

In [1]:

```
%matplotlib inline
from sklearn import datasets
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

Contextualização

O **Câncer de Mama** é um dos tipos de câncer mais comuns entre as mulheres no Brasil e no Mundo. Segundo o INCA (Instituto Nacional de Câncer), no Brasil, o Câncer de Mama representa cerca de 29,5% dos novos casos de câncer em mulheres e nesse grupo representa 16% dos óbitos por tumores. Quanto antes for realizado seu diagnóstico, a compreensão e dimensão do problema, será possível elaborar a melhor forma de contra-ataque.

A diferenciação dos tumores malignos e benignos se dá pela aparência e estrutura das células atacadas pelo tumor. Os do tipo benigno não possuem a capacidade de provocar metástase, diferente dos malignos que são agressivos e têm a capacidade de infiltrar outros órgãos.

Conhecendo os dados

Importando os dados:

In [2]:

```
df = datasets.load_breast_cancer()
X_names = df.feature_names
```

Essa base possui 10 features diferenciadas em 3 categorias: Mean, Standard Error e "Worst"(média dos 3 maiores valores), totalizando 30 variáveis.

- radius
- texture
- perimeter
- area
- smoothness
- compactness
- concavity
- concave points
- symmetry
- fractal dimension

O dataset contém 569 exemplos, sendo que não há valores faltantes. Ela foi construída por Dr. William H. Wolberg, W. Nick Street e Olvi L. Mangasarian, da universidade de Wisconsin, em 1995.

In [3]:

```
X = pd.DataFrame(df.data, columns=df.feature_names)
y = pd.DataFrame(df.target, columns=["target"])
```

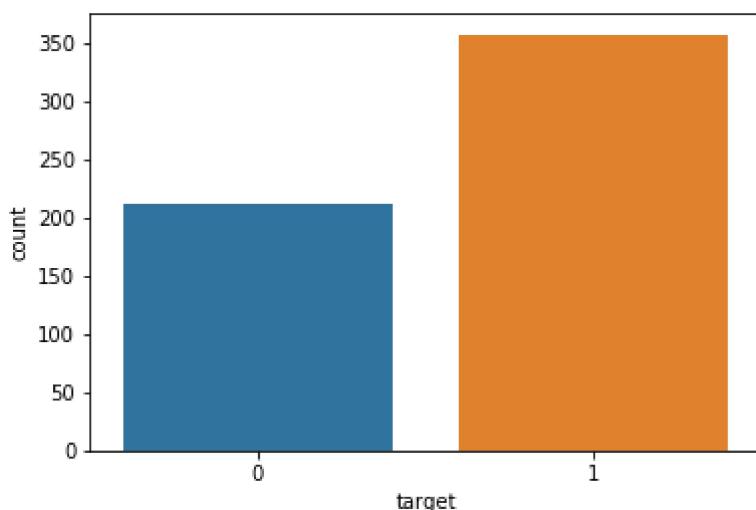
A variável target representa o objetivo da nosso problema de classificação, sendo 0 maligno e 1 benigno.

In [10]:

```
sns.countplot(y.target)
y.target.value_counts()
```

Out[10]:

```
1    357
0    212
Name: target, dtype: int64
```



Por motivos de clareza vamos inverter os valores da classificação, ficando 0 para benignos e 1 para malignos.

In [4]:

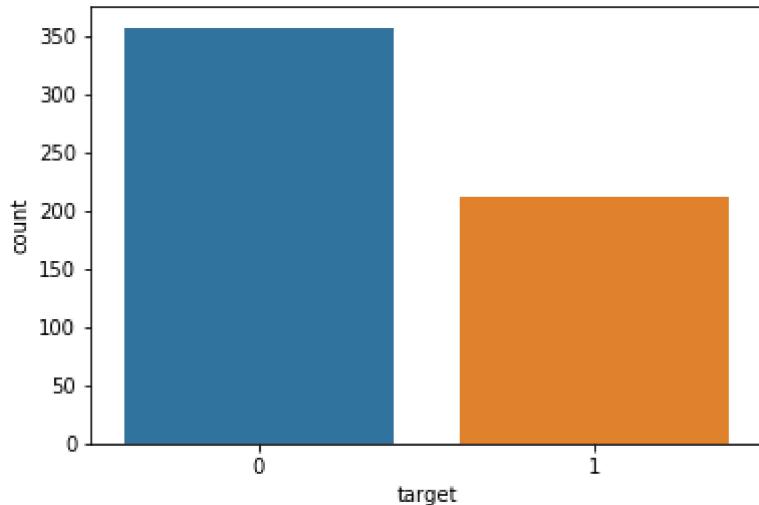
```
invert = y.target == 1
y.loc[(invert == True), "target"] = 0
y.loc[(invert == False), "target"] = 1
```

In [8]:

```
sns.countplot(y.target)
y.target.value_counts()
```

Out[8]:

```
0    357
1    212
Name: target, dtype: int64
```



In [5]:

```
df = pd.concat([X, y], axis=1)
```

Nas 5 primeiras linhas conseguimos ver que os valores não estão normalizados, e utilizar modelos que utilizam distâncias como o KNN poderá dar mais importância para features com uma escala maior.

In [355]:

```
df.iloc[0:5, 0:17]
```

Out[355]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1

In [356]:

df.iloc[0:5, 17:32]

Out[356]:

	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	w compactr
0	0.01587	0.03003	0.006193	25.38	17.33	184.60	2019.0	0.1622	0.6
1	0.01340	0.01389	0.003532	24.99	23.41	158.80	1956.0	0.1238	0.1
2	0.02058	0.02250	0.004571	23.57	25.53	152.50	1709.0	0.1444	0.4
3	0.01867	0.05963	0.009208	14.91	26.50	98.87	567.7	0.2098	0.8
4	0.01885	0.01756	0.005115	22.54	16.67	152.20	1575.0	0.1374	0.2

Outliers

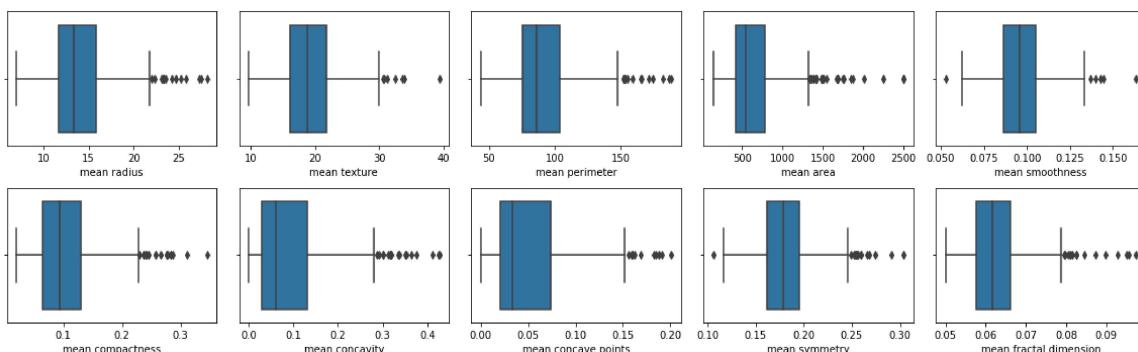
Em uma análise dos outliers das variáveis Mean, vemos que eles se concentram nos maiores valores de cada feature.

In [12]:

```
plt.figure(figsize=(16, 5))

cont = 1
for i in X_names:
    plt.subplot(2, 5, cont)
    sns.boxplot(x=i, data=df)
    cont += 1
    if cont > 10:
        break

plt.tight_layout()
```



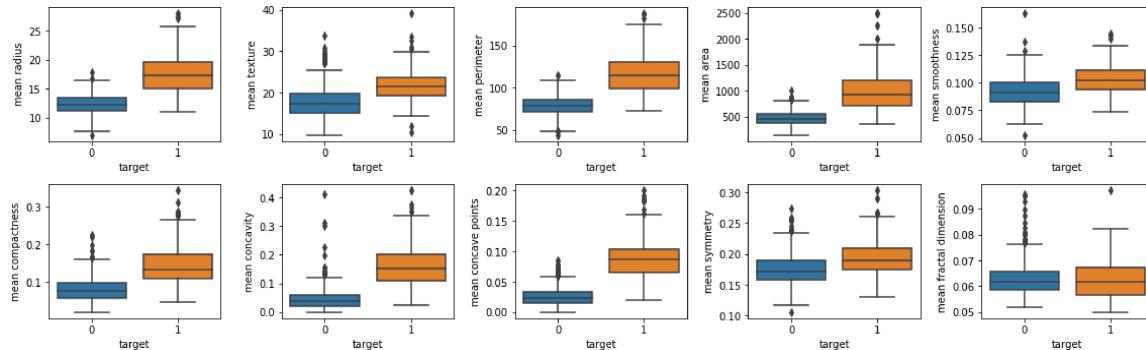
No plot abaixo fica evidente que os maiores valores em casos de variáveis que representam valores métricos(radius, perimeter, area...) significam que há uma maior classificação de tumores malignos, fazendo com que sejam muito significativos para nossa modelagem.

In [13]:

```
plt.figure(figsize=(16, 5))

cont = 1
for i in X_names:
    plt.subplot(2, 5, cont)
    sns.boxplot(x="target", y=i, data=df)
    cont += 1
    if cont > 10:
        break

plt.tight_layout()
```



Feature Scalling

Algumas das técnicas implementadas no `sklearn` para fazer o escalonamento das features:

- **Standard Scaler:** pressupõe que os dados possuem uma distribuição normal. A média e o desvio padrão é calculado e os dados são escalonados a partir desses valores fazendo com que a média dos dados escalonados seja 0.
- **Normalizer:** faz com que a sua normal(L1 ou L2) se iguale a 0.
- **Min-Max Scaler:** transforma os dados para que fique em um range de 0 até 1, utilizando os valores mínimos e máximos da feature.
- **The Robust Scaler:** utiliza a distância interquartil para definir os novos valores. Essa técnica não possui grande influência dos outliers já que não é utilizado o valor de média para o cálculo, que sofre influência direta dos outliers.

Utilizaremos o Min-Max Scaler já que os outliers são significativos para o entendimento do nosso problema.

In [5]:

```
X_scaled = MinMaxScaler().fit_transform(X)
```

In [6]:

```
df_scaled = pd.DataFrame(X_scaled, columns=X_names)
df_scaled["target"] = y.target
```

Visualização dos Dados

Vendo a correlação entre as variáveis notamos que há uma grande correlação entre as variáveis métricas(radius, perimeter, area...) entre todos as medidas Mean, Standard Error e Worst.

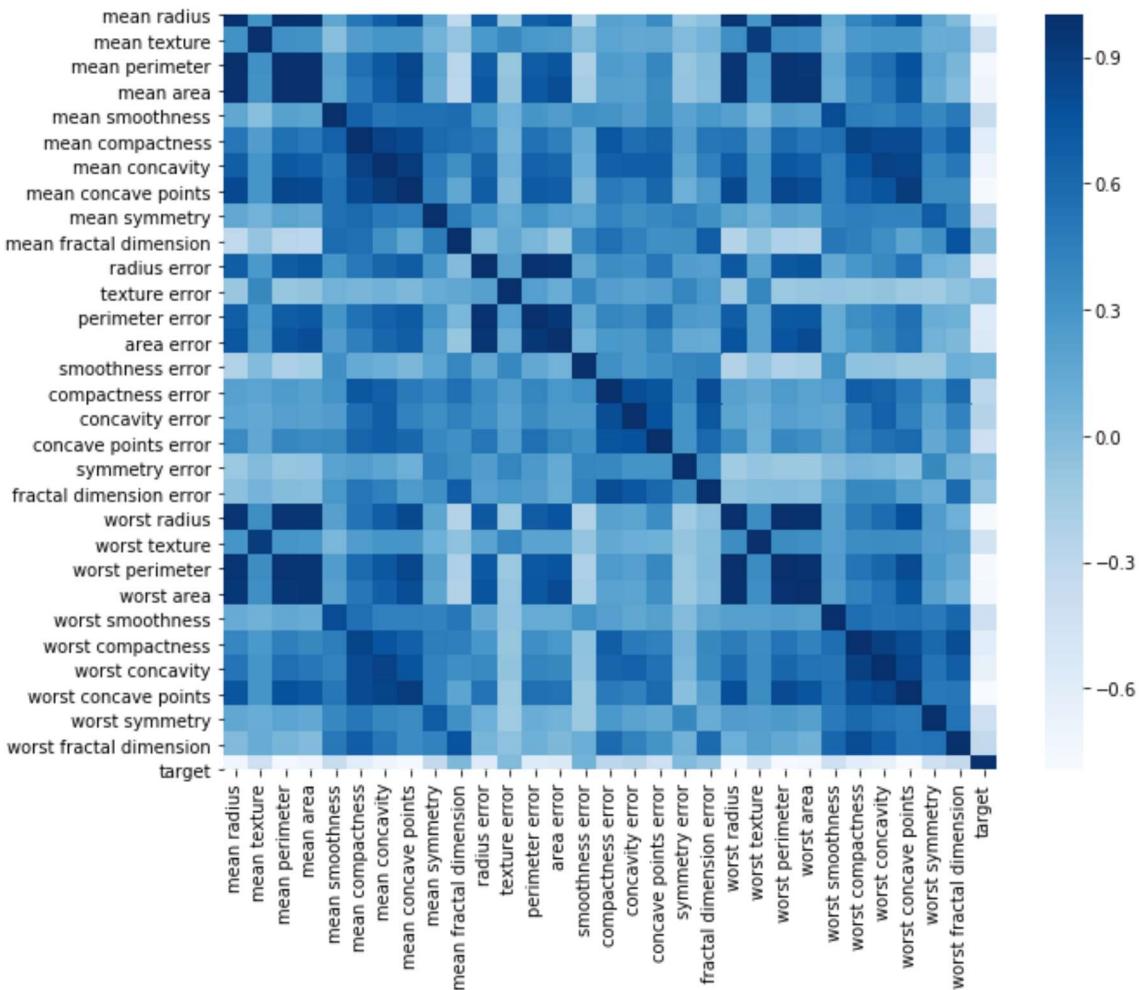
A variável que possui menor correlação com as demais features é a fractal dimension, juntas com algumas outras variáveis da classe de Standard Error.

In [359]:

```
plt.figure(figsize=(10, 8))
sns.heatmap(df_scaled.corr(), cmap="Blues")
```

Out[359]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2b1483fa888>
```



Através do distplot vemos quais variáveis separam melhor as classes. Mais uma vez as variáveis que demonstram medidas de métricas mantém uma grande semelhança entre como suas distribuições se comportam em relação às classes.

