



python で データ解析

5. XGBoost を用いた機械学習の実践
～ 2 値分類問題 ～

メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- その他

※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

kaggle: <https://www.kaggle.com/>

- 企業や研究者がデータを投稿し、世界中の統計家やデータ分析その最適なモデルをコンペという形で競い合う、予測モデリング及び分析手法関連のプラットフォーム及びその運営会社 (Wikipedia より: <https://ja.wikipedia.org/wiki/Kaggle>)
- 初心者向けのチュートリアルや上級者のプログラム公開、Discussion 等があり、コンペに参加しなくても非常に勉強になる
- 次頁の titanic データは kaggle の学習用コンペのもの、kaggle に登録(無料)すればダウンロードすることが出来る

Competitions

Grow your data science skills by competing in our exciting competitions. Find help in the [Documentation](#) or learn about [InClass competitions](#).

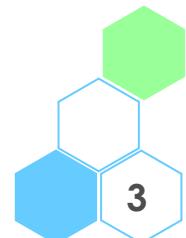
Your Competitions

Active Closed Pinned Hosted

Competition	Description	Category
Titanic: Machine Learning from Disaster	Start here! Predict survival on the Titanic and get familiar with ML basics Getting Started • Ongoing • 17246 Teams	Knowledge
House Prices: Advanced Regression Techniques	Predict sales prices and practice feature engineering, RFs, and gradient boosting Getting Started • Ongoing • 4440 Teams	Knowledge

All Competitions

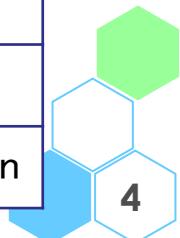
Active (Not Entered) Completed InClass All Categories ▾ Default Sort ▾



使用するデータ: *Titanic*

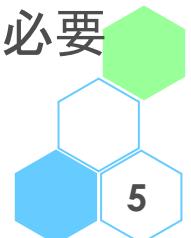
- [kaggle](#) のコンペで使用されている、[タイタニック号に関するデータ](#)
- train.csv(学習用データ)と test.csv(テストデータ)がある

変数	定義	中身の補足
PassengerId	ID	train.csv: 1～891、test.csv: 892～1309
Survived	生死 → 目的変数	0 = No, 1 = Yes: test.csv にはなし
Pclass	乗車券のクラス	≒経済状況: 1=1st, 2=2nd, 3=3rd
Name	名前	例: "Heikkinen, Miss. Laina"、敬称付き
Sex	性別	male, female
Age	年齢	
SibSp	同乗している兄弟や配偶者の数	
Parch	同乗している親や子供の数	
Ticket	乗車券番号	例: PC 17599
Fare	料金	
Cabin	客室	例: C85
Embarked	乗船港	C = Cherbourg, Q = Queenstown, S = Southampton

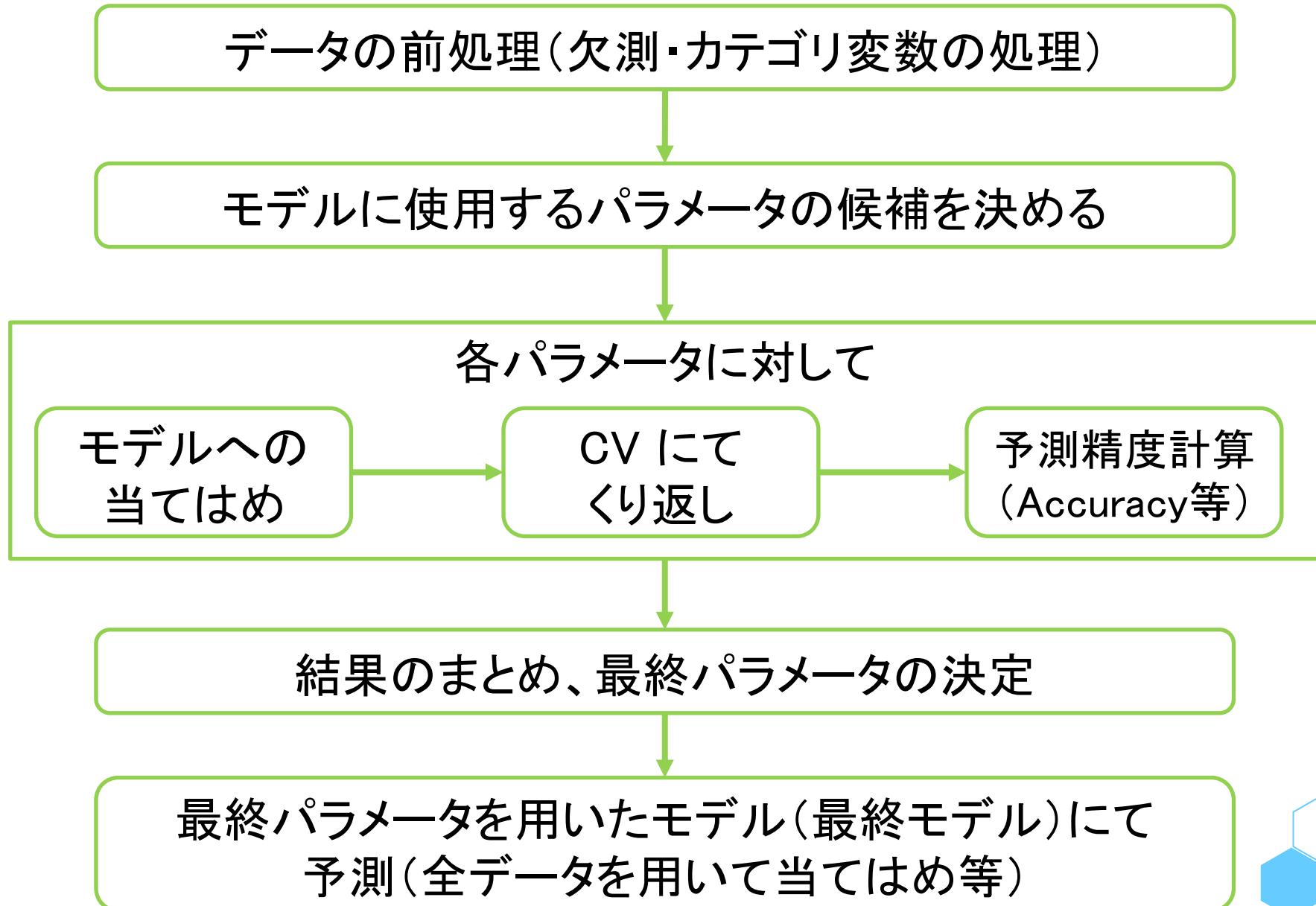


XGBoost: GBDT のライブラリ

- 勾配ブースティング(Gradient Boosting Decision Tree、GBDT)のためのライブラリで、決定木をベースとしたモデル
 - 目的変数と予測値から計算される目的関数を改善するよう決定木を作成してモデルに追加
 - #1を、ハイパーパラメータで設定した決定木の本数分くり返す
 - Random Forest では木を並列に作成するが、GBDT では木を直接に作成し、直前まで作成した決定木の予測値を新しい決定木に加味することで微修正
- 使いやすい上に精度が高く、分類でも回帰でも適用できる
 - kaggle のコンペでは、GBDT をまず最初に適用すること
- 変数は数値、データの数値の大きさには意味がなく、大小関係のみが影響 → 標準化などのスケーリング処理は GBDT の場合はほぼ意味がない
- 欠測値があってもそのまま処理できる
- 決定木の分岐のくり返しによって変数間の交互作用を反映（だが、交互作用を反映した変数を別途作成した方が良いが）
- カテゴリ変数は label encoding での変換が基本、one-hot encoding は必要に応じて、target encoding がより有効だが上級者でないと扱いが難
- 疎行列(scipy のcsr_matrix/csc_matrix)に対応可



XGBoost: 手順



メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- その他

※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

前処理

```

import numpy      as np
import pandas     as pd
from    sklearn.preprocessing import LabelEncoder

# train.csv と test.csv の結合
df1 = pd.read_csv('C:/py/titanic/train.csv', header=0)
df0 = pd.read_csv('C:/py/titanic/test.csv',   header=0)
df1["Is_train"] = 1
df0["Is_train"] = 0
df = pd.concat([df1, df0])

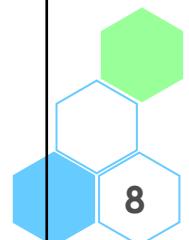
# Name、Ticket を数値化 (Cabinは欠測多数のため放置)
cat_cols = ['Name', 'Ticket']
for c in cat_cols:
    le = LabelEncoder() # LabelEncoding
    le.fit(df[c])
    df[c] = le.transform(df[c])

# 'Sex' をカテゴリ化
df.Sex = df.Sex.apply(lambda x : 1 if x == 'male' else 0)

# 'Embarked' をカテゴリ化
my_mapping = {'S':1, 'C':2, 'Q':3}
df.Embarked = df.Embarked.replace(my_mapping) # NaN は一旦放置
df.Embarked = df.Embarked.fillna(0)          # NaN を 0 に置換
df.Embarked = df.Embarked.astype('int8')       # 整数型に変換

df = pd.concat([df.drop(labels=['Embarked'], axis=1),
    pd.get_dummies(df.Embarked, drop_first=True,
    prefix='Embarked')], axis=1)

```



前処理

```
# 'SibSp' と 'Parch' から家族数 'Family' を計算
df['Family'] = df.SibSp + df.Parch

# 学習用データとテストデータに分割
train_x = df.query('Is_train == 1')[['Pclass', 'Sex', 'Age',
                                         'Fare', 'Embarked_1', 'Embarked_2', 'Embarked_3', 'Family']]
train_y = df.query('Is_train == 1')['Survived']
test_x = df.query('Is_train == 0')[['Pclass', 'Sex', 'Age',
                                       'Fare', 'Embarked_1', 'Embarked_2', 'Embarked_3', 'Family']]
id = df.query('Is_train == 0')['PassengerId']
```

- train_x: Pclass、Sex、Age、Embarked、Family(家族の数)に絞る
 - Name(名前)、Ticket(乗船券番号)、Cabin(客室)は扱いが難しそうなので一旦除外

	Pclass	Sex	Age	Fare	Embarked_1	Embarked_2	Embarked_3	Family
0	3	1	22.0	7.2500	1	0	0	1
1	1	0	38.0	71.2833	0	1	0	1
2	3	0	26.0	7.9250	1	0	0	0
3	1	0	35.0	53.1000	1	0	0	1
4	3	1	35.0	8.0500	1	0	0	0



