応用数学

線形代数学

1. 行列
   ・行列の積：「行（行列の横方向）」×「列（行列の縦方向）」で行列の成分を求める。
   ・行基本変形：各行を〇倍、行同士を加算・入れ替えして、連立方程式を解く。
   ・逆行列：行列の積が単位行列になる行列。
   ・逆行列の求め方：掃き出し法で求めることができる。
   ・逆行列が存在しない条件：行列式がゼロ。
   ・行列式の求め方：複雑な場合、行列式は展開できる。行を入れ替えると符号が変わる。

2. 固有値・固有ベクトル
   ・固有値：m×n の行列の場合、固有値は n 個存在する。
   ・固有ベクトル：1 つの固有値に対する固有ベクトルは複数存在する。（比率が重要）
   ・求め方：$|A-\lambda I|=0$ として、行列式を解く。

3. 固有値分解
   ・固有値分解：固有値と固有ベクトルを求め、$A=V\Lambda V^{-1}$ に変換する。

4. 特異値分解
   ・特異値分解：正方行列以外の固有値分解で、$M=USV^{T}$ を満たす単位ベクトル。
   ・求め方：MMT を固有値分解すると M の特異値と特異ベクトルの 2 乗が求まる。
   ・利用例：特異値を減らすことで、画像のデータ量を効果的に減らせる。

統計学

1. 集合
   ・集合：集合は要素の集まりで、ある集合との関係（含まれるか）を区別できる。
   ・和集合：A または B に含まれる部分。
   ・共通部分：A と B の両方に含まれる部分。
   ・絶対補：A でない部分。
   ・総体補：B のうち A に含まれない部分。

2. 確率
   ・確率：頻度確立（発生頻度）と、ベイズ確率（信念の度合い）の 2 つの考え方がある。

・確率の定義：P(A)=事象 A の起こる数/すべての事象が起こる数
・条件付き確率：P(A|B)=A と B の和集合/事象 B の確率
・独立事象の同時確率：因果関係のない事象の同時発生確率は確率の乗算で求まる。
・ベイズ則：A と B の和集合は、事象 A 条件下の場合と事象 B 条件下の場合は等価。

3. 統計
・期待値：ある確率分布の確率変数の平均値。
・標準偏差：期待値からのズレを大きさ（正の値）で示す。
・ベルヌーイ分布：コイントスのような表・裏の発生割合を示す。
・推定値：^（ハット）を付与して表す。
・不偏分散：標本分散のデータ数（サンプル）が少ないことによる影響を修正。

情報科学

1. 情報科学
・人は情報の増え方（わかりやすさ）を比率でとらえている。
・元の状況からの変化を考えた場合、増加比率の違いを考えるようなこと。

2. 自己情報量
・情報の変化を比率でとらえたもの(1/W)を積分すると log になる。
・確率 P で表すと W の逆数なので、情報量は-log(P(x))になる。

3. シャノンエントロピ
・自己情報量の期待値
・例：コイントスすることで、どれだけ新しく情報を得ることができるか。
・シャノンエントロピが最大になる部分を求めることで予測できる。

4. ダイバージェンス
・カルバック・ライブラー　ダイバージェンス
・同じ事象・確率変数における異なる確率分布 P,Q の違いを表す。
・ダイバージェンスは情報量よりも距離の概念に近い（ようなもの)。
・Q の情報量と P の情報量の差をもとに表現されている。

5. 交差エントロピ
・想定情報（暗号）による信号圧縮のようなもの
・Q についての自己情報量を P の分布で平均する。

機械学習

線形回帰モデル

1. 線形回帰モデル
   ・回帰問題を直線で予測する。
   ・教師あり学習で、教師データに正解データを含む。
   ・予測値は^（ハット）をつけ、正解データと異なることを明示する。
   ・線形結合：入力とパラメータの内積に切片を加える。
   ・モデルのパラメータ：予測値（出力）に対して特徴量の影響を決定する重みの集合。
   ・最小二乗法(MSE)によりパラメータを推定する。

2. データ分割/学習
   ・データは、学習用データ(train)と検証用データ(test)の 2 つに分割する必要がある。
   ・未学習のデータでモデルの汎化性能を測定するため。

3. ハンズオン
   ・E 資格では、Scikit-learn や Numpy で実装を問われることがあるので要注意。
   ・fit は平均 2 乗誤差を最小にする点を探している。

線形単回帰分析

In [11]:
```
#カラムを指定してデータを表示
df[['RM']].head()
```

Out[11]:

	RM
0	6.575
1	6.421
2	7.185
3	6.998
4	7.147

In [12]:
```
# 説明変数
data = df.loc[:, ['RM']].values
```

In [13]:
```
#dataリストの表示(1-5)
data[0:5]
```

Out[13]:
```
array([[6.575],
       [6.421],
       [7.185],
       [6.998],
       [7.147]])
```

In [14]:
```
# 目的変数
target = df.loc[:, 'PRICE'].values
```

In [15]:
```
target[0:5]
```

Out[15]:
```
array([24. , 21.6, 34.7, 33.4, 36.2])
```

In [16]:
```
## sklearnモジュールからLinearRegressionをインポート
from sklearn.linear_model import LinearRegression
```

In [17]:
```
# オブジェクト生成
model = LinearRegression()
#model.get_params()
#model = LinearRegression(fit_intercept = True, normalize = False, copy_X = True,  n_jobs = 1)
```

In [18]:
```
# fit関数でパラメータ推定
model.fit(data, target)
```

Out[18]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

In [19]:
```
#予測
model.predict([[1]])
```

Out[19]:
```
array([-25.5685118])
```

重回帰分析(2変数)

```
In [20]: #カラムを指定してデータを表示
         df[['CRIM', 'RM']].head()
```

Out[20]:

	CRIM	RM
0	0.00632	6.575
1	0.02731	6.421
2	0.02729	7.185
3	0.03237	6.998
4	0.06905	7.147

```
In [21]: # 説明変数
         data2 = df.loc[:, ['CRIM', 'RM']].values
         # 目的変数
         target2 = df.loc[:, 'PRICE'].values

In [22]: # オブジェクト生成
         model2 = LinearRegression()

In [23]: # fit関数でパラメータ推定
         model2.fit(data2, target2)

Out[23]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                          normalize=False)

In [24]: model2.predict([[0.2, 7]])

Out[24]: array([29.43977562])
```

回帰係数と切片の値を確認

```
In [25]: # 単回帰の回帰係数と切片を出力
         print('推定された回帰係数: %.3f, 推定された切片 : %.3f' % (model.coef_, model.intercept_))
```

推定された回帰係数: 9.102, 推定された切片 : -34.671

```
In [26]: # 重回帰の回帰係数と切片を出力
         print(model.coef_)
         print(model.intercept_)
```

```
[9.10210898]
-34.67062077643857
```

考察

サンプルコードに一部誤りがあり、エラーとなったので、修正して再実行した。

Scikit-learn や Numpy、Pandas はデータ分析で、これまで使っていたので、難なく進めることができた。

非線形回帰モデル

1. 非線形回帰モデル
   - ・複雑な非線形構造を内在する現象に対応する。
   - ・線形結合：基底関数（非線形関数）とパラメータベクトルを使用。
   - ・最小二乗法や最尤法によりパラメータを推定する。
   - ・未学習：学習用データに対するモデルの表現力が低い。
   - ・過学習：検証用データに対する汎化性能が低い。差が大きい。

2. 正則化法
   ・モデルの複雑さにより値が大きくなる正則化項をつける。
   ・Ridge 推定量：制約を与えることで、パラメータを 0 に近づける MSE を求める。
   ・Lasso 推定量：ダイヤモンド型の制約で、いくつかのパラメータを正確に 0 にする。
   ・Lasso 推定量(スパース推定)は計算コストを低減することができるメリットもある。

3. モデル選択
   ・ホールドアウト法：大量のデータがないと良い性能評価を与えない欠点がある。
   ・クロスバリデーション（交差検証）：学習用データと検証用データの組合せを変えた複数のイテレータで CV 値（精度の平均）を計測する。
   ・汎用性能を測る場合、クロスバリデーションのほうが優秀な方法。

ロジスティック回帰モデル

1. ロジスティック回帰モデル
   ・数値入力からクラス分類する。
   ・特徴量を入力し、0 または 1 の出力をデータとして扱う。
   ・線形結合：入力とパラメータの内積に切片を加える。
   ・シグモイド関数に線形結合を入力。
   ・シグモイド関数の微分は自身で表現可能。（予習教材で学習済み。）

2. 最尤推定
   ・尤度関数を最大化するパラメータを求める。
   ・対数尤度関数と尤度関数の最大点は同一なので、計算優位性のある対数を使用する。

3. 勾配降下法
   ・ある学習率で反復学習によりパラメータを更新する。
   ・対数尤度関数を係数とバイアスに関して偏微分する。
   ・データ容量や計算時間の問題を持っている。

4. 確率的勾配降下法
   ・データを確立的に選んでパラメータを更新する。
   ・データ容量や計算時間が少ないので、効率的に計算コストが使える。

5. モデルの評価
   ・混同行列：検証用データの結果とモデルの予測結果を表で分類し、各個数を表記。

・再現率：Positive な検証用データを Positive と予測できる割合。
・適合率：Positive の予測結果の内、検証用データも Positive な割合。
・F 値：適合率と再現率の調和平均。値が高いと、適合率及び再現率がともに高い。

