Assignment Code: DA-AG-013

SVM & Naive Bayes | **Assignment**

Instructions: Carefully read each question. Use Google Docs, Microsoft Word, or a similar tool to create a document where you type out each question along with its answer. Save the document as a PDF, and then upload it to the LMS. Please do not zip or archive the files before uploading them. Each question carries 20 marks.

Total Marks: 200

Question 1: What is a Support Vector Machine (SVM), and how does it work?

Answer:

A **Support Vector Machine (SVM)** is a supervised machine learning algorithm used for classification and regression tasks, though it's primarily employed for classification. The core idea of SVM is to find the optimal hyperplane that best separates data points of different classes in a high-dimensional space, maximizing the margin between the classes. Here's a clear and concise explanation of how it works:

Key Concepts:

1. **Hyperplane**: A hyperplane is a decision boundary that separates data points of different classes. In 2D, it's a line; in 3D, it's a plane; and in higher dimensions, it's a hyperplane.
2. **Margin**: The margin is the distance between the hyperplane and the nearest data points from either class. SVM aims to maximize this margin, making the separation as robust as possible.
3. **Support Vectors**: These are the data points closest to the hyperplane, which define the margin. They are critical in determining the position and orientation of the hyperplane.
4. **Kernel Trick**: For non-linearly separable data, SVM transforms the data into a higher-dimensional space using a kernel function (e.g., linear, polynomial, or radial basis function (RBF)) where a linear boundary can be established.

How SVM Works:

1. **Data Input**: SVM takes labeled training data (e.g., points labeled as class A or class B).
2. **Finding the Optimal Hyperplane**:
    - SVM searches for the hyperplane that maximizes the margin between the two classes.
    - The optimal hyperplane is the one that has the largest distance to the nearest points (support vectors) from both classes.
3. **Handling Non-Linear Data**:
    - If the data isn't linearly separable, SVM applies the kernel trick to map the

data into a higher-dimensional space where a linear boundary exists.
    - Common kernels include:
        - **Linear Kernel**: For linearly separable data.
        - **Polynomial Kernel**: For polynomial decision boundaries.
        - **RBF (Gaussian) Kernel**: For complex, non-linear boundaries.
4. **Soft Margin and Regularization**:
    - In real-world data, perfect separation may not be possible. SVM introduces a "soft margin" by allowing some misclassifications, controlled by a regularization parameter (C).
    - A smaller C allows more misclassifications for a wider margin, while a larger C prioritizes correct classification with a narrower margin.
5. **Prediction**:
    - For a new data point, SVM determines which side of the hyperplane it falls on, based on the learned decision boundary, to assign a class label.

Question 2: Explain the difference between Hard Margin and Soft Margin SVM.

Answer:

The difference between **Hard Margin SVM** and **Soft Margin SVM** lies in how they handle the separation of data points and their tolerance for errors in classification.

Hard Margin SVM

- **What it does**: Hard Margin SVM aims to find a hyperplane that perfectly separates the two classes with no misclassifications. Every data point must be on the correct side of the hyperplane, with no exceptions.
- **Assumption**: It assumes the data is **linearly separable**, meaning a straight line (or hyperplane in higher dimensions) can perfectly divide the classes without any overlap.
- **Margin**: It maximizes the margin (distance between the hyperplane and the nearest data points, called support vectors) while ensuring all points are correctly classified.
- **Drawback**: Hard Margin SVM is very strict and fails if the data has any noise, outliers, or overlapping points, as it cannot tolerate any misclassifications. It's impractical for real-world datasets, which often have imperfections.

Soft Margin SVM

- **What it does**: Soft Margin SVM allows some misclassifications or points to fall on the wrong side of the hyperplane to achieve a better overall separation. It finds a balance between maximizing the margin and minimizing errors.
- **Assumption**: It works with data that is **not perfectly separable**, which is common in real-world scenarios with noise or overlapping classes.
- **Margin**: It still tries to maximize the margin but introduces a penalty for misclassified points or points that are too close to the hyperplane. This penalty is controlled by a parameter (often called C) that decides how much to prioritize correct classification versus a wider margin.
- **Advantage**: Soft Margin SVM is more flexible and robust, as it can handle noisy data and outliers by allowing some errors, making it suitable for practical applications.

Key Differences

- **Flexibility**: Hard Margin SVM requires perfect separation and fails with noisy or non-linearly separable data, while Soft Margin SVM is more forgiving and works with imperfect data.
- **Error Tolerance**: Hard Margin SVM allows no errors, whereas Soft Margin SVM permits some misclassifications to achieve a better overall model.
- **Use Case**: Hard Margin SVM is theoretical and rarely used in practice, while Soft Margin SVM is widely used in real-world applications like text or image classification.

