インテル® DAAL を利用して実際のマシンラーニングの問題を解く

Oleg Kremnyov インテル コーポレーション テクニカルインターン
Ivan Kuzmin インテル コーポレーション ソフトウェア・エンジニアリング・マネージャー
Gennady Fedorov インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

マシンラーニングは、統計、データマイニング、人工知能 (AI) 分野において重要な役割を果たし、その役割は拡大しつつあります。データの急激な増加に伴い、データの学習はさらに進むと考えられ、またそれはビジネスの成長に不可欠です。特定の問題を解くためには、次のようなさまざまな要因を考慮して、適切なアルゴリズムとライブラリーを選択します。

- 問題のクラス (例 : 分類、回帰)
- 入力データ
- パフォーマンス要件
- 予測の正解率
- モデルの解釈可能性

この選択にはさまざまなスキルが必要になるため、マシンラーニング普及の妨げとなっています。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

この記事では、2 つの実際のマシンラーニングの問題をベースに、適切なアルゴリズムの選択に使用できる条件について述べます。ここで使用する問題の 1 つは、有名な **Kaggle** (英語) プラットフォームで予測モデリングに使用されている問題で、もう 1 つは、データマイナーが最良のモデルの生成を競う同サイトの解析コンペティションからの問題です。アルゴリズムの実装には、以下のライブラリーを使用します。

- **scikit-learn**: Python* データ・サイエンティストの間で最もよく使用されているライブラリー
- **R**: データ解析と統計計算の言語
- **インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL)**: インテル® プラットフォーム上でのデータ解析とマシンラーニング向けに最適化されたビルディング・ブロックを提供するパフォーマンス・ライブラリー

実際のマシンラーニングでは、CPU とメモリーの要件が高いことが多いため、インテル® Xeon Phi™ プロセッサーは理想的なプラットフォームと言えます。インテル® DAAL は、インテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ プロセッサー向けに最適化されたマシンラーニング・アプリケーションを素早く開発できるように支援します。ここでは、**Kaggle: リーフ分類** (英語) と **タイタニック : 大惨事からのマシンラーニング** (英語) の 2 つの実際の問題でインテル® DAAL を利用して KNN (K 近傍法)、ブースティング、サポート・ベクトル・マシン (SVM) を実装し、scikit-learn および R の同じアルゴリズムと結果を比較します。

Kaggle を使用する理由

Kaggle は、企業や研究者が投稿したデータに対し、世界中の統計学者とデータマイナーが最良のモデルの生成を競う、予測モデリングと解析コンペティションのプラットフォームです。[1] 2016 年 5 月時点では、194 カ国に 536,000 の登録ユーザー (Kagglers と呼ばれる) が存在する、世界で最も大きく、多様なコミュニティーの 1 つです。Kaggle は、これまでに 200 を超えるデータ・サイエンス・コンペティションの実績があります。

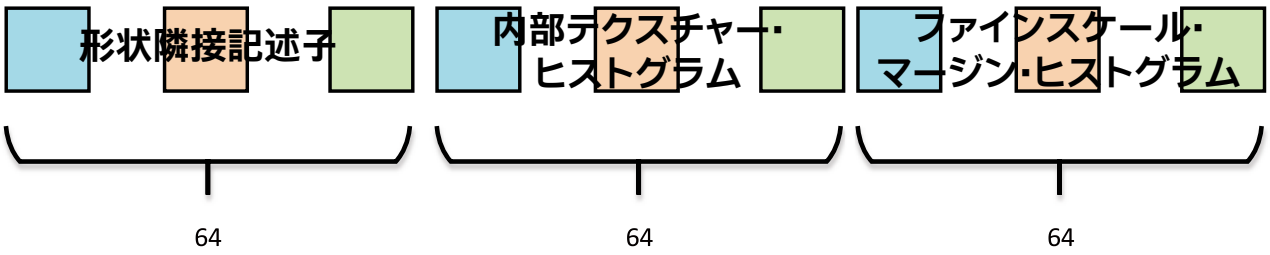
各コンペティションの目標は、特定の実際の問題に対して最良のモデルを生成することです。モデルは、通常、テスト・データ・セットに対する予測の正解率で評価されます。参加者は、テスト・データ・セットで予測を実行して、モデルを瞬時に評価し、ランキングで結果を確認できます。

この記事では、異なるアルゴリズムとライブラリーにより生成されたモデルを評価し、Kaggle コンペティションでどのような結果が得られるか見てみます。

リーフ分類 (Leaf Classification)

世界には数十万種類の植物が存在します。これらの分類は、歴史的に問題が起こりやすく、重複して識別されることがしばしばありました。この Kaggle チャレンジでは、葉のイメージと抽出された特徴 (例えば、形状、マージン、テクスチャー) を基に 99 種類の植物を正確に識別するため、分類器を訓練します。訓練データには 990 の葉のイメージがあり、テストデータには 594 のイメージが含まれています (**図 1**)。イメージごとに次の 3 つの特徴も提供されます : 形状隣接記述子、内部テクスチャー・ヒストグラム、ファインスケール・マージン・ヒストグラム (**図 2**)。各特徴は、葉のサンプルごとにそれぞれ 64 の属性ベクトルで表されます。

