UDACITY

‹ Back to Machine Learning Engineer Nanodegree

Finding Donors for CharityML

REVIEW

HISTORY

Requires Changes

2 SPECIFICATIONS REQUIRE CHANGES

Hi,

This is a great submission of this project. Just a couple of details and you are done.

Please keep up the good work 🙌.

Best regards,

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**

Great work here! You correctly calculated all the dataset statistics.

It is clearly indicating that the classes are imbalanced. Here are useful articles on handling imbalanced datasets:

- https://www.quora.com/In-classification-how-do-you-handle-an-unbalanced-training-set
- https://blog.dominodatalab.com/imbalanced-datasets/
- https://www.kdnuggets.com/2017/06/7-techniques-handle-imbalanced-data.html

Another option is to use with the `pandas` method `describe`:

```
data.describe()
```

And here is a useful diagram showing the pipeline to implement a supervised learning classifier:

Diagramhttps://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/163968/1529527760/Overview.png

Preparing the Data

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

OK, great! Just with a couple of lines, you were able to apply the transformations.

For your reference, you can check out this article which explains when and why we use One Hot Encoding.

Something to note here is that we can also use Label Encoder as an alternative in case where we have a huge number of output classes Multi Class predictions. Label Encoder can be implemented as below:

`encoder = LabelEncoder() income = encoder.fit_transform(income_raw)`

Pro tip: This reference provides 7 different encoding strategies. Binary encoding is a great choice for cases where the number of categories for a given feature is very high.

Evaluating Model Performance

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

Great job calculating the accuracy and the F-score for a Naive predictor!

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

Note that the F-score is higher than the accuracy which seems counter-intuitive since the F-score is a more elaborate calculation. That happens because a value of beta = 0.5 attenuates the influence of false negatives. In other words, this value of beta weights more the positive predictions (>50K) than the negative one (<=50K).

Also note that one interesting aspect to this predictor is that precision is equivalent to accuracy, and recall is always one. Hence a simpler implementation:

```
accuracy = n_greater_50k / n_records
fscore = (1.25) * accuracy / (0.25 * accuracy + 1)
```

Pro tip: I recommend having a look at this great article to understand more about choosing the right metric for classification problems.

You could check this link for further understanding precision and recall.

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.

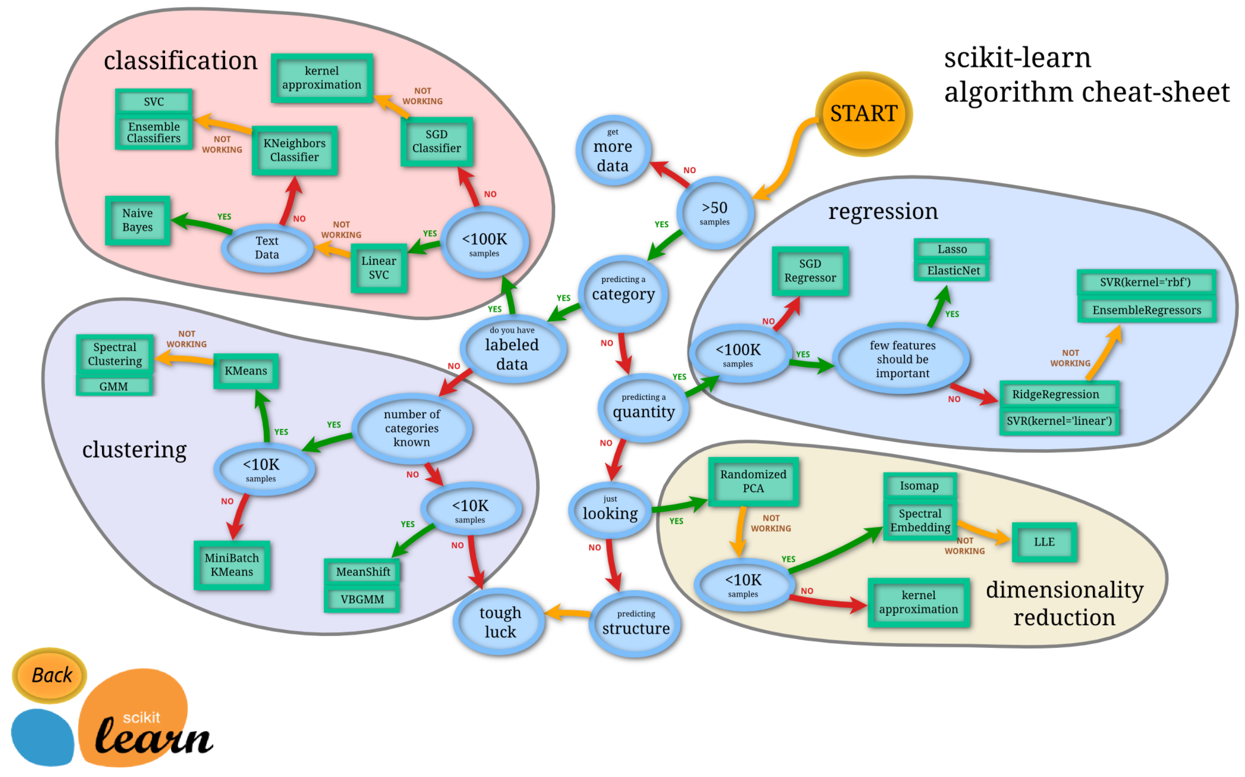
Very nice job mentioning some real-world application, strengths/weakness and reasoning for your choice!

