Technical Methodology

SA eCommerce Customer Analytics Project

1. Project Framework and Methodology Overview

1.1 Advanced CRISP-DM Implementation

The project follows an enhanced Cross-Industry Standard Process for Data Mining (CRISP-DM) framework, incorporating modern data science practices and business intelligence methodologies:

Enhanced CRISP-DM Phases:

1. **Business Understanding** - Strategic alignment and stakeholder requirements
2. **Data Understanding** - Comprehensive data exploration and quality assessment
3. **Data Preparation** - Advanced preprocessing and feature engineering
4. **Modelling** - Machine learning and statistical analysis implementation
5. **Evaluation** - Multi-dimensional validation and business impact assessment
6. **Deployment** - Production-ready systems and monitoring frameworks

1.2 Integrated Research Methodology

Multi-Modal Analytical Approach:

- **Descriptive Analytics:** Understanding current state through comprehensive EDA
- **Diagnostic Analytics:** Root cause analysis of business challenges and opportunities
- **Predictive Analytics:** Machine learning models for churn prediction and CLV forecasting
- **Prescriptive Analytics:** Optimisation recommendations and strategic guidance
- **Cognitive Analytics:** Advanced pattern recognition and behavioural insights

1.3 Theoretical Framework Foundation

Customer Analytics Theory:

- **RFM Analysis Framework:** Recency, Frequency, Monetary value segmentation
- **Customer Lifetime Value Theory:** Predictive value modelling and optimisation
- **Behavioural Economics Principles:** Decision-making patterns and intervention strategies

- **Geographic Market Theory:** Spatial analysis and market penetration optimisation

2. Enhanced Data Collection and Infrastructure

2.1 Advanced Data Architecture

Cloud-Native Infrastructure:

- **Platform:** Google BigQuery with advanced data warehouse optimisation
- **Data Lake Architecture:** Structured and semi-structured data integration
- **Real-Time Processing:** Apache Kafka for streaming data ingestion
- **Data Governance:** Comprehensive data lineage and quality monitoring

2.2 Comprehensive Dataset Integration

```python
# Advanced Data Integration Framework
class DataIntegrator:
    def __init__(self, project_id, dataset_id):
        self.client = bigquery.Client(project=project_id)
        self.dataset_ref = self.client.dataset(dataset_id)
        self.data_quality_metrics = {}

    def integrate_sources(self):
        """Comprehensive multi-source data integration"""
        sources = {
            'customers': self.load_customer_data(),
            'orders': self.load_order_history(),
            'reviews': self.load_customer_reviews(),
            'nps': self.load_nps_surveys(),
            'churn': self.load_churn_data(),
            'activity': self.load_website_activity()
        }
        return self.validate_and_merge(sources)
```

2.3 Data Quality Assurance Framework

Multi-Dimensional Quality Assessment:

- **Completeness:** Missing value analysis with imputation strategies
- **Accuracy:** Cross-validation against external sources and business rules
- **Consistency:** Format standardisation and categorical value normalisation
- **Timeliness:** Data freshness monitoring and update frequency validation
- **Uniqueness:** Duplicate detection and resolution protocols
- **Validity:** Range checks and constraint validation

3. Advanced Data Preprocessing and Feature Engineering

3.1 Sophisticated Feature Engineering Pipeline

```
class AdvancedFeatureEngineer:
    def __init__(self, data):
        self.data = data
        self.feature_store = {}

    def create_customer_features(self):
        """Advanced customer-level feature engineering"""
        # Temporal features
        self.data['customer_age_days'] = (pd.Timestamp.now() -

pd.to_datetime(self.data['RegisteredDate'])).dt.days
        self.data['days_since_last_order'] = (pd.Timestamp.now() -

pd.to_datetime(self.data['LastOrderDate'])).dt.days

        # Behavioural features
        self.data['order_frequency'] = (self.data['NumberOfOrders'] /
                                       (self.data['customer_age_days'] /
30.44))
        self.data['avg_order_value'] = self.data['TotalSpend'] /
self.data['NumberOfOrders']

        # Risk indicators
        self.data['return_rate'] = (self.data['NumberOfReturnedOrders'] /
                                   self.data['NumberOfOrders'])
        self.data['cancellation_rate'] = (self.data['NumberOfCanceledOrders']
/
                                         self.data['NumberOfOrders'])

        # Engagement metrics
        self.data['order_consistency'] = self.calculate_order_consistency()
        self.data['category_diversity'] = self.calculate_category_diversity()

        return self.data

    def create_geographic_features(self):
        """Geographic and market penetration features"""
        # Market penetration metrics
        province_stats = self.data.groupby('Province').agg({
            'TotalSpend': ['sum', 'mean', 'std'],
            'CustomerID': 'count'
        }).round(2)

        # Urban/rural classification
        urban_cities = ['Johannesburg', 'Cape Town', 'Durban', 'Pretoria']
        self.data['is_urban'] = self.data['City'].isin(urban_cities)
```

```python
        return province_stats

    def create_product_features(self):
        """Product performance and preference features"""
        # Product affinity scores
        self.data['premium_preference'] = self.calculate_premium_affinity()
        self.data['brand_loyalty'] = self.calculate_brand_consistency()

        # Purchase timing patterns
        self.data['seasonal_shopper'] = self.identify_seasonal_patterns()
        self.data['promotion_sensitivity'] =
self.calculate_promotion_response()

        return self.data
```