とりあえずモデル構築 → 予測

- データの変換 → 学習の実行 → 結果の確認(指標はlogloss) → 正解率の計算

```
# xgboost 用のデータ構造に変換
dtrain = xgb.DMatrix(tr_x, label=tr_y)
dvalid = xgb.DMatrix(va_x, label=va_y)
dtest = xgb.DMatrix(test_x)

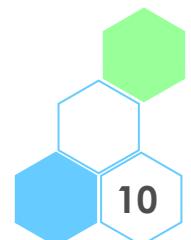
# ハイパーパラメータの設定
params = {'objective': 'binary:logistic',
           'random_state': 777}
num_round = 200

# 学習の実行、バリデーションデータもモデルに指定、スコアをモニタリング
# 引数 watchlist に学習データとバリデーションデータをセット
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]
model = xgb.train(params, dtrain, num_round, evals=watchlist)

# バリデーションデータでのスコアの確認
va_pred = model.predict(dvalid)
score = log_loss(va_y, va_pred)
print(f'logloss: {score:.5f}')

# 予測（予測値ではなく、1である確率を出力）
pred = model.predict(dtest)

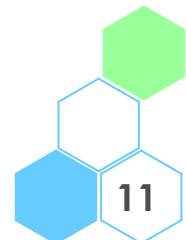
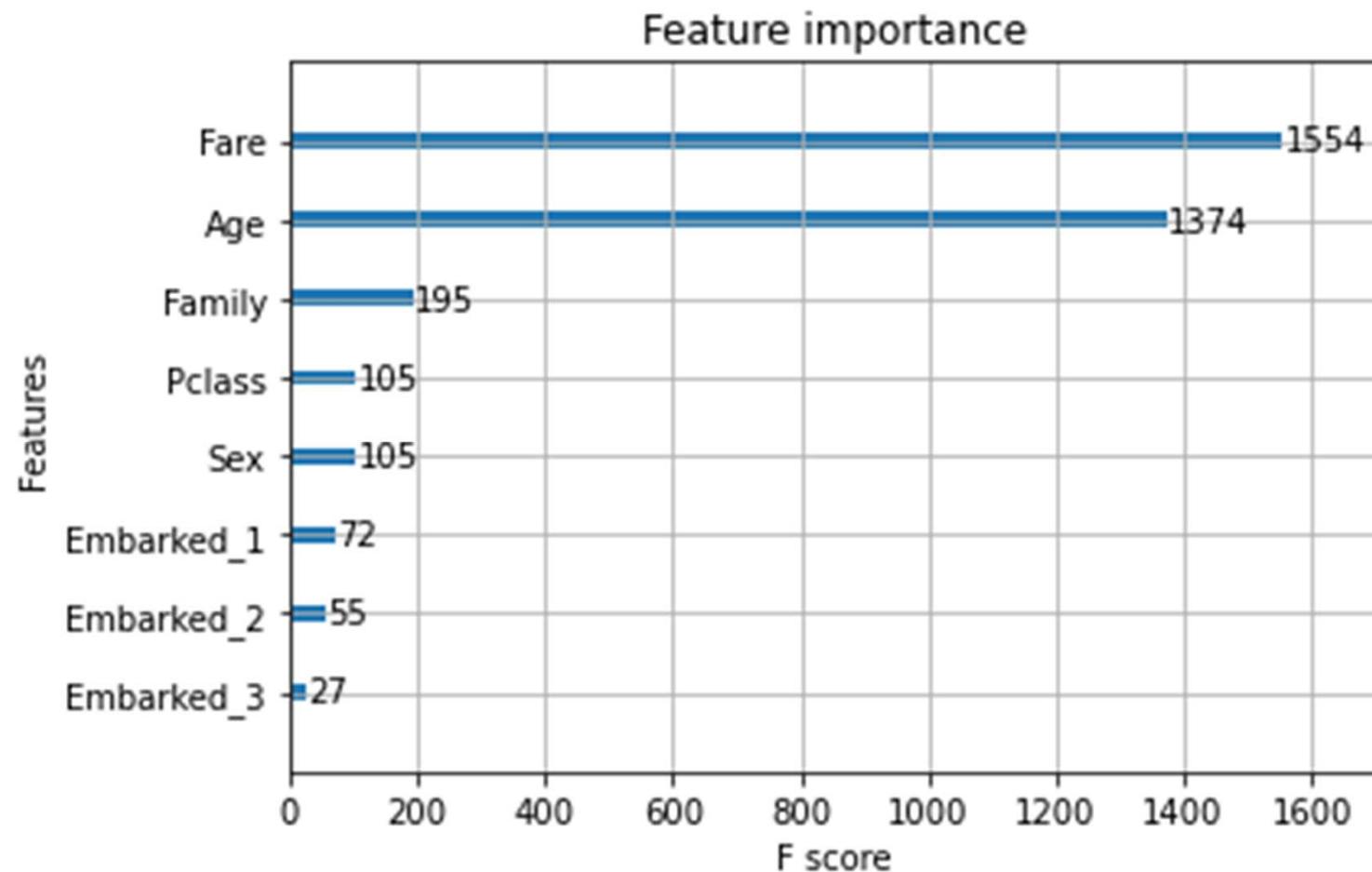
# 予測値と一致率
pred_label = np.where(pred > 0.5, 1, 0)
score = accuracy_score(pred_label, test_y)
print('score:{0:.5f}'.format(score))
```



とりあえずモデル構築 → 予測

- 重要度:どの変数が予測に寄与したか可視化

```
xgb.plot_importance(model)
```



予測結果を Kaggle のコンペに提出

- 予測結果を csv ファイルに出力

```
out = pd.DataFrame({'PassengerId':id, 'Survived':pred_label})  
out.to_csv('C:/py/titanic/gender_submission.csv', index=False)
```

Overview Data Notebooks Discussion Leaderboard My Submissions **Submit Predictions** ← ① クリック

Step 1
Upload submission file

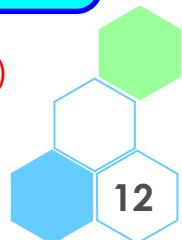
Step 2
Describe submission

Briefly describe your submission

kaggle のコンペの
score: 0.74880

Make Submission ← ③ クリックして提出 (スコアが表示)

Titanic のコンペ : <https://www.kaggle.com/c/titanic/submit>



参考: 他の手法との比較

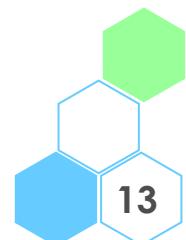
```
# 「学習用データとテストデータに分割」の前で Age、Fare の欠測を中央値で補完
# (xgboost以外は欠測を許容しないため)
df.Age = df.Age.fillna(df.Age.median())
df.Fare = df.Fare.fillna(df.Fare.median())

# SGD: score=0.71052
from sklearn.linear_model import SGDClassifier
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=100,
random_state=777)
clf.fit(train_x, train_y.values.ravel())
pred = clf.predict(test_x)
pred_label = np.where(pred > 0.5, 1, 0)

# CART: score=0.73923
from sklearn.tree import DecisionTreeClassifier, plot_tree
clf = DecisionTreeClassifier()
clf.fit(train_x, train_y.values.ravel())
pred = clf.predict(test_x)
pred_label = np.where(pred > 0.5, 1, 0)

# RandomForest: score=0.76076 ⇒ この問題には RandomForest が適している?
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier()
clf.fit(train_x, train_y.values.ravel())
pred = clf.predict(test_x)
pred_label = np.where(pred > 0.5, 1, 0)

# xgboost: score=0.74162
dtrain = xgb.DMatrix(train_x, label=train_y)
dtest = xgb.DMatrix(test_x)
params = {'objective': 'binary:logistic', 'random_state': 777}
num_round = 200
model = xgb.train(params, dtrain, num_round)
pred = model.predict(dtest)
pred_label = np.where(pred > 0.5, 1, 0)
```



メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- その他

※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

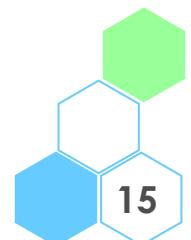
バリデーション: Hold-out 法

- 手法の詳細: <https://xgboost.readthedocs.io/en/latest/tutorials/index.html>
- 扱えるデータの種類: https://xgboost.readthedocs.io/en/latest/python/python_intro.html
- パラメータの種類: <https://xgboost.readthedocs.io/en/latest/parameter.html>
- データは 8 ~ 9 頁処理後の train_x, train_y をさらに分割している
 - train_x, train_y, test_x: それぞれ学習データと目的変数、テストデータ
 - tr_x, tr_y, va_x, va_y: それぞれ学習データと目的変数、バリデーションデータと目的変数

```
from sklearn.model_selection import KFold
from sklearn.metrics import log_loss
import xgboost as xgb

# 学習データを、さらに学習データとバリデーションデータに分割
kf = KFold(n_splits=4, shuffle=True, random_state=777)
tr_idx, va_idx = list(kf.split(train_x))[0]
tr_x, va_x = train_x.iloc[tr_idx], train_x.iloc[va_idx]
tr_y, va_y = train_y.iloc[tr_idx], train_y.iloc[va_idx]

# xgboost 用のデータ構造に変換
dtrain = xgb.DMatrix(tr_x, label=tr_y)
dvalid = xgb.DMatrix(va_x, label=va_y)
dtest = xgb.DMatrix(test_x)
```



バリデーション: Hold-out 法

- データの変換 → 学習の実行 → 結果の確認(指標はlogloss) → 正解率の計算

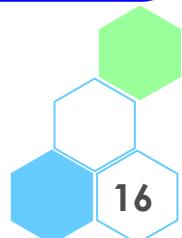
```
# ハイパーパラメータの設定
params      = {'objective': 'binary:logistic',
               'random_state': 777}
num_round  = 200

# 学習の実行、バリデーションデータもモデルに指定、スコアをモニタリング
# 引数 watchlist に学習データとバリデーションデータをセット
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]
model = xgb.train(params, dtrain, num_round, evals=watchlist)

# バリデーションデータでのスコアの確認
va_pred = model.predict(dvalid)
score = log_loss(va_y, va_pred)
print(f'logloss: {score:.5f}')

# 予測値
pred      = model.predict(dtest)
pred_label = np.where(pred > 0.5, 1, 0)
```

logloss: 0.75291
kaggle のコンペの
score: 0.74880

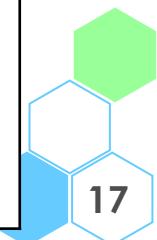


バリデーション: Hold-out 法

- xgboost のアルゴリズムの概要
 - 決定木によるブースティングを行う
 - 正則化された目的関数によりモデルの複雑化を防止(目的関数を変えることで、回帰問題にも、2 値／多値の分類問題にも対応できる)
 - 決定木の作成(葉の最適な重みと分岐を探索・調整し目的関数の減少を図る)
 - 他にも、過学習の防止、データ数の少ない葉を構成しない、深さの制限などが裏で稼働
- 目的関数
 - 回帰: reg:squarederror (平均二乗誤差を最小化するよう学習)
 - 2値の分類: binary:logistic (loglossを最小化するよう学習)
 - 多値の分類: multi:softprob (multiclass loglossを最小化するよう学習)
- early_stopping_rounds=50 で、50 回くり返しても指標が改善しない場合に学習を中止

```
# モニタリングをlogloss、early_stopping_roundsを設定
params = {'objective':'binary:logistic', 'random_state':777,
          'eval_metric': 'logloss'}
num_round = 200
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]
model     = xgb.train(params, dtrain, num_round, evals=watchlist,
                      early_stopping_rounds=50)

# 最適な決定木の本数で予測 (ntree_limitを指定しないと最適時の予測にならない)
pred = model.predict(dtest, ntree_limit=model.best_ntree_limit)
```



バリデーション: クロスバリデーション(CV)

- fold 数は 4~5 回程度が一般的

```

from    sklearn.model_selection import KFold
from    sklearn.metrics           import log_loss
import xgboost as xgb

params    = {'objective': 'binary:logistic',
            'random_state': 777}
num_round = 200
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]
scores    = []

kf = KFold(n_splits=4, shuffle=True, random_state=777)
for tr_idx, va_idx in kf.split(train_x):
    tr_x, va_x = train_x.iloc[tr_idx], train_x.iloc[va_idx]
    tr_y, va_y = train_y.iloc[tr_idx], train_y.iloc[va_idx]
    dtrain     = xgb.DMatrix(tr_x, label=tr_y)
    dvalid     = xgb.DMatrix(va_x, label=va_y)
    model      = xgb.train(params, dtrain, num_round,
                           evals=watchlist)
    va_pred    = model.predict(dvalid)
    score      = log_loss(va_y, va_pred)
    scores.append(score)

# 各スコア(logloss)の平均
print(np.mean(scores))