Here might be some ideas to think about and look for in the data/model regarding which to choose:

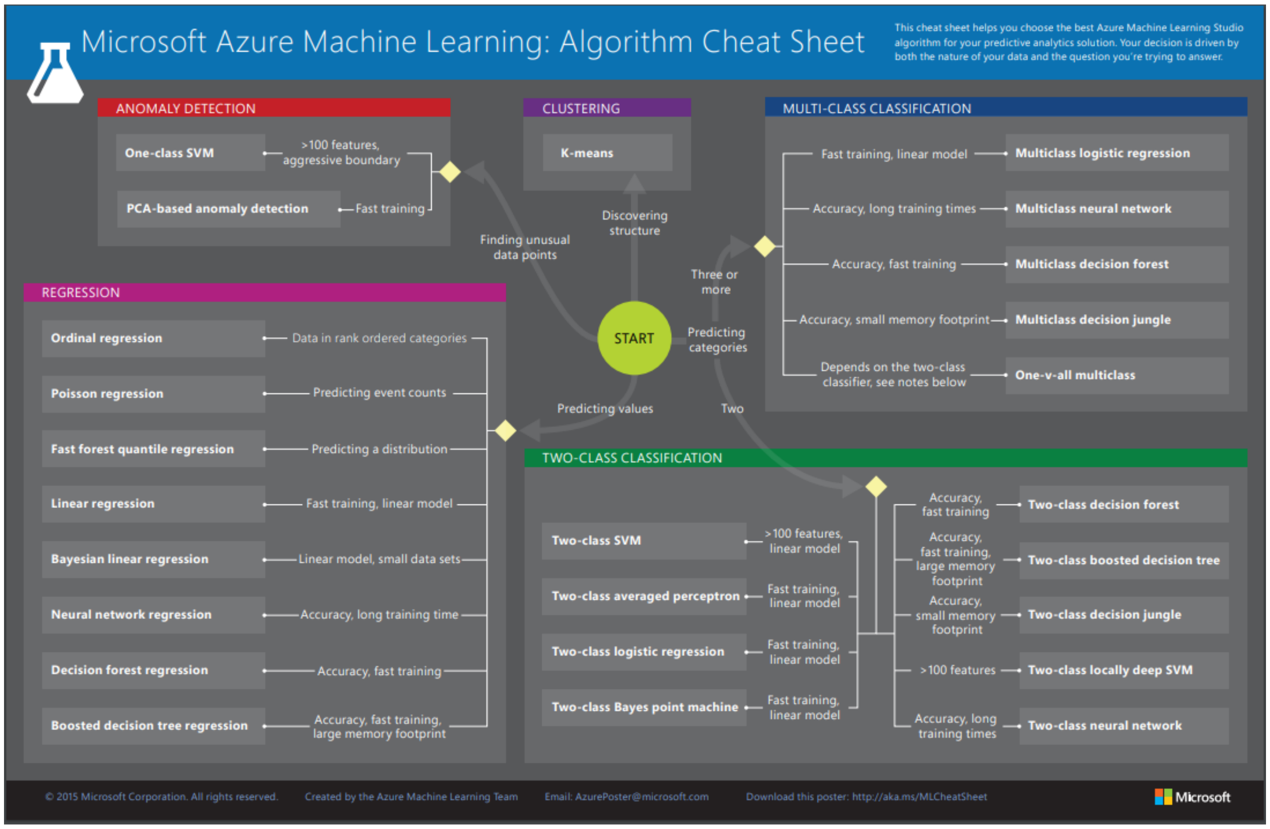
- The predictive power of the model
- The runtime of the model and how it will scale to much more data
- The interpretability of the model
- How often we will need to run the model and/or if it supports online learning. How many categorical features / numerical?
- Distribution of target variable? Linear data?
- Non-linear data?
- Outliers?
- Missing data? (amount of data)