3.2 Advanced Statistical Preprocessing

Outlier Detection and Treatment:

- **Isolation Forest:** For multivariate outlier detection
- **Z-Score Analysis:** For univariate outlier identification
- **Interquartile Range (IQR):** For robust outlier boundaries
- **Business Rule Validation:** Domain-specific outlier assessment

Missing Value Treatment:

- **Multiple Imputation:** Using chained equations for robust imputation
- **K-Nearest Neighbours:** For similarity-based imputation
- **Forward/Backward Fill:** For temporal data sequences
- **Domain-Specific Rules:** Business logic-driven imputation strategies

4. Enhanced Analytical Techniques and Modelling

4.1 Advanced Customer Segmentation Methodology

Multi-Dimensional Segmentation Framework:

```
class AdvancedCustomerSegmentation:
    def __init__(self, customer_data):
        self.data = customer_data
        self.models = {}

    def calculate_enhanced_clv(self):
        """Advanced CLV calculation with predictive components"""
        # Historical CLV
        historical_clv = self.calculate_historical_clv()

        # Predictive CLV using XGBoost
        future_clv = self.predict_future_clv()

        # Combined weighted CLV
        combined_clv = (0.6 * historical_clv) + (0.4 * future_clv)

        return combined_clv

    def calculate_churn_probability(self):
        """Multi-model churn probability ensemble"""
        # Individual model predictions
        rf_prob = self.random_forest_churn_model()
        gb_prob = self.gradient_boosting_churn_model()
        lr_prob = self.logistic_regression_churn_model()

        # Ensemble prediction with weighted averaging
        ensemble_prob = (0.4 * rf_prob) + (0.35 * gb_prob) + (0.25 * lr_prob)

        return ensemble_prob

    def create_dynamic_segments(self):
        """Dynamic segmentation with time-based evolution"""
        # CLV and churn thresholds with confidence intervals
        clv_threshold = self.calculate_adaptive_clv_threshold()
        churn_threshold = self.calculate_adaptive_churn_threshold()

        # Four-quadrant segmentation
        conditions = [
            (self.data['CLV'] > clv_threshold) & (self.data['ChurnProb'] <
churn_threshold),
            (self.data['CLV'] <= clv_threshold) & (self.data['ChurnProb'] <
churn_threshold),
            (self.data['CLV'] > clv_threshold) & (self.data['ChurnProb'] >=
```

```
churn_threshold),
            (self.data['CLV'] <= clv_threshold) & (self.data['ChurnProb'] >=
churn_threshold)
        ]

        segment_labels = ['Champions', 'Loyal Customers', 'At Risk', 'Lost
Causes']

        return np.select(conditions, segment_labels, default='Unclassified')
```

4.2 Advanced Churn Prediction with Model Explainability

```
class ExplainableChurnModel:
    def __init__(self):
        self.model = None
        self.explainer = None
        self.feature_importance = None

    def train_ensemble_model(self, X_train, y_train):
        """Advanced ensemble model with hyperparameter optimisation"""
        # Handle class imbalance with SMOTE
        smote = SMOTE(random_state=42, k_neighbors=3)
        X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

        # Hyperparameter optimisation with Optuna
        study = optuna.create_study(direction='maximise')
        study.optimise(self.objective, n_trials=100)

        # Best model training
        best_params = study.best_params
        self.model = RandomForestClassifier(**best_params, random_state=42)
        self.model.fit(X_resampled, y_resampled)

        # Model explainability setup
        self.setup_explainability(X_train)

        return self.model

    def setup_explainability(self, X):
        """SHAP-based model explainability"""
        self.explainer = shap.TreeExplainer(self.model)
        self.shap_values = self.explainer.shap_values(X)

        # Feature importance analysis
        self.feature_importance = pd.DataFrame({
            'feature': X.columns,
            'importance': self.model.feature_importances_
        }).sort_values('importance', ascending=False)
```

```
        return self.explainer

    def explain_prediction(self, customer_data):
        """Individual customer churn explanation"""
        shap_values = self.explainer.shap_values(customer_data)

        explanation = {
            'churn_probability':
self.model.predict_proba(customer_data)[0][1],
            'key_factors': self.get_top_factors(shap_values[0]),
            'recommendation': self.generate_recommendation(shap_values[0])
        }

        return explanation
```