1 葉のイメージ



2 特徴の概要

1 つのアプローチとして、マシンラーニング・アルゴリズムのリストを訓練データに適用して、検証データでその正解率を評価し、最適なアルゴリズムとハイパーパラメーターを見つけます。scikit-learn は、Python* データ・サイエンティストの間で最もよく使用されているライブラリーで、広範なアルゴリズムを提供します。**Kaggle カーネル** (英語) では、10 のアルゴリズムの予測の正解率を解析しました。その結果、線形判別式解析と KNN アルゴリズムで検証データに対して最良の結果が得られました (詳細は Kaggle カーネルのページを参照してください)。

線形判別式解析の Python* (scikit-learn) 実装 :

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
clf = LinearDiscriminantAnalysis()
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

intel Software

高速な Python* による データ処理

Anaconda* を利用するインテル® Distribution for Python* により ワークフローとアプリケーション・パフォーマンスをスピードアップしましょう。

コミュニティー・サポート・バージョン (無料) をダウンロードするか、 インテル® Parallel Studio XE の 30 日間評価版としてご利用いただけます。

インテル® Parallel Studio XE 評価版ダウンロード ›

コミュニティー・サポート・バージョンのダウンロード (英語) ›

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation

KNN の Python* (scikit-learn) 実装 :

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(k=4)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

線形判別式解析の R 実装 :

```
library (MASS)
r <- lda(formula = Species ~ ., data = train)
plda = predict(object = r, newdata = test)
test_predictions = plda$class
```

KNN の R 実装 :

```
library(class)
test_predictions = knn(X_train, X_test, y_train, k=4)
```

インテル® DAAL は、KD 木アルゴリズムと低レベルの最適化によりインテル® アーキテクチャー上で非常に高速で正解率が高い、KNN のスケーラブルなバージョンを提供します。[2]

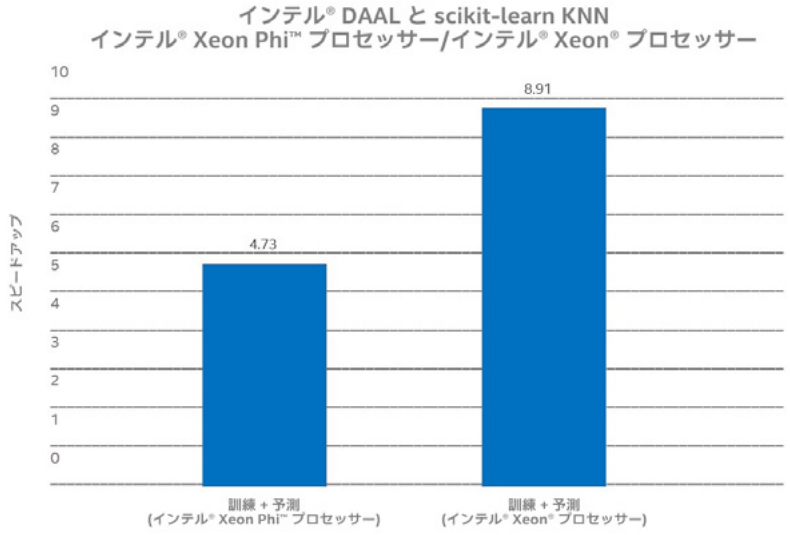
KNN 訓練段階の Python* (インテル® DAAL) 実装 :

```
from daal.algorithms.kdtree_knn_classification import training, prediction
from daal.algorithms import classifier, kdtree_knn_classification
trainAlg = kdtree_knn_classification.training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter.k = 4
trainingResult = trainAlg.compute()
```

