In [1]:
```
# setup
from mlwpy import *
%matplotlib inline
```

Baseline Classifiers

We can imagine four levels of learning systems:

1. **Baseline methods**—prediction based on simple statistics or random guesses,
2. **Simple off-the-shelf** learning methods—predictors that are generally less resource-intensive,
3. **Complex off-the-shelf** learning methods—predictors that are generally more resource-intensive, and
4. **Customized**, boutique learning methods.

In `sklearn`, there are ***four*** baseline classification methods (dummy calssifiers), Two baseline methods are random:

1. **uniform**: choose **evenly** among the target classes based on the number of classes
2. **stratified**: choose evenly among the target classes based on frequency of those classes

these methods will behave differently when a dataset has rare occurrences,lets say we have a dataset with 95% healthy and 5% sick the ***uniform*** will choose 50-50 between the two classes while the other it will choose based on the percentage of the classes so 95-5

Two methods return a constant value:

1. **constant** (surprise?): return one target class that we've picked out and
2. **most_frequent**: return the single most likely class. most_frequent is also available under the name **prior**

In [2]:
```
iris = datasets.load_iris()

tts = skms.train_test_split(iris.data, iris.target, test_size=.33, random_state=21)

(iris_train_ftrs, iris_test_ftrs, iris_train_tgt, iris_test_tgt) = tts
```

In [3]:
```
# normal usage: build-fit-predict-evaluate
baseline = dummy.DummyClassifier(strategy="most_frequent")
baseline.fit(iris_train_ftrs, iris_train_tgt)
base_preds = baseline.predict(iris_test_ftrs)
base_acc = metrics.accuracy_score(base_preds, iris_test_tgt)
print(base_acc)
```

```
0.3
```

In [4]:
```
# lets compare their performance
strategies = ['constant', 'uniform', 'stratified', 'prior', 'most_frequent']

# set up args to create different DummyClassifier strategies
baseline_args = [{'strategy':s} for s in strategies]
baseline_args[0]['constant'] = 0 # class 0 is setosa

accuracies = []
for bla in baseline_args:
    baseline = dummy.DummyClassifier(**bla)
    baseline.fit(iris_train_ftrs, iris_train_tgt)
    base_preds = baseline.predict(iris_test_ftrs)
    accuracies.append(metrics.accuracy_score(base_preds, iris_test_tgt))

display(pd.DataFrame({'accuracy':accuracies}, index=strategies))
```

	accuracy
constant	0.3600
uniform	0.3800
stratified	0.3400
prior	0.3000
most_frequent	0.3000

The uniform methods (uniform, stratified) will return different result every run but the other will be the same

Beyond Accuracy: Metrics for Classification

In [5]:
```
str(sorted(metrics.SCORERS.keys()))
```

Out[5]:
```
"['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_precision', 'balanced_accuracy', 'completeness_score', 'explained_variance', 'f1', 'f1_macro', 'f1_micro', 'f1_samples', 'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'jaccard', 'jaccard_macro', 'jaccard_micro', 'jaccard_samples', 'jaccard_weighted', 'max_error', 'mutual_info_score', 'neg_brier_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_absolute_percentage_error', 'neg_mean_gamma_deviance', 'neg_mean_poisson_deviance', 'neg_mean_squared_error', 'neg_mean_squared_log_error', 'neg_median_absolute_error', 'neg_root_mean_squared_error', 'normalized_mutual_info_score', 'precision', 'precision_macro', 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'rand_score', 'recall', 'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc', 'roc_auc_ovo', 'roc_auc_ovo_weighted', 'roc_auc_ovr', 'roc_auc_ovr_weighted', 'top_k_accuracy', 'v_measure_score']"
```

