In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

%matplotlib inline
```

In [2]:
```
nlp_emails = pd.read_csv('cleaned_emails.csv')
```

In [3]:
```
nlp_X = np.array(nlp_emails['nlp_X'])
```

In [4]:
```
y = np.array(nlp_emails['spam'])
```

Text Vestorization

In [5]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

tv = TfidfVectorizer(max_features= 2500)
tv_nlp_X = tv.fit_transform(nlp_X)
tv_nlp_X = tv_nlp_X.toarray()
```

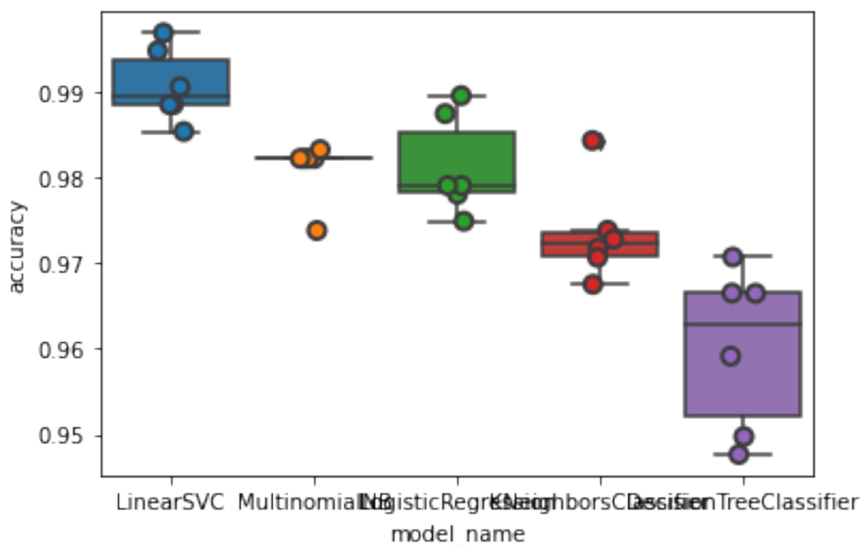
Splitting the dataset into the Training set and Test set

In [6]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(tv_nlp_X, y, test_
```

Generate several models to compare their accuracy and recall rate. RandomForest and LDA are among the lowest so I remove them for saving the time.

In [7]:
```
import warnings
warnings.filterwarnings("ignore")
from sklearn.linear_model import LogisticRegression
#from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
#from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import cross_val_score
```

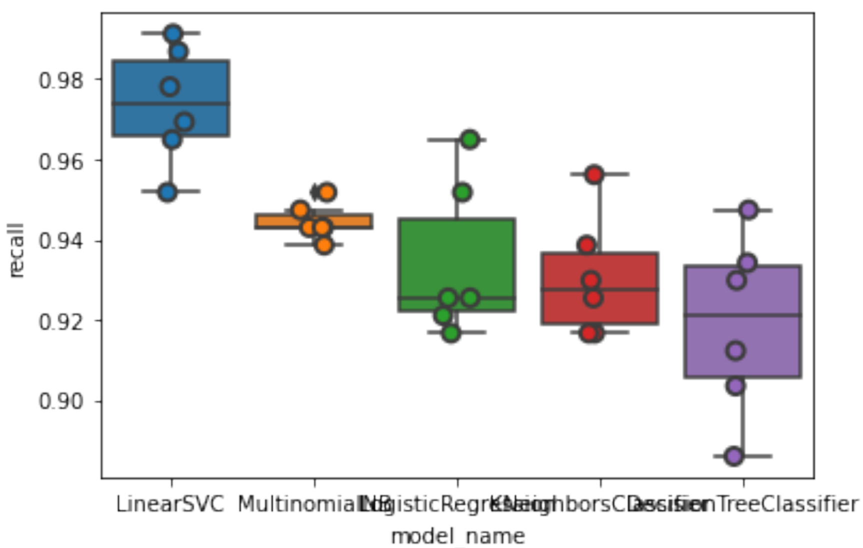
```
models = [
    #RandomForestClassifier(n_estimators=50, max_depth=3, random_state
    LinearSVC(),
    MultinomialNB(),
    LogisticRegression(random_state=0),
    KNeighborsClassifier(),
    #LinearDiscriminantAnalysis(),
    DecisionTreeClassifier(),
]
CV = 6
cv_df = pd.DataFrame(index=range(CV * len(models)))
entries = []
for model in models:
  model_name = model.__class__.__name__
  accuracies = cross_val_score(model, tv_nlp_X, y, scoring='accuracy',
  for fold_idx, accuracy in enumerate(accuracies):
    entries.append((model_name, fold_idx, accuracy))
cv_df = pd.DataFrame(entries, columns=['model_name', 'fold_idx', 'accu
import seaborn as sns
sns.boxplot(x='model_name', y='accuracy', data=cv_df)
sns.stripplot(x='model_name', y='accuracy', data=cv_df,
              size=8, jitter=True, edgecolor="gray", linewidth=2)
plt.show()
```



In [8]:
```
cv_df.groupby('model_name').accuracy.mean()
```

Out[8]:
```
model_name
DecisionTreeClassifier    0.960022
KNeighborsClassifier      0.973462
LinearSVC                 0.990747
LogisticRegression        0.981319
MultinomialNB             0.980970
Name: accuracy, dtype: float64
```

In [9]:
```
models = [
    #RandomForestClassifier(n_estimators=50, max_depth=3, random_state
    LinearSVC(),
    MultinomialNB(),
    LogisticRegression(random_state=0),
    KNeighborsClassifier(),
    #LinearDiscriminantAnalysis(),
    DecisionTreeClassifier(),
]
CV = 6
cv_df = pd.DataFrame(index=range(CV * len(models)))
entries = []
for model in models:
  model_name = model.__class__.__name__
  recall = cross_val_score(model, tv_nlp_X, y, scoring='recall', cv=CV
  for fold_idx, recall in enumerate(recall):
    entries.append((model_name, fold_idx, recall))
cv_df = pd.DataFrame(entries, columns=['model_name', 'fold_idx', 'reca
import seaborn as sns
sns.boxplot(x='model_name', y='recall', data=cv_df)
sns.stripplot(x='model_name', y='recall', data=cv_df,
              size=8, jitter=True, edgecolor="gray", linewidth=2)
plt.show()
```



Recall

Recall, also known as the sensitivity, hit rate, or the true positive rate (TPR), is the proportion of the total amount of relevant instances that were actually retrieved. It answers the question “What proportion of actual positives was identified correctly?"

In [10]:
```
cv_df.groupby('model_name').recall.mean()
```

Out[10]:
```
model_name
DecisionTreeClassifier    0.918860
KNeighborsClassifier      0.930556
LinearSVC                 0.973684
LogisticRegression        0.934211
MultinomialNB             0.944444
Name: recall, dtype: float64
```

Model Evaluation & Selection

Train/test/split sampling

Select the right size of test dataset to avoid Overfitting or Variable Bias(underfitting)

In [11]:
```


from sklearn.model_selection import train_test_split, StratifiedShuffl
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
import seaborn as sns
from scipy import stats
from sklearn.model_selection import StratifiedKFold
```

since LinearSVC reaches the highes scores both in Accuracy and Recall. I will use LinearSVC to do the model evaulation by different test spliting sampling.

In [12]:
```
from sklearn.svm import LinearSVC
LSVCclf = LinearSVC()
LSVCclf.fit(X_train,y_train)
```

Out[12]:
```
LinearSVC()
```

In [13]:
```
y_pred = LSVCclf.predict(X_test)
```

In [14]:
```
from sklearn.metrics import accuracy_score,classification_report,confu

print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1398
           1       0.99      0.96      0.97       493

    accuracy                           0.99      1891
   macro avg       0.99      0.98      0.98      1891
weighted avg       0.99      0.99      0.99      1891
```

In [15]:
```
# Importing the metrics package from sklearn library
from sklearn import metrics
# Creating the confusion matrix
cm = metrics.confusion_matrix(y_test, y_pred)
# Assigning columns names
cm_df = pd.DataFrame(cm,
            columns = ['Predicted Negative', 'Predicted Positive'],
            index = ['Actual Negative', 'Actual Positive'])
# Showing the confusion matrix
cm_df
```

Out[15]:

	Predicted Negative	Predicted Positive
Actual Negative	1391	7
Actual Positive	19	474

In [16]:
```
print('True:', y_test[0:25])
print('Pred:', y_pred[0:25])
```

```
True: [0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 0 0 0 0]
Pred: [0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0]
```

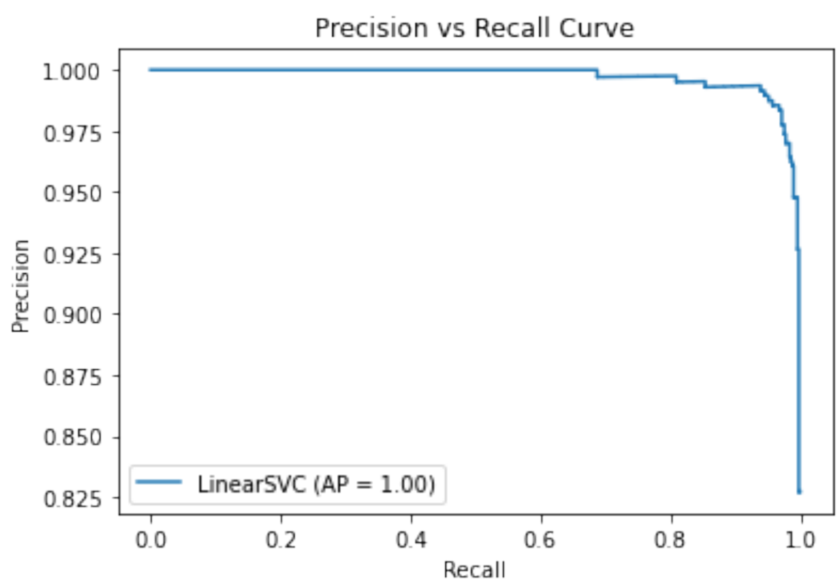
Which metrics should you focus on?

Choice of metric depends on your business objective

- Spam filter* (positive class is "spam"): Optimize for precision or specificity because false negatives (spam goes to the inbox) are more acceptable than false positives (non-spam is caught by the spam filter)
- Fraudulent transaction* detector (positive class is "fraud"): Optimize for sensitivity because false positives (normal transactions that are flagged as possible fraud) are more acceptable than false negatives (fraudulent transactions that are not detected)

In [17]:
```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize

disp = plot_precision_recall_curve(LSVCclf, X_test, y_test)
disp.ax_.set_title('Precision vs Recall Curve')
```

Out[17]:
```
Text(0.5, 1.0, 'Precision vs Recall Curve')
```



Sensitivity: When the actual value is positive, how often is the prediction correct? How "sensitive" is the classifier to detecting positive instances? Also known as "True Positive Rate" or "Recall"

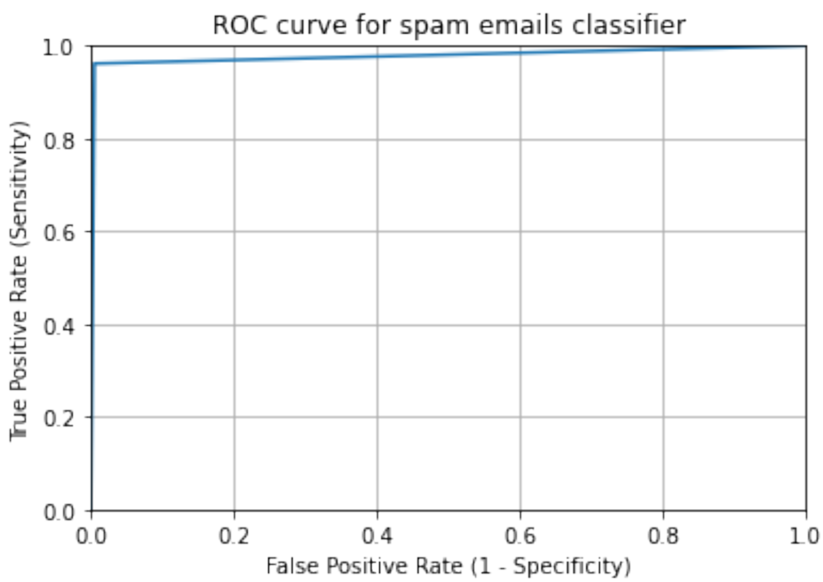
Specificity: When the actual value is negative, how often is the prediction correct? How "specific" (or "selective") is the classifier in predicting positive instances?

ROC Curves and Area Under the Curve (AUC)¶

Question: Wouldn't it be nice if we could see how sensitivity and specificity are affected by various thresholds, without actually changing the threshold?

Answer: Plot the ROC curve

In [19]:
```
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)
plt.plot(fpr, tpr)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.title('ROC curve for spam emails classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.grid(True)
```



In [20]:
```
# define a function that accepts a threshold and prints sensitivity an
def evaluate_threshold(threshold):
    print('Sensitivity:', tpr[thresholds > threshold][-1])
    print('Specificity:', 1 - fpr[thresholds > threshold][-1])
evaluate_threshold(0.5)
```

```
Sensitivity: 0.9614604462474645
Specificity: 0.9949928469241774
```

Caculate the AUC

AUC is useful as a single number summary of classifier performance. If you randomly chose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a higher predicted probability to the positive observation. AUC is useful even when there is high class imbalance (unlike classification accuracy).

In [21]:
```
print(metrics.roc_auc_score(y_test, y_pred))
```

```
0.978226646585821
```

Stratified sampling

Split train and test set will result a better testing accuracy, but it provides a high variance estimate since changing which observation happen to be in the testing set can signifivantly change testing accuracy

Stratified sampling While splitting, we need to ensure that the distribution of features as well as target remains the same in the training and test sets. For ex: Consider a problem where we're trying to classify an observation as fraudulent or not. While splitting, if the majority of fraud cases went to the test set, the model won't be able to learn the fraudulent patterns, as it doesn't have access to many fraud cases in the training data. In such cases, stratified sampling should be done, as it maintains the proportion of different classes in the train and test set. In this project: spam emails are the minority cases so that using Stratified sampling will be better

Avoid High Bias or High Variance

A good choice of hyperparameters ensures that parameters learnt are corresponding to a good loss minima (more generalizable model). Generalizable models are less prone to overfitting. They have consistently good performance on train and test data.

High bias implies our estimate based on the observed data is not close to the true parameter. (aka underfitting). **High variance** implies our estimates are sensitive to sampling. They'll vary a lot if we compute them with a different sample of data (aka overfitting).

Validation strategies can be broadly divided into 2 categories: **Holdout validation** and **cross validation**.

a)Single holdout: Varying test Size by testing the recall scores

Implementation

The basic idea is to split our data into a training set and a holdout test set. Train the model on the training set and then evaluate model performance on the test set. We take only a single holdout—hence the name

> step1: split target data into 2 subesets

> step2: Choose LinearSVC as the learnining algorithm

> step3: Predict on the test data using the trained model. Choose an appropriate metric for performance estimation(I choose recall to the classification task). Assess predictive performance by comparing predictions and ground truth.

> Step 4: If the performance estimate computed in the previous step is satisfactory, combine the train and test subset to train the model on the full data with the same hyperparameters.

Stratified sampling

While splitting, we need to ensure that the distribution of features as well as target remains the same in the training and test sets. For ex: Consider a problem where we're trying to classify an observation as fraudulent or not. While splitting, if the majority of fraud cases went to the test set, the model won't be able to learn the fraudulent patterns, as it doesn't have access to many fraud cases in the training data. In such cases, stratified sampling should be done, as it maintains the proportion of different classes in the train and test set.

In this project: spam emails are the minority cases so that using Stratified sampling will be better

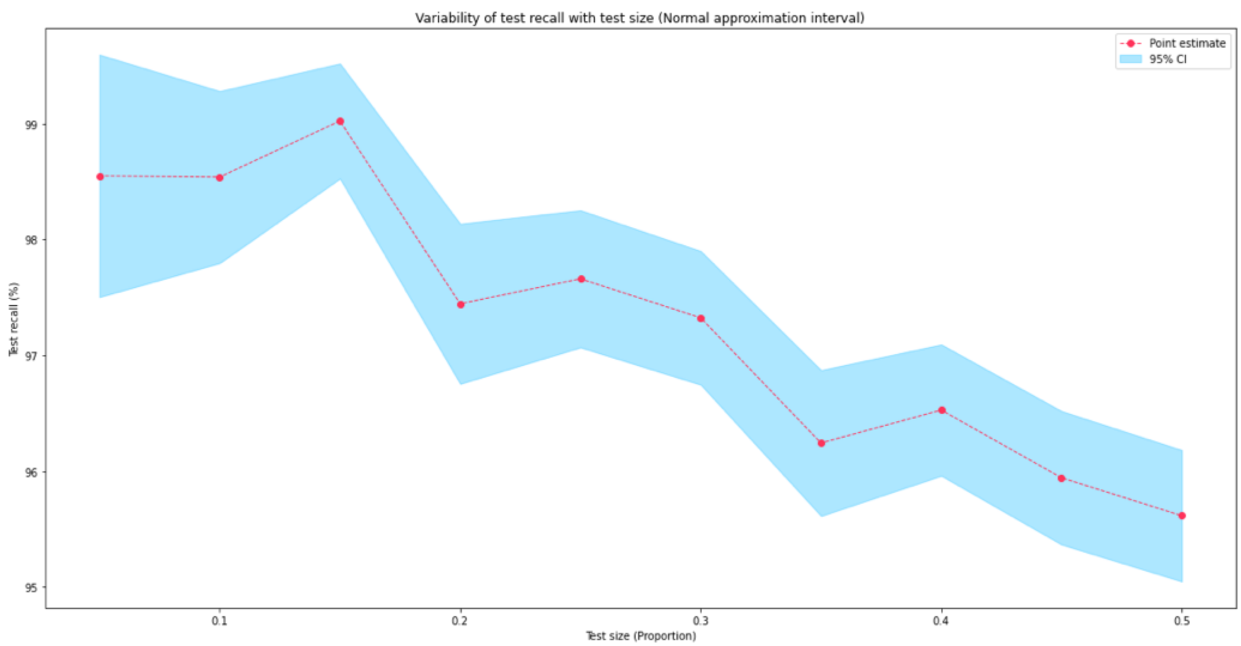
In [22]:
```
x= tv_nlp_X
Y=y
```

In [23]:
```
# varying hold out size
test_size = np.arange(0.05, 0.55, 0.05)

trn_recall = []
tst_recall = []

for sz in test_size:
    #stratified sampling
    sss = StratifiedShuffleSplit(n_splits=1, test_size=sz, random_stat

    #train-test split
    for trn_idx, tst_idx in sss.split(x, y):
        x_trn, y_trn, x_tst, y_tst = x[trn_idx], y[trn_idx], x[tst_idx
            tst_idx]

    #model fitting
    clf = LinearSVC()
    clf.fit(x_trn, y_trn)

    #model prediction
    pred_tst = clf.predict(x_tst)
    pred_trn = clf.predict(x_trn)

    #perfromance evaluation
    tst_recall.append(recall_score(y_tst, pred_tst))
    trn_recall.append(recall_score(y_trn, pred_trn))
```

In [24]:
```
# 95% CI calculation using normal approximation method
ui = []
li = []
for i, n in enumerate(test_size):
    p = tst_recall[i]
    sigma = np.sqrt(p * (1 - p) / (n * 10000))
    ui.append(p + 1.96 * sigma)
    li.append(p - 1.96 * sigma)
```

In [25]:
```
#CI plot
def lineplotCI(x_data, y_data, low_CI, upper_CI, x_label, y_label, tit
    # Create the plot object
    _, ax = plt.subplots(figsize=(20, 10))
    # Plot the data, set the linewidth, color and transparency of the
    # line, provide a label for the legend
    ax.plot(
        x_data,
        y_data,
        lw=1,
        color='#FF335B',
        alpha=1,
```

```
            label='Point estimate',
            linestyle='--',
            marker='o')
    # Shade the confidence interval
    ax.fill_between(
        x_data, low_CI, upper_CI, color='#33C4FF', alpha=0.4, label='9
    # Label the axes and provide a title
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    # Display legend
    ax.legend(loc='best')