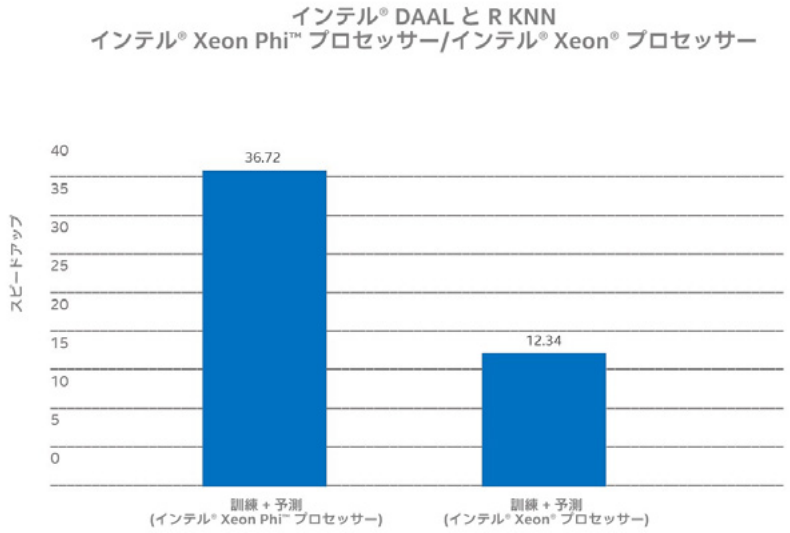
KNN 予測段階の Python* (インテル® DAAL) 実装 :

```
predictAlg = kdtree_knn_classification.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

図 3 と図 4 は、パフォーマンスの比較グラフです。ベンチマークのシステム構成は、この記事の最後にある「システム構成と使用したツール」を参照してください。

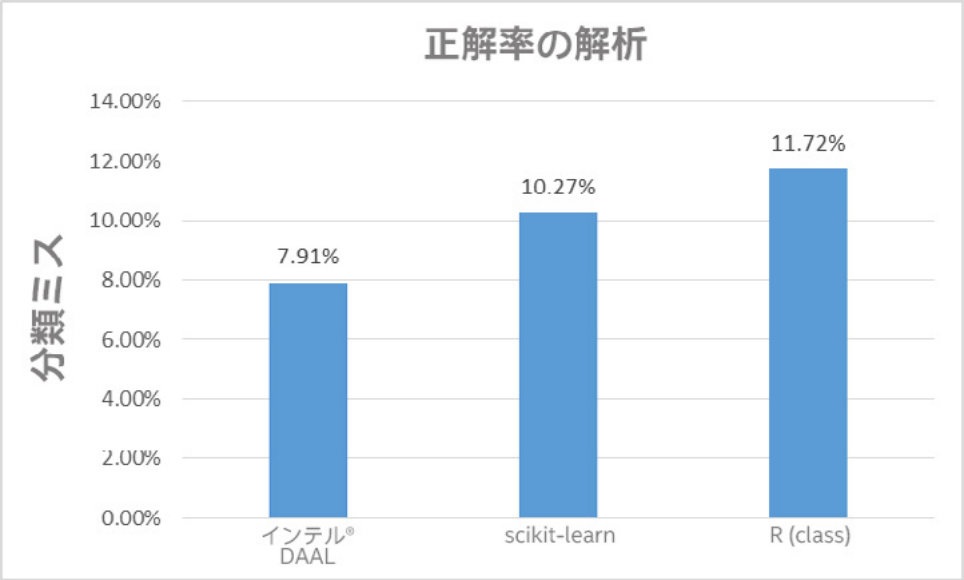


3 scikit-learn と比較したインテル® DAAL のスピードアップ



4 R と比較したインテル® DAAL のスピードアップ

図 5 は、正解率の比較グラフです。



5 正解率の比較 KNN

より少ない次元でデータに適用することで、KNN の正解率を向上できます。統計的決定理論に従って、条件付き (離散型) 分布 $P(G|X)$ (G は予測するラベル) が分かっており、0-1 損失関数を使用する場合、$P(G_k|X=x)=\max_{g \in G} P(g|X=x)$ であれば $\hat{G}(x)=G_k$ と予測できます。KNN 分類は、x の隣接において $P(G_k|X=x)$ が定数であると仮定します。そのため、次元数が大きいほど、k 訓練サンプルを含む x の隣接が大きくなります。この問題では、サンプル数が大きくないため、条件付けの代用として隣接を使用すると失敗します。収束はしますが、次元数の増加に伴い収束率が低下します。詳細は、『The Elements of Statistical Learning (統計学習の基礎)』[3] のセクション 2.4 と 2.5 を参照してください。

線形判別式解析 (LDA) アルゴリズムを使用してオリジナルデータを前処理することで、KNN の正解率を向上できます。ここでは、LDA (nComponents=40) で入力データを前処理して、前処理済みのデータで KNN モデルを訓練しました。

LDA による前処理 (Python*) の実装 :

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components = 40)
X_train_reduced = lda.fit_transform(X_train, y_train)
X_test_reduced = lda.transform(X_test)
```

KNN の Python* (scikit-learn) 実装 :

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(k=4)
clf.fit(X_train_reduced, y_train)
test_predictions = favorite_clf.predict(X_test_reduced)
```

LDA による前処理 (R) の実装 :

```
library (MASS)
r <- lda(formula = Species ~ ., data = train)
plda = predict(object = r, newdata = train)
X_train_reduced = plda$x
plda = predict(object = r, newdata = test)
X_test_reduced = plda$x
```

KNN の R (class) 実装 :