4.3 Advanced Geographic and Market Analysis

Spatial Analytics Framework:

```
class GeospatialAnalytics:
    def __init__(self, geographic_data):
        self.data = geographic_data
        self.market_potential = {}

    def calculate_market_penetration(self):
        """Advanced market penetration analysis"""
        # Population-weighted penetration
        penetration_metrics = self.data.groupby('Province').apply(
            lambda x: {
                'revenue_per_capita': x['TotalRevenue'].sum() /
x['Population'].iloc[0],
                'customer_density': len(x) / x['Area_km2'].iloc[0],
                'market_share': x['TotalRevenue'].sum() /
self.data['TotalRevenue'].sum(),
                'growth_potential': self.calculate_growth_potential(x)
            }
        )

        return penetration_metrics

    def identify_expansion_opportunities(self):
        """AI-driven market opportunity identification"""
        # Clustering for similar market characteristics
        market_features = ['population_density', 'income_level',
'urbanisation',
                          'internet_penetration', 'competition_index']
```

```
        kmeans = KMeans(n_clusters=5, random_state=42)
        market_clusters = kmeans.fit_predict(self.data[market_features])

        # Opportunity scoring
        opportunity_scores =
self.calculate_opportunity_scores(market_clusters)

        return opportunity_scores
```

5. Advanced Validation and Model Performance

5.1 Comprehensive Model Validation Framework

Multi-Level Validation Strategy:

```
class ModelValidation:
    def __init__(self, models):
        self.models = models
        self.validation_results = {}

    def statistical_validation(self, X, y):
        """Statistical significance and robustness testing"""
        # Cross-validation with multiple metrics
        cv_scores = cross_validate(
            self.models['churn'], X, y,
            cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
            scoring=['accuracy', 'precision', 'recall', 'f1', 'roc_auc'],
            return_train_score=True
        )

        # Bootstrap confidence intervals
        bootstrap_results = self.bootstrap_validation(X, y,
n_iterations=1000)

        return cv_scores, bootstrap_results

    def business_validation(self, predictions, actual_outcomes):
        """Business impact validation"""
        # Calculate business metrics
        retention_lift = self.calculate_retention_lift(predictions,
actual_outcomes)
        revenue_impact = self.calculate_revenue_impact(predictions,
actual_outcomes)
        cost_effectiveness = self.calculate_intervention_costs(predictions)

        business_metrics = {
            'retention_lift': retention_lift,
            'revenue_impact': revenue_impact,
```

```
            'cost_effectiveness': cost_effectiveness,
            'roi': revenue_impact / cost_effectiveness
        }

        return business_metrics

    def temporal_validation(self, time_series_data):
        """Time-based validation for temporal stability"""
        # Walk-forward validation
        validation_windows = self.create_validation_windows(time_series_data)

        temporal_performance = []
        for train_window, test_window in validation_windows:
            model_performance =
self.evaluate_window_performance(train_window, test_window)
            temporal_performance.append(model_performance)

        return temporal_performance
```

5.2 A/B Testing and Experimental Framework

```
class ExperimentalFramework:
    def __init__(self):
        self.experiments = {}
        self.results = {}

    def design_retention_experiment(self, customer_segments):
        """A/B test design for retention strategies"""
        experiment_design = {
            'name': 'retention_strategy_test',
            'hypothesis': 'Personalised retention offers increase customer
retention',
            'treatment_groups': {
                'control': 'standard_retention_email',
                'treatment_a': 'personalised_discount_offer',
                'treatment_b': 'exclusive_product_access',
                'treatment_c': 'loyalty_points_bonus'
            },
            'success_metrics': ['retention_rate', 'clv_change',
'engagement_score'],
            'sample_size': self.calculate_sample_size(effect_size=0.1,
power=0.8),
            'duration': '6_weeks'
        }

        return experiment_design

    def analyse_experiment_results(self, experiment_data):
```

```
        """Statistical analysis of A/B test results"""
        # Bayesian A/B testing
        bayesian_results = self.bayesian_ab_analysis(experiment_data)

        # Frequentist analysis with multiple comparisons correction
        frequentist_results = self.frequentist_analysis(experiment_data)

        # Business impact assessment
        business_impact = self.calculate_business_impact(experiment_data)

        return {
            'bayesian': bayesian_results,
            'frequentist': frequentist_results,
            'business_impact': business_impact,
            'recommendation': self.generate_experiment_recommendation()
        }
```