Not all of these are for classifiers

In [6]:
```
knn = neighbors.KNeighborsClassifier()

# help(knn.score) # verbose, but complete

print(knn.score.__doc__)
```

```
        Return the mean accuracy on the given test data and labels.

        In multi-label classification, this is the subset accuracy
        which is a harsh metric since you require for each sample that
        each label set be correctly predicted.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            Test samples.

        y : array-like of shape (n_samples,) or (n_samples, n_outputs)
            True labels for `X`.

        sample_weight : array-like of shape (n_samples,), default=None
            Sample weights.

        Returns
        -------
        score : float
            Mean accuracy of ``self.predict(X)`` wrt. `y`.
```

So the score metric for knn is the accuracy , which it might not be the best most of the time
let's say we build a model to predic spam emails and made the classifiers always choose emails as not spam, so if for every 100 email only 5 were spam then our model has a score/ accuracy is 95% which is high but our classifier is not actually classifying

Eliminating Confusion from the Confusion Matrix

`innocent untill proven guilty` if our classifier predict if a suspect guilty or not there could be two types of error

- He is innocent but we predicted to be guilty (we put an innocent person in prison)
- He is guilty but we predicted to be innocent (we free a guilty criminal)

  Both mistakes are bad

the both classes (guilty / innocent) are positive and negative outcome, we call the +ve: (1) the more risky, (2) the less likely, or (3) the more interesting outcome.

Another example is if we need to clean a metal pot, what kind of errors we could get

1. guess that is hot and not clean it but we will get in trouble for being laze
2. guess that is cold and get burnt when touching it

Ways of Being Wrong

	***I think*: pot is hot**	***I think*: pot is cold**
pot *is* hot	I thought hot it is hot I'm right	I thought cold it isn't cold I'm wrong
pot *is* cold	I thought hot it isn't hot I'm wrong	I thought cold it is cold I'm right

We are going to replace this example whith some general terms, Right--> True (i predicted correctly), Hot -->(Positive)

	***I think*: pot is hot (Positive)**	***I think*: pot is cold (Negative)**
pot *is* hot	True (predicted) Positive	False (predicted) Negative
pot *is* cold	False (predicted) Positive	True (predicted) Negative

SO `True/False` represent the prdiction whether i predicted right or wrong, `+ve/-ve` represent the class (hot is +ve and cold is -ve)

- **True Positive**, means that (1) I was correct (True) when (2) I claimed the pot was hot (Positive).
- **True Negative**, means that (1) I was correct (True) when (2) I claimed the pot was cpld (Negative).
- **False Positive**, means that (1) I was wrong (False) when (2) I claimed the pot was hot (Positive).
- **False Negative**, means that (1) I was wrong (False) when (2) I claimed the pot was hot (Negative).

	Predicted Positive (PredP)	**Predicted Negative (PredN)**
Real Positive (RealP)	True Positive (TP)	False Negative (FN)
Real Negative (RealN)	False Positive (FP)	True Negative (TN)

There are a few mathematical relationships here:

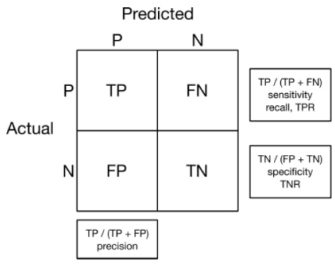
1. `RealP = TP + FN` and `RealN = FP + TN`
2. `PredP = TP + FP` and `PredN = FN + TN`

Metrics from the Confusion Matrix

If we are doctors we care how well we can define if the person is sick (+ve class), so we want to know form the real +ve how many did get right -- > $\frac{TP}{TP+FN}$, ***The term for this is*** `sensitivity` ***or*** `recall` ***or*** `TPR` ***(TRUE +ve rate)*** Where the error here is the **FN**

What if we care about the healthy people (-ve class), --> $\frac{TN}{TN+FP}$ **The term for this is `specificity` or TNR

What if i want to know from all the +ve prediction that where made how many of them were right -->--> $\frac{TP}{TP+FP}$, which is called `precision` . Try saying, "How precise are our positive predictions?

