Published in Towards Data Science

Alex Ortner Follow

May 28, 2020 · 13 min read · · Listen

Save

Top 10 Binary Classification Algorithms [a Beginner's Guide]

How to implement the 10 most important binary classification algorithms with a few lines of Python

Photo by Javier Allegue Barros on Unsplash

Introduction

Binary classification problems can be solved by a variety of machine learning algorithms ranging from Naive Bayes to deep learning networks. Which solution performs best in terms of runtime and accuracy depends on the data volume (number of samples and features) and data quality (outliers, imbalanced data).

This article provides an overview and code examples you can easily try out yourself. The aim is to get first working results with Python quickly. I will keep things short here and will explain each algorithm in detail. I have added references to each algorithm in case you want to understand more about the algorithm, its underlying theory and how to tune its parameters.

In this article, we will focus on the top 10 most common binary classification algorithms:

1. Naive Bayes

2. Logistic Regression
3. K-Nearest Neighbours
4. Support Vector Machine
5. Decision Tree
6. Bagging Decision Tree (Ensemble Learning I)
7. Boosted Decision Tree (Ensemble Learning II)
8. Random Forest (Ensemble Learning III)
9. Voting Classification (Ensemble Learning IV)
10. Neural Network (Deep Learning)

To keep things as simple as possible, we will only use three Python libraries in this tutorial: *Numpy*, *Sklearn* and *Keras*.

In the code examples, I always import the necessary Python module right on top of the the code snippet to make clear that it is used next. You can load them all in the beginning of your script.

I have created a Jupyter Notebook with all the code that you can find in my Github repository: https://github.com/alexortner/teaching/tree/master/binary_classification

Dataset

In this article, we will perform a binary sentiment analysis of movie reviews, a common problem in natural language processing.

We are using the IMDB dataset of highly polar movie reviews in the form of text comments on different movies and a positive or negative score.

The goal is to train a model that can predict the review by analysing the sentiment in the comment text.

Data import

We are loading the dataset directly from *Keras*. This has the advantage that we need no file download and almost no data preparation in our code. The description from the Keras website states:

> *This is a dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a list of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".*
>
> *As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.*
>
> *(citation from: https://keras.io/api/datasets/imdb/)*

```
import keras.datasets as keras_data

(imdb_train_data,imdb_train_labels),(imdb_test_data,imdb_test_labels)
= keras_data.imdb.load_data(num_words=10000)
```

num_words limits the text import to the 10,000 most used words.

Quick overview of the data

The training and test data contain 25000 samples each. The text from the *Keras* source is already tokenised, meaning that it is encoded in a sequence of integers where each number corresponds to one word.

```
print(imdb_train_data.shape)
print(imdb_test_data.shape)

print(imdb_train_data[5])

# output
(25000,)
(25000,)
[1, 778, 128, 74, 12, 630, 163, 15, 4, 1766, 7982, 1051, 2, 32, 85,
```

```
156, 45, 40, 148, 139, 121, 664, 665, 10, 10, 1361, 173, 4, 749, 2,
16, 3804, 8, 4, 226, 65, 12, 43, 127, 24, 2, 10, 10]
```

In case you would like to see the comment in clear text, you can map the index back to the original words by using the *get_word_index* function from the dataset. Keep in mind that punctuation and spaces have already been removed from the dataset and therefore can't be restored.

```
word_index = keras_data.imdb.get_word_index()

# map index to word mapping into a Python dict
reverse_word_index = dict([(value,key) for (key,value) in
word_index.items()])

# map the word index to one of the tokenized comments
decoded_word_index = ''.join([reverse_word_index.get(i-3,'?') for i in
imdb_train_data[5]])
```

Print the 10 most common words in the word index list:

```
for key in sorted(reverse_word_index.keys()):
    if(key<=10):
        print("%s: %s" % (key, reverse_word_index[key]))

# output
1: the
2: and
3: a
4: of
5: to
6: is
7: br
8: in
9: it
10: i
```