6. Advanced Visualisation and Dashboard Architecture

6.1 Interactive Dashboard Development

Tableau Advanced Implementation:

```
-- Advanced calculated fields for Tableau
-- Dynamic CLV calculation with parameters
IF [CLV Calculation Method] = "Historical" THEN
    ([Total Spend] / [Number of Orders]) * [Order Frequency] * [Customer
Lifespan]
ELSEIF [CLV Calculation Method] = "Predictive" THEN
    [Predicted Future Value] + [Historical CLV] * 0.6
ELSE
    [Historical CLV]
END

-- Dynamic segmentation with adjustable thresholds
IF [CLV] > [CLV Threshold Parameter] AND [Churn Probability] < [Churn
Threshold Parameter] THEN
    "Champions"
ELSEIF [CLV] <= [CLV Threshold Parameter] AND [Churn Probability] < [Churn
Threshold Parameter] THEN
    "Loyal Customers"
ELSEIF [CLV] > [CLV Threshold Parameter] AND [Churn Probability] >= [Churn
Threshold Parameter] THEN
    "At Risk High Value"
ELSE
    "At Risk Low Value"
END
```

6.2 Real-Time Analytics Implementation

```
class RealTimeAnalytics:
    def __init__(self, streaming_config):
        self.kafka_consumer = self.setup_kafka_consumer(streaming_config)
        self.redis_client = redis.Redis(host='localhost', port=6379, db=0)

    def process_real_time_events(self):
        """Real-time customer behaviour processing"""
        for message in self.kafka_consumer:
            event_data = json.loads(message.value)

            # Real-time churn risk scoring
            churn_score = self.calculate_real_time_churn_risk(event_data)

            # Update customer risk profile
            self.update_customer_profile(event_data['customer_id'],
churn_score)

            # Trigger interventions if needed
            if churn_score > 0.7:

self.trigger_retention_intervention(event_data['customer_id'])

    def update_dashboard_metrics(self, new_data):
        """Real-time dashboard updates"""
        # Update Redis cache for dashboard
        self.redis_client.setex(
            f"customer_metrics_{new_data['customer_id']}",
            3600,
            json.dumps(new_data)
        )

        # Trigger dashboard refresh
        self.notify_dashboard_update()
```

7. Enhanced Ethical Considerations and Compliance

7.1 Advanced Privacy and Ethics Framework

POPIA Compliance Enhancement:

```
class PrivacyFramework:
    def __init__(self):
        self.consent_manager = ConsentManager()
        self.data_minimiser = DataMinimiser()
        self.anonymiser = DataAnonymiser()

    def ensure_data_compliance(self, customer_data):
        """Comprehensive data privacy compliance"""
        # Consent verification
        consented_data =
self.consent_manager.filter_consented_customers(customer_data)

        # Data minimisation
        minimal_data =
self.data_minimiser.extract_necessary_features(consented_data)

        # Anonymisation for analytics
        anonymised_data =
self.anonymiser.anonymise_sensitive_fields(minimal_data)

        return anonymised_data

    def bias_detection_and_mitigation(self, model_predictions):
        """Algorithmic bias detection and mitigation"""
        # Demographic parity assessment
        parity_metrics = self.calculate_demographic_parity(model_predictions)

        # Equalised odds assessment
        odds_metrics = self.calculate_equalised_odds(model_predictions)

        # Bias mitigation strategies
        if parity_metrics['bias_detected']:
            mitigated_predictions =
self.apply_bias_mitigation(model_predictions)
            return mitigated_predictions

        return model_predictions
```

7.2 Model Interpretability and Transparency

```
class ModelTransparency:
    def __init__(self, models):
        self.models = models
        self.explanation_cache = {}

    def generate_model_cards(self):
        """Comprehensive model documentation"""
        for model_name, model in self.models.items():
            model_card = {
                'model_overview': self.generate_model_overview(model),
                'training_data': self.document_training_data(model),
                'performance_metrics':
self.extract_performance_metrics(model),
                'bias_assessment': self.assess_model_bias(model),
                'limitations': self.document_limitations(model),
                'intended_use': self.define_intended_use(model)
            }

            self.save_model_card(model_name, model_card)

    def explain_business_decisions(self, customer_id, decision):
        """Business-friendly explanations for model decisions"""
        technical_explanation = self.get_technical_explanation(customer_id)

        business_explanation = {
            'decision': decision,
            'confidence': technical_explanation['confidence'],
            'key_factors': self.translate_factors_to_business(
                technical_explanation['factors']
            ),
            'recommended_actions': self.generate_action_recommendations(
                customer_id, decision
            )
        }

        return business_explanation
```

