SVM & Naive Bayes | Assignment

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 that classifies data by finding the optimal boundary (called a hyperplane) that separates different classes with the maximum margin. It's widely used for classification tasks and can also handle regression problems.

How SVM Works Step by Step

1. **Data Representation**
    - Each data point is plotted in an n-dimensional space (where *n* = number of features).
    - For example, if you have two features, the data points are plotted in 2D space.
2. **Finding the Hyperplane**
    - SVM tries to find a line (in 2D), plane (in 3D), or hyperplane (in higher dimensions) that best separates the classes.
    - The "best" hyperplane is the one that maximizes the **margin** — the distance between the hyperplane and the nearest data points from each class (called **support vectors**).
3. **Support Vectors**
    - These are the critical data points closest to the hyperplane.
    - They "support" or define the boundary. Removing them would change the position of the hyperplane.
4. **Linear vs. Nonlinear Separation**
    - If the data is linearly separable, SVM finds a straight hyperplane.
    - If not, SVM uses the **kernel trick** (e.g., polynomial, radial basis function) to project data into a higher-dimensional space where separation is possible.
5. **Decision Function**

- Once trained, the SVM predicts the class of new data points by checking which side of the hyperplane they fall on.

Why SVM is Powerful

- **High accuracy** on small to medium datasets.
- **Effective in high-dimensional spaces** (e.g., text classification, genomics).
- **Robust to overfitting** when tuned properly.
- **Versatile**: works for both linear and non-linear problems.

Real-World Example

In healthcare, an SVM could classify whether a patient has a disease based on lab results. The algorithm would find the boundary that best separates "disease" vs. "no disease" cases, ensuring minimal misclassification.

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

Answer: Hard Margin SVM requires perfectly separable data with no misclassifications, while Soft Margin SVM allows some errors to improve generalization on noisy or overlapping datasets**.**

- **Core idea:**
  Hard margin demands a perfect split—no mistakes allowed.
- Soft margin accepts a few mistakes to get a boundary that generalizes better.
- **Data requirements:**
  Hard margin: Works only when classes are perfectly separable.
  Soft margin: Works when data is messy, noisy, or overlapping.
- **Tolerance to errors**:
  Hard margin: Zero tolerance—every point must be on the correct side of the margin.
- Soft margin: Some tolerance—violations are allowed but penalized.
- **Outliers and noise:**
  Hard margin: Very sensitive—one outlier can break the model.
  Soft margin: More robust—outliers have limited influence.
- **How it's controlled:**
  Hard margin: No slack variables, just maximize margin.
  Soft margin: Uses slack variables \xi _i and the C parameter to balance margin vs. errors.
    - Large C: Fewer errors, tighter fit (risk overfitting).
    - Small C: More tolerance, wider margin (better generalization).
- **When to use which:**
  Use hard margin only for clean, perfectly separable datasets (rare in practice).
  Use soft margin for real-world data with noise or overlaps—typically the default choice.
- **Quick example:**
  Hard margin: Two clusters with a clear gap; a straight line cleanly separates them.
  Soft margin: Overlapping clusters; the model allows a few points on the "wrong" side to keep a wider, more reliable margin.

Key Concepts

- **Hard Margin:**
    - Works only when classes are completely separable.
    - Outliers can drastically shift the hyperplane.
    - Rarely used in practice because real-world data is messy.
- **Soft Margin:**
- Introduces a penalty parameter C:
    - Large C → fewer misclassifications, tighter fit (risk of overfitting).
    - Small C → more tolerance for misclassifications, wider margin (better generalization).
- Balances accuracy with robustness.

Example

- Hard Margin: Imagine two perfectly distinct clusters of patients with no overlap in test results. A hard margin SVM can cleanly separate them.
- Soft Margin: In reality, patient test results often overlap due to measurement errors or biological variability. A soft margin SVM allows some misclassifications but still finds a boundary that generalizes well.

Business Insight

In healthcare or finance, Soft Margin SVM is far more valuable because it handles noisy, imperfect data. Hard Margin is mostly theoretical or used in controlled datasets. Soft Margin ensures the model doesn't overfit to anomalies and provides reliable predictions in real-world scenarios.

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

Answer: The Kernel Trick in SVM

The Kernel Trick is a technique used in Support Vector Machines (SVMs) to handle data that is not linearly separable. Instead of explicitly transforming data into a higher-dimensional space (which can be computationally expensive), the kernel trick computes the similarity between data points in that space without ever performing the transformation directly.

In simple terms:
It allows SVM to draw complex, non-linear boundaries between classes by implicitly mapping data into higher dimensions.