6. ハンズオン

1. ロジスティック回帰

不要なデータの削除・欠損値の補完

In [4]:
```
#予測に不要と考えるからうをドロップ（本当はここの情報もしっかり使うべきだと思っています）
titanic_df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1, inplace=True)

#一部カラムをドロップしたデータを表示
titanic_df.head()
```

Out[4]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	male	22.0	1	0	7.2500	S
1	1	1	female	38.0	1	0	71.2833	C
2	1	3	female	26.0	0	0	7.9250	S
3	1	1	female	35.0	1	0	53.1000	S
4	0	3	male	35.0	0	0	8.0500	S

In [5]:
```
#nullを含んでいる行を表示
titanic_df[titanic_df.isnull().any(1)].head(10)
```

Out[5]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
5	0	3	male	NaN	0	0	8.4583	Q
17	1	2	male	NaN	0	0	13.0000	S
19	1	3	female	NaN	0	0	7.2250	C
26	0	3	male	NaN	0	0	7.2250	C
28	1	3	female	NaN	0	0	7.8792	Q
29	0	3	male	NaN	0	0	7.8958	S
31	1	1	female	NaN	1	0	146.5208	C
32	1	3	female	NaN	0	0	7.7500	Q
36	1	3	male	NaN	0	0	7.2292	C
42	0	3	male	NaN	0	0	7.8958	C

In [6]:
```
#Ageカラムのnullを中央値で補完

titanic_df['AgeFill'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())

#再度nullを含んでいる行を表示（Ageのnullは補完されている）
titanic_df[titanic_df.isnull().any(1)]

#titanic_df.dtypes
```

Out[6]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	AgeFill
5	0	3	male	NaN	0	0	8.4583	Q	29.699118
17	1	2	male	NaN	0	0	13.0000	S	29.699118
19	1	3	female	NaN	0	0	7.2250	C	29.699118
26	0	3	male	NaN	0	0	7.2250	C	29.699118
28	1	3	female	NaN	0	0	7.8792	Q	29.699118
29	0	3	male	NaN	0	0	7.8958	S	29.699118

1. ロジスティック回帰

実装(チケット価格から生死を判別)

In [7]:
```
#運賃だけのリストを作成
data1 = titanic_df.loc[:, ["Fare"]].values
```

In [8]:
```
#生死フラグのみのリストを作成
label1 =  titanic_df.loc[:,["Survived"]].values
```

In [9]:
```
from sklearn.linear_model import LogisticRegression
```

In [10]:
```
model=LogisticRegression()
```

In [11]:
```
model.fit(data1, label1)
```

```
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:433: FutureWarning: Default
solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥utils¥validation.py:761: DataConversionWarning: A col
umn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

Out[11]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

In [13]:
```
model.predict([[62]])
```

Out[13]:
```
array([1], dtype=int64)
```

In [16]:
```
model.predict_proba([[61]])
```

Out[16]:
```
array([[0.50345564, 0.49654436]])
```

In [17]:
```
X_test_value = model.decision_function(data1)
```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```
# # 決定関数値（絶対値が大きいほど識別境界から離れている）
# X_test_value = model.decision_function(X_test)
# # 決定関数値をシグモイド関数で確率に変換
# X_test_prob = normal_sigmoid(X_test_value)
```

In [18]:
```
print (model.intercept_)

print (model.coef_)
```

```
[-0.93290045]
[[0.01506685]]
```

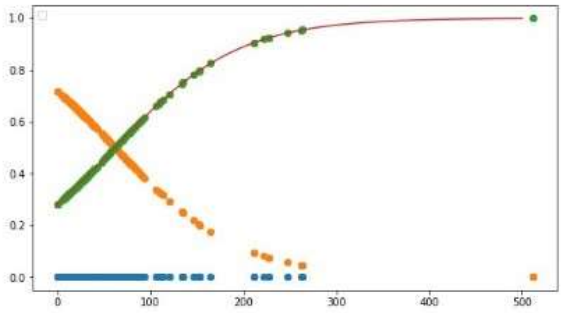
In [19]:

```
w_0 = model.intercept_[0]
w_1 = model.coef_[0,0]

# def normal_sigmoid(x):
#     return 1 / (1+np.exp(-x))

def sigmoid(x):
    return 1 / (1+np.exp(-(w_1*x+w_0)))

x_range = np.linspace(-1, 500, 3000)

plt.figure(figsize=(9,5))
#plt.xkcd()
plt.legend(loc=2)


# plt.ylim(-0.1, 1.1)
# plt.xlim(-10, 10)

# plt.plot([-10,10],[0,0], "k", lw=1)
# plt.plot([0,0],[-1,1.5], "k", lw=1)
plt.plot(data1,np.zeros(len(data1)), 'o')
plt.plot(data1, model.predict_proba(data1), 'o')
plt.plot(x_range, sigmoid(x_range), '-')
#plt.plot(x_range, normal_sigmoid(x_range), '-')
#
```

```
No handles with labels found to put in legend.
```

Out[19]:

```
[<matplotlib.lines.Line2D at 0x28ccb539828>]
```



1. ロジスティック回帰

実装(2変数から生死を判別)

In [ ]:
```
#AgeFillの欠損値を埋めたので
#titanic_df = titanic_df.drop(['Age'], axis=1)
```

In [20]:
```
titanic_df['Gender'] = titanic_df['Sex'].map({'female': 0, 'male': 1}).astype(int)
```

In [21]:
```
titanic_df.head(3)
```

Out[21]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	AgeFill	Gender
0	0	3	male	22.0	1	0	7.2500	S	22.0	1
1	1	1	female	38.0	1	0	71.2833	C	38.0	0
2	1	3	female	26.0	0	0	7.9250	S	26.0	0

In [22]:
```
titanic_df['Pclass_Gender'] = titanic_df['Pclass'] * titanic_df['Gender']
```

In [23]:
```
titanic_df.head()
```

Out[23]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	AgeFill	Gender	Pclass_Gender
0	0	3	male	22.0	1	0	7.2500	S	22.0	1	4
1	1	1	female	38.0	1	0	71.2833	C	38.0	0	1
2	1	3	female	26.0	0	0	7.9250	S	26.0	0	3
3	1	1	female	35.0	1	0	53.1000	S	35.0	0	1
4	0	3	male	35.0	0	0	8.0500	S	35.0	1	4

In [24]:
```
titanic_df = titanic_df.drop(['Pclass', 'Sex', 'Gender','Age'], axis=1)
```

In [25]:
```
titanic_df.head()
```

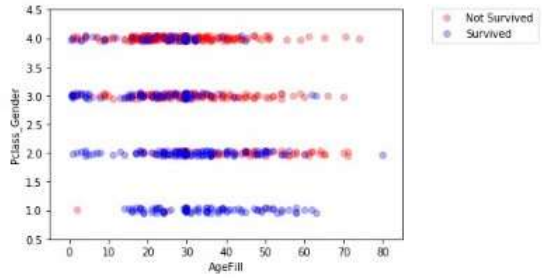
Out[25]:

	Survived	SibSp	Parch	Fare	Embarked	AgeFill	Pclass_Gender
0	0	1	0	7.2500	S	22.0	4
1	1	1	0	71.2833	C	38.0	1
2	1	0	0	7.9250	S	26.0	3
3	1	1	0	53.1000	S	35.0	1
4	0	0	0	8.0500	S	35.0	4

In [ ]:
```
# 重要だよ！！！
# 境界線の式
#    w_1・x + w_2・y + w_0 = 0
#    ⇒ y = (-w_1・x - w_0) / w_2

# # 境界線 プロット
# plt.plot([-2.2], map(lambda x: (-w_1 * x - w_0)/w_2, [-2.2]))

# # データを重ねる
# plt.scatter(X_train_std[y_train==0, 0], X_train_std[y_train==0, 1], c='red', marker='x', label='train 0')
# plt.scatter(X_train_std[y_train==1, 0], X_train_std[y_train==1, 1], c='blue', marker='x', label='train 1')
# plt.scatter(X_test_std[y_test==0, 0], X_test_std[y_test==0, 1], c='red', marker='o', s=60, label='test 0')
# plt.scatter(X_test_std[y_test==1, 0], X_test_std[y_test==1, 1], c='blue', marker='o', s=60, label='test 1')
```

In [26]:
```
np.random.seed = 0

xmin, xmax = -5, 85
ymin, ymax = 0.5, 4.5

index_survived = titanic_df[titanic_df["Survived"]==0].index
index_notsurvived = titanic_df[titanic_df["Survived"]==1].index

from matplotlib.colors import ListedColormap
fig, ax = plt.subplots()
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
sc = ax.scatter(titanic_df.loc[index_survived, 'AgeFill'],
                titanic_df.loc[index_survived, 'Pclass_Gender']+(np.random.rand(len(index_survived))-0.5)*0.1,
                color='r', label='Not Survived', alpha=0.3)
sc = ax.scatter(titanic_df.loc[index_notsurvived, 'AgeFill'],
                titanic_df.loc[index_notsurvived, 'Pclass_Gender']+(np.random.rand(len(index_notsurvived))-0.5)*0.1,
                color='b', label='Survived', alpha=0.3)
ax.set_xlabel('AgeFill')
ax.set_ylabel('Pclass_Gender')
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
ax.legend(bbox_to_anchor=(1.4, 1.03))
```

Out[26]:
```
<matplotlib.legend.Legend at 0x28ccacca320>
```



In [27]:
```
#運賃だけのリストを作成
data2 = titanic_df.loc[:, ["AgeFill", "Pclass_Gender"]].values
```