8. Advanced Deployment and Monitoring

8.1 Production-Ready Deployment Architecture

```
class ProductionDeployment:
    def __init__(self, model_registry):
        self.model_registry = model_registry
        self.monitoring = ModelMonitoring()
        self.alerting = AlertingSystem()

    def deploy_model_pipeline(self, model_version):
        """Production model deployment with monitoring"""
        # Model validation in staging
        staging_results = self.validate_in_staging(model_version)

        if staging_results['validation_passed']:
            # Blue-green deployment
            deployment_result = self.blue_green_deploy(model_version)

            # Setup monitoring
            self.monitoring.setup_model_monitoring(model_version)

            # Configure alerts
            self.alerting.configure_performance_alerts(model_version)

            return deployment_result
        else:
            raise DeploymentValidationError("Model failed staging
validation")

    def monitor_model_drift(self):
        """Continuous model drift monitoring"""
        drift_metrics = {
            'data_drift': self.monitoring.calculate_data_drift(),
            'concept_drift': self.monitoring.calculate_concept_drift(),
            'performance_drift':
self.monitoring.calculate_performance_drift()
        }

        # Trigger retraining if drift detected
        if any(metric > threshold for metric, threshold in
drift_metrics.items()):
            self.trigger_model_retraining()

        return drift_metrics
```

8.2 Continuous Learning and Adaptation

```
class ContinuousLearning:
    def __init__(self):
        self.learning_pipeline = LearningPipeline()
```

```
        self.feedback_loop = FeedbackLoop()

    def implement_online_learning(self, new_data_stream):
        """Online learning for model adaptation"""
        # Incremental learning with new data
        for batch in new_data_stream:
            # Validate data quality
            if self.validate_batch_quality(batch):
                # Update model incrementally
                self.learning_pipeline.partial_fit(batch)

                # Evaluate updated performance
                performance = self.evaluate_incremental_performance()

                # Deploy if improvement detected
                if performance['improvement'] > 0.02:
                    self.deploy_updated_model()

    def feedback_integration(self, business_outcomes):
        """Integrate business feedback into model improvement"""
        # Collect intervention outcomes
        intervention_results =
self.feedback_loop.collect_outcomes(business_outcomes)

        # Update reward function
        updated_rewards = self.update_reward_function(intervention_results)

        # Retrain with reinforcement learning
        self.retrain_with_reinforcement(updated_rewards)
```

9. Future Enhancements and Innovation

9.1 Advanced AI and Machine Learning

Next-Generation Capabilities:

- **Deep Learning Integration:** Neural networks for complex pattern recognition
- **Natural Language Processing:** Review sentiment analysis and customer feedback processing
- **Computer Vision:** Product image analysis for quality assessment
- **Reinforcement Learning:** Dynamic pricing and intervention optimization

9.2 Emerging Technology Integration

Innovation Roadmap:

- **IoT Integration:** Smart device data for enhanced customer insights

- **Blockchain:** Supply chain transparency and customer trust
- **Edge Computing:** Real-time decision making at point of interaction
- **Quantum Computing:** Advanced optimisation for complex business problems

10. Reproducibility and Documentation Standards

10.1 Comprehensive Environment Management

```
# Docker containerisation for reproducibility
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

# Environment variables
ENV PYTHONPATH=/app
ENV GOOGLE_APPLICATION_CREDENTIALS=/app/credentials/bigquery_key.json

# Command to run application
CMD ["python", "src/main.py"]
```

10.2 Automated Testing and Validation

```
class AutomatedTesting:
    def __init__(self):
        self.test_suite = TestSuite()

    def run_comprehensive_tests(self):
        """Automated testing pipeline"""
        test_results = {
            'data_quality_tests': self.test_suite.run_data_quality_tests(),
            'model_performance_tests': self.test_suite.run_model_tests(),
            'business_logic_tests': self.test_suite.run_business_tests(),
            'integration_tests': self.test_suite.run_integration_tests()
        }

        return test_results
```

Report done by: Aviwe Dlepu

Technical Methodology Version: 2.0
Last Updated: August 2025
Framework Compliance: CRISP-DM 2.0, MLOps Standards
Review Cycle: Quarterly methodology assessment and enhancement