Variáveis como smoothness, texture, simetry e fractal dimension(mean e worst) não demonstram ter uma diferenciação entre as classes por si só junto com as variáveis de Standard Error.

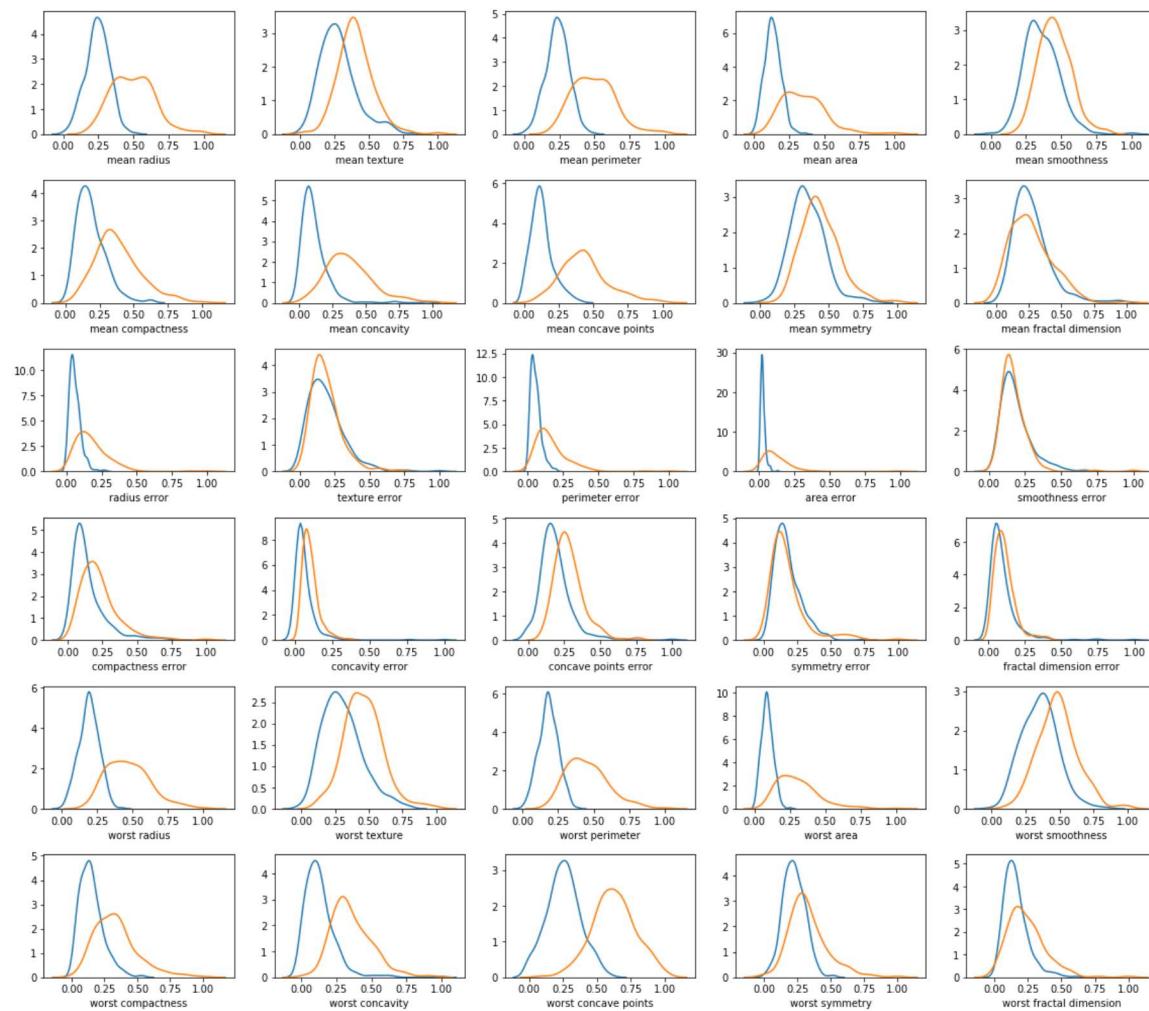
In [372]:

```
maligno = df_scaled.target == 1
benigno = df_scaled.target == 0

plt.figure(figsize=(16, 14))

cont = 1
for i in X_names:
    plt.subplot(6, 5, cont)
    sns.distplot(df_scaled.loc[benigno, i], hist=False)
    sns.distplot(df_scaled.loc[maligno, i], hist=False)

    cont += 1
plt.tight_layout()
```



A partir variáveis Mean veremos como os valores plotados em par se comportam com as classes.