In [28]:
```
data2
```

Out[28]:
```
array([[22.        ,  4.        ],
       [38.        ,  1.        ],
       [26.        ,  3.        ],
       ...,
       [29.69911765,  3.        ],
       [26.        ,  2.        ],
       [32.        ,  4.        ]])
```

In [29]:
```
#生死フラグのみのリストを作成
label2 =  titanic_df.loc[:,["Survived"]].values
```

In [30]:
```
model2 = LogisticRegression()
```

In [31]:
```
model2.fit(data2, label2)
```

In [32]:
```
model2.predict([[10,1]])
```

Out[32]:
```
array([1], dtype=int64)
```

In [33]:
```
model2.predict_proba([[10,1]])
```

Out[33]:
```
array([[0.06072391, 0.93927609]])
```

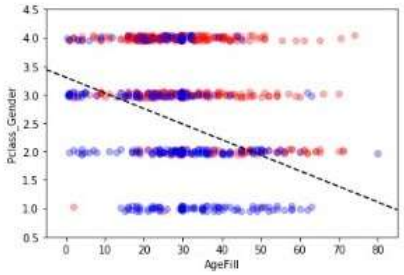
In [34]:
```
titanic_df.head(3)
```

Out[34]:

	Survived	SibSp	Parch	Fare	Embarked	AgeFill	Pclass_Gender
0	0	1	0	7.2500	S	22.0	4
1	1	1	0	71.2833	C	38.0	1
2	1	0	0	7.9250	S	26.0	3

In [35]:
```
h = 0.02
xmin, xmax = -5, 85
ymin, ymax = 0.5, 4.5
xx, yy = np.meshgrid(np.arange(xmin, xmax, h), np.arange(ymin, ymax, h))
Z = model2.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape)

fig, ax = plt.subplots()
levels = np.linspace(0, 1.0)
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
#contour = ax.contourf(xx, yy, Z, cmap=cm, levels=levels, alpha=0.5)

sc = ax.scatter(titanic_df.loc[index_survived, 'AgeFill'],
                titanic_df.loc[index_survived, 'Pclass_Gender']+(np.random.rand(len(index_survived))-0.5)*0.1,
                color='r', label='Not Survived', alpha=0.3)
sc = ax.scatter(titanic_df.loc[index_notsurvived, 'AgeFill'],
                titanic_df.loc[index_notsurvived, 'Pclass_Gender']+(np.random.rand(len(index_notsurvived))-0.5)*0.1,
                color='b', label='Survived', alpha=0.3)

ax.set_xlabel('AgeFill')
ax.set_ylabel('Pclass_Gender')
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
#fig.colorbar(contour)

x1 = xmin
x2 = xmax
y1 = -1*(model2.intercept_[0]+model2.coef_[0][0]*xmin)/model2.coef_[0][1]
y2 = -1*(model2.intercept_[0]+model2.coef_[0][0]*xmax)/model2.coef_[0][1]
ax.plot([x1, x2] ,[y1, y2], 'k--')
```

Out[35]:
```
[<matplotlib.lines.Line2D at 0x28ccb69bfd0>]
```



考察

データの中身を確認しながら実行することで、平均値で補完されている状況が理解できた。

入力値を変更することにより、予測結果が変わることも確認した。

可視化はこれまでもいくつか試したことはあったが、改めて色々な表現があり、もっと調べて試してみたいと感じた。

主成分分析

1. 主成分分析

・次元圧縮により分析や可視化を実現する。
・線形変換後の変数分散が最大となる射影軸を求める。
・無限に解があるので、ノルム 1 の制約を入れる。
・ラグランジュ関数を微分して最適解を求めることは、分散共分散行列の固有値と固有ベクトルを求めることになる。
・第 1 主成分と第 2 主成分は直交する。
・寄与率：ある主成分分散の全分散に対する割合は、ある主成分の情報量の割合となる。
・累積寄与率を求めることで、圧縮による情報損失量の割合を示す。

2. ハンズオン

In [ ]:
```
#https://ohke.hateblo.jp/entry/2017/08/11/230000を参考に利用しています。
```

In [1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
cancer_df = pd.read_csv('../data/cancer.csv')
```

In [3]:
```
print('cancer df shape: {}'.format(cancer_df.shape))
```

```
cancer df shape: (569, 33)
```

In [4]:
```
cancer_df
```

perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	fractal_dimension_worst	Unnamed: 32
184.60	2019.0	0.16220	0.66560	0.71190	0.26540	0.4601	0.11890	NaN
158.80	1956.0	0.12380	0.18660	0.24160	0.18600	0.2750	0.08902	NaN
152.50	1709.0	0.14440	0.42450	0.45040	0.24300	0.3613	0.08758	NaN
98.87	567.7	0.20980	0.86630	0.68690	0.25750	0.6638	0.17300	NaN
152.20	1575.0	0.13740	0.20500	0.40000	0.16250	0.2364	0.07678	NaN
103.40	741.6	0.17910	0.52490	0.53550	0.17410	0.3985	0.12440	NaN
153.20	1606.0	0.14420	0.25760	0.37840	0.19320	0.3063	0.08368	NaN
110.60	897.0	0.16540	0.36820	0.26780	0.15560	0.3196	0.11510	NaN
106.20	739.3	0.17030	0.54010	0.53900	0.20600	0.4378	0.10720	NaN

In [5]:
```
cancer_df.drop('Unnamed: 32', axis=1, inplace=True)
cancer_df
```

exture_worst	perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	fractal_dimension_worst
17.33	184.60	2019.0	0.16220	0.66560	0.71190	0.26540	0.4601	0.11890
23.41	158.80	1956.0	0.12380	0.18660	0.24160	0.18600	0.2750	0.08902
25.53	152.50	1709.0	0.14440	0.42450	0.45040	0.24300	0.3613	0.08758
26.50	98.87	567.7	0.20980	0.86630	0.68690	0.25750	0.6638	0.17300
16.67	152.20	1575.0	0.13740	0.20500	0.40000	0.16250	0.2364	0.07678
23.75	103.40	741.6	0.17910	0.52490	0.53550	0.17410	0.3985	0.12440
27.66	153.20	1606.0	0.14420	0.25760	0.37840	0.19320	0.3063	0.08368
28.14	110.60	897.0	0.16540	0.36820	0.26780	0.15560	0.3196	0.11510
30.73	106.20	739.3	0.17030	0.54010	0.53900	0.20600	0.4378	0.10720

・diagnosis: 診断結果 (良性がB / 悪性がM) ・説明変数は3列以降、目的変数を2列目としロジスティック回帰で分類

```
In [6]:
# 目的変数の抽出
y = cancer_df.diagnosis.apply(lambda d: 1 if d == 'M' else 0)
```

```
In [7]:
# 説明変数の抽出
X = cancer_df.loc[:, 'radius_mean':]
```

```
In [8]:
# 学習用とテスト用でデータを分離
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# 標準化
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ロジスティック回帰で学習
logistic = LogisticRegressionCV(cv=10, random_state=0)
logistic.fit(X_train_scaled, y_train)

# 検証
print('Train score: {:.3f}'.format(logistic.score(X_train_scaled, y_train)))
print('Test score: {:.3f}'.format(logistic.score(X_test_scaled, y_test)))
print('Confustion matrix:\n{}'.format(confusion_matrix(y_true=y_test, y_pred=logistic.predict(X_test_scaled))))
```

```
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
```

```
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
```

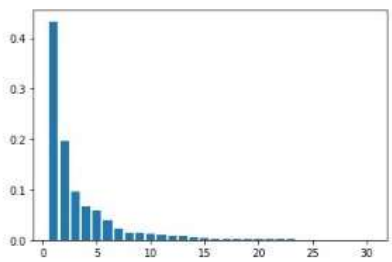
```
Train score: 0.988
Test score: 0.972
Confustion matrix:
[[89  1]
 [ 3 50]]
```

```
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
C:¥Users¥k.nakano¥Miniconda3¥envs¥myenv¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:758: ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
```

・検証スコア97%で分類できることを確認

```
In [9]:
pca = PCA(n_components=30)
pca.fit(X_train_scaled)
plt.bar([n for n in range(1, len(pca.explained_variance_ratio_)+1)], pca.explained_variance_ratio_)
```

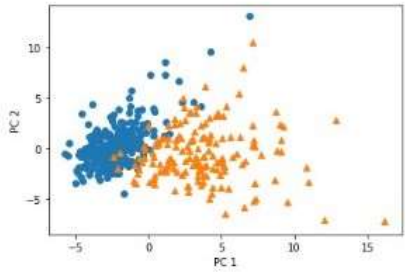
```
Out[9]: <BarContainer object of 30 artists>
```



```
In [10]:
# PCA
# 次元数2まで圧縮
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
print('X_train_pca shape: {}'.format(X_train_pca.shape))
# X_train_pca shape: (426, 2)

# 寄与率
print('explained variance ratio: {}'.format(pca.explained_variance_ratio_))
# explained variance ratio: [ 0.43315126  0.19586506]

# 散布図にプロット
temp = pd.DataFrame(X_train_pca)
temp['Outcome'] = y_train.values
b = temp[temp['Outcome'] == 0]
m = temp[temp['Outcome'] == 1]
plt.scatter(x=b[0], y=b[1], marker='o') # 良性は○でマーク
plt.scatter(x=m[0], y=m[1], marker='^') # 悪性は△でマーク
plt.xlabel('PC 1') # 第1主成分をx軸
plt.ylabel('PC 2') # 第2主成分をy軸
```

```
X_train_pca shape: (426, 2)
explained variance ratio: [0.43315126 0.19586506]
```