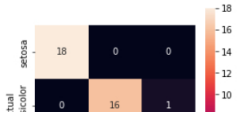


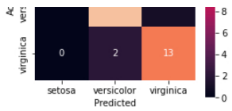
Coding the Confusion Matrix

In [7]:
```
tgt_preds = (neighbors.KNeighborsClassifier().fit(iris_train_ftrs, iris_train_tgt).predict(iris_test_ftrs))

print("accuracy:", metrics.accuracy_score(iris_test_tgt, tgt_preds))

cm = metrics.confusion_matrix(iris_test_tgt, tgt_preds)
print("confusion matrix:", cm, sep="\n")
```

```
accuracy: 0.94
confusion matrix:
[[18  0  0]
 [ 0 16  1]
 [ 0  2 13]]
```

In [8]:
```
fig, ax = plt.subplots(1, 1, figsize=(4, 4))
ax = sns.heatmap(cm, annot=True, square=True, xticklabels=iris.target_names, yticklabels=iris.target_names)
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual');
```





Dealing with Multiple Classes: Multiclass Averaging

Fom the previous example the percision for the 3 class are {1, $\frac{16}{18}$, $\frac{13}{14}$}, how do we combine these three values into one, well we could use the mean,This method of summarizing the predictions is called `macro` by sklearn

In [9]:
```
macro_prec = metrics.precision_score(iris_test_tgt, tgt_preds, average='macro')
print("macro:", macro_prec)

cm = metrics.confusion_matrix(iris_test_tgt, tgt_preds)
n_labels = len(iris.target_names)
print("should equal 'macro avg':",
# correct column # columns
(np.diag(cm) / cm.sum(axis=0)).sum() / n_labels)
```

```
macro: 0.9391534391534391
should equal 'macro avg': 0.9391534391534391
```

In [10]:
```
# micro takes all the correct predictions and divides by all the predictions we made.
# micro = (18 + 16 + 13 ) / (18 + 17 + 15)

print("micro:", metrics.precision_score(iris_test_tgt, tgt_preds, average='micro'))

cm = metrics.confusion_matrix(iris_test_tgt, tgt_preds)
print("should equal avg='micro':",
# TP.sum() / (TP&FP).sum() -->
# all correct / all preds
np.diag(cm).sum() / cm.sum())
```

```
micro: 0.94
should equal avg='micro': 0.94
```

In [11]:
```
print(metrics.classification_report(iris_test_tgt, tgt_preds))
# average is a weighted macro average (see text)

# verify sums-across-rows
cm = metrics.confusion_matrix(iris_test_tgt, tgt_preds)
print("row counts equal support:", cm.sum(axis=1))
```

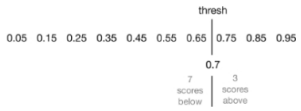
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        18
           1       0.89      0.94      0.91        17
           2       0.93      0.87      0.90        15

    accuracy                           0.94        50
   macro avg       0.94      0.94      0.94        50
weighted avg       0.94      0.94      0.94        50