```
library (class)
test_predictions = knn(X_train_reduced, X_test_reduced, y_train, k=4)
```

KNN 訓練段階の Python* (インテル® DAAL) 実装 :

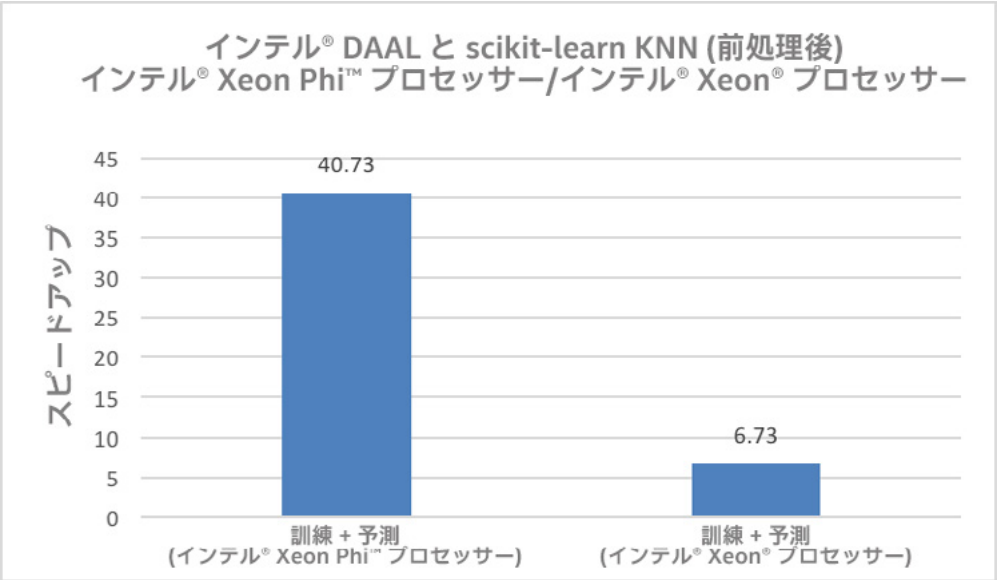
```
from daal.algorithms.kdtree_knn_classification import training, prediction
from daal.algorithms import classifier, kdtree_knn_classification
trainAlg = kdtree_knn_classification.training.Batch()
trainAlg.input.set(classifier.training.data, X_train_reduced)
trainAlg.input.set(classifier.training.labels, y_train_reduced)
trainAlg.parameter.k = 4
trainingResult = trainAlg.compute()
```

KNN 予測段階の Python* (インテル® DAAL) 実装 :

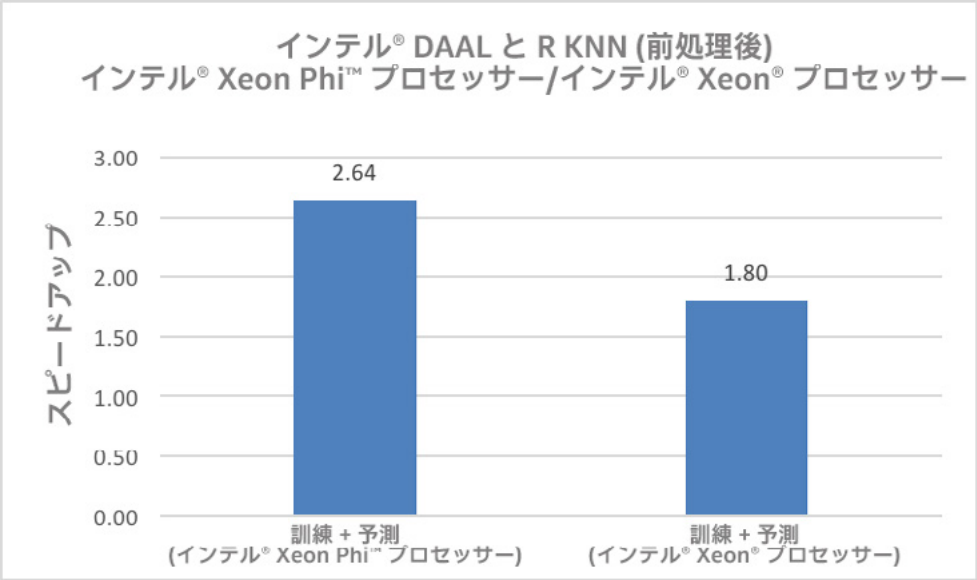
```
predictAlg = kdtree_knn_classification.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test_reduced)
predictAlg.input.setModel(classifier.prediction.model,
└→ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