```
Out[10]: Text(0, 0.5, 'PC 2')
```



考察

30 次元のデータで 97%の分類精度があることが理解できた。

寄与率をグラフ表示することで、どのくらいの情報損失量かを可視化できる。

2 次元に落とすと 65%の情報量なので、結果があいまいになることがわかった。

アルゴリズム

1. k 近傍法

・最近傍のデータ k 個が最も多く所属するクラスに分類する。

・k の数を変化させると分類結果も変化する。決定境界が滑らかになっていく。

・ハンズオン

k近傍法

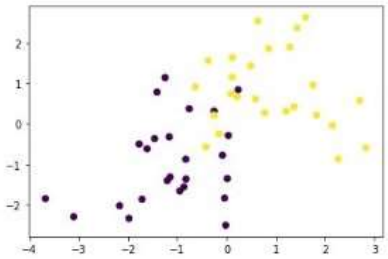
In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

訓練データ生成

In [2]:
```
def gen_data():
    x0 = np.random.normal(size=50).reshape(-1, 2) - 1
    x1 = np.random.normal(size=50).reshape(-1, 2) + 1.
    x_train = np.concatenate([x0, x1])
    y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
    return x_train, y_train
```

In [3]:
```
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
```

Out[3]: <matplotlib.collections.PathCollection at 0x26f8bb40198>



学習

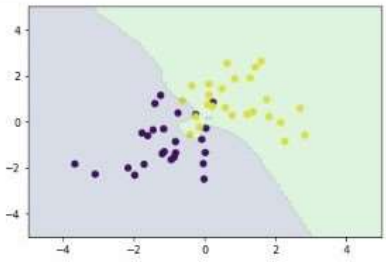
陽に訓練ステップはない

予測

予測するデータ点との、距離が最も近いk個の、訓練データのラベルの最頻値を割り当てる

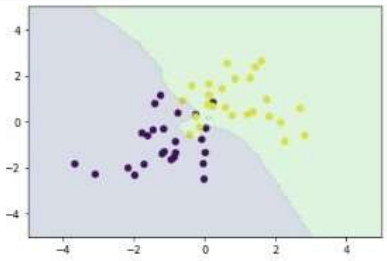
In [4]:
```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

def knc_predict(n_neighbors, x_train, y_train, X_test):
    y_pred = np.empty(len(X_test), dtype=y_train.dtype)
    for i, x in enumerate(X_test):
        distances = distance(x, X_train)
        nearest_index = distances.argsort()[:n_neighbors]
        mode, _ = stats.mode(y_train[nearest_index])
        y_pred[i] = mode
    return y_pred

def plt_resut(x_train, y_train, y_pred):
    xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
    xx = np.array([xx0, xx1]).reshape(2, -1).T
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
    plt.contourf(xx0, xx1, y_pred.reshape(100, 100).astype(dtype=np.float), alpha=0.2, levels=np.linspace(0, 1, 3))
```

In [5]:
```
n_neighbors = 3

xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T

y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```



numpy実装

In [6]:
```
from sklearn.neighbors import KNeighborsClassifier
knc = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X_train, ys_train)
plt_resut(X_train, ys_train, knc.predict(X_test))
```



2. k 平均法(k-means)

・与えられたデータを k 個のクラスタに分類する。

・クラスタリング：特徴の似ているもの同士をグループ化する。

・各クラスタの中心と全データの距離を計算し、最適な各クラスタの中心を求める。

・各クラスタの中心の初期値が重要。ランダムで与えると結果がよい。

・教師なし学習の 1 つ。

・ハンズオン

k平均クラスタリング(k-means)

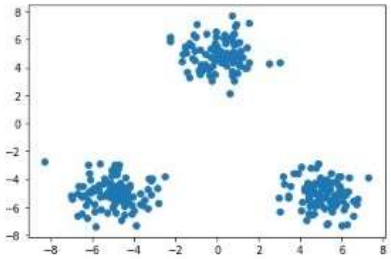
In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

データ生成

In [2]:
```
def gen_data():
    x1 = np.random.normal(size=(100, 2)) + np.array([-5, -5])
    x2 = np.random.normal(size=(100, 2)) + np.array([5, -5])
    x3 = np.random.normal(size=(100, 2)) + np.array([0, 5])
    return np.vstack((x1, x2, x3))
```

In [3]:
```
#データ作成
X_train = gen_data()
#データ描画
plt.scatter(X_train[:, 0], X_train[:, 1])
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x2874703dc18>
```



学習

k-meansアルゴリズムは以下のとおりである

1) 各クラスタ中心の初期値を設定する

2) 各データ点に対して、各クラスタ中心との距離を計算し、最も距離が近いクラスタを割り当てる

3) 各クラスタの平均ベクトル（中心）を計算する

4) 収束するまで2, 3の処理を繰り返す

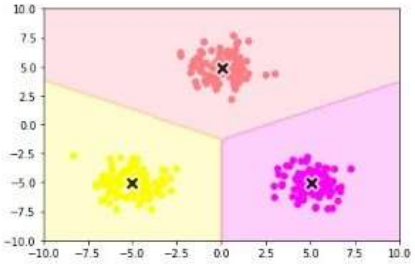
In [4]:
```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

n_clusters = 3
iter_max = 100

# 各クラスタ中心をランダムに初期化
centers = X_train[np.random.choice(len(X_train), n_clusters, replace=False)]

for _ in range(iter_max):
    prev_centers = np.copy(centers)
    D = np.zeros((len(X_train), n_clusters))
    # 各データ点に対して、各クラスタ中心との距離を計算
    for i, x in enumerate(X_train):
        D[i] = distance(x, centers)
    # 各データ点に、最も距離が近いクラスタを割り当
    cluster_index = np.argmin(D, axis=1)
    # 各クラスタの中心を計算
    for k in range(n_clusters):
        index_k = cluster_index == k
        centers[k] = np.mean(X_train[index_k], axis=0)
    # 収束判定
    if np.allclose(prev_centers, centers):
        break
```

クラスタリング結果

In [5]:
```
def plt_result(X_train, centers, xx):
    #  データを可視化
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_pred, cmap='spring')
    # 中心を可視化
    plt.scatter(centers[:, 0], centers[:, 1], s=200, marker='X', lw=2, c='black', edgecolor="white")
    # 領域の可視化
    pred = np.empty(len(xx), dtype=int)
    for i, x in enumerate(xx):
        d = distance(x, centers)
        pred[i] = np.argmin(d)
    plt.contourf(xx0, xx1, pred.reshape(100, 100), alpha=0.2, cmap='spring')
```

In [6]:
```
y_pred = np.empty(len(X_train), dtype=int)
for i, x in enumerate(X_train):
    d = distance(x, centers)
    y_pred[i] = np.argmin(d)
```

In [7]:
```
xx0, xx1 = np.meshgrid(np.linspace(-10, 10, 100), np.linspace(-10, 10, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

plt_result(X_train, centers, xx)
```



numpy実装

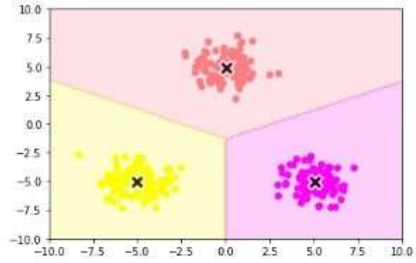
In [8]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=0).fit(X_train)
```

In [9]:
```
print("labels: {}".format(kmeans.labels_))
print("cluster_centers: {}".format(kmeans.cluster_centers_))
kmeans.cluster_centers_
```

