6.2 Evaluating Classification Models

May 9, 2019

1 6.2 Evaluating Classification Models

Just as with regression models, we need ways to measure how good a classification model is. With regression models, the main metrics were MSE, RMSE, and MAE. With classification models, the main metrics are accuracy, precision, and recall. All of these metrics can be calculated on either the training data or the test data. We can also use cross validation to estimate the value of the metric on test data.

First, let's train a 9-nearest neighbors model on the wine data, just so that we have a model to evaluate. The following code is copied from Section 5.1.

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
pd.options.display.max_rows = 5

reds = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/da
                   sep=";")
whites = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/
                     sep=";")

reds["color"] = "red"
whites["color"] = "white"

wines = pd.concat([reds, whites],
                  ignore_index=True)
```

In [2]:
```
wines.head()
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	\
0	7.4	0.70	0.00	1.9	0.076	
1	7.8	0.88	0.00	2.6	0.098	
2	7.8	0.76	0.04	2.3	0.092	
3	11.2	0.28	0.56	1.9	0.075	
4	7.4	0.70	0.00	1.9	0.076	

	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	\
0	11.0	34.0	0.9978	3.51	0.56	

```
1                25.0                  67.0   0.9968  3.20       0.68
2                15.0                  54.0   0.9970  3.26       0.65
3                17.0                  60.0   0.9980  3.16       0.58
4                11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality color
0      9.4        5   red
1      9.8        5   red
2      9.8        5   red
3      9.8        6   red
4      9.4        5   red
```

```
In [3]: from sklearn.preprocessing import StandardScaler
        from sklearn.neighbors import KNeighborsClassifier

        # define the training data
        X_train = wines[["volatile acidity", "total sulfur dioxide"]]
        y_train = wines["color"]

        # standardize the data
        scaler = StandardScaler()
        scaler.fit(X_train)
        X_train_sc = scaler.transform(X_train)

        # fit the 9-nearest neighbors model
        model = KNeighborsClassifier(n_neighbors=9)
        model.fit(X_train_sc, y_train)
```

```
Out[3]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=9, p=2,
                    weights='uniform')
```

We will start by calculating training metrics, so we need predictions for the observations in the training data.

```
In [4]: y_train_pred = model.predict(X_train_sc)
        y_train_pred
```

```
Out[4]: array(['red', 'red', 'red', ..., 'white', 'white', 'white'], dtype=object)
```

2 Metrics for Classification

Because the labels y_i in a classification model are categorical, we cannot calculate the difference $y_i - \hat{y}_i$ between the actual and predicted labels, as we did for regression model. But we can determine if the predicted label $\hat{y}_i$ is correct ($\hat{y}_i = y_i$) or not ($\hat{y}_i \neq y_i$). For example, the **error rate** is defined to be:

$$\text{error rate} = \text{proportion where } \hat{y}_i \neq y_i$$

With classification models, it is more common to report the performance in terms of a score, like **accuracy**, where a higher value is better:

$$\text{accuracy} = \text{proportion where } \hat{y}_i = y_i$$

```
In [5]: accuracy = (y_train_pred == y_train).mean()
        accuracy
```

```
Out[5]: 0.96059719870709559
```

If you ever forget how to calculate accuracy, you can have Scikit-Learn do it for you. It just needs to know the true labels and the predicted labels:

```
In [6]: from sklearn.metrics import accuracy_score

        accuracy_score(y_train, y_train_pred)
```

```
Out[6]: 0.96059719870709559
```

The problem with accuracy is that it is sensitive to the initial distribution of classes in the training data. For example, if 99% of the wines in the data set were white, it would be trivial to obtain a model with 99% accuracy: the model could simply predict that every wine is white. Even though such a model has high overall accuracy, it is remarkably bad for red wines. We need some way to measure the "accuracy" of a model for a particular class.

