Group S433 (3 Members): Mini Project Report

Abstract

Several predictive models are trained on a dataset of 750 songs, consisting of high-level features, to predict which songs Andreas Lindholm is going to like. Gradient boosting was found to be the most accurate method with a 5-fold cross-validation accuracy of 0.8285, and is thus our chosen method to be used "in production".

1 Introduction

In this report several machine learning algorithms are employed in order to predict which songs Andreas Lindholm is going to like. A dataset of 750 songs, consisting of high-level features of the songs such as acousticness, danceability, energy etc, each of which Andreas has labeled with Like or Dislike, are used to train the predictive models. The models are then evaluated on a second "test" dataset of 200 songs which have not been classified beforehand.

2 Methods

In this section the different machine learning algorithms, or methods, which were explored are described.

2.1 k-Nearest Neighbour

The k-Nearest Neighbor algorithm is the transposition of the old adage "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck" to the domain of machine learning. Given a test input $\mathbf{x}_*$, we compute its distance to all the training inputs $\mathbf{x_i}, i = 1, \ldots, n$, then select the k nearest ones, and, finally, classify $\mathbf{x}_*$ based on the majority class of its k nearest neighbors.

When resorting to a pure majority voting approach, every neighbor has the same impact on the classification, which makes the algorithm sensitive to the choice of k. One way to reduce the impact of k is to add a weight (inversely proportional to the distance to $\mathbf{x}_*$) to the contribution of each nearest neighbor $\mathbf{x_j}, j = 1, \ldots, k$ on the voting. This approach ensures that training inputs located more far away from $\mathbf{x}_*$ have a weaker impact on the classification compared to those located close to $\mathbf{x}_*$.

2.2 Logistic regression

Logistic regression is a binary classification model which takes a linear regression approach and uses the logistic function to generate either 1 or 0 (the binary output, often called y). In the specific case of the music data set 1 corresponds to Andreas liking a song and 0 to him not liking a song. Since the sum of the probabilities of a song either belonging to class 0 or 1 will always be 1 it is sufficient to predict the probability of a song belonging to one, not both, of the classes. With $x = [1, x_1, x_2, ..., x_p]^T$ as input the linear regression is written as:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p = \beta^T x \tag{1}$$

The logistic equation is defined as follows:

$$h(z) = \frac{e^z}{1 + e^z} \tag{2}$$

Preprint. Under review.

Combining equations 1 and 2 we get the following for y belonging to 1:

$$p(y = 1|x) = \frac{e^{\beta^T x}}{1 + e^{\beta^T x}} \tag{3}$$

$\hat{\beta}$ is learned by maximising the β argument in the likelihood function, just like in linear regression.

2.3 Linear and quadratic discriminant analysis

Linear and quadratic discriminant analysis (LDA and QDA respectively) which try to maximize the differences between the classes while at the same time minimizing the variance. The methods model the probability that a set of inputs belong to a given class. The methods do so by making use of the factorization in equation 4

$$p(x, y) = p(x|y)p(y) \tag{4}$$

where x is a numerical value and y is a categorical variable. $p(x|y)$ is assumed to be a Gaussian distribution, i.e.

$$p(x|y) = N(x|\mu, \Sigma) \tag{5}$$

where μ is the mean vector and Σ the covariance matrix which both depend on y. μ is assumed to be different for all classes in both LDA and QDA whereas Σ is assumed to the same of all classes in LDA but different for all classes in QDA.
Since y can be classified into M categories the distribution $p(y)$ is modelled with M parameters

$$\{\pi_m\}_{m=1}^{M} \tag{6}$$

The parameters are learned in the following manner. The relative occurrence of class A in the training data is given as the number of class m occurrences divided by the total number of class occurrences, i.e.