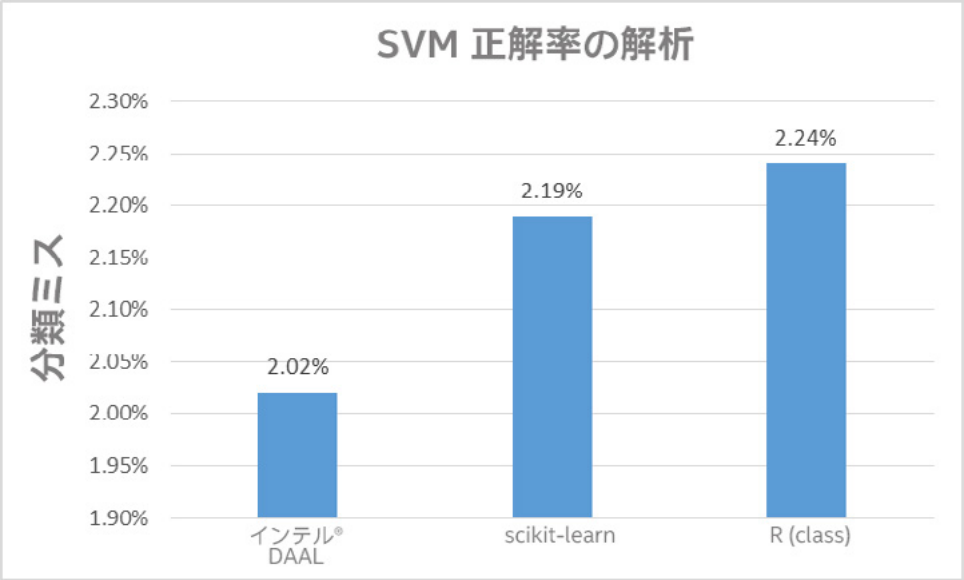
図 6、7、および 8 にデータの前処理を適用した結果を示します。



6 scikit-learn と比較したインテル® DAAL のスピードアップ



7 R と比較したインテル® DAAL のスピードアップ

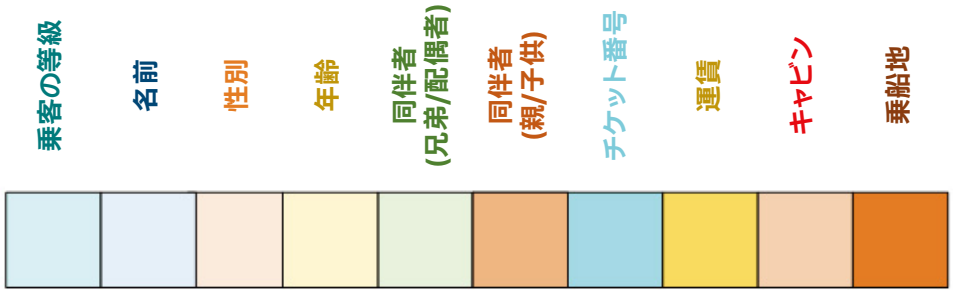


8 前処理後の KNN の正解率の比較

マシンラーニングにおいて特徴が重要な役割を果たすことは明白です。正確に予測するためには、適切な特徴を選択することが不可欠です。また、インテル® DAAL は、テストしたライブラリーの中で最も高い正解率とパフォーマンスを達成しました。

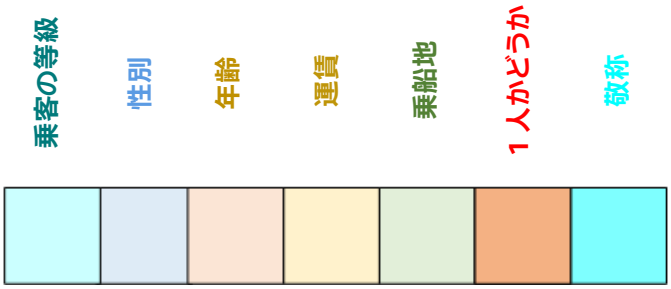
タイタニック : 大惨事からのマシンラーニング

別の Kaggle コンペティションは、歴史上最も有名な海難事故の 1 つであるタイタニック号の沈没を基にしたものです。1912 年 4 月 15 日、初航海中のタイタニック号が氷山に衝突して沈没し、乗客乗員 2,224 中 1,502 人が亡くなりました。この大惨事は世界に衝撃を与え、これをきっかけに船舶の安全規制が改善されました。このチャレンジでは、異なる等級の乗客と乗員を解析して、生存者を予測します。入力データには、**図 9** に示す特徴が含まれています。詳細は、Kaggle の **Data > Overview** (英語) を参照してください。



9 タイタニックのオリジナルの特徴の概要

1 つのアプローチとして、データをマシンラーニング・モデルの訓練に使用できる特徴ベクトルに前処理します。そして、前処理済みデータに対していくつかの分類を試して、最良のアルゴリズムを見つけます。この **Kaggle カーネル** (英語) では、特徴を処理してオリジナルの特徴から次の特徴を構築します (**図 10** を参照): 乗員の等級、性別、年齢 (特徴のビニングにより変換)、運賃 (特徴のビニングにより変換)、乗船地、1 人かどうか (兄弟姉妹 / 配偶者 / 子供 / 親がタイタニック号に乗船していない場合 true)、敬称 (Mrs./Miss/Mr./Master)。そして、scikit-learn の 10 のアルゴリズムを使用してテストを行い、予測の正解率を比較します。その結果、ガウスカーネルを利用した SVM 分類器で最も高い正解率が得られました (詳細は、Kaggle カーネルのページを参照してください)。SVM パラメーターは、交差検証により取得されます。



10 タイタニックの前処理済み特徴の概要

SVM の Python (scikit-learn) 実装 :

```
from sklearn.svm import SVC
clf = SVC(C = 5, gamma = 1.5)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

SVM の R 実装 :

```
library(e1071)
model <- svm(X_train, y_train, gamma=1.5, cost=5)
test_predictions <- predict(model, X_test)
```

ガウスカーネルを利用した SVM には、時間のかかる指数計算が多数含まれています。インテル® DAAL では、これらの計算がインテル® アーキテクチャー向けに高度に最適化されており、SVM モデルを素早く作成できます。

SVM 訓練段階の Python* (インテル® DAAL) 実装 :

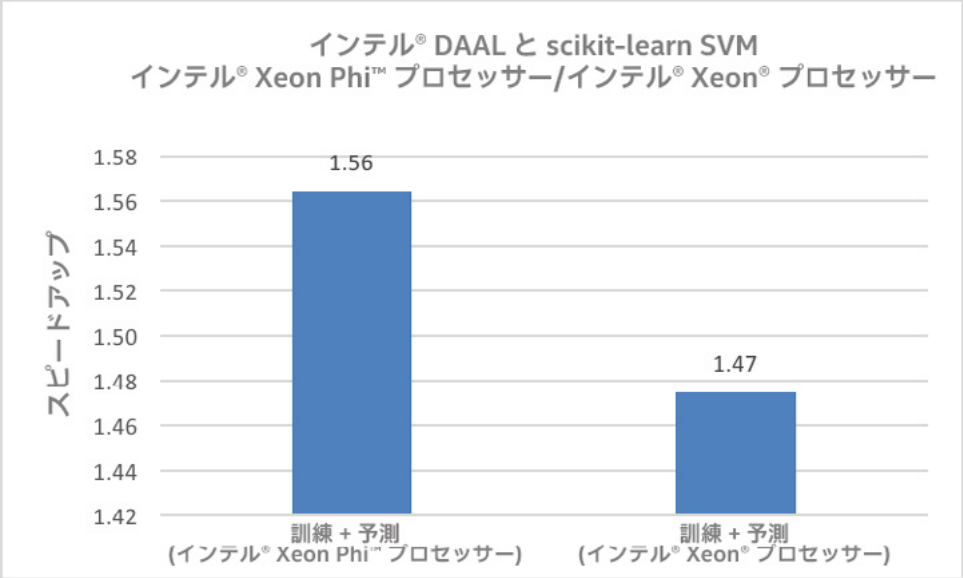
```
from daal.algorithms.svm import prediction, training
from daal.algorithms import kernel_function, classifier
import daal.algorithms.kernel_function.rbf
trainAlg = svm.training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
kernel = kernel_function.rbf.Batch()
kernel.parameter.sigma = 1.5
trainAlg.parameter.C = 5
trainAlg.parameter.kernel = kernel
trainAlg.parameter.cacheSize = 60000000
trainingResult = trainAlg.compute()
```

SVM 予測段階の Python* (インテル® DAAL) 実装 :

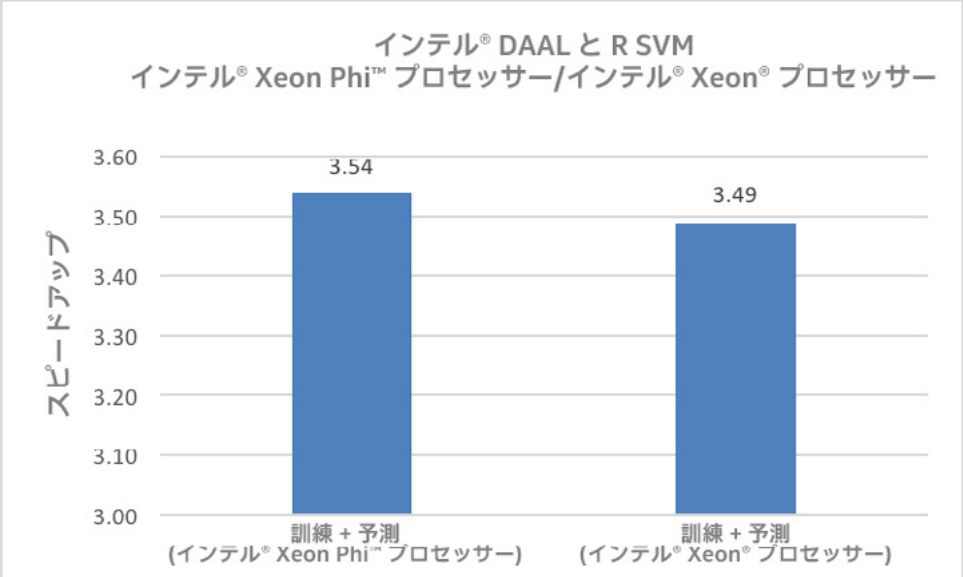
```
predictAlg = svm.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
trainingResult.get(classifier.training.model))
predictAlg.parameter.kernel = kernel
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

図 11 と図 12 は、パフォーマンスの比較グラフです。



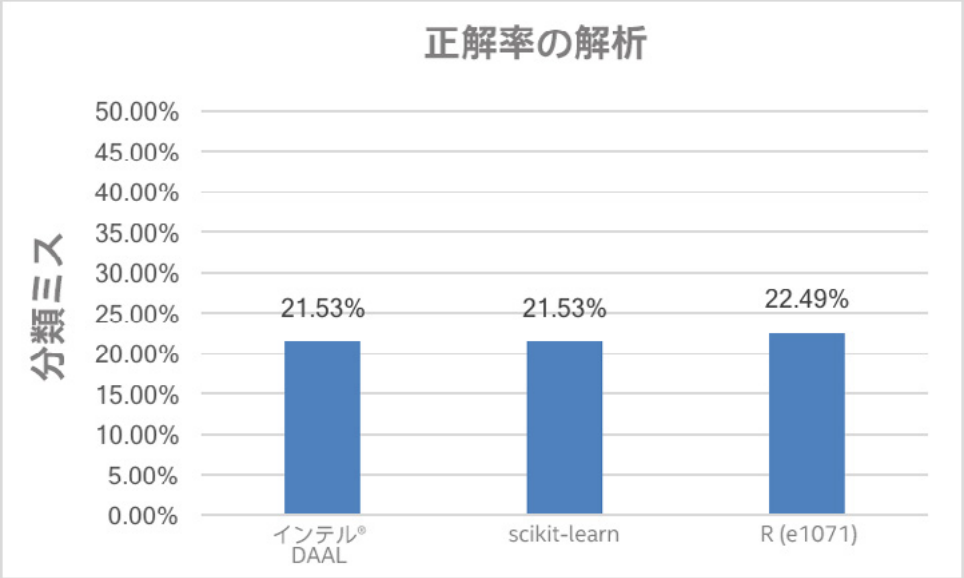
11 scikit-learn と比較したインテル® DAAL のスピードアップ



12 R と比較したインテル® DAAL のスピードアップ

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

図 13 は、正解率の比較グラフです。



13 SVM の正解率の解析

インテル® DAAL と scikit-learn の正解率が高く、インテル® DAAL のパフォーマンスが最も優れていることが分かります。

この分類問題にブースティング分類器を適用します。ブースティングは、過去 20 年間に登場した最も強力な学習アプローチの 1 つです。多数の弱学習器からの出力を組み合わせて、強力な学習機を作成します。[3] ここでは、次のブースティング・アルゴリズムについて考えます。

- AdaBoost
- BrownBoost
- LogitBoost
- 勾配ブースティング

これらのアルゴリズムについては、多数のリソースで詳しく説明されています。[4, 5, 6]

Python* (scikit-learn) の AdaBoost 実装 :

```
from sklearn.ensemble import AdaBoostClassifier
clf = AdaBoostClassifier(n_estimators=1000)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

Python* (scikit-learn) の勾配ブースティング実装 :

```
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier(n_estimators=1000)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

Python* (インテル® DAAL) の AdaBoost (訓練) 実装 :

```
from daal.algorithms.adaboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

Python* (インテル® DAAL) の AdaBoost (予測) 実装 :

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
└→ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Python* (インテル® DAAL) の BrownBoost (訓練) 実装 :

```
from daal.algorithms.brownboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

Python* (インテル® DAAL) の BrownBoost (予測) 実装 :

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
└→ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Python* (インテル® DAAL) の LogitBoost (訓練) 実装 :

```
from daal.algorithms.brownboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

Python* (インテル® DAAL) の LogitBoost (予測) 実装 :

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
└→ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

AdaBoost の R (fastAdaBoost) 実装 :

```
library(fastAdaBoost)
model <- adaboost(Survived ~ ., train, 1000)
pred <- predict(model, newdata=test)
```

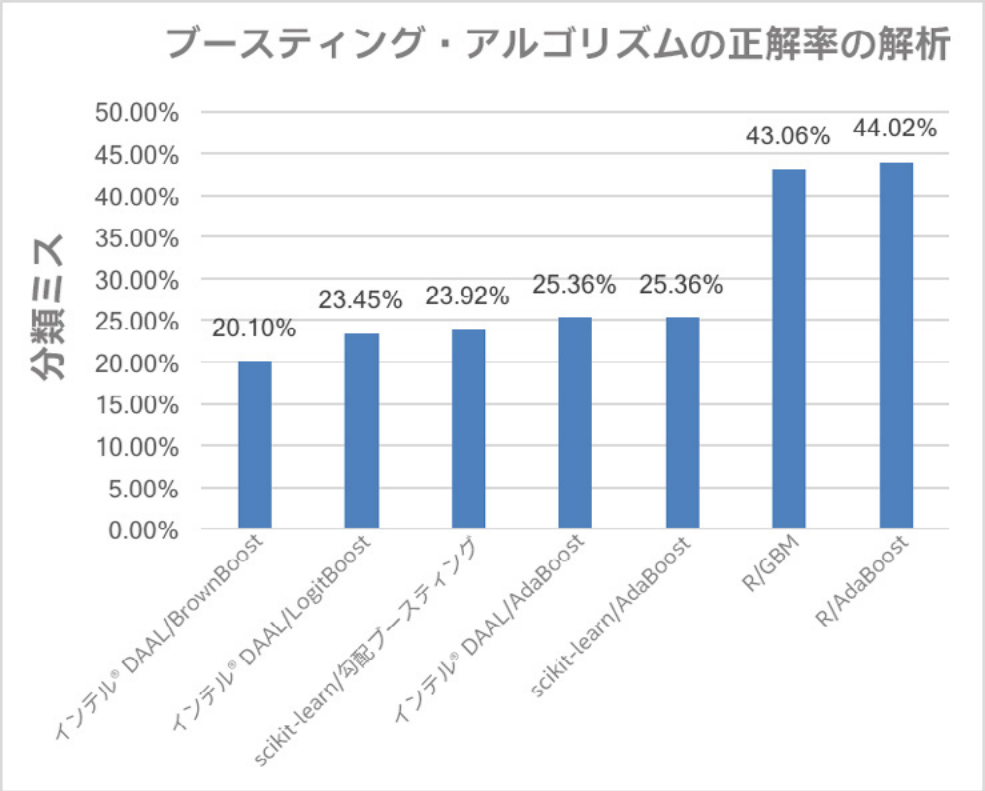
LogitBoost の R (caTools) 実装 :