row counts equal support: [18 17 15]
```

F_1 Score

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

ROC curves

Our classifier can do more than predicting the label of the class it can also the give us the probability of the prediction (how confident are we in the prediction) we can set a threshold to determine between the +ve and -ve class where the left of the bar is the -ve class and on its right is the +ve class

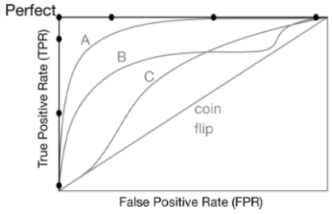


2 A medium-high threshold produces a low number of claimed hits (diseases).

every probability on the left of the threshold is considered -ve class and on its right is the +ve class, If we moved the bar all the way to the left (threshold = 0) all our prediction will be to the +ve class and the FN = 0 (as we don't predict -ve no more), If we moved the bar all the way to the right (threshold = 1) all our prediction will be to the -ve class and the FP = 0 (as we don't predict -ve no more)

How to choose the best threshold to solve this tradeoff
ROC curves are normally drawn in terms of sensitivity (also called true positive rate(TPR) or recall with respect to FPR (1 - TNR).
We want to have a high TPR: 1.0 is perfect. We want a low FPR: 0.0 is great

Patterns in the ROC



1. Low threshold means we move everything to be +ve class, This occurs in the top-right corner of the ROC graph, since we have a high number of +ve class then we have a high number of FP
2. A high threshold means we make many things negative, It's hard to get over a high bar to be a positive thats why there is little FP

Four things stand out in the ROC graph

1. The y = x line from the bottom-left corner to the top-right corner represents coin flipping: randomly guessing the target class. Any decent classifier should do better than this. Better means pushing towards the top-left of the diagram.
2. A perfect classifier lines up with the left side and the top of the box.
3. Classifier A does strictly better than B and C. A is more to the top-left than B and C. everywhere.

4. B and C flip-flop. At the higher thresholds B is better; at lower thresholds C is better.

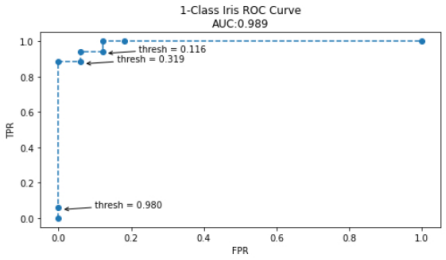
Binary ROC

First we need to our classification to binary classification, we will only work with one class and see if we predicted correct or not

In [12]:
```
is_versicolor = iris.target == 1   # boolean col (1 for versicolor 0 otherwise)

tts_1c = skms.train_test_split(iris.data, is_versicolor, test_size=.33, random_state = 21)
(iris_1c_train_ftrs, iris_1c_test_ftrs, iris_1c_train_tgt, iris_1c_test_tgt) = tts_1c

# build, fit, predict (probability scores) for NB model
gnb = naive_bayes.GaussianNB()
prob_true = (gnb.fit(iris_1c_train_ftrs, iris_1c_train_tgt).predict_proba(iris_1c_test_ftrs)[:, 1]) # all rows for the True col
# predict_proba return arraylike with the shape[n of samples, n of classes]=="True"
```

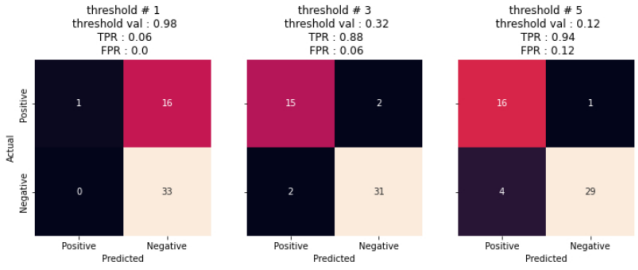
In [13]:
```
fpr, tpr, thresh = metrics.roc_curve(iris_1c_test_tgt, prob_true)
auc = metrics.auc(fpr, tpr)
print("FPR       : {}".format(fpr), "TPR       : {}".format(tpr),"Threshold : {}".format(thresh), sep='\n')

# create the main graph
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(fpr, tpr, 'o--')
ax.set_title("1-Class Iris ROC Curve\nAUC:{:.3f}".format(auc))
ax.set_xlabel("FPR")
ax.set_ylabel("TPR");

# do a bit of work to label some points with their
# respective thresholds
investigate = np.array([1, 3, 5])
for idx in investigate:
    th, f, t = thresh[idx], fpr[idx], tpr[idx]
    ax.annotate('thresh = {:.3f}'.format(th),
    xy=(f+.01, t-.01), xytext=(f+.1, t),
    arrowprops = {'arrowstyle':'->'})
```