Just to name a few.

Can also check out this flowchart from Sklearn as a rough guideline:



And this from Azure:


Microsoft Azure Machine Learning: Algorithm Cheat Sheet
This cheat sheet helps you choose the best Azure Machine Learning Studio algorithm for your predictive analytics solution. Your decision is driven by both the nature of your data and the question you're trying to answer.
ANOMALY DETECTION
One-class SVM
>100 features, aggressive boundary
PCA-based anomaly detection
Fast training
CLUSTERING
K-means
Discovering structure
Finding unusual data points
MULTI-CLASS CLASSIFICATION
Fast training, linear model
Multiclass logistic regression
Accuracy, long training times
Multiclass neural network
Accuracy, fast training
Multiclass decision forest
Accuracy, small memory footprint
Multiclass decision jungle
Depends on the two-class classifier, see notes below
One-v-all multiclass
Three or more
START
Predicting categories
Predicting values
Two
REGRESSION
Ordinal regression
Data in rank ordered categories
Poisson regression
Predicting event counts
Fast forest quantile regression
Predicting a distribution
Linear regression
Fast training, linear model
Bayesian linear regression
Linear model, small data sets
Neural network regression
Accuracy, long training time
Decision forest regression
Accuracy, fast training
Boosted decision tree regression
Accuracy, fast training, large memory footprint
TWO-CLASS CLASSIFICATION
Two-class SVM
>100 features, linear model
Two-class averaged perceptron
Fast training, linear model
Two-class logistic regression
Fast training, linear model
Two-class Bayes point machine
Fast training, linear model
Accuracy, fast training
Two-class decision forest
Accuracy, fast training, large memory footprint
Two-class boosted decision tree
Accuracy, small memory footprint
Two-class decision jungle
>100 features
Two-class locally deep SVM
Accuracy, long training times
Two-class neural network
© 2015 Microsoft Corporation. All rights reserved.
Created by the Azure Machine Learning Team
Email: AzurePoster@microsoft.com
Download this poster: http://aka.ms/MLCheatSheet
Microsoft

This reference provides many visualization resources for machine learning algorithms. I recommend it to get a visual intuition of ML techniques.

This reference from sklearn compares the decision boundaries of the major classifiers.

In general, with model selection it's a good idea to try out simpler methods like Logistic Regression or Naive Bayes as a benchmark, and then move on to non-linear classifiers such as your choice of Decision Trees and ensemble methods. It may be exciting and tempting to apply the most sophisticated models right from the start, but in real production projects, the golden rule is you only want to use as much sophistication as needed – in order to maximize efficiency.

If you're interviewing for machine learning roles, you'll find it's common for recruiters to ask questions about comparing different ML models and the most suitable use cases for each. The following additional reading will hopefully help you ace these questions.

- Comprehensive SAS article on model selection
- Great Quora thread comparing common ML models

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

Excellent work implementing the train_predict function! This part is the main driver for the entire project. With this pipeline you can see how the performance of the 3 models changes when using different training sizes passed to the sample_size parameter.

ML pipelines are awesome tools. A Pipeline can be specified as a sequence of stages which runs these sequence of stages one after the other. What this achieves is that, imagine you are working on large ML projects and you need to find out which model performs best. You'll then often experiment with the models to see which one works better and also what combination of parameters work best. A pipeline helps us with repeatable task without the same implementation steps for each algorithm every time. It basically helps you with the hard work.