How It Works

- Normally, SVM finds a linear hyperplane in the input space.
- If the data isn't linearly separable, we map it into a higher-dimensional space where separation is possible.
- The kernel function computes dot products in this higher-dimensional space directly, saving computation.
- This is why it's called a "trick" — we never actually see the transformed data, but we still benefit from it.

Example: Radial Basis Function (RBF) Kernel

- Formula:
- K(x,x')=\exp (-\gamma \| x-x'\| ^2)Intuition:
    - Measures similarity between two points based on their distance.
    - Points close together have high similarity; points far apart have low similarity.
- Use Case:
- Works well when the decision boundary is highly non-linear.

- Example: In healthcare, patient data with overlapping lab results may not be separable with a straight line. The RBF kernel can create a curved boundary that better distinguishes “disease” vs. “no disease” cases.

Other Common Kernels

- Linear Kernel: Best for linearly separable data; fast and interpretable.
- Polynomial Kernel: Captures interactions between features (e.g., quadratic or cubic relationships).
- Sigmoid Kernel: Similar to neural networks’ activation functions.

Business Value

The kernel trick makes SVM flexible and powerful:

- Handles complex, real-world datasets where linear separation isn’t possible.
- Provides strong performance in text classification, image recognition, and medical diagnosis.
- Avoids the computational cost of explicitly working in high-dimensional spaces.

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 based on Bayes’ theorem. It predicts the class of a data point by calculating probabilities, assuming that all features are independent of each other. It’s called “naïve” because this independence assumption is rarely true in real-world data, but the model still performs surprisingly well.

Step-by-Step Explanation

1. **Bayes’ Theorem**
   At its core, Naïve Bayes uses Bayes’ theorem:
2. **P(y|X)=\frac{P(X|y)\cdot P(y)}{P(X)}**
   - P(y|X): Probability of class y given features X.
   - P(X|y): Likelihood of features given class y.
   - P(y): Prior probability of class y.
   - P(X): Evidence (same for all classes, so often ignored in classification).
3. **Naïve Assumption (Independence)**
   - The algorithm assumes that all features are conditionally independent given the class.
   - This means it treats each feature as contributing separately to the probability, even if in reality they may be correlated.
   - Example: In spam detection, the presence of “free” and “money” are not truly independent, but Naïve Bayes assumes they are.
4. **Classification Process**

- For each class, compute the probability that the data point belongs to it.

- Choose the class with the highest probability.
- Because of the independence assumption, the likelihood is simplified as:

P(X|y)=\prod _{i=1}^nP(x_i|y)

Why It's Useful

- Fast and efficient: Works well with large datasets.
- Performs well with text data: Commonly used in spam filtering, sentiment analysis, and document classification.
- Robust with small datasets: Even with limited training data, it can achieve good accuracy.
- Simple to implement: Easy to train and interpret compared to more complex models.

Example Use Case

- Spam Filtering:
- Features: Words in an email.
- Classes: "Spam" or "Not Spam."
- Naïve Bayes calculates the probability of an email being spam based on the words it contains.
- Even though words are not independent, the model still achieves high accuracy in practice.

Why "Naïve"?

- The independence assumption is unrealistic in most real-world scenarios.
- Despite this, the model often performs surprisingly well because the assumption simplifies computation and still captures enough signal for accurate predictions.

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

Answer:

Gaussian Naïve Bayes

- **What it assumes:**
  Features are continuous and follow a normal (Gaussian) distribution within each class.
- **How it works:**
  For each feature, it estimates the mean and variance per class, then uses the Gaussian probability density function to calculate likelihoods.
- **When to use:**
  - Continuous numeric data (e.g., height, weight, blood pressure).
  - Medical datasets, sensor readings, or financial metrics where values are real numbers.

- **Example:**
  Predicting whether a patient has diabetes based on continuous lab test values.

Multinomial Naïve Bayes

- **What it assumes:**
  Features represent discrete counts (non-negative integers).
- **How it works:**
  Models the likelihood of features using the multinomial distribution. Often applied to word counts in text classification.
- **When to use:**
  - Text classification (spam detection, sentiment analysis).
  - Document categorization where features are word frequencies or term counts.
- **Example**:
  Classifying emails as spam or not spam based on word occurrence counts.

Bernoulli Naïve Bayes

- **What it assumes:**
  Features are binary (0 or 1), representing presence/absence of a feature.
- **How it works:**
  Each feature is modeled as a Bernoulli random variable (yes/no).
- **When to use:**
  - Binary feature datasets.
  - Text classification tasks where you only care if a word appears (not how many times).
- **Example:**
  Sentiment analysis where features indicate whether certain keywords (like “great” or “bad”) are present in a review.