```
labels: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 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 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 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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1]
cluster_centers: [[ 5.08381004 -5.06254519]
 [ 0.07687045  4.92203721]
 [-5.02517419 -5.01282305]]
```

Out[9]:
```
array([[ 5.08381004, -5.06254519],
       [ 0.07687045,  4.92203721],
       [-5.02517419, -5.01282305]])
```

In [10]:
```
plt_result(X_train, kmeans.cluster_centers_, xx)
```



サポートベクターマシン

・線形モデルの正負で 2 値分類する。
・線形結合：入力とパラメータの内積に切片を加える。
・符号関数に線形結合を入力。
・マージン：線形判別関数と最近傍点の距離。
・マージンが最大となる線形判別関数を求める。
・分離超平面を構成する学習データは、サポートベクターだけで残りのデータは不要。
・ソフトマージン SVM：線形分離できない場合、誤差を許容しペナルティを与える。
・トレードオフパラメータの大小で決定境界が変化する。
・非線形カーネル（ガウシアン）を用いて分離を可視化できる。
・特徴空間に写像することで、線形分離する。

・ハンズオン

サポートベクターマシン(SVM)

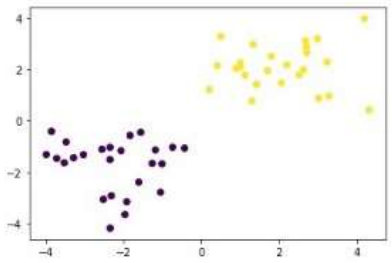
In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

訓練データ生成① （線形分離可能）

In [2]:
```
def gen_data():
    x0 = np.random.normal(size=50).reshape(-1, 2) - 2.
    x1 = np.random.normal(size=50).reshape(-1, 2) + 2.
    X_train = np.concatenate([x0, x1])
    ys_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
    return X_train, ys_train
```

In [3]:
```
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x26d3ed2eda0>
```



学習

特徴空間上で線形なモデル$y(\boldsymbol{x}) = \boldsymbol{w}\boldsymbol{\phi}(\boldsymbol{x}) + b$を用い、その正負によって2値分類を行うことを考える。

サポートベクターマシンではマージンの最大化を行うが、それは結局以下の最適化問題を解くことと同じである。

ただし、訓練データを$\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]^{\mathrm{T}}, \boldsymbol{t} = [t_1, t_2, \ldots, t_n]^{\mathrm{T}} (t_i = \{-1, +1\})$とする。

$\min_{\boldsymbol{w},b} \quad \frac{1}{2}||\boldsymbol{w}||^2$

subject to $\quad t_i(\boldsymbol{w}\boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1 \quad (i = 1, 2, \cdots, n)$

ラグランジュ乗数法を使うと、上の最適化問題はラグランジュ乗数$\boldsymbol{a}(\geq 0)$を用いて、以下の目的関数を最小化する問題となる。

$L(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{n} a_i t_i (\boldsymbol{w}\boldsymbol{\phi}(\boldsymbol{x}_i) + b - 1) \qquad \cdots (1)$

目的関数が最小となるのは、$\boldsymbol{w}, b$に関して偏微分した値が0となるときなので、

$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{n} a_i t_i \boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{0}$

$\frac{\partial L}{\partial b} = \sum_{i=1}^{n} a_i t_i = \boldsymbol{a}^{\mathrm{T}} \boldsymbol{t} = 0$

これを式(1)　に代入することで、最適化問題は結局以下の目的関数の最大化となる。

$$\begin{aligned} \tilde{L}(\boldsymbol{a}) &= \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j t_i t_j \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_j) \\ &= \boldsymbol{a}^{\mathrm{T}} \boldsymbol{1} - \frac{1}{2} \boldsymbol{a}^{\mathrm{T}} H \boldsymbol{a} \end{aligned}$$

ただし、行列Hのi行j列成分は$H_{ij} = t_i t_j \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_j) = t_i t_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$である。また制約条件は、$\boldsymbol{a}^{\mathrm{T}} \boldsymbol{t} = 0 (\frac{1}{2}||\boldsymbol{a}^{\mathrm{T}} \boldsymbol{t}||^2 = 0)$である。

この最適化問題を最急降下法で解く。目的関数と制約条件を$\boldsymbol{a}$で微分すると、

$\frac{d\tilde{L}}{d\boldsymbol{a}} = \boldsymbol{1} - H\boldsymbol{a}$

$\frac{d}{d\boldsymbol{a}} (\frac{1}{2}||\boldsymbol{a}^{\mathrm{T}} \boldsymbol{t}||^2) = (\boldsymbol{a}^{\mathrm{T}} \boldsymbol{t}) \boldsymbol{t}$

なので、$\boldsymbol{a}$を以下の二式で更新する。

$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta_1 (\boldsymbol{1} - H\boldsymbol{a})$

$\boldsymbol{a} \leftarrow \boldsymbol{a} - \eta_2 (\boldsymbol{a}^{\mathrm{T}} \boldsymbol{t}) \boldsymbol{t}$

In [4]:

```
t = np.where(ys_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

eta1 = 0.01
eta2 = 0.001
n_iter = 500

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)
```

予測

新しいデータ点$\boldsymbol{x}$に対しては、$y(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\phi(\boldsymbol{x}) + b = \sum_{i=1}^{n} a_i t_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b$の正負によって分類する。

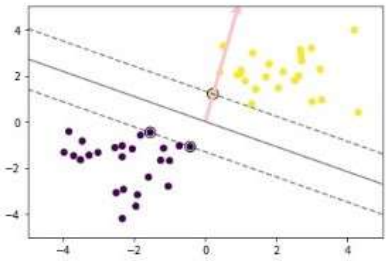
ここで、最適化の結果得られた$a_i (i = 1, 2, \ldots, n)$の中で$a_i = 0$に対応するデータ点は予測に影響を与えないので、$a_i > 0$に対応するデータ点（サポートベクトル）のみ保持しておく。bはサポートベクトルのインデックスの集合を$\mathcal{S}$とすると、$b = \frac{1}{S} \sum_{s \in S} \left(t_s - \sum_{i=1}^{n} a_i t_i k(\boldsymbol{x}_i, \boldsymbol{x}_s) \right)$によって求める。

In [5]:
```
index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
```

In [6]:
```
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)
```

In [7]:
```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
                    s=100, facecolors='none', edgecolors='k')
# 領域を可視化
#plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
                    levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])


# マージンと決定境界を可視化
plt.quiver(0, 0, 0.1, 0.35, width=0.01, scale=1, color='pink')
```

Out[7]: <matplotlib.quiver.Quiver at 0x26d3ed805c0>

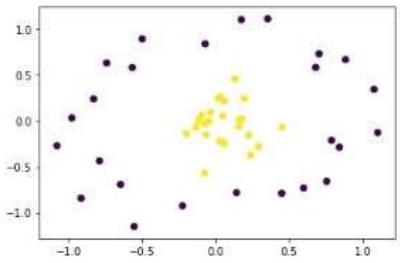


訓練データ生成② （線形分離不可能）

In [8]:
```
factor = .2
n_samples = 50
linspace = np.linspace(0, 2 * np.pi, n_samples // 2 + 1)[:-1]
outer_circ_x = np.cos(linspace)
outer_circ_y = np.sin(linspace)
inner_circ_x = outer_circ_x * factor
inner_circ_y = outer_circ_y * factor

X = np.vstack((np.append(outer_circ_x, inner_circ_x),
               np.append(outer_circ_y, inner_circ_y))).T
y = np.hstack([np.zeros(n_samples // 2, dtype=np.intp),
               np.ones(n_samples // 2, dtype=np.intp)])
X += np.random.normal(scale=0.15, size=X.shape)
x_train = X
y_train = y
```

In [9]:
```
plt.scatter(x_train[:,0], x_train[:,1], c=y_train)
```

Out[9]:
```
<matplotlib.collections.PathCollection at 0x26d3f295358>
```



学習

元のデータ空間では線形分離は出来ないが、特徴空間上で線形分離することを考える。

今回はカーネルとしてRBFカーネル（ガウシアンカーネル）を利用する。

In [10]:
```
def rbf(u, v):
        sigma = 0.8
        return np.exp(-0.5 * ((u - v)**2).sum() / sigma**2)

X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# RBFカーネル
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i, j] = rbf(X_train[i], X_train[j])

eta1 = 0.01
eta2 = 0.001
n_iter = 5000

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)
```

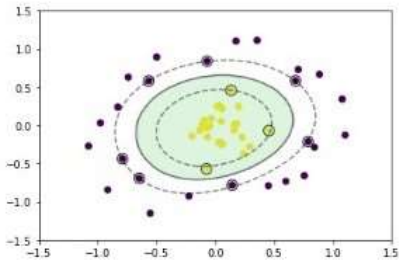
予測

In [11]:
```
index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
```

In [12]:
```
xx0, xx1 = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * rbf(X_test[i], sv)
y_pred = np.sign(y_project)
```

In [13]:
```
# 訓練データを可視化
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
                        s=100, facecolors='none', edgecolors='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
                      levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
```

Out[13]:
```
<matplotlib.contour.QuadContourSet at 0x26d3f2f3f60>
```



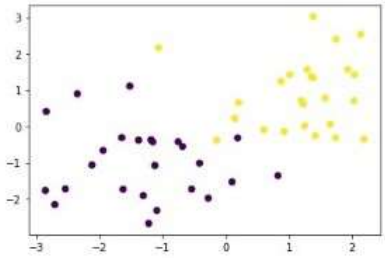
ソフトマージンSVM

訓練データ生成③（重なりあり）

In [14]:
```
x0 = np.random.normal(size=50).reshape(-1, 2) - 1.
x1 = np.random.normal(size=50).reshape(-1, 2) + 1.
x_train = np.concatenate([x0, x1])
y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
```

In [15]:
```
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
```

Out[15]:
```
<matplotlib.collections.PathCollection at 0x26d3f3618d0>
```



学習

分離不可能な場合は学習できないが、データ点がマージン内部に入ることや誤分類を許容することでその問題を回避する。

スラック変数$\xi_i \geq 0$を導入し、マージン内部に入ったり誤分類された点に対しては、$\xi_i = |1 - t_i y(\boldsymbol{x}_i)|$とし、これらを許容する代わりに対して、ペナルティを与えるように、最適化問題を以下のように修正する。

$$\min_{\boldsymbol{w},b} \quad \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

$$\text{subject to} \quad t_i(\boldsymbol{w}\boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1 - \xi_i \quad (i = 1, 2, \cdots, n)$$

ただし、パラメータCはマージンの大きさと誤差の許容度のトレードオフを決めるパラメータである。この最適化問題をラグランジュ乗数法などを用いると、結局最大化する目的関数はハードマージンSVMと同じとなる。

$$\tilde{L}(\boldsymbol{a}) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$$

ただし、制約条件が$a_i \geq 0$の代わりに$0 \leq a_i \leq C (i = 1, 2, \ldots, n)$となる。（ハードマージンSVMと同じ$\sum_{i=1}^{n} a_i t_i = 0$も制約条件）

In [16]:
```
X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