There are standard workflows in applied machine learning. Standard because they overcome common problems like data leakage in your test harness. Scikit-learn provides a Pipeline utility to help automate machine learning workflows. Pipelines work by allowing for a linear sequence of data transforms to be chained together culminating in a modeling process that can be evaluated.

The goal is to ensure that all of the steps in the pipeline are constrained to the data available for the evaluation, such as the training dataset or each fold of the cross-validation procedure.

You can learn more about Pipelines in scikit-learn from the API documentation.

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

Great job generating the results with the 3 sample sizes. However, you'll need to set random states on the classifiers for all models to make your results reproducible. This is a useful best practice to help with debugging and fine-tuning later on. It also helps the reviewers reproduce the exact same result when assessing your code.

Here is an example of how to do it:

`LogisticRegression(random_state=999)

You can check the documentation of each model and see if they have the random_state parameter:

- LogisticRegression

Here is an image that shows the accuracy variation depending on the random state value:

![Example](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/164676/1520840659/2018-03-10_09_26_37-Udacity_Reviews.png

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 model.

It might not be fair to compare different models with their default parameters. For example, Decision Trees tend to not perform well with their default parameters while their tuned model can surpass the tuned SVM.

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.

Well done on the explanation provided so far. However you need to explain a little bit more.

Please have a look at these links that try to explain ML models in simple terms:

- https://rayli.net/blog/data/top-10-data-mining-algorithms-in-plain-english/
- http://blog.echen.me/2011/03/14/laymans-introduction-to-random-forests/
- https://prateekvjoshi.com/2014/05/05/what-is-adaboost/
- http://xgboost.readthedocs.io/en/latest/model.html

How a Decision Tree works?

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 job tuning your classifier! However, you also need to set a random state for your classifier here.

And use beta = 0.5, not 2.

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.

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.

Very intuitive features you described.
Feature Selection is a very critical step in any Machine Learning algorithm’s workflow. Top reasons to use feature selection are:

- It enables the algorithm to train faster
- It reduces the complexity of a model and makes it easier to interpret
- It improves the accuracy of a model if the right subset is chosen

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.

It is tough to guess.

As you can see in this example, intuition about the feature importances in any ML problem is a good initial approach, but a thorough method is a better approach since its conclusions are based on the data relations between features and label.

You may also notice that most of the 'important' features are numerical features, any ideas in why this is true?

There are not many algorithms that use `feature_importances_` (some use `coef_` instead, but they are not interchangeable). `feature_importance_` is always a good idea to check out for tree based algorithms.

Modern day datasets are very rich in information. So understanding the dataset is always a good choice as this always saves a lot of troubles later. The data are often high dimensional and it is quite common to see datasets with hundreds of features and is not unusual to see it go to tens of thousands.

So, first thing that comes to our mind is: More features mean better for the model right ? Well not always! When a model is presented data with very high dimensionality which means a huge number of features, models usually choke. This is because:

- As no of features increases, training time increases exponentially.
- Models often suffer from overfitting with more no of features.

As this nice article points out, "Sometimes Less is better!"

Something is note here that there are a lot of methods for implementing feature selection. Below are some of the most common one:

- SelectBest
- SelectPercentile

A more advanced method is Recursive Feature Elimination. Recursive feature elimination is based on the idea to repeatedly construct a model (for example an SVM or a regression model) and choose either the best or worst performing feature (for example based on coefficients), setting the feature aside and then repeating the process with the rest of the features. This process is applied until all features in the dataset are exhausted. Features are then ranked according to when they were eliminated. As such, it is a greedy optimization for finding the best performing subset of features.

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.

Pro-Tip 1: Instead of solely picking a subset of features, would be a good idea 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!

Pro-Tip 2: Feature selection (and deciding on the number of features to use in training your classifier) is an important consideration in Machine Learning problems. In general, the more features you train on, the better your accuracy, but the greater your compute cost. Experienced Machine Learning engineers understand this dynamic and are able to make trade-offs suitable for each different use case. Also note that a good feature set contains features that are highly correlated with the class, yet uncorrelated with each other. In general, feature reduction is a great way to fight the curse of dimensionality (https://medium.freecodecamp.org/the-curse-of-dimensionality-how-we-can-save-big-data-from-itself-d9fa0f872335).

RESUBMIT

DOWNLOAD PROJECT

Learn the best practices for revising and resubmitting your project.

RETURN TO PATH

Student FAQ