# Call the function to create plot
lineplotCI(
    x_data=test_size,
    y_data=100 * np.array(tst_recall),
    low_CI=100 * np.array(li),
    upper_CI=100 * np.array(ui),
    x_label='Test size (Proportion)',
    y_label='Test recall (%)',
    title=
    'Variability of test recall with test size (Normal approximation i
```



Choice of test size

Keeping aside a large amount of data for the test can result in an underestimation of predictive power (high bias**). But the estimate will be more stable (low variance**), as shown in the figure above. This consideration is more relevant for smaller datasets. When test size are in the range of 15%, it reaches the highes recall rate.

Repeated HoldOut for the recall scores

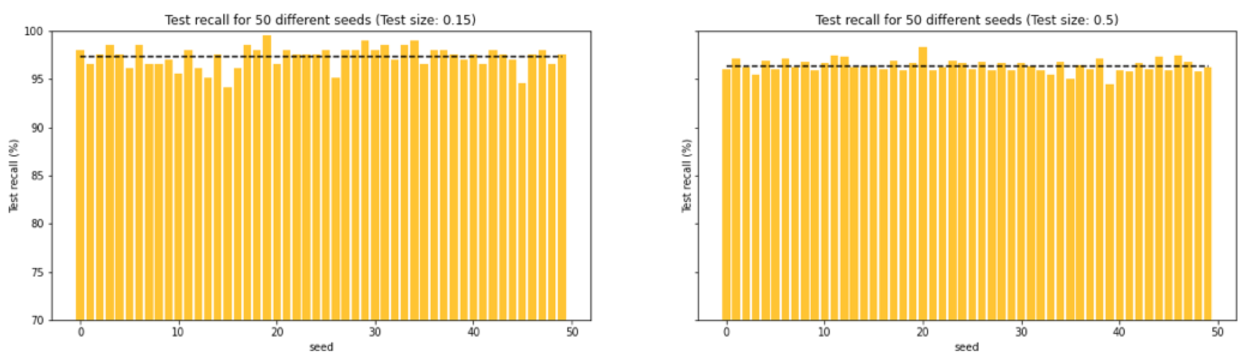
In [26]:
```
test_size = 0.15
# repeating for 50 different seeds
seed = np.random.randint(0, 1000, 50)
trn_recall_1 = []
tst_recall_1 = []
for state in seed:
    sss = StratifiedShuffleSplit(
        n_splits=1, test_size=test_size, random_state=state)

    for trn_idx, tst_idx in sss.split(x, y):
        x_trn, y_trn, x_tst, y_tst = x[trn_idx], y[trn_idx], x[tst_idx
            tst_idx]

    clf = LinearSVC()

    clf.fit(x_trn, y_trn)

    pred_tst = clf.predict(x_tst)
    pred_trn = clf.predict(x_trn)

    tst_recall_1.append(recall_score(y_tst, pred_tst))
    trn_recall_1.append(recall_score(y_trn, pred_trn))
```

In [27]:
```
test_size = 0.5
seed = np.random.randint(0, 1000, 50)
trn_recall_2 = []
tst_recall_2 = []
for state in seed:
    sss = StratifiedShuffleSplit(
        n_splits=1, test_size=test_size, random_state=state)

    for trn_idx, tst_idx in sss.split(x, y):
        x_trn, y_trn, x_tst, y_tst = x[trn_idx], y[trn_idx], x[tst_idx
            tst_idx]

    clf = LinearSVC()

    clf.fit(x_trn, y_trn)

    pred_tst = clf.predict(x_tst)
    pred_trn = clf.predict(x_trn)

    tst_recall_2.append(recall_score(y_tst, pred_tst))
    trn_recall_2.append(recall_score(y_trn, pred_trn))
```

In [28]:
```
df = pd.DataFrame()

df['0.15'] = tst_recall_1

df['0.5'] = tst_recall_2

df['seed'] = df.index
```

In [29]:
```
# plotting accuracy score for 50 different iterations
_, axes = plt.subplots(1, 2, figsize=(20, 5), sharey=True)
axes[0].bar(df.seed, 100 * df['0.15'], color='#FFC433')
axes[0].plot(df.seed, [100 * np.mean(df['0.15'])] * df.shape[0], "k--"
axes[0].set_title("Test recall for 50 different seeds (Test size: 0.15
axes[1].bar(df.seed, 100 * df['0.5'], color='#FFC433')
axes[1].plot(df.seed, [100 * np.mean(df['0.5'])] * df.shape[0], "k--")
axes[1].set_title("Test recall for 50 different seeds (Test size: 0.5)
plt.ylim([70, 100])
axes[0].set_xlabel("seed")
axes[0].set_ylabel("Test recall (%)")
axes[1].set_xlabel("seed")
axes[1].set_ylabel("Test recall (%)")
```

Out[29]:
```
Text(0, 0.5, 'Test recall (%)')
```



In [30]:
```
print(f"Mean recall for test size (0.15): {100*np.mean(df['0.15'])}")
```

```
Mean recall for test size (0.15): 97.34634146341467
```

In [31]:
```
print(f"Mean recall for test size (0.5): {100*np.mean(df['0.5'])}")
```

```
Mean recall for test size (0.5): 96.38304093567245
```

How confident are we in our estimates? From the above steps, we'll get a point estimate of the true predictive power of our model. But this single number doesn't mean anything unless we know how confident we are in this estimate. Defining the confidence interval around this point estimate would tell us how much this estimate can vary for a different set of model inputs. Let's discuss a way of estimating this interval.

Normal approximation interval

Suppose we're choosing accuracy as the proxy for predictive power of the model. Let's look at the calculation for the confidence interval (CI) in this case: Normal approximation interval

$$SE_{repeated} = \sqrt{(\sum_{i=1}^{k}(ACC_i - ACC_{avg})^2)/(k-1)}$$
$$CI = ACC_{avg} \pm t * SE_{repeated}$$

n the above formula, SE is the standard error and t is the value coming from t-distribution with degree of freedom as k-1. We're using t-distribution because we're calculating SE from the sample.

Type *Markdown* and LaTeX: α^2

In [32]:
```
stats.t.ppf([.025, .975], 49)  #t-value for 95% confidence
```

Out[32]:
```
array([-2.00957523,  2.00957523])
```

In [33]:
```
ui, li = np.mean(
    df['0.15']) + np.sqrt(np.var(df['0.15']) * 50 / 49) * 2.00957523, 
        df['0.15']) - np.sqrt(np.var(df['0.15']) * 50 / 49) * 2.009575

li, ui
```

Out[33]:
```
(0.9506098337251948, 0.9963169955430987)
```

In [34]:
```
ui, li = np.mean(
    df['0.5']) + np.sqrt(np.var(df['0.5']) * 50 / 49) * 2.00957523, np
        df['0.5']) - np.sqrt(np.var(df['0.5']) * 50 / 49) * 2.00957523

li, ui
```

Out[34]:
```
(0.9501809655571888, 0.9774798531562604)
```

Empirical interval

Empirical interval is suggested when our samples don't follow a normal distribution and the value of k is high

In [35]:
```
np.percentile(df['0.15'],97.5)
```

Out[35]: 0.9902439024390244

In [36]:
```
np.percentile(df['0.15'],2.5)
```

Out[36]: 0.9474390243902439

In [37]:
```
np.percentile(df['0.5'],97.5)
```

Out[37]: 0.9751461988304093

In [38]:
```
np.percentile(df['0.5'],2.5)
```

Out[38]: 0.9512792397660819

K-fold cross validation

The most common approach for model evaluation is cross validation. Let's quickly go through the steps:

- Choose a value of k and divide the data into k equal subsets
- Combine k-1 subsets and consider it as a training fold and the remaining one as a test fold
- Conduct the holdout method to get test performance (let's choose recall for now)
- Repeat 2nd and 3rd steps, k times with a different subset as test fold
- Point estimate of predictive power is the average of the k different test accuracies

Choice of k

Small k: High bias (less data for training in each fold) but low variance (more data in test)
High k: Low bias but high variance Below is the comparison of variance and bias for 5-fold CV, 10-fold CV and 10-fold CV repeated multiple times from this project. It get a little bit higher recall in the fold of 10, but fold of 5 get less erros of standard deviation.

In [40]:
```
from sklearn.model_selection import cross_val_score
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
folds_score = []
for trn_idx, tst_idx in skf.split(x, y):
    x_trn, y_trn, x_tst, y_tst = x[trn_idx], y[trn_idx], x[tst_idx], y
    clf = LinearSVC()

    clf.fit(x_trn, y_trn)

    pred_tst = clf.predict(x_tst)

    folds_score.append(recall_score(y_tst, pred_tst))

mean_recall = np.mean(folds_score)

std_recall = np.std(folds_score)

mean_recall,std_recall
```

Out[40]: (0.9751530707735088, 0.004814031824714703)

In [41]:
```
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=0)
folds_score = []
for trn_idx, tst_idx in skf.split(x, y):
    x_trn, y_trn, x_tst, y_tst = x[trn_idx], y[trn_idx], x[tst_idx], y
    clf = LinearSVC()

    clf.fit(x_trn, y_trn)

    pred_tst = clf.predict(x_tst)

    folds_score.append(recall_score(y_tst, pred_tst))

mean_recall = np.mean(folds_score)

std_recall = np.std(folds_score)

mean_recall,std_recall
```

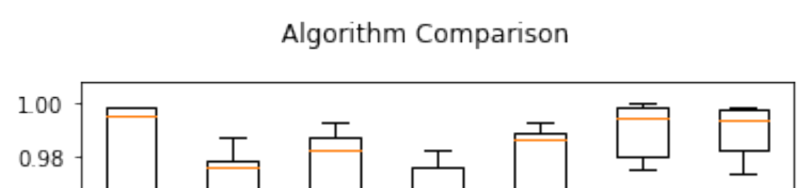
Out[41]: (0.9766208673250322, 0.011662798708020498)

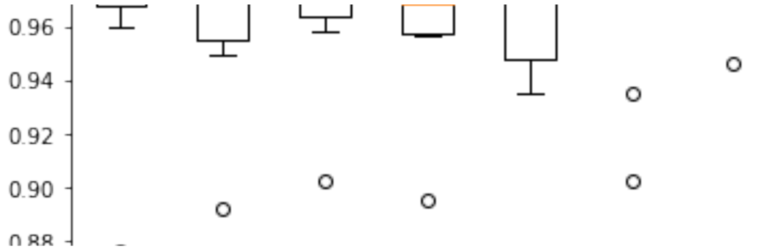
K_fold Cross Validation for multiple algorithms

In [42]:
```

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.svm import LinearSVC

seed = 7
# prepare models
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
models.append(('LSVC',LinearSVC()))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state=seed)
    cv_results = model_selection.cross_val_score(model, tv_nlp_X, y, c
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
# boxplot algorithm comparison
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

```
LR: 0.966142 (0.053062)
LDA: 0.955838 (0.039655)
KNN: 0.963346 (0.039060)
CART: 0.952347 (0.039431)
NB: 0.954447 (0.056702)
SVM: 0.979231 (0.031774)
LSVC: 0.986909 (0.015730)
```