```
FPR       : [0.     0.     0.     0.0606 0.0606 0.1212 0.1212 0.1818 1.    ]
TPR       : [0.     0.0588 0.8824 0.8824 0.9412 0.9412 1.     1.     1.    ]
Threshold : [1.9799 0.9799 0.3937 0.319  0.2639 0.116  0.1061 0.05   0.    ]
```



Remember that each point represent a different confusion matrix

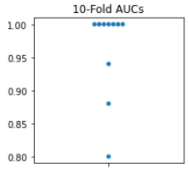
In [14]:
```
itle_fmt = "Threshold {}\n~{:5.3f}\nTPR : {:.3f}\nFPR : {:.3f}"

pn = ['Positive', 'Negative']
add_args = {'xticklabels': pn, 'yticklabels': pn, 'square':True}

fig, axes = plt.subplots(1, 3, sharey = True, figsize=(12, 4))

for ax, thresh_idx in zip(axes.flat, investigate):

    preds_at_th = prob_true < thresh[thresh_idx]
    cm = metrics.confusion_matrix(1-iris_1c_test_tgt, preds_at_th)
    sns.heatmap(cm, annot=True, cbar=False, ax=ax, **add_args)

    ax.set_xlabel('Predicted')
    ax.set_title('threshold # '+ str(thresh_idx) +
                 '\n threshold val : ' +  str(round(thresh[thresh_idx], 2)) +
                 '\n TPR : ' +  str(round(tpr[thresh_idx], 2)) +
                 '\n FPR : ' + str(round(fpr[thresh_idx], 2)))

axes[0].set_ylabel('Actual');
# note: e.g. for threshold 3
# FPR = 1-spec = 1 - 31/(31+2) = 1 - 31/33 = 0.0606...
```



AUC: Area-Under-the-(ROC)-Curve

The AUC is an overall measure of classifier performance at a series of thresholds. It summarizes a lot of information and subtlety into one number

In [15]:
```
fig,ax = plt.subplots(1, 1, figsize=(3, 3))
model = neighbors.KNeighborsClassifier(3)
cv_auc = skms.cross_val_score(model, iris.data, iris.target==1, scoring='roc_auc', cv=10)
ax = sns.swarmplot(y = cv_auc)
ax.set_title('10-Fold AUCs');
```



Many of the folds have perfect results.

Multiclass Learners, One-versus-Rest, and ROC

metrics.roc_curve is ill-equipped to deal with multiclass problems, we can come around that using `OvR` (one versus rest). This is similar to what we did before in the binary ROC
(class 0 versus 1,2 - class 1 versus 0,2 - class 2 versus 0, 1)

We can do this using `label_binarize(Data to encode, data classes) and it return a matrix of (n record, n classes)`

In [16]:
```
checkout = [0, 50, 100]
print("Original Encoding")
print(iris.target[checkout])
```

```
Original Encoding
[0 1 2]
```

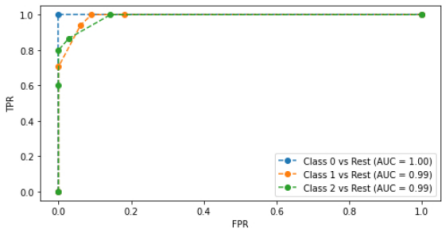
In [17]:
```
print("'Multi-label' Encoding")
print(skpre.label_binarize(iris.target, [0, 1, 2])[checkout])
```

```
'Multi-label' Encoding
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

In [18]:
```
iris_multi_tgt = skpre.label_binarize(iris.target, [0, 1, 2])

# im --> "iris multi"

(im_train_ftrs, im_test_ftrs,im_train_tgt, im_test_tgt) = skms.train_test_split(iris.data
                                                                                ,iris_multi_tgt
                                                                                ,test_size=.33
                                                                                ,random_state=21)

im_test_ftrs.shape
```

Out[18]: (50, 4)