```

logloss: 0.62284
 kaggle のコンペの
 score: 0.73684

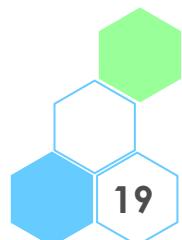
パラメータチューニング

- クロスバリデーション等で指標を確認しながらパラメータチューニングして精度を上げる
- ちなみに、精度があまり良くない DecisionTreeClassifier() でもチューニングで精度が上がる

```
# 9頁（前処理）の「学習用データとテストデータに分割」の前で
# Age、Fare の欠測を中央値で補完（xgboost以外は欠測を許容しないため）
df.Age = df.Age.fillna(df.Age.median())
df.Fare = df.Fare.fillna(df.Fare.median())

# CART
from sklearn.tree import DecisionTreeClassifier, plot_tree
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(train_x, train_y.values.ravel())
pred = clf.predict(test_x)
pred_label = np.where(pred > 0.5, 1, 0)
```

kaggle のコンペの
score: 0.74880



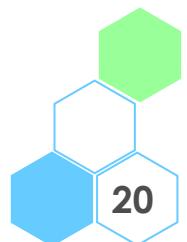
パラメータチューニング: *Grid Search*

- GridSearchCV にてパラメータチューニング

```
from sklearn.model_selection import GridSearchCV
import xgboost as xgb

model = xgb.XGBClassifier()
params = {'booster': ['gbtree'],
          'n_estimators': [500],
          'objective': ['binary:logistic'],
          'learning_rate': [0.05, 0.1],
          'max_depth': [3, 4, 5, 6, 7, 8, 9],
          'min_child_weight': [0.1, 1, 10, 100],
          'colsample_bytree': [0.6, 0.8, 0.95, 1],
          'colsample_bylevel': [0.1, 0.3, 0.5],
          'gamma': [0, 0.001, 0.01, 0.1, 1],
          'subsample': [0.7, 0.8, 0.9],
          'alpha': [0.0], 'lambda': [1.0], 'random_state': [777]
         }
gs = GridSearchCV(model, param_grid=params, cv=4, n_jobs=-1,
                  scoring='accuracy')
gs.fit(train_x, train_y)
print(gs.best_params_)
print(gs.best_score_)
```

```
{'alpha': 0.0, 'booster': 'gbtree', 'colsample_bylevel': 0.5,
 'colsample_bytree': 0.95, 'gamma': 0, 'lambda': 1.0, 'learning_rate':
 0.1, 'max_depth': 3, 'min_child_weight': 10, 'n_estimators': 500,
 'objective': 'binary:logistic', 'random_state': 777, 'subsample': 0.9}
0.8451096836747061
```



パラメータチューニング: *Grid Search*

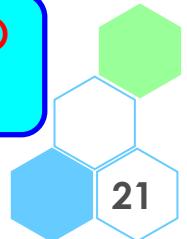
- 前頁の best parameters を使用して予測

```
dtrain      = xgb.DMatrix(train_x, label=train_y)
dtest       = xgb.DMatrix(test_x)

# n_estimators→num_round、learning_rate→eta
num_round   = 500 # モニタリングし、過学習にならない200を選択
params      = {'alpha': 0.0, 'booster': 'gbtree',
               'colsample_bylevel': 0.5, 'colsample_bytree': 0.95,
               'gamma': 0, 'lambda': 1.0, 'eta': 0.1, 'max_depth': 3,
               'min_child_weight': 10, 'objective': 'binary:logistic',
               'random_state': 777, 'subsample': 0.9}

model       = xgb.train(params, dtrain, num_round)
pred        = model.predict(dtest)
pred_label  = np.where(pred > 0.5, 1, 0)
```

kaggle のコンペの
score: 0.77272



パラメータチューニング: *Grid Search*

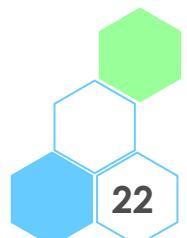
- GridSearchCV にて評価スコアを logloss に変更

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import log_loss, make_scorer
import xgboost as xgb

model = xgb.XGBClassifier()
params = {'booster': ['gbtree'],
           'n_estimators': [200],
           'objective': ['binary:logistic'],
           'learning_rate': [0.05, 0.1],
           'max_depth': [3, 4, 5, 6, 7, 8, 9],
           'min_child_weight': [0.1, 1, 10, 100],
           'colsample_bytree': [0.6, 0.8, 0.95, 1],
           'colsample_bylevel': [0.1, 0.3, 0.5],
           'gamma': [0, 0.001, 0.01, 0.1, 1],
           'subsample': [0.7, 0.8, 0.9],
           'alpha': [0.0],
           'lambda': [1.0],
           'random_state': [777]
          }

logloss = make_scorer(log_loss, greater_is_better=False,
                      needs_proba=True)
gs = GridSearchCV(model, param_grid=params, cv=4, n_jobs=-1,
                  scoring=logloss)
gs.fit(train_x, train_y)
print(gs.best_params_)
print(gs.best_score_)
```

kaggle のコンペの
score: 0.77272



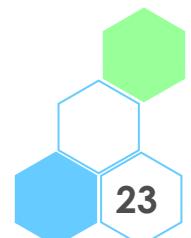
パラメータチューニング: お試しで *RandomForest*

```
# 「学習用データとテストデータに分割」の前で Age、Fare の欠測を中心値で補完
df.Age = df.Age.fillna(df.Age.median())
df.Fare = df.Fare.fillna(df.Fare.median())

from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
params = {'bootstrap': [True],
          'n_estimators': [100, 300, 500, 1000],
          'max_features': ['auto', 'sqrt'],
          'max_depth': [5, 10, 20, 50, 100],
          'min_samples_leaf': [1, 2, 4],
          'min_samples_split': [2, 5, 10],
          'criterion': ['gini', 'entropy']}
}
gs = GridSearchCV(model, param_grid=params, cv=4, n_jobs=-1,
                  scoring='accuracy')
gs.fit(train_x, train_y.values.ravel())
print(gs.best_params_) # 下記赤字がベストなパラメータ
print(gs.best_score_)

clf = RandomForestClassifier(bootstrap=True,
                            criterion='entropy', max_depth=10, max_features='auto',
                            min_samples_leaf=2, min_samples_split=5, n_estimators=300)
clf.fit(train_x, train_y.values.ravel())
pred = clf.predict(test_x)
pred_label = np.where(pred > 0.5, 1, 0)
```

kaggle のコンペの
score: 0.77990



パラメータチューニング: ベイズ最適化

【参考資料】Yassine Alouini氏

Hyperopt the Xgboost model: <https://www.kaggle.com/yassinealouini/hyperopt-the-xgboost-model>

```
from sklearn.model_selection import KFold
from sklearn.metrics import log_loss
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
import xgboost as xgb

def score(params):
    num_round = int(params['n_estimators'])
    kf = KFold(n_splits=4, shuffle=True, random_state=777)
    scores = []
    del params['n_estimators']
    for tr_idx, va_idx in kf.split(train_x):
        tr_x, va_x = train_x.iloc[tr_idx], train_x.iloc[va_idx]
        tr_y, va_y = train_y.iloc[tr_idx], train_y.iloc[va_idx]
        dtrain = xgb.DMatrix(tr_x, label=tr_y)
        dvalid = xgb.DMatrix(va_x, label=va_y)
        model = xgb.train(params, dtrain, num_round)
        va_pred = model.predict(dvalid, ntree_limit=model.best_iteration+1)
        score = log_loss(va_y, va_pred) # 適当な評価関数に変更
        scores.append(score)
    loss = np.mean(scores) # 値が大きい方が良い指標である場合は (-1) を掛ける
    return {'loss': loss, 'status': STATUS_OK}
```

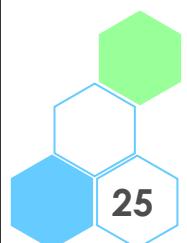


パラメータチューニング: ベイズ最適化

- `hp.choice()`: 複数の選択肢から選ぶ
- `hp.uniform()`: 一様分布から抽出(引数: 下限、上限)
- `hp.quniform()`: 一様分布のうち間隔ごとの値から抽出(引数: 下限、上限、間隔)
- `hp.loguniform()`: 一様分布から抽出(引数: 下限、上限、対数をとった値が指定される)

```
def optimize(trials):
    space = {'seed': 777,
              'booster': 'gbtree',
              'objective': 'binary:logistic',
              'eval_metric': 'error',
              'n_estimators': hp.quniform('n_estimators', 100, 1000, 1),
              'eta': hp.quniform('eta', 0.025, 0.5, 0.025),
              'max_depth': hp.choice('max_depth', np.arange(2, 10, dtype=int)),
              'min_child_weight': hp.loguniform('min_child_weight', np.log(0.1), np.log(10)),
              'subsample': hp.quniform('subsample', 0.6, 0.95, 0.05),
              'colsample_bytree': hp.quniform('colsample_bytree', 0.6, 0.95, 0.05),
              'colsample_bylevel': hp.quniform('colsample_bylevel', 0.1, 0.5, 0.05),
              'gamma': hp.loguniform('gamma', np.log(1e-8), np.log(1.0)),
              'alpha' : hp.loguniform('alpha', np.log(1e-8), np.log(1.0)),
              'lambda' : hp.loguniform('lambda', np.log(1e-6), np.log(10.0))
    }
    best = fmin(score, space, algo=tpe.suggest, trials=trials,
                max_evals=25)
    return best # 探索回数は25回程度でもOK、100回あれば十分探索可能

trials = Trials()
best   = optimize(trials)
print(best)
```



パラメータチューニング: ベイズ最適化

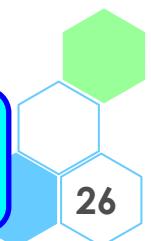
- 前回の best parameters を使用して予測
 - あたりを付ける際はベイズ最適化、その後の探索は Grid Search ?

```
dtrain      = xgb.DMatrix(train_x, label=train_y)
dtest       = xgb.DMatrix(test_x)

# n_estimators→num_round
num_round   = 667
params      = {'alpha': 0.00021675703317206054,
               'colsample_bylevel': 0.15000000000000002,
               'colsample_bytree': 0.65, 'eta': 0.07500000000000001,
               'gamma': 1.4388954604198139e-05,
               'lambda': 8.69794923455225e-05, 'max_depth': 3,
               'min_child_weight': 7.433352853051877,
               'subsample': 0.7000000000000001}

model       = xgb.train(params, dtrain, num_round)
pred        = model.predict(dtest)
pred_label = np.where(pred > 0.5, 1, 0)
```

kaggle のコンペの
score: 0.77033



メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- その他

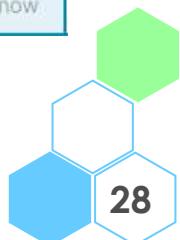
※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

XGBoost: 変数(特徴量)の選択・作成

- 予測を行う場合、まず XGBoost で行うのは悪くない
 - 精度が高く、変数変換が必ずしも必要ない上、欠測があっても動作する
- 「説明変数(特徴量)の選択・作成」→「ハイパーパラメータの選択」→「モデルの学習・クロスバリデーションによる評価」をくり返す
 - **上記作業のうち「説明変数の選択・作成」が最も重要**
 - 作業の終盤で、モデルの変更(XGBoostと他のモデルのアンサンブル等)やハイパーパラメータの最終調整を行う
- ということで、「変数の選択・作成」をやり直してみる
 - 今回の titanic データの場合、8 ~ 9 頁のデータのままでは精度は頭打ち、各説明変数と「生存/死亡」の関連をちゃんと調べたうえで、変数の選択・作成を行う
 - 下記サイトでは更に変数を作りこんでいるが、ここではそこまで踏み込まない…
当方は 0.787 で頭打ちでした

2313 NobuoFunao 0.78708 66 now

- Kaggle チュートリアル Titanic で上位 2% 以内に入るノウハウ
<https://qiita.com/jun40vn/items/d8a1f71fae680589e05c>



XGBoost: 変数(特徴量)の選択・作成

- 各変数の欠測状況 → Fare で 1 例欠測、Age と Cabin で欠測多数