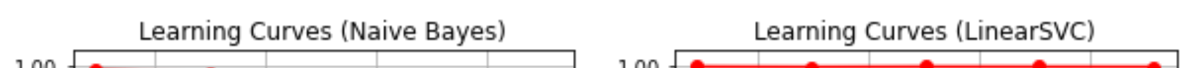
Evaluate the learning curves

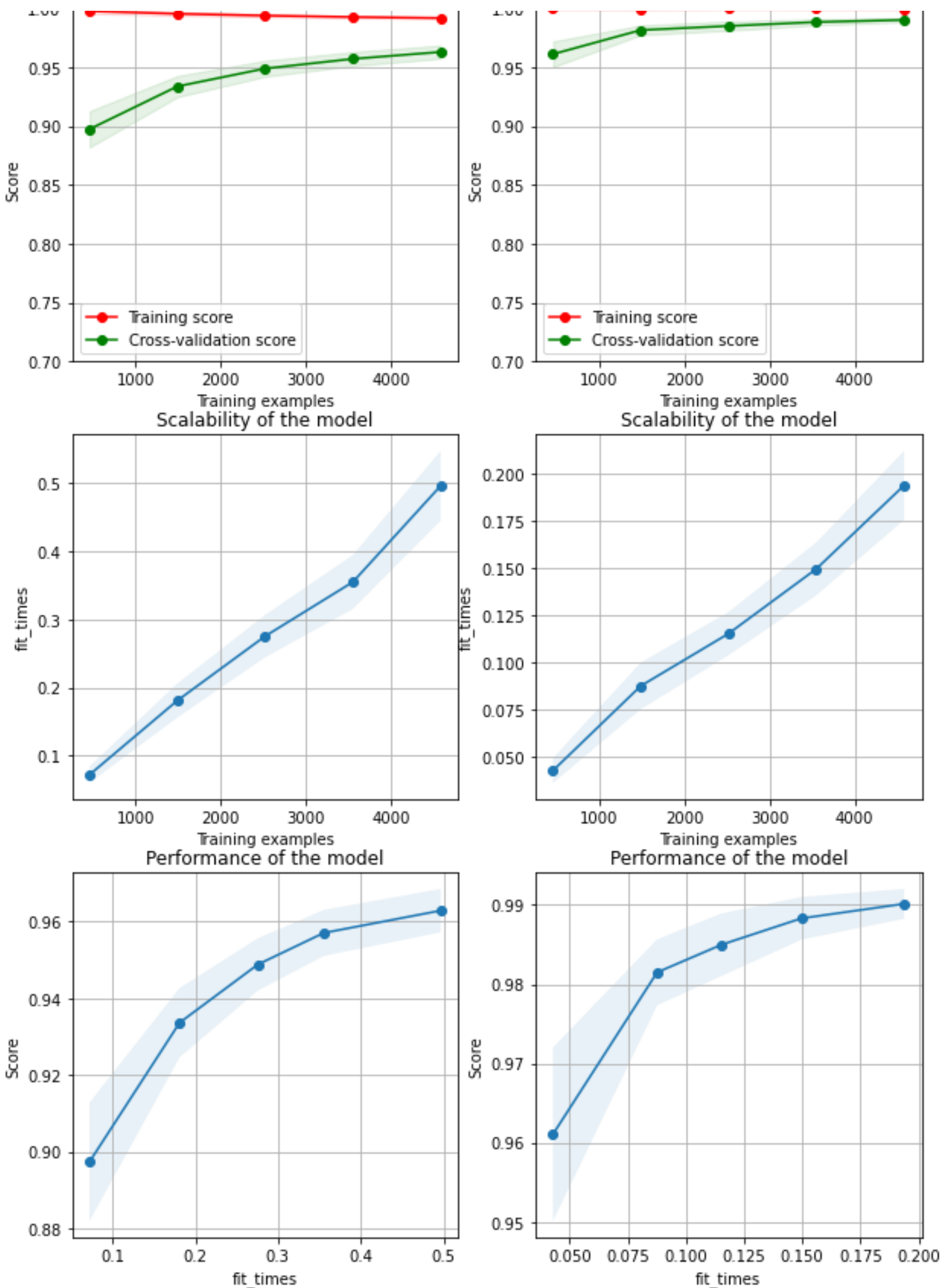
Learning curves constitute a great tool to diagnose bias and variance in any supervised learning algorithm. For classification training learning score is base on the accuracy, the higher the better.

In [43]:

```
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit


def plot_learning_curve(estimator, title, x, y, axes=None, ylim=None,
                        n_jobs=None, train_sizes=np.linspace(.1, 1.0,

    if axes is None:
        _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title(title)
    if ylim is not None:
        axes[0].set_ylim(*ylim)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, x, Y, cv=cv, n_jobs=n_jobs,
                       train_sizes=train_sizes,
                       return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

    # Plot learning curve
    axes[0].grid()
    axes[0].fill_between(train_sizes, train_scores_mean - train_scores
                         train_scores_mean + train_scores_std, alpha=0
                         color="r")
    axes[0].fill_between(train_sizes, test_scores_mean - test_scores_s
                         test_scores_mean + test_scores_std, alpha=0.1
```

```
                         color="g")
    axes[0].plot(train_sizes, train_scores_mean, 'o-', color="r",
                 label="Training score")
    axes[0].plot(train_sizes, test_scores_mean, 'o-', color="g",
                 label="Cross-validation score")
    axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, 'o-')
    axes[1].fill_between(train_sizes, fit_times_mean - fit_times_std,
                         fit_times_mean + fit_times_std, alpha=0.1)
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("fit_times")
    axes[1].set_title("Scalability of the model")

    # Plot fit_time vs score
    axes[2].grid()
    axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
    axes[2].fill_between(fit_times_mean, test_scores_mean - test_score
                         test_scores_mean + test_scores_std, alpha=0.1
    axes[2].set_xlabel("fit_times")
    axes[2].set_ylabel("Score")
    axes[2].set_title("Performance of the model")

    return plt
fig, axes = plt.subplots(3, 2, figsize=(10, 15))



title = "Learning Curves (Naive Bayes)"
# Cross validation with 100 iterations to get smoother mean test and t
# score curves, each time with 20% data randomly selected as a validat
cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=0)

estimator = GaussianNB()
plot_learning_curve(estimator, title, x, Y, axes=axes[:, 0], ylim=(0.7
                    cv=cv, n_jobs=4)

title = r"Learning Curves (LinearSVC)"

cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator = LinearSVC()
plot_learning_curve(estimator, title, x, Y, axes=axes[:, 1], ylim=(0.7
                    cv=cv, n_jobs=4)

plt.show()
```




Training score
Cross-validation score
Score
Training examples
Training score
Cross-validation score
Score
Training examples
Scalability of the model
fit_times
Training examples
Scalability of the model
fit_times
Training examples
Performance of the model
Score
fit_times
Performance of the model
Score
fit_times

Ensemble Methods

Bagging

In [44]:
```
import itertools

import seaborn as sns
import matplotlib.gridspec as gridspec

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import MultinomialNB

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 [2]:
```
#from sklearn.feature_extraction.text import TfidfVectorizer

#tv = TfidfVectorizer(max_features= 2500)
#tv_nlp_X = tv.fit_transform(nlp_X)
#tv_nlp_X = tv_nlp_X.toarray()

#vocab = tv.get_feature_names()
#pd.DataFrame(np.round(tv_nlp_X, 2), columns=vocab)
#len(vocab)
```

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
trans = MinMaxScaler()
data = trans.fit_transform()
```

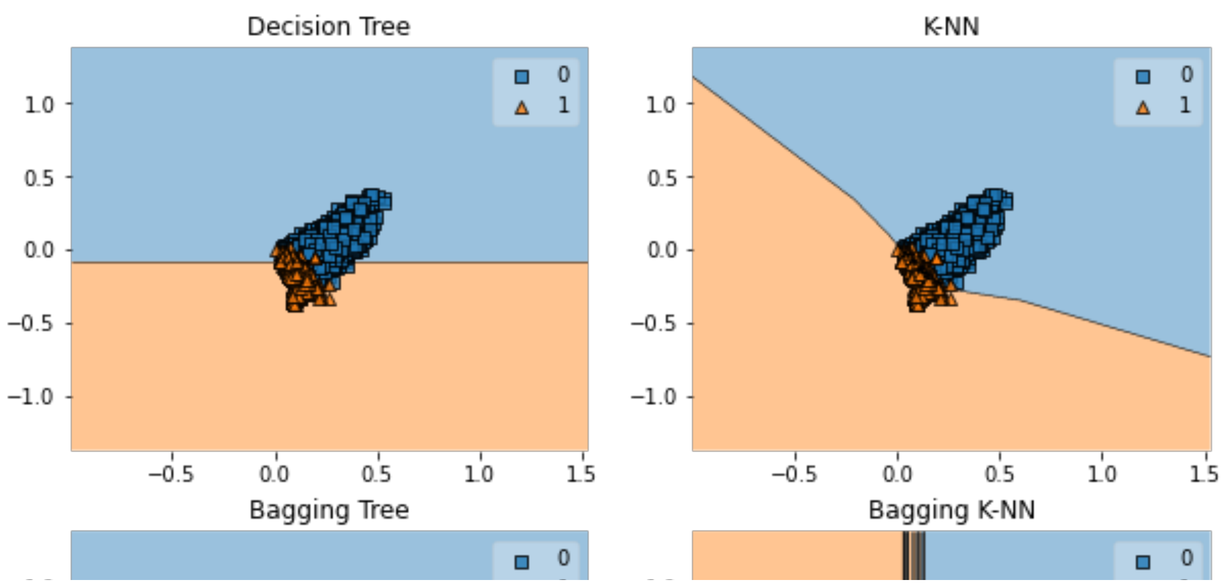
In [154]:
```
from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=2)
svd.fit(tv_nlp_X)
X_new=svd.fit_transform(tv_nlp_X)
```