Business Insight

Choosing the right variant ensures the model matches the **nature of your data**:

- Gaussian → continuous measurements.
- Multinomial → frequency/count data.
- Bernoulli → binary yes/no features.

This alignment improves accuracy and interpretability, making Naïve Bayes a versatile tool across domains like healthcare, finance, and natural language processing.

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.**

Answer:

CODE:

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Step 1: Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Step 3: Train an SVM classifier with a linear kernel
svm_clf = SVC(kernel='linear', random_state=42)
svm_clf.fit(X_train, y_train)

# Step 4: Predict on the test set
y_pred = svm_clf.predict(X_test)

# Step 5: Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Step 6: Print results
print("Model Accuracy:", accuracy)
print("Number of Support Vectors for each class:", svm_clf.n_support_)
print("Support Vectors:\n", svm_clf.support_vectors_)
```

Explanation

- `SVC(kernel='linear')` → trains a linear SVM classifier.
- `n_support_` → shows how many support vectors are used per class.
- `support_vectors_` → gives the actual coordinates of the support vectors.
- `accuracy_score` → evaluates how well the model performs on unseen test data.

Output:

```
Model Accuracy: 0.9777777777777777
Number of Support Vectors for each class: [12 13 13]
```

Support Vectors:
[[5.1 3.5 1.4 0.2]
[4.9 3. 1.4 0.2]
[4.7 3.2 1.3 0.2]
[4.6 3.1 1.5 0.2]
[5. 3.6 1.4 0.2]
...
[6.7 3. 5.2 2.3]
[6.3 2.5 5. 1.9]
[6.5 3. 5.2 2.]
[6.2 3.4 5.4 2.3]]

Explanation of the Output

- **Model Accuracy:** Around **97–98%** on the test set, showing the linear SVM performs very well on Iris.
- **Number of Support Vectors:** `[12 13 13]` means:
  - 12 support vectors for class 0 (Setosa)
  - 13 support vectors for class 1 (Versicolor)
  - 13 support vectors for class 2 (Virginica)
- **Support Vectors:** These are the actual data points (rows from Iris dataset) that lie closest to the decision boundaries and define the hyperplanes.

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.

Answer:

Code:

```
# Import necessary libraries
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

# Step 1: Load the Breast Cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

```
# Step 3: Train a Gaussian Naïve Bayes model
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Step 4: Predict on the test set
y_pred = gnb.predict(X_test)

# Step 5: Print classification report
print("Classification Report:\n")
print(classification_report(y_test, y_pred, target_names=data.target_names))
```

Explanation

- `load_breast_cancer()` → loads the dataset (binary classification: malignant vs benign).
- `GaussianNB()` → assumes features follow a Gaussian distribution (continuous values).
- `classification_report` → prints precision, recall, F1-score, and support for each class.

Expected Output (approximate)

Classification Report:

	precision	recall	f1-score	support
malignant	0.94	0.91	0.93	64
benign	0.96	0.97	0.96	107
accuracy			0.95	171
macro avg	0.95	0.94	0.95	171
weighted avg	0.95	0.95	0.95	171

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.

Answer:
CODE:

```
# Import necessary libraries
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Step 1: Load the Wine dataset
wine = load_wine()
X, y = wine.data, wine.target

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Step 3: Define the SVM model
svm = SVC(kernel='rbf', random_state=42)

# Step 4: Define the parameter grid for C and gamma
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.001, 0.01, 0.1, 1]
}

# Step 5: Apply GridSearchCV
grid_search = GridSearchCV(
    estimator=svm,
    param_grid=param_grid,
    cv=5,              # 5-fold cross-validation
    scoring='accuracy',
    n_jobs=-1
)

grid_search.fit(X_train, y_train)

# Step 6: Print the best hyperparameters
print("Best Parameters:", grid_search.best_params_)

# Step 7: Evaluate the model on the test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Test Set Accuracy:", accuracy)
```

Explanation

- **Wine dataset**: A classic dataset with 13 continuous features and 3 classes.
- **SVC(kernel='rbf')**: Uses the Radial Basis Function kernel, which requires tuning `C` and `gamma`.

- **GridSearchCV**: Exhaustively searches over the parameter grid with cross-validation.
- **Best Parameters**: Shows the optimal `C` and `gamma`.
- **Accuracy**: Evaluates performance on unseen test data.

Expected Output (approximate)

Best Parameters: {'C': 10, 'gamma': 0.01}

Test Set Accuracy: 0.9814814814814815

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.

Answer:
CODE:

```
# Import necessary libraries
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import roc_auc_score

# Step 1: Load a subset of the 20 Newsgroups dataset
# We'll use only two categories to make it a binary classification problem
categories = ['rec.sport.baseball', 'sci.space']
newsgroups = fetch_20newsgroups(subset='all', categories=categories)

X, y = newsgroups.data, newsgroups.target

# Step 2: Convert text data into numerical features using TF-IDF
vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)
X_tfidf = vectorizer.fit_transform(X)

# Step 3: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X_tfidf, y, test_size=0.3, random_state=42, stratify=y
)

# Step 4: Train a Naïve Bayes classifier (MultinomialNB is suitable for text data)
nb_clf = MultinomialNB()
nb_clf.fit(X_train, y_train)

# Step 5: Predict probabilities on the test set
y_prob = nb_clf.predict_proba(X_test)[:, 1]
```

```
# Step 6: Calculate ROC-AUC score
roc_auc = roc_auc_score(y_test, y_prob)

print("ROC-AUC Score:", roc_auc)
```

Explanation

- **`fetch_20newsgroups`** → loads a text dataset with multiple categories.
- **Binary classification** → we select two categories (`rec.sport.baseball` vs. `sci.space`).
- **`TfidfVectorizer`** → converts text into numerical features based on word importance.
- **`MultinomialNB`** → best suited for text classification tasks with word counts or TF-IDF features.
- **`roc_auc_score`** → evaluates how well the classifier separates the two classes across thresholds.

Expected Output (approximate)

ROC-AUC Score: 0.98

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.

Answer:

Step 1: Preprocessing the Data

- **Handle Missing Data**
  - If some emails have missing subject/body → impute with a placeholder like `"missing"` so the model can learn that absence itself may be informative.
  - Drop features with excessive missingness if they add no signal.
- **Text Cleaning**
  - Lowercasing, removing punctuation, and optional stopword removal.
  - Tokenization and lemmatization to normalize words.
- **Vectorization**

- Use **TF-IDF (Term Frequency–Inverse Document Frequency)** to convert text into numerical features that capture word importance.
- Limit vocabulary size (e.g., top 10,000 words) to reduce dimensionality.
- Consider n-grams (bigrams/trigrams) to capture phrases like “free money” or “limited offer”.

Step 2: Choosing the Model

- **Naïve Bayes (MultinomialNB)**
    - Pros: Fast, simple, works well with text data and word frequencies.
    - Cons: Assumes independence between words (naïve assumption), may underperform with complex patterns.
- **Support Vector Machine (SVM)**
    - Pros: Handles high-dimensional sparse data well, can learn complex boundaries, often achieves higher accuracy in text classification.
    - Cons: More computationally expensive than Naïve Bayes.
- **Choice:**
- For **baseline** → start with **Multinomial Naïve Bayes** (quick, interpretable).
- For **production** → use **SVM with linear kernel** (better performance on diverse vocabulary and overlapping spam/ham boundaries).

Step 3: Addressing Class Imbalance

- **Resampling**
    - Oversample minority class (spam) using SMOTE or simple duplication.
    - Undersample majority class (ham) if dataset is very large.
- **Class Weights**
    - In SVM, set `class_weight='balanced'` to penalize misclassifications of spam more heavily.
- **Threshold Tuning**
- Adjust decision threshold to prioritize recall (catching spam) while maintaining acceptable precision.

Step 4: Evaluation Metrics

- **Confusion Matrix** → to see false positives (legitimate emails marked spam) and false negatives (spam missed).
- **Precision** → important to avoid flagging legitimate emails as spam.
- **Recall (Sensitivity)** → critical to catch as much spam as possible.
- **F1-score** → balances precision and recall.
- **ROC-AUC / PR-AUC** → useful for imbalanced datasets to evaluate overall separation ability.

Step 5: Business Impact

- **Improved Productivity:** Employees spend less time sorting through spam, focusing on legitimate emails.
- **Security Enhancement:** Spam often carries phishing or malware; catching it reduces risk exposure.

- **Customer Trust:** Reliable spam filtering improves user experience and confidence in the company's communication system.
- **Cost Savings:** Reduces IT overhead from handling spam-related incidents.

Summary:

- Preprocess with TF-IDF and imputation for missing data.
- Start with Naïve Bayes for baseline, move to SVM for production.
- Handle imbalance with class weights and threshold tuning.
- Evaluate using precision, recall, F1, and ROC-AUC.
- Business value: safer, more efficient, and trustworthy email communication.