C = 1
eta1 = 0.01
eta2 = 0.001
n_iter = 1000

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.clip(a, 0, C)
```

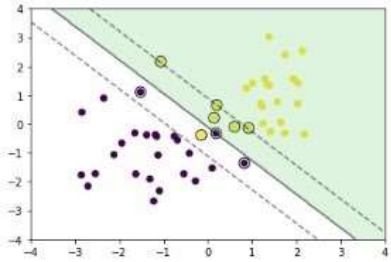
予測

In [17]:
```
index = a > 1e-8
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
```

In [18]:
```
xx0, xx1 = np.meshgrid(np.linspace(-4, 4, 100), np.linspace(-4, 4, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)
```

In [19]:
```
# 訓練データを可視化
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
                    s=100, facecolors='none', edgecolors='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
                    levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
```

Out[19]: <matplotlib.contour.QuadContourSet at 0x26d3f3c6f98>



深層学習：前編 1

Section1：入力層～中間層

・入力毎に重みがかけられ、次の中間層に情報を伝播する。
・入力毎にインデックスとして j が与えられる。
・各層にバイアス項が加えられる。

Section2：活性化関数

・次の層への出力の大きさを決める非線形関数。
・次の層への信号伝播の ON/OFF や強弱を決める。関数を変えると伝播が変わる。
・ステップ関数：0 または 1 の結果を与える。線形分離可能でないと使えない。
・シグモイド関数：0～1 を緩やかに変化する結果を与える。勾配消失問題の課題あり。
・RELU 関数：0 以上で入力値をそのまま結果として与える。現在の主流。

Section3：出力層

1. 誤差関数
    ・出力層はある関数で処理することで、人間が知覚できる理論値として算出できる。
    ・出力結果の数値のばらつきが少ないと学習データが少ないということがある。
    ・出力（予測値）と訓練データ（正解値）の誤差について誤差関数を使用する。
    ・誤差を最小化していくことが学習の目標で、誤差関数はツール。

2. 出力層の活性化関数
    ・出力層と中間層では活性化関数が異なる。
    ・信号の大きさ（比率）をそのままに変換。（価値はそのままに。）
    ・分類問題の場合、出力層の出力の総和を 1 にする必要がある。
    ・恒等写像：回帰の場合で、誤差関数は二乗誤差を使用。
    ・ソフトマックス関数：多クラス分類で誤差関数は交差エントロピを使用。
    ・シグモイド関数：二乗分類で誤差関数は交差エントロピを使用。
    ・上記、活性化関数と誤差関数の組合せで計算の相性がよい。

Section4：勾配降下法

1. 勾配降下法
    ・学習を繰り返すことで、誤差を最小にするパラメータを求める。(最適化する。)
    ・学習率が大きすぎると最小値にたどり着かず発散する。
    ・学習率が小さすぎると収束まで時間がかかる。
    ・大域的極小解に収束させたい。
    ・学習率の決定も考慮が必要。
    ・Momentum、AdaGrad、Adadelta、Adam の向上アルゴリズムがある。
    ・エポック：学習回数

2. 確率的勾配降下法
    ・ランダムに抽出したサンプルの誤差からパラメータを最適化する。
    ・計算コスト削減と局所極小解への収束リスク軽減のメリットがある。
    ・オンライン学習が可能になる。

3. ミニバッチ勾配降下法
    ・ランダムに分割したミニバッチ（データ集合）の平均誤差から最適化する。
    ・確率的勾配降下法のメリットに加え、計算資源を有効活用できるメリットがある。
    ・CPU のスレッド並列化も活用できる。

Section5：誤差逆伝播法

・誤差を出力層から微分し、入力層まで逆伝播する。
・再帰的計算を避け、最小限の計算で微分値を計算する。
・順番に偏微分を繰り返す。

確認テスト

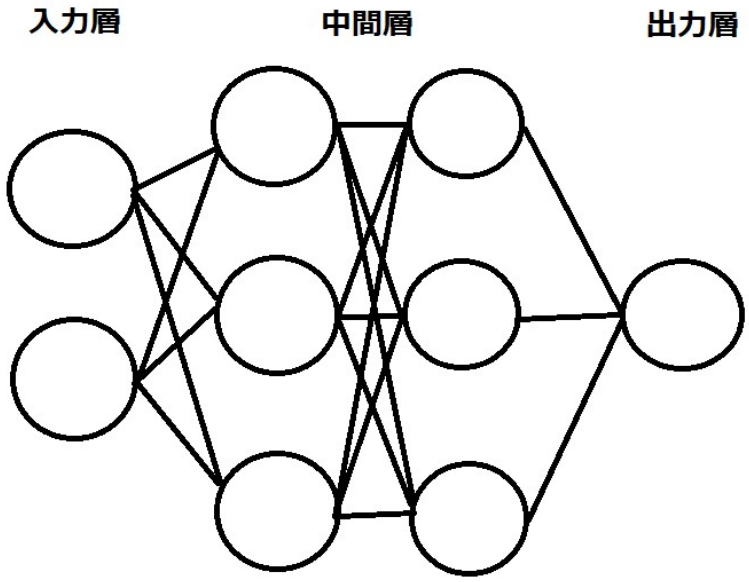
全体像

Q：ディープラーニングは、結局何をやろうとしているのか 2 行以内で述べよ。
また、次の中のどの値を最適化が最終目標か。全て選べ。(1 分)
①入力値[X] ②出力値[Y] ③重み[W] ④バイアス[b]
⑤総入力[u] ⑥中間層入力[z] ⑦学習率[p]

A：誤差を最小化するパラメータを求めること。
③重み[w]と④バイアス[b]

Q：次のネットワークを紙にかけ。
入力層：2 ノード 1 層
中間層：3 ノード 2 層
出力層：1 ノード 1 層
（5 分）

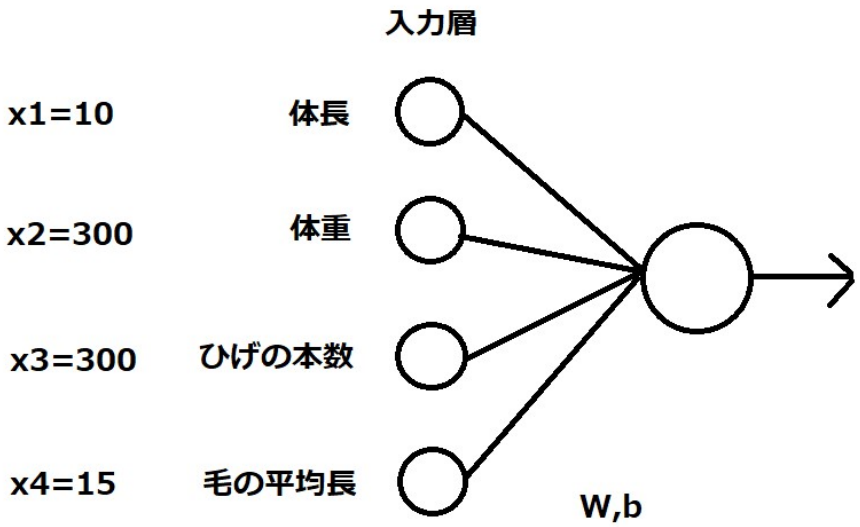
A：



Section1：入力層～中間層

Q：この図式に動物分類の実例を入れてみよう。(3 分)

A：



Q：この数式を Python で書け。(3 分) NumPy のライブラリを使う。

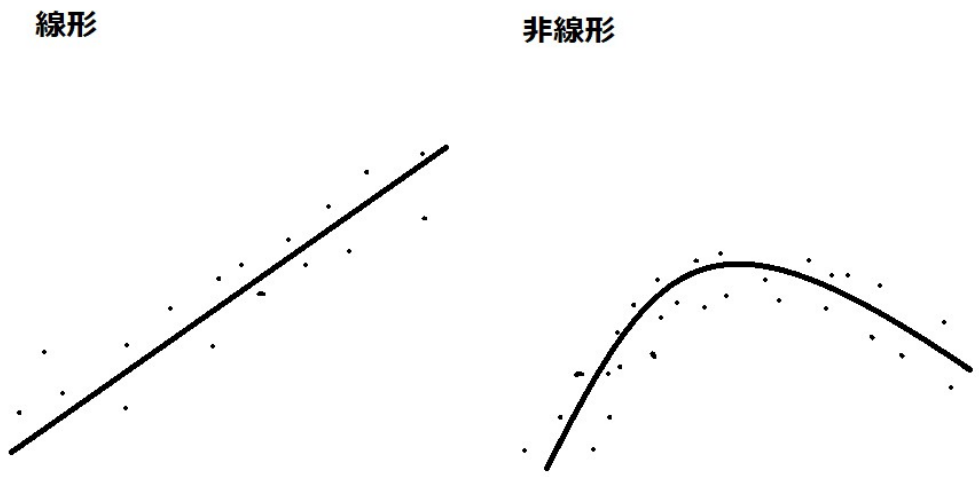
A：u = np.dot(x, W) + b
次元が合わないとエラーになることがあるので要注意。

Q：1-1 のファイルから中間層の出力を定義しているソースを抜き出せ。(3 分)

A：
# 中間層出力
z = functions.relu(u)　　<<--
print_vec("中間層出力", z)

Section2：活性化関数
Q：線形と非線形の違いを図にかいて簡単に説明せよ。(3 分)

A：データ点の分布の特徴に近い線を引くとき、直線で表せるのが線形、曲線で表せるのが非線形となる。



Q：配布されたソースコードより該当する箇所を抜き出せ。(3 分)

A：
# 中間層出力
z = functions.sigmoid(u)　<<--
print_vec("中間層出力", z)

Section3：出力層

Q：なぜ、引き算ではなく二乗するのか述べよ。
下式の 1/2 はどういう意味を持つか述べよ。(2 分)

A：大きさ（正符号）で結果を求めたいから。
1/2 は二乗を微分した時に最終的な計算を簡単にするため。

Q：①～③の数式に該当するソースコードを示し、一行づつ処理の説明をせよ。(5 分)

A：①def softmax(x)
②np.exp(x)
③np.sum(np.exp(x), axis=0)
x が 2 次元の場合、x を転置して扱い、x の最大値を減算した値を x として、y を計算し、y

を転置して戻り値を返す。

Q：①～②の数式に該当するソースコードを示し、一行づつ処理の説明をせよ。(5 分)

A：①def cross_entropy_error(d,y)
②-np.sum(np.log(y[np.arange(batch_size),d]+1e-7))/batch_size
y が 1 次元の場合、d と y の配列形状を変換し、d と y のサイズが同じなら、d を最大値にする。y のサイズを batch size として、交差エントロピを計算する。
※1e-7 は微小な値を与えて 0 にならない実装上の工夫がされている。

Section4：勾配降下法

Q：該当するソースコードを探してみよう。(1 分)

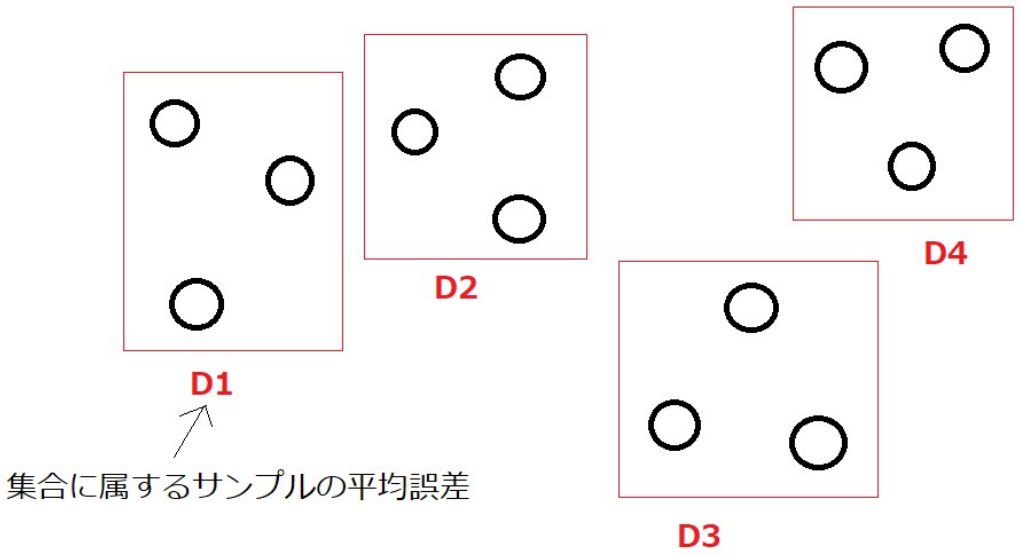
A：
grad = backward(x, d, z1, y)　<<--∇E の式
for key in ('W1', 'W2', 'b1', 'b2'):
　　network[key]　-= learning_rate * grad[key]　<<--w の式

Q：オンライン学習とは何か。2 行でまとめよ。(2 分)

A：既存の学習済みモデルに対して、利用しているユーザーからの情報のみで再学習することとことができる。

Q：この数式の意味を図に書いて説明せよ。(5 分)

A：



Section5：誤差逆伝播法

Q：誤差逆伝播法では不要な再帰的処理を避ける事が出来る。
既に行った計算結果を保持しているソースコードを抽出せよ。(3 分)

A：