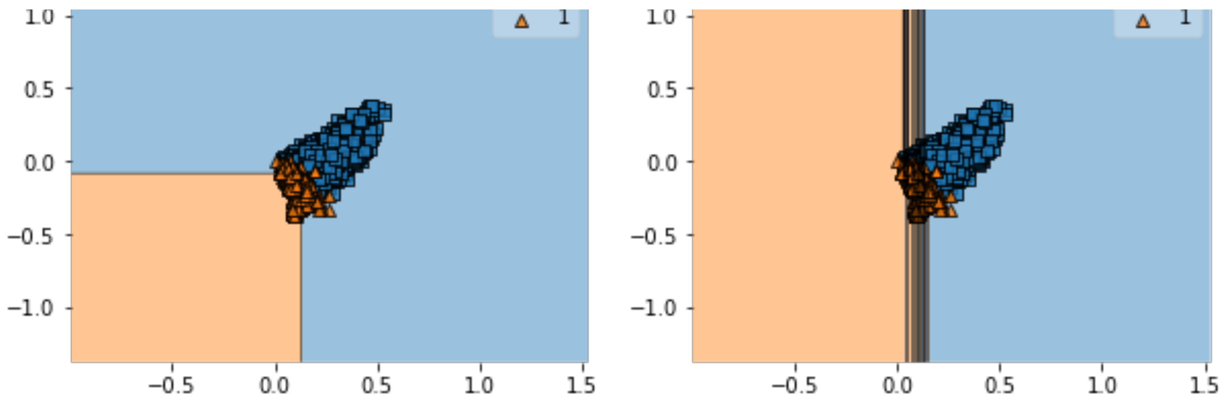
In [155]:
```
X_new
```

Out[155]:
```
array([[ 0.12491249, -0.32158119],
       [ 0.08279099, -0.02431283],
       [ 0.11412287, -0.1527904 ],
       ...,
       [ 0.2674734 ,  0.03995376],
       [ 0.27994105,  0.14885369],
       [ 0.16095834, -0.17607605]])
```

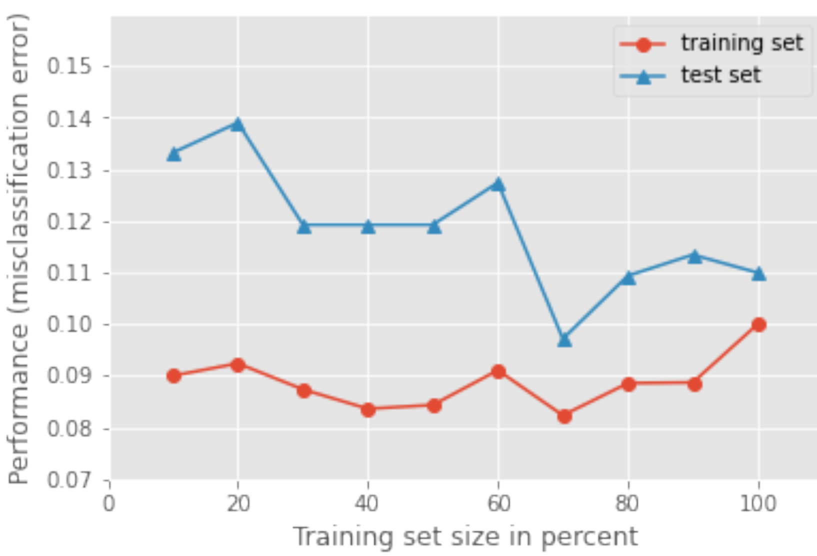
In [159]:
```
label = ['Decision Tree', 'K-NN', 'Bagging Tree', 'Bagging K-NN']
clf_list = [clf1, clf2, bagging1, bagging2]

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_new, y, cv=3, scoring='accuracy')
    print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.st

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

plt.show()
```

```
Accuracy: 0.85 (+/- 0.01) [Decision Tree]
Accuracy: 0.94 (+/- 0.00) [K-NN]
Accuracy: 0.90 (+/- 0.01) [Bagging Tree]
Accuracy: 0.86 (+/- 0.02) [Bagging K-NN]
```



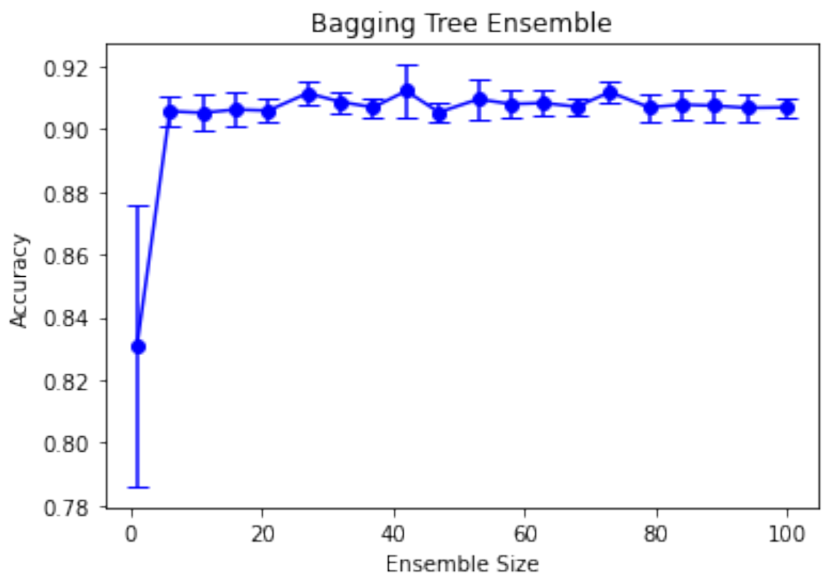


In [160]:
```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_siz

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, bagging1, print
plt.show()
```



In [161]:
```
#Ensemble Size
num_est = np.linspace(1,100,20).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
    scores = cross_val_score(bg_clf, X_new, y, cv=3, scoring='accuracy
    bg_clf_cv_mean.append(scores.mean())
    bg_clf_cv_std.append(scores.std())
```

In [162]:
```
plt.figure()
(_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_st
for cap in caps:
    cap.set_markeredgewidth(1)
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('Baggir
plt.show()
```

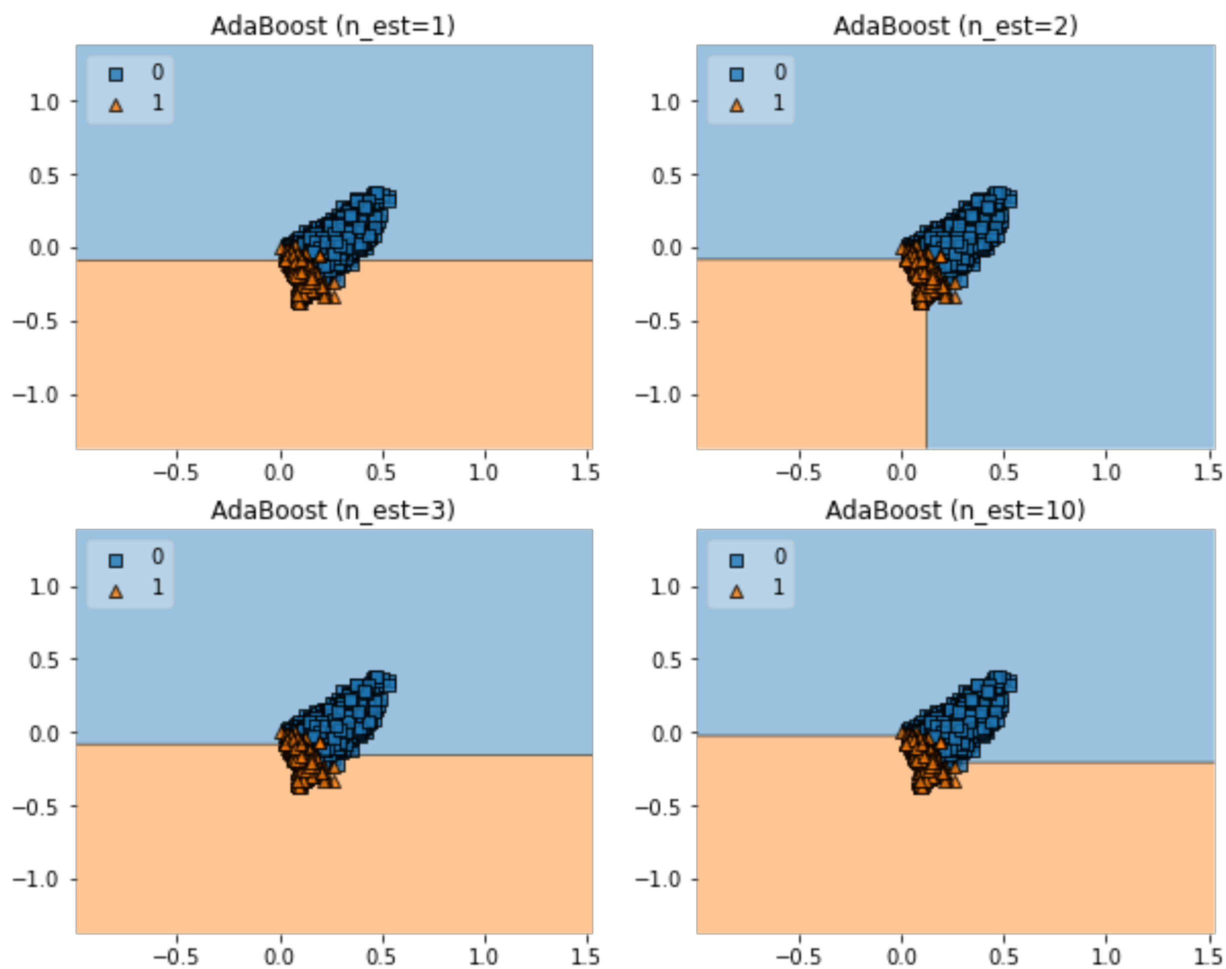


Boosting

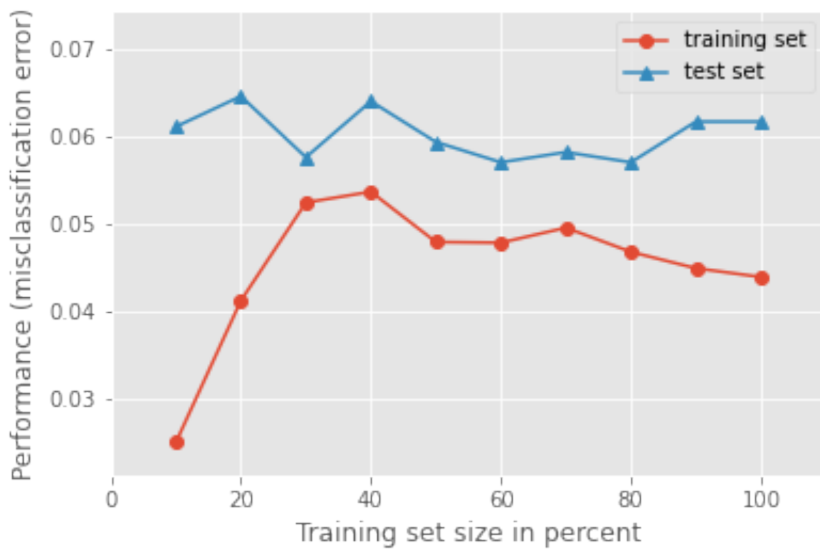
In [163]:
```
import itertools
import numpy as np

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



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 [167]:
```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=1)