$$\hat{\pi}_m = \frac{n_m}{n} \tag{7}$$

The mean vector μ_m of each class is given by

$$\hat{\mu}_m = \frac{1}{n_m} \sum_{i:y_i=m} x_i \tag{8}$$

In the case of LDA a common covariance matrix is used, given by

$$\hat{\Sigma} = \frac{1}{n - M} \sum_{m=1}^{M} \sum_{i:y_i=m} (x_i - \hat{\mu}_m)(x_i - \hat{\mu}_m)^T \tag{9}$$

but QDA requires that an individual covariance matrix is learned for each class.
When all the parameters are computed, or "learned", predictions can be obtained by computing the probability that a set of inputs belong to a certain class.
The predictive LDA classifier is given as

$$p(y = m|x_*) = \frac{\hat{\pi}_m N(x_*|\hat{\mu}_m, \hat{\Sigma})}{\sum_{j=1}^{M} \hat{}_m N(x_*|\hat{\mu}_j, \hat{\Sigma}_j)} \tag{10}$$

and for QDA it is similar but with individual covariance matrices for each class. The class prediction for a set of inputs is then found as the class for which it is most likely predicted to be.

2.4 Ensemble methods

2.4.1 Random forests

The idea behind random forests is to create a collection of different trees in a way that increases randomness and decreases variance. In order to achieve this goal, bagging is used: each tree is trained on different data, obtained by bootstrapping samples from the dataset. To further add randomness, at each node, the choices that each decision tree can make are limited by forcing it to pick from a random subset of all the features rather than from the whole set.

These two forms of randomness end up reducing the variance without effecting the bias. Another benefit is that there is no need to prune the trees. To set the number of trees in the forest, they keep being added until the error stops decreasing. Once the set of trees are trained, a classification can be reached by a majority vote in the forest.

2.4.2 Boosting

Boosting is an ensemble method that trains several high-bias base models and combines their predictions in such a way that the combination of the models result in improved overall prediction. The bias is thus reduced by forming an ensemble of weak models which together act like one strong model. Like bagging, boosting is both an ensemble method but also a meta-algorithm since the base methods which are combined into a single strong method can be just about any method. The difference is that in bagging the base models are learned in parallel whereas in boosting the base models are learned in sequence in a way such that the next base models tries to improve on the performance with respect to the previous one. This improvement on the base models is done by modifying the training data set in a way as to increase the probability of sampling the data points for which the previous model performed poorly.

2.4.3 AdaBoost

AdaBoost, short for Adaptive Boosting, is a popular boosting algorithm used for binary classifcation. AdaBoost combines multiple high bias "weak learners", most often decision trees consisting of one node and two leaves (stumps), into a single strong model. Each subsequent weak base model is improved by taking the mistakes of the previously constructed into account. The classification prediction of the strong, combined, model is given by a weighted majority vote. Each weak base model votes either +1 or -1 and the prediction is set to 1 if the combined output is positive and -1 if the combined output is negative, i.e.

$$\hat{y}_{boost}^{(B)}(x) = sign\left(\sum_{b=1}^{B} \alpha^{(b)} \hat{y}^{(b)}(x_j)\right) \tag{11}$$

The sequential learners is done by constructing weights for the individual data points. T

$$w_i^{(b)} \triangleq exp\left(-y_i \sum_{j=1}^{b-1} \alpha^{(j)} \hat{y}^{(j)}(x)\right) \tag{12}$$

The weighted misclassification error for the b:th data point is found as

$$E_{train}^{(b)} \triangleq ... = \sum_{i=1}^{n} \frac{w_i^{(b)}}{\sum_{j=1}^{n} w_j^{(b)}} \mathbb{1}\left(y_i \neq \hat{y}^{(b)}(x_i)\right) \tag{13}$$

The optimal value for α is then found as

$$\alpha^{(b)} = \frac{1}{2} ln\left(\frac{1 - E_{train}^{(b)}}{E_{train}^{(b)}}\right) \tag{14}$$

The procedure for learning the AdaBoost model is then to first assign weights $w_i = \frac{1}{n}$ to all n data points. Then for $b = 1, ..., B$ train a weak classifier $\hat{y}^{(b)(x)}$ on the weighted training data. Compute the misclassification training error $E_{train}^{(b)}$, the optimal value for $\alpha(b)$, and then compute the the weights w_i to be used in the training on the next weak learner. This procedure is repeated until B weak learners have been trained. When the weak learners have been trained the predictive capability of the strong model operates according with equation 11, i.e. by a weighted majority vote.