```
# 誤差逆伝播
def backward(x, d, z1, y):
    print("¥n#####  誤差逆伝播開始  #####")

    grad = {}

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    #   出力層でのデルタ
    delta2 = functions.d_sigmoid_with_loss(d, y)
    #   b2 の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    #   W2 の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    #   中間層でのデルタ
```

```
        delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
        # b1 の勾配
        grad['b1'] = np.sum(delta1, axis=0)
        #   W1 の勾配
        grad['W1'] = np.dot(x.T, delta1)

        print_vec("偏微分_dE/du2", delta2)
        print_vec("偏微分_dE/du2", delta1)

        print_vec("偏微分_重み 1", grad["W1"])
        print_vec("偏微分_重み 2", grad["W2"])
        print_vec("偏微分_バイアス 1", grad["b1"])
        print_vec("偏微分_バイアス 2", grad["b2"])

        return grad
```

Q：2 つの空欄に該当するソースコードを探せ。(3 分)

A：
u での微分：delta2 = functions.d_mean_squared_error(d, y)
w2 での微分：grad['W2'] = np.dot(z1.T, delta2)

演習問題

Section1：入力層～中間層

重み項での配列初期化とバイアス項での数値の初期化

In [7]:

```
# 順伝播（単層・単ユニット）

# 重み
#W = np.array([[0.1], [0.2]])

## 試してみよう_配列の初期化
#W = np.zeros(2)
#W = np.ones(2)
#W = np.random.rand(2)
W = np.random.randint(5, size=(2))

print_vec("重み", W)


# バイアス
#b = 0.5

## 試してみよう_数値の初期化
#b = np.random.rand() # 0~1のランダム数値
b = np.random.rand() * 10 -5  # -5~5のランダム数値

print_vec("バイアス", b)

# 入力値
x = np.array([2, 3])
print_vec("入力", x)


# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.relu(u)
print_vec("中間層出力", z)
```

```
*** 重み ***
[0 2]

*** バイアス ***
2.802129592069507

*** 入力 ***
[2 3]

*** 総入力 ***
8.802129592069507

*** 中間層出力 ***
8.802129592069507
```

考察

・NumPy を活用した初期値の与え方を試して学んだ。

・NumPy がランダムに初期値を生成してくれるのは、記述も簡単で非常に使い易い。

・重みの配列を増やし、次元数の矛盾によるエラー内容の確認と、入力の次元数を増やし対処することも試し確認した。

Section2：活性化関数

重み項での配列初期化と活性化関数の変更

In [8]:
```
# 順伝播（単層・複数ユニット）

# 重み
#W = np.array([
#    [0.1, 0.2, 0.3],
#    [0.2, 0.3, 0.4],
#    [0.3, 0.4, 0.5],
#    [0.4, 0.5, 0.6]
#])

## 試してみよう_配列の初期化
#W = np.zeros((4,3))
#W = np.ones((4,3))
#W = np.random.rand(4,3)
W = np.random.randint(5, size=(4,3))

print_vec("重み", W)

# バイアス
b = np.array([0.1, 0.2, 0.3])
print_vec("バイアス", b)

# 入力値
x = np.array([1.0, 5.0, 2.0, -1.0])
print_vec("入力", x)


#  総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
#z = functions.sigmoid(u)
z = functions.relu(u)
print_vec("中間層出力", z)
```

```
*** 重み ***
[[0 4 4]
 [3 2 4]
 [1 2 2]
 [1 4 0]]

*** バイアス ***
[0.1 0.2 0.3]

*** 入力 ***
[ 1.  5.  2. -1.]

*** 総入力 ***
[16.1 14.2 28.3]

*** 中間層出力 ***
[16.1 14.2 28.3]
```

考察

・単層・複数ノードで NumPy を活用した初期値の与え方を試して学んだ。

・活性化関数をシグモイド関数から RELU 関数に変更し、変化を確認した。

・from common import functions で、functions.py を読み込んでいることも理解した。

Section5：誤差逆伝播法

RELU 関数からシグモイド関数に変更する。入力値の設定を変更する。

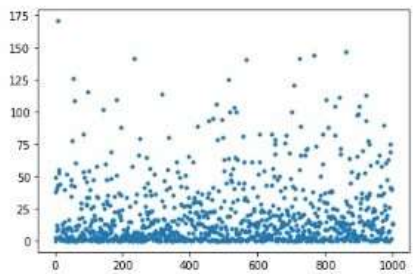
In [1]:
```python
# 確率勾配降下法
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print("shape: " + str(x.shape))
    print("")
```

In [4]:
```python
# サンプルとする関数
#yの値を予想するAI

def f(x):
    y = 3 * x[0] + 2 * x[1]
    return y

# 初期設定
def init_network():
    # print("##### ネットワークの初期化 #####")
    network = {}
    nodesNum = 10
    network['W1'] = np.random.randn(2, nodesNum)
    network['W2'] = np.random.randn(nodesNum)
    network['b1'] = np.random.randn(nodesNum)
    network['b2'] = np.random.randn()

    # print_vec("重み1", network['W1'])
    # print_vec("重み2", network['W2'])
    # print_vec("バイアス1", network['b1'])
    # print_vec("バイアス2", network['b2'])

    return network

# 順伝播
def forward(network, x):
    # print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    u1 = np.dot(x, W1) + b1
    #z1 = functions.relu(u1)

    ## 試してみよう
    z1 = functions.sigmoid(u1)

    u2 = np.dot(z1, W2) + b2
    y = u2

    # print_vec("総入力1", u1)
    # print_vec("中間層出力1", z1)
    # print_vec("総入力2", u2)
    # print_vec("出力1", y)
    # print("出力合計: " + str(np.sum(y)))

    return z1, y

# 誤差逆伝播
def backward(x, d, z1, y):
    # print("\n##### 誤差逆伝播開始 #####")

    grad = {}

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
```

```
    # 出力層でのデルタ
    delta2 = functions.d_mean_squared_error(d, y)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 中間層でのデルタ
    #delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)