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

In [168]:
```
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):
    boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=n_e
    boosting.fit(X_new, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X_new, y=y, clf=boosting, legend=2)
    plt.title(label)

plt.show()
```

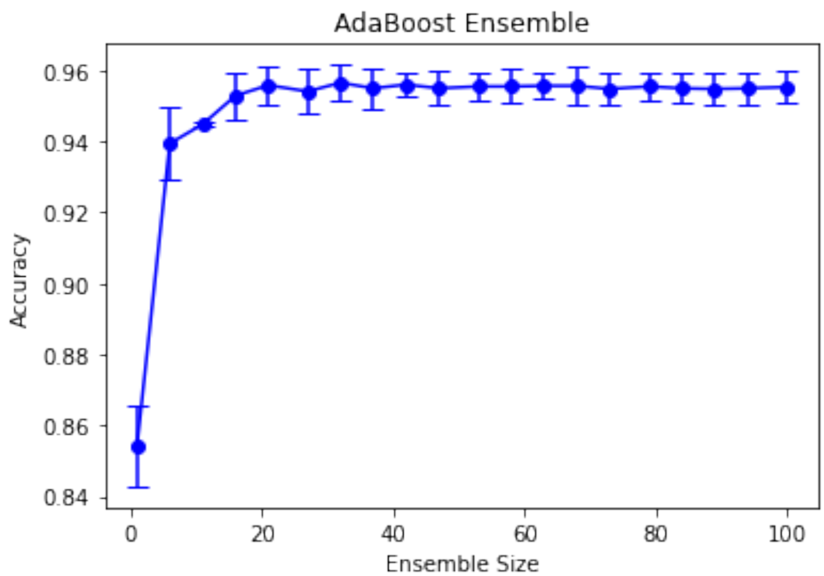


In [169]:
```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_siz

boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=10)

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, boosting, print
plt.show()
```



In [170]:
```
#Ensemble Size
num_est = np.linspace(1,100,20).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_es
    scores = cross_val_score(ada_clf, X_new, y, cv=3, scoring='accurac
    bg_clf_cv_mean.append(scores.mean())
    bg_clf_cv_std.append(scores.std())
```

In [171]:
```
plt.figure()
(_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_st
for cap in caps:
    cap.set_markeredgewidth(1)
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('AdaBoo
plt.show()
```



Stacking

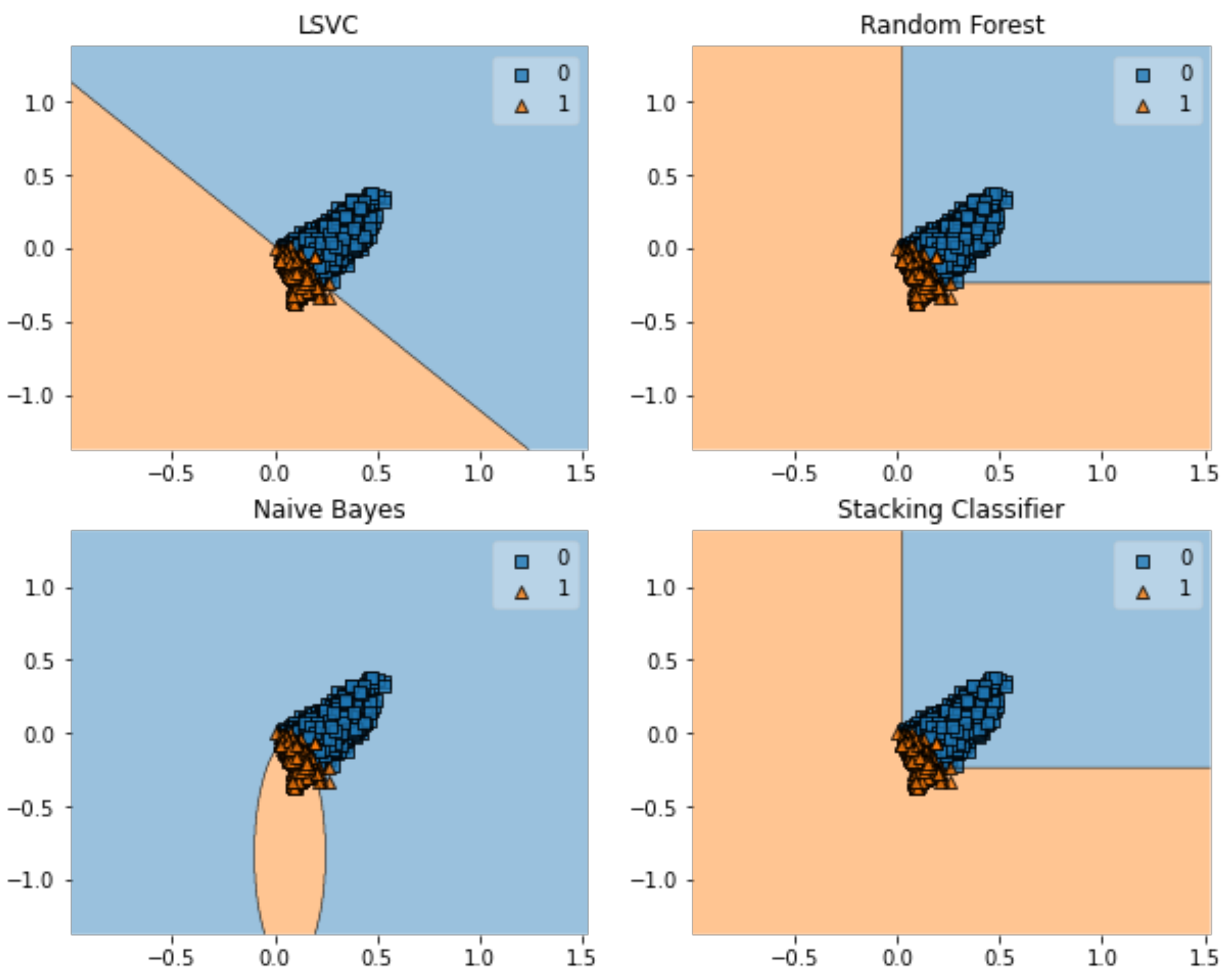
In [174]:
```
from mlxtend.classifier import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
```

In [178]:
```
clf1 = LinearSVC()
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                          meta_classifier=lr)
```

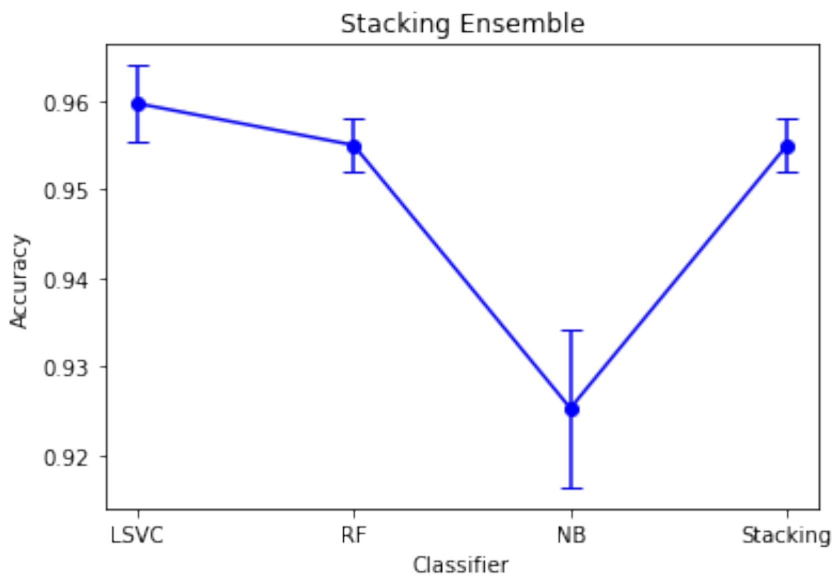
In [184]:
```
label = ['LSVC', 'Random Forest', 'Naive Bayes', 'Stacking Classifier'
clf_list = [clf1, clf2, clf3, sclf]

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):
```

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

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

plt.show()
```

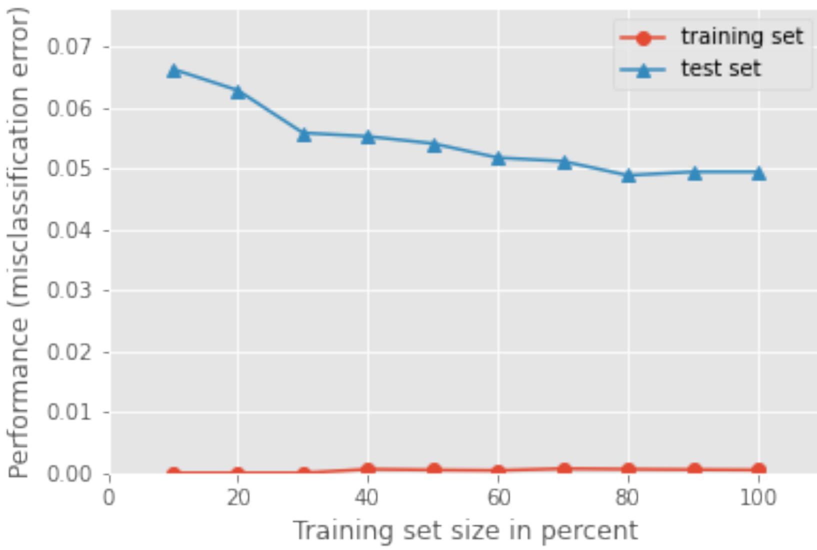
```
Accuracy: 0.96 (+/- 0.00) [LSVC]
Accuracy: 0.95 (+/- 0.00) [Random Forest]
Accuracy: 0.93 (+/- 0.01) [Naive Bayes]
Accuracy: 0.95 (+/- 0.00) [Stacking Classifier]
```



In [185]:
```
#plot classifier accuracy
plt.figure()
(_, caps, _) = plt.errorbar(range(4), clf_cv_mean, yerr=clf_cv_std, c=
for cap in caps:
    cap.set_markeredgewidth(1)
plt.xticks(range(4), ['LSVC', 'RF', 'NB', 'Stacking'])
plt.ylabel('Accuracy'); plt.xlabel('Classifier'); plt.title('Stacking
plt.show()
```



In [183]:
```
#plot learning curves
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_siz

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, sclf, print_moc
plt.show()
```



In [ ]:

In [ ]:

In [138]:
```
from gensim.models import word2vec
#import nltk
#from nltk.tokenize import word_tokenize