```
# train.csv と test.csv の結合
df1 = pd.read_csv('C:/py/titanic/train.csv', header=0)
df0 = pd.read_csv('C:/py/titanic/test.csv', header=0)
df1["Is_train"] = 1
df0["Is_train"] = 0
df = pd.concat([df1, df0])

# 各変数の欠測状況
print(df.info())
```

#	Column	Non-Null Count	Dtype
0	PassengerId	1309 non-null	int64
1	Survived	891 non-null	float64
2	Pclass	1309 non-null	int64
3	Name	1309 non-null	object
4	Sex	1309 non-null	object
5	Age	1046 non-null	float64
6	SibSp	1309 non-null	int64
7	Parch	1309 non-null	int64
8	Ticket	1309 non-null	object
9	Fare	1308 non-null	float64
10	Cabin	295 non-null	object
11	Embarked	1307 non-null	object
12	Is_train	1309 non-null	int64



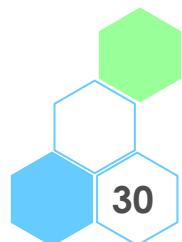
XGBoost: 変数(特徴量)の選択・作成

- Fare の 1 例欠測を、Pclass==3 での中央値(8)で補完

```
print(df[df.Fare.isnull()])      # 欠測レコードの情報 : Pclass==3 (中央値は8)
print(df['Fare'].groupby(df['Pclass']).describe()) # 各Pclassの運賃
df.Fare = df.Fare.apply(lambda x : x if np.isnan(x)==False else 8)
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	...
1044	NaN	3	Storey, Mr. Thomas	male	60.5	0	
Parch	Ticket	Fare	Cabin	Embarked	Is_train		
0	3701	NaN	NaN	S	0		

Pclass	count	mean	std	min	25%	50%	75%	max
1	323.0	87.508992	80.447178	0.0	30.6958	60.0000	107.6625	512.3292
2	277.0	21.179196	13.607122	0.0	13.0000	15.0458	26.0000	73.5000
3	708.0	13.302889	11.494358	0.0	7.7500	<u>8.0500</u>	15.2458	69.5500

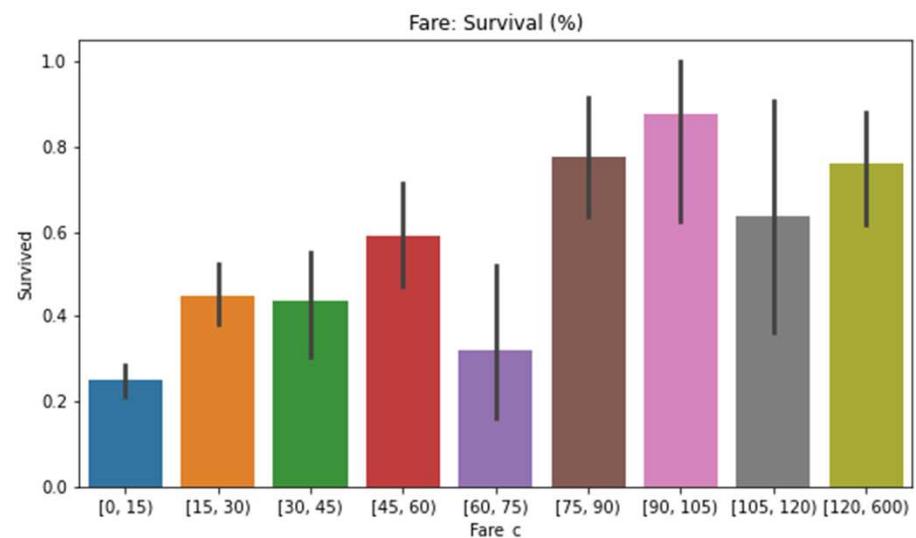
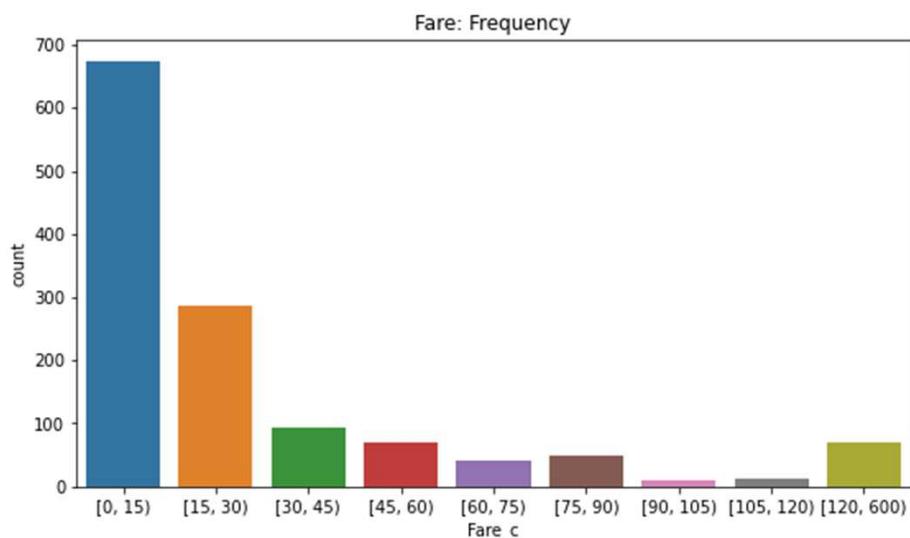


XGBoost: 変数(特徴量)の選択・作成

- Fare と生存割合の関係を調査

```
print(df.Fare.describe())
df['Fare_c'] = pd.cut(df.Fare, [0, 15, 30, 45, 60, 75, 90, 105, 120, 600],
right=False) # 0~80まで、labelsでラベル付与
print(df.Fare_c.value_counts(sort=False, dropna=False)) # 結果は省略

plt.figure(figsize=(20, 10))
plt.subplot(1, 2, 1)
plt.title("Fare: Frequency")
sns.countplot(data=df, x='Fare_c')
plt.subplot(1, 2, 2)
plt.title("Fare: Survival (%)")
sns.barplot(data=df, x='Fare_c', y='Survived')
plt.show()
```



XGBoost: 変数(特徴量)の選択・作成

- 'Mr.': 男性
- 'Mrs.': 既婚女性
- 'Miss.': 未婚女性
- 'Ms.': 女性
- 'Master.': 少年、青年男性
- 'Don.': 男性の高位・年配の人(スペイン)
- 'Dona.': 女性の高位・年配の人(スペイン)
- 'Rev.': 牧師・聖職者(Reverend)
- 'Dr.': 医者
- 'Mme.': 既婚女性(Madame、フランス)
- 'Mlle.': 未婚女性(Mademoiselle、フランス)
- 'Major.': 軍人(少佐)
- 'Lady.': 高貴な女性
- 'Sir.': 高貴な男性
- 'Col.': 軍人(Colonel、大佐)
- 'Capt.': 船長(Captain)
- 'the': 高貴な女性(the Countessの文字切れ)
- 'Jonkheer.': 高貴な男性(オランダ、ベルギー)

```
# Name について : スペース区切りで2つ目の要素が敬称 (Title)
```

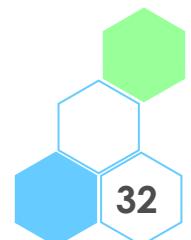
```
df['Title'] = df['Name'].apply(lambda x:  
x.split(',') [1]).apply(lambda x: x.split() [0])  
print(df.Title.unique())
```

```
# "the" は "the Countess." の一部
```

```
print(df[df.Name.str.contains('the ')])
```

```
['Mr.' 'Mrs.' 'Miss.' 'Master.' 'Don.' 'Rev.' 'Dr.' 'Mme.' 'Ms.' 'Major.'  
'Lady.' 'Sir.' 'Mlle.' 'Col.' 'Capt.' 'the' 'Jonkheer.' 'Dona.]
```

Name	Sex	Age	SibSp	¥
473 Jerwan, Mrs. Amin S (Marie Marthe Thuillard)	female	23.0		0
710 Mayne, Mlle. Berthe Antonine ("Mrs de Villiers")	female	24.0		0
759 Rothes, the Countess. of (Lucy Noel Martha Dye...)	female	33.0		0



XGBoost: 変数(特徴量)の選択・作成

- Title と生存割合の関係を調査

```

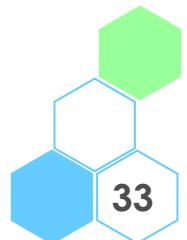
print(df.Title.value_counts()) # 各Titleの頻度、結果は省略
print(df[['Title', 'Survived']].groupby(['Title'],
as_index=False).mean()) # 各Titleの生存割合、結果は省略
print(pd.crosstab(df.Title, df.Survived, margins=True)) # クロス表

plt.figure(figsize=(12, 10))
plt.subplot(2, 1, 1)
plt.title("Title: Frequency")
sns.countplot(data=df, x='Title')
plt.subplot(2, 1, 2)
plt.title("Title: Survival (%)")
sns.barplot(data=df, x='Title', y='Survived')
plt.show()

```

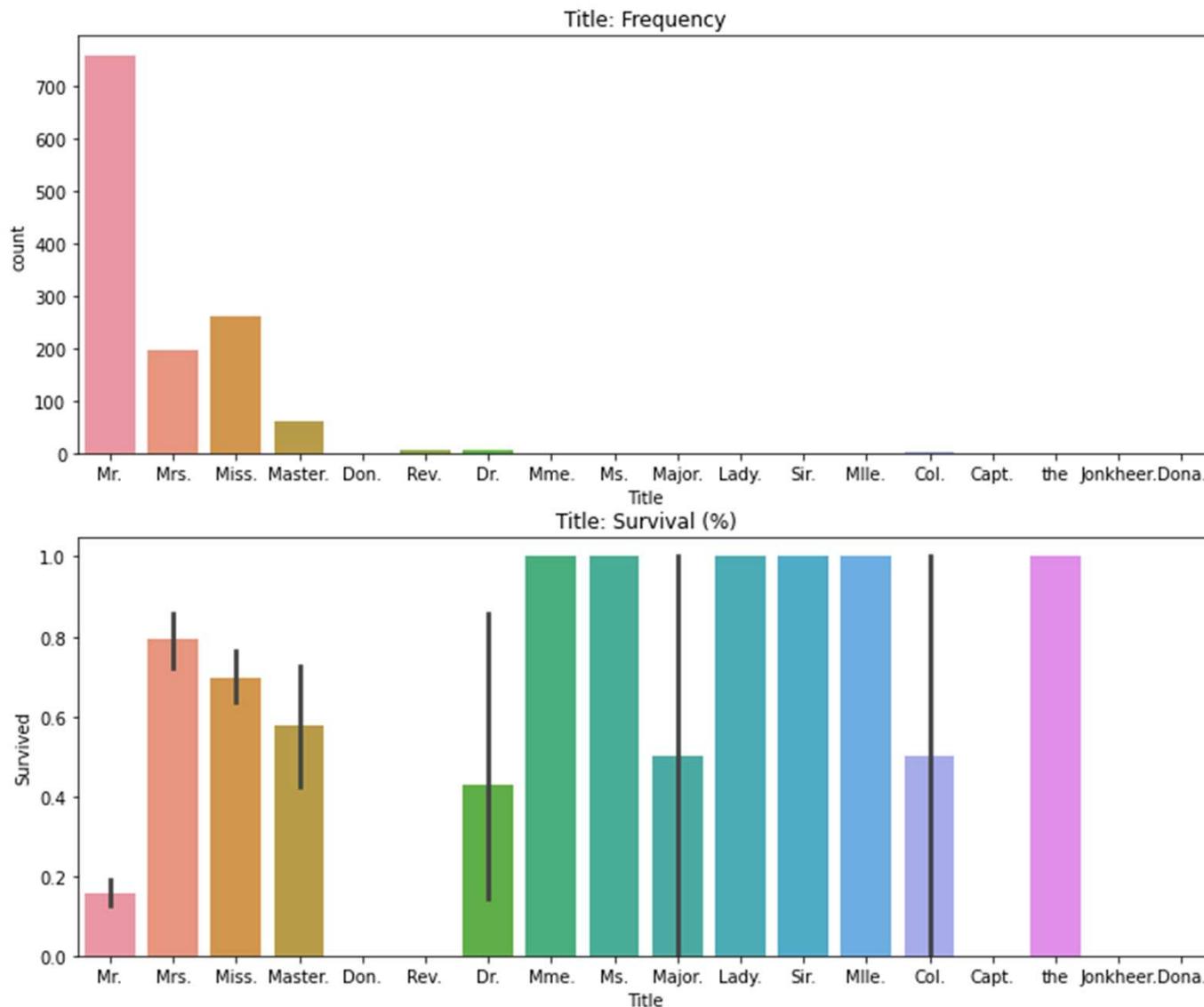
Survived	0.0	1.0	All
Title			
Capt.	1	0	1
Col.	1	1	2
Don.	1	0	1
Dr.	4	3	7
Jonkheer.	1	0	1
Lady.	0	1	1
Major.	1	1	2
Master.	17	23	40
Miss.	55	127	182
Mlle.	0	2	2

Survived	0.0	1.0	All
Title			
Mme.	0	1	1
Mr.	436	81	517
Mrs.	26	99	125
Ms.	0	1	1
Rev.	6	0	6
Sir.	0	1	1
the	0	1	1
All	549	342	891



XGBoost: 変数(特徴量)の選択・作成

- Title と生存割合の関係を調査

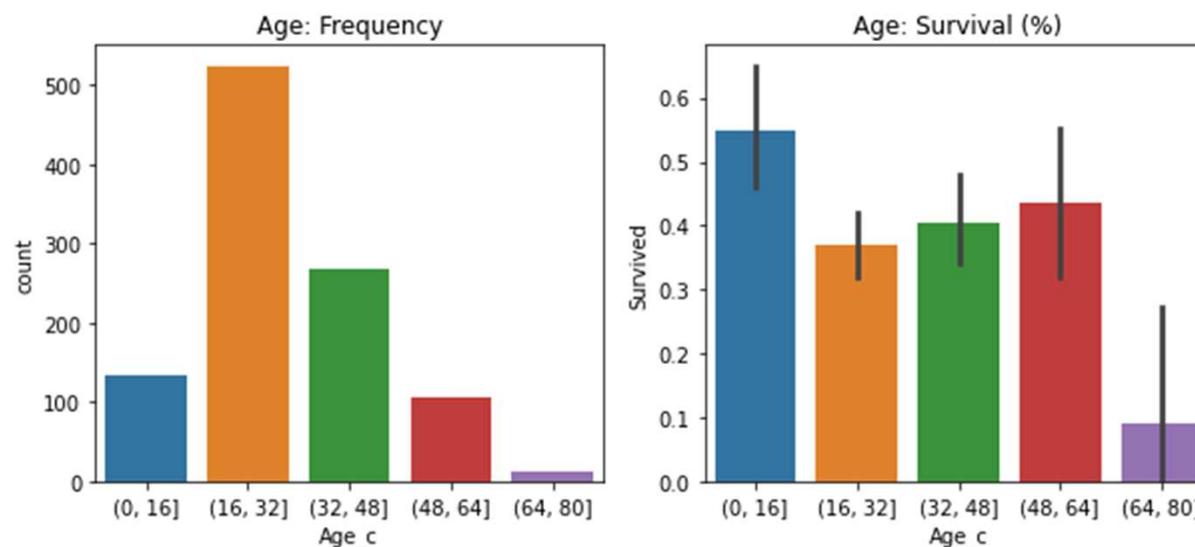


XGBoost: 変数(特徴量)の選択・作成

- Age と生存割合の関係を調査

```
print(df.Age.describe())
df['Age_c'] = pd.cut(df.Age, range(0, 90, 16), right=True) # 0~80まで
print(df.Age_c.value_counts(sort=False, dropna=False)) # 結果は省略

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("Pclass: Frequency")
sns.countplot(data=df, x='Age_c')
plt.subplot(1, 2, 2)
plt.title("Pclass: Survival (%)")
sns.barplot(data=df, x='Age_c', y='Survived')
plt.show()
```

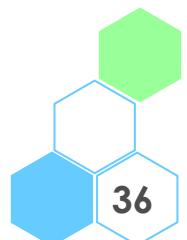


XGBoost: 変数(特徴量)の選択・作成

- Age の欠測補完>Titleが若者である場合は15歳、それ以外は30歳)
⇒ Age_imp へ
- Title について、生存率の高いカテゴリに大きめの数値を割り当て

```
# Age について
def f(x):
    if np.isnan(x.Age)==False:
        return x.Age
    else:
        if x.Title in ['Miss.', 'Master.', 'Mlle.']:
            return 15
        else:
            return 30
df['Age_imp'] = df.apply(lambda x: f(x), axis=1)

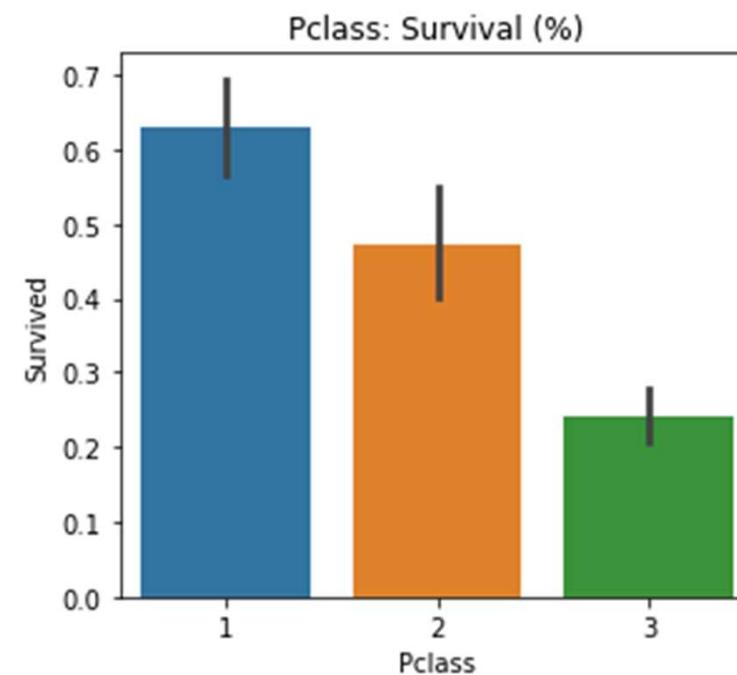
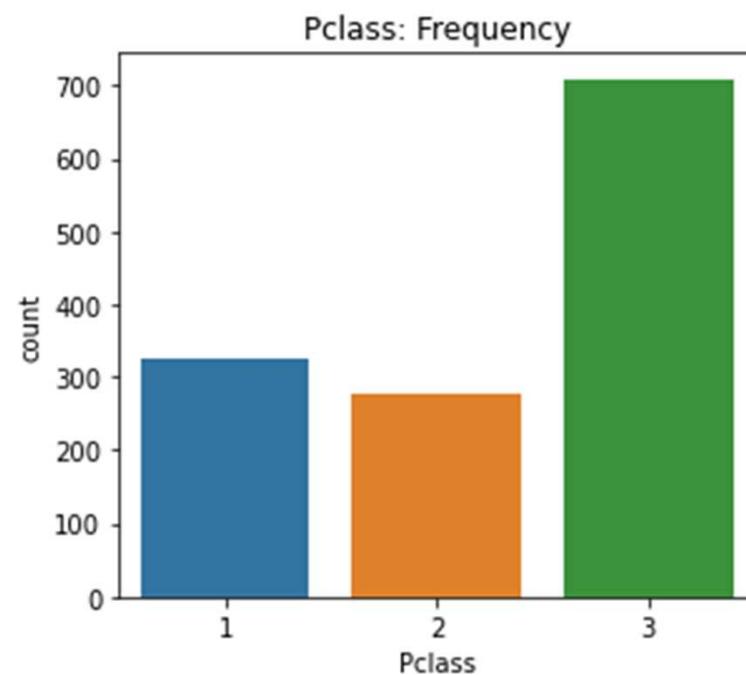
# Title について
def f(x):
    if x in ['Don.', 'Rev.', 'Capt.', 'Jonkheer.', 'Dona.']:
        return 0
    elif x == 'Mr.':
        return 1
    elif x in ['Mrs.', 'Miss.', 'Master.', 'Dr.', 'Major.', 'Col.']:
        return 2
    else: # x in ['Mme.', 'Ms.', 'Lady.', 'Sir.', 'Mlle.', 'the']
        return 3
df['Title_c'] = df['Title'].apply(f)
```



XGBoost: 変数(特徴量)の選択・作成

- Pclass と生存割合の関係を調査

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("Pclass: Frequency")
sns.countplot(data=df, x='Pclass')
plt.subplot(1, 2, 2)
plt.title("Pclass: Survival (%)")
sns.barplot(data=df, x='Pclass', y='Survived')
plt.show()
```

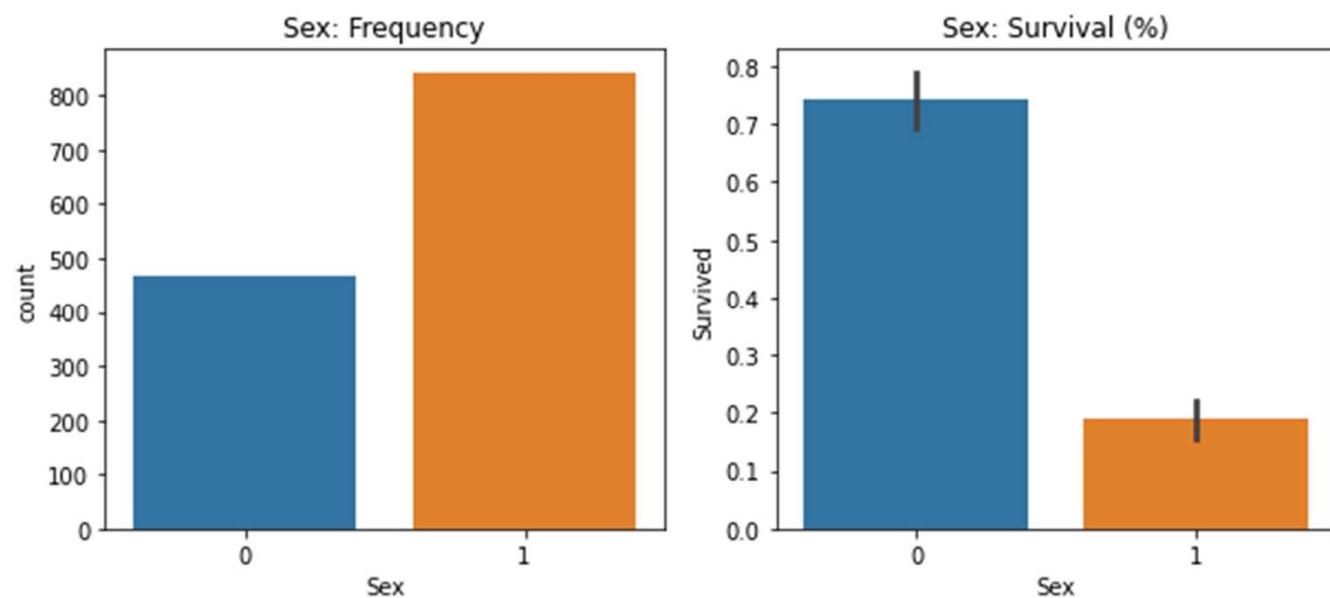


XGBoost: 変数(特徴量)の選択・作成

- Sex を数値化、生存割合の関係を調査

```
df.Sex = df.Sex.apply(lambda x : 1 if x == 'male' else 0)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("Sex: Frequency")
sns.countplot(data=df, x='Sex')
plt.subplot(1, 2, 2)
plt.title("Sex: Survival (%)")
sns.barplot(data=df, x='Sex', y='Survived')
plt.show()
```



XGBoost: 変数(特徴量)の選択・作成

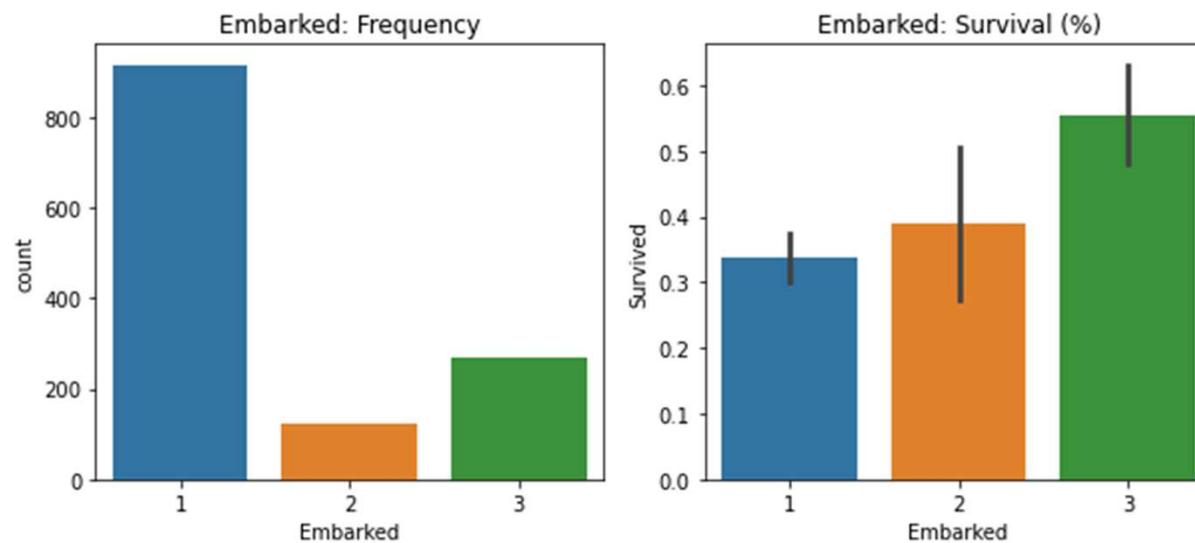
- Embarked を数値化、生存割合の関係を調査

```

my_mapping = {'S':1, 'Q':2, 'C':3}          # 生存率の順で
df.Embarked = df.Embarked.replace(my_mapping) # NaN は一旦放置
df.Embarked = df.Embarked.fillna(1)         # NaN を 1 に置換
df.Embarked = df.Embarked.astype('int8')      # 整数型に変換

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("Embarked: Frequency")
sns.countplot(data=df, x='Embarked')
plt.subplot(1, 2, 2)
plt.title("Embarked: Survival (%)")
sns.barplot(data=df, x='Embarked', y='Survived')
plt.show()

```



XGBoost: 変数(特徴量)の選択・作成

- その他の処理

```

### SibSp と Parch から家族数 Family を計算
df['Family'] = df.SibSp + df.Parch

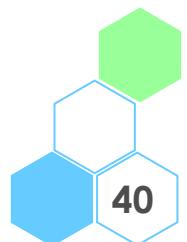
### Ticket について、同じ Ticket を持っている数 Ticket_n を生成
df_tmp = df[['Ticket','PassengerId']].groupby(['Ticket'],
as_index=False).count()
df_tmp.columns = ['Ticket', 'Ticket_n']
df = pd.merge(df, df_tmp, on='Ticket', how='left')

le = LabelEncoder()
le.fit(df['Ticket'])
df['Ticket'] = le.transform(df['Ticket'])

### 閾値の探索
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

scores = []
kf      = KFold(n_splits=4, shuffle=True, random_state=777)
for i in [0.47, 0.48, 0.49, 0.50, 0.51, 0.52, 0.53, 0.54, 0.55]:
    for tr_idx, va_idx in kf.split(train_x):
        tr_x, va_x = train_x.iloc[tr_idx], train_x.iloc[va_idx]
        tr_y, va_y = train_y.iloc[tr_idx], train_y.iloc[va_idx]
        dtrain     = xgb.DMatrix(tr_x, label=tr_y)
        dvalid    = xgb.DMatrix(va_x, label=va_y)
        model     = xgb.train(params, dtrain, num_round)
        va_pred   = model.predict(dvalid)
        pred_label = np.where(va_pred > i, 1, 0)
        score     = accuracy_score(va_y, pred_label)
        scores.append(score)
print('threshold: %.2f, %.5f' % (i, np.mean(scores)))

```



XGBoost: 変数(特徴量)の選択・作成

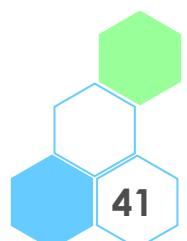
- 変数同士の相関や変数の重要度を確認

```
# 変数全体の相関行列
r = df.corr() # method='spearman'でも確認
plt.figure(figsize=(15, 10))
sns.heatmap(r, annot=True, fmt='.2f', cmap='Blues', vmin=-1, vmax=1)
plt.show()

# Survived に関連がありそうな変数に絞って相関行列
r = df[['Survived', 'Pclass', 'Sex', 'Fare', 'Title_c', 'Embarked']].corr()
plt.figure(figsize=(10, 10))
sns.heatmap(r, annot=True, fmt='.2f', cmap='Blues', vmin=-1, vmax=1)
plt.show()

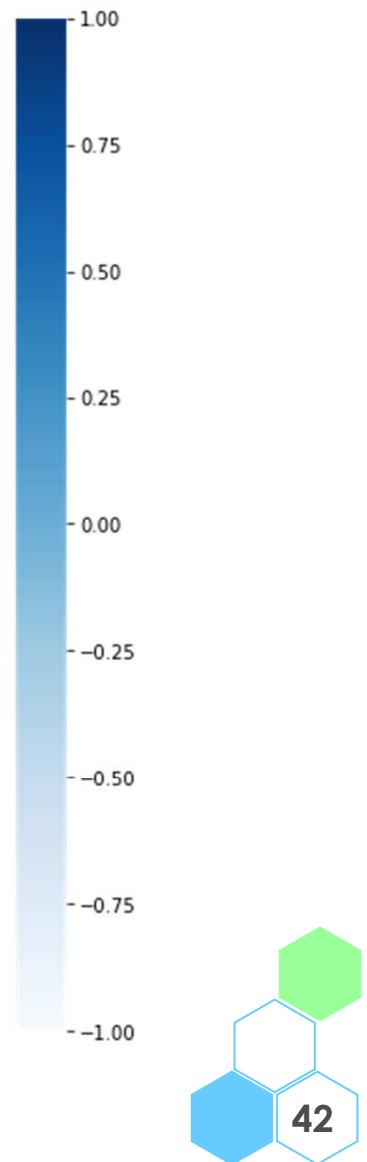
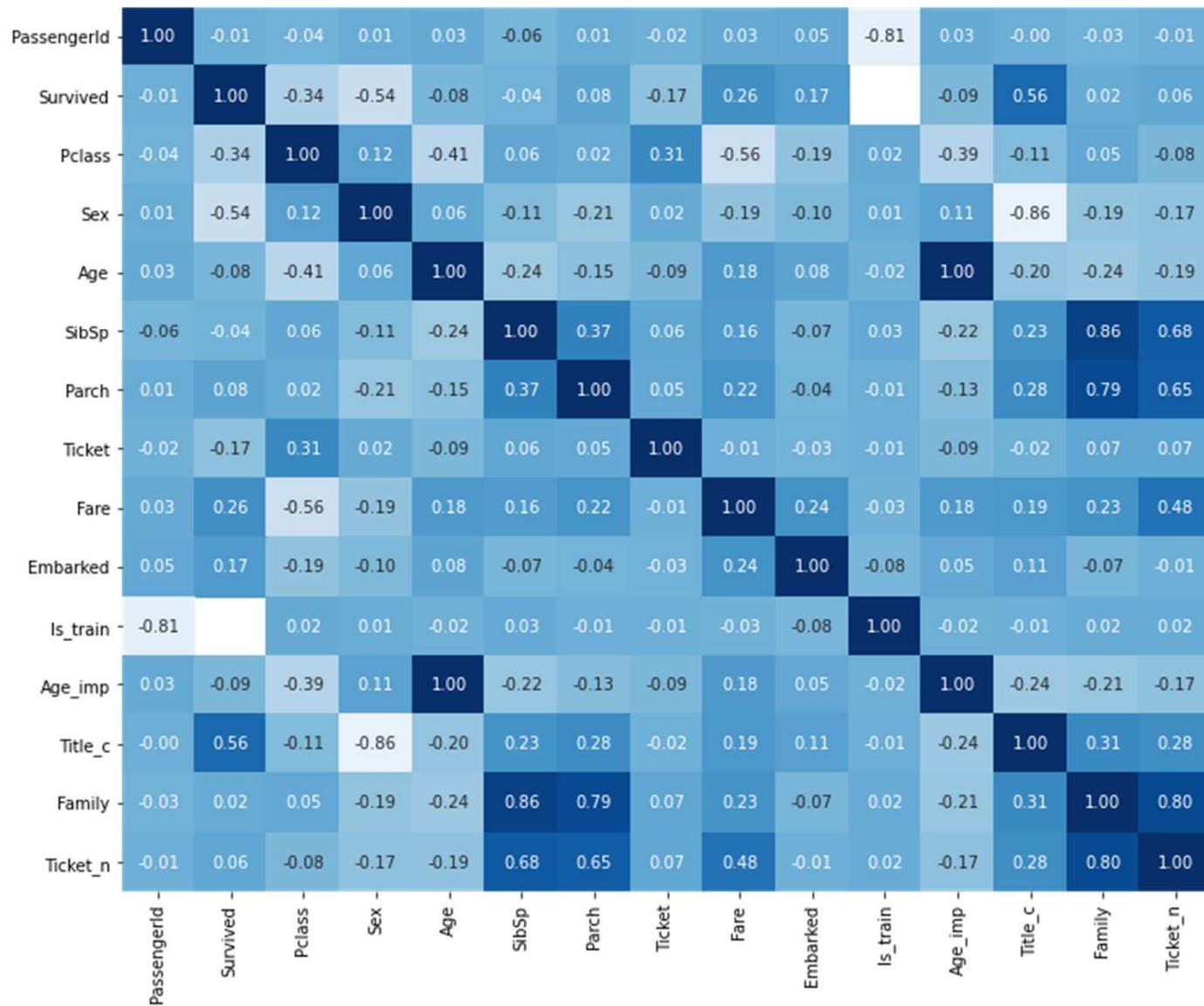
train_x = df.query('Is_train == 1')[['Pclass', 'Sex', 'Fare', 'Embarked', 'Title_c', 'Age_imp', 'Family', 'Ticket', 'Ticket_n']]
train_y = df.query('Is_train == 1')['Survived']
test_x = df.query('Is_train == 0')[['Pclass', 'Sex', 'Fare', 'Embarked', 'Title_c', 'Age_imp', 'Family', 'Ticket', 'Ticket_n']]
id = df.query('Is_train == 0')['PassengerId']
dtrain = xgb.DMatrix(train_x, label=train_y)
dtest = xgb.DMatrix(test_x)
num_round = 200
params = {'alpha': 0.0, 'booster': 'gbtree', 'colsample_bylevel': 0.5, 'colsample_bytree': 0.8, 'gamma': 1, 'lambda': 1.0, 'eta': 0.1, 'max_depth': 7, 'min_child_weight': 0.1, 'objective': 'binary:logistic', 'random_state': 777, 'subsample': 0.7}

# とりあえず変数の重要度を表示（この後、変数を絞って再出力する等、いろいろ）
model = xgb.train(params, dtrain, num_round)
xgb.plot_importance(model)
```



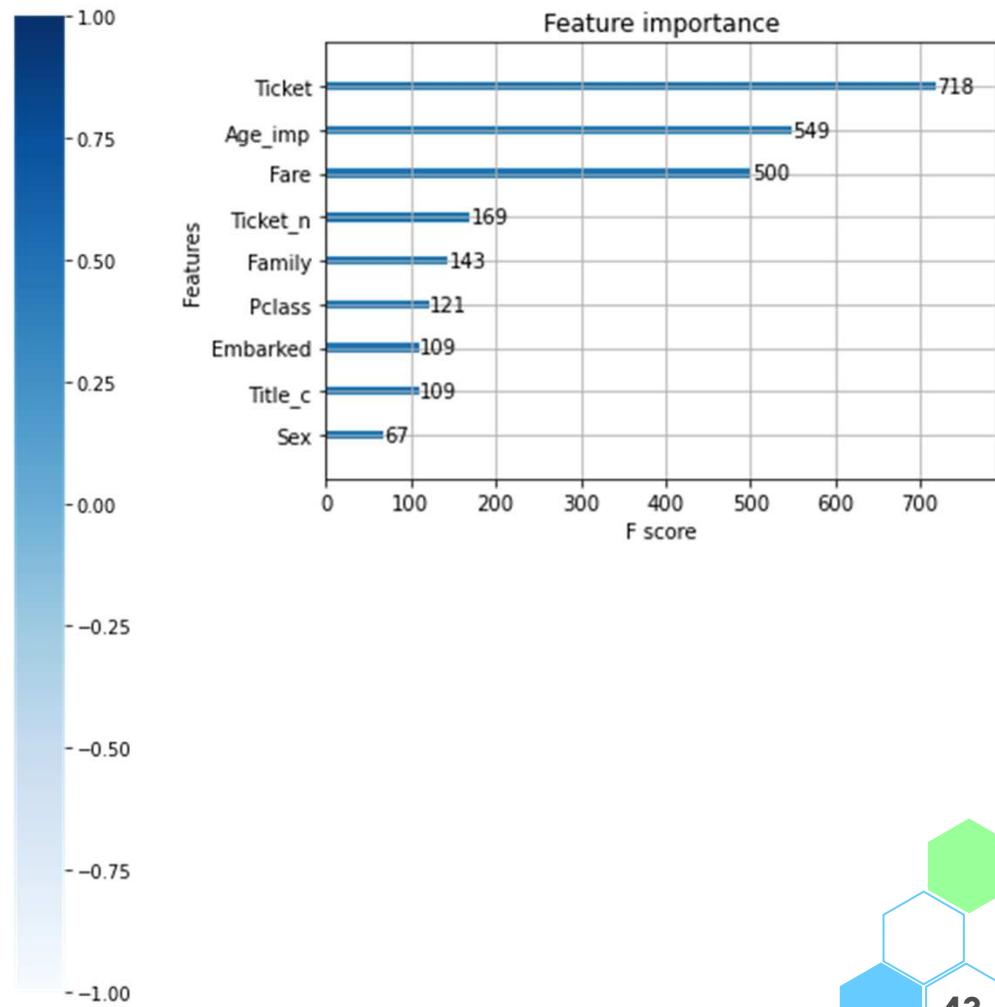
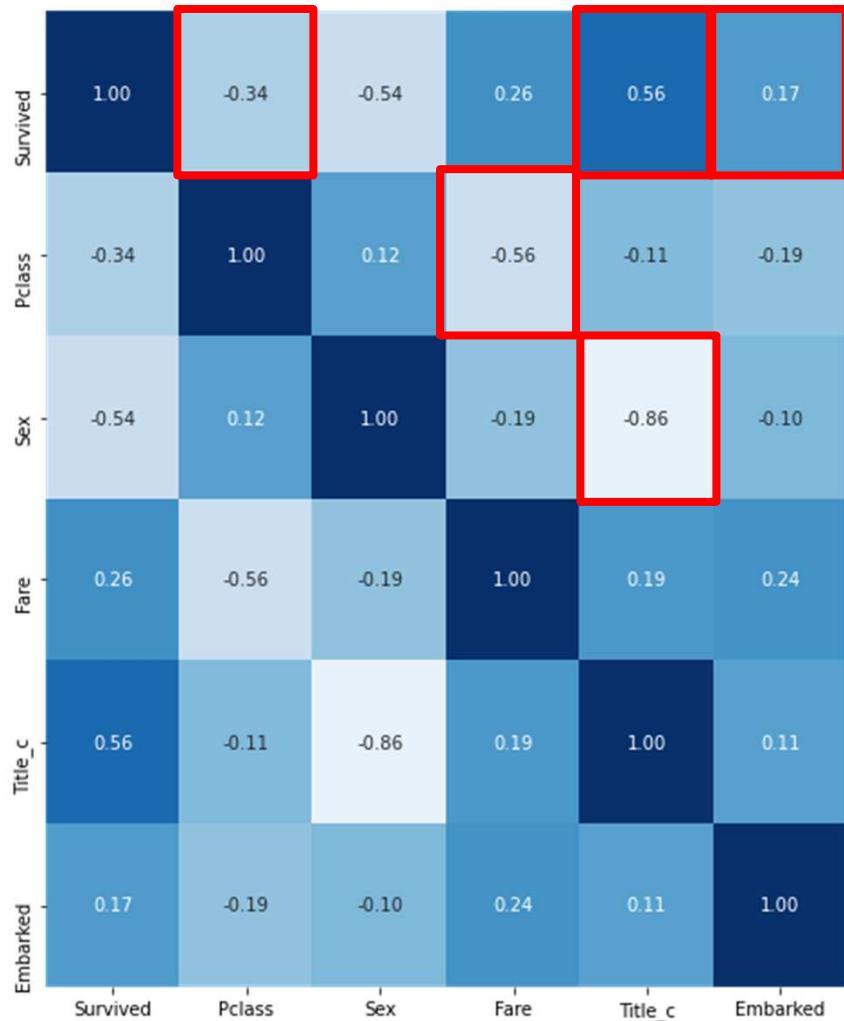
XGBoost: 変数(特徴量)の選択・作成

- 変数同士の相関や変数の重要度を確認



XGBoost: 変数(特徴量)の選択・作成

- 変数同士の相関や変数の重要度を確認
⇒ Pclass、Title_c、Embarked に絞って予測を行うことにする



XGBoost: 変数(特徴量)の選択・作成

- パラメータチューニング後、Pclass、Title_c、Embarked に絞って予測を行う
- Pclass、Title_c、Embarked と関連が高そうな変数(Ticket や Age_imp)を追加して予測を行うと、逆に score は下がった…

```
# 学習用データとテストデータに分割
train_x = df.query('Is_train == 1')[['Pclass', 'Embarked', 'Title_c']]
train_y = df.query('Is_train == 1')['Survived']
test_x = df.query('Is_train == 0')[['Pclass', 'Embarked', 'Title_c']]
id = df.query('Is_train == 0')['PassengerId']

# 予測、提出用データの作成
dtrain = xgb.DMatrix(train_x, label=train_y)
dtest = xgb.DMatrix(test_x)

# n_estimators→num_round、learning_rate→eta
num_round = 250
params = {'alpha': 0.0, 'booster': 'gbtree', 'colsample_bylevel': 0.1, 'colsample_bytree': 0.8, 'gamma': 0, 'lambda': 1.0, 'eta': 0.05, 'max_depth': 3, 'min_child_weight': 0.1, 'objective': 'binary:logistic', 'random_state': 777, 'subsample': 0.7}
model = xgb.train(params, dtrain, num_round)
pred = model.predict(dtest)
pred_label = np.where(pred > 0.5, 1, 0)

out = pd.DataFrame({'PassengerId': id, 'Survived': pred_label})
out.to_csv('C:/py/titanic/gender_submission.csv', index=False)
```

kaggle のコンペの
score: 0.78708



XGBoost: おまけ、閾値を微修正

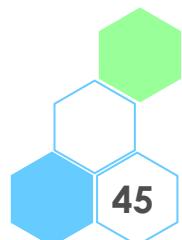
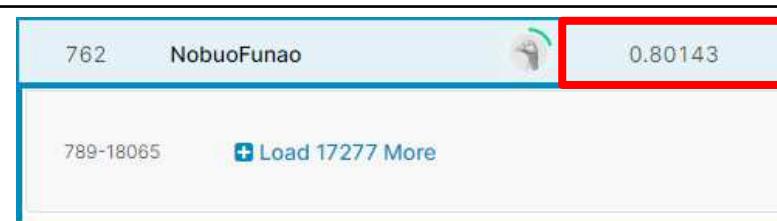
- score が 80% を超えると良いらしいので、44 頁の結果を使って小細工

```
# 生存率 ⇒ 生存割合は 4 割弱、閾値を 0.5 にしてしまうと生存者を多めに設定？
df.Survived.hist()

# 予測値（0-1変換前）⇒ 0.42～0.7 の間のデータは 0.584 ~ 0.603 のデータが
# 約 20 個、これらを死亡にすべく閾値を 0.61 に設定
np.sort(pred)

# 閾値の探索（没）
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
scores = []
kf = KFold(n_splits=10, shuffle=True, random_state=777)
for i in [0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7]:
    for tr_idx, va_idx in kf.split(train_x):
        tr_x, va_x = train_x.iloc[tr_idx], train_x.iloc[va_idx]
        tr_y, va_y = train_y.iloc[tr_idx], train_y.iloc[va_idx]
        dtrain = xgb.DMatrix(tr_x, label=tr_y)
        dvalid = xgb.DMatrix(va_x, label=va_y)
        model = xgb.train(params, dtrain, num_round)
        va_pred = model.predict(dvalid)
        pred_label = np.where(va_pred > i, 1, 0)
        score = accuracy_score(va_y, pred_label)
        scores.append(score)
    print('threshold: %.2f, %.5f' % (i, np.mean(scores)))

# 予測値再計算
pred_label = np.where(pred > 0.61, 1, 0)
out = pd.DataFrame({'PassengerId':id, 'Survived':pred_label})
out.to_csv('C:/py/titanic/gender_submission.csv', index=False)
```



メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- **その他**

※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

予測精度の指標

実際のカテゴリ

予測されたカテゴリ

	Positive	Negative
Positive	True Pos. (TP)	False Neg. (FN)
Negative	False Pos. (FP)	True Neg. (TN)

- Accuracy = $(TP+TN) / (TP+FN+FP+TN)$
- Error = $1 - \text{Accuracy}$
- True positive rate (TPR) = Recall = $TP / (TP+FN)$
- False positive rate (FPR) = $FP / (FP+TN)$
- Precision = $TP / (TP+FP)$
- ROC 曲線の AUC = 横軸 FPR、縦軸 TPR とした曲線の AUC
- Kappa 係数 = $(\text{Accuracy} - Pe) / (1-Pe)$
 - 期待一致率 $Pe = [(TP+FN)(TP+FP) + (TN+FN)(TN+FP)] / (TP+FN+FP+TN)^2$
 $= (\text{偶然 True と一致する確率}) + (\text{偶然 False と一致する確率})$
- $F1 = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$

print() と *format()* による書式変換

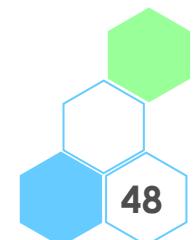
```
x = 1.23 ; y = 456 ; z = "Score"
```

print('total: %.2f' % x)	# 非推奨	'f': 浮動小数
print("total: {:.<6.2f}".format(x))	# 推奨	< ^ >: 左、中央、右寄せ
print("total: {:.2g}".format(x))	# 推奨	.[桁数]g で全体の桁数を指定
print('%s is %d.' % (z, y))	# 非推奨	'd': 10進整数、's': 文字列
print('{} is {}'.format(z, y))	# 推奨	format() の引数を順番に出力
print('{1} is {0}'.format(y, z))	# 推奨	{0}, {1} で 1, 2 番目の値を出力
print('{0} is {:0>6d}'.format(z, y))	# 推奨	半角を 0 で埋める
print('{0} is {:.%}'.format(z, y))	# 推奨	パーセント表記