2.4.4 Gradient boosting

Gradient boosting is a general algorithm which ensembles several weak learners into a single strong model, in a sequential fashion just like adaptive boosting. The difference from adaptive boosting is that gradient boosting generalizes further by allowing optimization of any differentiable loss function. The method takes a functional optimization view; a cost function is optimized over the function space in each iteration by choosing a function in the negative gradient direction.
Each sequential learner is constructed with the objective of minimizing

$$J(c) = \frac{1}{n} \sum_{i=1}^{n} L(y_i \cdot \sum_{j=1}^{b-1} \alpha^{(j)} \hat{y}^{(j)}(x)) \tag{15}$$

for the final strong model, where L can be any arbitrary loss function. The idea is to choose $\hat{y}(b)(x)$ such that

$$\hat{y}(b)(x_j) \approx \frac{\partial L\left(y_i \cdot \sum_{j=1}^{b-1} \alpha^{(j)} \hat{y}^{(j)}\right)(x)}{\partial c} \tag{16}$$

for $j = 1, ..., B$. This is learned using regression; $\hat{y}(b)$ takes x_i as input and $\sum_{j=1}^{b-1} \alpha^{(j)} \hat{y}^{(j)}$ as output. Regression trees are commonly used as a base regression model, however the choice is arbitrary. The gradient boosting algorithm works in the following manner. First initialize

$$\alpha^{(j=0)} \hat{y}^{(j=0)}(x_0) \tag{17}$$

as a constant. Then for $b = 1, ..., B$ compute the negative gradient loss function, train a base regression model to fit the gradient values and update the boosted, aggregated, model. The predictive output is then given as

$$\hat{y}_{boost}^{(B)} = sign\left(\sum_{j=1}^{B} \alpha^{(j)} \hat{y}^{(j)}(x)\right) \tag{18}$$

3 Application

3.1 Data pre-processing

Observation points which are distant from other points are known as "outliers". Fitting a classifiers to data containing several outliers in the data may influence the misclassification error on new data in a negative way.

The z-score is a measure of the number of standard deviations away from the mean that a certain data point resides.

Using a threshold of $z \leq 3$, 91 data points areremoved data set. This amounts to 12.1% of the original data points (which were 750 in total) so it could in fact be so that removing such a large share of the data may have a negative effect overall, but this is difficult to know.

3.2 Learning the models

Logistic regression

The class LogisticRegression is imported from scikit-learn and trained on the data. A ROC curve is produced to find a good threshold for the classifier.

k-NN

The data is standardatized using `StandardScaler`, from `sklearn.preprocessing`. Then `GridSearchCV` (with 5-fold cross-validation), from `sklearn.model_selection`, is used in conjunction with `KNeighborsClassifier`, from `sklearn.neighbors`, to arrive at $k = 16$ maximizing the accuracy.

LDA and QDA

Instead of implementing LDA and QDA from scratch, the classes LinearDiscriminantAnalysis and QuadraticDiscriminantAnalysis available by the scikit-learn libraries are used. When it comes to LDA and QDA there doesn't exist any parameters to be varied, or "tuned", in order to improve on the model. Therefore, when the models are learned they are ready to be applied.

Random forest

The random forest method is implemented with the RandomForestClassifier class. The most important parameters to be tuned are the maximum depth of the tree and the number of trees in the forest, i.e. the number of weak learners. The most accurate model is found with the use of grid search, i.e. testing and evaluation different combinations of models with respect to different parameter values. The validation of the accuracies of the models are performed using the K-fold cross validation method.

AdaBoost

AdaBoost is implemented with the use of the class AdaBoostClassifier. The most significant design choice variables are which base estimator to choose, the number of weak learners (of the base model) to train, the learning rate and the maximum tree depth in case that the base model is chosen to be a decision tree. This most accurate solution is found via the use of grid search. The validation of the accuracies of the models are performed using the K-fold cross validation method.

Gradient boosting

Similarly to the AdaBoost method the gradient boosting method is implemented using the class GradientBoostingClassifier. The most important tuning parameters are the number of weak learners, the learning rate and the maximum depth. The tuning to find the most accurate model with respect to varying the parameters is performed using a grid search. The accuracies of the models are assessed using the K-folk cross validation method.

3.3 Cross validation

K-fold cross validation is used to get a good estimate of how well each model performs. The data are split into a training set and a test set and the model is trained "as usual". In k-fold cross validation this is done k times. The data is split into k sections and each section "takes turns" being the test data and the rest being training data. The model gets trained on the training data each time since the training data changes. The accuracy of the model is evaluated for each of the k runs. The average of all accuracies is then taken to be the final accuracy of the cross validation.

3.4 Performance measures

The accuracy measure, defined as

$$a = \frac{TP + TN}{TP + FP + TN + FN} \tag{19}$$

where, TP/TN are the true positives/negatives and FP/FN are the false positives/negatives, is an obvious choice to compare the performance of classifiers. However, since it treats every class as equally important, it might not always be the most adequate, especially for data sets with imbalanced class distributions. Two other metrics relevant in applications where the successful detection of one of the classes is more significant than the detection of other classes are precision and recall:

$$p = \frac{TP}{TP + FP} \tag{20}$$

$$r = \frac{TP}{TP + FN} \tag{21}$$

Precision determines the fraction of records that actually turns out to be positive in the group the classifier has declared as a positive class. The higher the precision is, the lower the number of false positive errors committed by the classifier. Recall measures the fraction of positive examples correctly predicted by the classifier. Both these metrics can be summarized by the F_1 score, which represents their harmonic mean:

$$F_1 = \frac{2}{\frac{1}{r} + \frac{1}{p}} \tag{22}$$

A high F_1 score guarantees that both precision and recall are reasonably high.

4 Evaluation

In this section the accuracies of the most accurate classifiers for each method are presented. The mean accuracy is computed as the average accuracy for a 5-fold cross validation, and presented in table 1.

The gradient boosting algorithm with a maximum tree depth of 4, 76 weak learners and a learning rate of 0.08 is found to be the most accurate method with a mean accuracy of 0.8285 for a 5-hold cross validation and 0.805 when applied on the "test data set".

Classifier	Accuracy	Classifier	Accuracy
Logistic regression	0.6479	kNN	0.8115
LDA	0.8102	QDA	0.7572
Random forest	0.8224	AdaBoost	0.8224
Gradient boosting	0.8285		

Table 1: Classifiers and respective accuracy

5 Conclusion

Looking at Table 1, it can be seen that Gradient Boost is the most accurate classifier and is thus our choice of method for use "in production".

6 Ethical discussion

In this section the discussion regarding task (b) is presented. As machine learning engineers we understand that there is risk that prejudices and cultural biases are repeated or amplified in a model. These biases may be subtle or not, but the question is whether we have a responsibility to inform and educate or clients (or end users) about these risks? Furthermore, in a general should machine learning engineers make sure that their solutions, or products, are carefully checked for such biases?

It can be argued that the greater role of any engineer, no matter the discipline, is to improve the quality of life of humankind, even if ever so slightly, by the use of technology. If that is true, then providing an insure company with a product which may reduce the quality of life of many people due to introducing unjustifiable biases without their knowledge is immoral. As an example, the delivered machine learning solution or software, may find unethical ways of charging more for insurance premiums to certain groups by making inferences which should not be done at all from an ethical standpoint, as it violates the right to equality and non-discrimination. Another strong reason to inform and educate the client about the possibility of these biases being introduced is the legal repercussions of them. In many countries, discriminating against race, gender, sexual-orientation, etc., is illegal, so by using a machine learning model that is biased towards one or more of these attributes, a company could be breaking the law.

However, in a general sense, if machine learning engineers are to be held morally responsible for all possible prejudices and cultural biases which may be repeated and amplified in a model, that may be limiting to the growth of machine learning applications and in the end be a detriment to the well-being of human-kind as a whole. People may perhaps not dare to deploy their models on real data for fear of being held responsible if something bad happens. Furthermore, if this is the viewpoint of the society this may be written into laws requiring a strict control of the deployment of models, which might hinder the laymen, non-professionals or even smaller companies from exploring the possibilities of machine learning applications. Another reason not to mention these possible machine learning biases to our fictitious insurance company client is the fact that correcting the biases would most likely mean a reduction in profit for the client, which would devalue the product we have developed.

In summary, our personal opinion is that machine learning engineers should not only have a moral responsibility of making sure their solutions are carefully checked for machine-learning biases, they should in fact be forced to do it by appropriate legislation. Deregulation leads to chaos, inequality, and, ultimately, suffering for the most vulnerable in society.

References

[1] Marsland, S. (2014) . *Machine Learning - An Algorithmic Perspective*, London: London: Chapman Hall/CRC.

[2] Lindholm, A., Wikström, J., Lindsten F., Schön, Thomas B. (2019)
Supervised Machine Learning - Lecture notes for the statistical machine learning course, Department of information tachnology, Uppsala University

Appendices

Loading data

```
url = 'http://www.it.uu.se/edu/course/homepage/sml/project/training_data.csv'
songs = pd.read_csv(url)
```

Data pre-processing: removing outliers

```
from scipy import stats
z = np.abs(stats.zscore(songs)) # Creating z matrix, first array corresponds to row
songs_cleaned = songs[(z < 3).all(axis=1)] # Cleaning dataset by only keeping non-outliers

X_cleaned = songs_cleaned.drop(columns=['label'])
y_cleaned = songs_cleaned['label']

X = X_cleaned
y=y_cleaned
```

Logistic regression

```
model = skl_lm.LogisticRegression()
model.fit(X_train, y_train)
model.predict(X_train)
prediction_probs = model.predict_proba(X_test)

#ROC
false_positive_rate = []
true_positive_rate = []
prediction_probs = model.predict_proba(X_test)
like_index = np.argwhere(model.classes_ == 1).squeeze()
threshold = np.linspace(0.00, 1, 101)
actual_likes = np.sum(y_test == 1)
actual_dislikes = np.sum(y_test == 0)

for t in threshold:
    prediction = np.where(prediction_probs[:, like_index] > t, 'like', 'dislike')

    FP = np.sum((prediction == 'like') & (y_test == 0))
    TP = np.sum((prediction == 'like') & (y_test == 1))

    false_positive_rate.append(FP/actual_dislikes)
    true_positive_rate.append(TP/actual_likes)

plt.plot(false_positive_rate, true_positive_rate)
for i in [0, 1, 10, 50, 98, 100]:
  plt.text(false_positive_rate[i], true_positive_rate[i], f"t={threshold[i]:.2f}")
plt.ylabel('true positive rate')
plt.xlabel('false positive rate')
plt.show()

prediction = np.where(prediction_probs[:, like_index] > 0.6, 1, 0)

acc = accuracy_score(y_test, prediction)
print(f"accuracy: {acc}")

#cross-validation
scores = cross_val_score(model, X, y, cv=10, scoring='accuracy')
print(scores)
meanScore = np.mean(scores)
print(meanScore)
```

k-NN code

```
# Grid Search
scaler = skl_pre.StandardScaler().fit(X)
model_gscv = skl_nb.KNeighborsClassifier()
param_grid = {"n_neighbors": np.arange(1, 51)}

knn_gscv = skl_msel.GridSearchCV(model_gscv, param_grid, cv=5)
knn_gscv.fit(scaler.transform(X), y)
print("best param: {} | best acc: {:3.2f}".format(knn_gscv.best_params_,knn_gscv.best_score_))

# k-NN classifier
k_best = 16

X_train, X_test, y_train, y_test = skl_msel.train_test_split(X, y, test_size=0.2)
scaler = skl_pre.StandardScaler().fit(X_train)
model = skl_nb.KNeighborsClassifier(n_neighbors=k_best)
model.fit(scaler.transform(X_train), y_train)
y_pred = model.predict(scaler.transform(X_test))

# Calculate the error in the predictions
print("Mean squared error: {:3.2f}".format(skl_met.mean_squared_error(y_test, y_pred)))
print("Mean absolute error: {:3.2f}".format(skl_met.mean_absolute_error(y_test, y_pred)))
print("Accuracy: {:3.2f}".format(skl_met.accuracy_score(y_test, y_pred)))

target_names=['Like','Dislike']
report = skl_met.classification_report(y_test, y_pred, target_names=target_names)
print(report)

cf_matrix = skl_met.confusion_matrix(y_test, y_pred)
print("Confusion matrix:\n",np.array_str(cf_matrix))
```

LDA code

```
# LDA Classifier
clf = skl_da.LinearDiscriminantAnalysis()
accuracy = np.mean(cross_val_score(clf, X, y, cv=5))
print(accuracy)
```