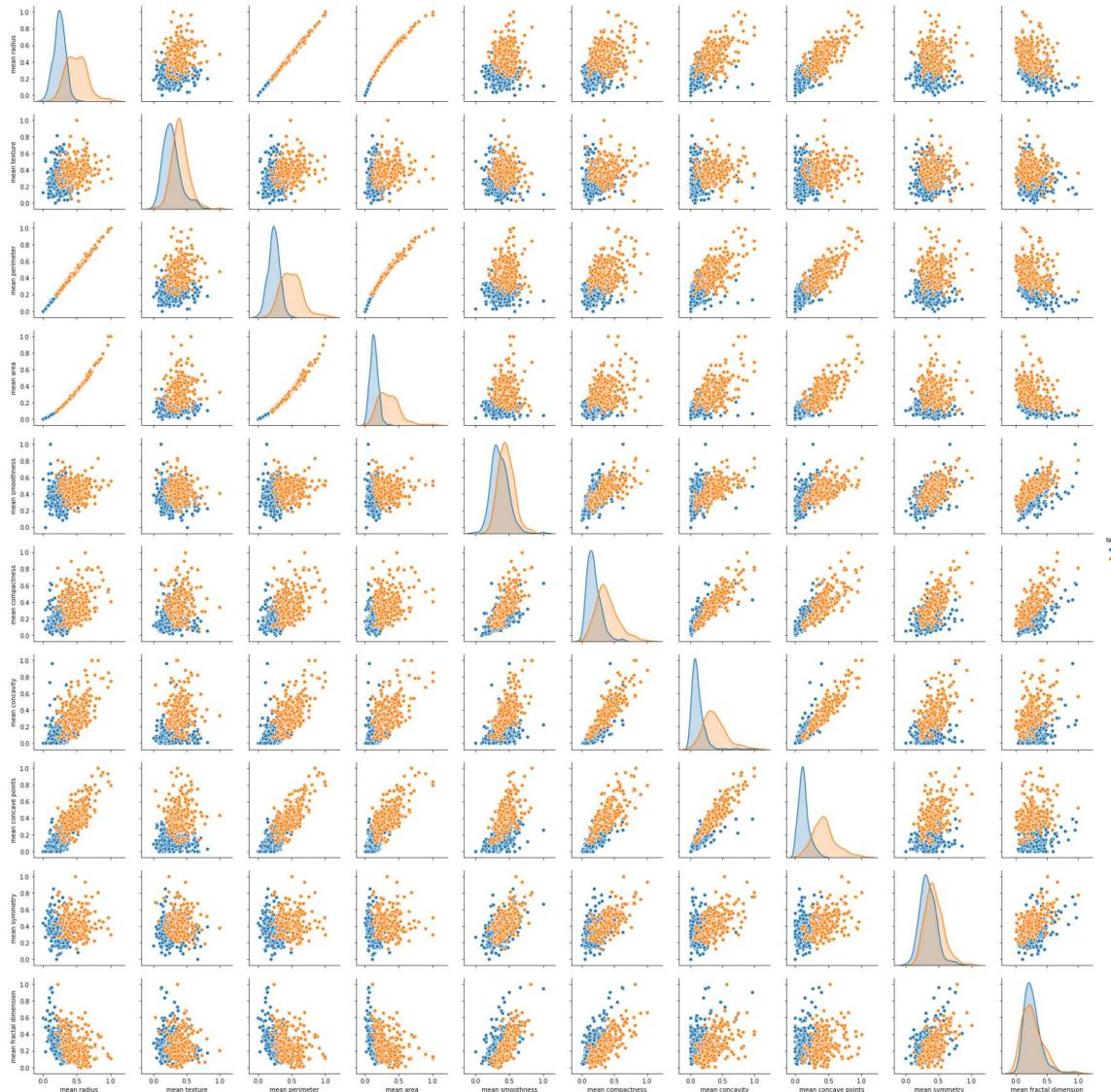
A classe dos tumores Benignos ficam com os menores valores entre as medidas como mostrado dos distplots, sendo muito fácil imaginar que modelos lineares terão bom desempenhos com esses valores. As variáveis como smoothness, texture e simetry não possuem um padrão de comportamento tão claro quando somente essas variáveis em par são levadas em consideração.

In [374]:

```
sns.pairplot(df_scaled, vars=[ "mean radius", "mean texture", "mean perimeter", "mean area", "mean smoothness", "mean compactness", "mean concavity", "mean concave points", "mean symmetry", "mean fractal dimension"] ,hue = "target")
```

Out[374]:

<seaborn.axisgrid.PairGrid at 0x2b1314308c8>



Visualização com o PCA e Feature Extraction

O PCA é uma técnica de Machine Learning não supervisionado que é utilizado para extrair features do dataset através de componentes principais. As componentes são definidas a partir da variância entre as features, e as mais significativas são as que explicam a maior variância possível.

Para tentar ter uma visão geral de toda a base de dados Utilizaremos o PCA para extrair as componentes que melhor representam a variância do dataset.

In [20]:

```
model_PCA = PCA()
model_PCA.fit(X_scaled)
```

Out[20]:

```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

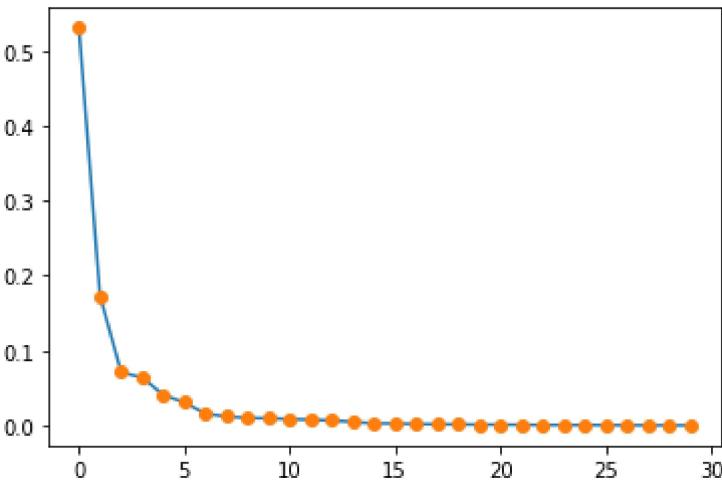
Vemos o quanto da variância é explicado pelas componentes no plot abaixo. Para se utilizar a menor dimensionalidade possível utilizamos o método do cotovelo, onde mostra o maior ganho com a menor quantidade de componentes. No nosso caso será utilizado 2 componentes.

In [21]:

```
plt.plot(model_PCA.explained_variance_ratio_)
plt.plot(model_PCA.explained_variance_ratio_, 'o')
```

Out[21]:

```
[<matplotlib.lines.Line2D at 0x152e7dfdb08>]
```



In [126]:

```
porc_variancia = model_PCA.explained_variance_ratio_[0] + model_PCA.explained_variance_ratio_[1]
print("Porcentagem da variâcia explicada pelas 2 principais componentes: ", porc_variancia)
```

Porcentagem da variâcia explicada pelas 2 principais componentes: 0.70381
17901347674

Conseguimos ver como as classes estão se comportando com essas componentes no scatterplot abaixo.

In [22]:

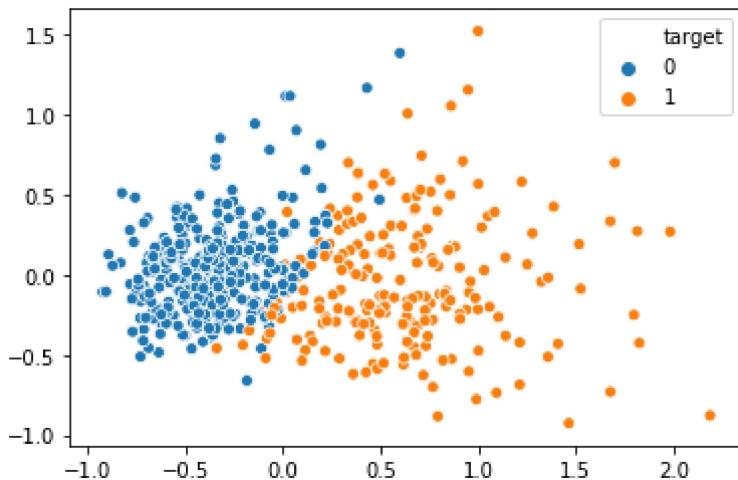
```
model_pca = PCA(2)
pc = model_pca.fit_transform(X_scaled)
```

In [23]:

```
sns.scatterplot(x = pc[:, 0], y = pc[:, 1], hue=y.target)
```

Out[23]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x152e7da6f88>
```



Como mostrados no `pairplot` acima, as variáveis tendem a ter um comportamento onde é possível visualizar uma distinção linear entre as classes, o que foi mostrado nas 2 componentes extraídas do PCA

Separando Treino e Teste

Para validar os modelos que serão criados iremos separar o dataset em 2 bases, Treino e Teste. A base de teste terá somente 20% do tamanho total e ficará com 114 exemplos e a base de treino ficará com 455 exemplos.