    ## 試してみよう
    delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

    delta1 = delta1[np.newaxis, :]
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    x = x[np.newaxis, :]
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    # print_vec("偏微分_重み1", grad["W1"])
    # print_vec("偏微分_重み2", grad["W2"])
    # print_vec("偏微分_バイアス1", grad["b1"])
    # print_vec("偏微分_バイアス2", grad["b2"])

    return grad

# サンプルデータを作成
data_sets_size = 100000
data_sets = [0 for i in range(data_sets_size)]

for i in range(data_sets_size):
    data_sets[i] = {}
    # ランダムな値を設定
    #data_sets[i]['x'] = np.random.rand(2)

    ## 試してみよう_入力値の設定
    data_sets[i]['x'] = np.random.rand(2) * 10 -5 # -5~5のランダム数値

    # 目標出力を設定
    data_sets[i]['d'] = f(data_sets[i]['x'])

losses = []
# 学習率
learning_rate = 0.07

# 抽出数
epoch = 1000

# パラメータの初期化
network = init_network()
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key]  -= learning_rate * grad[key]

    # 誤差
    loss = functions.mean_squared_error(d, y)
    losses.append(loss)

print("##### 結果表示 #####")
lists = range(epoch)
```

```
plt.plot(lists, losses, '.')
# グラフの表示
plt.show()
```

```
##### 結果表示 #####
```



考察
・サンプルコードが RELU 関数なのに、中間層でのデルタ計算にシグモイド関数の導関数を使用していたので、修正し実行した。
・RELU 関数からシグモイド関数に変更するとグラフのばらつきが増えることを確認した。
・入力値の設定を変更すると更にグラフのばらつきが増えることを確認した。

修了課題

Q1.課題の目的とは？　どのような工夫ができそうか。

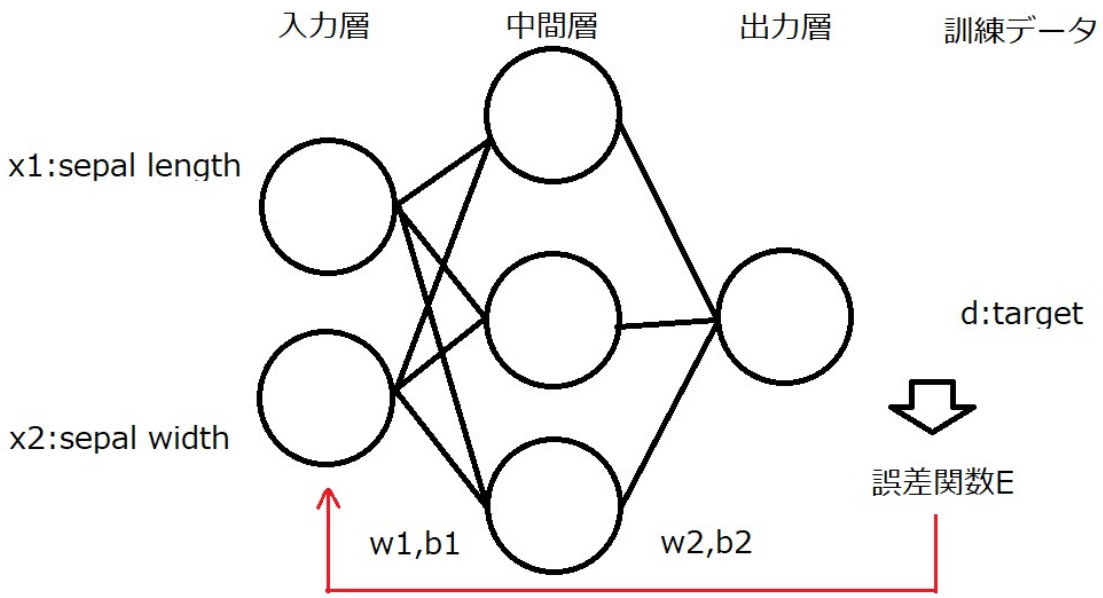
A1.構造図（設計書）どおりの実装ができるか。
設定値を変更できるようにする。
実データを加工して、入力値を変更できるようにする。

Q2.課題を分類タスクで解く場合の意味は何か。

A2.iris データが判別分析用データセットであることと、予測結果との誤差がわかりやすい。

Q3.iris データとは何か 2 行で述べよ。

A3.3 種類のあやめに関する 4 個の計測値からなる判別分析やクラスタ分析などの研究に使われるテストデータ。scikit-learn のデータセットとして準備されている。


入力層
中間層
出力層
訓練データ
x1:sepal length
x2:sepal width
d:target
誤差関数E
w1,b1
w2,b2

修了課題　- 深層学習：前編１

irisデータセットを利用したNNの構築　-　versicolorの判別

In [1]:
```
import sys
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# irisデータセットの準備
from sklearn.datasets import load_iris
iris = load_iris()
```

In [3]:
```
# irisデータセットの説明
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
```

In [4]:
```
# データセットの形状
iris.data.shape
```

Out[4]: (150, 4)

In [5]:
```
# データセットをデータフレームに格納
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['target'] = iris.target
#df.loc[df['target'] == 0, 'target'] = "setosa"
#df.loc[df['target'] == 1, 'target'] = "versicolor"
#df.loc[df['target'] == 2, 'target'] = "virginica"
```

In [6]:
```
# versicolor以外を0に置換
df = df.replace({'target':{2:0}})
```

In [7]:
```
df.shape
```

Out[7]: (150, 5)

In [8]:
```
df.head()
```

Out[8]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
target               150 non-null int64
dtypes: float64(4), int64(1)
memory usage: 5.9 KB
```

In [10]:
```
# 入力値を準備
df1 = df.drop(['petal length (cm)','petal width (cm)','target'], axis=1)
df1 = df1.T
# 目標値を準備
df2 = df['target']
df2 = df2.T
```

In [11]:
```
# サンプルデータを作成
data_sets_size = df.shape[0]
data_sets = [0 for i in range(data_sets_size)]

for i in range(data_sets_size):
    data_sets[i] = {}

    ## 入力値を設定
    data_sets[i]['x'] = df1[i].values

    # 目標出力を設定
    data_sets[i]['d'] = df2[i]
```

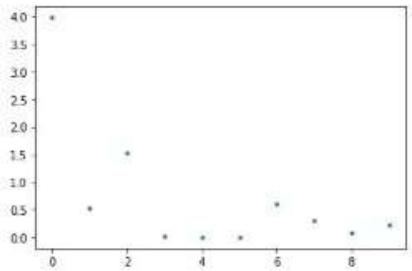
In [12]:
```
data_sets
```

Out[12]:
```
[{'x': array([5.1, 3.5]), 'd': 0},
 {'x': array([4.9, 3. ]), 'd': 0},
 {'x': array([4.7, 3.2]), 'd': 0},
 {'x': array([4.6, 3.1]), 'd': 0},
 {'x': array([5. , 3.6]), 'd': 0},
 {'x': array([5.4, 3.9]), 'd': 0},
 {'x': array([4.6, 3.4]), 'd': 0},
 {'x': array([5. , 3.4]), 'd': 0},
 {'x': array([4.4, 2.9]), 'd': 0},
 {'x': array([4.9, 3.1]), 'd': 0},
 {'x': array([5.4, 3.7]), 'd': 0},
 {'x': array([4.8, 3.4]), 'd': 0},
 {'x': array([4.8, 3. ]), 'd': 0},
 {'x': array([4.3, 3. ]), 'd': 0},
 {'x': array([5.8, 4. ]), 'd': 0},
 {'x': array([5.7, 4.4]), 'd': 0},
 {'x': array([5.4, 3.9]), 'd': 0},
 {'x': array([5.1, 3.5]), 'd': 0},
 {'x': array([5.7, 3.8]), 'd': 0},
```

In [13]:
```
# どの種類のあやめ(yの値)を予想する

# シグモイド関数
def sigmoid(x):
    return 1/(1 + np.exp(-x))

# シグモイド関数の導関数
def d_sigmoid(x):
    dx = (1.0 - sigmoid(x)) * sigmoid(x)
    return dx

# 平均二乗誤差
def mean_squared_error(d, y):
    return np.mean(np.square(d - y)) / 2

# 平均二乗誤差の導関数
def d_mean_squared_error(d, y):
    if type(d) == np.ndarray:
        batch_size = d.shape[0]
        dx = (y - d)/batch_size
    else:
        dx = y - d
    return dx

# 初期設定
def init_network():
    network = {}
    nodesNum = 3
    network['W1'] = np.random.randn(2, nodesNum)
    network['W2'] = np.random.randn(nodesNum)
    network['b1'] = np.random.randn(nodesNum)
    network['b2'] = np.random.randn()
    return network

# 順伝播
def forward(network, x):
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    u1 = np.dot(x, W1) + b1
    z1 = sigmoid(u1)
    u2 = np.dot(z1, W2) + b2
    y = u2
    return z1, y

# 誤差逆伝播
def backward(x, d, z1, y):
    grad = {}
```

```
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 出力層でのデルタ
    delta2 = d_mean_squared_error(d, y)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 中間層でのデルタ
    delta1 = np.dot(delta2, W2.T) * d_sigmoid(z1)

    delta1 = delta1[np.newaxis, :]
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    x = x[np.newaxis, :]
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    return grad

losses = []
# 学習率
learning_rate = 0.07

# 抽出数
epoch = 10

# パラメータの初期化
network = init_network()
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key]  -= learning_rate * grad[key]

    # 誤差
    loss = mean_squared_error(d, y)
    losses.append(loss)

print("##### 結果表示 #####")
lists = range(epoch)


plt.plot(lists, losses, '.')
# グラフの表示
plt.show()
```

```
##### 結果表示 #####
```



参照ソースコード　－　iris-practice.ipynb