QDA code

```
# QDA Classifier
clf = skl_da.QuadraticDiscriminantAnalysis()
accuracy = np.mean(cross_val_score(clf, X, y, cv=5))
print(accuracy)
```

Random Forest Code

```
# Tuning / finding best model by use of grid search
N=np.arange(1,100,1)
D=np.arange(1,10,1)
accuracy=np.zeros([len(N),len(D)])
for a,n in enumerate(D,start=0):
  for b,d in enumerate(D,start=0):
    clf = RandomForestClassifier(max_depth=int(d), n_estimators=int(n))
    accuracy[a,b]=np.mean(cross_val_score(clf, X, y, cv=5))

# The max value
max_accuracy=accuracy.max()
indices = np.where(accuracy == accuracy.max())

if len(indices[0])>=1:
  num_learners_idx=indices[0][0]
  max_depth_idx=indices[1][0]
```

```
else:
  num_learners_idx=indices[0]
  max_depth_idx=indices[1]

n_l = N[num_learners_idx]
m_d = D[max_depth_idx]

print(f"Max accuracy: {max_accuracy} \n \n Parameters: \n Base model: Decision tree \n Max depth: {m_d} \n

# Constructing and training the best model
clf_best = RandomForestClassifier(max_depth=m_d, n_estimators=n_l)
clf_best.fit(X,y)
```

6.1 AdaBoost Code

```
# AdaBoost

# Tuning / finding best model by use of grid search
N=np.arange(40,100,2)
D=np.arange(2,8,2)
R=np.arange(0.02,0.10,0.02)
accuracy=np.zeros((len(N),len(D),len(R)))
for a,n in enumerate(N,start=0):
  for b,d in enumerate(D,start=0):
    for c,r in enumerate(R,start=0):
      clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=int(d)), n_estimators=int(n)
      accuracy[a,b,c]=np.mean(cross_val_score(clf, X, y, cv=5))

# The max value
max_accuracy=accuracy.max()
indices = np.where(accuracy == accuracy.max())

if len(indices[0])>=1:
  num_learners_idx=indices[0][0]
  max_depth_idx=indices[1][0]
  learning_rate_idx=indices[2][0]
else:
  num_learners_idx=indices[0]
  max_depth_idx=indices[1]
  learning_rate_idx=indices[2]

n_l = N[num_learners_idx]
m_d = D[max_depth_idx]
l_r = R[learning_rate_idx]

print(f"Max accuracy: {max_accuracy} \n \n Parameters: \n Base model: Decision tree \n Max depth: {m_d} \n

# Constructing and training the "best" model
clf_best = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=m_d), n_estimators=n_l, learni
clf_best.fit(X,y)
```

6.2 Gradient boosting code

```
# Gradient boosting

# Tuning / finding best model by use of grid search
N=np.arange(40,100,2)
D=np.arange(2,8,2)
R=np.arange(0.02,0.10,0.02)
accuracy=np.zeros((len(N),len(D),len(R)))
for a,n in enumerate(N,start=0):
  for b,d in enumerate(D,start=0):
    for c,r in enumerate(R,start=0):
```

```
      clf = GradientBoostingClassifier(n_estimators=n, learning_rate=r, max_depth=d)
      accuracy[a,b,c]=np.mean(cross_val_score(clf, X, y, cv=5))

# The max value
max_accuracy=accuracy.max()
indices = np.where(accuracy == accuracy.max())

if len(indices[0])>=1:
  num_learners_idx=indices[0][0]
  max_depth_idx=indices[1][0]
  learning_rate_idx=indices[2][0]
else:
  num_learners_idx=indices[0]
  max_depth_idx=indices[1]
  learning_rate_idx=indices[2]

n_l = N[num_learners_idx]
m_d = D[max_depth_idx]
l_r = R[learning_rate_idx]

print(f"Max accuracy: {max_accuracy} \n \n Parameters: \n Base model: Decision tree \n Max depth: {m_d} \n

# Constructing and training the "best" model
clf_best = GradientBoostingClassifier(n_estimators=n_l, learning_rate=l_r, max_depth=m_d)
clf_best.fit(X,y)
```