# tokenize sentences in corpus
#wpt = nltk.WordPunctTokenizer()
#tokenized_corpus = [wpt.tokenize(document) for document in nlp_X]

# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 30          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

w2v_model = word2vec.Word2Vec(nlp_X, size=feature_size,
                          window=window_context, min_count=min_word_co
                          sample=sample, iter=50)
```

In [ ]:
```
#Visualizing word embeddings
```

In [140]:
```
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector


def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabu
                    for tokenized_sentence in corpus]
    return np.array(features)
```

In [143]:
```
w2v_feature_array = averaged_word_vectorizer(corpus=nlp_X, model=w2v_m
                                             num_features=feature_size
```

In [144]:
```
w2v_df=pd.DataFrame(w2v_feature_array)
```

In [147]:
```
w2v_df.shape
```

Out[147]: (5728, 100)

In [ ]:

In [ ]:

In [ ]:

In [133]:
```
X_df = pd.DataFrame(np.round(tv_nlp_X, 2), columns=vocab)
```

In [135]:
```
X_features= X_df[['vince','enron','research','attached','meeting','sub
```

In [ ]:
```
svd = TruncatedSVD(n_components=5, random_state=42)
data = svd.fit_transform(data)
```

In [ ]:

In [148]:
```
from sklearn.cluster import AffinityPropagation

ap = AffinityPropagation()
ap.fit(w2v_feature_array)
cluster_labels = ap.labels_
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel']
pd.concat([nlp_emails, cluster_labels], axis=1)
```

Out[148]:

	Unnamed: 0	text	spam	nlp_X	ClusterLabel
0	0	Subject: naturally irresistible your corporate...	1	subject naturally irresistible corporate ident...	11
1	1	Subject: the stock trading gunslinger fanny i...	1	subject stock trading gunslinger fanny merrill...	20
2	2	Subject: unbelievable new homes made easy im ...	1	subject unbelievable new homes easy im wanting...	193
3	3	Subject: 4 color printing special request add...	1	subject color printing special request additio...	225
4	4	Subject: do not have money , get software cds ...	1	subject money software cds software compatibil...	84
...	...	...	...	...	...
5723	5723	Subject: re : research and development charges...	0	subject research development charges gpg forwa...	209
5724	5724	Subject: re : receipts from visit jim , than...	0	subject receipts visit jim thanks invitation v...	79
5725	5725	Subject: re : enron case study update wow ! a...	0	subject enron case study update wow day super ...	175
5726	5726	Subject: re : interest david , please , call...	0	subject interest david shirley crenshaw assist...	143
5727	5727	Subject: news : aurora 5 . 2 update aurora ve...	0	subject news aurora update aurora version fast...	50

5728 rows × 5 columns

In [ ]:
```
new_nlp_emails = nlp_email[[]]
```

In [ ]:

In [ ]:

In [ ]:

In [109]:
```
from sklearn.impute import SimpleImputer

# Create an imputer object with a median filling strategy
imputer = SimpleImputer(strategy='median')

# Train on the training features
imputer.fit(X_df)

# Transform both training data and testing data
X = imputer.transform(X_df)
```

In [110]:

Out[110]:
```
array([[0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       ...,
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.14, ..., 0.  , 0.  , 0.  ]])
```

In [66]:
```
from sklearn.preprocessing import StandardScaler
```

In [89]:
```
X = tv_nlp_X
```

In [90]:
```
X.shape
```

Out[90]:
```
(5728, 2500)
```

In [91]:
```
scaler = StandardScaler()
```

In [92]:
```
X_scaler = scaler.fit_transform(X)
```

In [93]:
```
X_scaler
```

Out[93]:
```
array([[-0.06046171, -0.03520993, -0.12575412, ..., -0.06327534,
        -0.17307057, -0.07670435],
       [-0.06046171, -0.03520993, -0.12575412, ..., -0.06327534,
        -0.17307057, -0.07670435],
       [-0.06046171, -0.03520993, -0.12575412, ..., -0.06327534,
        -0.17307057, -0.07670435],
       ...,
       [-0.06046171, -0.03520993, -0.12575412, ..., -0.06327534,
        -0.17307057, -0.07670435],
       [-0.06046171, -0.03520993, -0.12575412, ..., -0.06327534,
        -0.17307057, -0.07670435],
       [-0.06046171, -0.03520993, 10.07568087, ..., -0.06327534,
        -0.17307057, -0.07670435]])
```

In [112]:
```
X_scaler = pd.DataFrame(np.round(X_scaler, 2))
```

In [116]:
```
X_scaler.head()
```

Out[116]:

	0	1	2	3	4	5	6	7	8	9	...	2490	2491	2492	2493
0	-0.06	-0.04	-0.13	-0.24	-0.08	-0.06	-0.11	-0.1	-0.08	-0.06	...	-0.08	-0.27	-0.22	-0.13
1	-0.06	-0.04	-0.13	-0.24	-0.08	-0.06	-0.11	-0.1	-0.08	-0.06	...	-0.08	-0.27	-0.22	23.85
2	-0.06	-0.04	-0.13	-0.24	-0.08	-0.06	-0.11	-0.1	-0.08	-0.06	...	-0.08	-0.27	-0.22	-0.13
3	-0.06	-0.04	-0.13	-0.24	-0.08	-0.06	-0.11	-0.1	-0.08	-0.06	...	-0.08	-0.27	-0.22	-0.13
4	-0.06	-0.04	-0.13	-0.24	-0.08	-0.06	-0.11	-0.1	-0.08	-0.06	...	-0.08	-0.27	-0.22	-0.13

5 rows × 2500 columns

In [113]:
```
y_df = pd.DataFrame(y)
```

In [123]:
```
y
```

Out[123]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

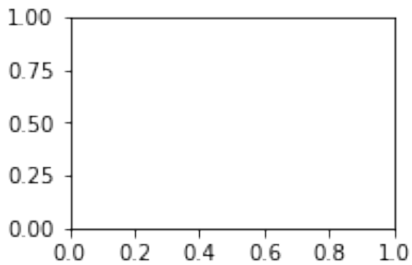
In [122]:
```
X = X_scaler.iloc[:,0:2499].values
y = y_df[0].values
y = y.reshape(-1)
```

In [127]:
```
label = ['Decision Tree', 'K-NN', 'Bagging Tree', 'Bagging K-NN']
clf_list = [clf1, clf2, bagging1, bagging2]

fig = plt.figure(figsize=(10, 8))
```

```
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

    clf.fit(X, y)
    value = 1.5
    width = 0.75

    fig, ax = plt.subplots()



    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=clf,filler_feature_value
                          filler_feature_ranges={2: width, 3: width, 4: wi
                          res=0.02, legend=2, ax=ax)
    plt.title(label)

plt.show()
```

```
Accuracy: 0.76 (+/- 0.00) [Decision Tree]

---------------------------------------------------------------------
------
ValueError                                Traceback (most recent call
last)
<ipython-input-127-b4f1cc748a46> in <module>
     19
     20     ax = plt.subplot(gs[grd[0], grd[1]])
---> 21     fig = plot_decision_regions(X=X, y=y, clf=clf,filler_feat
ure_values={2: value, 3: value, 4: value, 5: value},
     22                               filler_feature_ranges={2: width, 3:
width, 4: width, 5: width},
     23                               res=0.02, legend=2, ax=ax)

~/opt/anaconda3/lib/python3.8/site-packages/mlxtend/plotting/decision
_regions.py in plot_decision_regions(X, y, clf, feature_index, filler
_feature_values, filler_feature_ranges, ax, X_highlight, res, zoom_fa
ctor, legend, hide_spines, markers, colors, scatter_kwargs, contourf_
kwargs, scatter_highlight_kwargs)
    193         if not all(column_check):
    194             missing_cols = np.argwhere(~column_check).flatten
()
--> 195             raise ValueError(
    196                 'Column(s) {} need to be accounted for in eit
her '
```

```
    197                 'feature_index or filler_feature_values'.form
at(missing_cols))

ValueError: Column(s) [   6    7    8 ... 2496 2497 2498] need to be
accounted for in either feature_index or filler_feature_values


<Figure size 720x576 with 0 Axes>
```

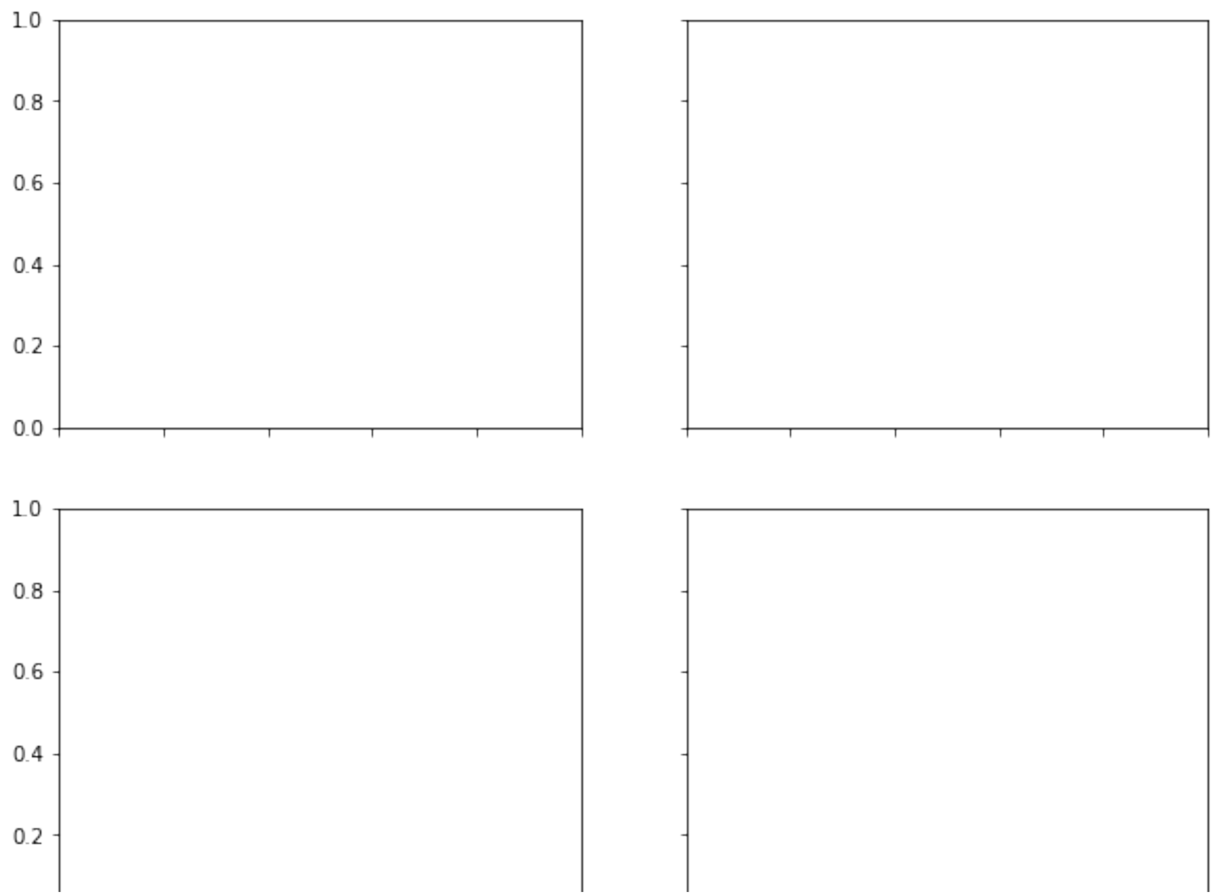


In [128]:

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC


# Training a classifier
svm = SVC(gamma='auto')
svm.fit(X, y)

# Plotting decision regions
fig, axarr = plt.subplots(2, 2, figsize=(10,8), sharex=True, sharey=Tr
values = [-4.0, -1.0, 1.0, 4.0]
width = 0.75
for value, ax in zip(values, axarr.flat):
    plot_decision_regions(X, y, clf=svm,
                          filler_feature_values={2: value},
                          filler_feature_ranges={2: width},
                          legend=2, ax=ax)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
    ax.set_title('Feature 3 = {}'.format(value))

# Adding axes annotations
fig.suptitle('SVM ')
plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------
------
ValueError                                Traceback (most recent call
```