```
total: 1.23
total: 1.23
total: 1.2

Score is 456.
Score is 456.
Score is 456.
Score is 000456.
Score is 45600.000000%.
```

参考 : <https://docs.python.org/ja/3/library/string.html#format-examples>



種々のデータ形式の読み込み

- CSV 形式

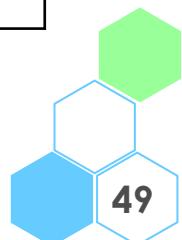
```
import pandas as pd  
  
df = pd.read_csv('./iris.csv', header=0)
```

- EXCEL 形式

```
import pandas as pd  
  
df = pd.read_excel('./iris.xlsx', sheet_name='Sheet1')
```

- SAS 形式: <https://pypi.org/project/sas7bdat/>

```
from sas7bdat import SAS7BDAT  
  
df = SAS7BDAT("iris.sas7bdat", encoding='cp932').to_data_frame()
```



XGBoost: Sklearn 用の Wrapper

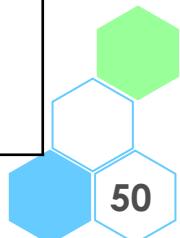
- Sklearn Wrapper: SGD の代わりに XGBoost
 - Sklearn っぽく実行できると、Sklearn での CV や グリッドサーチが流用できる
- 詳細:https://xgboost.readthedocs.io/en/latest/python/python_api.html

```
from    sklearn.model_selection import train_test_split
from    sklearn.metrics           import accuracy_score
from    sklearn.metrics           import confusion_matrix
import xgboost as xgb

iris = pd.read_csv('c:/py/iris.csv', header=0,
names=['SL', 'SW', 'PL', 'PW', 'SP'])
iris.loc[iris['SP']=='setosa',      'Y'] = int(1)
iris.loc[iris['SP']=='versicolor', 'Y'] = int(2)
iris.loc[iris['SP']=='virginica',  'Y'] = int(0)

x_list = ["SL", "SW", "PL", "PW"]
y_list = ["SP"]
data_x = iris[x_list]
data_y = iris[y_list]

x_train, x_test, y_train, y_test = \
train_test_split(data_x, data_y, test_size=0.2)
```



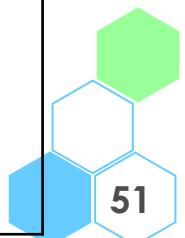
XGBoost: Sklearn 用の Wrapper

```
clf = xgb.XGBClassifier()
clf.fit(x_train, y_train.values.ravel()) # 実行
y_pred = clf.predict(x_test)           # 予測してみる
print(confusion_matrix(y_test, y_pred,
                       labels=['setosa', 'versicolor', 'virginica']))
print(accuracy_score(y_test, y_pred))
```

```
[[11  0  0]
 [ 0  7  0]
 [ 0  1 11]]
0.9666666666666667
```

```
# xgb.train() の場合
data_y = iris.Y
x_train, x_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2)

dtrain = xgb.DMatrix(x_train, label=y_train)
dtest = xgb.DMatrix(x_test, label=y_test)
param = {'objective': 'multi:softprob',
          'num_class': 3}
num_boost_round = 50
xgboost_model = xgb.train(param, dtrain, num_boost_round)
pred = xgboost_model.predict(dtest)
pred_label = np.argmax(pred, axis=-1)
```



XGBoost: Sklearn 用の Wrapper

- Sklearn Wrapper: DecisionTreeRegressor の代わりに XGBoost
 - Sklearn っぽく実行できると、Sklearn での CV や グリッドサーチが流用できる
- 詳細：https://xgboost.readthedocs.io/en/latest/python/python_api.html

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
import xgboost as xgb

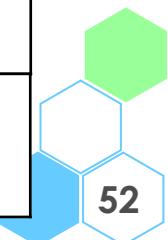
iris = pd.read_csv('c:/py/iris.csv', header=0, names=['SL', 'SW', 'PL', 'PW', 'SP'])
iris.loc[iris['SP']=='setosa', 'Y'] = int(1)
iris.loc[iris['SP']=='versicolor', 'Y'] = int(2)
iris.loc[iris['SP']=='virginica', 'Y'] = int(0)

x_list = ["SL", "PL", "PW"]
y_list = ["SW"]
data_x = iris[x_list]
data_y = iris[y_list]

x_train, x_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2)

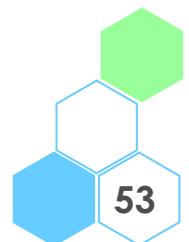
tree = xgb.XGBRegressor()
tree.fit(data_x, data_y)
cv = cross_val_score(tree, data_x, data_y, cv=10,
scoring='neg_root_mean_squared_error')
print(cv.mean())    # -RMSEの平均
print(cv.std())    # その標準偏差
```

-0.34072223813163593
 0.07860750358661242



Target Encodingについて

- ・ 「変数(特徴量)の選択・作成」で行っていた Target Encoding は「単純に全体のデータから生存割合を計算、割合の順に encoding 」したもの
 - ・ これはリーク(バリデーションデータの目的変数の情報を誤って取り込んで学習することにより、バリデーションで不当に高い値が出る状態)する可能性がある
 - ・ Target Encoding は予測精度を高めやすい手法であるが、リークを起こしやすいので取り扱いが難しい
- ・ 次頁以降では、リークを起こしにくいとされる CatBoost Encoder を用いて前処理をやり直す
 - ・ CatBoost Encoder については次回も扱う
- ・ 参考文献:
 - ・ CatBoost Encoder
https://contrib.scikit-learn.org/category_encoders/catboost.html
 - ・ CatBoost の論文、リファレンス
<https://papers.nips.cc/paper/2018/file/14491b756b3a51daac41c24863285549-Paper.pdf>
https://catboost.ai/docs/concepts/algorithm-main-stages_cat-to-numeric.html
 - ・ Categorical Encoders and Benchmark
<https://www.kaggle.com/subinium/11-categorical-encoders-and-benchmark>
 - ・ Python: Target Encoding のやり方について
<https://blog.amedama.jp/entry/target-mean-encoding-types>



変数(特徴量)の選択・作成【やり直し】

```

from category_encoders.cat_boost import CatBoostEncoder

# train.csv と test.csv の結合
df1 = pd.read_csv('C:/py/titanic/train.csv', header=0)
df0 = pd.read_csv('C:/py/titanic/test.csv', header=0)
df1["Is_train"] = 1
df0["Is_train"] = 0

# Nameについて
df1['Title'] = df1['Name'].apply(lambda x: x.split(',') [1]). \
apply(lambda x: x.split() [0])
df0['Title'] = df0['Name'].apply(lambda x: x.split(',') [1]). \
apply(lambda x: x.split() [0])

# CatBoost Encoding
from category_encoders.cat_boost import CatBoostEncoder
cat_vars = ['Pclass', 'Sex', 'Ticket', 'Cabin', 'Embarked', 'Title']
cbe = CatBoostEncoder()
target = df1['Survived']
df1_cbe = cbe.fit_transform(df1[cat_vars], target)
df0_cbe = cbe.transform(df0[cat_vars])

# データの結合('Is_train', 'PassengerId', 'Survived'は後で処理)
drop_vars = cat_vars + ['Name']
df1 = df1.drop(drop_vars, axis=1)
df0 = df0.drop(drop_vars, axis=1)
df1 = pd.concat([df1, df1_cbe], axis=1)
df0 = pd.concat([df0, df0_cbe], axis=1)
df = pd.concat([df1, df0])

# 学習用データとテストデータに分割
drop_vars = ['PassengerId', 'Survived', 'Is_train']
train_x = df.query('Is_train == 1').drop(drop_vars, axis=1)
train_y = df.query('Is_train == 1')['Survived']
test_x = df.query('Is_train == 0').drop(drop_vars, axis=1)
id = df.query('Is_train == 0')['PassengerId']

```



変数(特徴量)の選択・作成【やり直し】

```
# Hyperopt (コード一部省略)
def optimize(trials):
    space = {'seed': 777,
              'booster': 'gbtree',
              'objective': 'binary:logistic',
              'eval_metric': 'error',
              'n_estimators': hp.quniform('n_estimators', 100, 1000, 1),
              'eta': hp.quniform('eta', 0.025, 0.5, 0.025),
              'max_depth': hp.choice('max_depth', np.arange(2, 10, dtype=int)),
              'min_child_weight': hp.loguniform('min_child_weight', np.log(0.1), np.log(10)),
              'subsample': hp.quniform('subsample', 0.6, 0.95, 0.05),
              'colsample_bytree': hp.quniform('colsample_bytree', 0.6, 0.95, 0.05),
              'colsample_bylevel': hp.quniform('colsample_bylevel', 0.1, 0.5, 0.05),
              'gamma': hp.loguniform('gamma', np.log(1e-8), np.log(1.0)),
              'alpha' : hp.loguniform('alpha', np.log(1e-8), np.log(1.0)),
              'lambda' : hp.loguniform('lambda', np.log(1e-6), np.log(10.0))
    }
    best = fmin(score, space, algo=tpe.suggest, trials=trials, max_evals=100)
    return best # 探索回数は25回程度でもOK、100回あれば十分探索可能
trials = Trials()
best = optimize(trials)
print(best)

# 予測
dtrain = xgb.DMatrix(train_x, label=train_y)
dtest = xgb.DMatrix(test_x)

# n_estimators→num_round
num_round = 238
params = {'alpha': 1.31204934783169e-07, 'colsample_bylevel': 0.3500000000000003, 'colsample_bytree': 0.9, 'eta': 0.025, 'gamma': 0.012795089811378056, 'lambda': 0.7610266253205662, 'max_depth': 2, 'min_child_weight': 0.10533265957712513, 'subsample': 0.7000000000000001}

model = xgb.train(params, dtrain, num_round) # 0.79665, 0.80622
pred = model.predict(dtest)
pred_label = np.where(pred > 0.61, 1, 0)
```

kaggle のコンペの
score(閾値0.5):
0.79665
score(閾値0.61):
0.80622



メニュー

- データの準備、XGBoost の概要
- 2 値データの分類問題
 - 前処理 → とりあえず予測
 - バリデーションとパラメータチューニング
 - 変数(特徴量)の選択・作成
- その他

※ 本資料では、普通の python を python、Google Colaboratory を Colab と略記

参考文献

- Python 3.8.3 documentation
<https://docs.python.org/3/>
<https://docs.python.org/ja/3/>
- matplotlib documentation
<https://matplotlib.org/index.html>
https://matplotlib.org/1.5.1/faq/usage_faq.html
- Scikit-learn documentation
<https://scikit-learn.org/stable/>
https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html
- XGBoost documentation
<https://xgboost.readthedocs.io/en/latest/index.html>
- Sebastian Raschka & Vahid Mirjalili (2019) "Python Machine Learning, 3rd Edition", Packt Publishing
<https://github.com/rasbt/python-machine-learning-book-3rd-edition>
- note.nkmk.me: <https://note.nkmk.me/python/>
- kaggle: <https://www.kaggle.com/>

門脇 大輔 他著(2019)「Kaggleで勝つデータ分析の技術(技術評論社)」
<https://github.com/ghmagazine/kagglebook>