UDACITY

Back to Machine Learning Engineer Nanodegree

Finding Donors for CharityML

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Very impressive submission here, as you have good understanding of these techniques and you now have a solid understanding of the machine learning pipeline. Hopefully you can learn a bit more from this review and wish you the best of luck in your future!

Exploring the Data

Student's implementation correctly calculates the following:

- **Number of records**
- **Number of individuals with income >$50,000**
- **Number of individuals with income <=$50,000**
- **Percentage of individuals with income > $50,000**

All correct here and great use of pandas column slicing to receive these statistics!! Always a good idea to get a basic understanding of the dataset before doing more complex computations. As we now know that we have an unbalanced dataset, so we can plan accordingly.

Another great idea would be to preform some extra exploratory data analysis for the features. Could check out the library Seaborn. For example

```
import seaborn as sns
sns.factorplot('income', 'capital-gain', hue='sex', data=data, kind='bar', col='race', row='relationship')
```

Preparing the Data

Student correctly implements one-hot encoding for the feature and income data.

Great work with pd.get_dummies

Another way we could do this is with LabelEncoder from sklearn. As this would be suitable with a larger categorical range of values

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
income = le.fit_transform(income_raw)
# print one hot
print(income)
# then we can reverse it with
print(le.inverse_transform(income))
```

Evaluating Model Performance

Student correctly calculates the benchmark score of the naive predictor for both accuracy and F1 scores.

Impressive calculation. It is always a great idea to establish a benchmark in any type of problem. As these are now considered our "dumb" classifier results, as any real model should be able to beat these scores, and if they don't we may have some model issues.

The pros and cons or application for each model is provided with reasonable justification why each model was chosen to be explored.

Please list all the references you use while listing out your pros and cons.

Very nice job mentioning some real-world application, strengths / weaknesses and reasoning for your choice! Great to know even outside of this project!

Logistic Regression

- The big thing that should be noted here is that a Logistic Regression model is a linear classifier. It cannot fit non-linear data. Thus, the model creates a single straight line boundary between the classes. (http://stats.stackexchange.com/questions/79259/how-can-i-account-for-a-nonlinear-variable-in-a-logistic-regression) (http://stats.stackexchange.com/questions/93569/why-is-logistic-regression-a-linear-classifier)
- Interpretable with some help
- Great for probabilities, since this works based on the sigmoid function
- Can set a threshold value!!

KNN

- Since KNN is a lazy learning method classification time is nonlinear, training is fast(simply memorize the data), however classification is slow.
- The biggest strength is how simple and effective it is
- Robust to noisy data, as long as an appropriate `n_neighbor` is used.
- Effective with large amounts of data
- Sensitive to irrelevant or redundant features as all features contribute to the similarity

Ada Boost

- Combines multiple weak learners which can eventually lead to a more robust model, typically reduce the variance.
- They can fit noisy data.
- Another great thing that an Ada Boost model and tree methods in sklearn gives us is feature importances. Which we use later on.
- Can choose your 'weak learner'

Student successfully implements a pipeline in code that will train and predict on the supervised learning algorithm given.

Very nice implementation of the `train_predict()` function!

One thing to be aware of in the future, in `fbeta_score()`, you can't switch the `y_true` and `y_pred` parameter arguments. `y_true` has to come first. Check out this example to demonstrate this

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, fbeta_score
clf = DecisionTreeClassifier(max_depth=5, random_state=1)
clf.fit(X_train, y_train)
# the correct way
print(fbeta_score(y_test, clf.predict(X_test), 0.5))
```

```
# the incorrect way
print(fbeta_score(clf.predict(X_test), y_test, 0.5))
```

Student correctly implements three supervised learning models and produces a performance visualization.

Nice work setting random states in these models and subsetting with the appropriate training set size!

Improving Results

Justification is provided for which model appears to be the best to use given computational cost, model performance, and the characteristics of the data.

Good justification for your choice in your AdaBoost model, you have done a great job comparing these all in terms of predictive power and computational expense. Glad that you mention the extremely long prediction time of your KNN, as this model is a lazy learning model, so definitely not applicable for quick query times.

As in practice we always need to think about multiple things

- the predictive power of the model
- the runtime of the model and how it will scale to much more data
- the interpretability of the model
- how often we will need to run the model and/or if it support online learning.

Student is able to clearly and concisely describe how the optimal model works in layman's terms to someone who is not familiar with machine learning nor has a technical background.