```
library(caTools)
model <- LogitBoost(X_train, Y_train, nIter=1000)
pred <- predict(model, X_test)
```

勾配ブースティングの R (gbm) 実装 :

```
library(gbm)
model <- gbm(Survived ~ ., data=train, n.tree = 1000, shrinkage = 1)
predict(model, test, n.trees = 1000)
```

図 14 は、異なるブースティング・アルゴリズムの正解率です。



14 ブースティング・アルゴリズムの正解率の解析

インテル® DAAL の BrownBoost アルゴリズムが最も正解率が高いことが分かります。

マシンラーニングとインテル® DAAL を使用してデータ解析の問題を解く

マシンラーニングの問題を解くため適切なアルゴリズムを選択することは重要であり、さまざまな要因を考慮する必要があります。インテル® DAAL や scikit-learn のようなライブラリーは、広範なマシンラーニング・アルゴリズムを提供するため、問題に最適なアルゴリズムを選択することができます。

ここでは、インテル® DAAL を利用して、インテル® プラットフォームの性能を最大限に引き出し、モデルの訓練と予測を高速化する方法を示しました。ここで紹介したベンチマークでは、インテル® DAAL のほうが scikit-learn や R よりもパフォーマンスに優れており、正解率の高いモデルを生成できました。

参考資料 (英語)

1. **Wikipedia* の Kaggle の説明**

2. Patwary, Md. Mostofa Ali; Satish, Nadathur Rajagopalan; Sundaram, Narayanan; Liu, Jialin; Sadowski, Peter; Racah, Evan; Byna, Suren; Tull, Craig; Bhimji, Wahid; Prabhat, Dubey, Pradeep. 2016. "**PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures**," IEEE International Parallel and Distributed Processing Symposium.

3. Hastie, Trevor; Tibshirani, Robert; and Friedman, Jerome. 2009. The Elements of Statistical Learning, Second Edition. Springer International Publishing AG.

4. Freund, Yoav, and Schapire, Robert E. 1999. "Additive Logistic Regression: A Statistical View of Boosting," Journal of Japanese Society for Artificial Intelligence (14[5]), pp. 771–780.

5. Friedman, Jerome; Hastie, Trevor; and Tibshirani, Robert. 2000. "Additive Logistic Regression: A Statistical View of Boosting," The Annals of Statistics, 28(2), pp. 337–407.

BLOG HIGHLIGHTS

適切なテスト方法 (テスト理論パート 1)

JAKOB ENGBLOM ›

欧州宇宙機構のスキャパレリの火星着陸失敗に関する最近の投稿は、テスト、テストツール、および適切なテスト方法について議論されるきっかけとなりました。こちらのブログ (英語) と Wind River* のブログ (英語) で述べたとおり、明白なもの以外にもテストを拡大し、実際には簡単にテストできないものも (シミュレーションにより) テストすることは繰り返し議論されてきました。テスト理論に関するこのシリーズ (全 2 パート) では、テストに対する私の考えを要約し、いくつかの逸話を皆さんと共有したいと思います。

この記事の続きはこちら (英語) でご覧になれます。 ›

6. Friedman, Jerome. 2001. “Greedy Function Approximation: A Gradient Boosting Machine,” The Annals of Statistics 29(5), pp. 1189–1232.

システム構成と使用したツール

ベンチマークのシステム構成 :

インテル® Xeon® プロセッサー :

モデル : インテル® Xeon® プロセッサー E5-2699 v4 @ 2.20GHz
ソケットごとのコア数 : 22
ソケット数 : 2
合計メモリー : 256GB

インテル® Xeon Phi™ プロセッサー :

モデル : インテル® Xeon Phi™ プロセッサー 000A @ 1.40GHz
ソケットごとのコア数 : 68
ソケット数 : 1
RAM: 16GB

使用したソフトウェア・ツール :

- インテル® DAAL 2017 Beta Update 2
- R 3.3.2
- scikit-learn 0.19.1
- class パッケージ 7.3-14
- MASS パッケージ 7.3-45
- e1071 パッケージ 1.6-8
- fastAdaBoost パッケージ 1.0.0
- caTools パッケージ 1.17.1
- gbm パッケージ 2.1.1

インテル® DAAL の詳細 ›

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。