Suppose we want to know the "accuracy" of our model for class c. There are two ways to interpret "accuracy for class c". Do we want to know the accuracy among the observations our model *predicted to be* in class c or the accuracy among the observations that *actually were* in class c? The two options lead to two different notions of "accuracy" for class c: precision and recall.

The **precision** of a model for class c is the proportion of observations predicted to be in class c that actually were in class c.

$$\text{precision for class } c = \frac{\#\{i : \hat{y}_i = c \text{ and } y_i = c\}}{\#\{i : \hat{y}_i = c\}}.$$

The **recall** of a model for class c is the proportion of observations actually in class c that were predicted to be in class c.

$$\text{recall for class } c = \frac{\#\{i : \hat{y}_i = c \text{ and } y_i = c\}}{\#\{i : y_i = c\}}.$$

Another way to think about precision and recall is in terms of true positives (TP) and false positives (FP). A "positive" is an observation that the model identified as belonging to class c (i.e., $\hat{y}_i = c$). A true positive is one that actually was in class c (i.e., $\hat{y}_i = c$ and $y_i = c$), while a false positive is one that was not (i.e., $\hat{y}_i = c$ and $y_i \neq c$). True and false *negatives* are defined analogously.

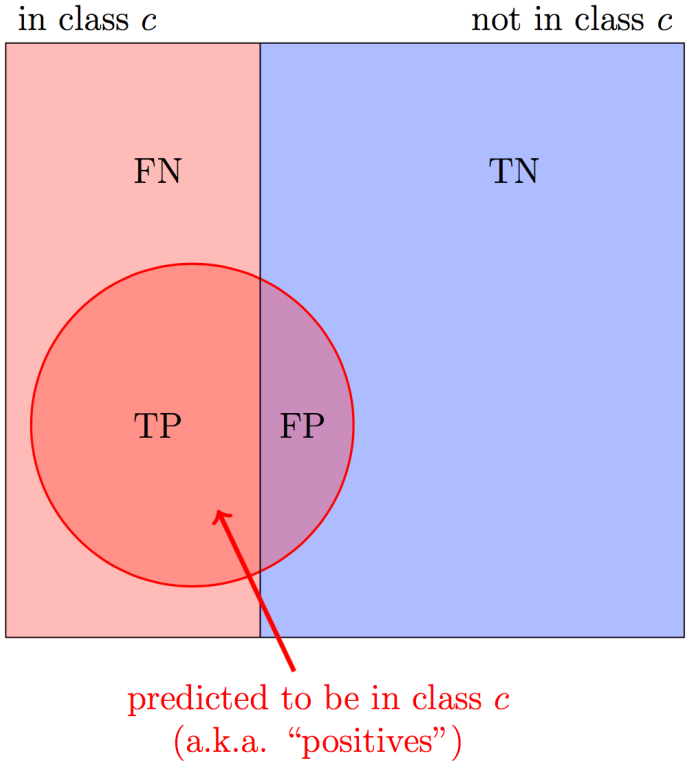
In the language of positives and negatives, the precision is the proportion of positives that are true positives:

$$\text{precision for class } c = \frac{TP}{TP + FP},$$

while the recall is the proportion of observations in class c that are positives (as opposed to negatives):

$$\text{recall for class } c = \frac{TP}{TP + FN}.$$

The diagram below may help you to remember which numbers go in the numerator and denominator. The precision is the proportion of the red rectangle that is a TP, while the recall is the proportion of the red circle that is a TP.