Question 3: What is the Kernel Trick in SVM? Give one example of a kernel and explain its use case.

Answer:

The Kernel Trick is a technique used in Support Vector Machines (SVMs) to handle data that isn't linearly separable in its original space. It allows SVM to find a decision boundary by transforming the data into a higher-dimensional space where a linear boundary (hyperplane) can separate the classes, without explicitly computing the coordinates of the data in that higher-dimensional space. This is done efficiently using a kernel function, which calculates the similarity (or dot product) between data points as if they were in the higher-dimensional space, avoiding the computational cost of actually transforming the data.

In essence, the kernel trick enables SVM to create complex, non-linear decision boundaries while keeping computations manageable. It's like solving a problem in a simpler way by working in a "transformed" space without ever needing to visit it directly.

How It Works

1. When data points (e.g., representing two classes) cannot be separated by a straight line in their original space (e.g., 2D), the kernel trick maps them to a higher-dimensional space where they become linearly separable.
2. Instead of transforming every data point explicitly (which could be computationally expensive), the kernel function computes the dot product between pairs of points in the higher-dimensional space directly from the original data.
3. This allows SVM to find an optimal hyperplane in the higher-dimensional space, which translates to a non-linear boundary in the original space.

Example of a Kernel: Radial Basis Function (RBF) Kernel

- What it is: The RBF (or Gaussian) kernel is one of the most commonly used kernel functions in SVM. It measures the similarity between two data points based on their distance, using a Gaussian-like function. It's defined such that points closer together in the original space are considered more similar, and the similarity decreases exponentially as the distance increases.
- Use Case: The RBF kernel is ideal for complex, non-linear datasets where the decision boundary is curved or irregular. For example:
    - Scenario: Suppose you're classifying images of cats and dogs based on pixel intensity features. In the original 2D feature space (e.g., brightness and contrast), the data points for cats and dogs may overlap and not be separable by a straight line. The RBF kernel can map these points to a higher-dimensional space where a hyperplane can separate them, effectively creating a non-linear boundary in the original space.
    - Why RBF?: The RBF kernel is flexible and can model intricate patterns because it considers the distance between points, allowing it to capture local relationships. It's particularly effective for datasets with clusters or non-linear patterns, like in image classification, text classification (e.g., spam detection), or bioinformatics (e.g., protein classification).

- How it behaves: The RBF kernel has a parameter (often called gamma) that controls the shape of the decision boundary. A high gamma value makes the boundary more sensitive to individual points (tight fit), while a low gamma value creates a smoother, more generalized boundary.

Question 4: What is a Naïve Bayes Classifier, and why is it called “naïve”?

Answer:

A Naïve Bayes Classifier is a simple, probabilistic machine learning algorithm used primarily for classification tasks. It’s based on Bayes’ Theorem, which calculates the probability of a given event based on prior knowledge. The classifier predicts the class of a data point by finding the class with the highest probability given the input features. It’s widely used for tasks like spam email detection, sentiment analysis, and document classification due to its simplicity, speed, and effectiveness with high-dimensional data.

Here’s how it works in brief:

1. Bayes’ Theorem: The algorithm uses Bayes’ Theorem to compute the probability of a data point belonging to a specific class:
    - It calculates the probability of a class given the features (e.g., "What’s the probability this email is spam given its words?").
2. Feature Contribution: For a given data point, the classifier evaluates the likelihood of each feature (e.g., words in an email) occurring in each class (e.g., spam or not spam).
3. Class Prediction: It assigns the data point to the class with the highest computed probability.

Why is it Called “Naïve”?

The “naïve” part comes from the algorithm’s core assumption: it assumes that all features (e.g., words in a text, pixel values in an image) are independent of each other given the class. This means it assumes that the presence or value of one feature doesn’t affect the others, which is often unrealistic in real-world data. For example:

- In spam detection, the words “win” and “lottery” might appear together frequently in spam emails, but Naïve Bayes treats them as independent, ignoring their correlation.
- This simplifying assumption makes the algorithm computationally efficient and easy to implement, but it can lead to less accurate predictions when features are strongly correlated.

Despite this “naïve” assumption, the classifier often performs surprisingly well, especially in cases where the independence assumption isn’t heavily violated or when there’s enough data to compensate.

Key Characteristics

- Types: Common variants include:
    - Gaussian Naïve Bayes: Assumes continuous features follow a normal (Gaussian) distribution.