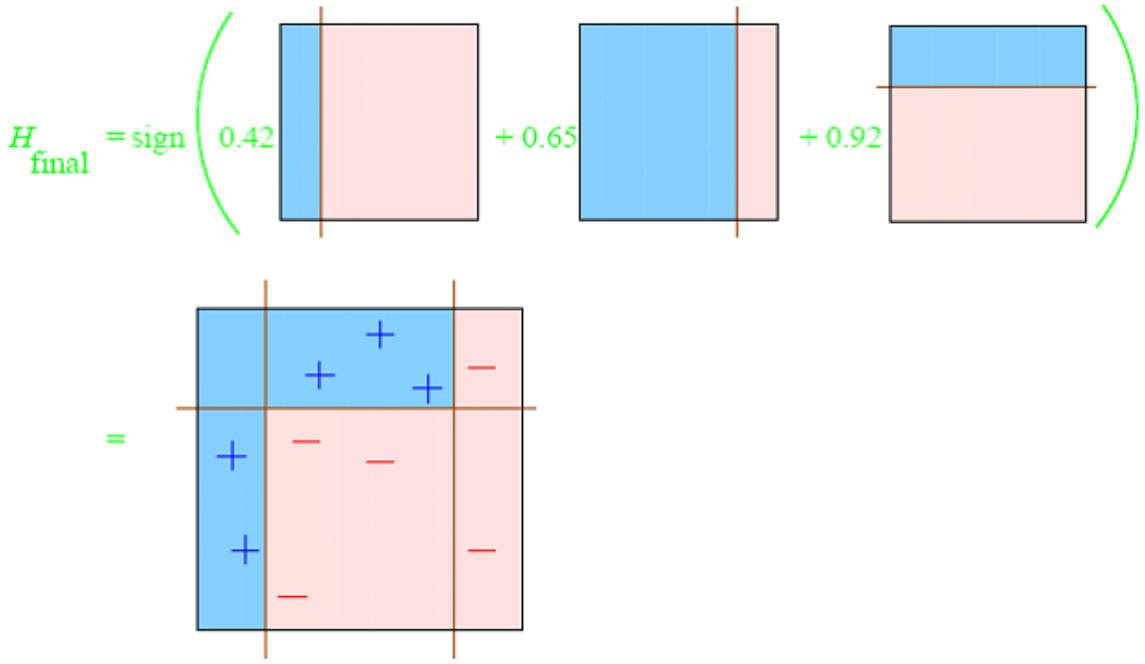
Good description of how your Ada Boost Model would be trained. You have captured the 'boosting' aspect of this model quite well. As this would be great for someone who is not familiar with machine learning nor has a technical background.

Maybe some other ideas for you

- "The basic principle of AdaBoost is to train a sequence of simple algorithms (called weak learners), such us small decision trees, with successive versions of the data. This simple models' predictions are just slightly better than random assumptions. Final prediction is produced combining the individual weak learners' predictions trough a weighted sum (or vote). Data is modified every iteration applying a weight to each training samples. Initially, those weights are the same for all samples, so the first iteration trains the weak learner with the original data. In successive iterations, the weights of the samples are individually modified so the learning algorithm is applied to re-weighted data. In each step training data points which were incorrectly predicted by the previous step model are increased in their weights and the

weights of the well-predicted data point are decreased. This iterative process produces that points which are difficult to predict get more influence, forcing the successive weak learners to focus on data which were not predicted by previous learners."

Maybe even a visual could also support your analysis



The final model chosen is correctly tuned using grid search with at least one parameter using at least three settings. If the model does not need any parameter tuning it is explicitly stated with reasonable justification.

Great use of GridSearch here and the hyper-parameters of `n_estimators` and `learning_rate` are definitely the most tuned hyper-parameters for an AdaBoostClassifier. Could also even look into tuning your base_estimator!
(http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html)

For example

```
clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(), random_stat
e = 42)
parameters = {
    'base_estimator__max_depth':[1,2,4,8],
    'base_estimator__min_samples_split':[10, 100, 200, 500,1000],
    'n_estimators': [50, 100, 200, 500],
    'learning_rate':[0.01, 0.1, 1]}
}
```

Pro Tip: With an unbalanced dataset like this one, one idea to make sure the labels are evenly split between the validation sets a great idea would be to use sklearn's StratifiedShuffleSplit

```
from sklearn.model_selection import StratifiedShuffleSplit
cv = StratifiedShuffleSplit(...)
grid_obj = GridSearchCV(clf, parameters,scoring=scorer, cv=cv)
```

Student reports the accuracy and F1 score of the optimized, unoptimized, models correctly in the table provided. Student compares the final model results to previous results obtained.

Good work comparing your tuned model to the untuned one.

Pro Tip: We could also examine the final confusion matrix. A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
%matplotlib inline

pred = best_clf.predict(X_test)
sns.heatmap(confusion_matrix(y_test, pred), annot = True, fmt = '')
```

Feature Importance

Student ranks five features which they believe to be the most relevant for predicting an individual's' income. Discussion is provided for why these features were chosen.

These are some great features to check out. Very intuitive.

Student correctly implements a supervised learning model that makes use of the `feature_importances_` attribute. Additionally, student discusses the differences or similarities between the features they considered relevant and the reported relevant features.

Could also use your gridSearch tuned AdaBoostClassifier here

```
importances = best_clf.feature_importances_
```

And not too shabby with your guesses above. It is tough. You may also notice that most of the 'important' features are numerical features, any ideas in why this is true?

As `feature_importances_` are always a good idea to check out for tree based algorithms. Some other ideas

- As tree based methods are good for this, if you use something like linear regression or logistic regression, the coefficients would be great to check out.
- Another idea would be to check out the feature_selection module in sklearn. As SelectKBest and SelectPercentile could also give you important features.
- You might also want to check out this post for some other really cool model interpretation techniques

Student analyzes the final model's performance when only the top 5 features are used and compares this performance to the optimized model from Question 5.

Would agree, as we always need to weigh the pros and cons here. Another idea, instead of solely picking a subset of features, would be to try out algorithms such as PCA. Which can be handy at times, as we can actually combine the most correlated/prevalent features into something more meaningful and still can reduce the size of the input. You will see this more in the next project!

Maybe something like this

```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.8, whiten=True)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

clf_pca = (clone(best_clf)).fit(X_train_pca, y_train)
pca_predictions = clf_pca.predict(X_test_pca)

print "\nFinal Model trained on PCA data\n------"
print "Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, pca_pr
edictions))
print "F-score on testing data: {:.4f}".format(fbeta_score(y_test, pca_predic
tions, beta = 0.5))
```

DOWNLOAD PROJECT

Rate this review