In [7]:

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state = 60)
```

In [92]:

```
print("Tamanho X_train:", X_train.shape)
print("Tamanho X_test:", X_test.shape)
```

```
Tamanho X_train: (455, 30)
Tamanho X_test: (114, 30)
```

Feature Selection com Cross Validation

A seleção das features é um dos principais papéis a serem realizados para conseguir uma boa performance dos modelos. Ter uma grande dimensionalidade significa que precisamos de muitos dados para fazer com que nossos modelos consigam aprender e conseguir uma boa generalização. Assim não conseguimos apresentar todas as possibilidades de combinações dos valores dos parâmetros possíveis para nosso modelo, fazendo com que eles fiquem sujeito ao overfitting e não generalizando para dados nunca vistos no treinamento.

Alguns dos métodos que podemos utilizar:

- **Árvore de decisão:** podemos utilizar uma árvore de decisão e treiná-la com a base de dados. A partir do poder de interpretabilidade desse modelo podemos escolher as features que foram utilizadas para a classificação da base de treino.
- **Regressão Logística com Normalização L1:** como a regressão logística cria coeficientes que representam a importância das features para o modelo, conseguimos utilizar a normalização L1 para diminuir os valores dos coeficientes e eliminar as features que tiveram os coeficientes zerados.
- **Seleção Univariada:** através de métricas como o p_Values é selecionado as k features que tenham um score maior(consequentemente um p_value menor).
- **Eliminação de Feature Recursivamente:** o RFE é uma técnica que atribui pesos para as features e utiliza um método de classificação para fazer a validação do modelo. A cada passo do processo os pesos são atualizados de acordo com sua importância, assim os que tiverem menor peso serão eliminados.

Utilizaremos a seleção univariada com o método `SelectKBest` do `sklearn` e escolheremos o melhor valor de K com um Cross Validation utilizando o modelo default de Regressão Logística do `sklearn`.

Função que utilizaremos par extrair os valores de acurácia:

In [111]:

```
def cv_features_logistic_regression(k_features, X_train, y_train, X_test, y_test):  
  
    kbest = SelectKBest(k=k_features)  
    features = kbest.fit(X_train, y_train.target)  
  
    X_train_selected = features.transform(X_train)  
    X_test_selected = features.transform(X_test)  
  
    model_lr = LogisticRegression(solver='lbfgs').fit(X_train_selected, y_train.target)  
    predicao_lr = model_lr.predict(X_test_selected)  
  
    print("k= ", k_features, ", acurácia: ", metrics.accuracy_score(y_test.target, predicao_lr))  
  
    return metrics.accuracy_score(y_test, predicao_lr)
```

Execução:

In [112]:

```
k_acuracia = []

for i in range(1, 31):
    k_acuracia.append([i, cv_features_logistic_regression(i, X_train, y_train, X_test,
y_test)])
```

k= 1 , acurácia: 0.8859649122807017
k= 2 , acurácia: 0.9210526315789473
k= 3 , acurácia: 0.9473684210526315
k= 4 , acurácia: 0.956140350877193
k= 5 , acurácia: 0.956140350877193
k= 6 , acurácia: 0.956140350877193
k= 7 , acurácia: 0.956140350877193
k= 8 , acurácia: 0.956140350877193
k= 9 , acurácia: 0.956140350877193
k= 10 , acurácia: 0.9473684210526315
k= 11 , acurácia: 0.9473684210526315
k= 12 , acurácia: 0.9473684210526315
k= 13 , acurácia: 0.9473684210526315
k= 14 , acurácia: 0.9473684210526315
k= 15 , acurácia: 0.9473684210526315
k= 16 , acurácia: 0.9649122807017544
k= 17 , acurácia: 0.9736842105263158
k= 18 , acurácia: 0.9824561403508771
k= 19 , acurácia: 0.9824561403508771
k= 20 , acurácia: 0.9824561403508771
k= 21 , acurácia: 0.9736842105263158
k= 22 , acurácia: 0.9736842105263158
k= 23 , acurácia: 0.9649122807017544
k= 24 , acurácia: 0.9824561403508771
k= 25 , acurácia: 0.9736842105263158
k= 26 , acurácia: 0.9736842105263158
k= 27 , acurácia: 0.9649122807017544
k= 28 , acurácia: 0.9649122807017544
k= 29 , acurácia: 0.9649122807017544
k= 30 , acurácia: 0.9649122807017544

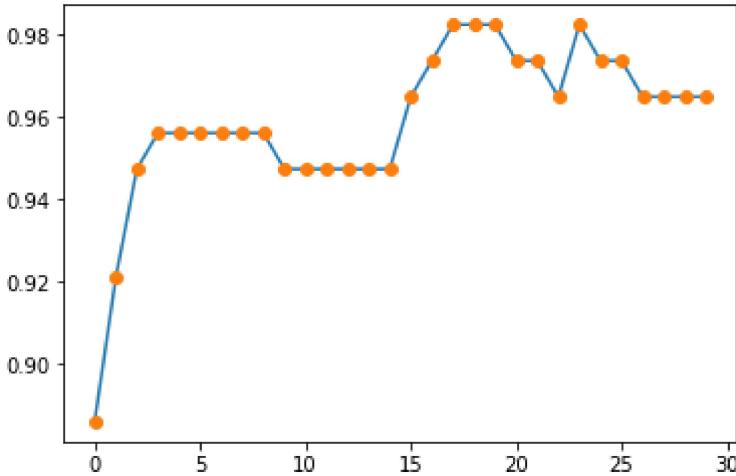
Através do plot das acuráciais abaixo vemos que com 18 features obtivemos uma acurácia de mais de 98%.

In [113]:

```
plt.plot(np.array(k_acuracia)[:,1])
plt.plot(np.array(k_acuracia)[:,1], 'o')
```

Out[113]:

```
[<matplotlib.lines.Line2D at 0x1c0eb9e4dc8>]
```



In [552]:

```
kbest = SelectKBest(k=18)
features = kbest.fit(X_train, y_train)

X_train_selected = features.transform(X_train)
X_test_selected = features.transform(X_test)

scores = pd.DataFrame()
scores["nomes"] = X.columns
scores["scores"] = features.scores_

valoresp = pd.DataFrame()
valoresp["features"] = X.columns
valoresp["p"] = features.pvalues_
```

E aqui temos as variáveis que foram escolhidas, junto com seus scores e p_value.

In [554]:

```
pd.concat([scores.sort_values(by = "scores", ascending=False).head(18), valoresp.sort_values(by = "p").head(18)], axis=1)
```

Out[554]:

	nomes	scores	features	p
27	worst concave points	734.793089	worst concave points	7.158229e-97
22	worst perimeter	707.310499	worst perimeter	1.447790e-94
20	worst radius	678.225165	worst radius	4.586536e-92
7	mean concave points	643.519259	mean concave points	5.386024e-89
2	mean perimeter	552.949139	mean perimeter	1.681093e-80
0	mean radius	516.911289	mean radius	6.620920e-77
23	worst area	505.656596	worst area	9.359715e-76
3	mean area	445.028076	mean area	2.577587e-69
6	mean concavity	394.942227	mean concavity	1.174258e-63
26	worst concavity	326.374734	worst concavity	2.438103e-55
5	mean compactness	234.020013	mean compactness	6.892190e-43
25	worst compactness	233.792519	worst compactness	7.431438e-43
10	radius error	206.945463	radius error	6.472655e-39
12	perimeter error	189.173040	perimeter error	3.234747e-36
13	area error	179.517047	area error	1.018919e-34
21	worst texture	118.102012	worst texture	1.327040e-24
1	mean texture	92.210010	mean texture	5.364287e-20
24	worst smoothness	92.002122	worst smoothness	5.853614e-20

In [575]:

```
nomes_variaveis = [i[0] for i in np.array(scores.sort_values(by = "scores", ascending=False).head(18).sort_index()[["nomes"]])]
X_train_selected = pd.DataFrame(X_train_selected, columns=nomes_variaveis)
X_test_selected = pd.DataFrame(X_test_selected, columns=nomes_variaveis)
```

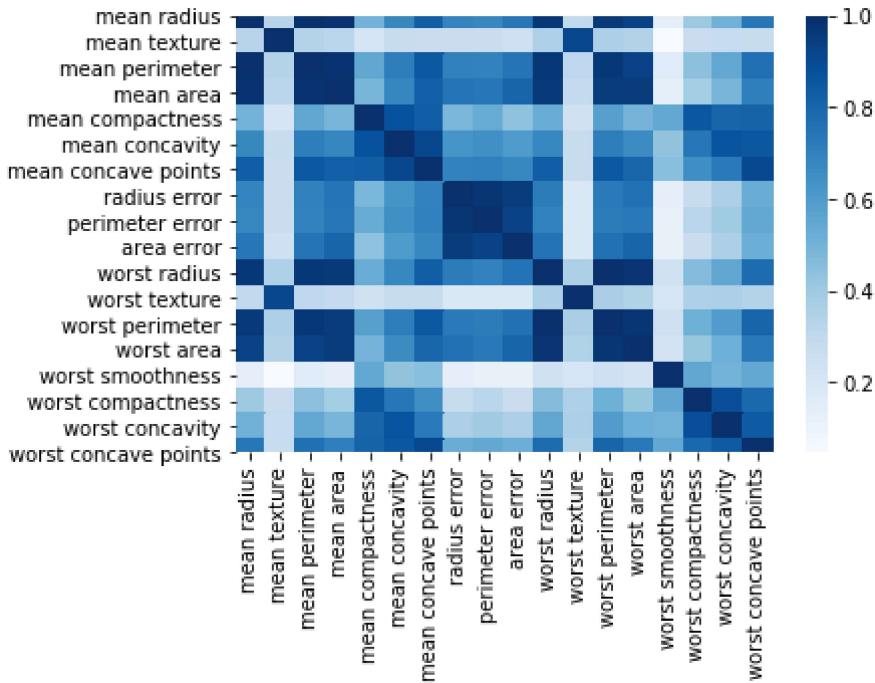
No heatmap abaixo temos a correlação entre as 18 variáveis escolhidas.

In [581]:

```
sns.heatmap(X_train_selected.corr(), cmap="Blues")
```

Out[581]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2b133646fc8>
```



Como as variáveis métricas possuem uma correlação muito grande entre si, manteremos somente a variável Area para cada categoria Mean, Standard Error e Worst.

In [588]:

```
X_train_selected.drop(["mean radius", "mean perimeter", "radius error", "perimeter erro  
r", "worst radius", "worst perimeter"], axis=1, inplace=True)  
X_test_selected.drop(["mean radius", "mean perimeter", "radius error", "perimeter erro  
r", "worst radius", "worst perimeter"], axis=1, inplace=True)
```

Com isso ficamos com 13 features selecionadas para a modelagem.

In [613]:

```
print(X_train_selected.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 455 entries, 0 to 454
Data columns (total 13 columns):
mean texture      455 non-null float64
mean area         455 non-null float64
mean compactness   455 non-null float64
mean concavity     455 non-null float64
mean concave points 455 non-null float64
area error        455 non-null float64
worst texture      455 non-null float64
worst area         455 non-null float64
worst smoothness    455 non-null float64
worst compactness   455 non-null float64
worst concavity     455 non-null float64
worst concave points 455 non-null float64
target             364 non-null float64
dtypes: float64(13)
memory usage: 46.3 KB
None
```

In [594]:

```
X_train_selected.head(5)
```

Out[594]:

	mean texture	mean area	mean compactness	mean concavity	mean concave points	area error	worst texture	worst area	sme
0	0.083192	0.149099	0.118275	0.055295	0.119384	0.019739	0.130597	0.093811	
1	0.274941	0.130477	0.157015	0.058341	0.146173	0.032944	0.237207	0.076190	
2	0.232330	0.122375	0.461996	0.388707	0.368539	0.033467	0.382729	0.099907	
3	0.168752	0.227869	0.094411	0.080037	0.126292	0.025996	0.148188	0.141319	
4	0.212039	0.198388	0.121373	0.082802	0.146322	0.048446	0.254797	0.138812	

Modelagem com GridSearch

Com o método `GridSearchCV` do `sklearn`, podemos testar quais serão os melhores valores dos hiperparâmetros dos modelos para o nosso dataset.

In [8]:

```
X_train = pd.DataFrame(X_train, columns=X.columns)[["mean texture", "mean area", "mean compactness", "mean concavity", "mean concave points", "area error", "worst texture", "worst area", "worst smoothness", "worst compactness", "worst concavity", "worst concave points"]]
X_test = pd.DataFrame(X_test, columns=X.columns)[["mean texture", "mean area", "mean compactness", "mean concavity", "mean concave points", "area error", "worst texture", "worst area", "worst smoothness", "worst compactness", "worst concavity", "worst concave points"]]
```

Função para utilizar o Grid Search.

In [9]:

```
def modelo_gs(modelo, hiperparametros):

    gs = GridSearchCV(modelo, hiperparametros, cv=3, verbose=0)

    gs.fit(X_train, y_train.target)

    print("Melhores hiperparâmetros: ", gs.best_params_)

    return gs.predict(X_test)
```

Random Forest

Random Forest é um algoritmo de Machine Learning supervisionado que utiliza o conceito Ensamble para criar diversas Árvores de Decisões aleatórias, cada um sendo treinado com features distintas. A classificação se dá a partir do voto majoritário dentre todos os classificadores. A ideia de treinar vários modelos 'fracos' é para garantir que cada um aprenda com informações diferentes e que ao juntar todo o conhecimento aprendido se possa chegar a uma conclusão mais assertiva.

Ao juntar diversos modelos mais simples para fazer a classificação conseguimos evitar o overfitting mas perdemos o poder de interpretabilidade.