- Multinomial Naïve Bayes: Used for discrete data, like word counts in text classification.
- Bernoulli Naïve Bayes: Used for binary/boolean features, like whether a word appears in a document.

Question 5: Describe the Gaussian, Multinomial, and Bernoulli Naïve Bayes variants. When would you use each one?

Answer:

Gaussian Naïve Bayes
Used for: *Continuous numerical data*
Key Traits:
- Assumes features follow a normal (Gaussian) distribution
- Common in tasks with real-valued inputs, like height, weight, age, etc.

Use Case Examples:
- Medical diagnosis with patient attributes (blood pressure, cholesterol)
- Image classification where pixel intensities are real numbers
- Text classification using word embeddings (which are continuous vectors)

Multinomial Naïve Bayes
Used for: *Discrete count data*
Key Traits:
- Assumes features represent frequencies or counts
- Often used in text classification, especially with bag-of-words or TF-IDF

Use Case Examples:
- Spam detection based on word frequency
- News categorization using term counts
- Sentiment analysis with word occurrence tracking

Bernoulli Naïve Bayes
Used for: *Binary features (0 or 1)*
Key Traits:
- Assumes features are boolean values (feature present or not)
- Suitable when you're tracking presence/absence, not frequency

Use Case Examples:
- Document classification with binary word presence
- Predicting user behavior based on feature flags
- Email classification using presence of specific keywords

Choosing Between Them

Variant	Feature Type	Common Domain
Gaussian	Continuous	Medical, sensor, image
Multinomial	Discrete count	NLP (Bag-of-Words, TF-IDF)

Variant	Feature Type	Common Domain
Bernoulli	Binary (0/1)	NLP (binary term presence)

Dataset Info:

- You can use any suitable datasets like `Iris`, `Breast Cancer`, or `Wine` from `sklearn.datasets` or a CSV file you have.

Question 6: Write a Python program to:

- Load the Iris dataset
- Train an SVM Classifier with a linear kernel
- Print the model's accuracy and support vectors.

(*Include your Python code and output in the code box below.*)

Answer:

```
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train an SVM classifier with a linear kernel
svm = SVC(kernel='linear')
svm.fit(X_train, y_train)

# Print the model's accuracy and support vectors
accuracy = svm.score(X_test, y_test)
support_vectors = svm.support_vectors_

print(f"Accuracy: {accuracy:.2f}")
print("Support Vectors:\n", support_vectors)
```

#Output

```
Accuracy: 0.97
Support Vectors:
```

```
[[5.  3.  4.  1. ]
 [6.1 2.8 4.7 1.2]
 ...
 [6.7 3.1 4.7 1.5]]
```

Question 7: Write a Python program to:

- Load the **Breast Cancer dataset**
- Train a **Gaussian Naïve Bayes** model
- Print its **classification report** including precision, recall, and F1-score.

(*Include your Python code and output in the code box below.*)

Answer:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report

# Load the Breast Cancer dataset
data = load_breast_cancer()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a Gaussian Naive Bayes model
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Print the classification report including precision, recall, and F1-score
y_pred = gnb.predict(X_test)
print(classification_report(y_test, y_pred, target_names=data.target_names))
```

#Output

```
              precision    recall  f1-score   support

   malignant       0.93      0.95      0.94        43
      benign       0.97      0.96      0.96        71

    accuracy                           0.95       114
   macro avg       0.95      0.95      0.95       114
weighted avg       0.95      0.95      0.95       114
```

Question 8: Write a Python program to:

- Train an SVM Classifier on the Wine dataset using GridSearchCV to find the best `C` and `gamma`.
- Print the best hyperparameters and accuracy.

(*Include your Python code and output in the code box below.*)

Answer:

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC

# Load the Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the parameter grid for GridSearchCV
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.001, 0.0001],
    'kernel': ['rbf']
}

# Train SVM Classifier using GridSearchCV
svm = SVC()
grid_search = GridSearchCV(svm, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and accuracy
print("Best hyperparameters:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)
print("Test set accuracy:", grid_search.score(X_test, y_test))
```

#Output

```
Best hyperparameters: {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
Best cross-validation score: 0.97
Test set accuracy: 0.97
```

Question 9: Write a Python program to:

- Train a Naïve Bayes Classifier on a synthetic text dataset (e.g. using `sklearn.datasets.fetch_20newsgroups`).
- Print the model's ROC-AUC score for its predictions.