In [19]:
```
# knn wrapped up in one-versus-rest (3 classifiers)
knn = neighbors.KNeighborsClassifier(n_neighbors=5)
ovr_knn = skmulti.OneVsRestClassifier(knn)
pred_probs = (ovr_knn.fit(im_train_ftrs, im_train_tgt).predict_proba(im_test_ftrs))

# make ROC plots
lbl_fmt = "Class {} vs Rest (AUC = {:.2f})"
fig,ax = plt.subplots(figsize=(8, 4))
for cls in [0, 1, 2]:
    fpr, tpr, _ = metrics.roc_curve(im_test_tgt[:,cls], pred_probs[:,cls])
    label = lbl_fmt.format(cls, metrics.auc(fpr,tpr))
    ax.plot(fpr, tpr, 'o--', label=label)
ax.legend()
ax.set_xlabel("FPR")
ax.set_ylabel("TPR");
```



Another Take on Multiclass:One-versus-One

another way for one-vs-rest is one-vs-one, So instead on class 0 vs 1, 2 it becomes 0 vs 1 and 0 vs 2 ans so on for the other classes
Note that the result is normalized thats why is not a whole number

so what we do with the result, we will have 2 result for every class. We sum the result and the class with the highest sum wins

In [20]:
```
knn = neighbors.KNeighborsClassifier(n_neighbors=5)
ovo_knn = skmulti.OneVsOneClassifier(knn)
pred_scores = (ovo_knn.fit(iris_train_ftrs, iris_train_tgt).decision_function(iris_test_ftrs)) # return the result sum for each c
df = pd.DataFrame(pred_scores)
df['class'] = df.values.argmax(axis=1) # the winnin class is the max sum
display(df.head())
```

	0	1	2	class
0	-0.2222	2.1667	1.1667	1
1	2.0000	1.0000	0.0000	0
2	2.0000	1.0000	0.0000	0
3	2.0000	1.0000	0.0000	0
4	-0.2222	2.1667	1.1667	1

Precision-Recall Curves

Just as we can look at the tradeoffs between sensitivity and specificity with ROC curves, we can evaluate the tradeoffs between precision and recall.

steps:

1. Train a model.
2. Get classification scores for each example.
3. Create a blank table for each pairing of classes.
4. For each pair of classes c1 and c2:
   - Find AUC of c1 against c2.
   - Find AUC of c2 against c1.
   - Entry for c1, c2 is the average of these AUCs.
5. Overall value is the average of the entries in the table.

A Note on Precision-Recall Tradeoff

There is a very important difference between the sensitivity-specificity curve and the precision-recall curve. With sensitivity-specificity, the two values represent portions of the .row totals. The tradeoff is between performance on real-world positives and negatives.

With precision-recall, we are dealing a column piece and a row piece out of the confusion matrix. So, they can vary more independently of each other More importantly, an increasing precision does not imply an increasing recall.

In this example we have precision = 0.5 and recall = 0.5 for the +ve class

In this example we have percision = 0.5 and recall = 0.5 for the +ve class

	PredP	PredN
RealP	5	5
RealN	5	5

consider what happens when we raise the threshold, We get more negtive prediction

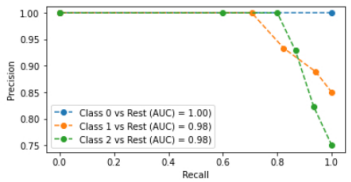
	PredP	PredN
RealP	3	7
RealN	3	7

Our percision stays the same = 0.6 but the recall is now worse = 0.3

Constructing a Precision-Recall Curve (PRC)

Similar to ROC but here we want both values to high (100%)

In [25]:
```
fig,ax = plt.subplots(figsize=(6, 3))
for cls in [0, 1, 2]:
    precision, recall, thr = metrics.precision_recall_curve(im_test_tgt[:,cls], pred_probs[:,cls])
    prc_auc = metrics.auc(recall, precision)
    label = "Class {} vs Rest (AUC) = {:.2f})".format(cls, prc_auc)
    ax.plot(recall, precision, 'o--', label=label)
ax.legend()
ax.set_xlabel('Recall')
ax.set_ylabel('Precision');
```