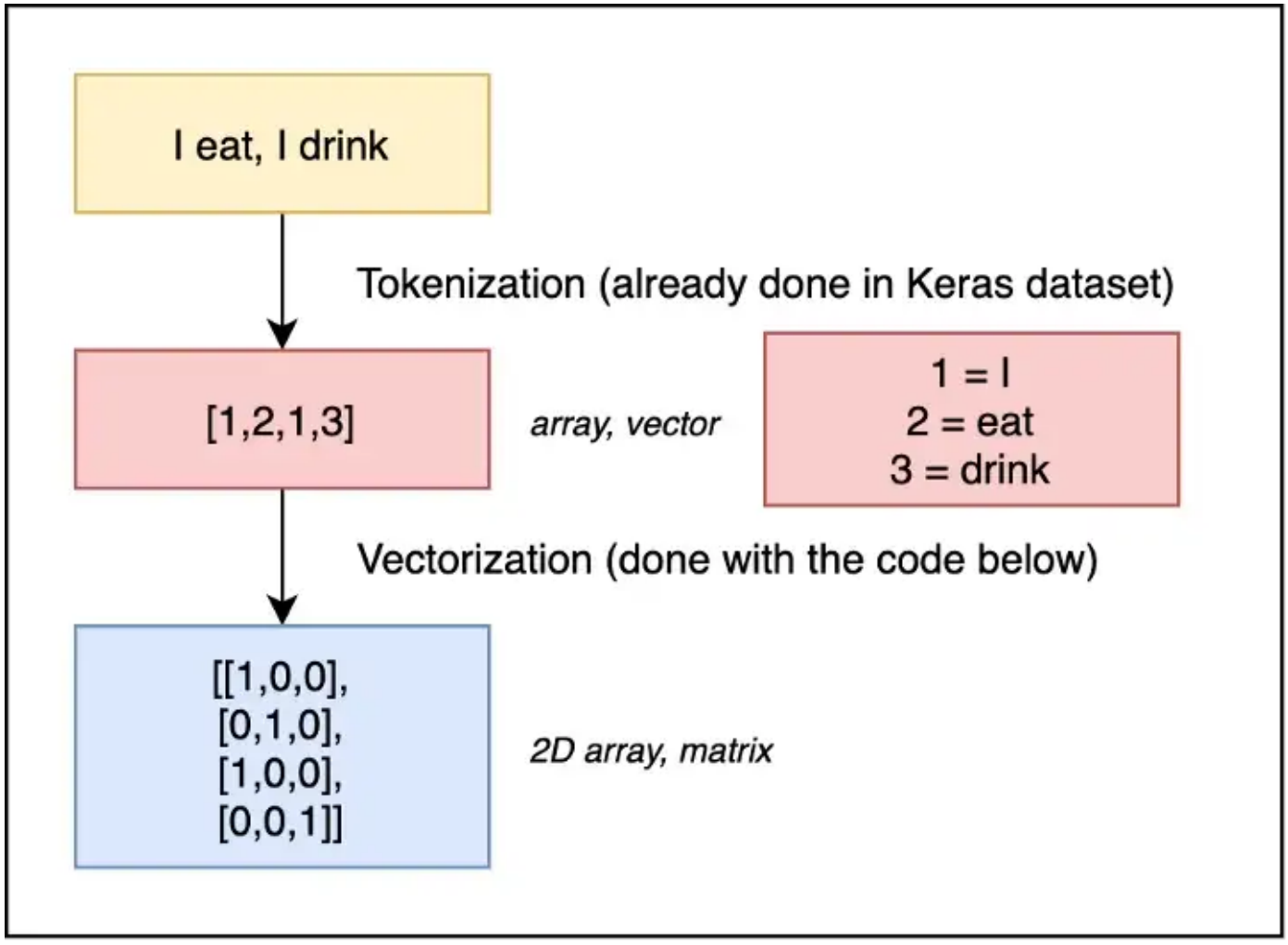
Print a decoded comment:

```
print(decoded_word_index)
```

```
# output
beginsbetterthanitendsfunnythattherussiansubmarinecrewoutperformsallot
heractorsit'slikethosesceneswheredocumentaryshotsbrbrspoilerpartthemes
sagedechiferedwascontrarytothewholestoryitjustdoesnotmeshbrbr
```

Data vectorisation and one-hot encoding

Currently, each sample is an array of integers with different length depending on the text length of the comment. The used algorithms need a tensor where every sample has the same number of features. For this, we one-hot encode our integer list by replacing each number with a vector of a length 10,000 which has the value 1 on the index position and 0 on all other positions. So we basically encode the integer sequence into a binary matrix.



Schematic view on tokenisation and vectorisations of the text (image by author)

For the vectorisation, we write a simple function

```
import numpy as np

def vectorize_sequence(sequences,dimensions):
    results=np.zeros((len(sequences),dimensions))
    for i, sequence in enumerate(sequences):
        results[i,sequence] = 1.
    return results
```

And use it to turn our test and training data into tensors

```
x_train=vectorize_sequence(imdb_train_data,10000)
x_test=vectorize_sequence(imdb_test_data,10000)
```

The label data is already a binary integer and we only have to convert it into a *Numpy* array and float type as *Keras* only accepts this kind of data type

```
y_train=np.asarray(imdb_train_labels).astype('float32')
y_test=np.asarray(imdb_test_labels).astype('float32')
```

A check on the shape of the tensors shows that we have now 25000 samples with 10000 features for the train and test dataset

```
print(x_train.shape)
print(x_test.shape)

#output
(25000, 10000)
(25000, 10000)
```


1. Naive Bayes

The Naive Bayes method is a supervised learning algorithm based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair

of features given the value of the class variable.

```
from sklearn.naive_bayes import MultinomialNB

mnb = MultinomialNB().fit(x_train, y_train)

print("score on test: " + str(mnb.score(x_test, y_test)))
print("score on train: "+ str(mnb.score(x_train, y_train)))
```

Summary: The Naive Bayes algorithm is very fast for this feature rich dataset (remember we have a tensor with 10,000 feature vectors) and already provides a good result of above 80%. The score on the training and test data are close to each other, which indicates that the we are not overfitting.

Results Naive Bayes

run time	2s
train score	0.87
test score	0.84

2. Logistic Regression

Logistic Regression is one of the oldest and most basic algorithms to solve a classification problem:


```
from sklearn.linear_model import LogisticRegression

lr=LogisticRegression(max_iter=1000)
lr.fit(x_train, y_train)

print("score on test: " + str(lr.score(x_test, y_test)))
print("score on train: "+ str(lr.score(x_train, y_train)))
```

Summary: The Logistic Regression takes quite a long time to train and does overfit. That the algorithm overfits can be seen in the deviation of the train data score (98%) to test data score (86%).

```
Results Logistic Regression
+-------------+------+
| run time    | 60s  |
+-------------+------+
| train score | 0.98 |
+-------------+------+
| test score  | 0.86 |
+-------------+------+
```

3. K-Nearest Neighbours

The k-nearest neighbors (KNN) algorithm is a supervised machine learning algorithm that can be used to solve both classification and regression problems. For KNN, it is known that it does not work so well with large datasets (high sample size) and in with many features (high dimensions) in particular. Our dataset with 25000 samples and 10000 features is already not optimal for this algorithm. I had to set two parameters, *algorithm = "brute" and n_jobs=-1* to get the classifier run in a recent time.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(algorithm = 'brute', n_jobs=-1)

knn.fit(x_train, y_train)

print("train shape: " + str(x_train.shape))
print("score on test: " + str(knn.score(x_test, y_test)))
print("score on train: "+ str(knn.score(x_train, y_train)))
```

Summary: As expected, this algorithm is not very well-suited for this kind of prediction problem. It takes 12 minutes, it predicts very poorly with only 62%, and shows overfitting tendencies.

Results K-Nearest Neighbours

run time	12m
train score	0.79
test score	0.62

4. Support Vector Machine

The Support Vector Machine is a simple algorithm for classification and regression tasks. It can provide high accuracy with less computation power very fast. Due to the large number of features, we are using the *LinearSVC.* It turned out that setting the regularisation parameter *C=0.0001* improves the quality of the prediction and reduces overfitting.

```
from sklearn.svm import LinearSVC

svm=LinearSVC(C=0.0001)
svm.fit(x_train, y_train)

print("score on test: " + str(svm.score(x_test, y_test)))
print("score on train: "+ str(svm.score(x_train, y_train)))
```

Summary: The Support Vector Machine is very fast, gives a high score for prediction and shows no overfitting issues.

Results Support Vector Machine

run time	2s
train score	0.86
test score	0.85

5. Decision Tree

A Decision Tree is a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules (if-else) inferred from the data features.

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()
clf.fit(x_train, y_train)

print("score on test: "  + str(clf.score(x_test, y_test)))
print("score on train: " + str(clf.score(x_train, y_train)))
```

Summary: Applying a single decision tree to this feature rich dataset leads to massive overfitting. Indeed, an accuracy of 100% means it has remembered exactly the training dataset and is therefore generalising poorly on the test data. What we can see here is one of the cons of single decision tree which can't handle data with too many features.

Results Decision Tree

run time	4m
train score	1.0
test score	0.70

However, this issue can be addressed by either tuning some parameter of the algorithm or introducing ensemble learning techniques. Here you can find an in-depth article about parameter tuning of the decision tree classifier: https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3. Next we will focus on some common ensemble learning approaches.

6. Bagging Decision Tree (Ensemble Learning I)

When a decision tree overfits, applying an ensemble learning algorithm like bagging might improve the quality of the prediction model. In bagging, the training data is increased by taking bootstraps from the training data. This means multiple samples are taken (with replacement) from the training data and the model is trained on these sub-datasets. The final prediction is the average over all predictions from each bootstrap sample.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# max_samples: maximum size 0.5=50% of each sample taken from the full
dataset

# max_features: maximum of features 1=100% taken here all 10K
# n_estimators: number of decision trees

bg=BaggingClassifier(DecisionTreeClassifier(),max_samples=0.5,max_feat
ures=1.0,n_estimators=10)
bg.fit(x_train, y_train)

print("score on test: " + str(bg.score(x_test, y_test)))
print("score on train: "+ str(bg.score(x_train, y_train)))
```

Summary: The Bagging Classifier is much slower as it basically runs 10 decision trees but one can see a reduction of the overfitting we saw on the single Decision Tree and an increase in the test score. Play around with the parameters to further improve this result.

Results Bagging Decision Tree

run time	10m
train score	0.93
test score	0.77

7. Boosting Decision Tree (Ensemble Learning II)

In general, we can't use Boosting to improve a completely overfitted model with *score = 1*. To apply Boosting, we first have to tweak the decision tree classifier a bit. It took me some trial and error until I got the best parameters for the Decision Tree and the AdaBoost Classifier. I am sure you can further improve by playing around a bit more.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

adb =
AdaBoostClassifier(DecisionTreeClassifier(min_samples_split=10,max_dep
th=4),n_estimators=10,learning_rate=0.6)
adb.fit(x_train, y_train)

print("score on test: " + str(adb.score(x_test, y_test)))
print("score on train: "+ str(adb.score(x_train, y_train)))
```

Summary: With Boosting and the improved underlying, decision tree we got rid of the overfitting. Even though we can only predict with a certainty of around 80%, we can do this solid on all test data.

Results Boosting Decision Tree

run time	5m
train score	0.80
test score	0.78

8. Random Forest (Ensemble Learning III)

The Random Forest Algorithm is another frequently used ensemble learning classifier which uses multiple decision trees. The Random Forest classifier is basically a modified bagging algorithm of a Decision Tree that selects the subsets differently. I found out that *max_depth=9* is a good value for this feature-rich dataset.

```
from sklearn.ensemble import RandomForestClassifier
```

```
# n_estimators = number of decision trees
rf = RandomForestClassifier(n_estimators=30, max_depth=9)
rf.fit(x_train, y_train)

print("score on test: " + str(rf.score(x_test, y_test)))
print("score on train: "+ str(rf.score(x_train, y_train)))
```

Summary: The Random Forest does not overfit and has a prediction score that is comparable to that of the Boosted Decision Tree but it performs much better as it is more than a magnitude faster (15 times faster).

Results Random Forest

run time	20s
train score	0.83
test score	0.80

9. Voting Classifier (Ensemble Learning IV)

This classifier from the ensemble learning toolbox evaluates different classifiers and selects the best out of it

> *The idea behind the* ***VotingClassifier*** *is to combine conceptually different machine learning classifiers and use a majority vote or the average predicted probabilities to predict the class labels. Such a classifier can be useful for a set of equally well performing models in order to balance out their individual weaknesses.*
>
> *(citation from: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html)*

So let's use this classifier to combine some of the models we had so far and apply the Voting Classifier on

- Naive Bayes (84%, 2s)
- Logistic Regression (86%, 60s, overfitting)

- Random Forest (80%, 20s)
- Support Vector Machine (85%, 10s)

Be aware that for this code snippet, all used model definitions have to be loaded in the Python kernel.

```
from sklearn.ensemble import VotingClassifier

# 1) naive bias = mnb
# 2) logistic regression =lr
# 3) random forest =rf
# 4) support vector machine = svm

evc=VotingClassifier(estimators=[('mnb',mnb),('lr',lr),('rf',rf),
('svm',svm)],voting='hard')
evc.fit(x_train, y_train)

print("score on test: " + str(evc.score(x_test, y_test)))
print("score on train: "+ str(evc.score(x_train, y_train)))
```

- **Summary:** Even if it is overfitting a bit, it got the best prediction score on the test data so far.

Results Voting Classifier

run time	2s
train score	0.91
test score	0.86

10. Neural Network (Deep Learning)

Deep learning uses an artificial neural network that uses multiple layers to progressively extract higher level features from the training data. We are using a simple three-layer network without any optimisation, except the usage of a small validation dataset. Here we are using *Keras* instead of *Sklearn.*

```
from keras import layers
from keras import models
from keras import optimizers
from keras import losses
from keras import metrics

# split an additional validation dataset
x_validation=x_train[:1000]
x_partial_train=x_train[1000:]
y_validation=y_train[:1000]
y_partial_train=y_train[1000:]

model=models.Sequential()
model.add(layers.Dense(16,activation='relu',input_shape=(10000,)))
model.add(layers.Dense(16,activation='relu'))
model.add(layers.Dense(1,activation='sigmoid'))
model.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=
['accuracy'])

model.fit(x_partial_train,y_partial_train,epochs=4,batch_size=512,vali
dation_data=(x_validation,y_validation))

print("score on test: " + str(model.evaluate(x_test,y_test)[1]))
print("score on train: "+ str(model.evaluate(x_train,y_train)[1]))
```

Summary: Using a neural network leads to the best test score that we achieved so far, at around 88%. However, the standard settings lead to an overfitting of the training data.

This issue can be handled easily by adding some minor tuning parameters which reduce the training score to 0.90.

In the following, I applied the 3 best practices for handling overfitting in a neural network:

1. reduce the network's size
2. adding some weight regularisation
3. adding dropout

```
from keras import layers
from keras import models
from keras import optimizers
```

```
from keras import losses
from keras import regularizers
from keras import metrics

# add validation dataset
validation_split=1000
x_validation=x_train[:validation_split]
x_partial_train=x_train[validation_split:]
y_validation=y_train[:validation_split]
y_partial_train=y_train[validation_split:]

model=models.Sequential()
model.add(layers.Dense(8,kernel_regularizer=regularizers.l2(0.003),act
ivation='relu',input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(8,kernel_regularizer=regularizers.l2(0.003),act
ivation='relu'))
model.add(layers.Dropout(0.6))
model.add(layers.Dense(1,activation='sigmoid'))
model.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=
['accuracy'])

model.fit(x_partial_train,y_partial_train,epochs=4,batch_size=512,vali
dation_data=(x_validation,y_validation))

print("score on test: " + str(model.evaluate(x_test,y_test)[1]))
print("score on train: "+ str(model.evaluate(x_train,y_train)[1]))
```

Results Deep Learning with Neural Network standard

run time	8s
train score	0.95
test score	0.88

Results Deep Learning with Neural Network optimised

run time	8s
train score	0.90
test score	0.88

Summary

We now trained 10 different machine learning algorithms on the same dataset to solve the same task, predicting a positive or negative review based on a sentiment analysis of

text passages. So let's see who wins in terms of the quality of the prediction and in run time. The parameter overfitting (OF) is just the difference from train score to test score. If it is large, it means we have high overfitting.

No	Algorithm	Time [s]	Train	Test	OF
1	Naive Bayes	**2**	0.87	0.84	0.03
2	Logistic Reg.	60	0.98	**0.86**	**0.12 !!**
3	KNN	720	0.79	0.62	**0.17** !!
4	SVM	**2**	0.86	0.85	0.01
5	Decision Tree	240	1.0	0.70	**0.3 !!**
6	Bagging DT	600	0.93	0.77	**0.16 !!**
7	Boosting DT	300	0.80	0.78	0.02
8	Random Forest	20	0.83	0.80	0.03
9	Voting Classifier	**2**	0.91	**0.86**	0.04
10	**Neuronal Network**	**8**	0.90	**0.88**	0.02

Best Runtime:

In terms of Runtime, the fastest algorithms are **Naive Bayes, Support Vector Machine, Voting Classifier** and the **Neural Network.**

Best Prediction Score:

In terms of the best prediction of the test dataset, the best algorithms are **Logistic Regression, Voting Classifier** and **Neural Network.**

Worst Overfitting:

The algorithms that are overfitting the most are **Logistic Regression, K-Nearest Neighbours, Decision Tree** and **Bagging Decision Tree.**

Best Algorithm:

The overall best approach to solve this text sentiment analysis task is the **Neural Network,** which is fast, has high accuracy and low overfitting. A little bit faster but a bit worse in prediction quality is the **Voting Classifier**.

Jupyter Notebooks

You can find all the code as a Jupyter Notebook in my Github repository: https://github.com/alexortner/teaching/tree/master/binary_classification

Here you will find the same top 10 binary classification algorithms applied to different machine learning problems and datasets.

1. IMDB Dataset — *Natural language processing* — binary sentiment analysis
2. FashionMNIST Dataset — *Computer vision* — binary image classification
3. Wisconsin Breast Cancer Dataset — simple binary classification

Thanks for reading!

I hope you enjoyed this article. In case you see anything that should be corrected, please comment so that I can fix it as soon as possible.

Machine Learning Python Data Science Deep Learning Classification

---