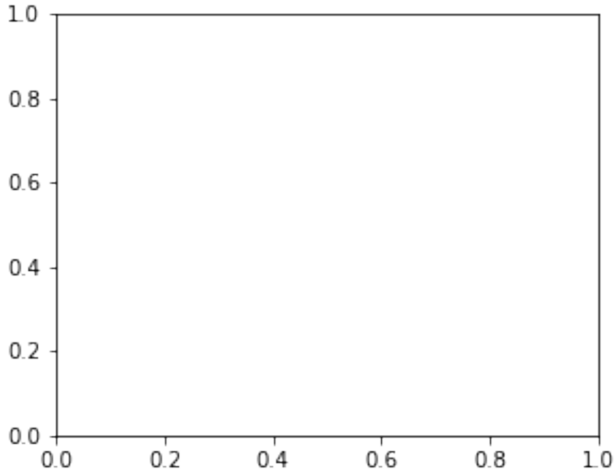
```
last)
<ipython-input-128-8e4c28318805> in <module>
     14 width = 0.75
     15 for value, ax in zip(values, axarr.flat):
---> 16     plot_decision_regions(X, y, clf=svm,
     17                           filler_feature_values={2: value},
     18                           filler_feature_ranges={2: width},

~/opt/anaconda3/lib/python3.8/site-packages/mlxtend/plotting/decision
_regions.py in plot_decision_regions(X, y, clf, feature_index, filler
_feature_values, filler_feature_ranges, ax, X_highlight, res, zoom_fa
ctor, legend, hide_spines, markers, colors, scatter_kwargs, contourf_
kwargs, scatter_highlight_kwargs)
    193         if not all(column_check):
    194             missing_cols = np.argwhere(~column_check).flatten
()
--> 195             raise ValueError(
    196                 'Column(s) {} need to be accounted for in eit
her '
    197                 'feature_index or filler_feature_values'.form
at(missing_cols))

ValueError: Column(s) [   3    4    5 ... 2496 2497 2498] need to be
accounted for in either feature_index or filler_feature_values
```



In [57]:

```python
label = ['Decision Tree', 'K-NN', 'Bagging Tree', 'Bagging K-NN']
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.stc

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

plt.show()
```

```
Accuracy: 0.76 (+/- 0.00) [Decision Tree]

---------------------------------------------------------------------
------
ValueError                                Traceback (most recent call
last)
<ipython-input-57-2052df8df26a> in <module>
     12     clf.fit(X, y)
     13     ax = plt.subplot(gs[grd[0], grd[1]])
---> 14     fig = plot_decision_regions(X=X, y=y, clf=clf)
     15     plt.title(label)
     16

~/opt/anaconda3/lib/python3.8/site-packages/mlxtend/plotting/decision
_regions.py in plot_decision_regions(X, y, clf, feature_index, filler
_feature_values, filler_feature_ranges, ax, X_highlight, res, zoom_fa
ctor, legend, hide_spines, markers, colors, scatter_kwargs, contourf_
kwargs, scatter_highlight_kwargs)
    176     if dim > 2:
    177         if filler_feature_values is None:
--> 178             raise ValueError('Filler values must be provided
when '
    179                              'X has more than 2 training feat
ures.')
    180

ValueError: Filler values must be provided when X has more than 2 tra
ining features.
```

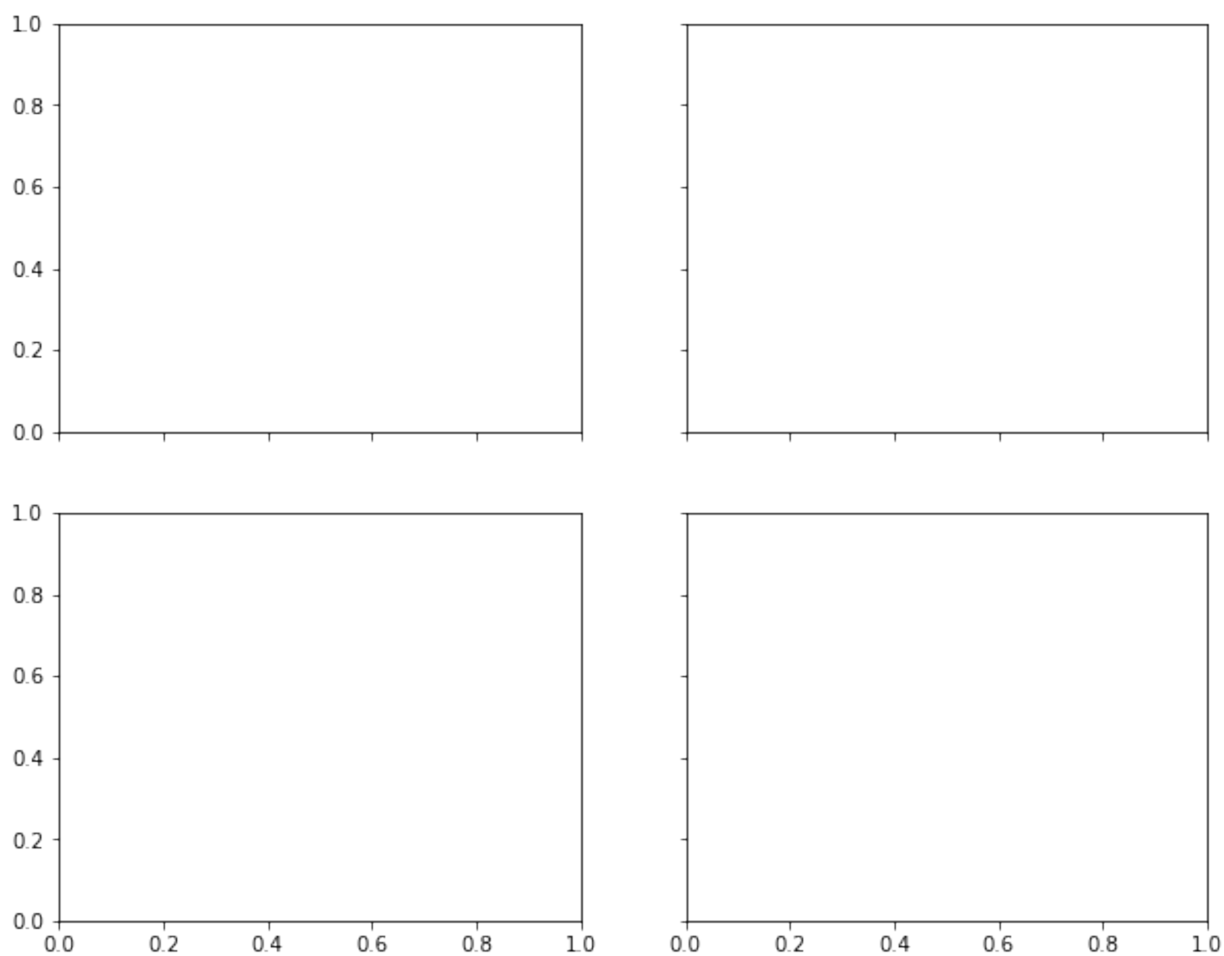


In [58]:

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC

# Training a classifier
svm = SVC(gamma='auto')
svm.fit(X, y)

# Plotting decision regions
fig, axarr = plt.subplots(2, 2, figsize=(10,8), sharex=True, sharey=Tr
values = [-4.0, -1.0, 1.0, 4.0]
width = 0.75
for value, ax in zip(values, axarr.flat):
    plot_decision_regions(X, y, clf=svm,
                          filler_feature_values={2: value},
                          filler_feature_ranges={2: width},
                          legend=2, ax=ax)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
    ax.set_title('Feature 3 = {}'.format(value))

# Adding axes annotations
fig.suptitle('SVM on Spam Emails')
plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------
------
ValueError                                Traceback (most recent call
last)
<ipython-input-58-e2f5f94244fc> in <module>
```

```
     13 width = 0.75
     14 for value, ax in zip(values, axarr.flat):
---> 15     plot_decision_regions(X, y, clf=svm,
     16                           filler_feature_values={2: value},
     17                           filler_feature_ranges={2: width},

~/opt/anaconda3/lib/python3.8/site-packages/mlxtend/plotting/decision
_regions.py in plot_decision_regions(X, y, clf, feature_index, filler
_feature_values, filler_feature_ranges, ax, X_highlight, res, zoom_fa
ctor, legend, hide_spines, markers, colors, scatter_kwargs, contourf_
kwargs, scatter_highlight_kwargs)
    193         if not all(column_check):
    194             missing_cols = np.argwhere(~column_check).flatten
()
--> 195             raise ValueError(
    196                 'Column(s) {} need to be accounted for in eit
her '
    197                 'feature_index or filler_feature_values'.form
at(missing_cols))

ValueError: Column(s) [   3    4    5 ... 2497 2498 2499] need to be
accounted for in either feature_index or filler_feature_values
```



In [ ]: