classifiers whose predictions are combined by Lostic Regression as a meta-classifier. We can see the blending of decision boundaries achieved by the stacking classifier.

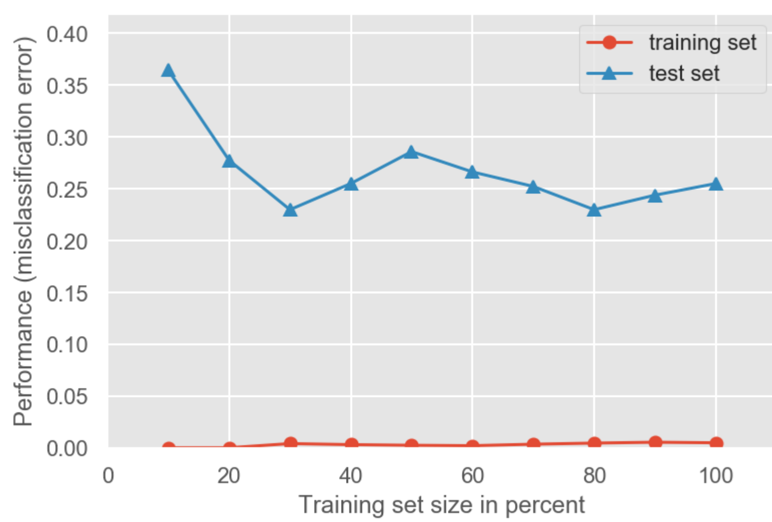
In [262]:

```
#plot classifier accuracy
plt.figure()
(_, caps, _) = plt.errorbar(range(4), clf_cv_mean, yerr=clf_cv_std, c='blue', fm
for cap in caps:
    cap.set_markeredgewidth(1)
plt.xticks(range(4), ['Regression', 'SVC', 'Naive Bayes', 'Stacking'])
plt.ylabel('Accuracy'); plt.xlabel('Classifier'); plt.title('Stacking Ensemble')
plt.show()
```



In [263]:

```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, sclf, print_model=False, 
plt.show()
```




```
End of code for Milestone 2. Please see top of document for more information.
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

###

End of working working file.

###

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Sentiment Analysis with Afinn

As a quick and dirty sanity check, I've set up Afinn in the early stages of data cleaning, and intend to keep a little record of Afinn's performance, as I increase the rigour of the data cleaning.

In [119]:

```
from afinn import Afinn

afn = Afinn(emoticons=True)
```

In [120]:

```
texts = np.array(df_sm['text_nav'])
sentiments = np.array(df_sm['sentiment'])

# extract data for model evaluation
#train_texts = texts[:10000]
#train_sentiments = sentiments[:10000]

#test_texts = texts[40000:60000]
#test_sentiments = sentiments[40000:60000]
sample_ids = [626, 533, 310, 123, 654, 400]
```

In [121]:

```
#for text_clean, sentiment in zip(texts[sample_ids], sentiments[sample_ids]):
#    print('TEXT:', texts)
#    print('Actual Sentiment:', sentiment)
#    print('Predicted Sentiment polarity:', afn.score(texts))
#    print('-'*60)
```