Na biblioteca do scikit-learn existe uma implementação com a classe `RandomForestClassifier()` e seus principais parâmetros são:

- **`n_estimators`**: quantidade de modelos que serão utilizados.
- **`criterion`**: critério de divisão dos nós. Existem 2 opções:
 - Gini: mede a impureza do nó
 - Entropy: mede o ganho de informação
- **`max_depth`**: a profundidade máxima de cada árvore.
- **`min_samples_split`**: quantidade mínima de registros por nó para se permitir a divisão do nó.
- **`min_samples_leaf`**: quantidade mínima de exemplos por folha.
- **`max_features`**: quantidade máxima de features utilizada por cada modelo.
- **`bootstrap`**: utilizar o Bootstrap para a criação do dataset de treinamento.
- **`class_weight`**: opção para balancear ou definir os pesos de cada classe.

Para simplificar nosso modelo vamos determinar somente os parametros `n_estimators` e `max_depth`.

In [129]:

```
hiperparametros = { "max_depth":np.arange(2, 21), "n_estimators":np.arange(3, 20)}
predicao_gs = modelo_gs(RandomForestClassifier(), hiperparametros)
```

Melhores hiperparâmetros: {'max_depth': 7, 'n_estimators': 12}

In [130]:

```
print(metrics.classification_report(y_test, predicao_gs))
```

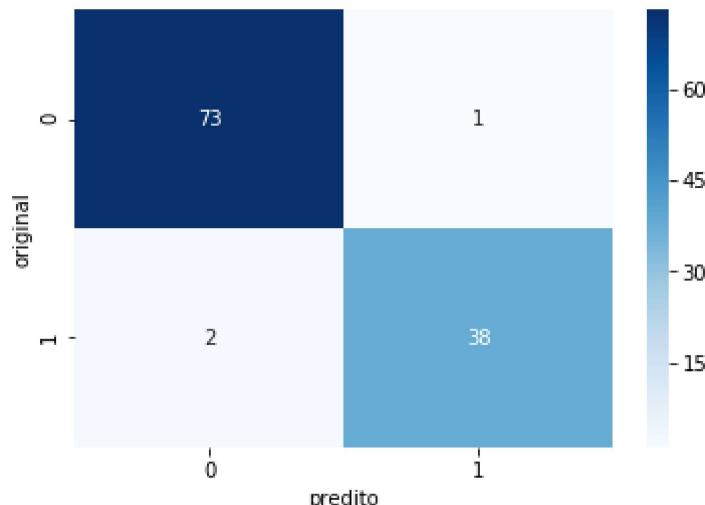
	precision	recall	f1-score	support
0	0.97	0.99	0.98	74
1	0.97	0.95	0.96	40
accuracy			0.97	114
macro avg	0.97	0.97	0.97	114
weighted avg	0.97	0.97	0.97	114

In [131]:

```
sns.heatmap(metrics.confusion_matrix(y_test, predicao_gs), cmap="Blues", annot=True)
plt.ylabel('original')
plt.xlabel('predito')
```

Out[131]:

Text(0.5, 15.0, 'predito')



Para os hiperparâmetros escolhidos foi obtido uma acurácia de 97% com somente 3 erros dentre os 114 exemplos da base de teste.

KNN

O KNN é um método de machine learning supervisionado que utiliza o conceito de distância para fazer a classificação do dataset. Existem duas implementações desse modelo no sklearn:

- o primeiro utiliza o valor k para determinar as quantidades de vizinhos que serão utilizados para a classificação.
- o segundo utiliza o raio da distância para determinar o vizinhos que serão utilizados para a classificação.

Para utilizar o KNN é necessários que os dados estejam escalonados para que todas as features tenham a mesma importância no cálculo das distâncias.

Utilizaremos o método `KNeighborsClassifier()` do sklearn sendo a primeira implementação explicada acima. Alguns dos principais hiperparâmetros:

- **n_neighbors**: a quantidade de vizinhos que será utilizado para classificação.
- **weights**: define se a distância entre os vizinhos terá um peso maior na classificação.
- **algorithm**: algoritmo utilizado para computar os vizinhos mais próximos. Utilizado para melhorar o desempenho.

In [132]:

```
hiperparametros = {"n_neighbors": [3, 5, 7, 9, 11, 13], "algorithm": ['ball_tree', 'kd_tree', 'brute'], "weights": ["uniform", "distance"]}
predicao_gs = modelo_gs(KNeighborsClassifier(), hiperparametros)
```

Melhores hiperparâmetros: {'algorithm': 'ball_tree', 'n_neighbors': 7, 'weights': 'uniform'}

In [133]:

```
print(metrics.classification_report(y_test, predicao_gs))
```

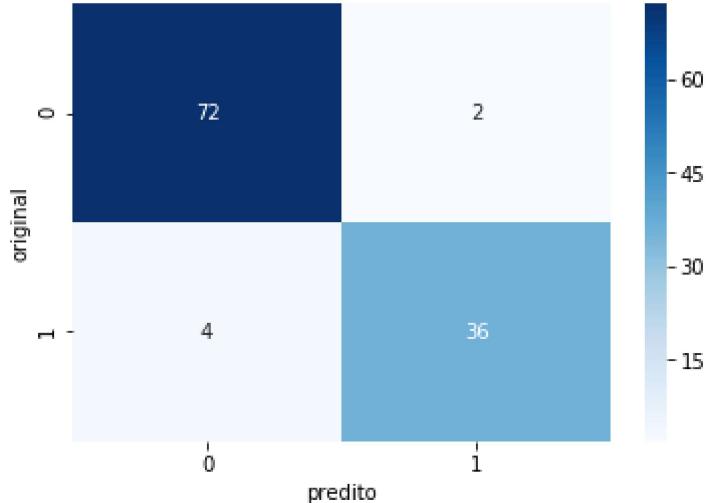
	precision	recall	f1-score	support
0	0.95	0.97	0.96	74
1	0.95	0.90	0.92	40
accuracy			0.95	114
macro avg	0.95	0.94	0.94	114
weighted avg	0.95	0.95	0.95	114

In [134]:

```
sns.heatmap(metrics.confusion_matrix(y_test, predicao_gs), cmap="Blues", annot=True)
plt.ylabel('original')
plt.xlabel('predito')
```

Out[134]:

```
Text(0.5, 15.0, 'predito')
```



Conseguimos uma acurácia de 95% com esse modelo, com 6 classificações erradas. Com esse modelo 4 pessoas que tinham o tumor maligno foram classificadas como benignos.

Logistic Regression

A Regressão Logística é um modelo de aprendizado supervisionado, chamado de modelo linear, que gera uma equação da reta que definirá um hiperplano para separar as classes. É utilizado a função sigmoid(também conhecida como logística) que retorna a probabilidade de um determinado exemplo pertencer a cada classe.