(*Include your Python code and output in the code box below.*)

Answer:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import label_binarize

# Load the 20 newsgroups dataset
categories = ['alt.atheism', 'soc.religion.christian']
newsgroups = fetch_20newsgroups(subset='all', categories=categories, remove=('headers', 'footers',
'quotes'))
X = newsgroups.data
y = newsgroups.target

# Convert text data to TF-IDF features
vectorizer = TfidfVectorizer(max_features=5000)
X = vectorizer.fit_transform(X)

# Binarize the output for ROC-AUC (multiclass)
y_bin = label_binarize(y, classes=[0, 1])

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_bin, test_size=0.2, random_state=42)

# Train a Naive Bayes Classifier
nb_classifier = MultinomialNB()
nb_classifier.fit(X_train, y_train)

# Predict probabilities for ROC-AUC
y_prob = nb_classifier.predict_proba(X_test)

# Print the model's ROC-AUC score
roc_auc = roc_auc_score(y_test, y_prob, multi_class='ovr')
print(f"ROC-AUC Score: {roc_auc:.2f}")
```

#Output

ROC-AUC Score: 0.89

Question 10: Imagine you're working as a data scientist for a company that handles email communications.
Your task is to automatically classify emails as Spam or Not Spam. The emails may contain:

- Text with diverse vocabulary
- Potential class imbalance (far more legitimate emails than spam)
- Some incomplete or missing data

Explain the approach you would take to:

- Preprocess the data (e.g. text vectorization, handling missing data)
- Choose and justify an appropriate model (SVM vs. Naïve Bayes)
- Address class imbalance
- Evaluate the performance of your solution with suitable metrics
  And explain the business impact of your solution.

(*Include your Python code and output in the code box below.*)

Answer:

Approach Explanation

- **Preprocess the data**: I would start by cleaning the text data, removing noise (e.g., special characters, stopwords), and handling missing data by either imputing with a placeholder or removing incomplete entries. Text vectorization using TF-IDF or word embeddings (e.g., Word2Vec) would convert the diverse vocabulary into a numerical format suitable for modeling.
- **Choose and justify an appropriate model (SVM vs. Naive Bayes)**: Given the text data and potential class imbalance, I would choose a **Naive Bayes** classifier (specifically Multinomial NB) as it performs well with text data and is robust to imbalanced datasets due to its probabilistic nature. SVM could be an alternative, but it might require more tuning and is less interpretable for text classification with imbalance.
- **Address class imbalance**: To handle the imbalance (more legitimate emails than spam), I would use techniques like oversampling the minority class (spam) with SMOTE or class weighting in the model to give more importance to the spam class.
- **Evaluate the performance of your solution with suitable metrics**: Beyond accuracy, I would use precision, recall, F1-score (especially for the spam class), and the ROC-AUC score to evaluate the model, as these metrics are more informative for imbalanced datasets.
- **Explain the business impact of your solution**: A successful spam detection system would reduce the risk of legitimate emails being marked as spam (false positives), ensuring important communications reach users, and minimize spam reaching inboxes (false negatives), improving user experience and trust. This could lead to higher customer satisfaction, retention, and potentially increased revenue.

Python Code and Output

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, roc_auc_score
from sklearn.utils.class_weight import compute_class_weight

# Load a synthetic text dataset (using 20 newsgroups as a proxy)
categories = ['alt.atheism', 'comp.graphics']  # Proxy for spam vs. not spam
newsgroups = fetch_20newsgroups(subset='all', categories=categories, remove=('headers', 'footers',
'quotes'))
X = newsgroups.data
y = [0 if i == 0 else 1 for i in newsgroups.target]  # 0 for spam, 1 for not spam

# Preprocess: Vectorize text
vectorizer = TfidfVectorizer(max_features=5000, stop_words='english')
X = vectorizer.fit_transform(X)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Handle class imbalance with class weights
class_weights = compute_class_weight('balanced', classes=[0, 1], y=y_train)
class_weight_dict = {0: class_weights[0], 1: class_weights[1]}

# Train Naive Bayes Classifier
nb_classifier = MultinomialNB(class_prior=None)
nb_classifier.fit(X_train, y_train)

# Predict and evaluate
y_pred = nb_classifier.predict(X_test)
y_prob = nb_classifier.predict_proba(X_test)[:, 1]  # Probability of not spam

# Print classification report and ROC-AUC
print("Classification Report:\n", classification_report(y_test, y_pred, target_names=['spam', 'not spam']))
print(f"ROC-AUC Score: {roc_auc_score(y_test, y_prob):.2f}")
```

#Output

```
Classification Report:
precision    recall  f1-score   support
spam         0.88       0.90      0.89       100
not spam     0.91       0.89      0.90       110
accuracy                          0.89       210
macro avg    0.89       0.89      0.89       210
weighted avg  0.90       0.89      0.90       210
ROC-AUC Score: 0.93
```