Let's calculate the precision and recall of our 9-nearest neighbors model for the red "class":

```
In [7]: true_positives = ((y_train_pred == "red") & (y_train == "red")).sum()

        precision = true_positives / (y_train_pred == "red").sum()
        recall = true_positives / (y_train == "red").sum()

        precision, recall
```

```
Out[7]: (0.92634920634920637, 0.91244527829893685)
```

Again, you can have Scikit-Learn calculate precision and recall for you. These functions work similarly to `accuracy_score` above, except we have to explicitly specify the class for which we want the precision and recall. For example, to calculate the precision and recall of the model for red wines:

```
In [8]: from sklearn.metrics import precision_score, recall_score

        (precision_score(y_train, y_train_pred, pos_label="red"),
         recall_score(y_train, y_train_pred, pos_label="red"))
```

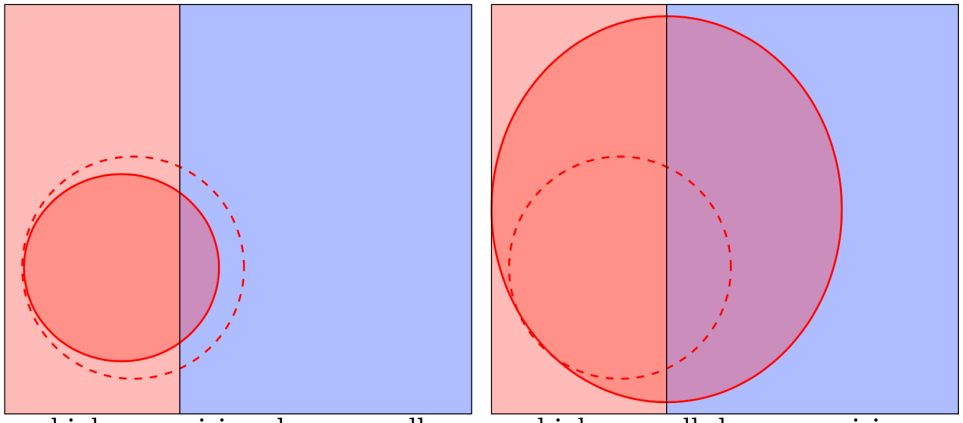
```
Out[8]: (0.92634920634920637, 0.91244527829893685)
```

It is important to specify `pos_label` because the precision and recall for other classes may be quite different:

```
In [9]: (precision_score(y_train, y_train_pred, pos_label="white"),
         recall_score(y_train, y_train_pred, pos_label="white"))
```

```
Out[9]: (0.97155627793579846, 0.97631686402613316)
```

In general, there is a tradeoff between precision and recall. For example, you can improve recall by predicting more observations to be in class c, but this will hurt precision. To take an extreme example, a model that predicts that *every* observation is in class c has 100% recall, but its precision would likely be poor. To visualize this phenomenon, suppose we expand the positives from the dashed circle to the solid circle, as shown in the figure below, at right. This increases recall (because the circle now covers more of the red rectangle) but decreases precision (because the red rectangle now makes up a smaller fraction of the circle).



higher precision, lower recall higher recall, lower precision

Likewise, you can improve precision by predicting fewer observations to be in class c (i.e., only the ones you are very confident about), but this will hurt recall. This is illustrated in the figure above, at left.

3 Validation Accuracy, Precision, and Recall in Scikit-Learn

We calculated the training accuracy of our classifier above. However, test accuracy is more useful in most cases. We can estimate the test accuracy using cross validation. We will have Scikit-Learn carry out the cross validation for us, including the computation of the accuracy score on each held-out subsample. We simply have to specify the right `scoring=` method.

```
In [10]: from sklearn.pipeline import Pipeline
         from sklearn.model_selection import cross_val_score

         pipeline = Pipeline([
             ("scaler", scaler),
             ("model", model)
         ])
```

```
        cross_val_score(pipeline, X_train, y_train,
                        cv=10, scoring="accuracy")
```

```
Out[10]: array([ 0.94       ,  0.94615385,  0.94615385,  0.94307692,  0.95692308,
                 0.96769231,  0.94923077,  0.94461538,  0.96918336,  0.94290123])
```

To obtain a single estimate of test accuracy from the 10 validation accuracies, we can take their average:

```
In [11]: cross_val_score(pipeline, X_train, y_train,
                         cv=10, scoring="accuracy").mean()
```

```
Out[11]: 0.95059307474279231
```

The validation accuracy is still high, but lower than the training accuracy. This makes sense because it is always harder to predict for future data than for current data.

Scikit-Learn can also calculate the precision and recall of a class c, but we need to manually convert the label to a binary label that is 1 (or `True`) if the observation is in class c and 0 (or `False`) otherwise. For example, the following code calculates the validation *recall* for red wines:

```
In [12]: is_red_train = (y_train == "red")

         cross_val_score(pipeline, X_train, is_red_train,
                         cv=10, scoring="recall").mean()
```

```
Out[12]: 0.88495676100628928
```

To calculate the validation *precision* for red wines, we just have to change the scoring method:

```
In [13]: cross_val_score(pipeline, X_train, is_red_train,
                         cv=10, scoring="precision").mean()
```

```
Out[13]: 0.91373083576211089
```

4 F1 Score: Combining Precision and Recall

We have replaced accuracy by two numbers: precision and recall. We can combine the precision and recall into a single number, called the **F1 score**.

The F1 score is defined to be the **harmonic mean** of the precision and the recall. That is,

$$\frac{1}{\text{F1 score}} = \frac{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}{2},$$

or equivalently,

$$\text{F1 score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

The harmonic mean of two numbers is always between the two numbers, but in general will be closer to the smaller number. For example, if precision is 90% and recall is 10%, then the harmonic mean is

$$\text{F1 score} = \frac{2 \cdot 0.9 \cdot 0.1}{0.9 + 0.1} = 18\%.$$

This is a desirable property of F1 scores because we want to encourage models to have both high precision *and* high recall. It is not sufficient for one of these to be high if the other is very low. In other words, we do not want to allow a high precision to cancel out a low recall, or vice versa.

The F1 score for red wines is:

```
In [14]: 2 * precision * recall / (precision + recall)
```

```
Out[14]: 0.91934467548834276
```

We could have also asked Scikit-Learn calculate this for us. If we know the actual and predicted labels, we can use the `f1_score` function, which works similarly to `precision_score` and `recall_score` from above:

```
In [15]: from sklearn.metrics import f1_score

         f1_score(y_train, y_train_pred, pos_label="red")
```

```
Out[15]: 0.91934467548834276
```

We can also have `cross_val_score` calculate and return the F1 scores on each held-out subsample:

```
In [16]: cross_val_score(pipeline, X_train, is_red_train,
                         cv=10, scoring="f1")
```

```
Out[16]: array([ 0.87043189,  0.88888889,  0.89361702,  0.87707641,  0.90666667,
                 0.93416928,  0.89589905,  0.88679245,  0.93710692,  0.88888889])
```

```
In [17]: cross_val_score(pipeline, X_train, is_red_train,
                         cv=10, scoring="f1").mean()
```

```
Out[17]: 0.89795374750626888
```

5 Exercises

Exercises 1-3 ask you to use the Titanic data set (`https://raw.githubusercontent.com/dlsun/data-science-bo`) to train various classifiers.

Exercise 1. Train a 5-nearest neighbors model to predict whether or not a passenger on a Titanic survived, using their age, sex, and class as features. Calculate the *training* accuracy, precision, and recall of this model for survivors.

```
In [18]: titanic = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/maste
         titanic.head()
```

Out[18]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	S	NaN	135.0	Montreal, PQ / Chesterville, ON
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

In [19]:
```
from sklearn.feature_extraction import DictVectorizer

vars_df = titanic[["age", "sex", "pclass", "survived"]].dropna()
features_df = vars_df[["age", "sex", "pclass"]]

features_df["pclass"] = features_df["pclass"].astype(str)

y_train = vars_df["survived"]

vec = DictVectorizer(sparse=False)
vec.fit(features_df.to_dict(orient="records"))
X_train = vec.transform(features_df.to_dict(orient="records"))

scaler = StandardScaler()
scaler.fit(X_train)
X_train_sc = scaler.transform(X_train)

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train_sc, y_train)

y_train_pred = model.predict(X_train_sc)
y_train_pred

print("Accuracy: ", accuracy_score(y_train, y_train_pred),
" Precision: ", precision_score(y_train, y_train_pred, pos_label=1),
" Recall: ", recall_score(y_train, y_train_pred, pos_label=1))
```

```
Accuracy:  0.84034416826  Precision:  0.849462365591  Recall:  0.740046838407
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
```

Exercise 2. Estimate the *test* accuracy, precision, and recall of your model from Exercise 1 for survivors.

```
In [20]: from sklearn.pipeline import Pipeline
         from sklearn.model_selection import cross_val_score

         #Using scaler, model, X_train, y_train from above

         pipeline = Pipeline([
             ("scaler", scaler),
             ("model", model)
         ])


         print("Accuracy: ", cross_val_score(pipeline, X_train, y_train,
                         cv=10, scoring="accuracy").mean())

         did_survive = (y_train == 1)

         print("Recall: ", cross_val_score(pipeline, X_train, did_survive,
                         cv=10, scoring="recall").mean())

         print("Precision: ", cross_val_score(pipeline, X_train, did_survive,
                         cv=10, scoring="precision").mean())
```

```
Accuracy:  0.779581155091
Recall:  0.681063122924
Precision:  0.755287826432
```

```
In [21]: y_train

Out[21]: 0       1
         1       1
                ..
         1307    0
         1308    0
         Name: survived, Length: 1046, dtype: int64
```

Exercise 3. You want to build a k-nearest neighbors model to predict where a Titanic passenger embarked, using their age, sex, and class.

- What value of k optimizes overall accuracy?
- What value of k optimizes the F1 score for Southampton?

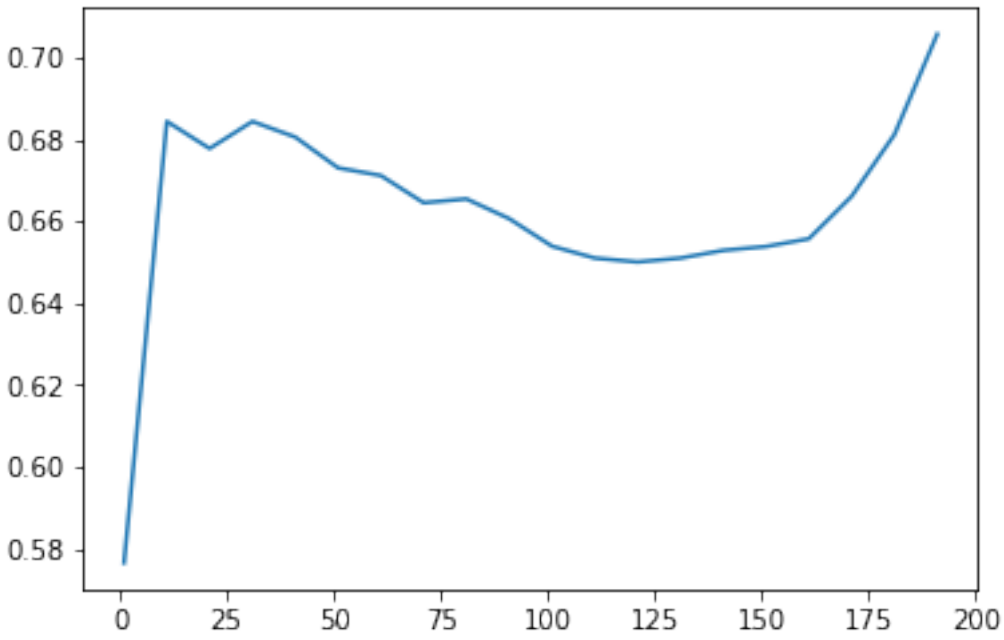
Does the same value of k optimize accuracy and the F1 score?

```
In [22]: from sklearn.pipeline import Pipeline
         from sklearn.model_selection import cross_val_score
         from sklearn.feature_extraction import DictVectorizer

         vars_df = titanic[["age", "sex", "pclass", "embarked"]].dropna()
         features_df = vars_df[["age", "sex", "pclass"]]

         features_df["pclass"] = features_df["pclass"].astype(str)

         y_train = vars_df["embarked"]


         def optimize_accuracy(k):
             model = KNeighborsClassifier(n_neighbors=k)
             pipeline = Pipeline([
                 ("vec", vec),
                 ("scaler", scaler),
                 ("model", model)
             ])
             return cross_val_score(pipeline, features_df.to_dict(orient="records"),
                                    vars_df["embarked"], cv=10, scoring="accuracy").mean()
```

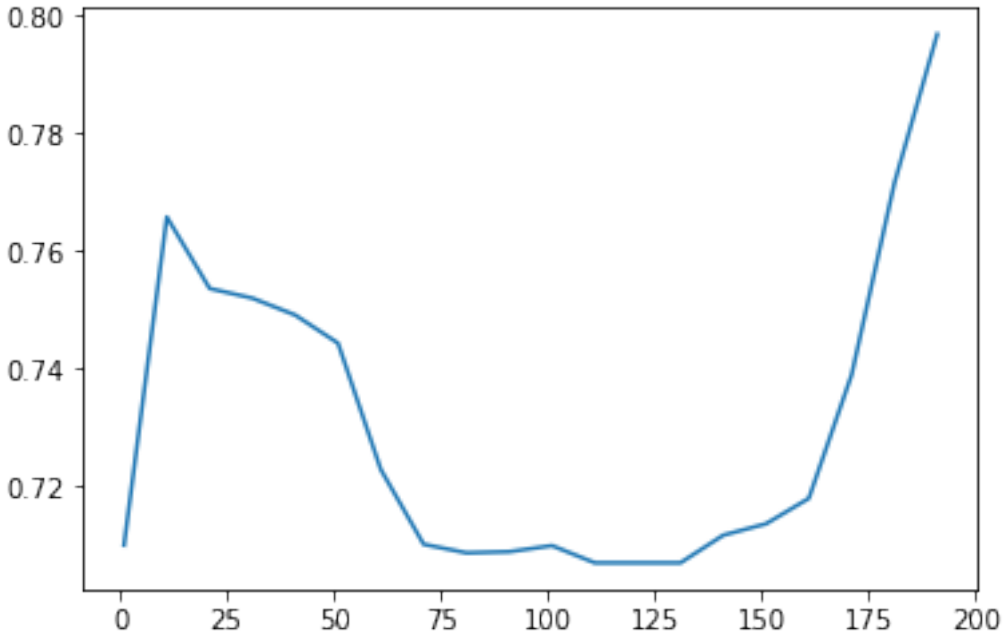
```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
```

```
In [23]: ks = pd.Series(range(1, 200, 10))
         ks.index = range(1, 200, 10)
         ks.apply(optimize_accuracy).plot.line()
```

```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7f92fcd44400>
```




```
In [24]: def optimize_f1(k):
             model = KNeighborsClassifier(n_neighbors=k)
             pipeline = Pipeline([
                 ("vec", vec),
                 ("scaler", scaler),
                 ("model", model)
             ])
             return cross_val_score(pipeline, features_df.to_dict(orient="records"),
                                    vars_df["embarked"]=="S", cv=10, scoring="f1").mean()

In [25]: ks = pd.Series(range(1,200,10))
         ks.index = range(1,200,10)
         ks.apply(optimize_f1).plot.line()

/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1143: UndefinedMetricW
  'precision', 'predicted', average, warn_for)


Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f92fd1fa198>
```


0.80
0.78
0.76
0.74
0.72
0
25
50
75
100
125
150
175
200