Principais Hiperparâmetros:

- **penalty**: usado para especificar a normalização utilizada na penalização. L1, L2, ElasticNet
- **tol**: tolerância para o critério de parada
- **C**: a força inversa da regularização, quanto menor o número maior será a penalização.
- **class_weight**: determina quais serão os pesos das classes. 'Balanced' já balanceia os pesos. Pode se passar um dicionário com os pesos de cada classe.
- **solver**: ['newton-cg', 'lbfgs', 'liblinear', 'sag' e 'saga'] algoritmos usados para otimização, dependendo do dataset alguns terão vantagens sobre outros
- **max_iter**: o número máximo de iterações que os algoritmos tem para converger.
- **multi_class**: ['ovr', 'multinomial', 'auto'] define o padrão de implementação sobre as classes, One-vs-rest ou Multinomial

Para simplificarmos escolheremos os valores somente do penalty e do C, os demais deixaremos como default.

In []:

```
hiperparametros = {"penalty":["l1", "l2"], "C":[0.5,1, 2, 3, 4, 5]}
predicao_gs = modelo_gs(LogisticRegression(), hiperparametros)
```

Melhores hiperparâmetros: {'C': 2, 'penalty': 'l1'}

In [34]:

```
print(metrics.classification_report(y_test, predicao_gs))
```

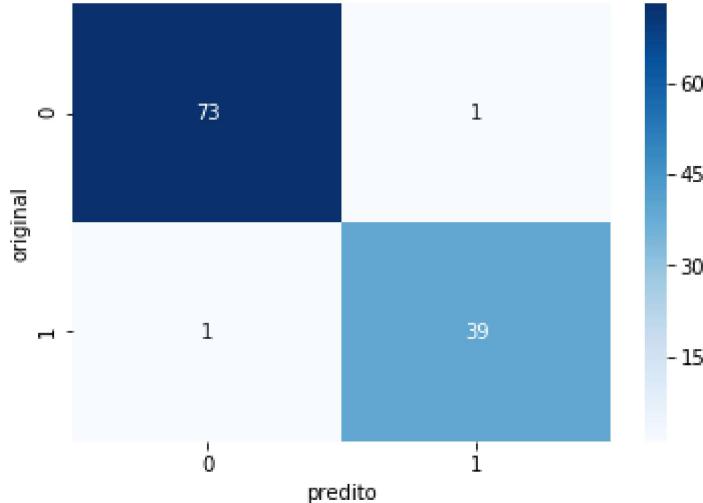
	precision	recall	f1-score	support
0	0.99	0.99	0.99	74
1	0.97	0.97	0.97	40
accuracy			0.98	114
macro avg	0.98	0.98	0.98	114
weighted avg	0.98	0.98	0.98	114

In [11]:

```
sns.heatmap(metrics.confusion_matrix(y_test, predicao_gs), cmap="Blues", annot=True)
plt.ylabel('original')
plt.xlabel('predito')
```

Out[11]:

```
Text(0.5, 15.0, 'predito')
```



Obtivemos uma acurácia de 98% com os parâmetros selecionado, além de um valor de 99 para o f1 score da classe de benignos.

SVM

Com um conceito semelhante ao da Regressão Logística, o SVM(Support Vector Machine) busca encontrar o melhor hiperplano que separe as classes. Exemplos de cada classe serão escolhidos e utilizados como vetores de suporte, com isso o hiperplano será criado entre esses exemplos com a maior margem possível entre eles.

O sklearn possui sua implementação no método `SVC` que é utilizado para a classificação. Seus principais hiperparâmetros:

- **C**: penalidade que será aplicada.
- **kernel**: tipo do kernel utilizado pelo algoritmo.
- **degree**: grau utilizado pelo kernel poly.
- **gamma**: coeficiente utilizado pelo kernel.
- **probability**: definir se será implementado a opção predição de probabilidade.

Definiremos somente os hiperparâmetros C, kernel e gamma deixando os demais como default.

In [138]:

```
hiperparametros = {"C": [0.001, 0.1, 1], "kernel": ['linear', 'rbf', 'sigmoid'], "gamma": ["scale"] }
predicao_gs = modelo_gs(SVC(), hiperparametros)
```

Melhores hiperparâmetros: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}

In [139]:

```
print(metrics.classification_report(y_test, predicao_gs))
```

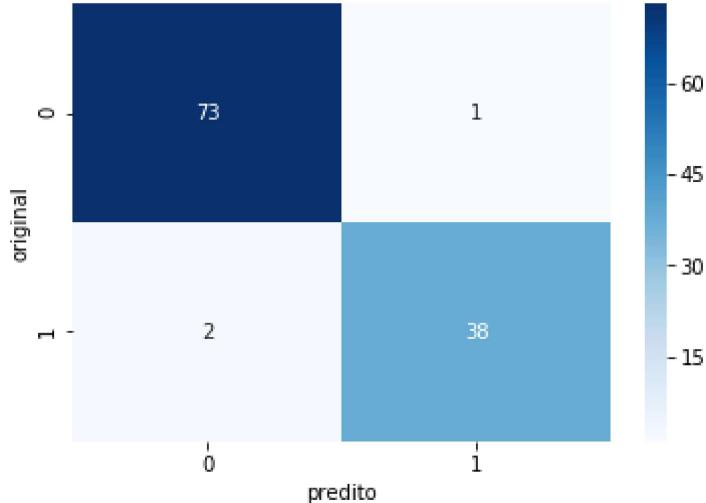
	precision	recall	f1-score	support
0	0.97	0.99	0.98	74
1	0.97	0.95	0.96	40
accuracy			0.97	114
macro avg	0.97	0.97	0.97	114
weighted avg	0.97	0.97	0.97	114

In [140]:

```
sns.heatmap(metrics.confusion_matrix(y_test, predicao_gs), cmap="Blues", annot=True)
plt.ylabel('original')
plt.xlabel('predito')
```

Out[140]:

```
Text(0.5, 15.0, 'predito')
```



Com o SVM obtivemos uma acurácia de 98% com somente 3 classificações erradas. A classe de malignos teve 2 classificações como benignos.

Análise dos modelos:

Todos os modelos tiveram um bom desempenho, sendo que o que teve menor acurácia foi o KNN. Já nas análises exploratórias dos dados conseguimos ver uma tendência linear entre os dados que se traduziu nos desempenhos da Regressão Logística e do SVM. Como esse é um problema já bem definido, com regras já criadas para a classificação de tumores, é fácil entender o por que do alto desempenho das Random Forests, e como as features foram selecionadas para a classificação, além de sua eficácia já ter sido comprovada ao longo dos últimos anos.

Conclusão e Melhorias Futuras

Apesar do bom desempenho podemos buscar melhores maneiras e nos aprofundar mais nos modelos. Com uma maior exploração dos hiperparâmetros iremos buscar um melhor desempenho, além de ajustar uma métrica de decisão do GridSearch para buscar uma melhor valor possível do Recall da classe de malignos.

Com 13 features conseguimos 98% de acurácia, mas podemos testar outras variáveis para comparar os resultados a partir de outros métodos de feature selection. Em vez de remover as variáveis uma outra abordagem seria utilizar técnicas como o PCA para extrair suas componentes principais e manter uma pequena dimensionalidade do dataset, não descartando totalmente as features que seriam excluídas do modelo.