The PRC for class 0 is perfect

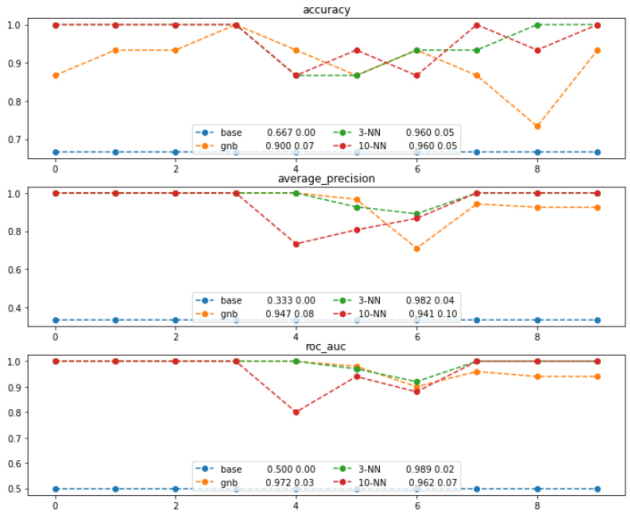
6.7 More Sophisticated Evaluation of Classifiers: Take Two

Binary

In [26]:
```
classifiers = {'base' : baseline,
               'gnb' : naive_bayes.GaussianNB(),
               '3-NN' : neighbors.KNeighborsClassifier(n_neighbors=10),
               '10-NN' : neighbors.KNeighborsClassifier(n_neighbors=3)}
```

In [27]:
```
# define the one_class iris problem so we don't have random ==1 around
iris_onec_ftrs = iris.data
iris_onec_tgt = iris.target == 1
```

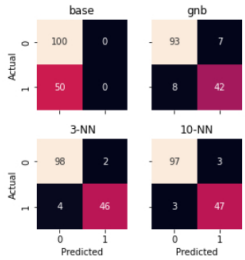
In [34]:
```
msrs = ['accuracy', 'average_precision', 'roc_auc']

fig, axes = plt.subplots(len(msrs), 1, figsize=(10, 8))
fig.tight_layout()

# each iteration will work with one classifier and get compute the 3 metrics for every cv (10 values for metric)
for mod_name, model in classifiers.items():
    cvs = skms.cross_val_score
    cv_results = {msr:cvs(model, iris_onec_ftrs, iris_onec_tgt,scoring = msr, cv=10) for msr in msrs} # {metric : array of 10 }

    # each iteration will plot one metric on one axis (at the end w'll have three plots with metric n)
    for ax, msr in zip(axes, msrs):
        msr_results = cv_results[msr]
        my_lbl = "{:12s} {:.3f} {:.2f}".format(mod_name, msr_results.mean(), msr_results.std()) # the box at the bottom
        ax.plot(msr_results, 'o--', label=my_lbl) # plot the 10 metric for every cv ( x axis is 1->10 and y = val)
        ax.set_title(msr)
        ax.legend(loc='lower center', ncol=2)
```



To see more details on where the precision comes from or what the ROC curves look like, we need to peel back a few layers

In [36]:
```
fig, axes = plt.subplots(2, 2, figsize=(4, 4), sharex=True, sharey=True) # 2*2 plots
fig.tight_layout()

# first iter -- > 0, ('base' , baseline)
#
#
# last iter -- > 4,('10-NN' : neighbors.KNeighborsClassifier(n_neighbors=3)

for ax, (mod_name, model) in zip(axes.flat, classifiers.items()):
    preds = skms.cross_val_predict(model, iris_onec_ftrs, iris_onec_tgt, cv=10)
```

```
    cm = metrics.confusion_matrix(iris.target==1, preds)
    sns.heatmap(cm, annot=True, ax=ax, cbar=False, square=True, fmt="d")
    ax.set_title(mod_name)

axes[1, 0].set_xlabel('Predicted')
axes[1, 1].set_xlabel('Predicted')
axes[0, 0].set_ylabel('Actual')
axes[1, 0].set_ylabel('Actual');
```



A Novel Multiclass Problem

In [ ]: